10주차 결과보고서

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

**1.**

**4bit Binary Parallel Adder**

도표, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

* Verilog source

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module fa(  input x, y, cin,  output s, cout  );  assign s = (x^y)^cin;  assign cout = (cin&(x^y))|(x&y);  endmodule  module padd(  input[3:0] a, b,  input cin,  output[3:0] s,  output cout  );    wire c1, c2, c3;    fa fa0(a[0], b[0], cin, s[0], c1);  fa fa1(a[1], b[1], c1, s[1], c2);  fa fa2(a[2], b[2], c2, s[2], c3);  fa fa3(a[3], b[3], c3, s[3], cout);  endmodule | `timescale 1ns / 1ps  module padd\_tb;  reg [3:0] a, b;  reg cin;  wire [3:0] s;  wire cout;  padd u\_padd(  .a(a ),  .b(b ),  .cin(cin ),  .s(s ),  .cout(cout )  );  initial begin  a = 4'D0;  b = 4'D0;  cin = 1'b0;  end  always@(a or b or cin)begin  a <= #10 a+4'D1;  b <= #20 a-4'D1;  cin <= #30 ~cin;  end  endmodule |

fa 모듈은 3개의 입력(x, y, cin)과 2개의 출력(s, cout)을 가지고 있다. 이때 cin은 자리올림 입력을, cout은 자리올림 출력을 의미하며, s는 합 출력을 나타낸다. s는 (x^y)^cin를 연산한 결과이고, cout은 (cin&(x^y))|(x&y)를 연산한 결과이다. padd 모듈은 4비트 입력 a와 b를 받고, 캐리 입력 cin을 받는다. 또한 4비트 합 출력 s와 캐리 출력인 cout을 출력한다. 내부적으로 fa 모듈을 사용하여 각 비트에 대한 전가산기를 구성했으며, 비트별로 전가산기를 연결하여 전체 4bit Binary Parallel Adder를 완성했다. 이때 4비트 입력 a, b의 경우 array를 이용해 한번에 입력받고, 4비트 출력 s 또한 array를 이용해 한번에 출력하는 방식을 사용했다. 또한 wire type 변수 c1, c2, c3를 선언해 각 전가산기의 출력 캐리가 그 다음 전가산기의 캐리 입력으로 연결될 수 있도록 설계했다.

* Schematic Diagram

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드로 Schematic Diagram을 생성한 결과이다. 이를 통해 작성한 코드가 회로에 어떤 식으로 구현되는지 알 수 있다. 각 전가산기의 출력 캐리가 그 다음 전가산기의 캐리 입력으로 연결되는 것을 확인할 수 있다. 또한 이는 각 비트에 대해 별도의 전가산기가 연결되어 병렬적으로 4비트의 이진수 덧셈을 수행할 수 있다.

* Simulation 결과

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

스크린샷, 사각형이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드를 시뮬레이션 돌린 결과이다. 시간에 따라 4비트 입력 a, b와 캐리 입력 cin의 값을 바꾸면 그에 따라 4비트 합 출력 s와 캐리 출력 cout의 값이 변화하는 것을 확인할 수 있다. 이 중, cout이 0이 되는 경우와 1이 되는 경우를 좀 더 자세히 살펴보겠다.

스크린샷, 사각형이(가) 표시된 사진

자동 생성된 설명

먼저, cout이 0인 경우이다. 7과 4를 더하는 경우 a는 0111, b는 0100이 되어 둘을 더하면 결과 값 s는 1011(b)이 되고 최종적으로 캐리 출력이 발생하지 않아 cout이 0이 될 것을 예측할 수 있다. 실제로 시뮬레이션 상에도 동일한 값이 나와 구현한 회로가 올바르게 동작하고 있음을 확인할 수 있다.

스크린샷, 사각형, 다채로움이(가) 표시된 사진

자동 생성된 설명

다음으로, cout이 1이 되는 경우이다. c와 9를 더하는 경우 a는 1100, b는 1001이 되어 둘을 더하면 결과 값 s는 0101(5)이 되고 가장 왼쪽 자리에서 자리 올림이 발생해 cout이 1이 될 것을 예측할 수 있다. 이 경우에도 실제 시뮬레이션 상에 동일한 값이 나와 구현한 회로가 올바르게 동작하고 있음을 확인할 수 있다.

* 출력 예시

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| a | b | cin | s | cout |
| 0000(0) | 0000(0) | 0 | 0000(0) | 0 |
| 0001(1) | 0000(0) | 0 | 0001(1) | 0 |
| 0010(2) | 1111(f) | 0 | 0001(1) | 1 |
| 0011(3) | 0000(0) | 1 | 0100(4) | 0 |
| 0100(4) | 0001(1) | 1 | 0110(6) | 0 |
| 0101(5) | 0010(2) | 1 | 1000(8) | 0 |
| 0110(6) | 0011(3) | 0 | 1001(9) | 0 |
| 0111(7) | 0100(4) | 0 | 1011(b) | 0 |
| 1000(8) | 0101(5) | 0 | 1101(d) | 0 |
| 1001(9) | 0110(6) | 1 | 0000(0) | 1 |
| 1010(a) | 0111(7) | 1 | 0010(2) | 1 |
| 1011(b) | 1000(8) | 1 | 0100(4) | 1 |
| 1100(c) | 1001(9) | 0 | 0101(5) | 1 |
| 1101(d) | 1010(a) | 0 | 0111(7) | 1 |
| 1110(e) | 1011(b) | 0 | 1001(9) | 1 |
| 1111(f) | 1100(c) | 1 | 1100(c) | 1 |
| 0000(0) | 1101(d) | 1 | 1110(e) | 0 |
| 0001(1) | 1110(e) | 1 | 0000(0) | 1 |

입력의 모든 케이스를 다룰 순 없지만 위 시뮬레이션 캡처에 나온 모든 출력 예시들을 표로 정리한 결과이다. 각 출력 예시를 직접 계산해보면 입력 값에 대해 올바른 출력이 나오는 것을 확인할 수 있다. 이를 통해 설계한 회로가 의도한 대로 잘 동작한다는 것을 실험적으로 확인할 수 있었다.

**2.**

**4bit Binary Parallel Subtractor**

도표, 라인, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명

* Verilog source

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module fs(  input x, y, bin,  output d, bout  );  assign d = (x^y)^bin;  assign bout = ((~(x^y))&bin)|(~x&y);  endmodule  module psub(  input[3:0] a, b,  input bin,  output[3:0] d,  output bout  );    wire b1, b2, b3;    fs fs0(a[0], b[0], bin, d[0], b1);  fs fs1(a[1], b[1], b1, d[1], b2);  fs fs2(a[2], b[2], b2, d[2], b3);  fs fs3(a[3], b[3], b3, d[3], bout);  endmodule | `timescale 1ns / 1ps  module psub\_tb;  reg [3:0] a, b;  reg bin;  wire [3:0] d;  wire bout;  psub u\_psub(  .a(a ),  .b(b ),  .bin(bin ),  .d(d ),  .bout(bout )  );  initial begin  a = 4'D0;  b = 4'D0;  bin = 1'b0;  end  always@(a or b or bin)begin  a <= #10 a+4'D1;  b <= #20 a-4'D1;  bin <= #30 ~bin;  end  endmodule |

fs 모듈은 3개의 입력(x, y, bin)과 2개의 출력(d, bout)을 가지고 있다. 이때 bin은 자리빌림 입력을, bout은 자리빌림 출력을 의미하며, d는 차이 출력을 나타낸다. d는 (x^y)^bin를 연산한 결과이며, bout은 (~(x^y))&bin와 (~x&y)의 OR(논리 합) 연산 결과이다. psub 모듈은 4비트 입력 a와 b를 받고, 빌림 입력 bin을 받는다. 또한 4비트 차이 출력 d와 빌림 출력인 bout을 출력한다. 내부적으로 fs 모듈을 사용하여 각 비트에 대한 전감산기를 구성했으며, 비트별로 전감산기를 연결하여 전체 4bit Binary Parallel Subtractor를 완성했다. 이때 4비트 입력 a, b의 경우 array를 이용해 한번에 입력받고, 4비트 출력 d 또한 array를 이용해 한번에 출력하는 방식을 사용했다. 또한 wire type 변수 b1, b2, b3를 선언해 각 전감산기의 빌림 출력이 그 다음 전감산기의 빌림 입력으로 연결될 수 있도록 설계했다.

* Schematic Diagram

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드로 Schematic Diagram을 생성한 결과이다. 이를 통해 작성한 코드가 회로에 어떤 식으로 구현되는지 알 수 있다. 각 전감산기의 빌림 출력이 그 다음 전감산기의 빌림 입력으로 연결되는 것을 확인할 수 있다. 또한 이는 각 비트에 대해 별도의 전감산기가 연결되어 병렬적으로 4비트의 이진수 뺄셈을 수행할 수 있다.

* Simulation 결과

스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드를 시뮬레이션 돌린 결과이다. 시간에 따라 4비트 입력 a, b와 빌림 입력 bin의 값을 바꾸면 그에 따라 4비트 차이 출력 d와 빌림 출력 bout의 값이 변화하는 것을 확인할 수 있다. 이 중, bout이 0이 되는 경우와 1이 되는 경우를 좀 더 자세히 살펴보겠다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

먼저, bout이 0인 경우이다. 8에서 5를 빼는 경우 a는 1000, b는 0101이 되어 둘을 빼면 결과 값 d는 0011(3)이 되고 최종적으로 자리 빌림 출력이 발생하지 않아 bout이 0이 될 것을 예측할 수 있다. 실제로 시뮬레이션 상에도 동일한 값이 나와 구현한 회로가 올바르게 동작하고 있음을 확인할 수 있다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

다음으로, bout이 1이 되는 경우이다. 2에서 f를 빼는 경우 a는 0010, b는 1111이 되고 실제로 0010에서 1111을 빼면 결과 값 d는 0011(3)이 되고 가장 왼쪽 자리에서 자리 빌림 출력이 발생해 bout이 1이 될 것을 예측할 수 있다. 이 경우에도 실제 시뮬레이션 상에 동일한 값이 나와 구현한 회로가 올바르게 동작하고 있음을 확인할 수 있다.

* 출력 예시

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| a | b | bin | d | bout |
| 0000(0) | 0000(0) | 0 | 0000(0) | 0 |
| 0001(1) | 0000(0) | 0 | 0001(1) | 0 |
| 0010(2) | 1111(f) | 0 | 0011(3) | 1 |
| 0011(3) | 0000(0) | 1 | 0010(2) | 0 |
| 0100(4) | 0001(1) | 1 | 0010(2) | 0 |
| 0101(5) | 0010(2) | 1 | 0010(2) | 0 |
| 0110(6) | 0011(3) | 0 | 0011(3) | 0 |
| 0111(7) | 0100(4) | 0 | 0011(3) | 0 |
| 1000(8) | 0101(5) | 0 | 0011(3) | 0 |
| 1001(9) | 0110(6) | 1 | 0010(2) | 0 |
| 1010(a) | 0111(7) | 1 | 0010(2) | 0 |
| 1011(b) | 1000(8) | 1 | 0010(2) | 0 |
| 1100(c) | 1001(9) | 0 | 0011(3) | 0 |
| 1101(d) | 1010(a) | 0 | 0011(3) | 0 |
| 1110(e) | 1011(b) | 0 | 0011(3) | 0 |
| 1111(f) | 1100(c) | 1 | 0010(2) | 0 |
| 0000(0) | 1101(d) | 1 | 0010(2) | 1 |
| 0001(1) | 1110(e) | 1 | 0010(2) | 1 |

입력의 모든 케이스를 다룰 순 없지만 위 시뮬레이션 캡처에 나온 모든 출력 예시들을 표로 정리한 결과이다. 각 출력 예시를 직접 계산해보면 입력 값에 대해 올바른 출력이 나오는 것을 확인할 수 있다. 이를 통해 설계한 회로가 의도한 대로 잘 동작한다는 것을 실험적으로 확인할 수 있었다.

**3.**

**BCD Adder**

도표, 기술 도면, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

* Verilog source

|  |  |
| --- | --- |
| Design Source Code | Simulation Source Code |
| `timescale 1ns / 1ps  module fa(  input x, y, cin,  output s, cout  );  assign s = (x^y)^cin;  assign cout = (cin&(x^y))|(x&y);  endmodule  module padd(  input[3:0] a, b,  input cin,  output[3:0] s,  output cout  );    wire c1, c2, c3;    fa fa0(a[0], b[0], cin, s[0], c1);  fa fa1(a[1], b[1], c1, s[1], c2);  fa fa2(a[2], b[2], c2, s[2], c3);  fa fa3(a[3], b[3], c3, s[3], cout);  endmodule  module bcdadd(  input[3:0] a, b,  input cin,  output[3:0] s,  output cout  );    wire k;  wire [3:0] z, S;    padd rca0(a, b, cin, z, k);  assign cout = (z[3]&z[2])|(z[3]&z[1])|k;  assign S = {1'b0, cout, cout, 1'b0};  padd rca1(z, S, 0, s);  endmodule | `timescale 1ns / 1ps  module bcdadd\_tb;  reg [3:0] a, b;  reg cin;  wire [3:0] s;  wire cout;  bcdadd u\_bcdadd(  .a(a ),  .b(b ),  .cin(cin ),  .s(s ),  .cout(cout )  );  initial begin  a = 4'D0;  b = 4'D0;  cin = 1'b0;  end  always@(a or b or cin)begin  a <= #10 a+4'D1;  b <= #20 a-4'D1;  cin <= #30 ~cin;  end  endmodule |

bcdadd 모듈은 두 개의 4비트 입력 a와 b를 받고, 캐리 입력 cin을 받는다. 결과로는 4비트 합 s와 캐리 출력 cout을 출력한다. 내부적으로 앞서 구현한 padd 모듈을 사용했으며, 추가적인 회로를 더해 BCD 코드의 연산을 수행할 수 있도록 구현했다. 이 회로의 연산 과정은 다음과 같다. 먼저 rca0는 입력 a와 b를 4bit Binary Parallel Adder로 더하고, 결과를 z에 저장한다. Cout은 z[3] & z[2], z[3] & z[1], k를 OR 연산하여 캐리 출력을 구한다. S는 결과 값을 BCD 코드로 보정하기 위한 4비트 값으로 cout이 0인 경우 0000(0)이 되며, cout이 1인 경우 0110(6)의 값을 가진다. rca1은 z와 S를 4bit Binary Parallel Adder로 더하여 최종 결과 s를 출력한다. 이러한 과정을 통해 bcdadd 모듈은 4비트 BCD 덧셈을 수행한다.

* Schematic Diagram

도표, 라인, 평면도, 그래프이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드로 Schematic Diagram을 생성한 결과이다. 이를 통해 작성한 코드가 회로에 어떤 식으로 구현되는지 알 수 있다. 이는 두 개의 4bit Binary Parallel Adder 모듈 rca0과 rca1을 사용하며 결과 값을 BCD 코드로 나타내기 위해 추가적인 회로를 더해 BCD 코드의 연산을 수행한다.

* Simulation 결과

스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

스크린샷, 사각형이(가) 표시된 사진

자동 생성된 설명

위에서 작성한 Verilog 코드를 시뮬레이션 돌린 결과이다. 시간에 따라 4비트 bcd 입력 a, b와 캐리 입력 cin의 값을 바꾸면 그에 따라 4비트 bcd 합 출력 s와 캐리 출력 cout의 값이 변화하는 것을 확인할 수 있다. 이 중, 결과 값에 bcd 보정이 발생하는 경우와 발생하지 않는 경우를 좀 더 자세히 살펴보겠다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

먼저, 결과 값 보정이 발생하지 않는 경우이다. 6과 3을 더하는 경우 a는 0110, b는 0011이 되어 둘을 더하면 결과 값 s는 1001(9)이 되고 최종적으로 캐리 출력이 발생하지 않아 cout이 0이 될 것을 예측할 수 있다. 이때 s는 bcd 코드로 표현할 수 있는 범위(0~9) 내에 있기 때문에 보정이 발생하지 않고 계산한 그대로의 값을 가진다. 실제로 시뮬레이션 상에도 동일한 값이 나와 구현한 회로가 올바르게 동작하고 있음을 확인할 수 있다.

스크린샷, 사각형이(가) 표시된 사진

자동 생성된 설명

다음으로, 결과 값 보정이 발생하는 경우이다. 8과 5를 더하는 경우 a는 1000, b는 0101이 되어 첫 번째 4비트 병렬 가산기의 결과 값은 1101(d)이 되는데, 이는 bcd 코드로 나타낼 수 없는 범위이다. 따라서 cout이 1이 될 것을 예측할 수 있으며 0110(6)을 더해주어 결과 값을 보정해준다. 이 경우에도 실제 시뮬레이션 상에 동일한 값이 나와 구현한 회로가 올바르게 동작하고 있음을 확인할 수 있다.

* 출력 예시

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| a | b | cin | s | cout |
| 0000(0) | 0000(0) | 0 | 0000(0) | 0 |
| 0001(1) | 0000(0) | 0 | 0001(1) | 0 |
| 0010(2) | 1111(f) | 0 | 0111(7) | 1 |
| 0011(3) | 0000(0) | 1 | 0100(4) | 0 |
| 0100(4) | 0001(1) | 1 | 0110(6) | 0 |
| 0101(5) | 0010(2) | 1 | 1000(8) | 0 |
| 0110(6) | 0011(3) | 0 | 1001(9) | 0 |
| 0111(7) | 0100(4) | 0 | 0001(1) | 1 |
| 1000(8) | 0101(5) | 0 | 0011(3) | 1 |
| 1001(9) | 0110(6) | 1 | 0110(6) | 1 |
| ~~1010(a)~~ | ~~0111(7)~~ | ~~1~~ | ~~1000(8)~~ | ~~1~~ |
| ~~1011(b)~~ | ~~1000(8)~~ | ~~1~~ | ~~1010(a)~~ | ~~1~~ |
| ~~1100(c)~~ | ~~1001(9)~~ | ~~0~~ | ~~1011(b)~~ | ~~1~~ |
| ~~1101(d)~~ | ~~1010(a)~~ | ~~0~~ | ~~1101(d)~~ | ~~1~~ |
| ~~1110(e)~~ | ~~1011(b)~~ | ~~0~~ | ~~1111(f)~~ | ~~1~~ |
| ~~1111(f)~~ | ~~1100(c)~~ | ~~1~~ | ~~0010(2)~~ | ~~1~~ |
| ~~0000(0)~~ | ~~1101(d)~~ | ~~1~~ | ~~0100(4)~~ | ~~1~~ |
| ~~0001(1)~~ | ~~1110(e)~~ | ~~1~~ | ~~0110(6)~~ | ~~1~~ |

입력의 모든 케이스를 다룰 순 없지만 위 시뮬레이션 캡처에 나온 모든 출력 예시들을 표로 정리한 결과이다. a, b에 bcd 코드 범위에 들어가지 않는 수가 입력되는 경우는 고려하지 않는다. 각 출력 예시를 직접 계산해보면 입력 값에 대해 올바른 출력이 나오는 것을 확인할 수 있다. 이를 통해 설계한 회로가 의도한 대로 잘 동작한다는 것을 실험적으로 확인할 수 있었다.

**4.** 결과 검토 및 논의 사항.

이번 실험에서는 4-bit binary parallel adder와 4-bit binary parallel subtractor의 개념을 이해하고, Verilog 코드로 이를 직접 구현했다. 시뮬레이션을 통해 두 수의 합과 차, 캐리와 빌림 수가 예상한 대로 출력되는지 확인한 결과 회로가 의도한 대로 정상적으로 작동함을 확인할 수 있었다. 또한 FPGA 실습에서도 해당 디자인이 정상 작동함을 확인했다.

다음으로 BCD Adder의 개념을 이해하고 직접 구현해보았다. 이때 앞에서 구현한 4-bit binary parallel adder를 이용하고, 추가적인 회로를 더해 BCD 코드의 연산을 수행할 수 있도록 구현했다. 위와 동일하게 시뮬레이션을 통해 예상한 BCD 코드 결과 값이 나오는지 확인하고, FPGA에서도 원하는 출력이 제대로 표시되는지를 확인한 결과 회로가 의도한 대로 작동함을 확인할 수 있었다.

이전 실험들과 달리 변수들을 array로 선언하고, wire type의 변수를 사용하는 등 새로운 type의 변수를 사용하여 코드를 작성하는 경험을 할 수 있었다. 또한 이전 실험에서 작성한 Full adder, Full subtractor module를 활용해 4-bit binary parallel adder와 4-bit binary parallel subtractor module을 구현하는 방식으로 코드를 작성했다. 이를 통해 다른 프로그래밍 언어에서 함수나 템플릿을 사용해 코드의 재사용성을 향상시키는 것처럼, Verilog에서도 유사한 방식으로 모듈을 이용할 수 있음을 알게 되었다.

**5.**

**Verilog 코드 작성 시, wire type으로 변수를 선언하면 입력 또는 출력으로 지정하지 않고 변수를 사용할 수 있다. wire**는 Verilog 프로그래밍 언어에서 사용되는 선언 키워드 중 하나이다. wire는 논리 회로에서 사용되며, 주로 논리 게이트의 출력을 나타내거나 모듈 간에 연결되는 데 활용된다. 이는 연속 할당(assign)과 함께 사용된다. 연속 할당은 논리식을 사용하여 변수에 값을 할당하는 데 사용되며, 일반적으로 조합 논리 회로에서 많이 활용된다. 또한 wire는 연결된 항목 간에 자동으로 연결되는 특징이 있다. 즉, 여러 모듈에서 wire를 공유할 경우, 이 변수는 자동으로 연결되어 값을 공유한다. reg와 달리 wire는 순서대로 순간적인 값을 나타낸다. reg는 시간에 따라 지속적인 값을 나타내므로, 시뮬레이션에서는 초기값을 갖는 반면, wire는 초기값이 없다.