

컴실2 발표 대본

안녕하세요. 이번 10주차 MSI / LSI 연산회로 발표를 맡게 된 이성진, 맹재영, 윤성민입니다. 발표 목차는 다음과 같습니다.

(MSI/LSI)

MSI는 Medium Scale Integrated Circuit의 약자로, 하나의 칩에 백에서 천개의 게이트가 구현되어 있는 집적 회로입니다. MSI의 대표적인 예로는 adder, subtractor, comparator, decoder, encoder, multiplexer, demultiplexer, ROM, PLA 등이 있습니다.

LSI는 Large Scale Integrated Circuit의 약자로, 하나의 칩에 수천에서 대략 10만개 정도의 게이트가 구현되어 있는 집적 회로입니다. LSI의 대표적인 예로는 마이크로프로세서 등이 있습니다.

(4-bit binary parallel adder)

Full Adder들을 병렬로 연결하여 여러 비트를 계산할 수 있는 Adder를 Parallel Adder라고 합니다. 그 중에서 2진수의 덧셈을 수행하는 Parallel Adder를 Binary Parallel Adder라고 부릅니다. 마찬가지로 2진수의 뺄셈을 수행하는 subtractor를 Binary Parallel Subtractor라고 부릅니다. Binary Parallel Subtractor은 2의 보수의 덧셈으로 생각하여 계산합니다.

(4-bit binary parallel subtractor)

저희가 오늘 다룰 Binary Parallel Adder/Subtractor는 4-bit를 연산할 4-bit Binary Parallel Adder/Subtractor입니다.

Parallel Adder 및 Parallel subtractor은 직렬형 계산기보다 연산 속도가 더 빠르며, 비용이 덜 든다는 장점이 있습니다. 그러나 carry bit의 존재 때문에 앞의 계산이 완료되지 않으면 계산을 할 수 없기 때문에 자릿수가 늘어날수록 계산 시간이 오래 걸린다는 단점이 있습니다.

(2's Complement Using XOR-1)

2의 보수라고 불리는 2's complement는 2진수의 모든 자릿수를 반전시키는 1의 보수를 취한 뒤, 1을 더해서 만들어집니다. 2의 보수의 예시는 화면의 그림과 같습니다. 5를 2진수로 나타내면 0101 이라는 수로 표현이 되는데, 이에 1의 보수를 취하면 1010이라는 수가 되고, 1010에 1을 더하면 1011이라는 수가 됩니다. 1011은 -5와 동일합니다.

이처럼 2의 보수를 사용하는 이유는 계산이 쉬워지기 때문입니다. 2의 보수는 계산 과정에서 원래 숫자의 음수로 취급되어 계산되기 때문에, 뺄셈 연산을 덧셈 연산과 동일하게 계산할 수 있습니다.

(2's Complement Using XOR-2)

다음 화면의 그림과 같이, 필요에 따라 XOR 연산을 통해 1의 보수를 원할 때 취할 수 있습니다. 인풋으로 0을 입력하면 출력값은 원래 입력으로 들어온 비트 그대로 나오고, 1을 입력하면 입력으로 들어온 비트에 1의 보수를 취해서 나오게 됩니다. 이를 응용하면 다음 슬라이드의 그림과 같이 Adder와 Subtractor을 원하는 대로 바꾸어서 연산할 수 있도록 계산기를 만들 수 있습니다.

(2's Complement Using XOR-3)

K값으로 0을 입력하면 B값이 변하지 않아 덧셈이 되고, K값으로 1을 입력하면 B값에 1의 보수를 취하고, 첫 번째 Full Adder에 Cin으로도 1이 들어오기 때문에 1을 더한 것과 같습니다. 결과적으로 Full Adder을 가지고 뺄셈 연산을 수행했습니다.

CLA (1) - 다음으로 4비트 carry look ahead adder에 대해 설명하겠습니다. 앞서 설명해드린 4비트 병렬 가산기를 구성하는 Full Adder는 여러 게이트들로 구성되어 있고 이 각각의 게이트들은 gate delay가 존재합니다. 따라서 carry값을 구하기 위해 이전의 carry값들의 연산 과정을 모두 기다려야 하는 4bit 병렬 가산기는 연산과정에서 gate delay가 많이 발생하게 되어 시간이 오래 걸린다는 단점이 있습니다.

CLA(2) - 이러한 병렬 가산기의 속도 문제를 해결하기 위한 방법이 CLA라고 하는 Carry Look Ahead Adder입니다. CLA Adder는 병렬 가산기에서 시간 측면의 단점을 발생시킨 carry 연산 부분을 그림에서처럼 CLA generator로 대체하여 속도의 문제를 해결합니다.

CLA(3) - CLA generator의 연산을 위해서 두 가지의 함수가 필요한데 하나는 carry generate 함수로 논리식은 $g_i = x_i y_i$ 이고 다른 하나는 carry propagate 함수로 논리식은 $p_i = x_i \oplus y_i$ 입니다. 여기서 G함수는 캐리 생성 함수로 기존 연산과 관계 없이 반드시 캐리가 생성됨을 확인하는 함수이고 P함수는 추가적으로 캐리가 발생할 가능성을 검사하는 함수입니다. 이 두 함수로 아래 두 줄을 보시면 Full Adder에서의 두 논리식을 G와 P를 이용해 다음과 같이 새로운 식으로 표현할 수 있습니다.

CLA(4) - 이때 C를 구해주는 식을 보면 좌변에는 C_{i+1} 이 있고 우변에는 C_i 가 있는 것을 볼 수 있습니다. 이는 초기 입력으로 주어지는 C_i 를 통해 아래 4개와 같이 재귀적으로 C_{i+1} 부터 C_{i+4} 까지 한번에 구할 수 있음을 의미합니다. 따라서 기존의 병렬 가산기처럼 이전의 carry값들의 연산과정을 모두 기다릴 필요 없이 초기 입력값만으로 모든 carry값을 구할 수 있게 되고 이로 인해 carry 연산으로 발생하는 속도 문제를 해결할 수 있게 됩니다.

BCD(1) - 다음으로 BCD adder에 대해 설명하겠습니다. BCD adder는 BCD 코드로 표현된 값들을 더하거나 빼는 연산을 해서 다시 BCD 코드로 표현하는 가산기를 말합니다. BCD코드는 이전 실습에서도 몇 번 다뤘었던 내용이라 모두 아시다시피 십진수 0~9까지를 나타내기 위한 코드이기 때문에 10이상의 십진수는 4비트 이진코드로 표현할 수 없습니다. 따라서 나와있는 왼쪽 예시와 같이 BCD연산에서 피연산자와 연산 결과가 모두 0~9안에 포함되어 있는 값이면 4비트로 표현하는데에 문제가 없지만 오른쪽 예시와 같이 연산 결과가 4비트 이진코드로 표현되지 않을 수도 있습니다.

BCD(2) - 이러한 경우 연산결과에 6에 해당하는 이진코드인 0110을 더해주어 12라는 값을 0001 0010의 BCD 코드로 표현할 수 있게 된다. 여기서 6만큼, 즉 0110만큼 더해주는 것은 BCD 코드에서 표현되는 4비트 이진코드 외에 사용되지 않는 10~15에 해당하는 6개의 4비트 이진코드가 있기 때문에 6만큼을 더해서 그에 맞는 BCD 코드로 표현해주는 것입니다.

BCD(3) - 이렇게 10이상의 두자리 십진수말고도 세자리 십진수도 다음과 같이 BCD 코드로 표현할 수 있습니다.

BCD(4) - 이 그림은 저희가 이번 실습에서 구현하는 BCD adder의 회로도를 나타낸 것입니다. 앞에서 설명해드린 BCD 연산과정을 이 회로도를 통해 다시 확인할 수 있습니다. 먼저 빨간색 네모칸의 부분은 입력으로 들어오는 2개의 4비트 이진수를 Full Adder를 통해 더하는 과정입니다. 여기서 덧셈 결과가 16이상의 값이 나오게 되면 Full Adder에서 carry out에 해당하는 K가 1의 값을 갖게 됩니다.

BCD(5) - 덧셈 후 연산 결과를 s3s2s1s0라고 할 때 연산 결과가 옆의 표에 나와있듯이 10~15에 해당하는 값이면 s3와 s2가 1이거나 s3와 s1이 1이거나 또는 s3,s2,s1이 모두 1의 값을 갖게 됩니다. 따라서 이를 확인하기 위해 s3와 s1, 그리고 s3와 s2를 and 게이트의 input으로 하여 연산결과가 10~15사이의 값인지 확인합니다. 따라서 연산 결과가 16이상이거나 10~15의 값을 갖게 되면 제일 왼쪽의 OR게이트는 1의 값을 출력하게 되고 연산결과가 0~9의 값이면 0의 값을 출력합니다.

BCD(6) - 이렇게 나온 OR게이트의 출력값이 0이면 연산 결과가 그대로 나오게 되고 1이 되면 연산결과인 s3s2s1s0와 그림에 나와있듯이 0110, 앞서 설명드렸듯이 6만큼을 더해 연산 결과를 BCD 코드로 표현하게 됩니다.

alu는 arethematic and logic unit의 약자로 산술연산과 논리연산과 추가적으로 다양한 연산결과를 출력해주는 장치입니다. 이때 2가지의 피연산자와 status, opcode를 입력받고 연산결과 와 status를 출력합니다. opcode는 어떤 연산을 하는지 알려주는 코드이고 MUX를 통해 연산결과를 선택하여 전달합니다. status는 주로 산술연산에서 carryin이 있는지 없는지 정도라고 생각하면 됩니다.

alu에서 회로구성을 조금더 살펴보면, a 와 b의 input이 주어졌을때, and와 or 연산을 mux에 연결하여 아까 opcode를 통해 필요한 결과를 선택하여 전달하는 모습입니다.

여기에 adder회로를 추가하면 mux의 선택지가 하나 늘어나는 셈이고 carryin status 입력이 추가적으로 필요해집니다.

또한 앞서 설명드린 2's complement를 사용하면 adder로 subtractor 기능까지 연산이 가능하여 그 부분만을 (1)번 위치에 추가하고 (2)번의 carryin은 1을 받게 됩니다.

출력의 status는 기본적으로 다음과같이 음수, 0, carry 또는 borrow out, overflow의 상태가 있는데 나머지는 직관적으로 바로 알수있고 overflow에 대해서만 추가적인 설명을 하자면 예를들어 4bit 에서 unsigned 이면 범위가 0~15이고 signed이면 범위가 -8~7인데 이때 signed 체계를 사용하게되면 8~15까지의 숫자를 표현하기가 불가능해지고 이럴경우 overflow가 발생했다고 말합니다.

이 표는 overflow가 발생 가능성이 있는 조건이고 여기까지가 alu에 관한 설명이었습니다. 감사합니다.