Lab8--Design and Utilization of Full Adder

姓名：潘子曰 学号：3180105354 专业：计算机科学与技术
课程名称：逻辑与计算机设计基础实验 同组学生姓名：张佳文
试验时间：2019-11-14 实验地点：紫金港东4-509 指导老师：洪奇军

1. Objectives & Requirements

1. Master the principle and logic functionality of **1-bit full adder**.
2. Understand the principle and **carry delay** of **ripple carry adder**.
3. Master the principle of **carry look-ahead adder**.
4. Master the principle of **adder-subtractor**.
5. Master the design method of **ALU redundancy design**.
6. Know the basic data I/O interaction method on FPGA development platform.

2. Contents & Principles

2.1 Tasks

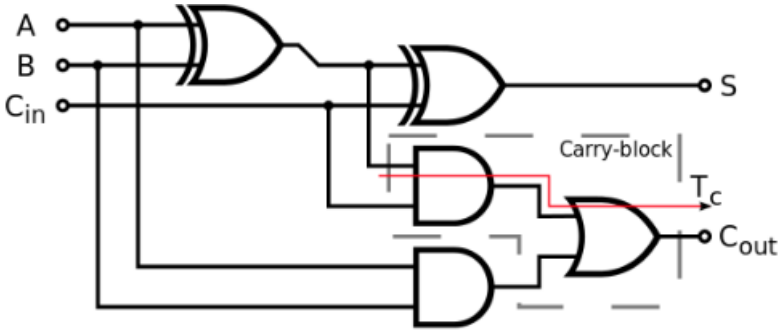
1. Design and realize **1-bit full adder** and do simulation verification.
2. Design 4-bit **ripple carry adder**.
3. Design **4-bit carry look-ahead module** (with **combinational group propagate module**) and do simulation test.
4. Design 32-bit **hybrid carry adder** and do simulation and physical verification.
5. Design **ALU module** by redundancy design and do physical verification.

2.2 Principles

Full Adder

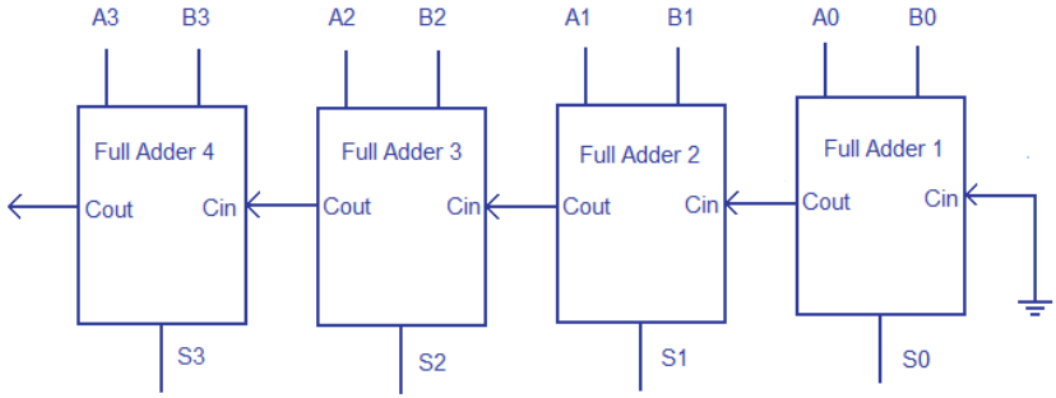
An adder is a digital circuit that performs addition of numbers. In many computers and other kinds of processors adders are used in the ALU.

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



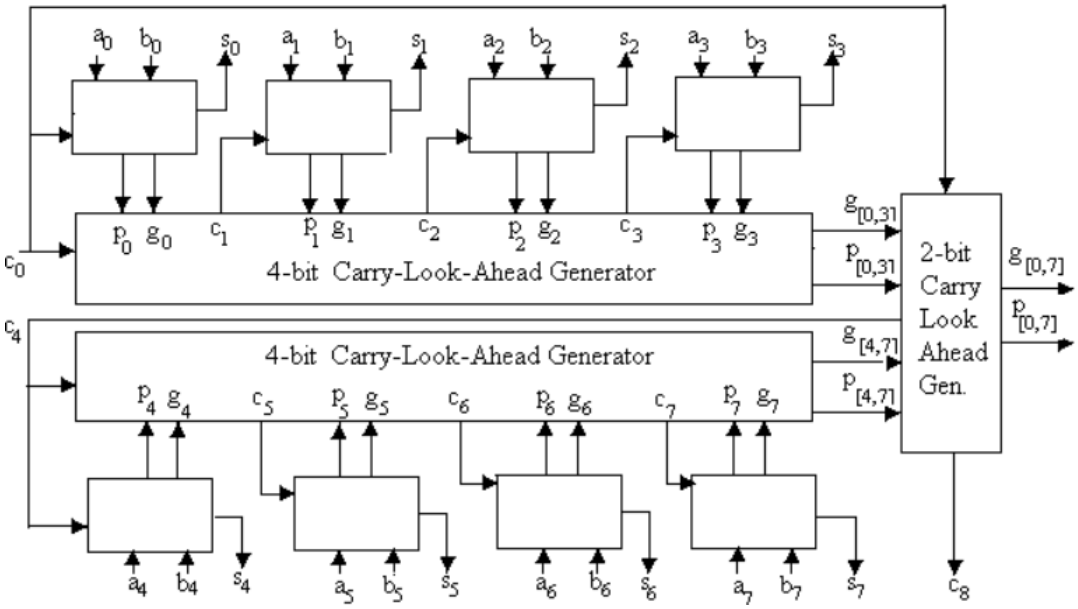
Design of Multibit Ripple Carry Adder

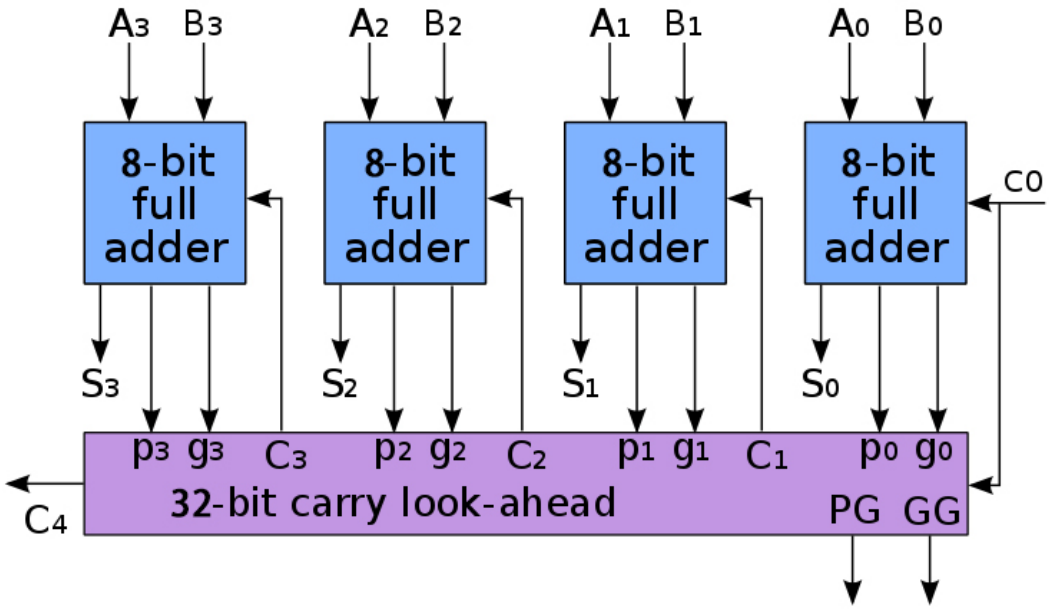
A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the succeeding next most significant full adder.



32-bit Full Adder

We implement this module by invoking the previously invoked modules: 1-bit full adders and then 8-bit ripple carry adders.



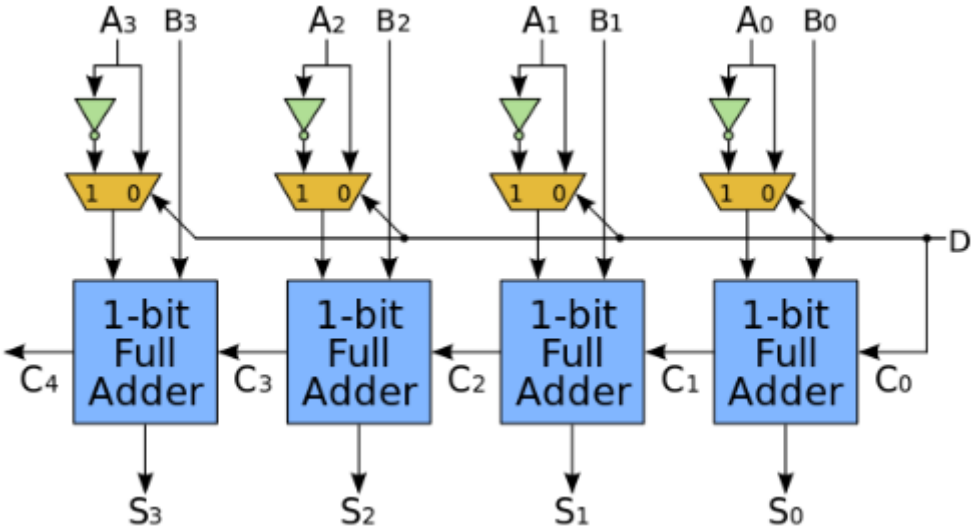


Multibit Ripple Carry Adder-Subtractor

The Adder-Subtractor performs

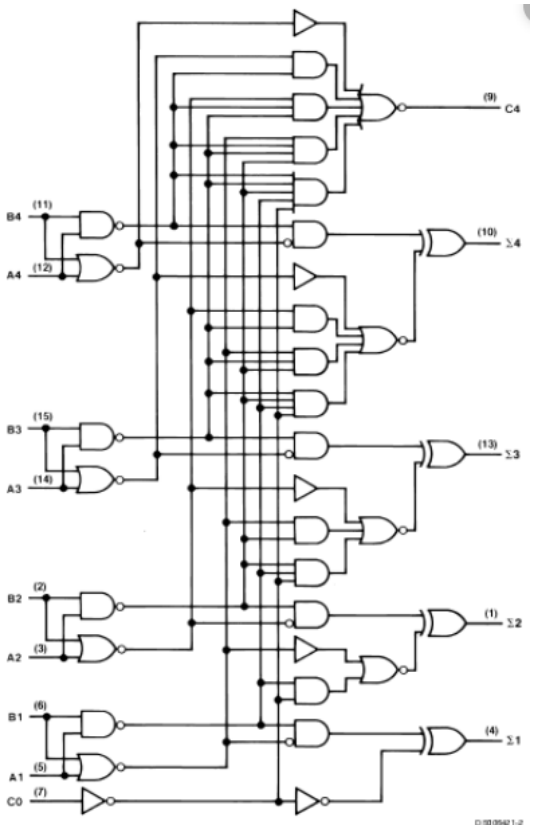
- addition when $D = 0$, or
- subtraction when $D = 1$.

This works because when $D = 1$ the A input to the adder is really $\bar{A}$ and the carry in is 1. Adding B to $\bar{A}$ and 1 yields the desired subtraction of $B - A$.



Carry Look-ahead Adder

The carry-lookahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder.



3. Major Experiment Instruments

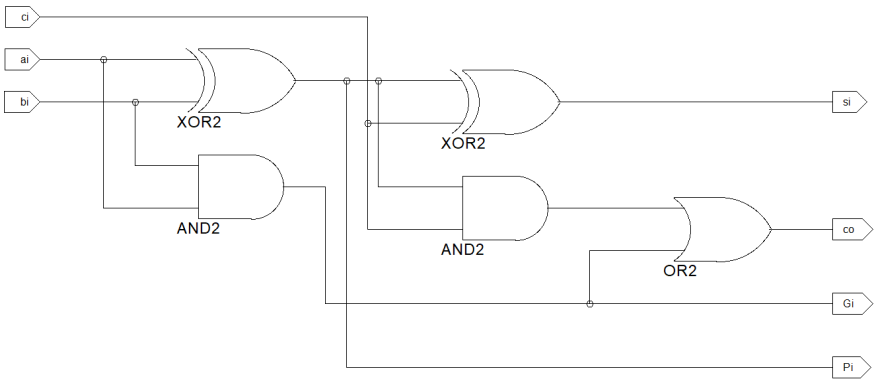
Equipment
Computer (Intel Core i7-9750H, 16GB memory)
Sword circuit design box
Xilinx ISE 14.7

4. Experiment Procedure

Task1: Exp081-ADC32

Implementation on 1-bit Adder

Use schematic to implement (**add.sch**). Draw the schematic as follows:



Simulation on 1-bit Adder

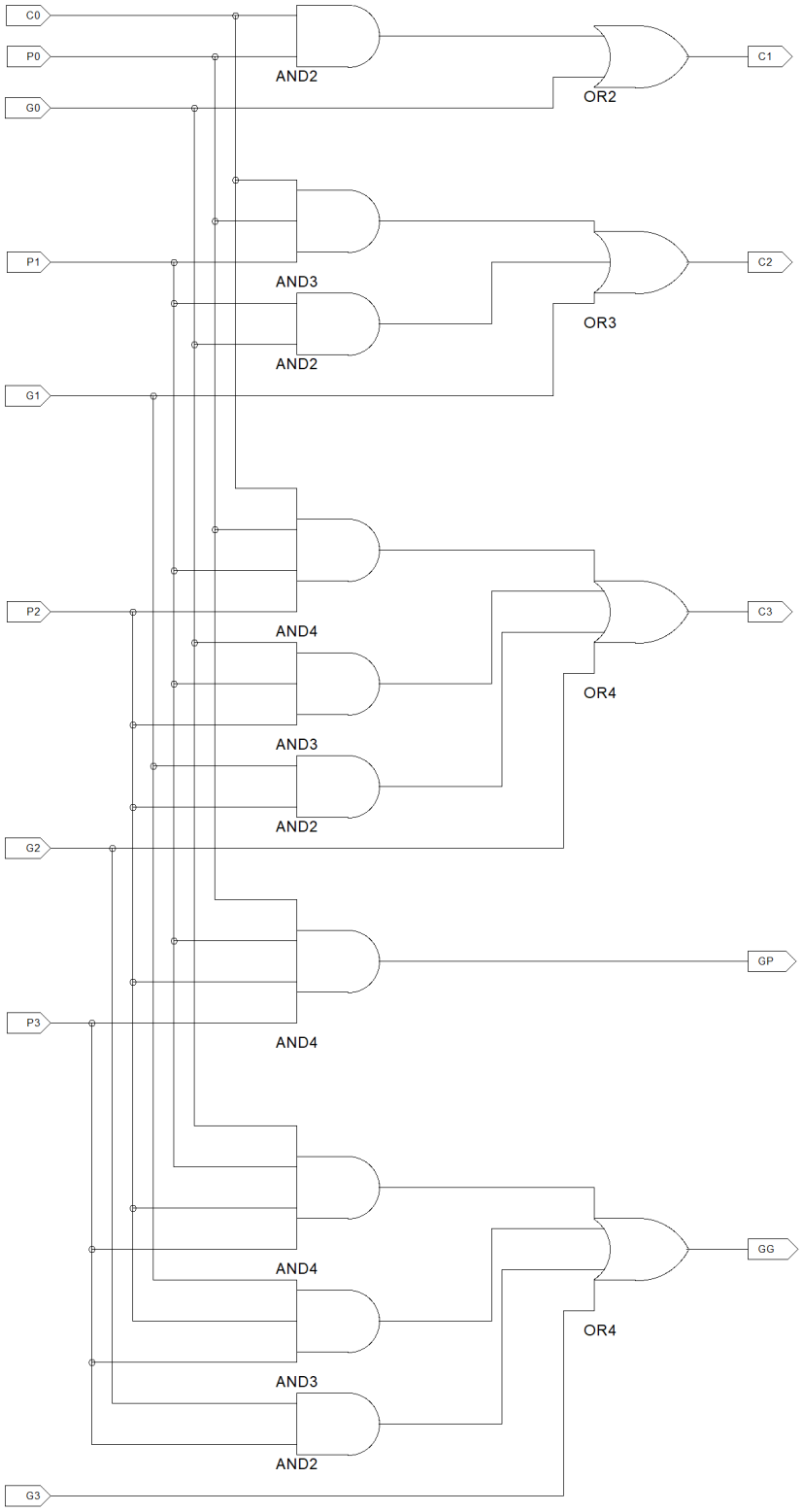
Using the following code to do simulation verification:

```
`timescale 1ns / 1ps
module add_add_sch_tb();

 // Inputs
reg ai;
reg bi;
reg ci;

// Output
wire Pi;
wire Gi;
wire co;
wire si;

// Instantiate the UUT
add UUT (
        .ai(ai),
        .bi(bi),
        .Pi(Pi),
        .ci(ci),
        .Gi(Gi),
        .co(co),
        .si(si)
);

// Initialize Inputs
    integer i = 0;
    initial begin
        ai = 0;
        bi = 0;
        ci = 0;

```

```
        for(i=0;i<=7;i=i+1)
        begin
            #50;
            {ci,ai,bi} = i+1;
        end
    end

endmodule
```

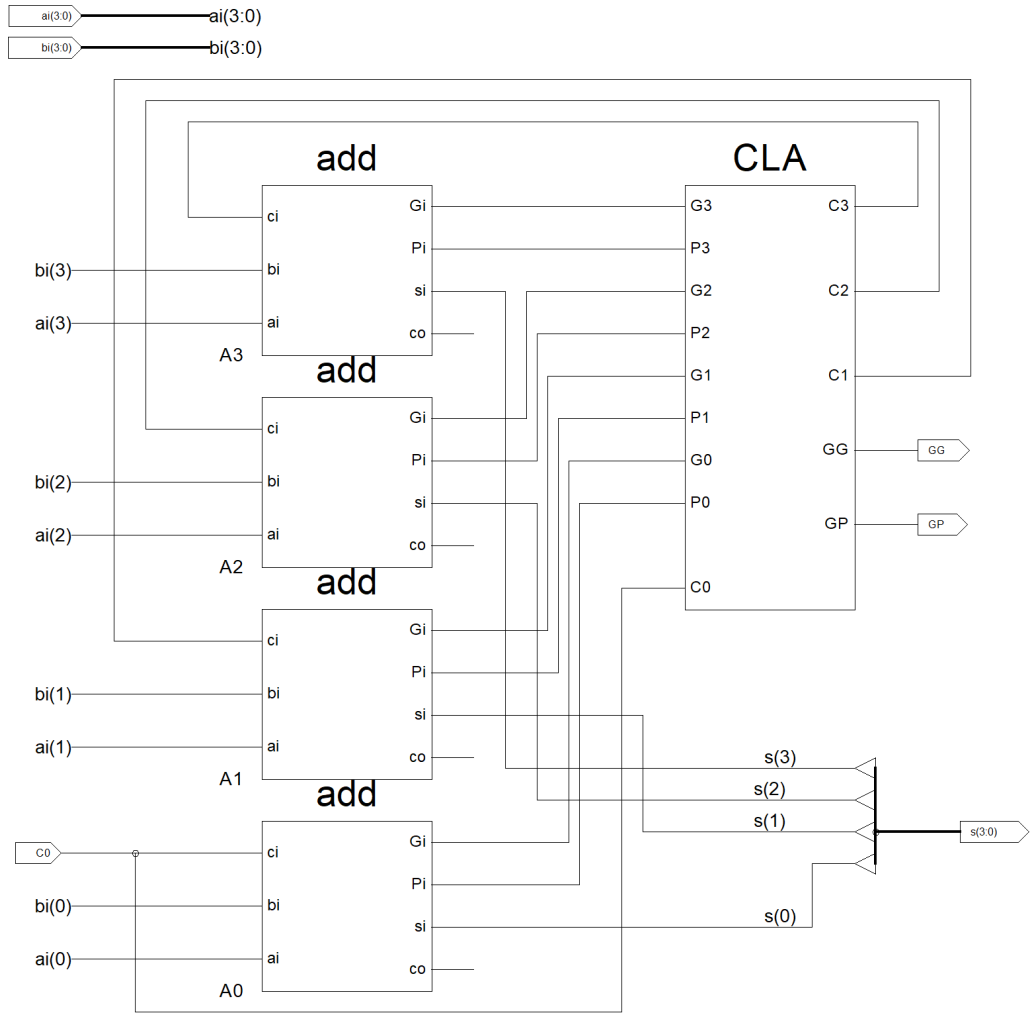
Implement 4-bit Carry Look-ahead Module

Create a new source file **CLA.sch** and draw following schematic:



Expand to 4-bit Carry Look-ahead Adder

New source file **add4b.sch**. Draw the schematic as follows:



Simulation on 4-bit Carry Look-ahead Adder

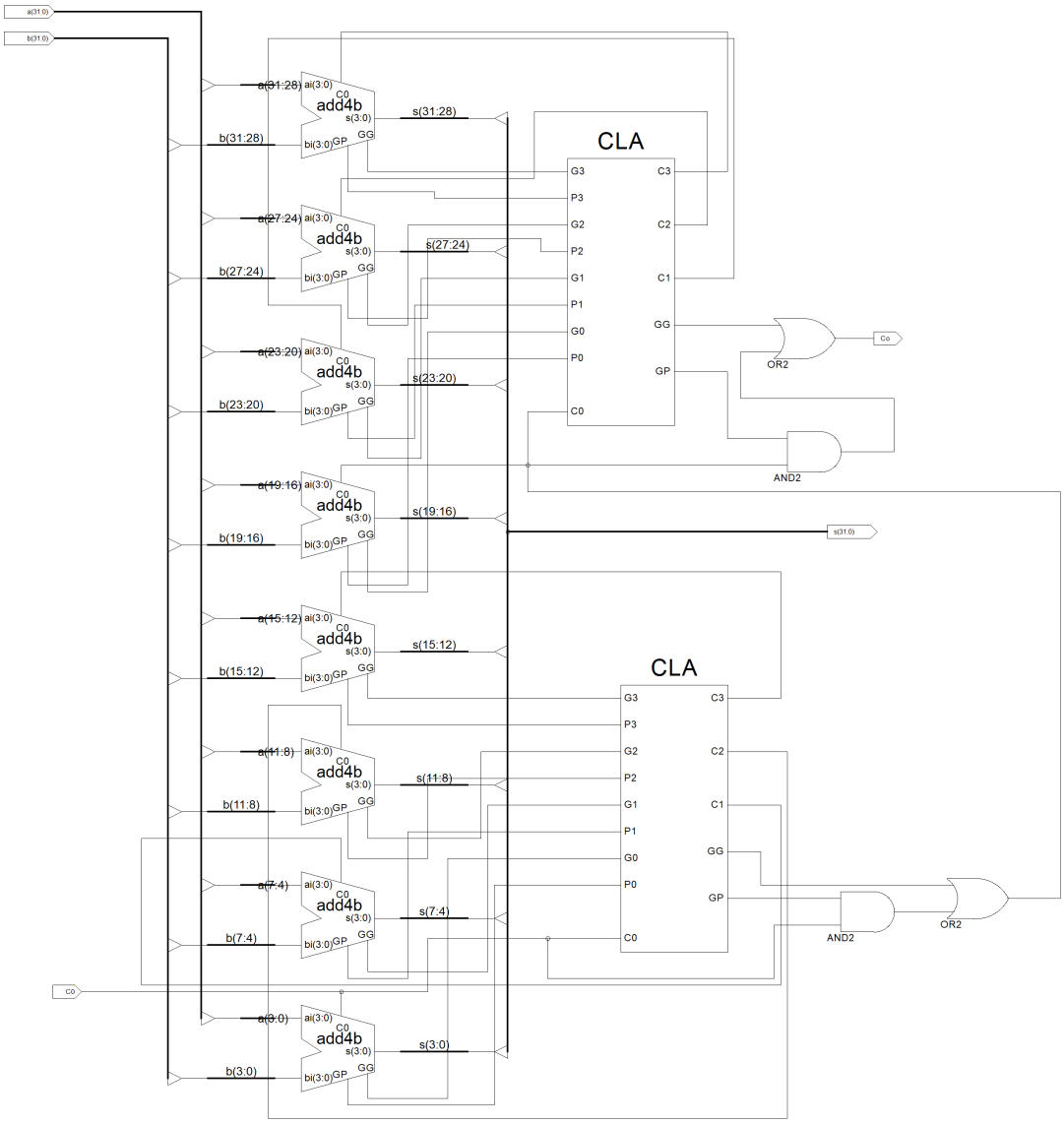
Using the following code to perform the simulation verification:

```
`timescale 1ns / 1ps
module add4b_add4b_sch_tb();

// Inputs
reg [3:0] ai;
reg [3:0] bi;
reg C0;

// Output
wire GG;
wire GP;
wire [3:0] s;

// Bidirs

// Instantiate the UUT
add4b UUT2 (
        .ai(ai),
        .bi(bi),
        .C0(C0),
        .GG(GG),
        .GP(GP),
        .s(s)
```

```
);
// Initialize Inputs

    integer i = 0, j=0;
    initial begin
        ai = 0;
        bi = 0;
        c0 = 0;

        for(i=0;i<=15;i=i+1)
        begin
            #50;
         ai = i+1;
         for(j=0;j<=15;j=j+1) begin
             #50;
             bi = j+1;
         end
        end
    end

endmodule
```

Implement 32-bit Full Adder

Create a new schematic **ADC32.sch**:



Partial Simulation on 32-bit Full Adder

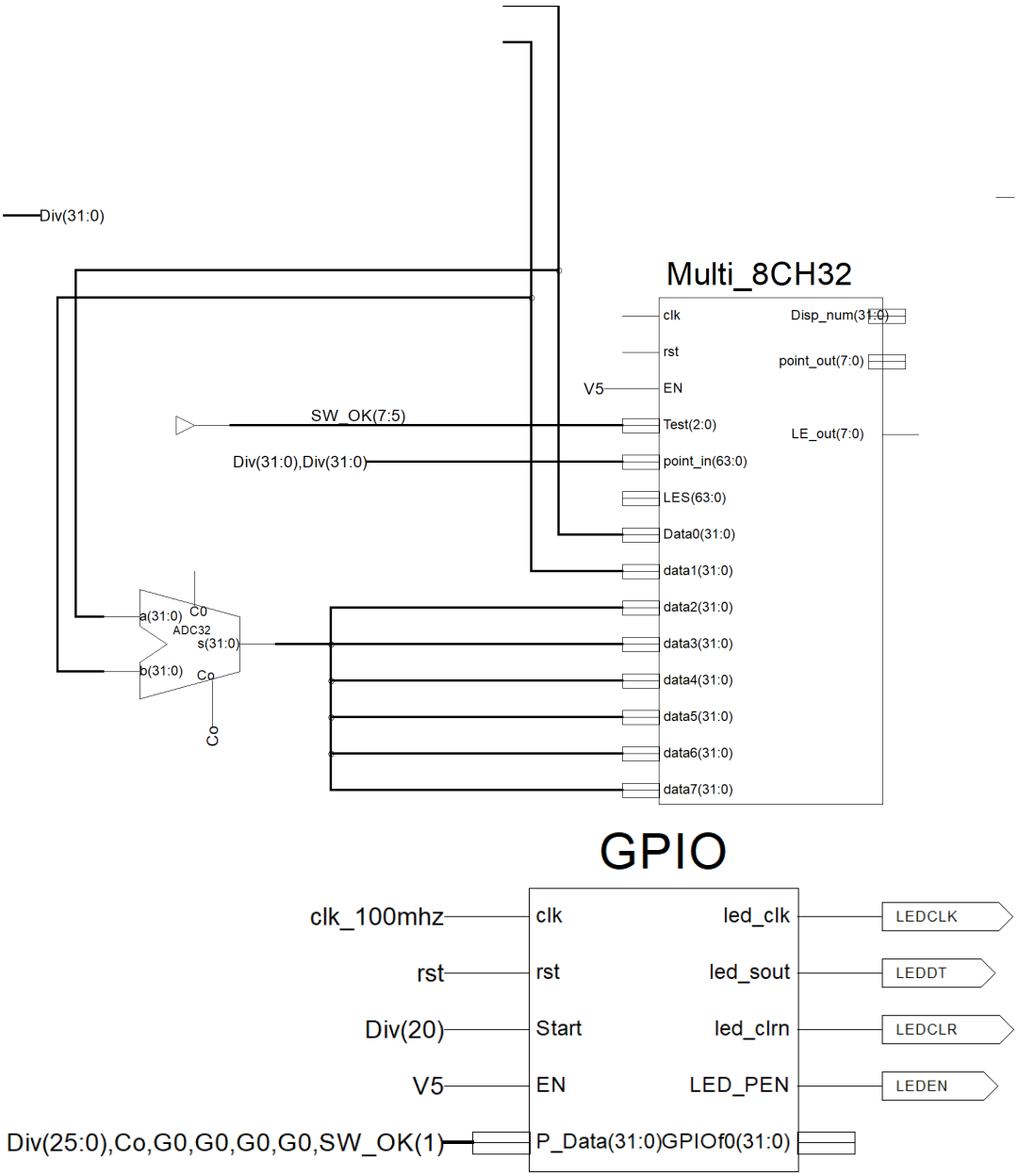
It's basically impossible to traverse all the input into the 32-bit full adder. Otherwise it will be a huge amount of data to be deal with. Therefore, I use only one group of data for simplicity.

```
`timescale 1ns / 1ps
module ADC32_ADC32_sch_tb();
// Inputs
reg [31:0] a;
reg [31:0] b;
reg C0;

// Output
wire Co;
wire [31:0] s;

// Bidirs

// Instantiate the UUT
    ADC32 UU3 (
        .a(a),
        .b(b),
        .C0(C0),
        .s(s),
        .Co(Co)
);
// Initialize Inputs

    initial begin
        a = 32'hAAAA;
        b = 32'h5555;
        C0 = 0;
        #50;
        a = 32'hFFFF;
        b = 32'h0000;
        C0 = 0;
        #50;
    end

endmodule
```

Physical Test

Using the **Framework_I/O** design implemented in lab07.

- SW[7:5]=000 is the first operand.
- SW[7:5]=001 is the second operand.
- SW[7:5]=100 displays the result.
- SW[15] controls the way of inputting.
- SW[0] controls graphic or digital display.



Task2: Exp082-ALU

During this task we are going to implement the while ALU(*AND, Add, Subtract, Or and Compare*).

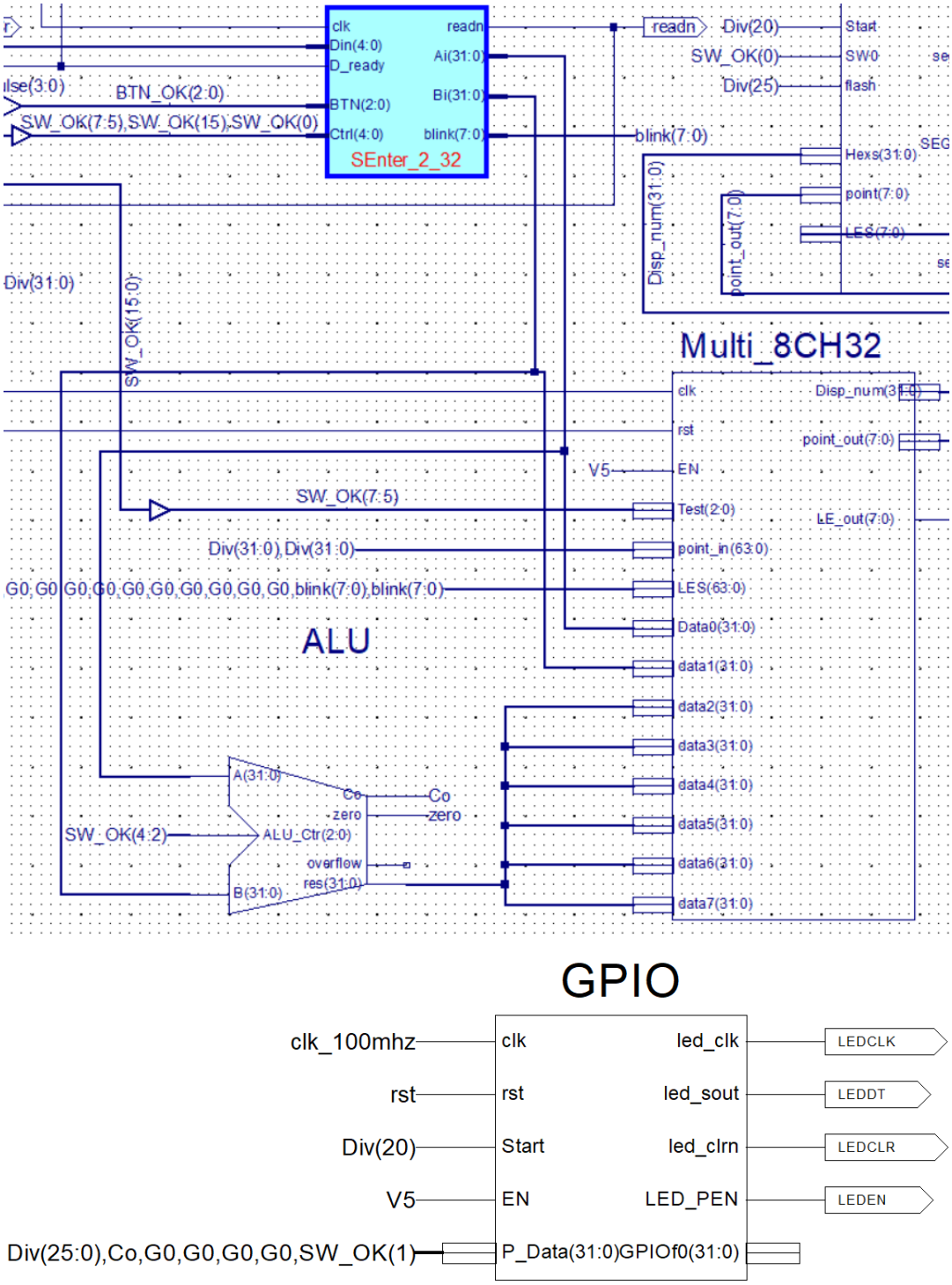
For simplicity, we implement ALU by verilog code:

```
`timescale 1ns / 1ps
module ALU(
    input [31:0] A,
    input [31:0] B,
    input [2:0] ALU_Ctr,
    output [31:0] res,
    output Co,
    output zero,
    output overflow
);

    wire [31:0] Sum, Bo, And, Or, Slt;
    wire sub = ALU_Ctr[2];
```

```
    assign Bo = B ^ {32{sub}};

 ADC32 ADD_32(              //Adder-Subtractor
        .a(A),
        .b(Bo),
        .C0(sub),
        .s(Sum),
        .Co(Co)
        );

    assign And = A & B;          //And
    assign Or = A | B;           //Or
    assign Slt = A < B? 1:0;     //Compare

 MUX8T1_32 MUX1(                 //Mux
 .I0(And),
 .I1(Or),
 .I2(Sum),
 .I3(32'hA5A5A5A5),
 .I4(32'hA5A5A5A5),
 .I5(32'h5A5A5A5A),
 .I6(Sum),
 .I7(Slt),
    .S(ALU_Ctr),
    .o(res)
    );

endmodule
```

Physical Test

We put this module in I/O framework.



5. Experiment Results and Analyses

Task1: Exp081-ADC32

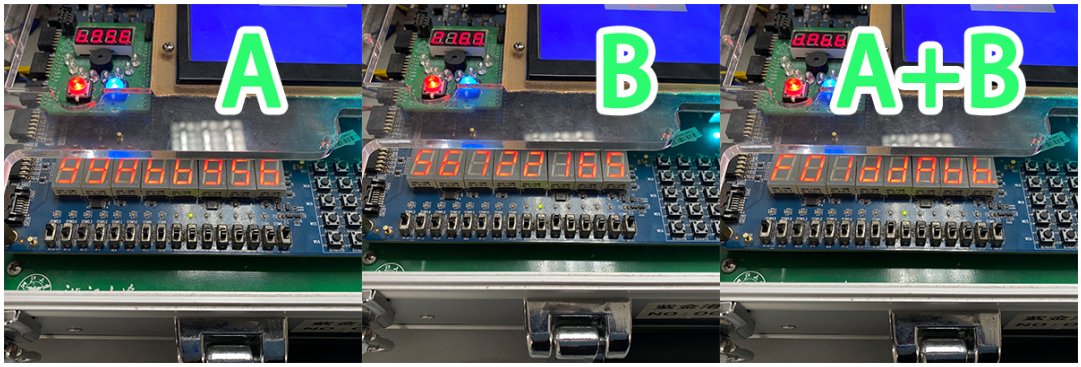
I did partial test with one test case.

First **set SW[15] to 1** to select the way of inputting. Then set **SW[0] to 1** to set digital display.

Enter the first operands within channel one (SW[7:5]=000), the then the second operands within channel two (SW[7:5]=001).

Then set SW[7:5]=100 to display the outcome.

This is a test case:

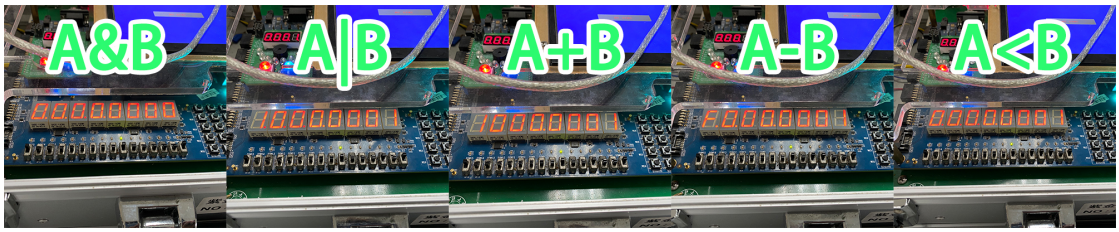


Task2: Exp082-ALU

First set **SW[0] to 1** to display digits and **SW[15] to 1** to change to the way of inputting. Then I input 00000001 to the first operand whose **control mode is SW[7:5]=000**, and also 10000000 to the second operand whose **control mode is SW[7:5]=001**.

Then we can output the result based on the following modes:

- **SW[4:2]=000**: A&B
- **SW[4:2]=001**: A|B
- **SW[4:2]=010**: A+B
- **SW[4:2]=110**: A-B
- **SW[4:2]=111**: A<B?



On top of the tests above, I also verify other functionality like selecting one digit and incrementing or decrementing to its value.

6. Discussion and Conclusion

Obviously, this experiment doesn't consume us much but depends on the **input and output framework** in Exp07. And it's also our first time to use the input and output framework in an experiment.

During this experiment, I met with one complication that at first time **I tried using verilog code to implement the CLA module**, writing several gate descriptions and combining them. Yet whatever I tried I found the simulation doesn't work correctly. And when a classmate asked me to help him debug his verilog code I found the similar problem. So from my conjectural thinking, there may be some difference between schematic and verilog implementation. And if conditions are allowed I will try find the essential cause and how to solve it.