10주차 결과보고서

전공: 컴퓨터공학과 학년: 3학년 학번: 20191599 이름: 송경호

**1.**

4-Bit Binary Parallel Adder는 진리표와 카르노맵을 통해서 Boolean Function을 구하기 보다는 Full Adder의 구조를 바탕으로 구현한다.

그래픽이(가) 표시된 사진

자동 생성된 설명

위는 Full Adder의 회로이다. Full Adder는 1개의 입력 비트 두 개를 더한 값을 출력하는 회로이다. 이때, 입력은 3 비트, 출력은 2 비트로 이루어져 있는 것을 볼 수 있는데 이는 Full Adder의 경우 들어오는 Carryin과 나가는 Carryout을 모두 고려할 수 있기 때문이다. 따라서 이러한 Full Adder의 특성을 이용하여 Full Adder를 연결하게 되면 n bit adder를 구할 수 있다. 이때의 연결은 하나의 Full Adder의 Carryout bit를 다음 Full Adder의 Carryin bit로 사용하는 것을 말한다. 해당 과정을 반복하여 4개의 Full Adder를 연결하여 4 bit Binary Parallel Adder를 아래와 같이 구성한다.

**텍스트, 도표, 폰트, 평면도이(가) 표시된 사진

자동 생성된 설명**

4 bit Binary Parallel Adder를 구성하기 위해선 Full Adder의 Boolean Function을 알아야 한다.

S = A ⊕ B ⊕ Cin

Cout = AB + Cin (A ⊕ B)

Full Adder의 출력을 구성하는 S bit와 Cout bit는 위와 같다. n번 Full Adder의 Cout bit가 n+1 Full Adder의 Cin bit임을 고려하며 4개의 Full Adder를 이어 4 bit Binary Parallel Adder를 구성할 수 있다. Verilog 코드는 아래와 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

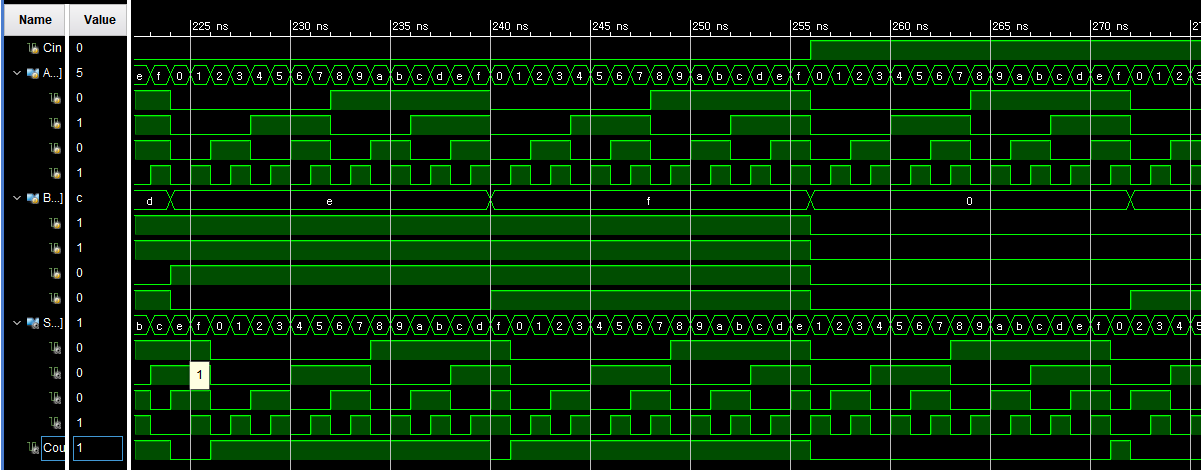
자동 생성된 설명

먼저 디자인 코드이다. 10주차는 실험 요구 사항에 따라 array를 이용했다. 4bit 값 두개를 입력으로 사용하므로 input[3:0] A와 input[3:0] B를 정의했고 이에 따라 4bit 값이 출력되므로 이 역시 output[3:0] S를 통해 array를 이용하여 정의했다. 마지막 carry bit인 Cout와 입력 carry bit인 Cin을 정의했으며 중간 과정에서 다음 full adder로 전달되는 carry bit는 wire을 이용해 정의했다. 또한 full adder의 Boolean function을 이용하여 S와 Cout을 구하고 해당 Cout을 다음 S의 연산에 Cin으로 사용하며 4 bit adder를 구현했다.

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

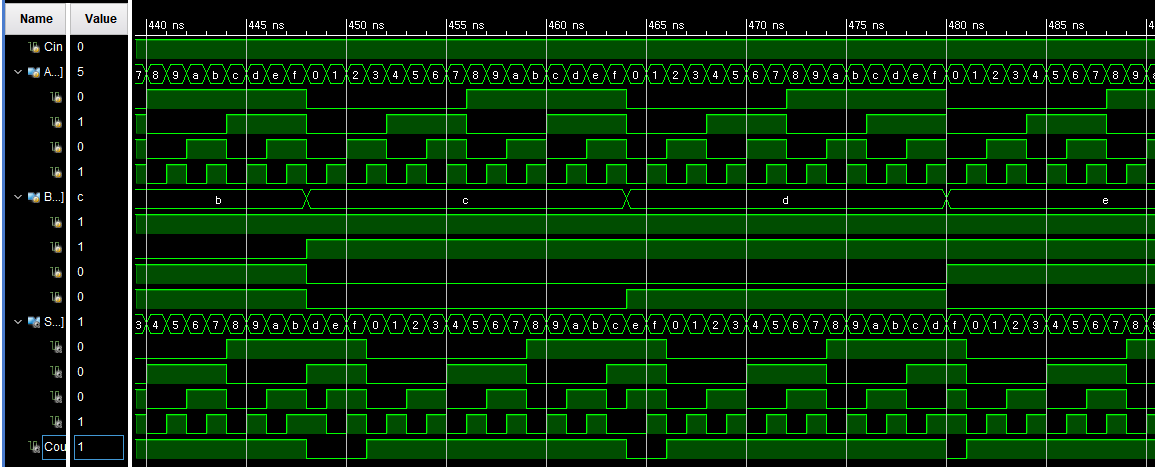
자동 생성된 설명

다음으로 테스트 밴치 코드이다. 입력 비트를 0으로 초기화 해주고 입력 A의 한 비트를 1ns마다 변경시킨다. A는 총 4비트이므로 총 16ns동안 모든 비트 변화를 확인할 수 있으므로 다음 입력인 B는 한 비트를 16ns마다 변경한다. 결론적으로 256ns동안 입력 A, B의 모든 입력 경우의 수를 확인할 수 있으며 최초 carryin bit는 이에 따라 256ns마다 변경된다. 따라서 512ns동안 모든 입력 경우의 수에 대한 출력을 확인할 수 있다. 아래는 위의 디자인 코드와 테스트 밴치 코드의 시뮬레이션 결과이다.



위 시뮬레이션 결과는 전체 512ns의 시뮬레이션 결과 중 일부이다. 먼저 A입력을 확인해보면 224ns부터 240ns까지 240ns부터 256ns까지 패턴이 반복되는 것을 알 수 있다. 이는 1ns마다 A의 값을 변경하여 16ns마다 패턴이 발생하기 때문이다. 또한 B 입력의 모든 경우를 위 사진에서 확인할 수는 없지만 16ns마다 B의 값이 1씩 증가하는 것을 확인할 수 있다.

245 ~ 246ns칸을 확인했을 때 A는 0101, B는 1111 로 각 각 5와 15이다. 10진수 계산 시 해당 결과는 20이 나와야 하며 이를 이진수로 나타내면 10100이 된다. 해당 출력에서 S는 0100 그리고 Cout는 1이며 4 bit의 덧셈이 잘 이루어진 것을 확인할 수 있다.



추가적인 시뮬레이션 결과이다. 이전 시뮬레이션 결과와 가장 큰 차이는 바로 Cin 비트가 1이라는 점이다. 443ns ~ 444ns 칸을 확인했을 때 A는 1011 B는 1011이며 계산 결과는 10110이 되어야 한다. 이때 Cin bit가 1이므로 해당 칸의 S는 0111, Cout은 1로 올바르게 계산된 것을 알 수 있다.

**2.**

4-Bit Binary Parallel Subtractor 역시 진리표와 카르노맵을 통해서 Boolean Function을 구하기 보다는 Full Subtractor의 구조를 바탕으로 구현한다.

도표, 스케치, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

위는 Full Subtractor의 회로이다. Full Subtractor는 1개의 입력 비트 두 개를 감산한 값을 출력하는 회로이다. 이때, 입력은 3 비트, 출력은 2 비트로 이루어져 있는 것을 볼 수 있는데 이는 Full Subtractor의 경우 들어오는 Borrowin과 나가는 Borrowout를 모두 고려할 수 있기 때문이다. 따라서 이러한 Full Subtractor의 특성을 이용하여 Full Subtractor를 연결하게 되면 n bit subtractor를 구할 수 있다. 이때의 연결은 하나의 Full Subtractor의 Borrowout bit를 다음 Full Subtractor의 Borrowin bit로 사용하는 것을 말한다. 해당 과정을 반복하여 4개의 Full Subtractor를 연결하여 4 bit Binary Parallel Subtractor를 아래와 같이 구성한다.

**도표, 평면도, 기술 도면, 스케치이(가) 표시된 사진

자동 생성된 설명**

4 bit Binary Parallel Subtractor를 구성하기 위해선 Full Subtractor의 Boolean Function을 알아야 한다.

D = A ⊕ B ⊕ Cin

Brout = A’B + Brin (A ⊕ B)’

Full Subtractor의 출력을 구성하는 D bit와 Brout bit는 위와 같다. n번 Full Subtractor의 Brout bit가 n+1 Full Subtractor의 Brin bit임을 고려하며 4개의 Full Subtractor를 이어 4 bit Binary Parallel Subtractor를 구성할 수 있다. Verilog 코드는 아래와 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

먼저 디자인 코드이다. 동일하게 array를 이용했다. 4bit 값 두개를 입력으로 사용하므로 input[3:0] A와 input[3:0] B를 정의했고 이에 따라 4bit 값이 출력되므로 이 역시 output[3:0] D를 통해 array를 이용하여 정의했다. 마지막 borrow bit인 Brout와 입력 borrow bit인 Brin을 정의했으며 중간 과정에서 다음 full subtractor로 전달되는 borrow bit는 wire을 이용해 정의했다. 또한 full subtractor의 Boolean function을 이용하여 D와 Brout을 구하고 해당 Brout을 다음 D의 연산에 Brin으로 사용하며 4 bit subtractor를 구현했다.

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

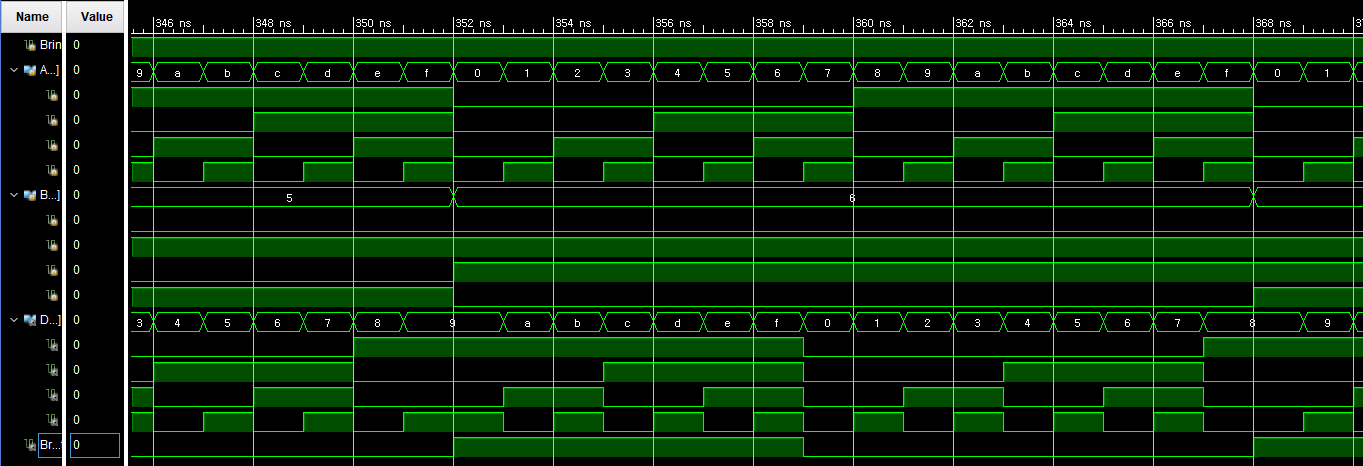
자동 생성된 설명

다음으로 테스트 밴치 코드이다. 입력 비트를 0으로 초기화 해주고 입력 A의 한 비트를 1ns마다 변경시킨다. A는 총 4비트이므로 총 16ns동안 모든 비트 변화를 확인할 수 있으므로 다음 입력인 B는 한 비트를 16ns마다 변경한다. 결론적으로 256ns동안 입력 A, B의 모든 입력 경우의 수를 확인할 수 있으며 최초 borrowin bit는 이에 따라 256ns마다 변경된다. 따라서 512ns동안 모든 입력 경우의 수에 대한 출력을 확인할 수 있다. 아래는 위의 디자인 코드와 테스트 밴치 코드의 시뮬레이션 결과이다.



위 시뮬레이션 결과는 전체 512ns의 시뮬레이션 결과 중 일부이다. 먼저 A입력을 확인해보면 1ns마다1씩 증가하는 것을 확인할 수 있다. 또한 B 입력 역시 16ns마다 1씩 증가하는 것을 확인할 수 있다. 또한 256ns이후로 Brin bit가 0에서 1로 전환되며 테스트 밴치 코드가 의도대로 잘 작동하는 것을 알 수 있다.

247 ~ 248ns칸을 확인했을 때 A는 0111, B는 1111 이며 A-B는 1000과 1버림이 발생해야 한다. 해당 출력에서 D는 1000 그리고 Brout는 1으로 4 bit의 뺄셈이 올바르게 이루어진 것을 확인할 수 있다.



추가적인 시뮬레이션 결과이다. 이전 시뮬레이션 결과와 가장 큰 차이는 바로 Brin 비트가 1이라는 점이다. 349ns ~ 350ns 칸을 확인했을 때 A는 D B는 5이며 두 값의 뺄셈 결과는 1000이 되어야 한다. 이때 Brin이 1이므로 해당 칸의 D는 0111, Br은 0으로 올바르게 계산된 것을 알 수 있다.

**3.**

BCD Adder는 두개의 BCD 값을 더하는 계산기이다. BCD Adder 4 bit adder와 같이 두개의 4bit 입력과 carry 비트를 입력받아 한 개의 4bit 출력과 carry out 비트를 출력한다. 이때, 입력값은 BCD 값이어야 하므로 10이상 즉 1010 이상의 값은 입력되지 않는 다는 것을 알 수 있다.

BCD 연산은 일반적인 4bit 연산과 약간의 차이점이 있다. 바로 10 이상의 수에 대해서 숫자 6에 대응되는 이진수 0100을 추가로 더해줘 올림을 계산해야 한다는 점이다. 예를 들어 13의 경우 이진수로 나타냈을 때 1101이고 10이상의 수이므로 0110을 더하여 0001 0011의 값을 갖도록 한다. 해당 수는 직관적으로 4개의 bit로 이루어진 이진수를 마치 10진수처럼 사용할 수 있게 된다.

도표, 기술 도면, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

위는 BCD Adder의 구조이다. 먼저 Carry out의 경우 4 bit의 Adder의 carry가 발생하거나 S3&S2가 1이거나 S3&S1가 1인 경우 발생한다. 이는 A와 B의 합이 10이상인 경우를 의미한다. 또한 앞서 설명했듯이 A와 B의 합이 10 이상인 경우 6(0110)을 더해줘야 하므로 0 Carry Carry 0 bit를 4 bit adder의 결과에 더해준다면 원하는 결과를 얻게 된다. Verilog 코드는 아래와 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

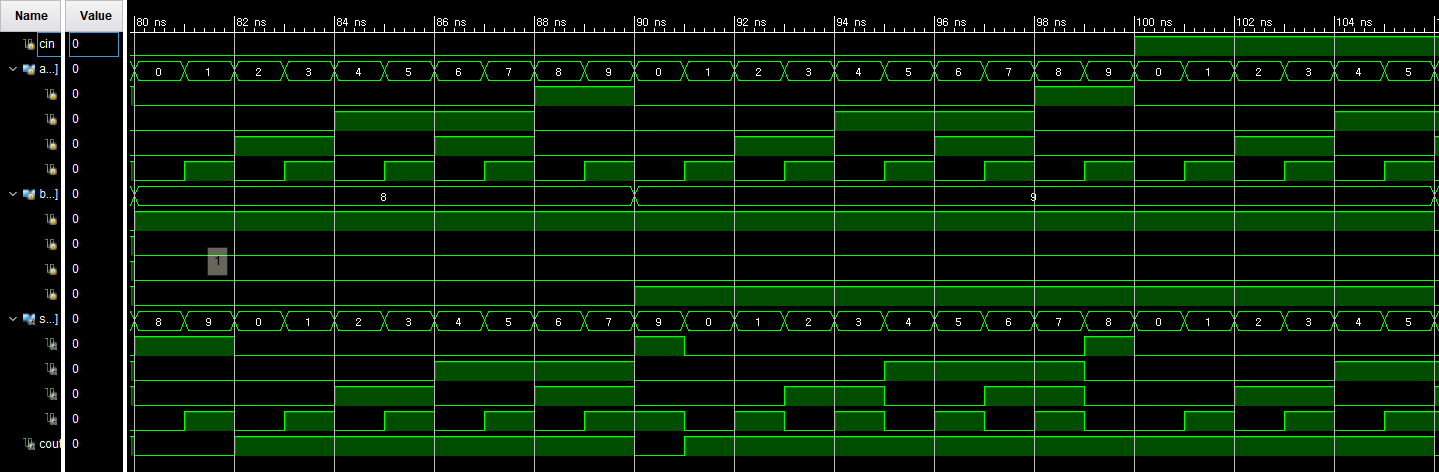
위는 BCD Adder의 디자인 코드이다. BCD Adder 역시 input, out 배열과 wire 배열을 통해 구현했다. a, b는 더하고자 하는 4 bit 입력, sum은 4 bit 출력이다. carryin과 carryout 역시 입력과 출력에 포함한다. 첫 4bit adder에서 사용되는 carry가 c1이며 출력이 S이고 두번째 4bit adder에서 사용되는 carry가 C이고 이들은 wire로 구현했다.

위의 구조에서 봤듯이 cout을 c1과 s를 이용하여 구하며 해당 cout 비트를 두번째 4bit adder의 입력으로 사용하는 것을 알 수 있다.

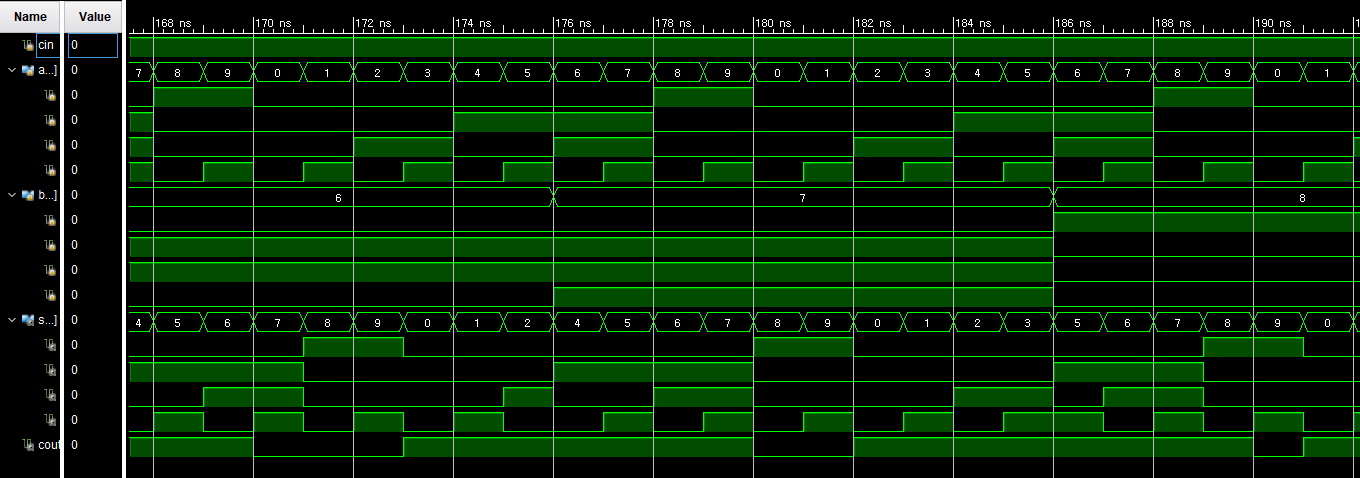
텍스트, 스크린샷, 폰트, 대수학이(가) 표시된 사진

자동 생성된 설명

다음은 테스트 밴치코드이다. 입력 a는 1ns마다 1씩 증가한다. 이때 BCD Adder의 경우 입력값으로 10 이상의 값이 들어올 수 없기 때문에 10에서 다시 0으로 입력을 초기화 해주었다. 따라서 10ns 동안 모든 입력 경우의 수를 확인할 수 있다. 이에 따라 두번째 입력의 변화 주기를 10ns로 주었으며 역시 10에서 다시 0으로 값을 초기화 해주도록 구현했다. 마지막으로 cin bit는 100ns마다 변화를 주어 a와 b의 모든 입력을 확인한 뒤 전환되도록 구현했다. 다음은 시뮬레이션 결과이다.



입력 A는 10ns를 주기로 값이 변화하는 것을 확인할 수 있다. 88ns ~ 89ns칸을 확인하면 A는 1000, B는 1000임을 알 수 있다. 따라서 A+B는 10000이며 이를 BCD 값으로 변화 시 6을 더하여 10110이 되어야 한다. 이때 출력의 sum bit는 0110 cout은 1로 BCD Adder가 올바르게 작동했음을 알 수 있다.



한가지 예시를 추가로 확인한다면 184ns~185ns에서 입력 A는 0100 B는 0111이다. 따라서 두 값을 더한 결과는 1011이 되어야 하며 이를 BCD 값으로 변화 시 6을 더해 10001이 되어야 한다. 이때 출력 sum bit는 0010이고 cout은 1이 되는데 cin이 1인 것을 고려했을 때 올바른 연산이 이루어졌음을 알 수 있다.

**4.**

이번 실험의 경우 카르노맵과 진리표를 구성하지 않고 코드를 구현했다. 회로들이 직관적이기도 했고 진리표는 입력의 경우의 수가 너무 많아 따로 작성하는 대신 기존에 배웠던 Full Adder와 Full Subtractor의 특성을 활용하여 회로를 구성했다.

또한 wire와 array 그리고 module 호출을 이용해 보다 쉽게 여러 입력을 관리하고 결과를 확인할 수 있었다.

테스트 밴치 코드에서는 보다 쉽게 입력들에 대한 출력을 볼 수 있도록 입력의 경우의 수만큼 변화 주기를 주었다. 결과적으로 보다 직관적으로 결과를 확인할 수 있었다.

시뮬레이션 결과와 실제 계산을 비교하며 Boolean function이 올바르게 작성된 것 역시 확인할 수 있었다.

**5.**

**5-1) MSI & LSI**

MSI와 LSI는 디지털 직접 회로의 유형이다. 해당 유형들은 회로내의 직접 소자의 개수로 결정되는데 각각 Medium Scale Integration과 Large Scale Integration의 약자로 MSI가 LSI에 비해 적은 수의 소자를 포함하는 것을 알 수 있다.

MSI는 일반적으로 100개에서 1000개의 직접 소자를 가지며 LSI는 1000개에서 100000개를 갖게된다. MSI는 주로 인코더, 디코더, 카운터, 멀티플렉서, 소형 기억 장치 등에서 사용되며 LSI는 마이크로프로세서, 통신칩, 공장 자동화 시스템에 사용되는 PLC등에서 사용된다.

또한 SSI도 존재하는데 MSI의 이전 단계로 수십 개의 트랜지스터만을 포함하고 소수의 논리게이트를 구현가능하다.

**5-2) 직접 회로의 유형**

직접 회로는 크게 아날로그 IC와 디지털 IC로 구분된다,

[아날로그 IC]

연산 증폭기

연산 증폭기는 입력 신호를 받아 증폭하고, 여러 수학 연산을 수행하는 회로이다. 주로 증폭기, 필터, 비교기, 연산회로 등에 사용된다.

신호 발생기

특정 주파수의 신호를 생성하는 회로이다. 555칩과 566칩 등이 존재하며 각 각 타이머 칩과 주파수 발생기이다.

컨버터

아날로그 신호와 디지털 신호의 변환을 수행한다. ADC와 DAC가 포함된다.

[디지털 IC]

마이크로 프로세서

중앙 처리 장치를 포함하는 컴퓨터 시스템의 핵심이다. 명령어를 해석하고 실행한다.

메모리

ROM과 RAM과 같이 데이터를 저장하는데 사용되는 회로이다.

플래시 메모리

ROM과 RAM의 특성을 결합한 메모리이며 전원이 꺼져도 데이터 보존이 가능하고 액세스 속도가 빠르다는 장점이 있다.