**6주차 결과보고서**

**전공: 경영학과 학년: 3학년 학번: 20190963 이름: 한다현**

**1.**

**6주차에는 가산기와 감산기, 부호 변환기의 개념을 이해하고 각각을 verilog를 통해 구현하는 실험을 진행하였다. 가산기 중에는 두 개의 1비트 수를 입력함으로써 sum과 carry를 얻을 수 있는 half adder와 두 개의 1비트 수와 carry를 더할 수 있는 full adder를 구현하였다. 감산기는 두개의 1비트 수의 뺄셈을 계산하여 difference와 borrow를 출력하는 half subtractor와 두 개의 1비트 수와 borrow의 뺄셈을 계산+하는 full subtractor를 구현하였다. 부호 변환기는 8421코드를 2421코드로 변환하는 변환기를 구현하였다. 이때 변환기는 최대한 nand 게이트와 nor 게이트로 구성되도록 설계하는 게 목적이었다. 각각을 verilog로 설계한 후 이를 FPGA와 연결하여 회로에서 정상적으로 동작하는지 확인하였다.**

**2.**

**Half adder는 verilog로 다음과 같이 구현할 수 있다.**

**module inv\_ha(**

**input a, b,**

**output s, c**

**);**

**assign s = a^b;**

**assign c = a&b;**

**endmodule**

**a, b는 half adder에 입력되는 두 개의 1비트 수이고 s는 sum, c는 carry를 의미한다. Sum은 a와 b를 XOR 게이트에 입력한 결과이고 carry는 a와 b를 AND 게이트에 입력한 결과이다. 이후 a를 aa, b를 bb, c를 c, s를 s에 대응시키고 a와 b를 일정 시간마다 변화하도록 설계하면 half adder의 simulation을 구현할 수 있다. 시뮬레이션을 구현하는 코드와 그 결과는 다음과 같다. 사진과 같이 s는 a와 b 중 하나만 1인 경우에는 1이 되고 c는 a와 b가 모두 1인 경우에만 1이 된다는 것을 확인할 수 있다.**

**module inv\_ha\_tb;**

**reg aa, bb;**

**wire s, c;**

**inv\_ha u\_inv(**

**.a(aa),**

**.b(bb),**

**.c(c),**

**.s(s));**

**initial begin**

**aa = 1'b0;**

**bb = 1'b1;**

**end**

**always @(aa or bb)**

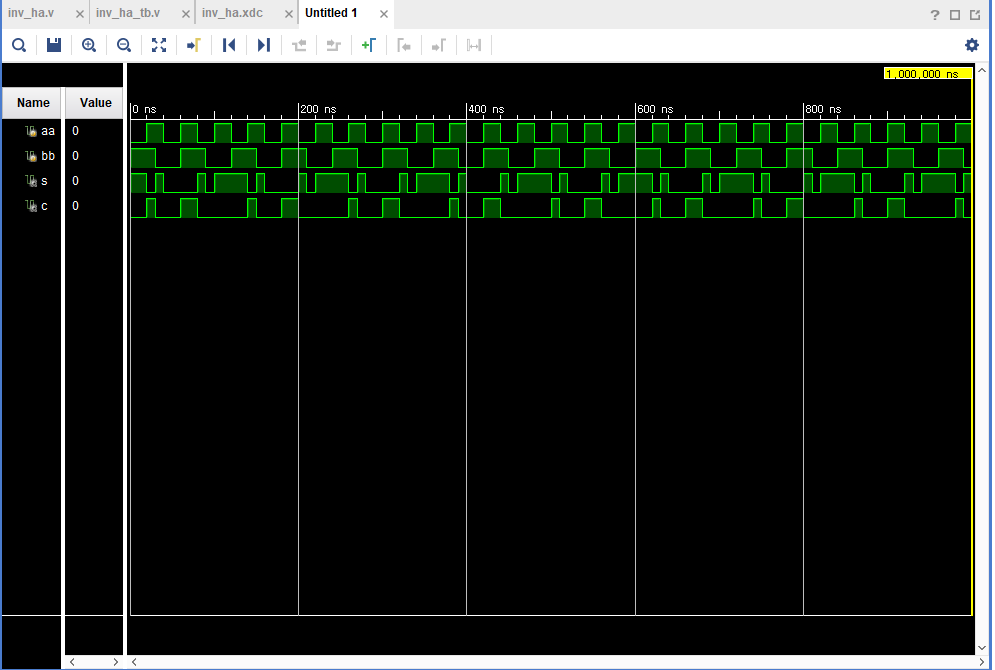
**begin**

**aa <= #20 ~ aa;**

**bb <= #30 ~ bb;**

**end**

**endmodule**

**Half adder의 진리표는 다음과 같다.**

|  |  |  |  |
| --- | --- | --- | --- |
| **In a** | **In b** | **Sum** | **carry** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** |

**Full adder는 verilog로 다음과 같이 구현할 수 있다.**

**module inv\_fa(**

**input a,b,c,**

**output s, co**

**);**

**assign s = (a^b)^c;**

**assign co = (a&b)|((a^b)&c);**

**endmodule**

**입력 a와 b는 두 개의 1비트 수이고 c는 이전 단계의 carry이며 s는 sum, co는 carry를 의미한다. Sum은 a, b, c가 입력인 XOR 게이트의 결과이고 carry는 a와 b가 입력인 AND 게이트의 결과이거나 a, b의 XOR 게이트 결과값과 c를 입력으로 하는 AND 게이트의 결과이다. 이후 a를 aa, b를 bb, c를 cc, s를 s, co를 co에 대응시키고 a와 b, c를 일정 시간마다 변화하도록 설계하면 full adder의 simulation을 구현할 수 있다. 시뮬레이션을 구현하는 코드와 그 결과는 다음과 같다. 사진과 같이 s는 a, b, c 중 1의 개수가 홀수인 경우에 1이 되고 c는 a와 b가 모두 1이거나 a, b 중 하나만 1이고 c가 1인 경우에 1이 된다는 것을 확인할 수 있다.**

**module inv\_fa\_tb;**

**reg aa, bb, cc;**

**wire s, co;**

**inv\_fa u\_inv(**

**.a(aa),**

**.b(bb),**

**.c(cc),**

**.co(co),**

**.s(s));**

**initial begin**

**aa = 1'b0;**

**bb = 1'b1;**

**cc = 1'b0;**

**end**

**always @(aa or bb or cc)**

**begin**

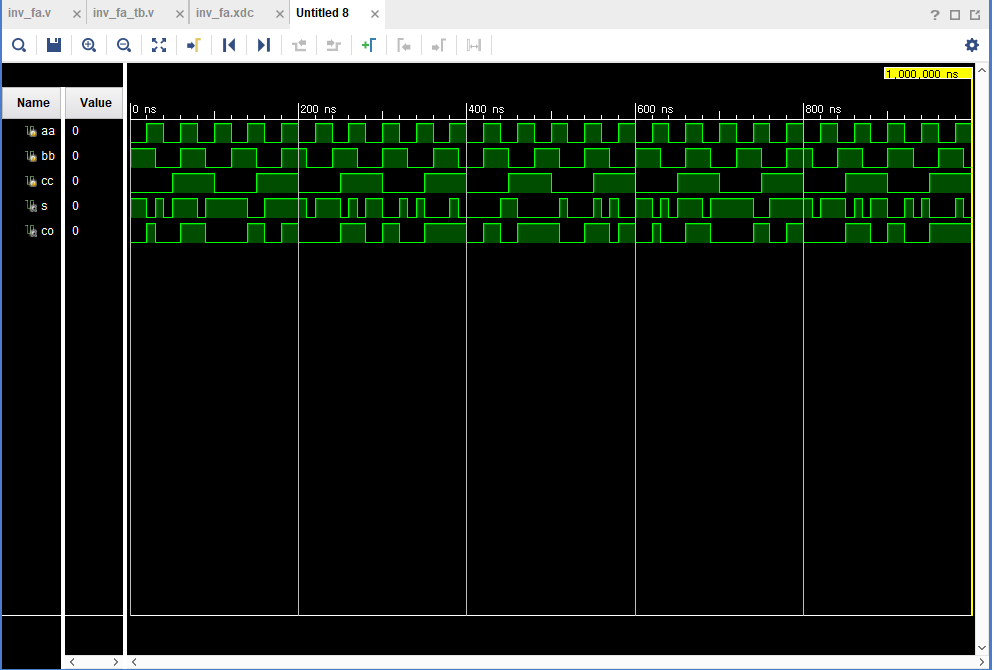
**aa <= #20 ~ aa;**

**bb <= #30 ~ bb;**

**cc <= #50 ~ cc;**

**end**

**endmodule**

****

**Full adder의 진리표는 다음과 같다.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **In a** | **In b** | **In c** | **Sum** | **Carry** |
| **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** |

**3.**

**Half subtractor는 verilog로 다음과 같이 구현할 수 있다.**

**module inv\_hs(**

**input a,b,**

**output d,bo**

**);**

**assign d = a^b;**

**assign bo = ~a&b;**

**endmodule**

**입력 a와 b는 1비트 수이고 d는 difference, bo는 borrow를 의미한다. Difference는 a와 b가 입력인 XOR 게이트의 결과이고 bo는 a의 부정과 b의 AND 게이트의 결과이다. 이후 a를 aa, b를 bb, d를 d, bo를 bo에 대응시키고 a와 b를 일정 시간마다 변화하도록 설계하면 half subtractor의 simulation을 구현할 수 있다. 시뮬레이션을 구현하는 코드와 그 결과는 다음과 같다. 사진과 같이 d는 a, b 중 하나만 1인 경우에 1이 되고 bo는 a가 0이고 b가 1인 경우에 1이 된다는 것을 확인할 수 있다.**

**module inv\_hs\_tb;**

**reg aa, bb;**

**wire d, bo;**

**inv\_hs u\_inv(**

**.a(aa),**

**.b(bb),**

**.d(d),**

**.bo(bo))**

**initial begin**

**aa = 1'b0;**

**bb = 1'b1;**

**end**

**always @(aa or bb)**

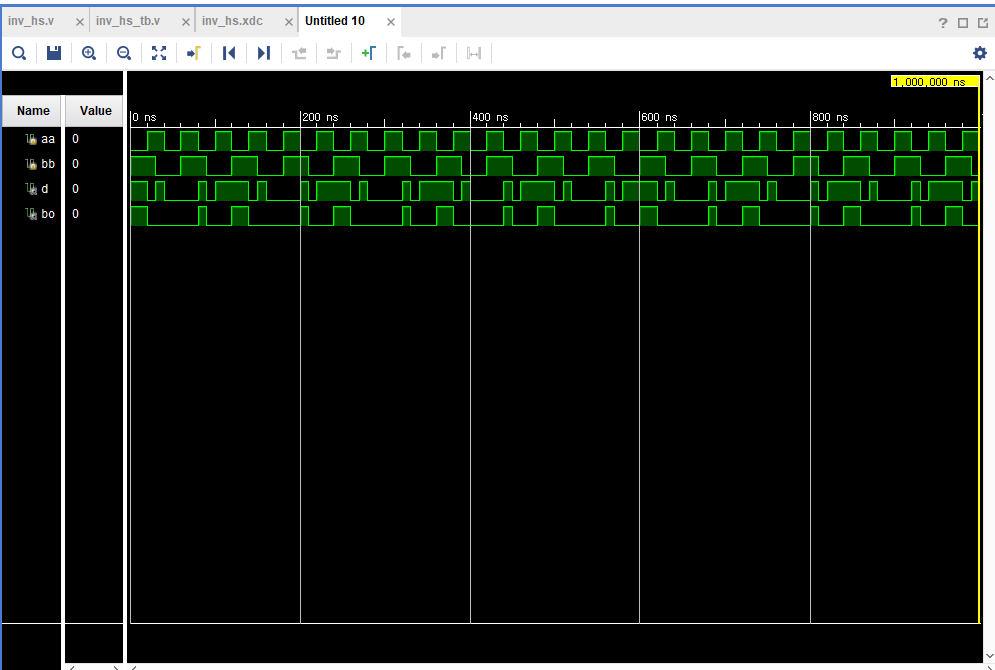
**begin**

**aa <= #20 ~ aa;**

**bb <= #30 ~ bb;**

**end**

**endmodule**

****

**Half subtractor의 진리표는 다음과 같다.**

|  |  |  |  |
| --- | --- | --- | --- |
| **In a** | **In b** | **Difference** | **borrow** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **1** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **0** |

**Full subtractor는 verilog로 다음과 같이 구현할 수 있다.**

**module inv\_fs(**

**input a,b,bi,**

**output d,bo**

**);**

**assign d = (a^b)^bi;**

**assign bo = (~a&b)|(~(a^b)&bi);**

**endmodule**

**입력 a와 b는 1비트 수이고 bi는 이전 단계의 borrow, d는 difference, bo는 borrow를 의미한다. Difference는 a, b, c가 입력인 XOR 게이트의 결과이고 borrow는 a의 부정과 b가 입력인 AND 게이트의 결과이거나 a, b의 XOR 게이트 결과값의 부정과 bi를 입력으로 하는 AND 게이트의 결과이다. 이후 a를 aa, b를 bb, bi를 bi, d를 d, bo를 bo에 대응시키고 a와 b, bi를 일정 시간마다 변화하도록 설계하면 full subtractor의 simulation을 구현할 수 있다. 시뮬레이션을 구현하는 코드와 그 결과는 다음과 같다. 사진과 같이 d는 a, b, bi 중 1의 개수가 홀수인 경우에 1이 되고 bo는 a가 0이고 b가 1이거나 a, b가 모두 0 또는 1이면서 bi가 1인 경우에 1이 된다는 것을 확인할 수 있다.**

**module inv\_fs\_tb;**

**reg aa, bb, bi;**

**wire d, bo;**

**inv\_fs u\_inv(**

**.a(aa),**

**.b(bb),**

**.bi(bi),**

**.d(d),**

**.bo(bo));**

**initial begin**

**aa = 1'b0;**

**bb = 1'b1;**

**bi = 1'b0;**

**end**

**always @(aa or bb or bi)**

**begin**

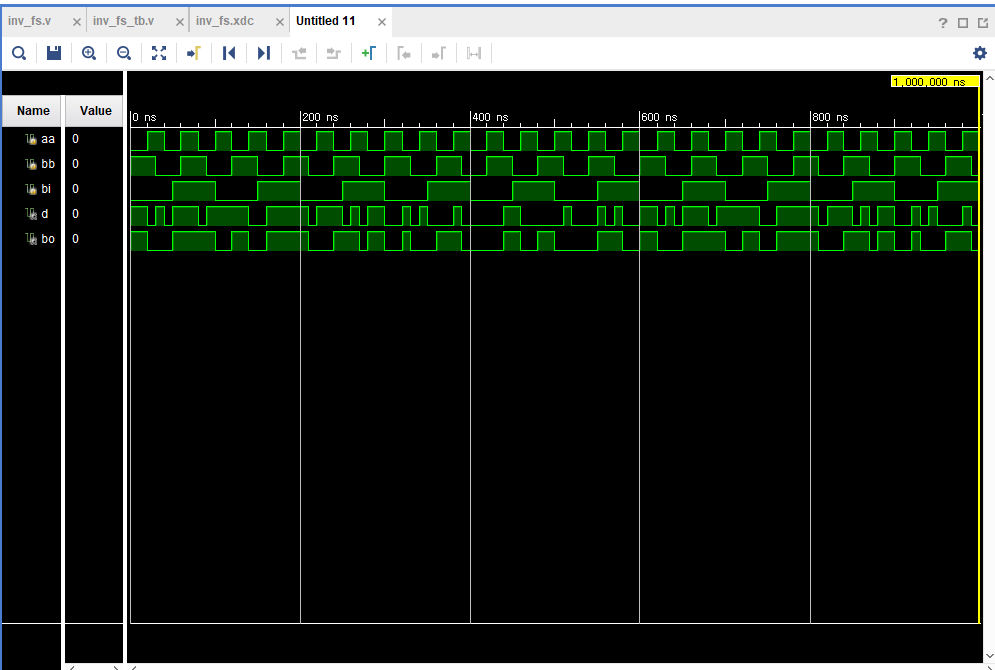
**aa <= #20 ~ aa;**

**bb <= #30 ~ bb;**

**bi <= #50 ~ bi;**

**end**

**endmodule**

****

**Full subtractor의 진리표는 다음과 같다.**

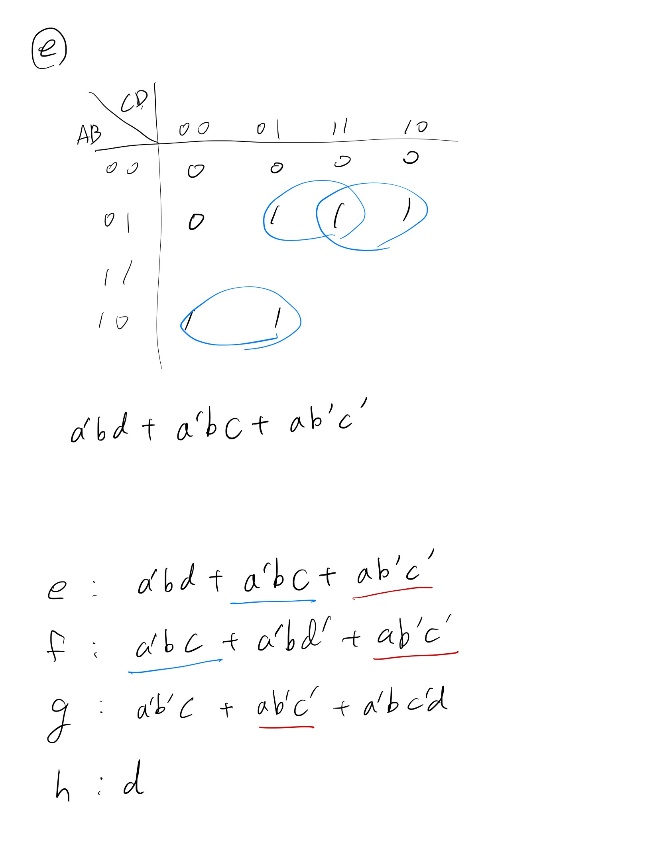
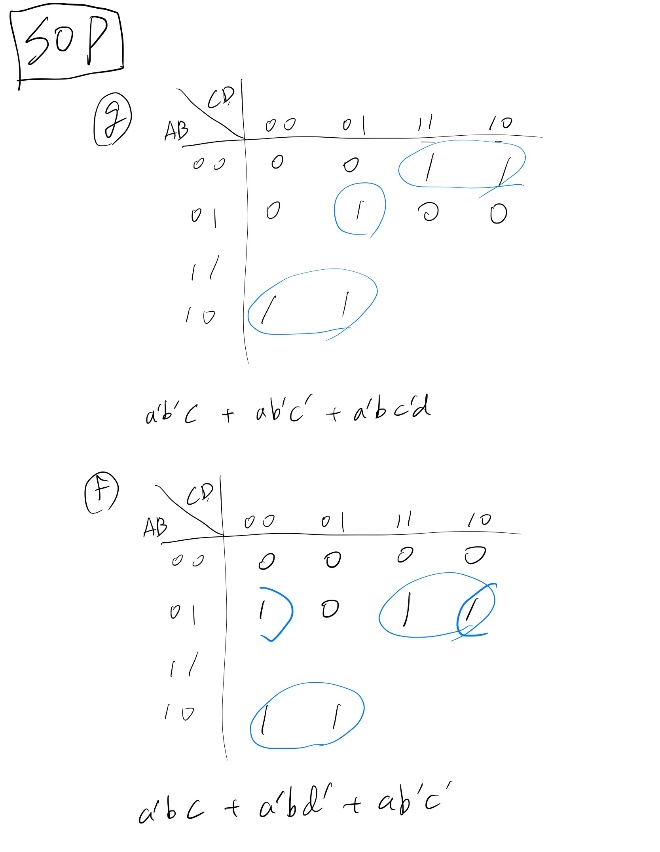
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **In a** | **In b** | **In bi** | **Difference** | **Borrow** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **1** |
| **0** | **1** | **0** | **1** | **1** |
| **0** | **1** | **1** | **0** | **1** |
| **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **0** | **0** |
| **1** | **1** | **0** | **0** | **0** |
| **1** | **1** | **1** | **1** | **1** |

**4.**

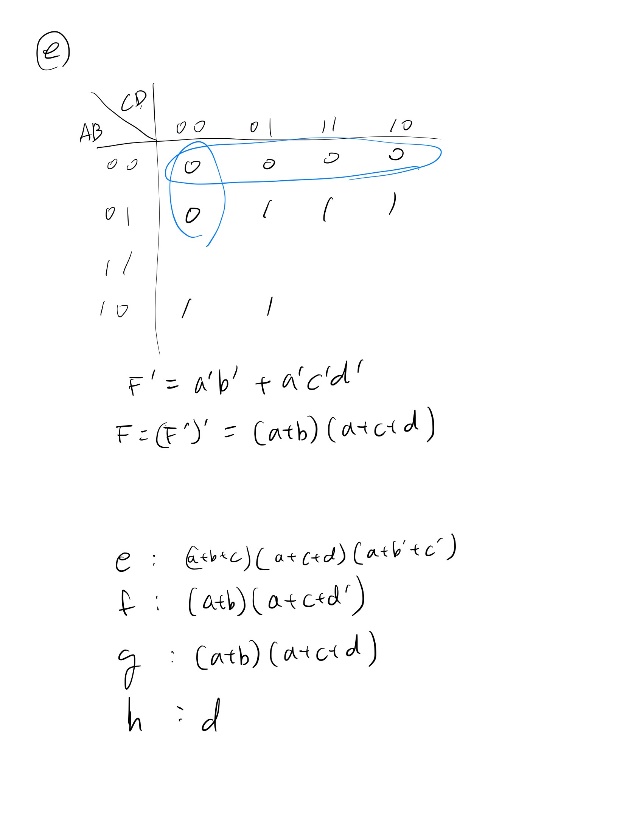
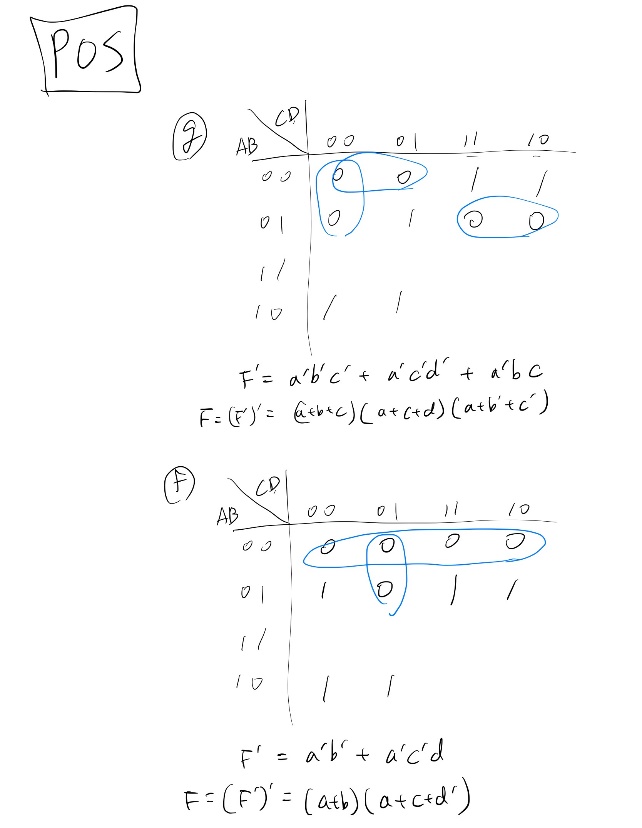
**8421코드(a,b,c,d)를 2421코드(e,f,g,h)로 변환하는 변환기의 진리표는 다음과 같다.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** |
| **0** | **0** | **1** | **0** | **0** | **0** | **1** | **0** |
| **0** | **0** | **1** | **1** | **0** | **0** | **1** | **1** |
| **0** | **1** | **0** | **0** | **0** | **1** | **0** | **0** |
| **0** | **1** | **0** | **1** | **1** | **0** | **1** | **1** |
| **0** | **1** | **1** | **0** | **1** | **1** | **0** | **0** |
| **0** | **1** | **1** | **1** | **1** | **1** | **0** | **1** |
| **1** | **0** | **0** | **0** | **1** | **1** | **1** | **0** |
| **1** | **0** | **0** | **1** | **1** | **1** | **1** | **1** |

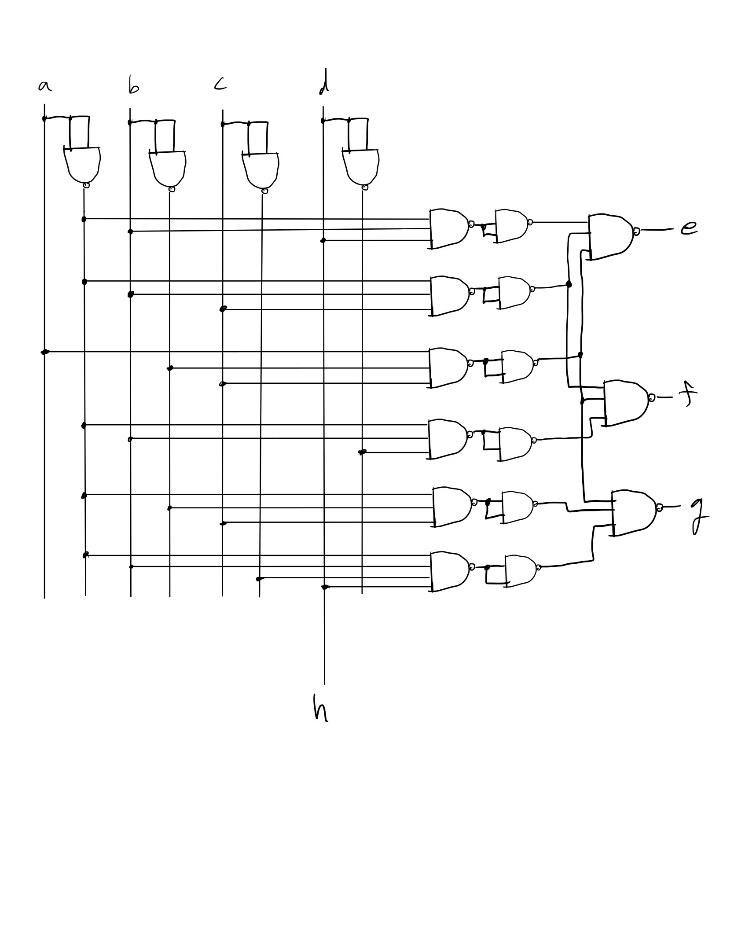
**이를 이용하여 카르노 맵을 작성한 후 SOP를 구하면 다음과 같다.**

****

**POS는 다음과 같다.**

****

**SOP를 바탕으로 코드 변환기를 다음과 같이 설계할 수 있다.**

****

**이제 a를 aa로, b를 bb로, c를 cc로, d를 dd로, e를 e로, f를 f로, g를 g로, h를 h로 대응시킨 후 a, b, c, d를 일정 시간마다 일정 시간마다 변화하도록 설계하면 코드 변환기의 simulation을 구현할 수 있다. 시뮬레이션을 구현하는 코드와 그 결과는 다음과 같다. 시뮬레이션 결과가 진리표와 동일하다는 것을 확인할 수 있다.**

**module inv\_cc\_tb;**

**reg aa, bb, cc, dd;**

**wire e,f,g,h;**

**inv\_cc u\_inv(**

**.a(aa),**

**.b(bb),**

**.c(cc),**

**.d(dd),**

**.e(e),**

**.f(f),**

**.g(g),**

**.h(h));**

**initial begin**

**aa = 1'b0;**

**bb = 1'b1;**

**cc = 1'b0;**

**dd = 1'b1;**

**end**

**always @(aa or bb or cc or dd)**

**begin**

**aa <= #20 ~ aa;**

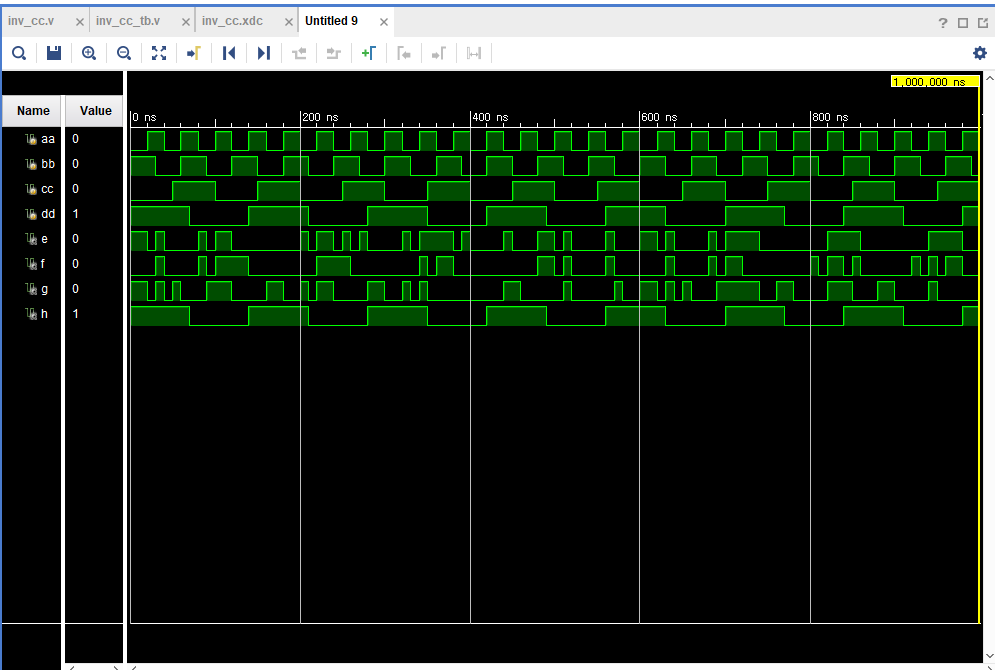
**bb <= #30 ~ bb;**

**cc <= #50 ~ cc;**

**dd <= #70 ~ dd;**

**end**

**endmodule**



**5. 결과 검토 및 논의 사항**

**논리 게이트를 이용하여 가산기와 감산기를 구현할 수 있다는 것을 확인하였다. 가산기에는 두 개의 1비트 수를 더하는 half adder와 두 개의 1비트 수와 이전 단계의 carry를 더하는 full adder가 있었고 각각의 가산기는 sum과 carry를 출력한다. 감산기에는 두 개의 1비트 수를 빼는 half subtractor와 두 개의 1비트 수와 이전 단계의 borrow를 빼는 full subtractor가 있었고 각각의 감산기는 difference와 borrow를 출력한다. 코드 변환기는 8421코드를 2421코드로 변환하는 기능을 수행하는데, 카르노 맵을 통해 SOP와 POS를 구한 뒤 NAND 게이트만 사용하여 구현할 수 있음을 확인하였다.**

**6.**

**조합 논리 회로의 종류에는 앞서 실험했던 가산기, 감산기, 코드 변환기와 함께 디코더가 있다. 디코더는 인풋의 개수에 따라 아웃풋의 개수가 바뀐다. 예를 들어, 2개의 인풋을 가진 디코더는 4개의 아웃풋을 출력할 수 있고 4개의 인풋을 가진 디코더는 16개의 아웃풋을 출력할 수 있다. 2개의 인풋을 가진 디코더를 예로 들면 A와 B가 있을 때 A의 부정과 B의 부정의 AND 게이트의 출력 값, A의 부정과 B의 AND 게이트 출력 값, A와 B의 부정의 AND 게이트 출력 값, A와 B의 AND 게이트 출력 값과 같이 최대 4개의 아웃풋을 생성할 수 있다. 이 때, 디코더 내에서 AND 게이트가 아닌 다른 논리 게이트를 사용하여 출력 값을 구할 수 있다.**