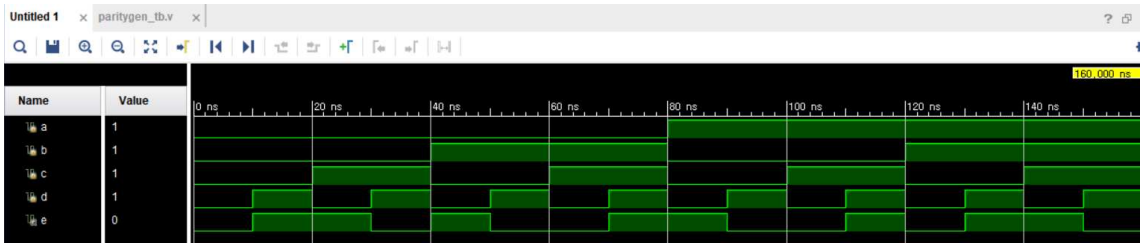
7주차 결과보고서

전공: 경영학과 학년: 4학년 학번: 20190808 이름: 방지혁

1.

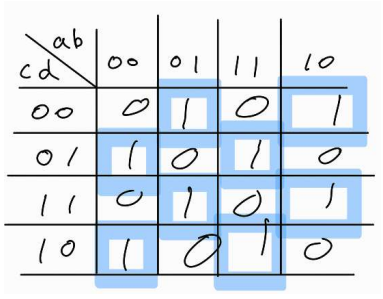
<even parity bit generator>



Input a	Input b	Input c	Input d	Output e
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

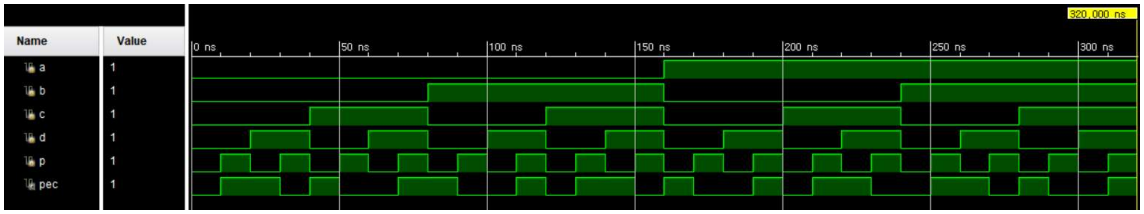
```
`timescale 1ns / 1ps

module paritygen(
  input a,b,c,d,
  output e
);

assign e = a^b^c^d;

endmodule
```

```
`timescale 1ns / 1ps

module paritygen_tb;
reg a , b, c, d;
wire e;

paritygen test(
  .a(a),
  .b(b),
  .c(c),
  .d(d),

  .e(e)
);

initial begin
a = 1'b0;
b = 1'b0;
c = 1'b0;
d = 1'b0;
#160
$finish;
end

always@(a or b or c or d) begin
a <= #80 ~a;
b <= #40 ~b;
c <= #20 ~c;
d <= #10 ~d;
end

endmodule
```



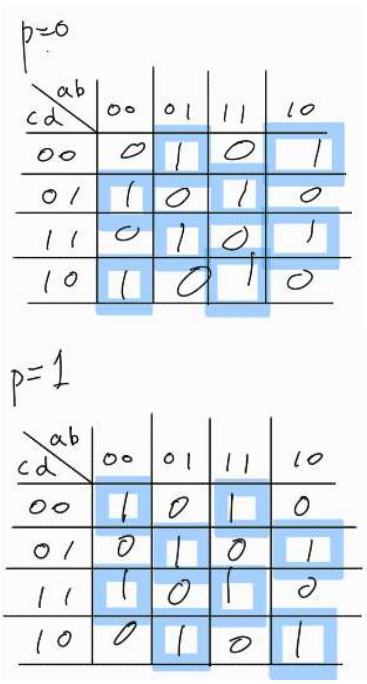
상단의 사진, 표 및 그림들을 even parity bit generator에 대한 것들입니다.
카르노 맵을 통해 even parity bit에 대한 식을 구할 수 있는데,
e = a'*b'*c'*d + a'*b'*c*d' + a'*b*c'*d' + a'*b*c*d + a*b*c'*d + a*b*c*d' + a*b'*c'*d' + a*b'*c*d = a^b^c^d
output은 input 중 1의 개수가 홀수 개일 경우 parity bit가 1이 되어야 하고, 짝수 개일 경우 parity bit는 0이 되어야 하기 때문에 여러 개의 input들에 xor한 것으로 계산할 수 있습니다.

<even parity bit checker>



Input a	Input b	Input c	Input d	Input p	Output pec
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	0
1	1	1	0	0	1
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	1

```
`timescale 1ns / 1ps

module paritycheck(
  input a, b, c, d, p,
  output pec
);

assign pec = a^b^c^d^p;

endmodule
```



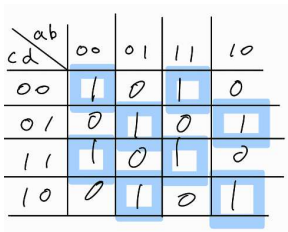

```
`timescale 1ns / 1ps

module paritycheck_tb;
reg a, b, c, d, p;
wire pec;

paritycheck test(
  .a(a),
  .b(b),
  .c(c),
  .d(d),
  .p(p),

  .pec(pec)
);

initial begin
a = 1'b0;
b = 1'b0;
c = 1'b0;
d = 1'b0;
p = 1'b0;
#320
$finish;
end

always@(a or b or c or d or p) begin
a <= #160 ~a;
b <= #80 ~b;
c <= #40 ~c;
d <= #20 ~d;
p <= #10 ~p;
end
endmodule
```

Output인 pec는 a^b^c^d^p로 나타낼 수 있습니다. 이는 a, b, c, d. p 중 1의 개수가 홀수인 경우 output이 1이 나오므로 이는 오류가 발생했다는 것을 표시합니다. 반면, 짝수인 경우 output은 0이 나오고 오류가 발생하지 않았음을 표시합니다.

2.

<odd parity bit generator>

Input a	Input b	Input c	Input d	Output e
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1




```
`timescale 1ns / 1ps

module paritygen(
  input a, b, c, d,
  output e
);

assign e = ~(a^b^c^d);

endmodule
```

```
`timescale 1ns / 1ps

module paritygen_tb;
reg a , b, c, d;
wire e;

paritygen test(
  .a(a),
  .b(b),
  .c(c),
  .d(d),

  .e(e)
);

initial begin
a = 1'b0;
b = 1'b0;
c = 1'b0;
d = 1'b0;
#160
$finish;
end

always@(a or b or c or d) begin
a <= #80 ~a;
b <= #40 ~b;
c <= #20 ~c;
d <= #10 ~d;
end

endmodule
```

상단의 사진, 표 및 그림들을 odd parity bit generator에 대한 것들입니다.

카르노 맵을 통해 even parity bit에 대한 식을 구할 수 있는데,

e = a'*b'*c'*d' + a'*b'*c*d + a'*b*c'*d + a'*b*c*d' + a*b*c'*d' + a*b*c*d + a*b'*c'*d + a*b'*c*d' = ~(a^b^c^d)

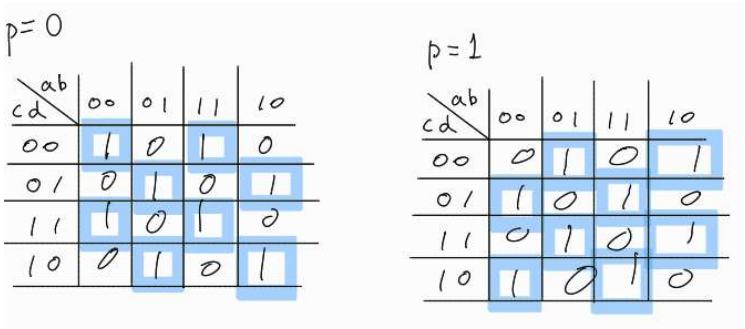
output은 input 중 1의 개수가 짝수 개일 경우 총 1의 개수가 홀수 개가 되어야 하기 때문에 parity bit가 1이 되어야 하고, 반면, 홀수 개일 경우 parity bit는 0이 되어야 하기 때문에 여러 개의 input들에 xor한 것에 부정을 취해준 것으로 계산할 수 있습니다.

<odd parity bit checker>


```v
`timescale 1ns / 1ps

module paritycheck(
  input a, b, c, d, p,
  output pec
);

assign pec = ~(a^b^c^d^p);

endmodule
```

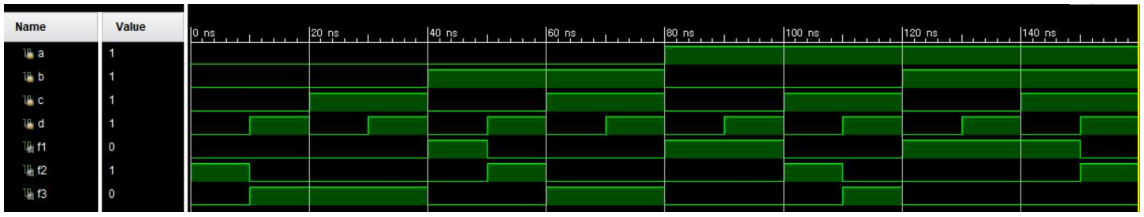
```v
`timescale 1ns / 1ps

module paritycheck_tb;
reg a, b, c, d, p;
wire pec;

paritycheck test(
  .a(a),
  .b(b),
  .c(c),
  .d(d),
  .p(p),

  .pec(pec)
);

initial begin
a = 1'b0;
b = 1'b0;
c = 1'b0;
d = 1'b0;
p = 1'b0;
#320
$finish;
end

always@(a or b or c or d or p) begin
a <= #160 ~a;
b <= #80 ~b;
c <= #40 ~c;
d <= #20 ~d;
p <= #10 ~p;
end
endmodule
```

Input a	Input b	Input c	Input d	Input p	Output PEC
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	0	1	1
1	0	0	1	0	1
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	0

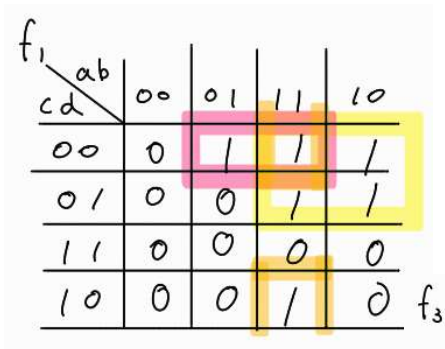


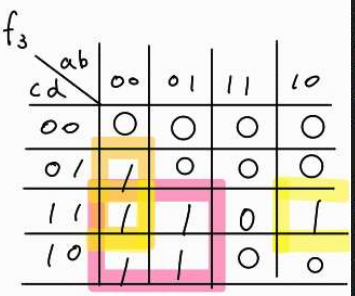
상단의 사진, 표 및 그림들을 odd parity bit checker에 대한 것들입니다.
카르노 맵을 통해 odd parity bit checker에 대한 식을 구할 수 있는데,
Output인 pec는 ~(a^b^c^d^p)로 나타낼 수 있습니다. 이는 a, b, c, d. p 중 1의 개수가 짝수인 경우 output이 1이 나오므로 이는 오류가 발생했다는 것을 표시합니다. 반면, 홀수인 경우 output은 0이 나오고 오류가 발생하지 않았음을 표시합니다.
즉, 여러 개의 input들에 xor한 것에 부정을 취해준 것으로 계산할 수 있습니다.

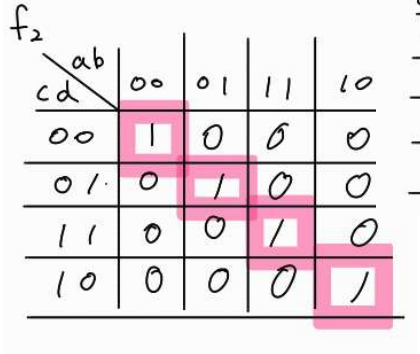
3 .




```
`timescale 1ns / 1ps

module twobitcomparator(
  input a, b, c, d,
  output f1, f2, f3
);

assign f1 = (a&(~c))|(b&(~c)&(~d))|(a&b&(~d)); // A > B
assign f2 = (~a)&(~b)&(~c)&(~d)|a&b&c&d|(~a)&b&(~c)&d|a&(~b)&c&(~d); // A = B
assign f3 = (~a)&c|(~a)&(~b)&d|(~b)&c&d; // A < B
endmodule
```








```
`timescale 1ns / 1ps

module twobitcomparator_tb;
reg a,b,c,d;
wire f1,f2,f3;

twobitcomparator test(
.a(a),
.b(b),
.c(c),
.d(d),

.f1(f1),
.f2(f2),
.f3(f3)
);


initial begin
a = 1'b0;
b = 1'b0;
c = 1'b0;
d = 1'b0;
#160
$finish;
end

always@(a or b or c or d) begin
a <= #80 ~a;
b <= #40 ~b;
c <= #20 ~c;
d <= #10 ~d;
end

endmodule
```

Input a	Input b	Input c	Input d	Output f1	Output f2	Output f3
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

만약, input ab가 input cd보다 크다면 (A>B), f1은 1을 출력합니다.
그 외의 경우(A <= B) 0을 출력합니다.
input ab가 input cd와 같은 값일 경우 (A=B), f2는 1을 출력합니다.
그 외의 경우(A != B) 0을 출력합니다.
input ab가 input cd보다 작은 경우 (A<B), f3은 1을 출력합니다.
그 외의 경우(A >= B) 0을 출력합니다.

상단에 첨부한 카르노맵을 통해 아래와 같은 식을 도출할 수 있었습니다.
Output f1 : (b&(~c)&(~d))|(a&(~c))|(a&b&(~d))
Output f2 : (~a)&(~b)&(~c)&(~d)|(~a)&b&(~c)&d|a&b&c&d|a&(~b)&c&(~d)
Output f3 : (~b)&c&d|(~a)&c|(~a)&(~b)&d

4.

FPGA를 통해 직접 하드웨어적으로 직접 구현해보았을 때, simulation을 통해 예상했던 결과와 똑같이 성공적으로 나왔습니다. 컴퓨터 연산에서 중요한 comparator의 실제 작동원리에 대해 직접 구현해보며 느낄 수 있었습니다. 다만, 카르노맵을 구하는 것이 시간이 많이 소요되어 굉장히 힘들었기에 다른 방식의 최적화 방법에 대해 갈구하게 되었습니다.

5.

LFSR

난수 생성, 데이터 암호화등에 사용되는 시프트 레지스터의 한 종류입니다. 초기 비트 값인 seed를 XOR 연산을 통해 다음 값들을 생성합니다. 각 CLK마다 비트가 Shift되고, 즉 값은 유한하기 때문에 일정 주기로 반복이 됩니다. 그렇기에 완전한 난수가 아니지만, 함수 선택을 잘한다면 랜덤하게 보이는 수열을 생성할 수 있습니다. 이처럼 주기적인 비트패턴을 만들어낼 수 있기 때문에 CRC에 사용되는 것과 같이 오류 검출 코드를 만들 수 있습니다.

CRC(Cyclic Redundancy Checking)

NIC(network interface controller)에 미리 정의된 divisor 정보가 존재합니다. dataword 뒷자리에 divisor의 '자릿수 - 1'만큼 0을 붙여 xor 연산합니다. 이 나머지를 dataword 뒷자리에 붙여 보냅니다. 수신자가 미리 약속된 동일한 divisor로 연산을 하였을 때, 나머지가 없다면 오류가 없다는 것입니다. 컴퓨터 연산상 굉장히 빠르고 간편하다는 장점이 존재합니다.