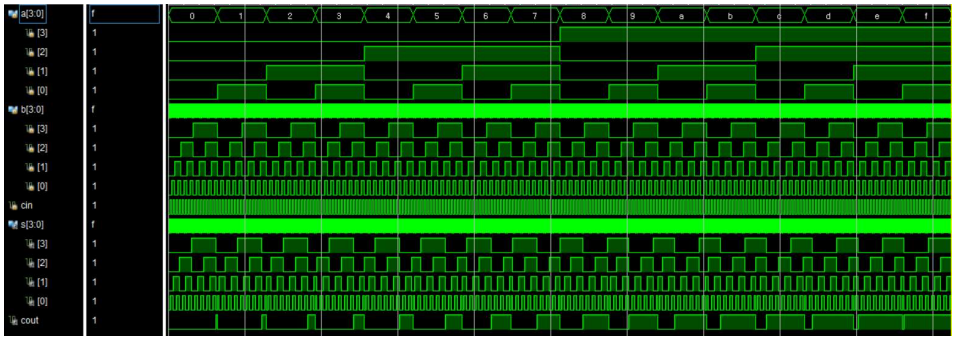
10주차 결과보고서

전공: 경영학과 학년: 4학년 학번: 20190808 이름: 방지혁

1.

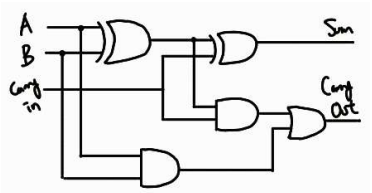


```
`timescale 1ns / 1ps

module sum(
input a, b, cin,
output sum, cout
);

assign sum = a^b^cin;
assign cout = a&b|((a^b)&cin);
endmodule

module parallelAdder(
input [3:0] a, b,
input cin,
output [3:0] s,
output cout
);
wire [3:1] c;

sum module00(
   .a(a[0]),
   .b(b[0]),
   .cin(cin),

   .sum(s[0]),
   .cout(c[1])
);

sum module01(
   .a(a[1]),
   .b(b[1]),
   .cin(c[1]),

   .sum(s[1]),
   .cout(c[2])
);

sum module02(
   .a(a[2]),
   .b(b[2]),
   .cin(c[2]),

   .sum(s[2]),
   .cout(c[3])
);

sum module03(
   .a(a[3]),
   .b(b[3]),
   .cin(c[3]),

   .sum(s[3]),
   .cout(cout)
);
endmodule
```

```
`timescale 1ns / 1ps

module parallelAdder_tb;
reg [3:0] a;
reg [3:0] b;
reg cin;
wire [3:0] s;
wire cout;

parallelAdder test (
.a(a),
.b(b),
.cin(cin),
.s(s),
.cout(cout)
);

initial begin
a = 4'b0000;
b = 4'b0000;
cin = 1'b0;
#512
$finish;
end

always @(cin or a or b) begin
a[3] <= #256 ~a[3];
a[2] <= #128 ~a[2];
a[1] <= #64 ~a[1];
a[0] <= #32 ~a[0];
b[3] <= #16 ~b[3];
b[2] <= #8 ~b[2];
b[1] <= #4 ~b[1];
b[0] <= #2 ~b[0];
cin <= #1 ~cin;
end

endmodule
```

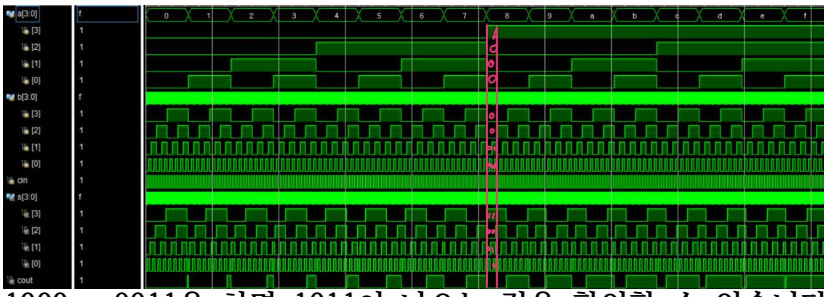
설명은 다음 장에

우선 전가산기 모듈인 sum을 먼저 구현합니다.



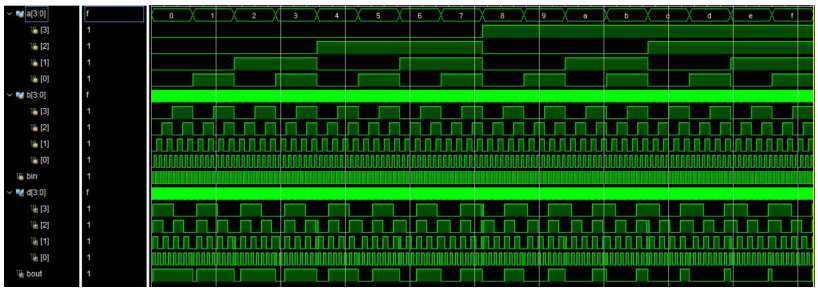
우리가 이전 실험에서 했던 바에서 알 수 있듯이 sum인 합은 a ^ b ^ cin으로 계산할 수 있습니다. 여기서 cin은 carry in입니다. carry out은 a & b | ((a ^ b) & cin)입니다. 해당 모듈을 이용해 전가산기 4개를 구현합니다. 전가산기의 input, output, wire은 배열을 이용해 선언합니다. 이 과정에서 한 가산기에서 발생한 carry out이 다음 bit 자릿수의 전가산기의 carry in으로 들어가야 하기 때문에 wire 문법을 활용하여 연결선을 지정해줍니다. 그렇게 총 4번의 전가산을 통해 구현할 수 있었습니다.

예시를 들자면,



1000 + 0011을 하면 1011이 나오는 것을 확인할 수 있습니다.

2.

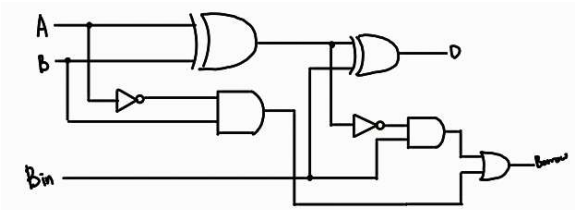


다음 장에 이어서

```
`timescale 1ns / 1ps

module sub(
input a, b, bin,
output diff, bout
);
assign diff = a^b^bin;
assign bout = b&(~a)|(~(a^b))&bin;
endmodule
module parallelSubtractor(
input [3:0] a, b,
input bin,
output [3:0] d,
output bout
);
wire [3:1] bb;
sub sub00 (
.a(a[0]),
.b(b[0]),
.bin(bin),
.diff(d[0]),
.bout(bb[1])
);
sub sub01 (
.a(a[1]),
.b(b[1]),
.bin(bb[1]),
.diff(d[1]),
.bout(bb[2])
);
sub sub02 (
.a(a[2]),
.b(b[2]),
.bin(bb[2]),
.diff(d[2]),
.bout(bb[3])
);
sub sub03 (
.a(a[3]),
.b(b[3]),
.bin(bb[3]),
.diff(d[3]),
.bout(bout)
);
endmodule
```

```
`timescale 1ns / 1ps

module parallelSubtractor_tb;
reg [3:0] a, b;
reg bin;
wire [3:0] d;
wire bout;

parallelSubtractor test (
.a(a),
.b(b),
.bin(bin),
.d(d),
.bout(bout)
);

initial begin
a = 4'b0000;
b = 4'b0000;
bin = 1'b0;
#512
$finish;
end

always @(bin or a or b) begin
a[3] <= #256 ~a[3];
a[2] <= #128 ~a[2];
a[1] <= #64 ~a[1];
a[0] <= #32 ~a[0];
b[3] <= #16 ~b[3];
b[2] <= #8 ~b[2];
b[1] <= #4 ~b[1];
b[0] <= #2 ~b[0];
bin <= #1 ~bin;
end

endmodule
```

우선 전감산기 모듈인 sub를 먼저 구현합니다.

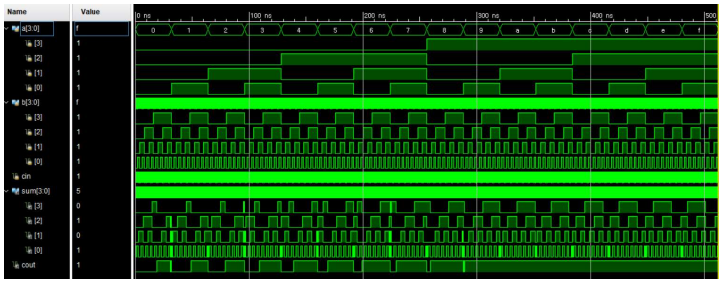


우리가 이전 실험에서 했던 바에서 알 수 있듯이 diff인 차는 a ^ b ^ bin으로 계산할 수 있습니다. bout인 빌림 수는 (~(a^b))&bin|b&(~a)입니다. 이렇게 전감산기 모듈을 구현합니다. 해당 모듈을 이용해 전감산기 4개를 구현합니다. 전감산기의 input, output, wire은 배열을 이용해 선언합니다. 이 과정에서 한 감산기에서 발생한 bout이 다음 bit 자릿수의 전가산기의 borrow in으로 들어가야 하기 때문에 wire 문법을 활용하여 연결선을 지정해줍니다. 그렇게 총 4번의 전감산을 통해 구현할 수 있었습니다.

예시를 들자면,

0111 - 0011을 하면 0100이 나오는 것을 확인할 수 있습니다.

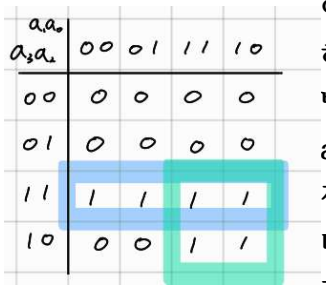
3.



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

module bcdAdder(
input [3:0] a, b,
input cin,
output [3:0] sum,
output cout);
wire [4:1]c;
wire [4:1]cc;
wire [3:0]s;
sum sum01(
.a(a[0]),
.b(b[0]),
.cin(cin),
.sum(s[0]),
.cout(c[1]));
sum sum02(
.a(a[1]),
.b(b[1]),
.cin(c[1]),
.sum(s[1]),
.cout(c[2]));
sum sum03(
.a(a[2]),
.b(b[2]),
.cin(c[2]),
.sum(s[2]),
.cout(c[3]));
sum sum04(
.a(a[3]),
.b(b[3]),
.cin(c[3]),
.sum(s[3]),
.cout(c[4]));
assign cout = c[4]|(s[3]&s[2])|(s[3]&s[1]);
sum sum05(
.a(s[0]),
.b(0),
.cin(0),
.sum(sum[0]),
.cout(cc[1]));
sum sum06(
.a(s[1]),
.b(cout),
.cin(cc[1]),
.sum(sum[1]),
.cout(cc[2]));
sum sum07(
.a(s[2]),
.b(cout),
.cin(cc[2]),
.sum(sum[2]),
.cout(cc[3]));
sum sum08(
.a(s[3]),
.b(0),
.cin(cc[3]),
.sum(sum[3]),
.cout(cc[4]));
endmodule
```

```
`timescale 1ns / 1ps
module bcdAdder_tb;
reg [3:0] a, b;
reg cin;
wire [3:0]sum;
wire cout;

bcdAdder test(
.a(a),
.b(b),
.cin(cin),
.sum(sum),
.cout(cout)
);
initial begin
a = 4'b0000;
b = 4'b0000;
cin = 1'b0;
#512
$finish;
end
always@(cin or a or b) begin
a[3] <= #256 ~ a[3];
a[2] <= #128 ~a[2];
a[1] <= #64 ~a[1];
a[0] <= #32 ~a[0];
b[3] <= #16 ~b[3];
b[2] <= #8 ~b[2];
b[1] <= #4 ~b[1];
b[0] <= #2 ~b[0];
cin <= #1 ~cin;
end
endmodule
```

BCD Adder는 4bit 자리의 두 수를 더하여 BCD로 나타내는 Adder입니다. 단, 0000부터 1001까지의 범위가 아닌 그 이상일시 carry가 발생합니다. 이 또한 4 bit adder에서 했던 것과 같은 방식으로 전가산기 모듈을 정의합니다. 전가산기 모듈 4개를 이용해 더한 다음, 최종 cout은 계산해줘야 합니다.



a_3a_2 \ a_1a_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

이 때 cout = c4|(s2&s3)|(s3&s1)입니다. 마지막 캐리가 발생하는 경우 뿐만 아니라 1001을 초과할 경우도 최종 carry가 발생하기 때문입니다. 이렇게 해당 출력을 두 번째 4bit adder의 b의 2번째, 3번째 자리에 넣습니다. a에는 기존의 전가산기 결과를 넣습니다. cout이 1인 경우 결과에 0110이 더해지고, 반대로 0인 경우 결과에 0000이 더해집니다. 그렇게 6이 더해지는 것으로 해석할 수도 있습니다.

4.

4-bit Binary Parallel 가산기, 감산기와 BCD Adder의 개념을 숙지한 후 verilog로 코드를 짜고 시뮬레이션을 돌렸습니다. 또한, FPGA 보드에 올려 구현이 잘되었는지 확인을 했습니다. 해당 실험을 하면서 wire 문법을 이용해 모듈간의 연결선을 지정한는 법을 숙지하였으며, verilog에서 배열을 사용하여 변수를 선언하는 법, 모듈의 선언과 사용법을 숙지했습니다. 단순한 하나의 게이트 구현이 아닌 parallel 가산기 같은 경우, 코드가 복잡해지면서 오류가 발생할 수 있기 때문에 모듈의 사용이 필요하다는 것을 절실하게 느꼈습니다. 또한, BCD adder를 설계하면서 이를 7 segment display에 출력한다면 시각적으로 더 이해하기 쉬울 것이라고 느꼈습니다.

5.

Wire

input 또는 output으로 지정하여 결과를 꼭 확인해야 할 필요가 없는 경우 사용하게 됩니다. 이번 보고서에서 나온 모든 회로 상에서 module을 사용하였는데, 이 모듈간의 입출력을 연결하는 연결선을 구현하기 위해서 design source 코드에서 wire 변수를 선언하면 됩니다.

부동소수점 연산

BCD나 일반 binary연산은 정수 표현에만 주로 사용됩니다. 부동 소수점 수는 ((-1)^부호) * 가수 * 2^지수로 표현될 수 있는데 여기서 가수는 유효 숫자 부분을 뜻하고, 지수는 숫자의 크기를 결정합니다. 2가지 방식이 존재하는데 단정밀도 방식과 배정밀도 방식입니다. 32비트(단정밀도)방식은 부호(1비트), 지수(8비트), 가수(23비트)로 구성되며, 64비트(배정밀도)방식은 부호(1비트), 지수(11비트), 가수(52비트)로 구성됩니다. 금융산업, 게임 그래픽스 등 여러 방면에 사용될 수 있습니다.