전공: 컴퓨터공학 학년: 2학년 학번: 20161603 이름: 신민준

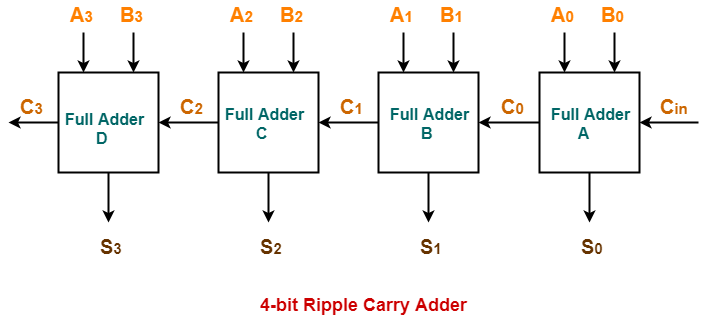
1. 4 bit binary parallel adder의 결과 및 simulation 과정에 대해서 설명하시오.

4 bit binary parallel adder는 1-bit 입력에 대한 full adder를 cascading design을 따라 4개 나열한 Ripple-carry adder이다. 따라서, 4-bit binary parallel adder를 구현하기에 앞서, 다음과 같이 full adder를 Verilog code로 작성하였다.

이 때 사용한 수식은 다음과 같다

|  |
| --- |
| `timescale 1ns / 1ps  module full\_adder(a, b, cin, sum, cout);  input a, b, cin;  output sum, cout;  assign cout = a&b|cin&(a^b);  assign sum = (a^b)^cin;  endmodule |

위의 full adder 모듈을 사용해 4-bit binary parallel adder를 구현하기 위해선, 다음과 같은 다이어그램을 따라 회로를 구성해야 한다.



추가적으로 sign bit을 입력받도록 해, 입력값 에 대해 XOR 연산을 입력받은 sign bit과 수행시키고, 에 sign bit을 전달해, 다음 다이어그램과 같은 회로도를 설계해 4-bit binary parallel subtractor의 역할도 동시에 수행할 수 있도록 구성했다. Subtractor 기능에 대해서는 2번 목차에서 추가적으로 설명한다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

설명한 모든 기능을 추가해 최종적으로 4-bit Ripple Carry Adder를 구현한 Verilog code는 다음과 같다.

|  |
| --- |
| `timescale 1ns / 1ps  module rca\_4bit(a, b, sign, s, c4);  input [3:0] a, b;  output [3:0] s;  input sign;  output c4;  wire c1, c2, c3;  full\_adder fa0(a[0], b[0]^sign, sign, s[0], c1);  full\_adder fa1(a[1], b[1]^sign, c1, s[1], c2);  full\_adder fa2(a[2], b[2]^sign, c2, s[2], c3);  full\_adder fa3(a[3], b[3]^sign, c3, s[3], c4);  endmodule |

입력을 받을 때, 각각 비트의 자리들을 따로 받게 되면 의미없이 코드가 복잡해지므로, 배열로 한꺼번에 입력 받는 방식을 선택했다. 출력 값인 s에 대해서도 같은 이유로 배열로 선택했다.

Net형 c1, c2, c3를 선언하고, 각각이 0번째, 1번째, 2번째 full adder의 출력을 다음 full adder로 연결하도록 구성했고, 최종적으로 3번째 full adder의 출력을 output line인 c4와 연결해 을 구현했다.

위 Verilog code로 구현한 모듈의 RTL Schematic은 다음과 같다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

이 때, 4 bit ripple carry adder를 구성하는 full adder들의 schematic은 다음과 같이 확인이 가능했다.

스크린샷이(가) 표시된 사진

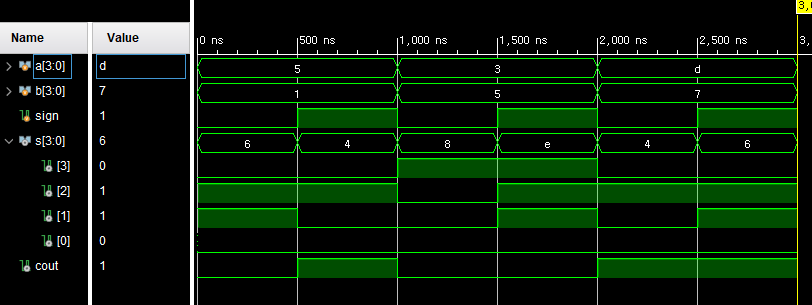
자동 생성된 설명

의도했던 대로 회로도가 그려진 것을 확인할 수 있었다.

구현한 4-bit binary parallel adder가 정확하게 작동하는 것을 확인하기 위해, 다음과 같이 Simulation code를 작성했다.

|  |
| --- |
| `timescale 1ns / 1ps  module rcasim;  reg [3:0] a, b;  reg sign;  wire [3:0] s;  wire cout;  initial a = 4'b0101;  initial b = 4'b0001;  initial sign = 1'b0;  // 5+1  rca\_4bit rca4(a, b, sign, s, cout);    initial begin  #500  assign sign = 1'b1;  // 5-1  #500  assign sign = 1'b0;  assign a=4'b0011;  assign b=4'b0101;  //3+5  #500  assign sign = 1'b1;  //3-5  #500  assign sign = 1'b0;  assign a=4'b1101;  assign b=4'b0111;  //13+7  #500  assign sign = 1'b1;  //13-7  #500  $finish;  end  endmodule |

4bit 길이의 배열 a, b에 계산할 두 값, 즉 operands를 입력하고, sign에 0 값을 넣어주면 결과가 cout과 4bit 길이의 배열 s에 저장된다. 위 simulation을 수행할 결과는 다음과 같다.



sign=0 인 경우에, 두 값 a, b에 addition을 수행한 값이 s와 cout에 정확하게 나타나는 것을 확인할 수 있다.

1. 4 bit binary parallel subtractor의 결과 및 simulation 과정에 대해서 설명하시오.

4-bit binary parallel subtractor를 구현하기 위해, 위 1. 번에서 구현한 ripple carry adder에 약간의 조정만을 더해 addition과 subtraction을 둘 다 수행할 수 있는 모듈을 만들었다.

기본적인 개념은 XOR을 사용한 연산의 특성을 사용했다. 즉,

이므로, 뺄셈을 수행하려면, 두 번째 operand인 b를 1과 XOR연산을 시켜 invert시킨 후, 에 1을 넣어 b의 값을 two’s complement상의 음수 표기법으로 바꾸고, a값을 바뀐 b와 정상적으로 덧셈을 시키면, 결과적으로 a와 b의 subtraction을 수행할 수 있다.

위 개념을 추가한 것이 위에서 보인 다음 코드이다.

|  |
| --- |
| `timescale 1ns / 1ps  module rca\_4bit(a, b, sign, s, c4);  input [3:0] a, b;  output [3:0] s;  input sign;  output c4;  wire c1, c2, c3;  full\_adder fa0(a[0], b[0]^sign, sign, s[0], c1);  full\_adder fa1(a[1], b[1]^sign, c1, s[1], c2);  full\_adder fa2(a[2], b[2]^sign, c2, s[2], c3);  full\_adder fa3(a[3], b[3]^sign, c3, s[3], c4);  endmodule |

Sign 비트를 입력받아, b의 각 bit와 xor 연산을 수행하고, 동시에 입력받은 sign 비트를 0번째 full adder의 인자로 연결시켜주어 sign bit가 1이면 결과적으로 뺄셈을 수행할 수 있도록 구현했다. XOR 연산의 특성 로부터, sign 비트가 0이면 정상적으로 덧셈이 실행됨 또한 확인할 수 있다.

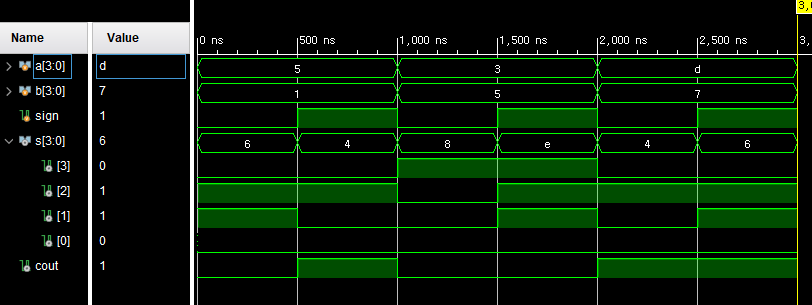
1.번 과정에서 작성한 Verilog code와 이번 subtractor를 구현한 코드는 동일하므로, RTL Schematic은 1.에서 보인 것과 동일하다.

추가적으로 구현한 이 모듈이 정상적으로 동작하는 지 확인하기 위해 동일한 simulation source를 실행시켜 확인해 보았다.

|  |
| --- |
| `timescale 1ns / 1ps  module rcasim;  reg [3:0] a, b;  reg sign;  wire [3:0] s;  wire cout;  initial a = 4'b0101;  initial b = 4'b0001;  initial sign = 1'b0;  // 5+1  rca\_4bit rca4(a, b, sign, s, cout);    initial begin  #500  assign sign = 1'b1;  // 5-1  #500  assign sign = 1'b0;  assign a=4'b0011;  assign b=4'b0101;  //3+5  #500  assign sign = 1'b1;  //3-5  #500  assign sign = 1'b0;  assign a=4'b1101;  assign b=4'b0111;  //13+7  #500  assign sign = 1'b1;  //13-7  #500  $finish;  end  endmodule |

1.번의 경우와 반대로, sign bit가 1이 될 때 a와 b 값의 뺄셈이 일어나 결과 값이 s와 cout에 정상적으로 출력되는지 확인하는 것이 목적이다.

위 시뮬레이션의 결과는 1.번에서 보인 결과와 동일하다.



Sign bit가 1일 때, 정상적으로 연산이 수행되어 s 배열과 cout에 의도한 값이 저장됨을 확인할 수 있다.

1. BCD Adder의 결과 및 Simulation 과정에 대해서 설명하시오.

BCD code의 덧셈 연산을 하기 위해, binary 형태의 결과와 BCD code 형태의 계산 결과를 비교하는 진리표를 작성해 의 Boolean function을 구했다.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| decimal | binary | | | | | BCD | | | | |
|  | c' | s3' | s2' | s1' | s0' | c | s3 | s2 | s1 | s0 |
| 10 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 11 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 12 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 13 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 14 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 15 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 16 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 17 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 18 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 19 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

구한 Boolean function을 포함해 BCD adder를 미리 구현한 4-bit RCA로 표현하면 다음 회로도와 같다. 이 때, 두 번째 RCA에서는 0110, 10진수 6을 더해 값이 올바른 BCD code로 출력되도록 해야 한다.

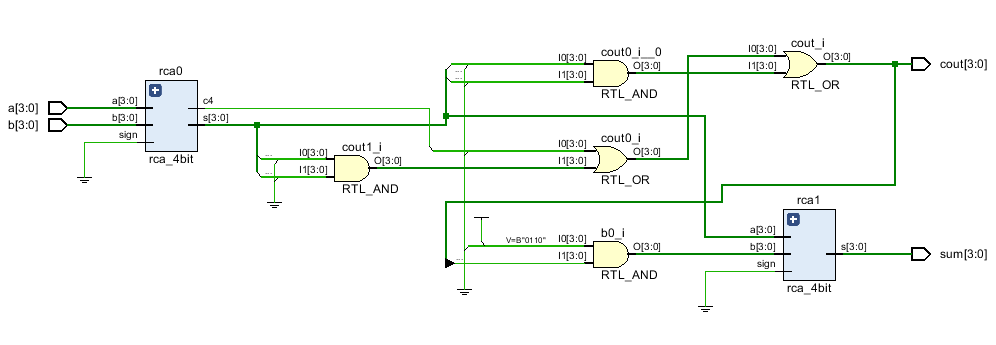


위의 회로도를 Verilog code로 구체화한 결과는 다음과 같다.

|  |
| --- |
| `timescale 1ns / 1ps  module bcd\_adder(a, b, sum, cout);  input [3:0] a, b;  output [3:0] sum, cout;  wire carryOut;  wire [3:0] binarySum;  rca\_4bit rca0(a, b, 0, binarySum, carryOut);  assign cout = carryOut|(binarySum[3]&binarySum[2])|(binarySum[3]&binarySum[1]);  rca\_4bit rca1(binarySum, 4'b0110 & {4{cout[0]}}, 0, sum);  endmodule |

이 때, carryOut은 최종적인 출력인 을 계산하기 위해 1차적으로 계산한 4bit adder의 결과이다. 두 번째 4-bit binary adder를 시행할 때 binary 값 0110을 의 값을 4bit로 확장한 수와 AND 연산을 시켜 값이 1일 때에만 추가적인 연산이 의미가 있도록 구성했다.

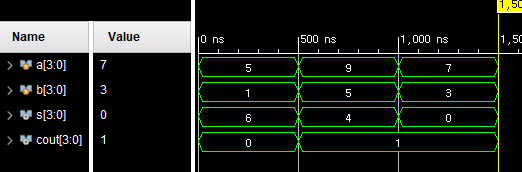
위처럼 구현한 Verilog code의 RTL Schematic 회로도는 다음과 같다.



작성한 design이 정상적으로 동작하는지 확인하기 위해 다음과 같이 simulation code를 작성했다.

|  |
| --- |
| `timescale 1ns / 1ps  module bcdsim;  reg [3:0] a, b;  wire [3:0] s, cout;  bcd\_adder bcd(a, b, s, cout);  initial a = 4'b0101; //BCD 5  initial b = 4'b0001; //BCD 1  // 5+1 = 6  initial begin  #500  assign a=4'b1001; //BCD 9  assign b=4'b0101; //BCD 5  // 9+5 = 14  #500  assign a=4'b0111; //BCD 7  assign b=4'b0011; //BCD 3  // 7+3 = 10  #500  $finish;  end  endmodule |

5+1, 9+5, 7+3의 세 연산을 simulation하는 코드로 작성했다. 위 simulation의 결과는 다음과 같이 나왔다.



의 값이 정상적으로 5+1=6, 9+5=14, 7+3=10 이라는 결과를 보임을 확인할 수 있다. 결론적으로 구현이 잘 되었다고 볼 수 있다.

1. 결과 검토 및 논의 사항

* 4-bit binary parallel adder & subtractor를 구현할 때, simulation을 통해 확인하고자 한 경우의 수는 두 수의 합과 차, 그리고 각각의 경우 overflow/underflow가 일어났을 때 정상적으로 예상하던 결과값이 나오는가 이다. 이 목적을 중점으로 simulation의 결과를 확인하면, 의도했던 결과가 나왔음을 확인할 수 있었고, FPGA에서 실제로 해당 design을 실행시켰을 때 정상적으로 작동했다.
* BCD adder에서는 4-bit binary number에 대한 연산을 수행하는 4-bit binary parallel adder를 사용하고, 추가적으로 결과 값에 조정을 가하는 회로를 더해 결과값이 BCD code 형태로 나오도록 의도했다. Simulation의 결과가 BCD code의 시스템 상 정상적으로 나왔고, 실제 FPGA에서도 원하는 출력이 올바르게 나옴을 확인할 수 있었다.

1. 추가 이론 조사 및 작성

* 이 전의 실험들과는 다르게, 이번 실험에서는 작성한 module를 활용해 조금 더 복잡한 새로운 module을 구현하는 방식으로 실험을 진행했다. 프로그래밍 언어에서 함수를 사용해 재사용성과 programmability를 개선하는 것 처럼, Verilog에서도 별반 다르지 않게 모듈을 이용할 수 있음을 볼 수 있었다.
* Module을 작성할 때, n-bit argument를 받는 경우에는 배열 상태의 input과 output을 통째로 넘겨주는 것이 simulation 코드를 작성할 때나, 실제 FPGA의 각 스위치와 mapping을 할 때 더 직관적이고 간결하게 원하는 결과를 얻을 수 있었다.