10주차 예비보고서

전공: 아트&테크놀로지학과 학년: 4학년 학번: 20191048 이름: 김도솔

**1.**

먼저 **4-Bit Binary Parallel Adder**는 4bit 짜리 두 이진수의 덧셈 연산을 병렬적으로 수행하는 회로이다. 이 회로는 전가산기(Full Adder) 4개를 사용해 구현할 수 있으며, 이때 각 전가산기의 출력 캐리는 그 다음 전가산기의 캐리 입력으로 연결된다. 각 비트에 대해 별도의 전가산기를 사용하여 동시에 4비트의 이진수 덧셈을 수행할 수 있으며, 각 전가산기는 두 개의 입력 비트와 Carry 입력(이전 비트의 올림)을 고려하여 합과 올림을 계산한다.

도표, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

[그림 1] 4-Bit Binary Parallel Adder

예를 들어, A와 B라는 4비트 숫자가 있다면, 첫 번째 전가산기에서는 , , 캐리 을 입력으로 받고, 출력 합 와 캐리 을 생성한다. 이때 는 출력 합의 첫 번째 비트를 나타내고, 은 다음 전가산기에 연결된다. 두 번째 전가산기에서는 , , 캐리 을 입력으로 받고, 출력 합 와 캐리 를 생성한다. 이때 은 출력 합의 두 번째 비트를 나타내고, 는 다음 전가산기에 연결된다. 세 번째 전가산기도 마찬가지로 , , 캐리 을 입력으로 받고, 출력 합 와 캐리 를 생성한다. 이때 는 출력 합의 세 번째 비트를 나타내고, 는 다음 전가산기에 연결된다. 네 번째 전가산기: , , 캐리 을 입력으로 받고, 출력 합 와 캐리 를 생성한다. 이때 는 출력 합의 네 번째 비트를 나타내고, 는 최종 출력 캐리가 된다. 따라서 4-Bit Binary Parallel Adder는 최종 계산 결과로 (합) = , (캐리) = 를 출력한다.

다음으로 **4-Bit Binary Parallel Subtractor**는 4bit 짜리 두 이진수의 뺄셈 연산을 병렬적으로 수행하는 회로이다. 이는 반감산기와 전감산기의 조합, 또는 감수의 보수 입력을 사용하는 전가산기의 조합 등 여러가지 방법으로 구현될 수 있다.

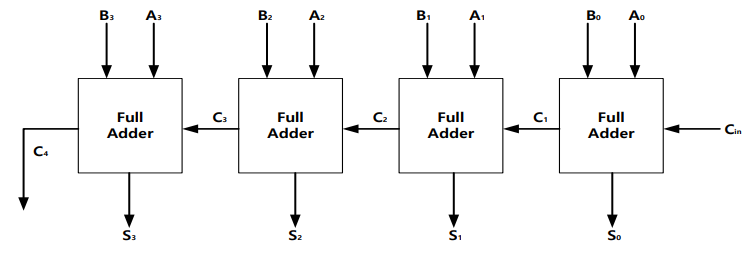
1. 전감산기(Full Subtractor) 4개를 사용해 구현하는 경우, 각 전감산기의 빌림 출력은 그 다음 전가산기의 빌림 입력으로 연결된다. 각 비트에 대해 별도의 전감산기를 사용하여 동시에 4비트의 이진수 뺄셈을 수행할 수 있으며, 각 전감산기는 두 개의 입력 비트와 빌림 입력(이전 비트의 빌림 수)을 고려하여 차이와 빌림을 계산한다. 동작 원리는 앞에서 살펴본 4-Bit Binary Parallel Adder 와 유사하며 최종적으로 (차이) = , (빌림) = 을 출력한다.

도표, 라인, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명

[그림 2] 4-Bit Binary Parallel Subtractor(전감산기 4개로 구현)

1. 4-Bit Binary Parallel Adder에 NOT 게이트를 추가해 구현하는 경우, Subtractor는 4-Bit Adder와 유사하게 작동하지만, 빼기 연산을 수행하기 위해 입력 값 중 하나를 2의 보수(complement)로 변환하여 처리한다. 즉, 첫 번째 입력(A)을 양수로, 두 번째 입력(B)을 해당 비트의 보수로 취급하여 덧셈을 수행한다. 이때 NOT gate를 통해 B의 1의 보수를 얻어내고, 입력 캐리를 통해 1을 더해 B의 2의 보수를 구할 수 있다.



[그림 3] 4-Bit Binary Parallel Subtractor(병렬 가산기에 NOT 게이트 추가)

**2.**

**Look-ahead carry**(예견 올림 수)는 이진 덧셈 과정에서 Carry 비트(올림 비트)를 빠르게 계산하기 위한 방법 중 하나이다. 앞서 살펴본 병렬 가산기는 연쇄적으로 이어져 이전 가산기에서 생성되어야 하는 캐리를 기다려야 한다는 단점이 있다. 즉, 병렬 가산기에서 각 자리의 가산기는 이전 가산기로부터 캐리를 받아야 자신의 연산을 수행할 수 있기 때문에 지연 시간이 발생한다. 이로 인해 더해야 하는 비트 수가 증가하면 전파 지연도 증가하게 된다. 이러한 단점을 보완하기 위해 사용하는 것이 Look ahead carry이다. 이는 입력된 두 비트와 해당 비트의 Carry를 고려하여 미리 다음 비트의 올림을 계산하는 방식을 말한다. 이 방법을 통해 각 비트의 올림을 미리 예측하고 계산함으로써, 전체 덧셈을 보다 빠르게 수행할 수 있다.

Look-ahead carry의 원리를 이용한 adder를 Carry-look-ahead adder(CLA)라 한다. 이는 n-bit adder의 각 전가산기에서 올림을 하위 비트에서부터 계산해오는 대신, 각 단계마다 Carry-Generate signal()과 Carry-Propagate signal() 두 가지 출력을 생성하여 올림을 기다리지 않고 즉시 계산하는 회로이다. CLA의 동작 원리는 다음과 같다.

도표, 스케치, 그림, 기술 도면이(가) 표시된 사진

자동 생성된 설명

[그림 4] 와 를 더하는 1 bit 논리 회로

위 그림은 와 를 더하는 1 bit 논리 회로로, 이를 분석하면 는 와 같은 식으로 나타낼 수 있으며 는 로 나타낼 수 있다. 출력 값인 S와 carry-out은 각각 , 와 같이 나타낼 수 있다. 위의 식을 풀어 4 비트의 carry 값을 나타낸 결과는 다음과 같다.

이때 ~가 모두 , , 로 구성되어 있으므로 4-bit CLA에서 각 전가산기는 이전 전가산기의 캐리 연산을 기다리지 않고 한 번에 ~까지 생성할 수 있다.

스크린샷, 텍스트, 디자인이(가) 표시된 사진

자동 생성된 설명

[그림 5] 4-bit CLA

**3.**

먼저 2의 보수(2's complement)는 이진수에서 음수를 나타내는 방법 중 하나이다. 이를 이용하면 뺄셈을 덧셈으로 바꾸어 계산할 수 있다. 2의 보수를 구하는 방법은 다음과 같다. 우선, 양수를 이진수로 표현하고, 이진수의 비트를 반전시킨다(1은 0으로, 0은 1로 바꾼다), 마지막으로 1을 더한다.

이제 이러한 2의 보수를 사용하여 가감산(덧셈과 뺄셈)을 수행하는 방법을 알아보자. 이진수에서 뺄셈은 빼는 수(감산수)의 2의 보수를 덧셈하는 것으로 바꿀 수 있다. 따라서, 어떤 숫자를 빼고 싶다면 해당 숫자의 2의 보수를 더하는 것과 같다. 예를 들어, 5 - 3을 계산하고 싶다면 우선 5와 3을 2진수로 표현한다. 5를 2진수로 표현하면 0101이고 3을 2진수로 표현하면 0011이다. 다음으로 3의 2의 보수를 구하면 1101이다. 이제 5 + (-3)을 수행하면 된다. 즉, 0101 + 1101 = 10010이다. 여기서 넘어간 자리수는 무시하고 마지막 결과에서 1을 빼면 0010이 되며, 이진수로 표현하면 2가 된다.

**XOR을 활용한 2‘s complement**는 이러한 원리를 이용하여 덧셈과 뺄셈을 동시에 수행할 수 있게 한다. 이는 XOR 연산이 비트를 뒤집는 연산이기 때문이다. XOR 연산의 특성을 다음과 같다.

따라서, 가산과 감산 기능을 동시에 구현하는 방법은, signal line의 값을 두 번째 입력값 B와 XOR 연산을 시켜 그 값을 각 전가산기의 입력으로 사용하는 것이다.

도표, 스케치, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명

[그림 6] XOR 게이트를 활용한 4-Bit Binary Parallel Adder/Subtractor

위 회로도에서 M에 0값을 주면 각 입력은 XOR 게이트를 통과해도 원래 값을 유지해 그대로 덧셈 연산을 수행할 수 있다.M에 1값을 주면 각 입력은 XOR 게이트를 통과해 1의 보수를 취한 값이 된다. 또한 첫 번째 자릿수의 전가산기에 으로 M 값 즉 1이 입력되어 LSB에 1을 더해주게 된다. 최종적으로 각 입력은 2의 보수를 취한 값이 되어 뺄셈 연산을 수행할 수 있다.

**4.**

BCD 코드는 십진수를 이진코드로 표기하는 방법 중 하나로, **BCD 연산**은 BCD 코드의 연산을 말한다. 이는 십진수의 각 자릿수(1의 자리, 10의 자리 등)를 이진수로 변환하여 연산을 수행한 후, 다시 10진수로 변환하는 과정을 거친다. BCD 연산의 한가지 특징은 BCD 연산 결과가 9보다 큰 경우, 즉 결과값이 BCD 코드에서 사용되지 않는 10~15에 해당하는 경우 그 결과값에 6(0110)을 더해서 보정해준다는 것이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

먼저 캐리가 발생하지 않는 경우를 살펴보자. 43과 35를 BCD 코드로 바꾸면 43은 0100 0011이 되고, 35는 0011 0101이 된다. 각 자리수 별로 이진수의 덧셈 연산을 수행하면 일의 자리 수의 연산의 결과는 1000, 십의 자리 수의 연산 결과는 0111로 캐리가 발생하지 않아 결과값을 바로 알 수 있다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

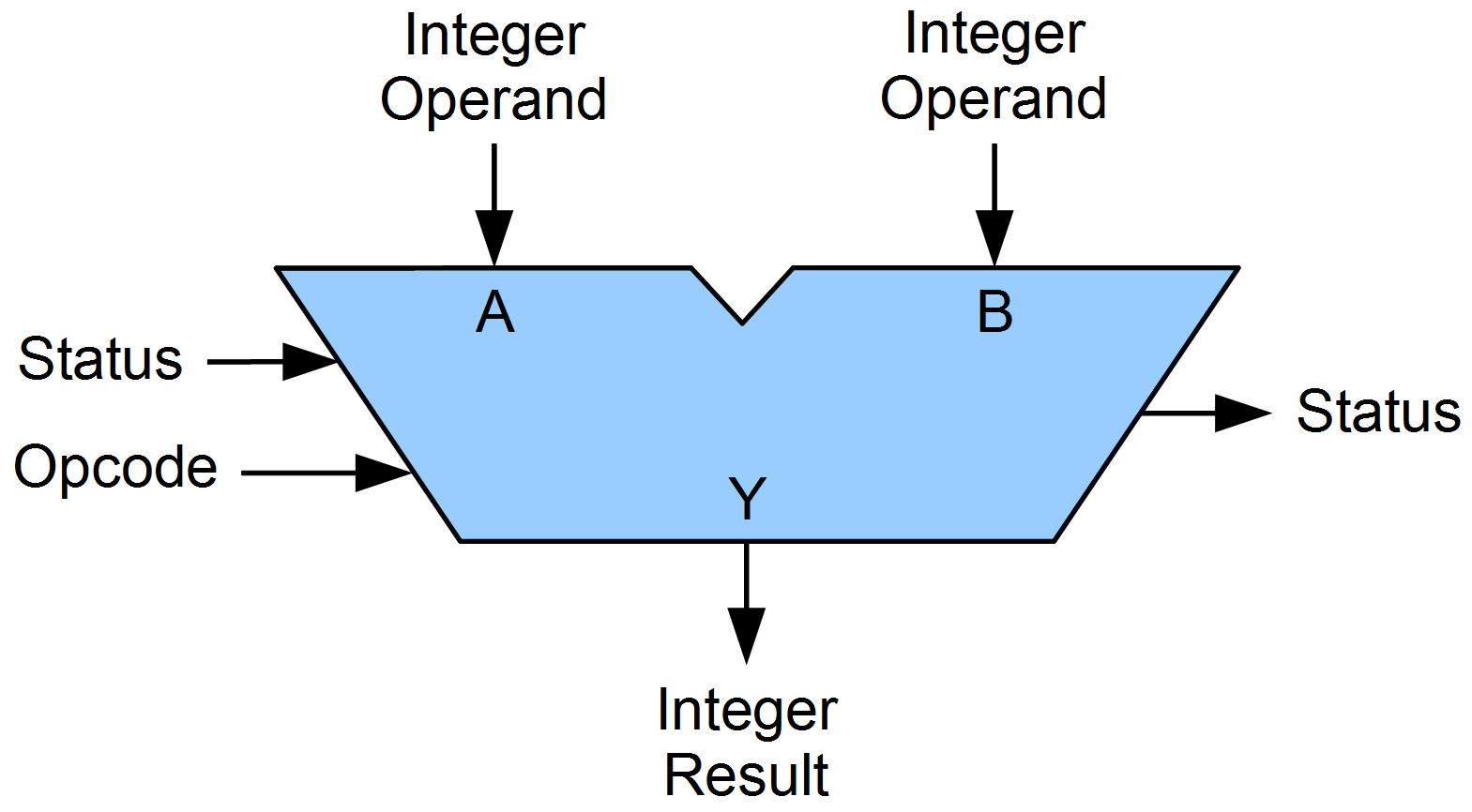
다음으로 캐리가 발생하는 경우를 알아보자. 7과 5를 BCD 코드로 바꾸면 7은 0111, 5는 0011이 된다. 이를 덧셈연산하면 결과로 1100을 얻게 되는데, 1100은 십진수 12로, BCD 코드에서 invalid codeword로 나타난다. 따라서 이 경우 6(0110)을 더해주어 결과값을 보정해준다. 그러면 0001의 캐리가 발생하고 결과값의 일의 자리는 0010이 되어 두 개의 digit으로 12를 나타낼 수 있다.

**5.**

**ALU**는 Arithmetic Logic Unit의 약자로, 컴퓨터의 중앙 처리 장치(CPU) 내에서 연산을 수행하는 부분이다. 주요 연산 기능은 다음과 같다.

1. 산술 연산(Arithmetic Operations): 덧셈, 뺄셈, 곱셈, 나눗셈 등의 산술 연산을 수행한다.
2. 논리 연산(Logical Operations): AND, OR, NOT, XOR와 같은 논리 연산을 수행한다. 이러한 연산은 비트 단위로 수행되어 데이터를 조작하고 비교한다.
3. 시프트 연산(Shift Operations): 비트를 왼쪽이나 오른쪽으로 이동하는 연산을 수행한다. 이는 데이터의 비트를 이동하거나 회전시키는 데 사용된다.
4. 비교 연산(Comparison Operations): 두 개의 데이터를 비교하여 크기나 동등 여부를 확인하는 연산을 수행한다. 이러한 연산은 두 값이 같은지, 크거나 작은지, 혹은 등호를 가지는지 판별하는 데 사용된다.

ALU는 레지스터에서 데이터를 받아 산술 및 논리 연산을 수행하고, 결과를 다시 레지스터에 저장한다. 이러한 동작은 프로그램이나 CPU 내의 명령어에 따라 달라진다. ALU는 CPU의 핵심 부분 중 하나로, 계산, 논리적 판단, 데이터 조작 등과 같은 다양한 연산을 담당하여 컴퓨터의 기능을 구현하는 데 중요한 역할을 한다.



[그림 7] Symbolic representation of ALU

**6.**

**FPU**는 Floating Point Unit의 약자로, 컴퓨터의 중앙 처리 장치(CPU)에 속하는 부분 중 하나이다. FPU는 부동 소수점 연산에 필요한 다양한 연산(덧셈, 뺄셈, 곱셈, 나눗셈)을 빠르고 정확하게 수행할 수 있는 전용 하드웨어이다. 이때 부동 소수점 형식은 실수를 표현하기 위해 지수와 가수의 형태로 나타내며 FPU는 이러한 부동 소수점 형식의 수치 계산을 처리한다. 주요 기능은 다음과 같다.

1. 부동 소수점 연산(Floating Point Operations): 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 부동 소수점 연산을 수행한다.
2. 높은 정밀도 계산(High Precision Calculations): 부동 소수점 연산은 고정 소수점 형식보다 더 높은 정밀도의 연산이 가능하다. FPU는 이러한 높은 정밀도의 연산을 지원한다.
3. 빠른 연산 및 최적화(Optimization for Speed): FPU는 부동 소수점 연산을 빠르게 처리하도록 설계되었다. 이를 통해 수치적 계산이나 과학 및 엔지니어링 분야에서 발생하는 복잡한 연산을 더 효율적으로 수행할 수 있다.

FPU는 CPU와 밀접하게 연결되어 있으며, 일반적으로 별도의 FPU 칩이나 CPU 내에 포함되어 있다. 특히, 수치적으로 계산이 중요한 응용 프로그램에서, 부동 소수점 연산을 빠르고 정확하게 수행하는 FPU는 핵심적인 요소이다. FPU는 컴퓨터의 성능을 향상시키는 데 중요한 부분이며, 과학, 공학, 그래픽스 및 데이터 분석 등 다양한 분야에서 활발하게 사용된다.