**10주차 결과 보고서**

20150555 남민혁

**1.** **4bit Binary Parallel Adder 의 결과 및 Simulation 과정에 대해서 설명하시오. (verliog source, 출력 예시, 과정 상세히 적을것!)**

기존에 구현한 Full Adder 4개를 병렬적으로 연결하여 4개의 비트를 더하는 4bit Binary Parallel Adder를 구현하고자 한다. 따라서 먼저 Full Adder에 대한 진리표와 간소화시킨 논리식을 구하면 다음과 같다.

**-Truth Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| A | B |  | S |  |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**-Karnaugh Map (S)**

|  |  |  |
| --- | --- | --- |
| AB | 0 | 1 |
| 00 | 0 | 1 |
| 01 | 1 | 0 |
| 11 | 0 | 1 |
| 10 | 1 | 0 |

**-Karnaugh Map ()**

|  |  |  |
| --- | --- | --- |
| AB | 0 | 1 |
| 00 | 0 | 0 |
| 01 | 0 | 1 |
| 11 | 1 | 1 |
| 10 | 0 | 1 |

**-최소화된 Boolean 함수**

**-Verilog 코딩**

다음은 위에서 살펴본 1bit Full Adder 4개를 병렬적으로 연결하여 구성한 4bit Binary Parallel Adder를 구현한 Verilog 코드이다. module adder1bit(A, B, Ci, S, Co);는 위에서 살펴본 1 bit adder에 대한 모듈이고, LSB부터 차례로 하나씩 덧셈 연산을 하여 4비트 연산을 수행하는 모듈은 module adder4bit(A, B, Ci, S, Co); 이다.

각 행은 논리회로 구성을 위한 Verilog 코드, 이를 바탕으로 모든 가능한 입력으로 부터 기대되는 출력이 나타나는지를 확인하기 위한 Test bench 코드이다. 끝으로 이를 FPGA에 이진 프로그램을 주입하여 입력을 받고, 출력 값을 LED 등에 나타내기 위한 Constraint 코드이다.

|  |  |
| --- | --- |
| Verilog | `timescale 1ns / 1ps  module adder1bit(A, B, Ci, S, Co);  input A, B, Ci;  output S, Co;  assign S = A^B^Ci;  assign Co = A&B | Ci&(A^B);  endmodule  module adder4bit(A, B, Ci, S, Co);  input [3:0] A, B; input Ci;  output [3:0] S; output Co;  wire [3:0] A, B, S; wire Ci, Co;  wire [2:0] C;  adder1bit add1(A[0], B[0], Ci, S[0], C[0]);  adder1bit add2(A[1], B[1], C[0], S[1], C[1]);  adder1bit add3(A[2], B[2], C[1], S[2], C[2]);  adder1bit add4(A[3], B[3], C[2], S[3], Co);  endmodule |
| Test Bench | `timescale 1ns / 1ps  module adder4bit\_tb();  reg clk, Ci; reg [3:0] A, B;  wire [3:0] S; wire Co;  adder4bit connect(A, B, Ci, S, Co);  initial begin  clk=0; A[3]=0; A[2]=1; A[1]=0; A[0]=1;  B[3]=0; B[2]=0; B[1]=0; B[0]=0; Ci=1;  end    always clk = #10 ~clk;  always @(posedge clk) begin  Ci <= ~Ci; B[0] <= #39.999 ~B[0];  B[1] <= #79.999 ~B[1]; B[2] <= #159.999 ~B[2];  B[3] <= #319.999 ~B[3];  end    endmodule |
| XDC | set\_property IOSTANDARD LVCMOS18 [get\_ports {A[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports Ci]  set\_property IOSTANDARD LVCMOS18 [get\_ports Co]  set\_property PACKAGE\_PIN J4 [get\_ports {A[3]}]  set\_property PACKAGE\_PIN L3 [get\_ports {A[2]}]  set\_property PACKAGE\_PIN K3 [get\_ports {A[1]}]  set\_property PACKAGE\_PIN M2 [get\_ports {A[0]}]  set\_property PACKAGE\_PIN K6 [get\_ports {B[3]}]  set\_property PACKAGE\_PIN J6 [get\_ports {B[2]}]  set\_property PACKAGE\_PIN L5 [get\_ports {B[1]}]  set\_property PACKAGE\_PIN L4 [get\_ports {B[0]}]  set\_property PACKAGE\_PIN F13 [get\_ports {S[3]}]  set\_property PACKAGE\_PIN F14 [get\_ports {S[2]}]  set\_property PACKAGE\_PIN F16 [get\_ports {S[1]}]  set\_property PACKAGE\_PIN E17 [get\_ports {S[0]}]  set\_property PACKAGE\_PIN AB15 [get\_ports Ci]  set\_property PACKAGE\_PIN F15 [get\_ports Co] |

**-Simulation 출력 결과 비교**

**Diagram, timeline

Description automatically generated**

모든 가능한 입력 값에 대한 출력값을 확인하기 위하여 Simulation을 시행한 결과 의도한 진리표와 모든 결과가 동일함을 확인하였다.

**-출력 예시**

위의 시뮬레이션 결과는 첫번째 입력 (A)에 5를 고정한 후, 두 번째 입력 (B)에 0부터 15까지 번걸아가며 넣은 결과가 S에 나타나는 구조이다.

가장 첫번째 A에 5, B에 0이 들어간 경우를 고려하자. 이 경우에는 두 수의 합산 결과인 5가 S에 나타난다. FPGA에서는 A와 B입력을 각각 0101과 0000으로 스위치를 설정하면, 스위치 위에 있는 LED에 0101의 형태로 불이 들어오는 것을 확인할 수 있다.

한편 Carry (Co)가 발생하는 경우, 즉 overflow가 발생하는 첫번째 경우인 5와 11(b)를 더한 경우를 고려하자. 이 경우에는 16이 연산 결과이나, 4비트로는 최대 15까지만 표기할 수 있으므로, LED 등에는 0001이 나타나고, Carry 비트에 해당하는 LED에 불이 들어오는 형태로 출력된다.

**-구현된 Schematic**

Diagram, schematic

Description automatically generated

**2. 4bit Binary Parallel Subtractor 의 결과 및 Simulation 과정에 대해서 설명하시오. (verliog source, 출력 예시, 과정 상세히 적을것!)**

기존에 구현한 Full Subtractor 4개를 병렬적으로 연결하여 4개의 비트를 빼는 4bit Binary Parallel Subtractor를 구현하고자 한다. 따라서 먼저 Full Subtractor에 대한 진리표와 간소화시킨 논리식을 구하면 다음과 같다.

**-Truth Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| A | B |  | D |  |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**-Karnaugh Map (D)**

|  |  |  |
| --- | --- | --- |
| AB | 0 | 1 |
| 00 | 0 | 1 |
| 01 | 1 | 0 |
| 11 | 0 | 1 |
| 10 | 1 | 0 |

**-Karnaugh Map ()**

|  |  |  |
| --- | --- | --- |
| AB | 0 | 1 |
| 00 | 0 | 1 |
| 01 | 1 | 1 |
| 11 | 0 | 1 |
| 10 | 0 | 0 |

**-최소화된 Boolean 함수**

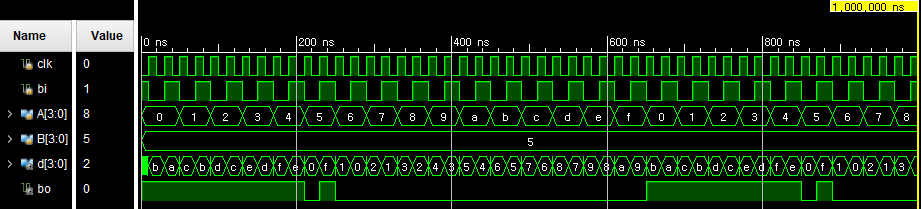
**-Verilog 코딩**

다음은 위에서 살펴본 1bit Full Subtractor 4개를 병렬적으로 연결하여 구성한 4bit Binary Parallel Subtractor를 구현한 Verilog 코드이다. module sub1bit(A, B, bi, d, bo);는 위에서 살펴본 1 bit subtractor에 대한 모듈이고, LSB부터 차례로 하나씩 뺄셈 연산을 하여 4비트 연산을 수행하는 모듈은 module sub4bit(A, B, bi, d, bo);이다.

각 행은 논리회로 구성을 위한 Verilog 코드, 이를 바탕으로 모든 가능한 입력으로 부터 기대되는 출력이 나타나는지를 확인하기 위한 Test bench 코드이다. 끝으로 이를 FPGA에 이진 프로그램을 주입하여 입력을 받고, 출력 값을 LED 등에 나타내기 위한 Constraint 코드이다.

|  |  |
| --- | --- |
| Verilog | `timescale 1ns / 1ps  module sub1bit(A, B, bi, d, bo);  input A, B, bi;  output d, bo;  assign d = A^(B^bi);  assign bo = B&bi | !A&(B^bi);  endmodule  module sub4bit(A, B, bi, d, bo);  input [3:0] A, B; input bi;  output [3:0] d; output bo;  wire [3:0] A, B, d; wire bi, bo;  wire [2:0] bout;  sub1bit sub1(A[0], B[0], bi, d[0], bout[0]);  sub1bit sub2(A[1], B[1], bout[0], d[1], bout[1]);  sub1bit sub3(A[2], B[2], bout[1], d[2], bout[2]);  sub1bit sub4(A[3], B[3], bout[2], d[3], bo);  endmodule |
| Test Bench | `timescale 1ns / 1ps  module sub4bit\_tb();  reg clk, bi; reg [3:0] A, B;  wire [3:0] d; wire bo;  sub4bit connect(A, B, bi, d, bo);  initial begin  clk=0; B[0]=1; B[1]=0; B[2]=1; B[3]=0;  bi=1; A[3]=0; A[2]=0; A[1]=0; A[0]=0;  end  always clk = #10 ~clk;  always @(posedge clk) begin  bi <= ~bi;  A[0] <= #39.999 ~A[0];  A[1] <= #79.999 ~A[1];  A[2] <= #159.999 ~A[2];  A[3] <= #319.999 ~A[3];  end  endmodule |
| XDC | set\_property IOSTANDARD LVCMOS18 [get\_ports {A[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports Ci]  set\_property IOSTANDARD LVCMOS18 [get\_ports Co]  set\_property PACKAGE\_PIN J4 [get\_ports {A[3]}]  set\_property PACKAGE\_PIN L3 [get\_ports {A[2]}]  set\_property PACKAGE\_PIN K3 [get\_ports {A[1]}]  set\_property PACKAGE\_PIN M2 [get\_ports {A[0]}]  set\_property PACKAGE\_PIN K6 [get\_ports {B[3]}]  set\_property PACKAGE\_PIN J6 [get\_ports {B[2]}]  set\_property PACKAGE\_PIN L5 [get\_ports {B[1]}]  set\_property PACKAGE\_PIN L4 [get\_ports {B[0]}]  set\_property PACKAGE\_PIN F13 [get\_ports {S[3]}]  set\_property PACKAGE\_PIN F14 [get\_ports {S[2]}]  set\_property PACKAGE\_PIN F16 [get\_ports {S[1]}]  set\_property PACKAGE\_PIN E17 [get\_ports {S[0]}]  set\_property PACKAGE\_PIN AB15 [get\_ports Ci]  set\_property PACKAGE\_PIN F15 [get\_ports Co] |

**-Simulation 출력 결과 비교**

****

모든 가능한 입력 값에 대한 출력값을 확인하기 위하여 Simulation을 시행한 결과 의도한 진리표와 모든 결과가 동일함을 확인하였다.

**-출력 예시**

위의 시뮬레이션 결과는 첫번째 입력 (A)에 0부터 15까지 번걸아가며 들어가고, 두 번째 입력 (B)에 5를 고정하여 연산한 결과가 S에 나타나는 구조이다.

Borrow가 발생하지 않는 가장 첫번째 경우, A에 5, B에 5가 들어간 경우를 고려하자. 이 경우에는 두 수의 합산 결과인 0이 S에 나타난다. FPGA에서는 A와 B입력을 각각 0101과 0101로 스위치를 설정하면, 스위치 위에 있는 LED에 0000의 형태로 불이 들어오는 것을 확인할 수 있다.

한편 Borrow(bo)가 발생하는 경우, 즉 overflow가 발생하는 첫번째 경우인 0 에서 5를 뺀 경우를 고려하자. 이 경우에는 -5가 연산 결과이나, 현재 부호를 포함하지 않는 4비트를 가정하므로. 최소 0까지만 표기할 수 있다. 따라서, LED 등에는 1011 (11)이 나타나고, Borrow 비트에 해당하는 LED에 불이 들어오는 형태로 출력된다.

**-구현된 Schematic**

**Diagram, schematic

Description automatically generated**

**3.** **BCD Adder 의 결과 및 Simulation 과정에 대해서 설명하시오. (verliog source, 출력 예시, 과정 상세히 적을것!)**

1번 문항에서 구현한 4bit Binary Parallel Adder를 활용하여 BCD 코드로 입력과 출력을 받는 덧셈 연산기로 변형하고자 한다. 그러나 이때 입력에 대하여 한자리 수를 더한다고 하였으므로, 입력에 대해서는 BCD코드와 이진수와 동일하다. 따라서 본 문항에서 처리해야하는 것은 1번 문항에서 구현한 결과 중에서 10이 넘는 두자리 수 결과에 대해서만 BCD 코드로 변형하는 것이 필요하다. 다음은1번 문항에 따라 Input과 같은 binary bit으로 표현된 결과가 출력되면 다시 BCD 코드로 변환하여 어떤 출력을 나타내야 하는지 진리표로 정리한 것이다.

**-Truth Table**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | | | | | **Output** | | | | | **DECIMAL** |
| **K** | **Z8** | **Z4** | **Z2** | **Z1** | **C** | **S8** | **S4** | **S2** | **S1** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 3 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 5 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 6 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 7 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 8 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 9 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 10 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 11 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 12 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 13 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 14 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 15 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 16 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 17 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 18 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 19 |

**-Verilog 코딩**

앞서 언급한 바와 같이 입력에 대해서는 1번 문항과 동일한 상황이므로, 1번 문항과 동일하게 앞서 살펴본 1bit Full Adder 4개를 병렬적으로 연결하여 구성하였다. module bcdadder1bit(A, B, Ci, S, Co);는 위에서 살펴본 1 bit adder에 대한 모듈이고, LSB부터 차례로 하나씩 덧셈 연산을 하여 4비트 연산을 수행하는 모듈은 module bcdadder4bit(A, B, Ci, S, Co);이다. 끝으로 module BCD(A, B, S, C);는 module bcdadder4bit(A, B, Ci, S, Co);을 통하여 구한 결과가 10을 넘는 경우 십의 자리 출력 값을 0001로 설정하고, 일의 자리 출력을 조정하는 모듈이다.

각 행은 논리회로 구성을 위한 Verilog 코드, 이를 바탕으로 모든 가능한 입력으로 부터 기대되는 출력이 나타나는지를 확인하기 위한 Test bench 코드이다. 끝으로 이를 FPGA에 이진 프로그램을 주입하여 입력을 받고, 출력 값을 LED 등에 나타내기 위한 Constraint 코드이다.

|  |  |
| --- | --- |
| Verilog | `timescale 1ns / 1ps  module bcdadder1bit(A, B, Ci, S, Co);  input A, B, Ci;  output S, Co;  assign S = A^B^Ci;  assign Co = A&B | Ci&(A^B);  endmodule  module bcdadder4bit(A, B, Ci, S, Co);  input [3:0] A, B; input Ci;  output [3:0] S; output Co;  wire [3:0] A, B, S; wire Ci, Co;  wire [2:0] C;  bcdadder1bit add1(A[0], B[0], Ci, S[0], C[0]);  bcdadder1bit add2(A[1], B[1], C[0], S[1], C[1]);  bcdadder1bit add3(A[2], B[2], C[1], S[2], C[2]);  bcdadder1bit add4(A[3], B[3], C[2], S[3], Co);  endmodule  module BCD(A, B, S, C);  input [3:0] A, B;  output [3:0] S; output C;  wire [3:0] B2, A2; wire Cout;  bcdadder4bit add1(A, B, 0, B2, Cout);  assign C = B2[3]&B2[1] | B2[3]&B2[2] | Cout;  assign A2 = 4'b0110\*C;  bcdadder4bit add2(A2, B2, 0, S);  endmodule |
| Test Bench | `timescale 1ns / 1ps  module BCD\_tb();  reg clk; reg [3:0] A, B;  wire [3:0] S; wire C;  BCD connect(A, B, S, C);  initial begin  clk = 0; A=4'd5;  B=4'd0;  end;  always clk = #10 ~clk;  always @(posedge clk) begin  B <= B+4'd1;  end  endmodule |
| XDC | set\_property IOSTANDARD LVCMOS18 [get\_ports {A[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {A[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {B[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[0]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[3]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[2]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports {S[1]}]  set\_property IOSTANDARD LVCMOS18 [get\_ports Ci]  set\_property IOSTANDARD LVCMOS18 [get\_ports Co]  set\_property PACKAGE\_PIN J4 [get\_ports {A[3]}]  set\_property PACKAGE\_PIN L3 [get\_ports {A[2]}]  set\_property PACKAGE\_PIN K3 [get\_ports {A[1]}]  set\_property PACKAGE\_PIN M2 [get\_ports {A[0]}]  set\_property PACKAGE\_PIN K6 [get\_ports {B[3]}]  set\_property PACKAGE\_PIN J6 [get\_ports {B[2]}]  set\_property PACKAGE\_PIN L5 [get\_ports {B[1]}]  set\_property PACKAGE\_PIN L4 [get\_ports {B[0]}]  set\_property PACKAGE\_PIN F13 [get\_ports {S[3]}]  set\_property PACKAGE\_PIN F14 [get\_ports {S[2]}]  set\_property PACKAGE\_PIN F16 [get\_ports {S[1]}]  set\_property PACKAGE\_PIN E17 [get\_ports {S[0]}]  set\_property PACKAGE\_PIN AB15 [get\_ports Ci]  set\_property PACKAGE\_PIN F15 [get\_ports Co] |

**-Simulation 출력 결과 비교**

**Timeline

Description automatically generated**

모든 가능한 입력 값에 대한 출력값을 확인하기 위하여 Simulation을 시행한 결과 의도한 진리표와 모든 결과가 동일함을 확인하였다.

**-출력 예시**

위의 시뮬레이션 결과는 첫번째 입력 (A)에 5를 고정한 후, 두 번째 입력 (B)에 0부터 15까지 번걸아가며 넣은 결과가 S에 나타나는 구조이다.

가장 첫번째 A에 5, B에 0이 들어간 경우를 고려하자. 이 경우에는 두 수의 합산 결과인 5가 S에 나타난다. FPGA에서는 A와 B입력을 각각 0101과 0000으로 스위치를 설정하면, 스위치 위에 있는 LED에 0000 / 0101의 형태로 불이 들어오는 것을 확인할 수 있다.

더 나아가 연산 결과가 10을 넘어가는 가장 첫번째 경우인 A에 5, B에 5가 들어가는 경우를 고려하자. 이 경우에는 연산 결과가 10을 넘어가므로, 십의 자리를 나타내는 C가 1로 설정되고, 일의 자리는 0이 될 수 있도록 연산 결과에 6을 더하는 과정을 거쳐 S에 출력된다. FPGA에서는 A와 B입력을 각각 0101과 0101을 스위치로 설정하면, 스위치 위에 있는 LED에 0001 / 0000 형태로 불이 들어오는 것을 확인할 수 있다.

한편 Simulation에서 기존 4bit parallel adder에서 overflow로 처리했던 16을 넘어가는 연산도 정상적인 결과를 출력할 수 있게 되는 것 처럼 보여지나, 본래 BCD Adder를 구성할 떄 한자리 BCD코드 (0~9) 두개의 연산에 대해서만 고려하므로, 해당 출력 결과는 본 상황에서는 고ㅕ할 부분이 아니다.

**-구현된 Schematic**

Diagram, schematic

Description automatically generated

**4. 결과 검토 및 논의 사항.**

이번주 실험에서 4비트 가산기, 4비트 감산기, 그리고 한자리수 BCD 코드에 대한 가산기를 구현하였다. 구현을 위하여 6주차에서 구현한 Full Adder와 Full Subtractor 를 활용하여 병렬적으로 연결하였다. 이를 통해 Verilog 코드로 작성하고 모든 가능한 입력에 대해 simulate 하고, FPGA에 이진 프로그램을 주입하여 기대되는 결과가 나타나는지 확인하였다. 그 결과 모든 디자인이 의도한 진리표와 다르지 않고 동일하게 결과가 출력됨을 확인하였다.

**5. 추가 이론 조사 및 작성.**

6주차 수업에서 살펴보았던4bit Carry Lookahead Adder를 다음과 같이 구현할 수 있다. 그러나 그 구현 방식이 복잡하여, 4비트 이상으로는 잘 사용하지 않는 것으로 알려져있다.

|  |  |
| --- | --- |
| Verilog | `timescale 1ns / 1ps  module CLA(in\_a,in\_b,in\_c,out\_s,out\_c);  input[7:0] in\_a,in\_b;  output[7:0] out\_s;  input in\_c;  output out\_c;  wire[7:0] w\_p,w\_g,w\_c;  assign w\_c[0] = (in\_a[0]& in\_b[0]) | (in\_b[0]&in\_c) | (in\_c & in\_a[0]);  assign w\_c[7:1] = w\_g[7:1] | (w\_p[7:1] & w\_c[6:0]);  assign out\_s[0] = in\_a[0]^in\_b[0]^in\_c;  assign out\_s[7:1] = in\_a[7:1]^in\_b[7:1]^w\_c[6:0];  assign w\_p[7:0] = in\_a[7:0]^in\_b[7:0];  assign w\_g[7:0] = in\_a[7:0]&in\_b[7:0];    assign out\_c = w\_c[7];  endmodule |
| Test Bench | `timescale 1ns / 1ps  module CLA\_tb();  reg clk, in\_c; reg [3:0] in\_a, in\_b;  wire [3:0] out\_s; wire out\_c;  CLA connect(in\_a,in\_b,in\_c,out\_s,out\_c);  initial begin  clk=0; in\_a[3]=0; in\_a[2]=1; in\_a[1]=0; in\_a[0]=1;  in\_b[3]=0; in\_b[2]=0; in\_b[1]=0; in\_b[0]=0; in\_c=1;  end    always clk = #10 ~clk;  always @(posedge clk) begin  in\_c <= ~in\_c; in\_b[0] <= #39.999 ~in\_b[0];  in\_b[1] <= #79.999 ~in\_b[1]; in\_b[2] <= #159.999 ~in\_b[2];  in\_b[3] <= #319.999 ~in\_b[3];  end    endmodule |

위의 모듈을 시뮬레이션 한 결과는 다음과 같다.

Diagram

Description automatically generated

**6. 참고 문헌**

강석태, “Verilog HDL Summary”, <http://vlsi.hongik.ac.kr/lecture/%EC%8B%A4%ED%97%98/Verilog_Summary.pdf>.

박재동, “[Verilog 예제] CLA (Carry Look Ahead) 모듈”, <http://egloos.zum.com/rabe/v/1285020>.

한국기술교육대학교 전기전자통신공학부, “디지털시스템 설계 및 실습” Verilog HDL 문법 강의자료, <https://cms3.koreatech.ac.kr/sites/yjjang/down/dsys11/M01_VerilogHDL01.pdf>.