Double digit BCD calculator Assignment Report

컴퓨터학과 2018320205 신대성

## Project Summary

### 환경

OS: Windows 10

TOOL: Quartus II 9.1sp2, FPGA (Cyclone IV E – EP4CE115F29C7)

### 프로젝트 개괄

↑ 전체적인 모듈

이 베릴로그 코드는 두 BCD input과 operator를 받아 그 합과 차 계산을 Calculate 모듈에서 수행합니다.

합은 두 BCD 값을 그대로 풀 애더(full adder)에 넣어서 수행합니다. 이때, 풀애더 모듈에서는 4bit끼리의 덧셈을 수행한 뒤, 만일 out 값이 10 이상이면 4’b0110 값을 더한 값을 out으로 내놓습니다. 덧셈은 게이트 연산을 통해 구현했고, out의 값을 조절해주는 건 case문을 이용해 구현했습니다. 전체 캐리가 생기면 100을 넘어가는 값이 됩니다.

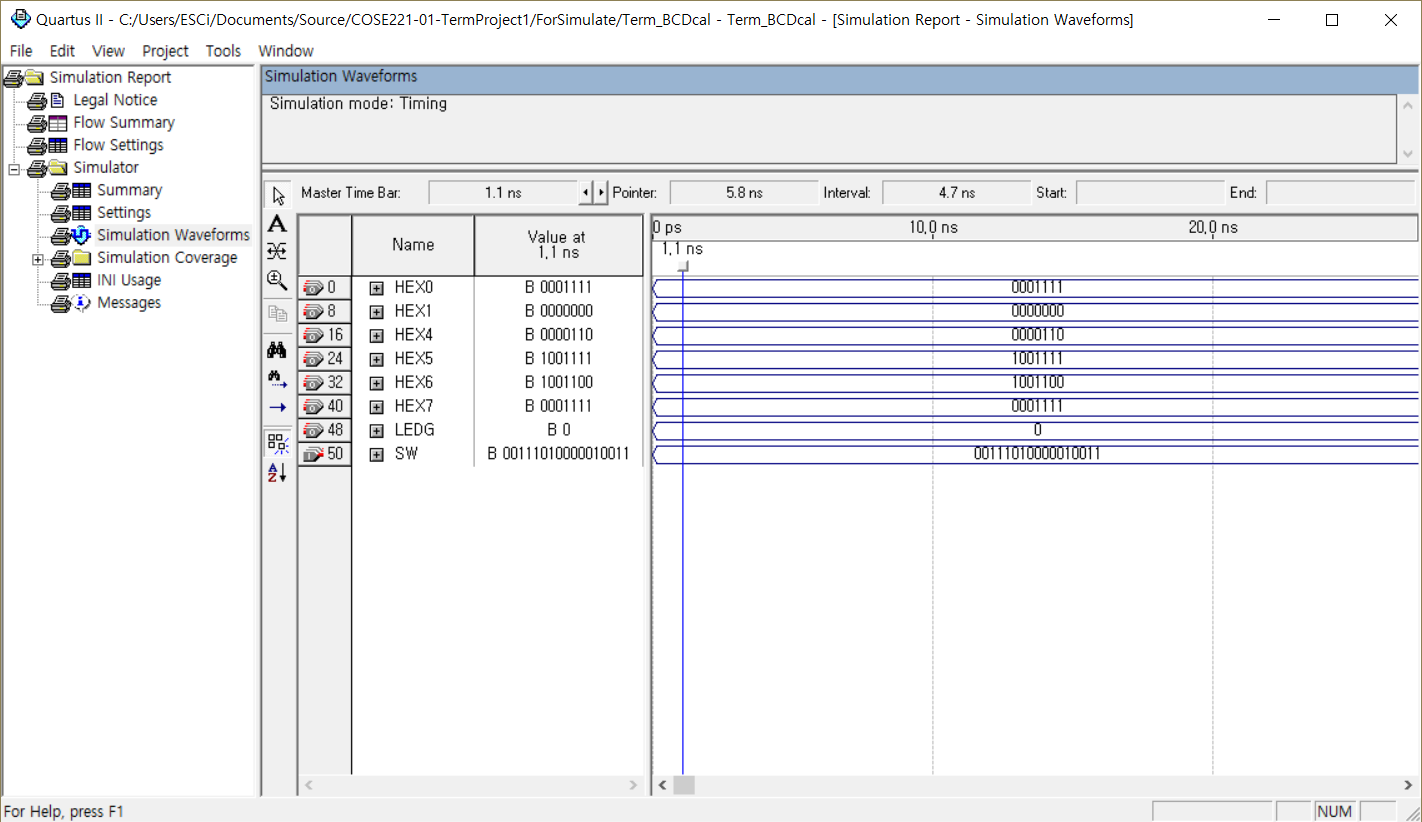
이 풀애더를 Calculator 모듈에서는 2회 수행하여 십의 자리와 일의 자리에 대한 덧셈 계산을 수행합니다. 그 계산한 BCD 값을 segment로 변환하여 출력합니다.

차는 right input을 Convert 모듈에서 10의 보수로 바꾸어 합처럼 Calculator 모듈에서 계산합니다. 사전에 미리 conv\_right를 만들어두고, SW[16]의 조작에 따라 conv\_right나 그냥 right를 선택해서 계산합니다. 이때 Calculator에 캐리가 생기면 정상적인 양수값이고, 캐리가 생기지 않으면 음수값입니다.

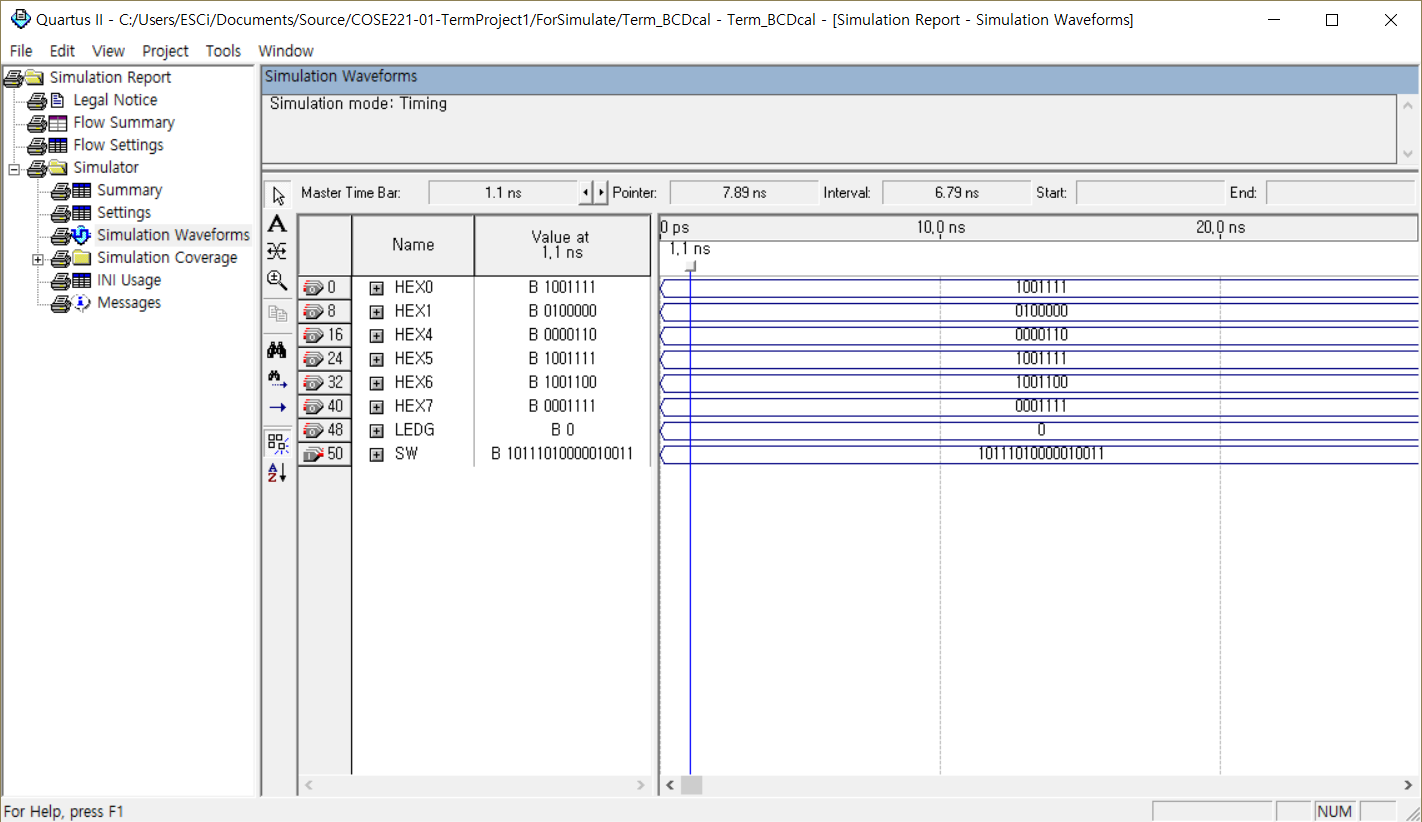
에러는 위의 Calculator 모듈에서 나온 캐리를 확인합니다. 합을 계산할때는 캐리가 발생하면 에러를 띄워야 하고, 차를 계산할때는 캐리가 발생하지 않으면 에러를 띄워야 합니다. 이를 스위치와의 XOR 연산을 통해 만들어내서, 그 값이 1일 때 에러인 것으로 계산합니다. 이 떄 결과값의 segment는 보이지 않도록 하고, LEDG[8]을 켜줍니다.

이외에도 input BCD에 문제가 있을경우는 LED는 켜주지 않지만 결과값의 segment를 보이지 않도록 합니다.

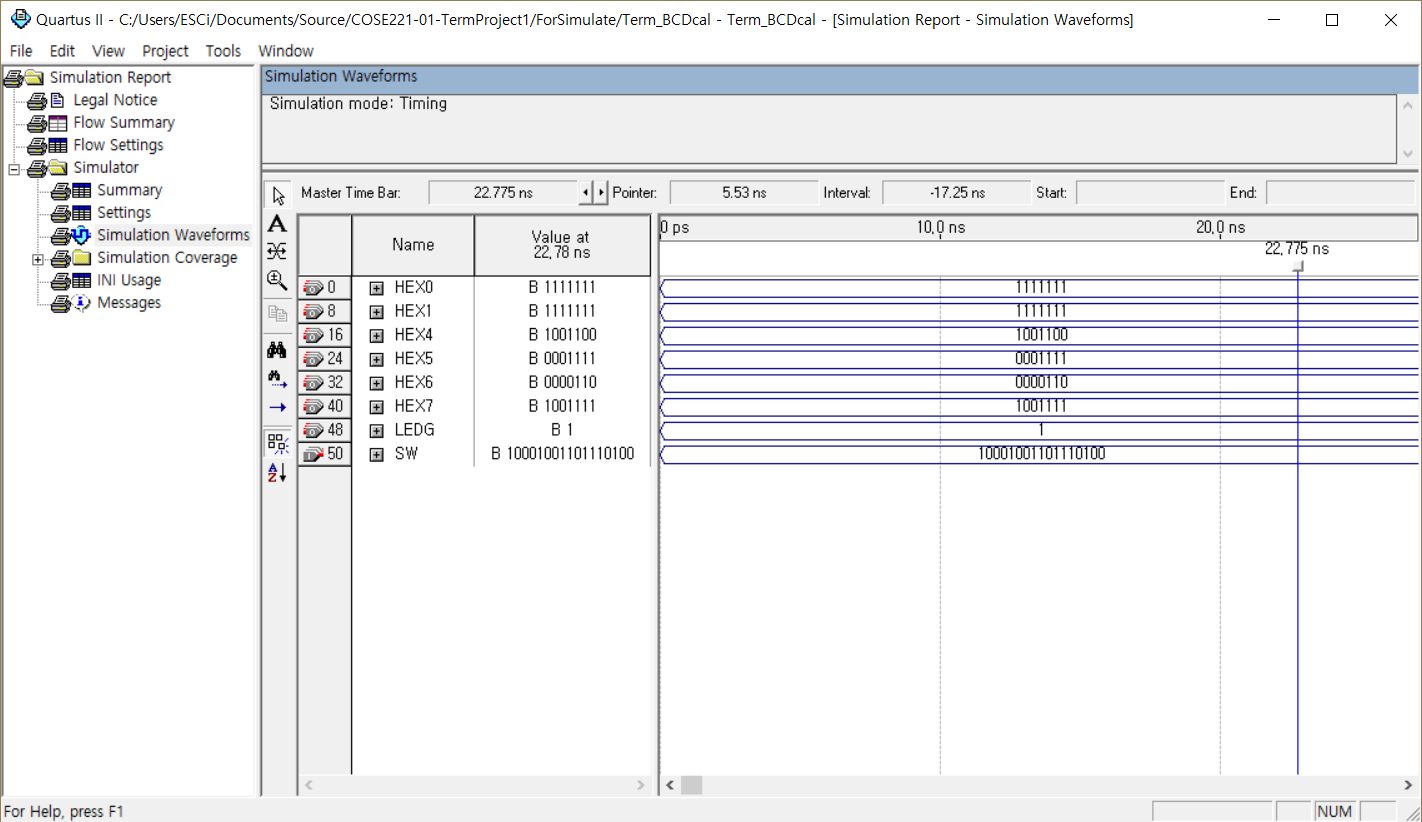
## Simulation Screenshot (4개)



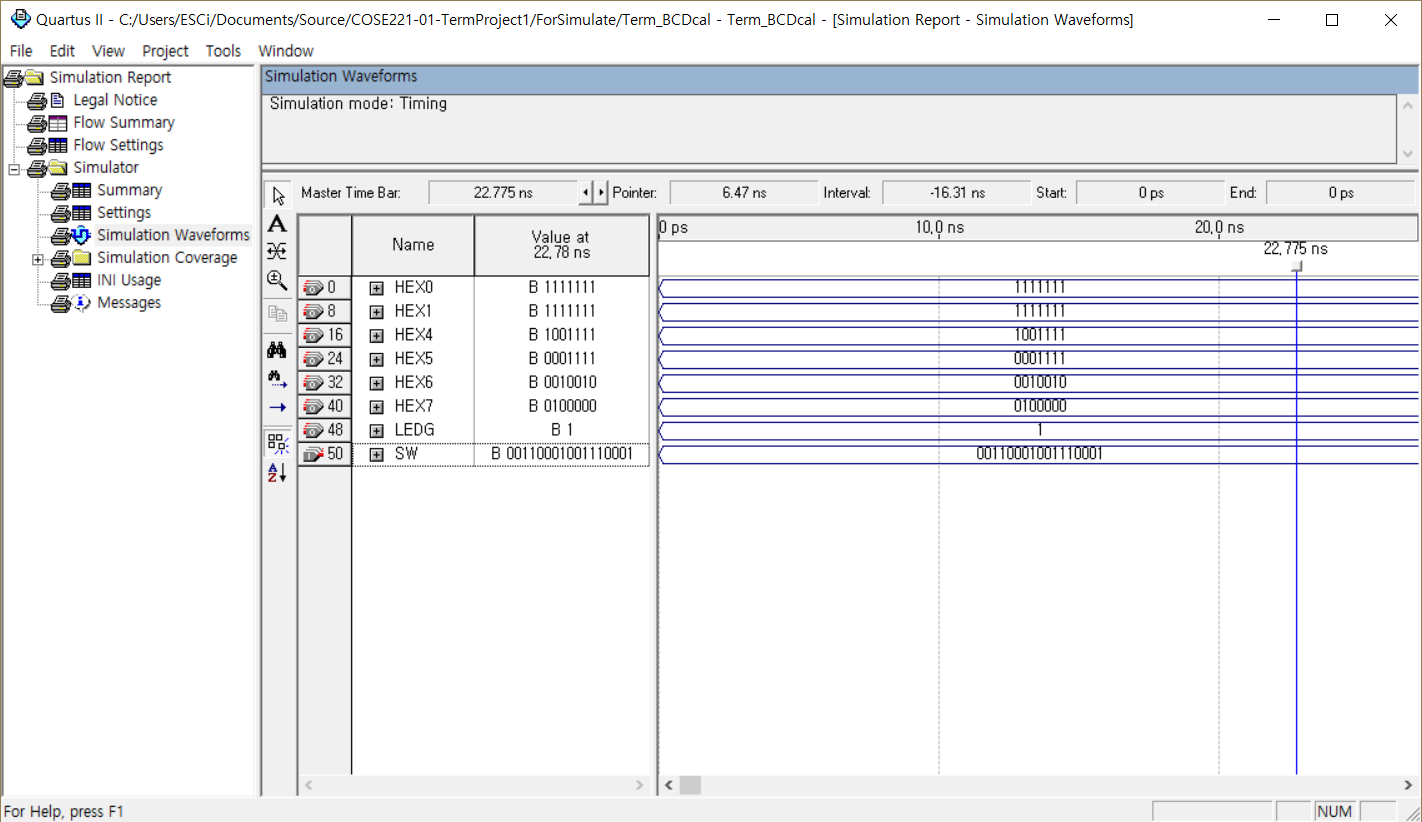
74 + 13 = 87  
input: 0(+) / 0111(7) / 0100(4) / 0001(2) / 0011(3)  
output:  
 left HEX(7, 6): 0001111(7) / 1001100(4)  
 right HEX(5, 4): 1001100(4) / 0000110(2)  
 out HEX (1, 0): 0000000(8) / 0001111(7)  
 led: 0(off)



74 – 13 = 61  
input: 1(-) / 0111(7) / 0100(4) / 0001(1) / 0011(3)  
output:  
 left HEX(7, 6): 0001111(7) / 1001100(4)  
 right HEX(5, 4): 1001111(1) / 0000110(3)  
 out HEX(1, 0): 0100000(6) / 1001111(1)  
 led: 0(off)



13 – 74 = underflow  
 input: 1(-) / 0001(1) / 0011(3) / 0111(7) / 0100(4)  
 output:  
 left HEX(7, 6): 1001111(1) / 0000110(3)  
 right HEX(5, 4): 0001111(7) / 1001100(4)  
 out HEX(1, 0): 1111111(none) / 1111111(none)  
 led: 1(on)



62 + 71 = overflow  
 input: 0(+) / 0110(6) / 0010(2) / 0111(7) / 0001(1)  
 output:  
 left HEX(7, 6): 0100000(6) / 0010010(2)  
 right HEX(5, 4): 0001111(7) / 1001111(1)  
 out HEX(1, 0): 1111111(none) / 1111111(none)  
 led: 1(on)

## HDL code

module Term\_BCDcal(SW, HEX0, HEX1, HEX4, HEX5, HEX6, HEX7, LEDG);

input [16:0] SW; // 16: mode select, 15:8: left num, 7:0: right num,

output reg [0:6] HEX0, HEX1; // output segment

output reg [0:6] HEX4, HEX5; // right segment

output reg [0:6] HEX6, HEX7; // left segment

output reg [8:8] LEDG; // error

// constant values

parameter Seg9 = 7'b000\_1100; parameter Seg8 = 7'b000\_0000; parameter Seg7 = 7'b000\_1111; parameter Seg6 = 7'b010\_0000; parameter Seg5 = 7'b010\_0100;

parameter Seg4 = 7'b100\_1100; parameter Seg3 = 7'b000\_0110; parameter Seg2 = 7'b001\_0010; parameter Seg1 = 7'b100\_1111; parameter Seg0 = 7'b000\_0001;

parameter SegErr = 7'b111\_1111;

reg [7:0] left, right; // numbers to calculate

wire [7:0] out;

wire [7:0] conv\_right;

wire cal\_err; // carry when calculating

reg num\_err; // error of BCD input

Convert conv (conv\_right, SW[7:0]);

Calculator cal (out, left, right, cal\_err);

// setting values

always@(\*)

begin

left = SW[15:8];

case(SW[16])

1'b1: right = conv\_right; // if operator is minus, make right number converted

default: right = SW[7:0];

endcase

end

// to detect error of BCD input

always@(\*)

begin

if (SW[3:0] > 9 || SW[7:4] > 9 || SW[11:8] > 9 || SW[15:12] > 9)

num\_err = 1'b1;

else num\_err = 1'b0;

end

// show variables to segment

always@(\*)

begin

case({cal\_err ^ SW[16], num\_err}) // invalid BCD, overflow/underflow detection

2'b00: begin

case(out[3:0]) // out segment

9:HEX0=Seg9; 8:HEX0=Seg8; 7:HEX0=Seg7; 6:HEX0=Seg6;

5:HEX0=Seg5; 4:HEX0=Seg4; 3:HEX0=Seg3; 2:HEX0=Seg2;

1:HEX0=Seg1; 0:HEX0=Seg0; default: HEX0 = SegErr;

endcase

case(out[7:4])

9:HEX1=Seg9; 8:HEX1=Seg8; 7:HEX1=Seg7; 6:HEX1=Seg6;

5:HEX1=Seg5; 4:HEX1=Seg4; 3:HEX1=Seg3; 2:HEX1=Seg2;

1:HEX1=Seg1; 0:HEX1=Seg0; default: HEX1 = SegErr;

endcase

LEDG[8] = 1'b0;

end

2'b10: begin // overflow/underflow error

HEX0 = SegErr;

HEX1 = SegErr;

LEDG[8] = 1'b1;

end

default: begin // invalid BCD error

HEX0 = SegErr;

HEX1 = SegErr;

LEDG[8] = 1'b0;

end

endcase

case(SW[3:0]) // right segment

9:HEX4=Seg9; 8:HEX4=Seg8; 7:HEX4=Seg7; 6:HEX4=Seg6;

5:HEX4=Seg5; 4:HEX4=Seg4; 3:HEX4=Seg3; 2:HEX4=Seg2;

1:HEX4=Seg1; 0:HEX4=Seg0; default: HEX4 = SegErr;

endcase

case(SW[7:4])

9:HEX5=Seg9; 8:HEX5=Seg8; 7:HEX5=Seg7; 6:HEX5=Seg6;

5:HEX5=Seg5; 4:HEX5=Seg4; 3:HEX5=Seg3; 2:HEX5=Seg2;

1:HEX5=Seg1; 0:HEX5=Seg0; default: HEX5 = SegErr;

endcase

case(SW[11:8]) // left segment

9:HEX6=Seg9; 8:HEX6=Seg8; 7:HEX6=Seg7; 6:HEX6=Seg6;

5:HEX6=Seg5; 4:HEX6=Seg4; 3:HEX6=Seg3; 2:HEX6=Seg2;

1:HEX6=Seg1; 0:HEX6=Seg0; default: HEX6 = SegErr;

endcase

case(SW[15:12])

9:HEX7=Seg9; 8:HEX7=Seg8; 7:HEX7=Seg7; 6:HEX7=Seg6;

5:HEX7=Seg5; 4:HEX7=Seg4; 3:HEX7=Seg3; 2:HEX7=Seg2;

1:HEX7=Seg1; 0:HEX7=Seg0; default: HEX7 = SegErr;

endcase

end

endmodule

// calculate whatever it is

module Calculator(outBCD, leftBCD, rightBCD, c\_err);

input [7:0] leftBCD, rightBCD;

output [7:0] outBCD;

output wire c\_err; // if this is 1'b1, return overflow/underflow error

wire c\_out0;

Full\_Adder fulladd1(outBCD[3:0], c\_out0, leftBCD[3:0], rightBCD[3:0], 1'b0);

Full\_Adder fulladd2(outBCD[7:4], c\_err, leftBCD[7:4], rightBCD[7:4], c\_out0);

endmodule

// 4 bit adder

module Full\_Adder(sumBCD, c\_out, leftBCD, rightBCD, c\_in);

input [3:0] leftBCD, rightBCD;

input c\_in;

output reg [3:0] sumBCD;

output reg c\_out;

wire [3:0] car; // carry

wire [3:0] temp\_out;

assign temp\_out[0] = leftBCD[0] ^ rightBCD[0] ^ c\_in;

assign car[0] = ((leftBCD[0] ^ rightBCD[0]) & c\_in) | (leftBCD[0] & rightBCD[0]);

assign temp\_out[1] = leftBCD[1] ^ rightBCD[1] ^ car[0];

assign car[1] = ((leftBCD[1] ^ rightBCD[1]) & car[0]) | (leftBCD[1] & rightBCD[1]);

assign temp\_out[2] = leftBCD[2] ^ rightBCD[2] ^ car[1];

assign car[2] = ((leftBCD[2] ^ rightBCD[2]) & car[1]) | (leftBCD[2] & rightBCD[2]);

assign temp\_out[3] = leftBCD[3] ^ rightBCD[3] ^ car[2];

assign car[3] = ((leftBCD[3] ^ rightBCD[3]) & car[2]) | (leftBCD[3] & rightBCD[3]);

always@(\*)

begin

if (car[3] == 0) begin

case(temp\_out)

4'b1010: begin // 10

sumBCD = 4'b0000;

c\_out = 1'b1;

end

4'b1011: begin // 11

sumBCD = 4'b0001;

c\_out = 1'b1;

end

4'b1100: begin // 12

sumBCD = 4'b0010;

c\_out = 1'b1;

end

4'b1101: begin // 13

sumBCD = 4'b0011;

c\_out = 1'b1;

end

4'b1110: begin // 14

sumBCD = 4'b0100;

c\_out = 1'b1;

end

4'b1111: begin // 15

sumBCD = 4'b0101;

c\_out = 1'b1;

end

default: begin // 0~9

sumBCD = temp\_out;

c\_out = car[3];

end

endcase

end

else begin

case(temp\_out)

4'b0000: sumBCD = 4'b0110; // 16

4'b0001: sumBCD = 4'b0111; // 17

4'b0010: sumBCD = 4'b1000; // 18

4'b0011: sumBCD = 4'b1001; // 19

default: sumBCD = 4'b0000; // except

endcase

c\_out = 1'b1;

end

end

endmodule

// get 9's and 10's complement of subtractBCD

module Convert(outBCD, subtractBCD);

input [7:0] subtractBCD;

wire [7:0] convertedBCD;

output wire [7:0] outBCD;

wire out;

// 9's complement

assign convertedBCD[7] = ~subtractBCD[7] & ~subtractBCD[6] & ~subtractBCD[5];

assign convertedBCD[6] = subtractBCD[6] ^ subtractBCD[5];

assign convertedBCD[5] = subtractBCD[5];

assign convertedBCD[4] = ~subtractBCD[4];

// 10's complement

assign convertedBCD[3] = (~subtractBCD[3] & ~subtractBCD[2]) & ( ~subtractBCD[1] | ~subtractBCD[0] );

assign convertedBCD[2] = (subtractBCD[2] & ~subtractBCD[1]) | (~subtractBCD[2] & subtractBCD[1] & subtractBCD[0]) | (subtractBCD[2] & subtractBCD[1] & ~subtractBCD[0]);

assign convertedBCD[1] = (subtractBCD[1] & subtractBCD[0]) | (~subtractBCD[1] & ~subtractBCD[0]);

assign convertedBCD[0] = subtractBCD[0];

assign outBCD = convertedBCD;

endmodule

## Troubleshooting process

Full Adder에서 캐리가 생기지 않아도 BCD 상에서는 캐리가 생겨야 되는 부분과 그냥 캐리가 생기는 부분을 구분하지 않고 작성하다가 이상한 합이나 차가 나오는 오류가 있었습니다. 이때 더하는 수에 대해 여러 케이스를 나눠 테스트하다가 첫자리를 더했을 때 16보다 커지면 오류가 생긴다는 걸 알아내고 케이스를 추가해 해결했습니다.

차를 구현할 때 10의 보수를 이용하려 했는데, 처음에는 십의 자리와 일의 자리 모두 10의 보수로 만들었다가 10만큼의 차이가 계속 났었습니다. 손으로 직접 해보니, 전체 수에 대한 10의 보수를 만들어내려면 십의 자리에는 9의 보수를, 일의 자리에는 10의 보수를 만들어야 했고, 이를 코드에 적용하여 정상 작동하게 만들었습니다.

차를 구현한 후, 보수를 이용하면 언더플로우는 나온 캐리값이 0일때로, 기존의 합일때의 에러 검출방식인 캐리가 1인지 확인하는 것과는 반대였습니다. 이를 스위치와 XOR 연산을 시켜, 덧셈(0)일 때 캐리가 1이면 에러, 뺄셈(1)일 때 캐리가 0이면 에러를 제대로 검출하게 만들었습니다.