9주차 결과보고서

전공 : 아트앤테크놀로지 학년 : 3학년 학번 : 20191098 이름 : 백승주

1. 기존의 전가산기는 덧셈을 수행할 때 input의 크기가 1비트였기 때문에 4비트 크기의 이진수들을 덧셈 연산하고자 전가산기들을 4개 병렬 연결한 형태가 4bit Binary Parallel Adder이다.

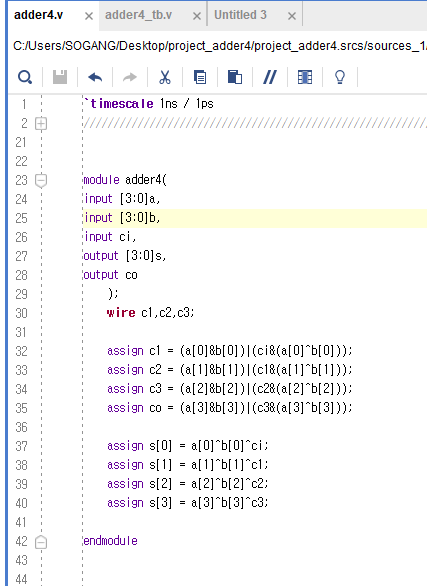
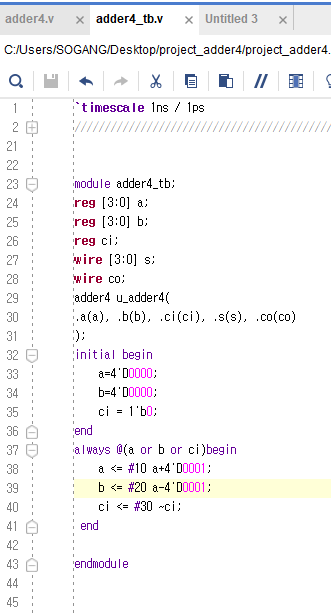


이 4bit Binary Parallel Adder를 소스 코드를 구현해보자.

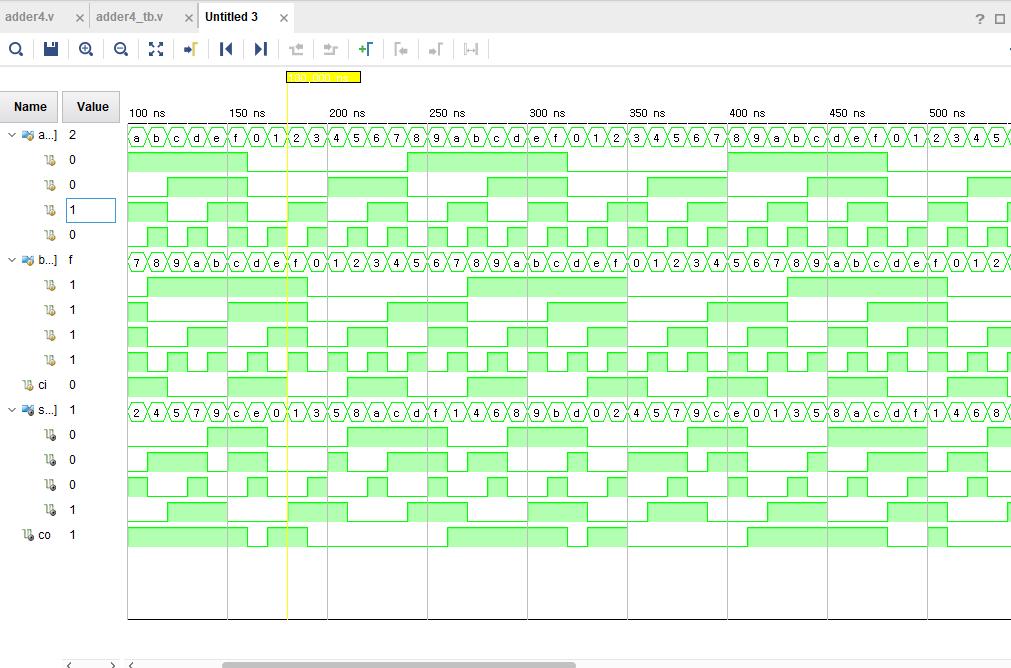
먼저 앞선 실습을 통해 구한 input a,b에 대한 full adder의 코드는 output sum 의 경우 ci^(a^b) 였고 carry out의 경우 ci(a&b)|(a^b) 였다. 이러한 full adder를 네 번 수행하기 때문에 input a, b 에 대해 input [3:0] 으로 선언해 배열로 선언해 각각 4개의 값들을 가지도록 한다. 마찬가지로 output s 역시 배열로 선언한다. 이후 s0, s1, s2, s3 c0, c1 ,c2, ci 에 대해 앞선 full adder 식을 이용해 선언할 수 있다.

결과값 s에 대해 식을 세우면 s[n] = a[n]^b[n]^c[n] 이다. 이 때 c[n]은 이전 단계 adder의 carry out 값이다.

또한 co과는 다르게c0, c1, c2 의 경우에는 각각 s0, s1, s2, 를 구하기 위해서 필요한 값들이고 최종 회로의 결과에는 필요가 없는 값들이다. 최종 회로의 결과에는 s와 co만 있으면 된다. 따라서 이러한 c0, c1, c2 의 경우 output co와 다르게 wire로 선언한다. 이에 따른 Verilog 코드는 다음과 같다.

위 코드를 통해 다음과 같은 시뮬레이션 결과를 얻을 수 있다.



위 시뮬레이션 결과를 자세히 살펴보자.

먼저 a = d , b= 0, ci = 1 인 경우를 살펴보자. 이 때 a배열은 0000의 값을 가지고 b의 배열은 1101의 값을 가진다. 이 경우 a와 b의 값을 더하면 그대로 1101로 b의 값이 동일하지만 ci의 값이 1이므로 1101에 1을 더해야 한다. 그 결과 1101 +1 = 1110이기 때문에 s에는 1110이 출력된다. 이를 16진수로 변환하면 e이다.

다음으로 a = 4 b= 1 ci = 0 인 경우를 살펴보자. 이 경우에는 a 배열은 0100, b 배열은 0001의 값을 가진다. 이 둘을 더하고 ci 값은 이번에는 0이었으므로 결과로 0101이 출력된다.

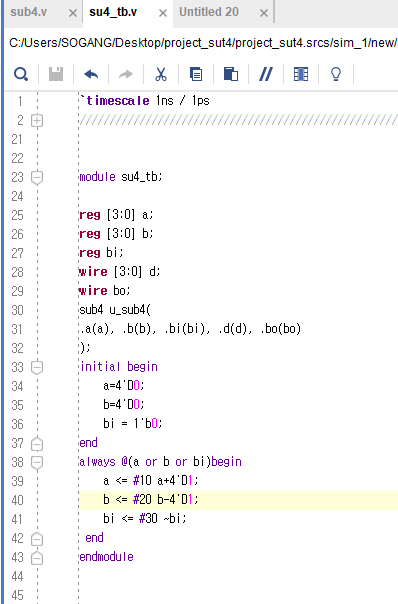
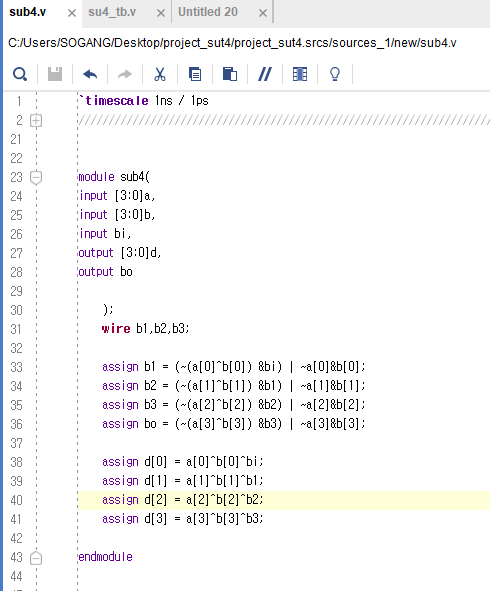
마지막으로 a = e, b = b , ci = 1 인 경우를 살펴보자. 이 때 a 배열의 값들은 1110이다. 또한 b의 배열의 값들은 1011이다.  
 이 두 값들을 더하면 그 값은 11001일 것이다. 또한 ci 값이 1이므로 이 1 값도 더해준다면 그 값들은 11010이어야 한다.  
앞서 선언한 s 값은 1010로 결과가 나왔다. 또한 co 값은 1비트의 값이 나왔다. Co는 carry out의 값을 의미하기 때문에 덧셈의 결과는 11010 임을 의미한다. 즉 이를 16진수로 나타내면 그 값은 a이다..

1. 앞서 살펴본 Binary Parallel Adder와 비슷하게 Binary Parallel Subtractor 역시 같은 방식으로 회로를 구성할 수 있다.

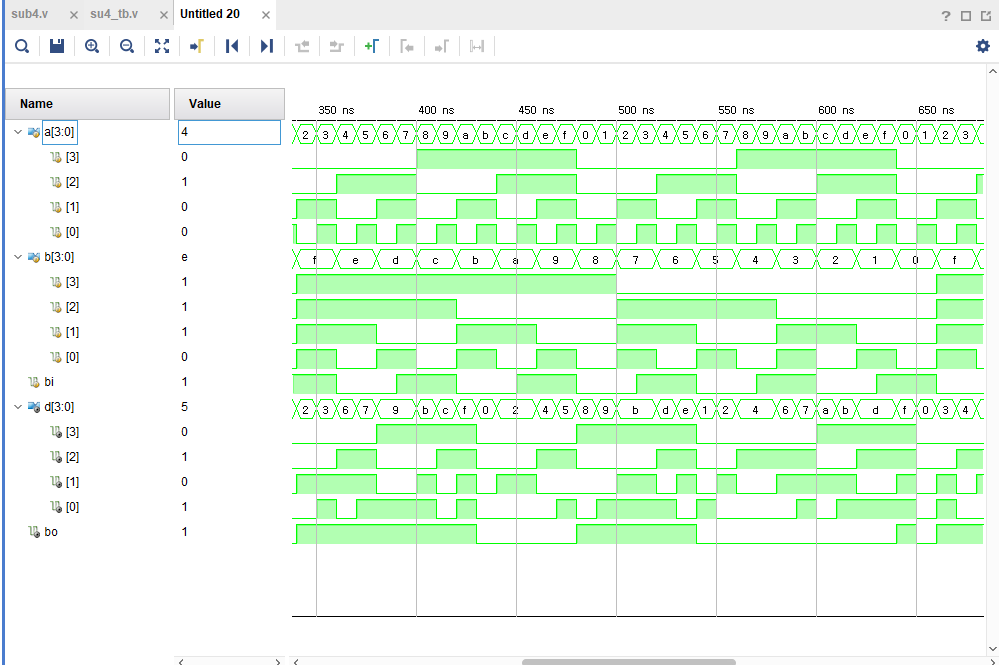


이전 실습에서 구한 전감산기의 뺄셈 값은 a^b^bi 였고 borrow out 값은 (bin& ~(a^b)) + ~a & b 였다. 이를 토대로 Binary Parallel Subtractor 의 뺄셈 결과는 4비트의 d값에 나누어 선언할 수 잇다. 이 때 각 전감산기를 수행하기 위해서는 첫 번째 subtractor를 제외하고는 이전 sutractor의 borrow out 값이 필요하다.

즉 d[n] = a[n]^b[n]^b(이전 단계 borrow out 값) 이고 이때의 Bn은 앞서 언급한 전감산기의 식을 이용해 b[n-1] = (~(a[n]^b[n])&~bi) + ~a[n] & b[n] 라 할 수 있다. 이를 토대로 Verilog 소스 코드를 작성하면 다음과 같다.



위 코드에서는 a값을 1씩 더했고 b값은 1씩 그 값을 빼주어 그 결과를 관찰하고자 했다. 시뮬레이션 결과는 다음과 같다.



위 시뮬레이션의 결과들을 관찰해보자.

먼저 a = 8 b = 4 인 bi = 0인 경우를 살펴보자. 이 경우 a 배열의 값들은 1000이었고 b 배열의 값들은 0100 이었다. 이 1000의 값에 0100 값을 그대로 뺀 결 결과는 d 배열들을 통해 나타나는데 d의 값을 관찰하면 0100이 출력되어 있다. 또한 bo값은 0이다. 즉 , a값이 8이고 b값이 4일 때 감산기를 통해 계산한 값이 4로 올바르게 출력된 것을 알 수 있다.

다음으로 a = 9 b = 4, bi =1 인 경우를 알아보자. 이 경우 a 배열의 값들은 1001 b 배열의 값들은 0111이었다. 시뮬레이션 결과 s 배열의 결과는 0100으로 4가 나왔다. 단순히 9 – 4를 한다면 그 결과가 5가 나와야 하지만 이 경우 bi 값이 1이었기 때문에 bi 값가지 a에서 빼줘야 한다. 따라서 s가 0100을 출력한 것이다.

마지막으로 a= 4, b = e bi= 0인 경우를 살펴보자. 이 때는 a의 값이 b보다 크다. A의 배열은 0100 이고 b의 배열은 1110 이다. 이 계산의 결과는 4 -e는

-a 이므로 s의 결과는 -a가 나와야 할 것이다. 그런데 시뮬레이션 속 s는 6으로 되어있다. 그 이유는 결과가 음수가 되었을 때 값을 borrow out 으로 넘겨주었기 때문이다. 이 borrow out 값이 1이면 -16을 의미한다. 따라서 -16 + 6 = -a 이기 때문에 올바른 결과가 나왔다는 것을 확인할 수 있다.

**3** . BCD code는 십진수를 4비트의 이진수로 변환한 형태인데 이러한 BCD code의 숫자들을 더하는 회로를 BCD adder라고 한다. 아래의 진리표는 일반 전가산기에서 나온 덧셈 결과인 c, s4 s3 s2 s1 에 대해 BCD adder를 규칙을 적용한 값들을 나타낸 진리표이다. 10 미만의 값은 그대로 나타나지만 10이상의 값들은 그 형태가 달라진다.

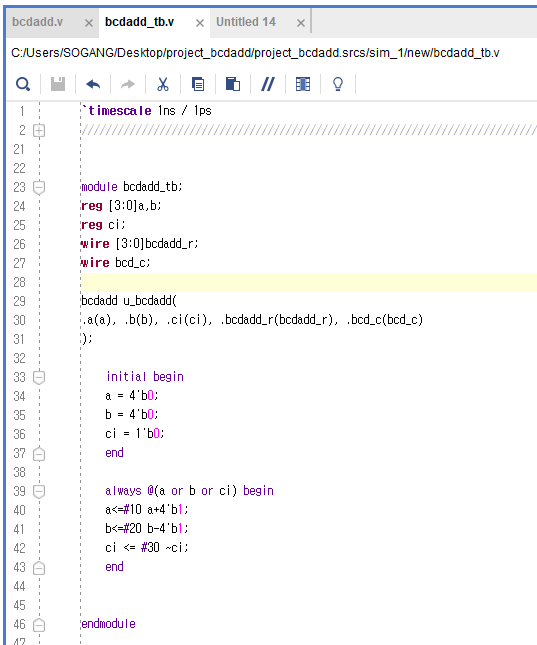
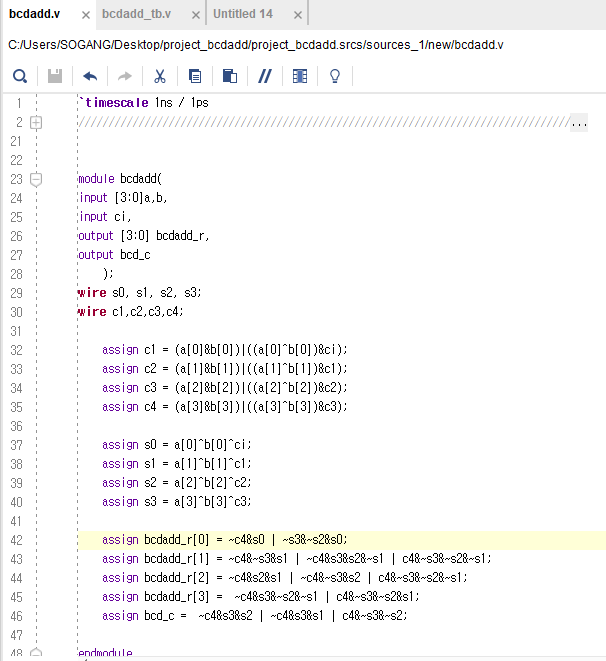
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C | s4 | s3 | s2 | s1 | Cr | Sr3 | Sr2 | Sr1 | Sr0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

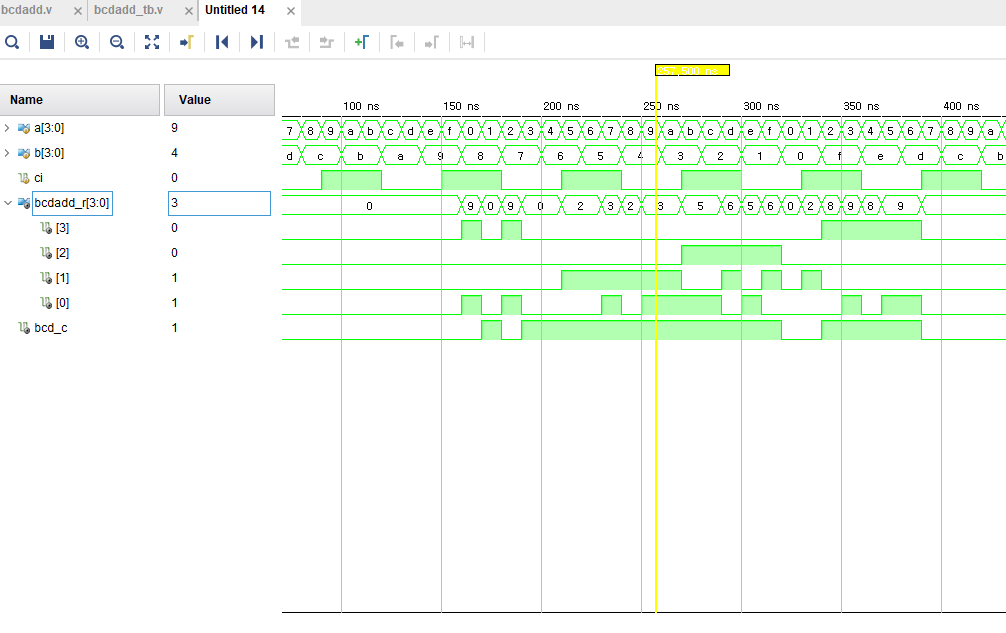
위 진리표에 따라 다음과 같은 카르노 맵을 만들 수 있다.

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

위 카르노 맵에 따라 Verilog 코드를 세우면 다음과 같다.





위 시뮬레이션 결과를 살펴보면 a = 9 b = 4, ci =0 일 때 앞선 4 비트 adder 였다면 더한 값인 9+4=3을 4비트의 이진수 1101값으로 나타냈겠지만 BCD adder는 bcd\_c 값을 1, 즉 10을 의미하는 1로 나타내고 bcdadd\_r 값을 0011, 즉 3을 의미하는 숫자로 나타냄으로써 덧셈의 결과를 표현한 것을 알 수 있다.

또한 a = 2, b=7, ci= 0 인 경우 이 둘의 합은 9로 10 미만의 값을 가지기 때문에 bcdadd\_r 변수 배열에 그 덧셈의 결과가 1001, 즉 9가 그대로 나타난 것을 확인할 수 있다.

마지막으로 a = 9 , b =c, ci= 1 일 때를 비롯한 많은 구간에서 bcdadd\_r 의 값과 bcd\_c의 값이 모두 0으로 나타난 것을 확인할 수 있는데 이 때 9+c+1 = 16 로 16진수의 16 값은 십진수로 표현했을 때 22로 19를 초과한 값이다. 이러한 19를 초과한 값들은 모두 0이 출력되는 것을 확인할 수 있다.

**4.** 4비트의 Binary Parallel Adder 와 Binary Parallel Subtractor BCD adder 의 코드를 짜고 그 시뮬레이션 결과를 확인했다. 이 코드를 짜는 과정에서 이전까지 사용한 적 없는 배열을 사용했는데 여러 비트의 숫자를 사용하는 회로를 시뮬레이션할 경우 이 배열을 통해 각각의 숫자들과 그 숫자들이 나타내는 2진수의 값을 함께 확인할 수 있기 때문에 유용하다는 확인할 수 있었다.

또한 이번에는 회로 내 앞선 adder나 subtractor의 결과가 그 다음 adder의 subtractor에는 필요하지만 최종 회로의 결과에는 필요 없는 경우가 종종 있었는데 이럴 경우 wire 변수를 사용해 시뮬레이션 결과에 나타나지 않게 할 수 있음을 알게 되었다.

**5.** 앞서 BCD adder 를 구현할 때 나는 결과값 19까지의 수에 대한 진리표를 만들어 BCD adder를 구현했다. 이 때문에 덧셈의 결과가 20 이상의 값들에 대해서는 BCD adder가 정상적으로 작동하지 않았다. 따라서 19 이후의 값들까지 모두 적용시키기 위해서는 회로의 식들을 다른 방법으로 고안해야 한다.

이 때 BCD adder의 결과가 4비트 전가산기 adder의 값에 0110, 즉 6을 더해준 값과 일치하다는 사실을 이용할 수 있다. 즉 덧셈의 결과들에 대해 9 이상의 값인지 판별해 9 이상의 값이라면 더하는 값이 6으로 고정된 다른 adder를 회로에 덧붙여 BCD adder를 구성할 수 있을 것이다. 이와 같이 회로를 구성하면 19이상의 값들에 대해서도 결과를 올바르게 출력할 것이다.