Logic Design 1st term project

2018320192 손재민

1. Project summary

a. Overview : 이 프로그램은 2자리 bcd 숫자 2개를 입력 받아 그 합 또는 차를 구한 뒤, 결과를 출력해주는 프로그램이다. 보드 상에서 SW[15]부터 SW[8]까지가 1번째 숫자에 해당하고, SW[7]부터 SW[0]까지가 2번째 숫자에 해당하며, SW[16]이 내려가 있으면 덧셈을, 올라가 있으면 뺄셈을 의미한다. HEX7과 HEX6이 1번째 숫자를, HEX5와 HEX4는 2번째 숫자를, 그리고 HEX1과 HEX0은 계산 결과에 해당하는 숫자를 출력한다. LEDG[8]은 오버플로우 또는 언더플로우가 발생했거나 입력 값이 적절한 bcd 숫자가 아닐 경우 켜지고 경우 켜지고 그렇지 않을 경우는 끄게 된다.

b. Modules

1) bcd_adder : 입력 값으로 bcd숫자 a와 b, 그리고 캐리 값 cin을 입력 받는다. 우선 a와 b, 그리고 cin으로 full_adder를 이용하여 합을 구한다. 이후 만약 캐리가 발생하였거나, 합이 bcd가 아닌 9보다 큰 값이 되었다면 bcd 캐리 값을 6으로 만들고, 아니면 0으로 만들어 이 bcd 캐리 값과 앞에서 구한 합을 다시 full_adder를 이용해 합을 구한다. 이후 마지막으로 구한 합을 sum에, 과 앞에서 발생했던 캐리를 car에 출력한다.

2) full_adder : 입력 값으로 4비트 이진수 a와 b, 그리고 캐리 값 cin을 입력 받는다. Full adder 4개를 이용하여 a와 b의 합을 구한 뒤 합을 sum에, 캐리를 car에 출력한다.

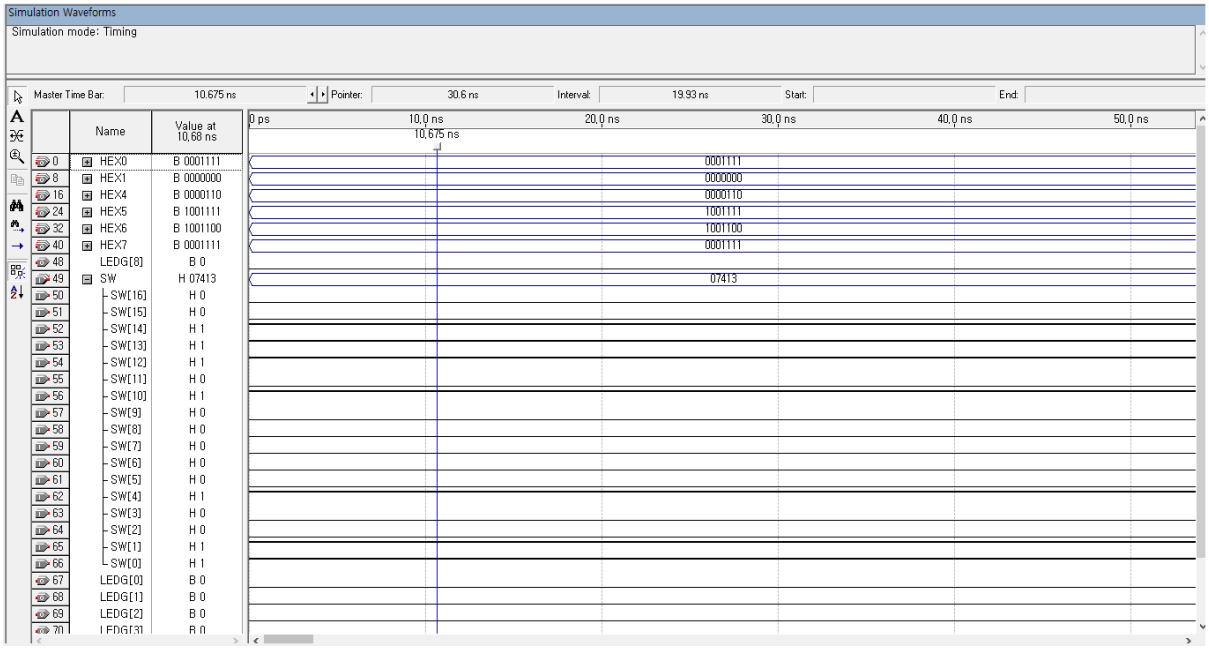
3) comple9_gen : 입력 값으로 bcd 숫자 num_origin 과 1자리 이진수 operator를 입력 받는다. 만약 operator가 0이라면, 즉 (+)라면 num_origin 그대로를, operator가 1이라면, 즉 (-)라면 num_origin의 9의 보수를 num_comple에 출력한다. 보수를 구하는 logic은 k-map을 그려 구했다.

4) midterm_sim : 입력 값으로 각각 SW[16]은 operator, SW[15] 부터 SW[8]까지 1번째 숫자, SW[7]부터 SW[0]까지 2번째 숫자를 입력 받는다. 우선 operator와 2번째 숫자를 comple9_gen에 입력하여 complement를 구한다. 덧셈일 경우 첫번째 캐리에 해당하는 car_0을 0으로, 뺄셈일 경우 car_0을 1로 만든 뒤(10진수로 생각하면 뺄셈을 음수의 합으로 바꾸는 과정에서 10의 보수를 구해야 하나, 실제로 구해진 것은 9의 보수이므로, 10의 보수를 사용하기 위해 1을 더하는 것을 의미한다.), complement와 1번째 숫자, 그리고 car_0 bcd_adder에 입력하여 합을 sum에 할당하고, 캐리를 car_2에 할당한다. 출력 부분에 대해서는, 우선 입력 받은 숫자들이 모두 적절한 bcd이고, 오버플로우나 언더플로우가 발생하지 않았다면 HEX7과 HEX6을 입력받은 1번째 숫자로, HEX5와 HEX4를 입력받은 2

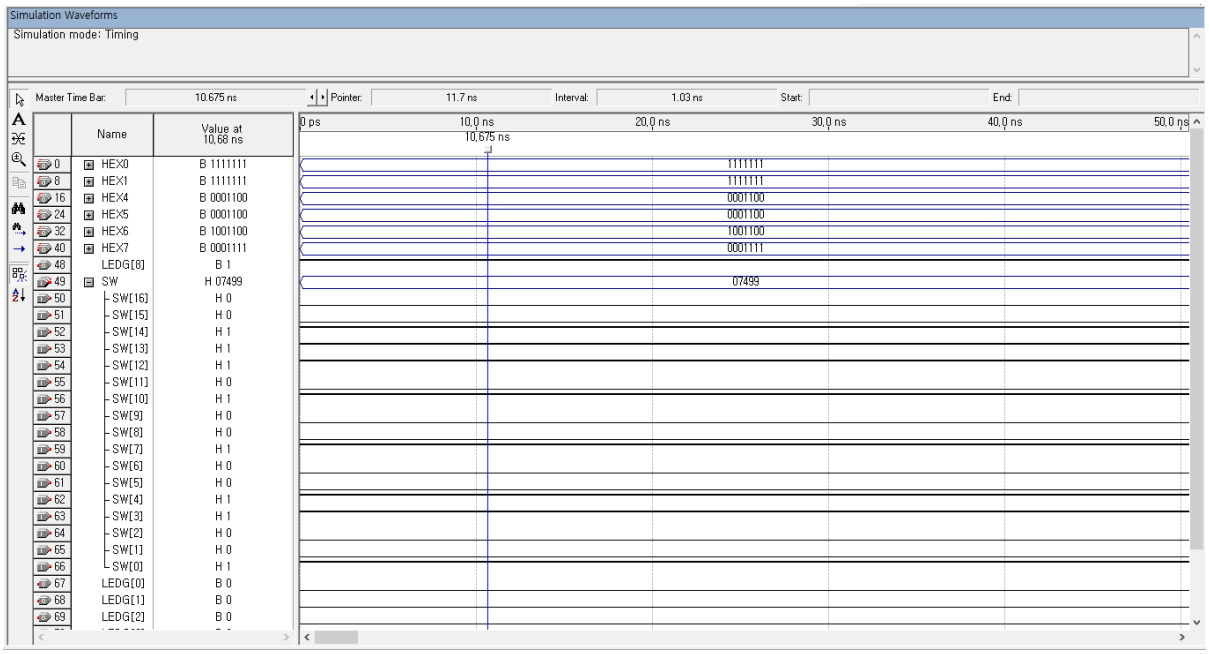
번째 숫자로, 그리고 HEX1과 HEX0을 sum에 해당하는 숫자로 출력하고 LEDG[8]을 끈다. 만약 입력 받은 숫자들 중에 bcd가 아닌 값이 있거나 오버플로우나 언더플로우가 발생했다면 bcd에 해당하는 HEX와 HEX1, HEX0을 모두 끄고 LEDG[8]을 켠다.

2. Simulation screenshot

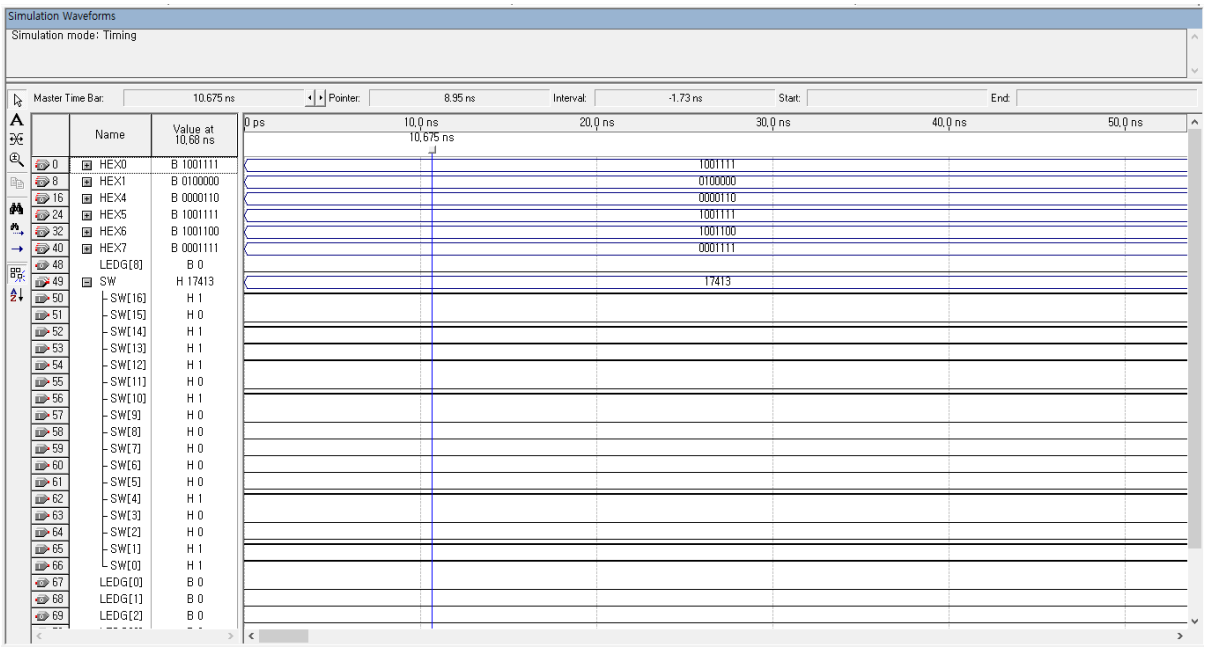
a. 74 + 13 = 87



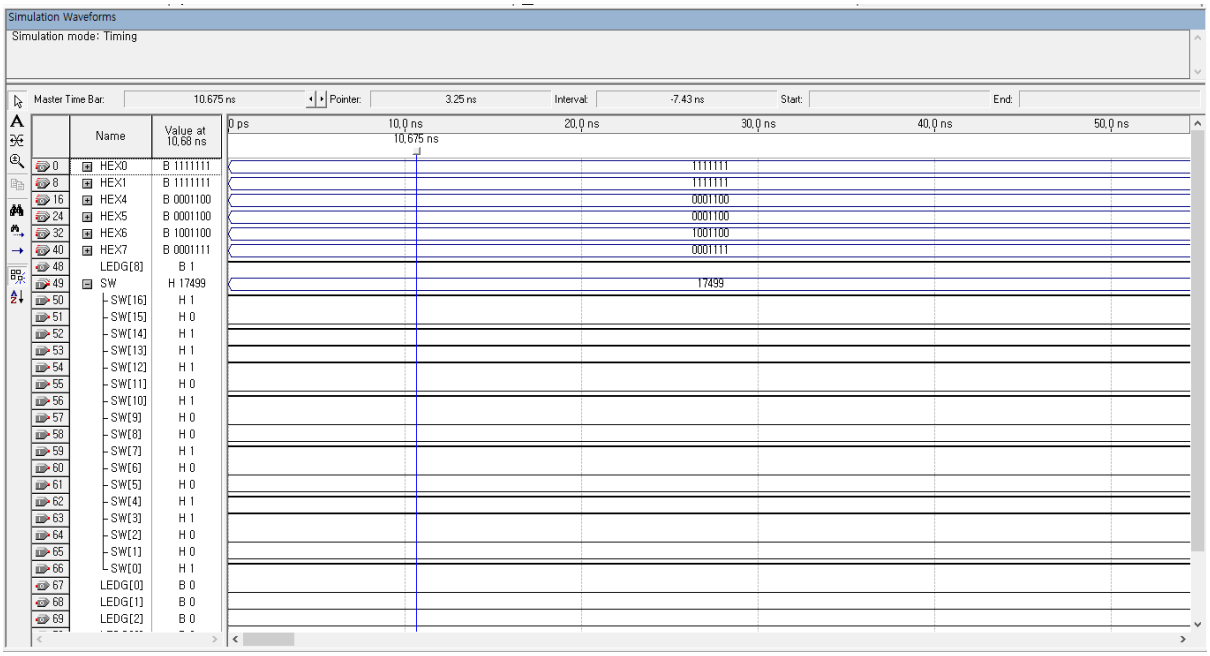
b. 74 + 99 = 173 (오버플로우)



c. 74 – 13 = 61



d. 74 – 99 = -25 (언더플로우)



3. HDL code

```
module midterm_sim(
         input [16:0] SW,
         output reg [0:6] HEX0, HEX1, HEX4, HEX5, HEX6, HEX7,
         output reg [8:0] LEDG
);
//input 1 is for first input, and 2 is for second input.
//operator is for (+) or (-).
reg [3:0] input1_1, input1_2, input2_1, input2_2, operator;
//flow detector detects whether overflow / underflow occurred or not.
reg car_0, flow_detector;
```

```
wire [3:0] sum1, sum2, complement1, complement2;
wire car_1, car_2;
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;

//Generate 9's complement of second input.
comple9_gen comple9_gen1 (input2_1, operator, complement1);
comple9_gen comple9_gen2 (input2_2, operator, complement2);
//Add first input and second input after making complement.
bcd_adder bcd_adder1 (sum1, car_1, complement1, input1_1, car_0);
bcd_adder bcd_adder2 (sum2, car_2, complement2, input1_2, car_1);

//Get BCD input from SW[15] to SW[8] as input 1, and from SW[7] to SW[0] as input2.
//Check if operator is (+) or (-).
always @(*)
begin
	input2_1 = SW[3:0];
	input2_2 = SW[7:4];
	input1_1 = SW[11:8];
	input1_2 = SW[15:12];
	operator = SW[16];
	car_0 = operator;
	//Overflow : carry occurred when operator is (+)
	//Underflow : carry not occurred when operator is (-)
	flow_detector = car_2 ^ operator;
end

//Determine segments' output.
always @(*)
begin
	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 = 7'b1111111;
	endcase
	case(SW[11:8])
		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 = 7'b1111111;
	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 = 7'b1111111;
	endcase
	case(SW[3:0])
		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 = 7'b1111111;
	endcase
	//Check if input is bcd number and flow has occurred.
	if(~flow_detector & (SW[15:12] < 10 & SW[11:8] < 10 & SW[7:4] < 10 & SW[3:0] < 10))
		//If input is proper and flow has not occurred, then print the result.
		begin
			case(sum2)
				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 = 7'b1111111;
			endcase
			case(sum1)
				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 = 7'b1111111;
			endcase
			LEDG[8] = 1'b0;
		end
	else
		//If input is improper or flow has occurred, then turn off result segment and turn on
led.
		begin
			HEX1 = 7'b1111111;
```

```
                            HEX0 = 7'b1111111;
                            LEDG[8] = 1'b1;
                    end
end
endmodule

//Full adder of bcd number.
//Add two bcd input a and b with carry cin and assign it to the sum.
//Carry is assigned to the car.
module bcd_adder(sum, car, a, b, cin);
output [3:0] sum;
output car;
input [3:0] a, b;
input cin;
wire [3:0] sum_temp, bcdcar;
wire car_temp, temp;

//At first, add two number as ordinary 4 bit binary number.
full_adder full_adder1(sum_temp, car_temp, a, b, cin);
//If the result is not bcd(result is larger than 9), then carry has occured.
assign car = car_temp | (sum_temp[3] & (sum_temp[2] | sum_temp[1]));
assign bcdcar[0] = 1'b0;
assign bcdcar[1] = car;
assign bcdcar[2] = car;
assign bcdcar[3] = 1'b0;
//If carry has occurred, then add 6 again to the result.
full_adder full_adder2(sum, temp, bcdcar, sum_temp, 1'b0);

endmodule

//Full adder of 4-bit binary number.
//Add two binary input a and b with carry cin and assign it to the sum.
//Carry is assigned to the car.
module full_adder(sum, car, a, b, cin);
output [3:0] sum;
output car;
input [3:0] a, b;
input cin;
wire [2:0] c;

assign sum[0]=a[0] ^ b[0] ^ cin;
assign c[0]=((a[0] ^ b[0]) & cin) | (a[0] & b[0]);

assign sum[1]=a[1] ^ b[1] ^ c[0];
assign c[1]=((a[1] ^ b[1]) & c[0]) | (a[1] & b[1]);

assign sum[2]=a[2] ^ b[2] ^ c[1];
assign c[2]=((a[2] ^ b[2]) & c[1]) | (a[2] & b[2]);

assign sum[3]=a[3] ^ b[3] ^ c[2];
assign car=((a[3] ^ b[3]) & c[2]) | (a[3] & b[3]);

endmodule

//Generates 9's complement of num_origin and assign the result to num_comple.
//If operator is (+), then the result would be just as same as the input(not 9's complement).
//If operator is (-), then the result would be 9's complement.
module comple9_gen(num_origin, operator, num_comple);
input [3:0] num_origin;
input operator;
output [3:0] num_comple;

//This logic is made by K-map.
assign num_comple[3] = (operator & ~num_origin[3] & ~num_origin[2] & ~num_origin[1]) | (~operator &
num_origin[3]);
assign num_comple[2] = (operator & (num_origin[2] ^ num_origin[1])) | (~operator & num_origin[2]);
assign num_comple[1] = num_origin[1];
assign num_comple[0] = operator ^ num_origin[0];

endmodule
```

4. Troubleshooting process

a) 이번 프로젝트를 진행하면서 전체적인 프로그램의 로직을 구상하는 것은 특별히 어렵지는 않았으나, 구상한 로직을 verilog로 옮기는 과정에서 조금 문제를 겪었다. 특정 조건이 맞을 때만 원하는 모듈을 실행하고 싶었으나 그런 코드를 작성하면 많은 오류가 발생했는데, 이는 verilog 상에서의 모듈을 다른 프로그래밍 언어의 함수와 같은 것이라 착각했기 때문에 일어나는 오류였다. 또 9의 보수를 구하는 과정에서 어떻게 이를 회로로 옮겨야 하는지 고민을 많이 했는데, k-map을 그리면 간단하게 게이트를 이용해서 나타낼 수 있다는 사실을 깨닫고 나니 별 어려움 없이 구할 수 있었다. 본 프로젝트를 진행하면서 Verilog로 프로그램을 작성할 때는 다른 프로그래밍 언어와는 다르게 마치 회로를 설계하듯이 게이트와 모듈의 연결을 고려하여 설계해야 한다는 사실을 배울 수 있었다.