**10주차 결과보고서**

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

**1.**

**4 bit binary parallel adder는 4bit 이진수 2개의 합을 출력하는 장치이다. 따라서 입력으로 4비트 이진수 a, b이 입력되고 4비트 이진수 s가 출력된다. 이번 실험에서는 4비트 이진수를 표현하기 위해 배열을 사용하였다. 배열을 선언하는 방법은 다음과 같다.**

**input[3:0] a, b**

**위 코드는 4비트 이진수인 입력 a, b를 선언한 코드이다. 이는 a[0]부터 a[3]과 b[0]부터 b[3]을 의미한다. 4bit binary parallel adder는 a[0]과 b[0]을 더하고 a[1]과 b[1]을 더하는 방식으로 덧셈을 진행한다. 출력 s도 마찬가지로 배열을 통해 다음과 같이 정의할 수 있다.**

**output[3:0] s**

**a[0]와 b[0]의 합이 s[0]가 되는 방식으로 연산이 진행된다. 4 bit binary parallel adder는 4개의 전가산기로 구성되어 있다. 전가산기는 두 입력과 carry가 더해져 출력이 결정된다. 따라서 4 bit binary parallel adder를 구현하기 위해서는 carry in을 의미하는 ci 역시 필요하다. 첫 번째 전가산기에서 출력되는 carry는 c1으로, 이는 두 번째 전가산기의 carry가 된다. 마찬가지로 두 번째 전가산기에서 출력되는 carry는 c2로, 이는 세 번째 전가산기의 carry가 되고, c3은 네 번째 전가산기에 입력되어 최종적으로 출력되는 carry는 carry out을 의미하는 co가 된다. co는 출력 값에 포함이 되어야 하므로 output으로 정의되고 나머지 carry, 즉 c1, c2, c3는 계산 과정에서는 필요하지만 출력될 필요는 없다. 이러한 변수들을 wire 문법을 사용해 정의할 수 있고, ci, co, c1, c2, c3에 대한 코드는 다음과 같다.**

**input ci**

**output co**

**wire c1, c2, c3;**

**ci와 co는 모듈 내에 선언되어야 하며 wire 구문은 모듈 밖에서 선언한다. 이제 이렇게 선언된 변수들을 사용해 4 bit binary parallel adder를 구현한다. 앞서 얘기했듯이 전가산기의 출력은 carry와 s이다. 전가산기에서 출력되는 s는 두 개의 XOR 게이트로 구성되어 있고, carry는 두 개의 AND 게이트, 하나의 XOR 게이트, 하나의 OR 게이트로 구성되어 있다. 각각의 전가산기에 들어가는 입력은 a[i]와 b[i], 그리고 carry이다. 첫 번째 전가산기에는 a[0]와 b[0]가 입력되고, 이 때의 carry는 4bit binary parallel adder에 입력되는 carry인 ci이다. 첫 번째 전가산기에서는 c1과 s[0]가 출력되는데, c1과 s[0]를 구현하는 코드는 다음과 같다.**

**assign c1 = (a[0] & b[0]) | (ci & (a[0] ^ b[0]));**

**assign s[0] = (a[0] ^ b[0]) ^ ci;**

**이렇게 출력된 c1은 두 번째 전가산기의 carry로 입력되며, s[0]는 출력되는 4비트 이진수의 네번째 숫자가 된다. 두 번째 전가산기의 출력인 c2와 s[1]도 같은 방식으로 구현할 수 있다.**

**assign c2 = (a[1] & b[1]) | (c1 & (a[1] ^ b[1]));**

**assign s[1] = (a[1] ^ b[1]) ^ c1;**

**두 번째 전가산기에는 a[1]과 b[1], 그리고 c1이 입력되므로, 이 세 가지 수로 c2와 s[1]을 구현한다. 세 번째 전가산기와 네 번째 전가산기도 동일한 방법으로 구현되고, 그 코드는 다음과 같다.**

**assign c3 = (a[2]&b[2])|(c2&(a[2]^b[2]));**

**assign s[2] = (a[2]^b[2])^c2;**

**assign co = (a[3]&b[3])|(c3&(a[3]^b[3]));**

**assign s[3] = (a[3]^b[3])^c3;**

**네 번째 전가산기에서는 c4가 아니라 최종적으로 출력되는 co를 구현한다. 각각의 전가산기에서 출력된 s[3]~s[0]은 4 비트 이진수의 각 자리의 수를 나타내고, co가 1일 경우 자리 올림이 발생했다는 것을 의미한다. 4 bit binary parallel adder의 최종적인 디자인 코드는 다음과 같다.**

**`timescale 1ns / 1ps**

**module adder(**

**input[3:0] a, b,**

**input ci,**

**output[3:0] s,**

**output co**

**);**

**wire c1, c2, c3;**

**assign c1 = (a[0]&b[0])|(ci&(a[0]^b[0]));**

**assign s[0] = (a[0]^b[0])^ci;**

**assign c2 = (a[1]&b[1])|(c1&(a[1]^b[1]));**

**assign s[1] = (a[1]^b[1])^c1;**

**assign c3 = (a[2]&b[2])|(c2&(a[2]^b[2]));**

**assign s[2] = (a[2]^b[2])^c2;**

**assign co = (a[3]&b[3])|(c3&(a[3]^b[3]));**

**assign s[3] = (a[3]^b[3])^c3;**

**endmodule**

**이렇게 설계된 디자인 코드를 바탕으로 시뮬레이션 코드를 작성하면 다음과 같다.**

**`timescale 1ns / 1ps**

**module adder\_tb();**

**reg[3:0] a,b;**

**reg ci;**

**wire[3:0] s;**

**wire co;**

**adder u\_inv(**

**.a(a),**

**.b(b),**

**.ci(ci),**

**.s(s),**

**.co(co));**

**initial begin**

**a = 4'b0000;**

**b = 4'b0001;**

**ci = 1'b0;**

**end**

**always @(a or b or ci) begin**

**a <= #10a + 4'b0001;**

**b <= #20b - 4'b0001;**

**ci <= #30 ~ ci;**

**end**

**endmodule**

**기존에 설계하던 시뮬레이션 코드와의 차이점은 배열을 사용했다는 점이다. Reg[3:0]과 wire[3:0]을 사용해 a, b와 s를 배열로 연결할 수 있다. 또 다른 차이점은 a, b를 4bit로 정의하고 일정 시간의 변화에 따라 a는 0001을 더해주고 b는 0001을 감산한 것이다. 1 비트의 경우 ci와 같이 ci의 값을 반전시켜주었지만 4 비트의 경우에는 특정 값을 빼거나 더하였다. 이러한 시뮬레이션 코드로 시뮬레이션 결과를 확인하면 다음과 같다.**

텍스트, 전자기기, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**위 시뮬레이션 결과의 예시처럼 0100과 1111의 합의 결과는 10100이다. 0100은 십진수로 4를 의미하고 1111은 십진수로 16을 의미하기 때문에 합은 20이다. 10100에서 맨 앞에 있는 1은 carry out을 의미한다. 이는 십진수로 16을 의미하기 때문에 s의 값인 0100과 16을 더하면 20이므로 올바르게 출력한 것을 알 수 있다. 이를 실험에서 설계한 디자인 코드에 대입해 보면, c1의 값은 1이 되고, s[0]의 값은 0이 된다. C2의 값은 1, s[1]의 값은 0이 되고, c3의 값은 1, s[2]의 값은 1이 되며, co의 값은 1, s[3]의 값은 0이 된다. 따라서 s[3]~[0]의 값은 0100이 되고, co의 값은 1이 되어 올바른 결과가 출력된 것을 확인할 수 있다.**

**2.**

**4bit binary parallel subtractor는 4비트 이진수 a, b와 borrow in이 입력되고 borrow out과 4비트 이진수 d가 출력된다. 4 bit binary parallel subtractor는 4개의 전감산기로 구성되어 있다. 전체적인 구동 방식은 4비트 이진 병렬 가산기와 비슷하다. 4개의 전감산기에 a, b의 각 자리에 해당하는 이진수와 borrow가 입력되면 다음 전감산기에 입력될 borrow와 각 자리에 해당하는 d가 출력된다. 4 bit binary parallel subtractor에서도 마찬가지로 배열을 이용해 4비트 이진수들을 선언한다. 4비트 이진수 a, b, d에 대한 코드는 다음과 같다.**

**input[3:0] a, b**

**output[3:0] d**

**이는 a[3] ~ a[0]과 b[3] ~ b[0], 그리고 d[3]~d[0]을 의미한다. 이 코드들로 4비트 이진수가 선언되었고 4bit binary parallel subtractor에 입력될 borrow in을 의미하는 bi와 감산기의 결과로 출력될 borrow out을 의미하는 bo를 정의하면 다음과 같다.**

**input bi**

**output bo**

**이 후 4비트 이진 병렬 가산기와 동일하게 연산 과정에서 필요한 출력이지만 최종 결과로 출력되지는 않는 전감산기에서 출력되는 borrow인 b1, b2, b3을 wire를 이용해 정의한다.**

**wire b1, b2, b3;**

**모든 변수를 정의한 뒤에는 전감산기에서 출력되는 각각의 borrow와 d를 구현한다. 첫 번째 전감산기에서는 a[0]와 b[0], 그리고 borrow in이 입력된다. 이를 통해 d[0]와 b1이 출력되고 b1은 두 번째 전감산기의 borrow로 입력된다. 두 번째 전감산기에서는 a[1]과 b[1], 그리고 b1이 입력되고 이를 통해 d[1]과 b2가 출력되며, b2는 다시 세 번째 전감산기의 borrow로 입력된다. 이와 동일한 과정으로 세 번째와 네 번째 전감산기의 연산이 실행된다. 네 번째 전감산기에서는 d[3]과 borrow out이 출력된다. 이렇게 출력된 bo가 4bit binary parallel subtractor의 borrow out이 되고, d[3]~d[0]이 4비트 이진 병렬 감산기에 입력된 4비트 이진수 a, b의 difference가 된다. 첫번째 전감산기에는 a[0]와 b[0], 그리고 bi가 입력되고 전감산기에서 출력되는 b1과 d[0]을 구현하는 코드는 다음과 같다.**

**assign b1 = (~a[0] & b[0]) | (~(a[0] ^ b[0]) & bi);**

**assign d[0] = (a[0] ^ b[0]) ^ bi;**

**b1은 a[0]의 부정과 b[0]가 모두 1이거나 a[0]와 b[0]의 XOR 게이트의 결과의 부정과 bi가 모두 1일 때 1이된다. d[0]는 a[0]과 b[0]의 XOR 게이트의 결과와 bi의 XOR 게이트의 결과가 1일 때 1이된다. 두 번째 전감산기에는 a[1]과 b[1], 그리고 첫 번째 전감산기에서 출력된 b1이 입력되어 b2와 d[1]이 출력된다. 이를 코드로 작성하면 다음과 같다.**

**assign b2 = (~a[1] & b[1]) | (~(a[1] ^ b[1]) & b1);**

**assign d[1] = (a[1] ^ b[1]) ^ b1;**

**이전과 마찬가지로, b2는 a[1]의 부정과 b[1]가 모두 1이거나 a[1]와 b[1]의 XOR 게이트의 결과의 부정과 b1가 모두 1일 때 1이된다. d[1]는 a[1]과 b[1]의 XOR 게이트의 결과와 b1의 XOR 게이트의 결과가 1일 때 1이된다. 이렇게 출력된 b2는 세 번째 전감산기의 borrow로 입력된다. 위와 동일한 방식으로 세 번째, 네 번째 전감산기를 구현하면 다음과 같다.**

**assign b3 = (~a[2] & b[2]) | (~(a[2] ^ b[2]) & b2);**

**assign d[2] = (a[2] ^ b[2]) ^ b2;**

**assign bo = (~a[3] & b[3]) | (~(a[3] ^ b[3]) & b3);**

**assign d[3] = (a[3] ^ b[3]) ^ b3;**

**네 번째 전감산기에서 출력된 borrow는 bo가 되고 이는 4비트 이진 병렬 감산기의 borrow out을 의미한다. 4bit binary parallel subtractor의 전체 디자인 코드는 다음과 같다.**

**`timescale 1ns / 1ps**

**module sub(**

**input[3:0] a, b,**

**input bi,**

**output[3:0] d,**

**output bo**

**);**

**wire b1, b2, b3;**

**assign b1 = (~a[0] & b[0]) | (~(a[0] ^ b[0]) & bi);**

**assign d[0] = (a[0] ^ b[0]) ^ bi;**

**assign b2 = (~a[1] & b[1]) | (~(a[1] ^ b[1]) & b1);**

**assign d[1] = (a[1] ^ b[1]) ^ b1;**

**assign b3 = (~a[2] & b[2]) | (~(a[2] ^ b[2]) & b2);**

**assign d[2] = (a[2] ^ b[2]) ^ b2;**

**assign bo = (~a[3] & b[3]) | (~(a[3] ^ b[3]) & b3);**

**assign d[3] = (a[3] ^ b[3]) ^ b3;**

**endmodule**

**디자인 코드를 구현한 다음에는 시뮬레이션 코드를 작성한다. 시뮬레이션 코드는 다음과 같다.**

**`timescale 1ns / 1ps**

**module sub\_tb();**

**reg[3:0] a,b;**

**reg bi;**

**wire[3:0] d;**

**wire bo;**

**sub u\_inv(**

**.a(a),**

**.b(b),**

**.bi(bi),**

**.d(d),**

**.bo(bo));**

**initial begin**

**a = 4'b0000;**

**b = 4'b0001;**

**bi = 1'b0;**

**end**

**always @(a or b or bi) begin**

**a <= #10a + 4'b0001;**

**b <= #20b - 4'b0001;**

**bi <= #30 ~ bi;**

**end**

**endmodule**

**여기서도 입력되는 두 4비트 이진수인 a, b와 출력되는 4비트 이진수인 d를 배열을 이용해 구현한다. 입력되는 이진수 a, b는 reg[3:0]을 사용하고, 출력되는 이진수 d는 wire[3:0]을 사용한다. 4비트 이진수 a, b, d를 비롯하여 모든 입력과 출력을 디자인 코드의 변수들과 연결한다. 하지만 디자인 코드에서 wire를 사용하여 정의한 b1, b2, b3는 최종적으로 출력되지 않기 때문에 연결하지 않는다. 이 후 a, b의 초기 값을 4비트 이진수인 0000과 0001로 각각 할당하고 bi를 1비트 이진수인 0으로 할당한 뒤 세 변수가 일정한 간격에 따라 변화하도록 설계한다. 일정한 간격에 따라 bi의 값은 이전 값의 보수가 되도록 설정하고, a, b에는 일정한 간격에 따라 0001이 더하거나 빼도록 설정한다. 이 시뮬레이션 코드를 구동한 결과는 다음과 같다.**

텍스트, 전자기기, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**위 시뮬레이션에서 보여지는 예시는 0100과 1111을 감산한 것이다. 즉, 십진수로 4에서 16을 감산한 것을 의미한다. 이 두 4비트 이진수를 실험에서 설계한 디자인 코드에 대입시켜 보면, b1의 값은 1이고 d[0]의 값은 0이다. B2의 값은 1, d[1]의 값은 0, b3의 값은 1, d[2]의 값은 1이며, bo의 값은 1, d[3]의 값은 0이다. Bo의 값이 1이라는 것은 자리 빌림이 발생했다는 것을 의미하고, 십진수로 나타내면 16을 의미하는 값을 빌려와 계산한 것이다. 따라서 자리 빌림이 발생한 0100과 1111의 값은 4이고 이는 0100으로 d의 값과 일치하는 것을 확인할 수 있으므로 올바른 결과가 출력되었다고 할 수 있다.**

**3.**

**BCD adder는 BCD 숫자를 더하는 장치이다. 이 장치에서도 4비트 크기의 두 이진수 a, b가 정의되어야 하기 때문에 앞서 설계했던 4bit binary parallel adder와 같이 배열을 사용해 4비트 이진수들을 정의해야 한다. 또한 최종적으로 출력될 s 역시 4비트 이진수이므로 출력 값도 배열을 통해 정의한다. BCD adder에도 처음 입력될 carry in과 최종적으로 출력될 carry out 역시 선언한다. 이를 코드로 작성하면 다음과 같다.**

**module bcd\_adder(**

**input[3:0] a, b,**

**input ci,**

**output[3:0] s,**

**output co**

**);**

**BCD adder는 두 개의 4bit binary adder로 구성되어 있다. 첫 번째 4bit binary adder에는 4비트 이진수 a, b와 carry in을 의미하는 ci가 입력된다. 이 가산기 내에서 생성되는 carry는 c1, c2, c3이고 가산기에서 출력되는 carry out을 k라고 정의하며 가산기에서 출력되는 덧셈의 결과를 4비트 이진수인 z라고 한다. k의 값이 1이거나 z[3]과 z[2]가 모두 1이거나, z[3]과 z[1]이 모두 1인 경우에 BCD adder의 carry out인 co의 값이 1이 되도록 설계한다. 이렇게 생성된 co는 두 번째 4bit binary adder의 입력이 된다. 두 번째 4bit binary adder에 입력되는 두 4비트 이진수는 0과 co를 조합한 이진수와 첫 번째 4bit binary adder의 출력인 z이다. 0과 co를 조합하는 방식은 0, co, co, 0이다. 즉, co의 값이 1이라면 0110이 입력되는 것이고, co의 값이 0이라면 0000이 입력되는 방식이다. 이렇게 생성된 4비트 이진수와 z의 값이 더해져 최종적으로 4비트 이진수인 s가 출력된다. 두 번째 4bit binary adder에서 생성되는 carry들은 c4, c5, c6이다. 두 개의 4bit binary adder 내에서 생성되는 모든 carry들은 최종적으로 출력될 필요가 없고, 첫 번째 4비트 이진 가산기의 출력인 z와 k 역시 최종적으로 출력되지 않기 때문에 모두 wire를 사용하여 선언한다.**

**wire k, c1, c2, c3, c4, c5, c6;**

**wire[3:0] z;**

**첫 번째 4비트 이진 병렬 가산기를 구현하는 방법은 1번 문항에서 구현했던 방식과 동일하다. 첫 번째 4비트 이진 병렬 가산기의 첫 번째 전가산기에는 a[0]와 b[0], ci가 입력되며 그로 인한 출력은 z[0]와 c1이고, c1은 두 번째 전가산기에 입력되는 carry가 된다. 이와 같은 방식으로 네 번째 전가산기의 출력까지 구현하면 다음과 같다.**

**assign c1 = (a[0]&b[0])|(ci&(a[0]^b[0]));**

**assign z[0] = (a[0]^b[0])^ci;**

**assign c2 = (a[1]&b[1])|(c1&(a[1]^b[1]));**

**assign z[1] = (a[1]^b[1])^c1;**

**assign c3 = (a[2]&b[2])|(c2&(a[2]^b[2]));**

**assign z[2] = (a[2]^b[2])^c2;**

**assign k = (a[3]&b[3])|(c3&(a[3]^b[3]));**

**assign z[3] = (a[3]^b[3])^c3;**

**네 개의 전가산기로 인해 4비트 이진수인 z가 생성되었고, 첫 번째 4비트 이진 가산기의 carry out인 k도 생성되었다. 이제 이렇게 얻은 출력들을 활용해 BCD adder의 carry out인 co를 설계하면 다음과 같다.**

**assign co = (z[3] & z[2]) | (z[3] & z[1]) | k;**

**이렇게 co가 구해졌으니 co와 0을 조합한 이진수와 4비트 이진 병렬 가산기에서 출력된 4비트 이진수인 z를 두 번째 4비트 이진 가산기에 입력한다. 앞서 얘기했던 것처럼 co가 1이라면 0110과 z가, co가 0이라면 0000과 z가 입력된다. 이 때 이진 병렬 가산기에 입력되는 ci는 0이라고 가정한다. 그러면 다음과 같은 코드를 작성하여 두 번째 4비트 이진 병렬 가산기를 구현할 수 있다.**

**assign c4 = (z[0]&0)|(0&(z[0]^0));**

**assign s[0] = (z[0] ^ 0) ^ 0;**

**assign c5 = (z[1]&co)|(c4&(z[1]^co));**

**assign s[1] = (z[1] ^ co) ^ c4;**

**assign c6 = (z[2]&co)|(c5&(z[2]^co));**

**assign s[2] = (z[2] ^ co) ^ c5;**

**assign s[3] = (z[3] ^ 0) ^ c6;**

**위 코드를 통해 최종적인 출력 값인 s[3]~s[0]의 4비트 이진수 s를 구할 수 있다. 따라서 결과적으로 BCD adder에서 co와 s가 출력된다. 이 때 출력되는 co는 4비트 이진 병렬 가산기의 co와 달리 십진수 16을 의미하지 않는다. Co가 1이라면 십의 자리 수가 1이 된다는 것을 의미하는데, 예를 들어, co가 1이고, s의 값이 0011인 경우에는 십진수로 13을 의미한다. 0011이 의미하는 3이 십진수의 일의 자리 수가 되고 co가 십진수의 십의 자리를 의미하기 때문이다. 전체적인 BCD adder의 전체적인 디자인 코드는 다음과 같다.**

**`timescale 1ns / 1ps**

**module bcd\_adder(**

**input[3:0] a, b,**

**input ci,**

**output[3:0] s,**

**output co**

**);**

**wire k, c1, c2, c3, c4, c5, c6;**

**wire[3:0] z;**

**assign c1 = (a[0]&b[0])|(ci&(a[0]^b[0]));**

**assign z[0] = (a[0]^b[0])^ci;**

**assign c2 = (a[1]&b[1])|(c1&(a[1]^b[1]));**

**assign z[1] = (a[1]^b[1])^c1;**

**assign c3 = (a[2]&b[2])|(c2&(a[2]^b[2]));**

**assign z[2] = (a[2]^b[2])^c2;**

**assign k = (a[3]&b[3])|(c3&(a[3]^b[3]));**

**assign z[3] = (a[3]^b[3])^c3;**

**assign co = (z[3] & z[2]) | (z[3] & z[1]) | k;**

**assign c4 = (z[0]&0)|(0&(z[0]^0));**

**assign s[0] = (z[0] ^ 0) ^ 0;**

**assign c5 = (z[1]&co)|(c4&(z[1]^co));**

**assign s[1] = (z[1] ^ co) ^ c4;**

**assign c6 = (z[2]&co)|(c5&(z[2]^co));**

**assign s[2] = (z[2] ^ co) ^ c5;**

**assign s[3] = (z[3] ^ 0) ^ c6;**

**endmodule**

**이제 이 디자인 코드를 바탕으로 시뮬레이션 코드를 작성한다. 시뮬레이션 코드는 앞서 작성한 4비트 이진 병렬 가산기와 유사하다. 두 4비트 이진수 a, b를 reg[3:0]을 사용해 배열로 선언하고, 출력되는 4비트 이진수 s를 wire[3:0]을 사용해 배열로 선언하며, ci와 co는 각각 reg와 wire를 사용해 선언한다. 이렇게 선언된 변수들을 디자인 코드에 해당하는 변수들과 각각 연결시켜준 뒤 a, b의 초기 값은 4비트 이진수인 0000과 0001로 설정하고, 시간이 변화함에 따라 a, b와 ci의 값이 변화하도록 설정한다. 이 때 a, b에는 0001이 각각 더해지거나 빠지도록 설계하고, ci는 보수를 취하도록 설정한다. 전체적인 시뮬레이션 코드는 다음과 같다.**

**`timescale 1ns / 1ps**

**module bcd\_adder\_tb();**

**reg[3:0] a,b;**

**reg ci;**

**wire[3:0] s;**

**wire co;**

**bcd\_adder u\_inv(**

**.a(a),**

**.b(b),**

**.ci(ci),**

**.s(s),**

**.co(co));**

**initial begin**

**a = 4'b0000;**

**b = 4'b0001;**

**ci = 1'b0;**

**end**

**always @(a or b or ci) begin**

**a <= #10a + 4'b0001;**

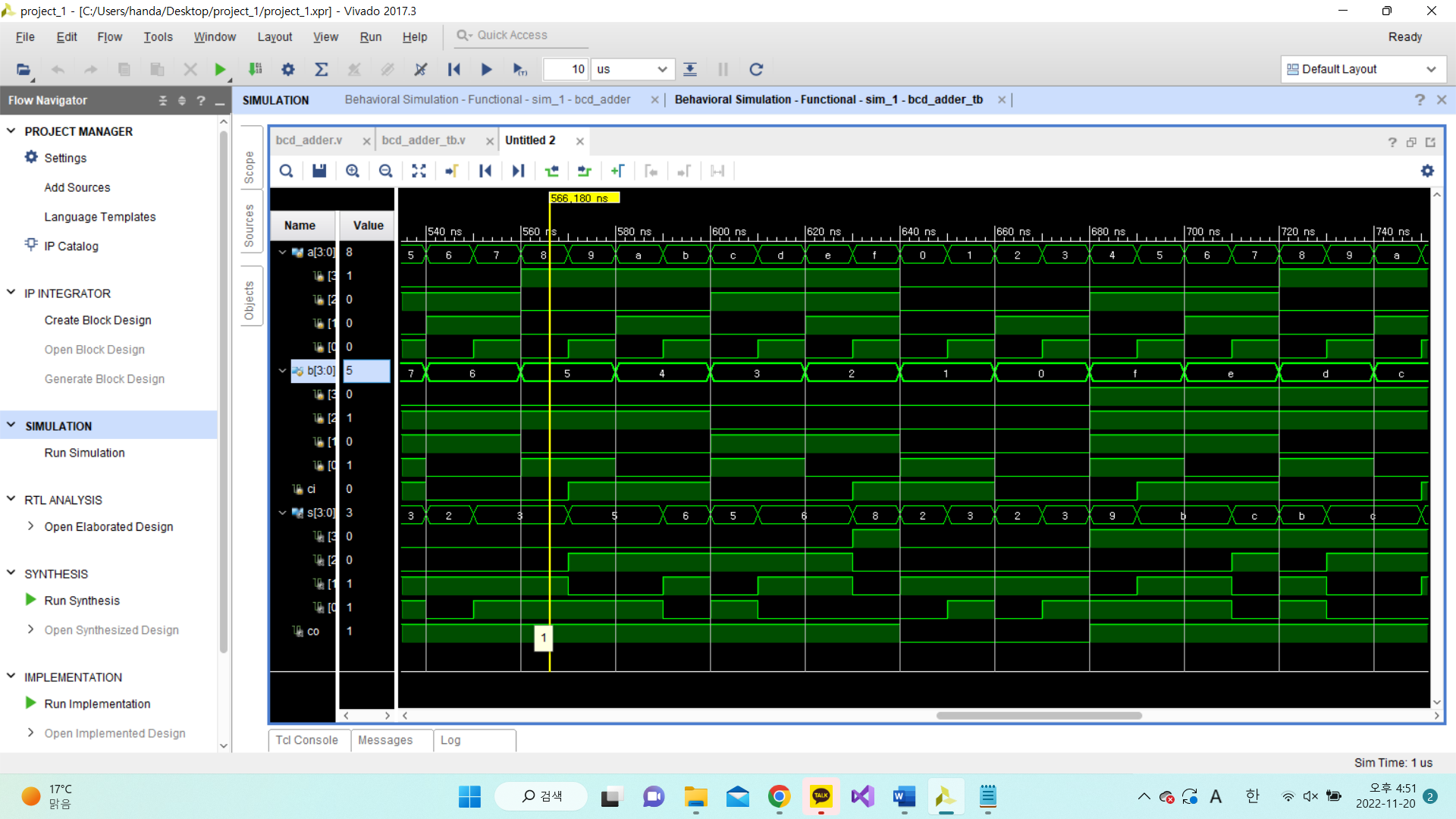
**b <= #20b - 4'b0001;**

**ci <= #30 ~ ci;**

**end**

**endmodule**

**이렇게 설계한 디자인 코드와 시뮬레이션 코드를 통해 시뮬레이션 결과를 확인하면 다음과 같다.**



**위 시뮬레이션에서 보이는 예시는 1000과 0101의 가산이다. 1000은 십진수로 8을 의미하고 0101은 십진수로 5를 의미하기 때문에 출력되는 값은 십진수 13을 의미해야 한다. 먼저 1000과 0101을 첫 번째 4비트 이진 병렬 가산기에 대입시키면 ci가 0이기 때문에 c1은 0, s[0]는 1이 된다. 마찬가지로 나머지 carry와 s[i]를 구해보면, c2는 0, s[1]은 0이고, c3은 0, s[2]는 1이며, k는 0, s[3]는 1이 된다. 따라서 k =0과 1101이 첫 번째 4비트 이진 병렬 가산기에서 출력된다. 이렇게 출력된 값들을 co를 구하는 식에 대입하면 그 값은 1이 된다. 따라서 두 번째 4비트 이진 병렬 가산기에 입력되는 두 4비트 이진수는 1101과 0110이다. 두 4비트 이진수를 두 번째 4비트 이진 병렬 가산기에 대입시켜 출력되는 값은 0011이다. 0011은 십진수의 일의 자리가 3이라는 것을 의미하고, co가 1이라는 것은 십진수의 십의 자리가 1이라는 것을 의미하기 때문에, 최종적인 결과는 13이 되고 이는 올바른 출력이기 때문에 BCD adder가 정상적으로 작동한다는 것을 확인할 수 있다.**

**4.**

**이번 실험에서는 4 bit binary parallel adder/subtractor와 BCD adder를 설계하였다. 4 bit binary parallel adder는 4개의 전가산기로 구성되어 있고, 4 bit binary parallel subtractor는 4개의 전감산기로 구성되어 있으며 BCD adder는 2개의 4비트 이진 병렬 가산기로 구성되어 있었다. 각각의 장치들은 두 개의 4비트 이진수와 carry in/borrow in이 입력되어 4비트 이진수와 carry out/borrow out가 결과로 출력되었다. 전가산기나 전감산기에는 4비트 이진수의 각 자리의 수가 입력되었고, 그 결과로 출력된 carry는 다음 전가산기/전감산기에 입력되는 carry가 되었으며 s[i]/d[i]는 출력되는 s/d의 각 자리의 수가 되었다. 4 bit binary parallel adder에서 출력되는 carry out은 자리 올림을 의미하며 4비트 이진수에서 carry의 값이 1이라는 것은 십진수 16을 의미한다. 4 bit binary parallel subtractor에서 출력되는 borrow out은 자리 빌림을 의미하며 마찬가지로 borrow의 값이 1이라는 것은 십진수 16을 빌려오는 것을 의미한다. 반면, BCD adder에서 carry out의 값은 십진수 16을 의미하는 것이 아니라 출력되는 십진수의 십의 자리 수를 의미한다. 즉 carry out이 1이고, 출력되는 4비트 이진수 s가 0001이라면 십진수 11을 의미하는 것이다. 이번 실험에서는 4 bit binary parallel adder와 4 bit binary parallel subtractor를 각각 전가산기와 전감산기를 사용해 따로 구현하였지만, 전가산기 4개를 이용하여 입력되는 carry in의 값에 따라 adder의 기능을 수행하기도 하고, subtractor의 기능을 수행하기도 하는 장치를 구현하는 것도 가능하여 이를 실험해보는 것도 도움이 될 것이다.**

**5.**

**회로에서 곱셈과 나눗셈을 연산하는 것도 가능하다. 곱셈은 덧셈 명령으로 수행되고, 나눗셈은 뺄셈 명령으로 수행되기 때문에 곱셈과 나눗셈을 연산할 때에도 가산회로와 감산회로가 필수적이다. 곱셈은 반복적인 덧셈으로 연산이 가능하다. 6 \* 4는 6을 4번 더한 것과 같기 때문에 가산 연산의 반복으로 수행할 수 있다. 나눗셈 연산은 조금 더 복잡한데, dividend에서 divisor를 빼고 남은 나머지가 divisor보다 크면 똑 같은 연산을 반복하고 나머지가 divisor보다 작으면 연산을 중단하며 최종적으로 감산하여 산출된 나머지가 나눗셈의 나머지가 된다. 또한 이 때 총 감산한 횟수가 나눗셈의 몫이 된다.**