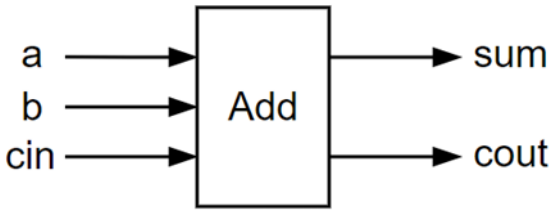
课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2024.1.19

实验一 加法器

一、实验目的

加法器是数字系统最基础的计算单元，用于产生两个数的和，加法器以二进制做运算，负数可以用二的补码来表示。减法器也是加法器，乘法器可以由加法器与移位器实现。由于加法器和乘法器会频繁使用，因此加法器的速度也影响着整个系统的计算速度。

使用不同方法设计 32bit 加法器：



A. 行波进位加法器，该加法器由一系列全加器级联而成。

B. 流水线加法器，流水线级数可以自己设定。

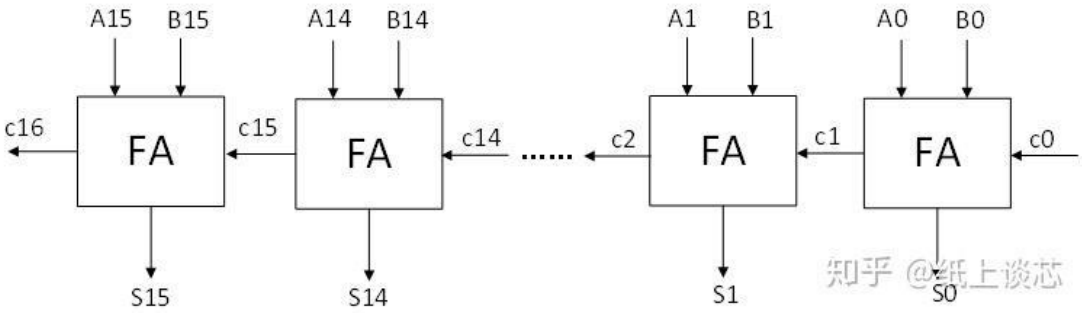
C. 超前进位加法器。思考如何优化超前进位加法器性能？

二、实验原理与设计思路

（1）32 位行波进位加法器

实验原理：

N-bit 加法器可以根据 1-bit 全加器组合而成。每个全加器的输出进位 cout 作为下一个全加器的输入进位 cin，这种加法器称为行波进位加法器。如一个 16 比特加法器的结构如下图所示，其中 A,B 为 16 比特的加数，S 为 A+B 的和，c16 为该加法器的输出。



由上图可以看出得到进位 c16 的结果依赖于 c15，c14，c13，…，c2，c1，c0，对于 32-bit，64-bit，128-bit 等加法器，进位链将显得更加长。所以，行波进位加法器设计简单，只需要级联全加器即可，但它的缺点在于超长的进位链，限制了加法器的性能。

设计思路：

输入两个 32 位的加数和一位进位，输出进位和 32 位的和。进位链 c 用于存储每一位的进位，从最低位到最高位一共 33 位（c[32] 用作输出进位）。由于需要实例化的模块数量较多，我们采用 generate 块来生成模块，在 generate 块内使用 for 循环生成 32 个一位全加器，并将输入进位与 c[0]捆绑，简化代码逻辑。

（2）四级流水线加法器

实验原理：

四级流水线加法器是一种通过将加法操作分成多个阶段来提高加法器性能的设计方法。流水线的基本思想是将一个长时间的计算过程分割成多个短时间的阶段，每个阶段可以并行处理不同的数据，从而提高总体的吞吐量和运算速度。对于 32 位的加法器，四级流水线加法器通过将加法操作分为四个阶段（流水线级），在每个阶段执行一个部分的加法操作，逐步完成 32 位加法。

四级流水线加法器的工作过程：

- 第一级：缓存输入并处理最低 8 位的加法。该阶段会将输入的 a 和 b 中的低 8 位进行加法运算，并计算出进位 c_1。
- 第二级：处理第 9-16 位，将第一阶段的加法结果和进位传递到第二阶段，再进行第二级的加法运算，得到 16 位的和及进位 c_2。
- 第三级：处理第 17-24 位，计算 16 位的和，并传递前一阶段的进位结果 c_3。
- 第四级：处理第 25-32 位，对剩下的 8 位进行加法，得到最终的和 s 和进位输出 cout。

设计思路：

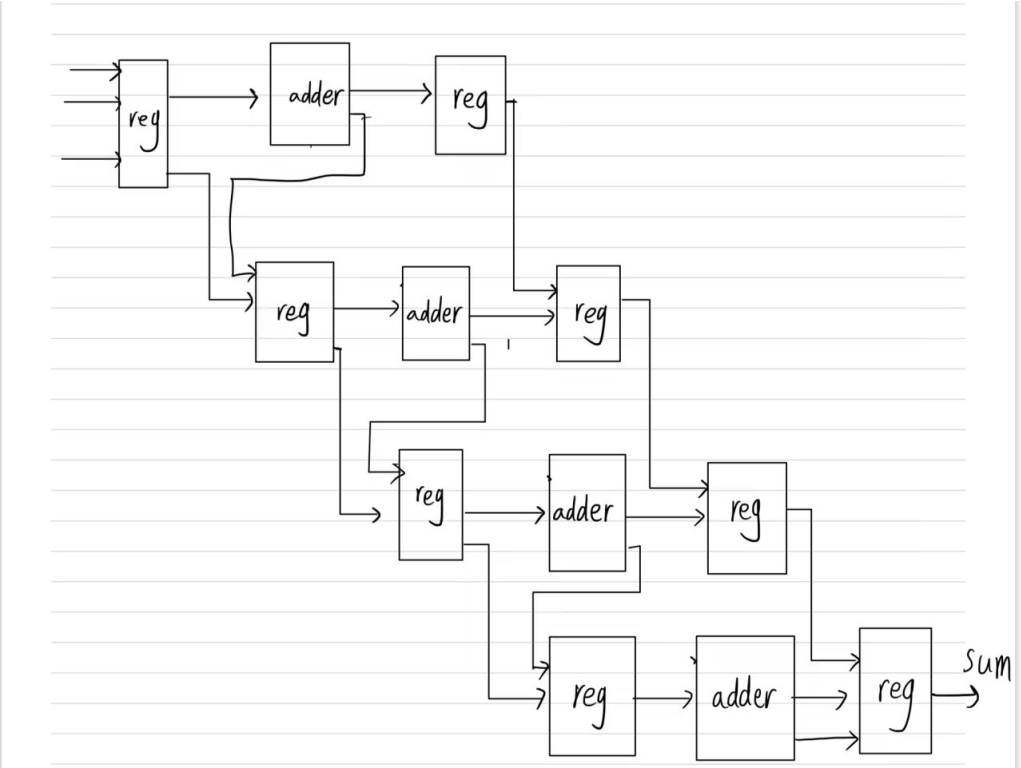
电路原理图如下

- 第一级负责接收输入 a 和 b 的值，存入触发器，并将其最低 8 位传递到第一级进行加法操作，同时处理进位输入。通过寄存器保存中间结果，确保每个时钟周期后数据能传递给下一个阶段。

• 第二级输入触发器接受前一级触发器的内容，并舍弃低八位，同时接收第一级的进位输出，对第二级输入触发器第 1 到第 8 位（原始数据的第 9 到 17 位）进行加法，同时处理相应的进位传递。

• 第三级在第二级结果的基础上进行计算，并将进位传递给下一阶段。

• 第四级继续计算并合并各级计算结果，输出最终的和和进位。



通过将 32 位加法操作分成四个流水线阶段，四级流水线加法器能够有效地提高运算效率。每个阶段在不同的时钟周期内并行计算不同部分的加法，最终将所有部分结果合并，输出最终的和和进位。这种设计不仅减少了单个加法操作的延迟，也为大规模数据处理提供了更高的吞吐量和并行能力。

（3）超前进位加法器

实验原理：

如果将位数从一位扩展到多位，高位进位信号总是依赖低位进位信号的，这就会造成非常可观的延迟，运算速度与进位传递速度有关。人们开始想办法优化电路结构，提取其共有部分：

$$\begin{cases} G = ab \\ P = a \oplus b \end{cases}$$

这里的 a 与 b 是输入的待求和数据，即可以是 1bit 的也可以是 N bits 的数据。于是，对于 N 位超前进位加法器，进位与求和的逻辑表达式可以重新写成如下形式：

$$\begin{cases} s_i = Pi \oplus c_{i-1}, & i = 1,\dots,N \\ c_i = G_i \oplus c_{i-1}P_i, & i = 1,\dots,N \\ c_0 = c_{in} \\ c_{out} = c_N \end{cases}$$

其中

$$\begin{cases} P_i = a_i \oplus b_i, & i = 0,1,\dots,N-1 \\ G_i = a_i b_i, & i = 0,1,\dots,N-1 \end{cases}$$

4bitCLA 利用上面产生的 P 与 G 信号，生成进位信号 c，其内部执行了下述操作：

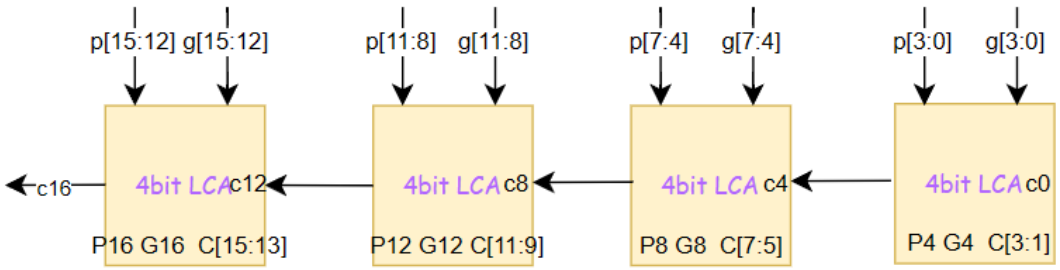
$$\begin{aligned} c_1 &= G_1 + c_0P_1 \\ c_2 &= G_2 + c_1P_2 = G_2 + (G_1 + c_0P_1)P_2 = G_2 + G_1P_2 + c_0P_2P_1 \\ c_3 &= G_3 + c_2P_3 = G_3 + (G_2 + G_1P_2 + c_0P_1P_1)P_3 = G_3 + G_2P_3 + G_1 \\ c_4 &= G_4 + c_3P_4 = G_4 + (G_3 + G_2P_3 + G_1P_3P_2 + c_0P_3P_2P_1)P_4 = G_4 \end{aligned}$$

最后的和数由进位传递信号 P 与进位信号 c 异或得到，注意第四位进位信号 c4 是高位进位，用来传递给下一级电路的，其计算逻辑如下

$$\begin{cases} s_0 = P_0 \oplus c_0 \\ s_1 = P_1 \oplus c_1 \\ s_2 = P_2 \oplus c_2 \\ s_3 = P_3 \oplus c_3 \end{cases}$$

如果涉及到大位数的超前进位加法器，还是使用上面的方法构建的话，会在进位链中引入超高扇入的逻辑门，使得整体电路规模以指数级增长。

为了控制超前进位电路的进位链复杂度，我们可以将一条较大的完整的超前进位链划分为多块较小的超前进位链块，也就是将 16bitCLA 分成 4 组 4bitCLA，如下图所示：



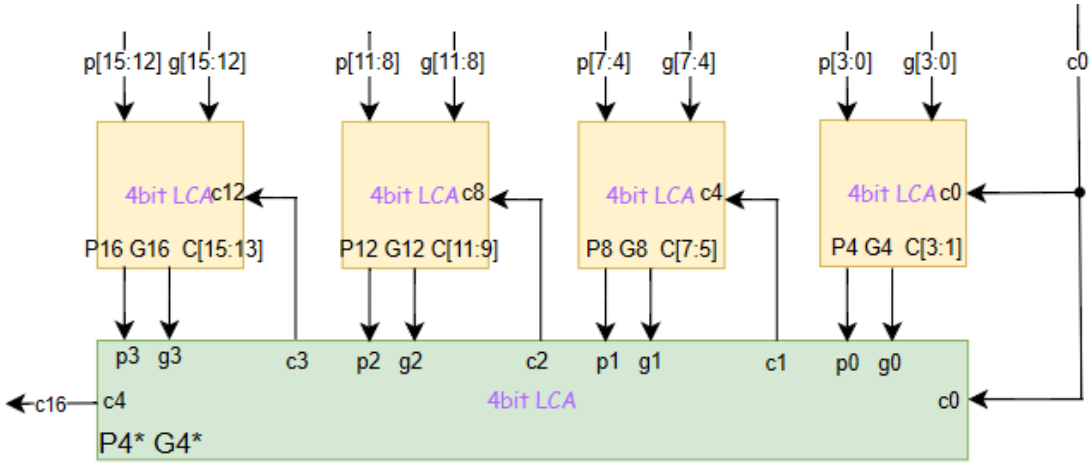
这就是最简单的组内超前进位、组间串行进位，也就是说每一个4bitCLA内部是超前进位的，而4组4bitLCA之间是串行进位的，c4、c8、c12与c16的产生不是并行的，c4的产生依赖于第一组4bitLCA的c0、c8的产生依赖于第二组4bitLCA的c4，以此类推。

同时，我们发现组间也有并行进位，c4也能拆开为超前进位的形式，组内、组间并行进位链的逻辑形态完全相同。

$$\begin{cases} G_4^* = G_4 + G_3P_4 + G_2P_4P_3 + G_1P_4P_3P_2 \\ P_4^* = P_4P_3P_2P_1 \end{cases}$$

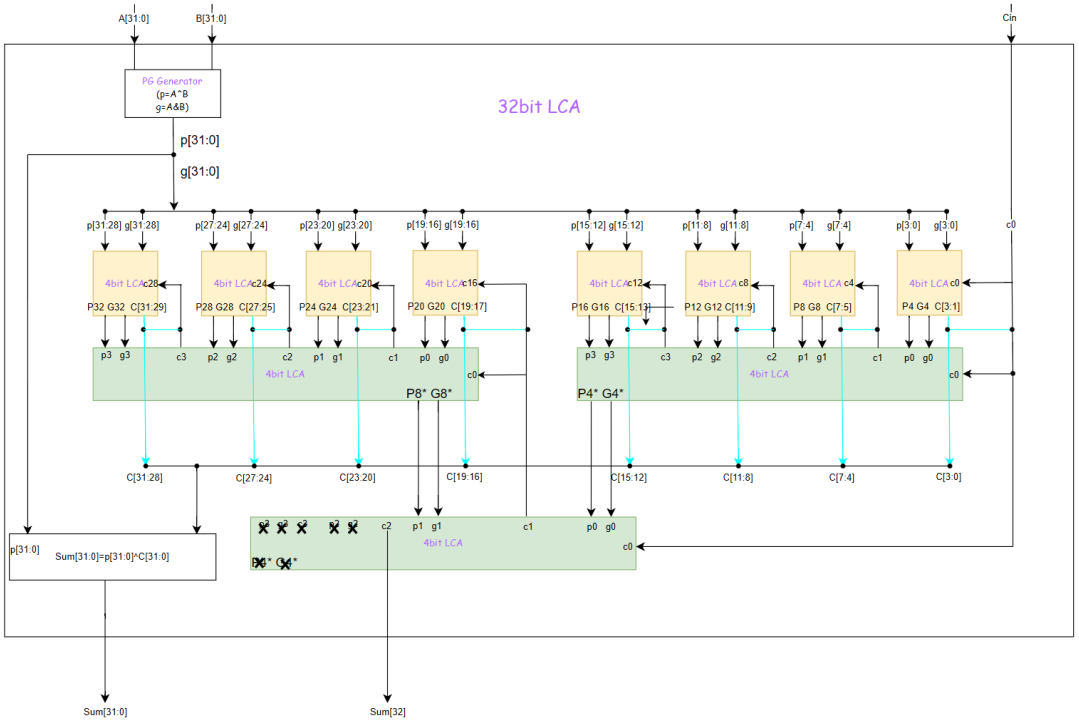
$$c_4 = G_4^* + P_4^* c_0$$

处理组间并行进位时可以完全复用原有的4bit CLA电路，只需要将原有4bitCLA模块中接入G4*、G8*、G12*、G16*、P4*、P8*、P12*、P16*即可产生之前需要串行等待的c4、c8、c12与c16，降低了延时的同时也没有增加额外的设计。



同理，扩展到32bitCLA的设计

课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2024.1.19



设计思路：

- 编写四位超前进位加法器的子模块 add_tc_4_4，输入 4 位 p，g，cin，输出 3 位 cout，和 P、G（用于实现组间的超前进位），使用纯组合电路编写。
- p、g 由 assign 直接产生，与输入 AB 直接相关。
- 实例化三层超前进位加法器，第一层的 cin 由第二层的 cout 给出，同理第二层的 cin 由第三层的 cout 给出。同时每一层的 p，g 依赖于前一层的 P，G 输出。

二、编码

（1）32 位行波进位加法器

RTL 设计代码：

```
module ripple_carry(
            input [31:0] a,
            input [31:0] b,
            input c_in,
            output [31:0] sum,
            output c_out);
    wire [32:0] c;
```

```
    assign c[0]=c_in;
    assign c_out=c[32];
     genvar i;
    generate
        for (i=0;i<32;i=i+1)begin
            full_adder u(a[i],b[i],c[i],sum[i],c[i+1]);
        end
    endgenerate
endmodule
module full_adder(
            input a,
            input b,
            input c_in,
            output sum,
            output c_out);
    assign c_out = a^b^c_in;
    assign sum = (a&b)|(b&c_in)|(a&c_in);

endmodule
```

Testbench 代码：

```
`timescale 1ns/1ps
module ripple_carry_tb;
    // 输入信号定义
    reg [31:0] a;                  // 32 位输入 a
    reg [31:0] b;                  // 32 位输入 b
    reg c_in;                       // 进位输入
    // 输出信号定义
    wire [31:0] sum;               // 32 位和
    wire c_out;                     // 进位输出
    // 实例化 ripple_carry 模块
    ripple_carry uut (
        .a(a),
        .b(b),
        .c_in(c_in),
        .sum(sum),
        .c_out(c_out)
    );
    // 测试用例
    initial begin
        // 初始化信号
        a = 32'b0;                  // 默认 a 为 0
```

```
        b = 32'b0;                  // 默认 b 为 0
        c_in = 1'b0;                // 默认进位输入为 0

        // 等待一段时间后进行初始化输出显示
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 1: a = 1, b = 1, c_in = 0
        a = 32'b00000000000000000000000000000001;
        b = 32'b00000000000000000000000000000001;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 2: a = 2, b = 3, c_in = 0
        a = 32'b00000000000000000000000000000010;
        b = 32'b00000000000000000000000000000011;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 3: a = 10, b = 5, c_in = 0
        a = 32'b00000000000000000000000000001010;
        b = 32'b00000000000000000000000000000101;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 4: a = 15, b = 20, c_in = 1 (with carry in)
        a = 32'b00000000000000000000000000001111;
        b = 32'b00000000000000000000000000010100;
        c_in = 1'b1;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 5: a = MAX, b = MAX, c_in = 1 (edge case)
        a = 32'b11111111111111111111111111111111;
        b = 32'b11111111111111111111111111111111;
```

```
        c_in = 1'b1;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 6: a = 0, b = 0, c_in = 1 (edge case)
        a = 32'b00000000000000000000000000000000;
        b = 32'b00000000000000000000000000000000;
        c_in = 1'b1;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 7: a = 255, b = 255, c_in = 0
        a = 32'b00000000000000000000000011111111;
        b = 32'b00000000000000000000000011111111;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Test case 8: a = 100, b = 150, c_in = 0
        a = 32'b00000000000000000000000001100100;
        b = 32'b00000000000000000000000010010110;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %b, b = %b, c_in = %b, sum = %b, c_out = %b", $time, a, b, c_in,
sum, c_out);

        // Finish simulation
        $finish;
    end
initial begin
$fsdbDumpfile("a.fsdb");
$fsdbDumpvars();
end
endmodule
```

（2）四级流水线加法器

RTL 设计代码：

课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2024.1.19

```
module pipe_line_adder(
    input rst_n,                // 复位信号
    input clk,                  // 时钟信号
    input cin,                  // 输入进位
    input [31:0] a,             // 输入 a
    input [31:0] b,             // 输入 b
    output reg cout,            // 输出进位
    output reg [31:0] s         // 输出和
);
    // 临时寄存器定义
    reg c_1, c_2, c_3, c_4;
    reg [31:0] a_1, b_1;
    reg [23:0] a_2, b_2;    // 第二级输入数据
    reg [15:0] a_3, b_3;    // 第三级输入数据
    reg [7:0] a_4, b_4;     // 第四级输入数据

    reg [7:0] s_1;      // 第一级和
    reg [15:0] s_2;     // 第二级和
    reg [23:0] s_3;     // 第三级和
 // 缓存输入并处理最低 8 位
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            c_1 <= 1'b0;
            a_1 <= 32'b0;
            b_1 <= 32'b0;
        end else begin
            c_1 <= cin;
            a_1 <= a;
            b_1 <= b;
        end
    end

    // 第一级：处理第 1-8 位，得到 8 位的和及进位
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            c_2 <= 1'b0;
            s_1 <= 8'b0;
            a_2 <= 24'b0;
            b_2 <= 24'b0;
        end else begin
            {c_2, s_1} <= a_1[7:0] + b_1[7:0] + c_1; // 低 8 位加法
            a_2 <= a_1[31:8];   // 高 24 位
            b_2 <= b_1[31:8];
```

```
        end
    end

    // 第二级：处理第 9-16 位，得到 16 位的和及进位
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            c_3 <= 1'b0;
            s_2 <= 16'b0;
            a_3 <= 16'b0;
            b_3 <= 16'b0;
        end else begin
            {c_3, s_2[15:8]} <= a_2[7:0] + b_2[7:0] + c_2; // 低 8 位加法
            a_3 <= a_2[23:8];   // 高 8 位
            b_3 <= b_2[23:8];
            s_2[7:0] <= s_1;      // 将第一级结果传递到 s_2 的低 8 位
        end
    end

    // 第三级：处理第 17-24 位，得到 24 位的和及进位
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            c_4 <= 1'b0;
            s_3 <= 24'b0;
            a_4 <= 8'b0;
            b_4 <= 8'b0;
        end else begin
            {c_4, s_3[23:16]} <= a_3[7:0] + b_3[7:0] + c_3; // 高 16 位加法
            a_4 <= a_3[15:8];   // 剩余的 8 位
            b_4 <= b_3[15:8];
            s_3[15:0] <= s_2;    // 将第二级结果传递到 s_3 的低 16 位
        end
    end

    // 第四级：处理最终的 32 位加法
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            cout <= 1'b0;
            s <= 32'b0;
        end else begin
            {cout, s[31:24]} <= a_4[7:0] + b_4[7:0] + c_4;   // 合并低 8 位
          s[23:0] <= s_3;   // 高 24 位直接传递
        end
    end
```

```
endmodule
```

Testbench 代码：

```
`TIMESCALE 1NS / 1PS

MODULE TB_PIPE_LINE_ADDER;

    // 输入信号
    REG RST_N;              // 复位信号
    REG CLK;                // 时钟信号
    REG CIN;                // 输入进位
    REG [31:0] A;           // 输入 A
    REG [31:0] B;           // 输入 B

    // 输出信号
    WIRE COUT;              // 输出进位
    WIRE [31:0] S;          // 输出和

    // 实例化流水线加法器
    PIPE_LINE_ADDER UUT (
        .RST_N(RST_N),
        .CLK(CLK),
        .CIN(CIN),
        .A(A),
        .B(B),
        .COUT(COUT),
        .S(S)
    );

    // 生成时钟信号
    ALWAYS BEGIN
        #5 CLK = ~CLK; // 每 5NS 反转一次时钟，周期 10NS
    END

    // 测试过程
    INITIAL BEGIN
        // 初始化信号
        CLK = 0;
        RST_N = 0;
        CIN = 0;
        A = 32'B0;
        B = 32'B0;
```

```
// 复位过程
#10;
RST_N = 1;// 解除复位

// 测试用例 1：简单加法
#10;
A = 32'H00000001;   // A = 1
B = 32'H00000001;   // B = 1
CIN = 0;            // 输入进位为 0
#10;
$DISPLAY("TEST CASE 1: A = %H, B = %H, CIN = %B, SUM = %H, COUT = %B", A, B, CIN, S, COUT);

// 测试用例 2：有进位的加法
#10;
A = 32'HFFFFFFFF;   // A = -1
B = 32'H00000001;   // B = 1
CIN = 0;            // 输入进位为 0
#10;
$DISPLAY("TEST CASE 2: A = %H, B = %H, CIN = %B, SUM = %H, COUT = %B", A, B, CIN, S, COUT);

// 测试用例 3：带有进位的加法
#10;
A = 32'H000000FF;   // A = 255
B = 32'H000000FF;   // B = 255
CIN = 1;            // 输入进位为 1
#10;
$DISPLAY("TEST CASE 3: A = %H, B = %H, CIN = %B, SUM = %H, COUT = %B", A, B, CIN, S, COUT);

// 测试用例 4：大数加法
#10;
A = 32'H7FFFFFFF;   // A = 2147483647
B = 32'H7FFFFFFF;   // B = 2147483647
CIN = 0;            // 输入进位为 0
#10;
$DISPLAY("TEST CASE 4: A = %H, B = %H, CIN = %B, SUM = %H, COUT = %B", A, B, CIN, S, COUT);

// 测试用例 5：全为 0 的加法
#10;
A = 32'B0;          // A = 0
B = 32'B0;          // B = 0
CIN = 0;            // 输入进位为 0
#10;
```

```
        $DISPLAY("TEST CASE 5: A = %H, B = %H, CIN = %B, SUM = %H, COUT = %B", A, B, CIN, S, COUT);

        // 测试用例 6：输入最大值的加法
        #10;
        A = 32'HFFFFFFFF;   // A = -1
        B = 32'HFFFFFFFF;   // B = -1
        CIN = 1;            // 输入进位为 1
        #10;
        $DISPLAY("TEST CASE 6: A = %H, B = %H, CIN = %B, SUM = %H, COUT = %B", A, B, CIN, S, COUT);

        // 测试结束
        $STOP;
    END

INITIAL BEGIN
      $FSDBDUMPFILE("PIPE_LINE_ADDER.FSDB");
      $FSDBDUMPVARS();
END

ENDMODULE
```

（3）超前进位加法器

RTL 设计代码：

```
`timescale 1ns/1ps
module add_tc_4_4(
      input    [3:0]    p,
      input    [3:0]    g,
      input    cin,

      output   P,        //组间进位信号（成组进位信号）
      output   G,        //组间进位信号（成组进位信号）
      output   [2:0]    cout      //第四位 cout 信号被拆分为 P 与 G
);

      assign P = p[3]&p[2]&p[1]&p[0];
      assign G = g[3]|(p[3]&g[2])|(p[3]&p[2]&g[1])|(p[3]&p[2]&p[1]&g[0]);

      assign cout[0] = g[0]|(p[0]&cin);
      assign cout[1] = g[1]|(p[1]&g[0])|(p[1]&p[0]&cin);
      assign cout[2] = g[2]|(p[2]&g[1])|(p[2]&p[1]&g[0])|(p[2]&p[1]&p[0]&cin);
```

```
endmodule

module carry_lookahead_adder
(
    input   [31:0]   A,
    input   [31:0]   B,
    input   Cin,
    output  [31:0]   Sum,
    output  Cout
);

    wire [31:0] p;
    wire [31:0] g;
    wire [31:0] c;
    wire c_x;

    wire [9:0]  P;
    wire [9:0]  G;

    assign c[0] = Cin;
    assign p = A^B;
    assign g = A&B;

    add_tc_4_4
add_tc_4_4_inst0_0(.p(p[3:0]),.g(g[3:0]),.cin(c[0]),.P(P[0]),.G(G[0]),.cout(c[3:1]));
    add_tc_4_4
add_tc_4_4_inst0_1(.p(p[7:4]),.g(g[7:4]),.cin(c[4]),.P(P[1]),.G(G[1]),.cout(c[7:5]));
    add_tc_4_4
add_tc_4_4_inst0_2(.p(p[11:8]),.g(g[11:8]),.cin(c[8]),.P(P[2]),.G(G[2]),.cout(c[11:9]));
    add_tc_4_4
add_tc_4_4_inst0_3(.p(p[15:12]),.g(g[15:12]),.cin(c[12]),.P(P[3]),.G(G[3]),.cout(c[15:13]));
    add_tc_4_4
add_tc_4_4_inst0_4(.p(P[3:0]),.g(G[3:0]),.cin(c[0]),.P(P[4]),.G(G[4]),.cout({c[12],c[8],c[4]}));

    add_tc_4_4
add_tc_4_4_inst1_0(.p(p[19:16]),.g(g[19:16]),.cin(c[16]),.P(P[5]),.G(G[5]),.cout(c[19:17]));
    add_tc_4_4
add_tc_4_4_inst1_1(.p(p[23:20]),.g(g[23:20]),.cin(c[20]),.P(P[6]),.G(G[6]),.cout(c[23:21]));
    add_tc_4_4
add_tc_4_4_inst1_2(.p(p[27:24]),.g(g[27:24]),.cin(c[24]),.P(P[7]),.G(G[7]),.cout(c[27:25]));
    add_tc_4_4
add_tc_4_4_inst1_3(.p(p[31:28]),.g(g[31:28]),.cin(c[28]),.P(P[8]),.G(G[8]),.cout(c[31:29]));
```

```
    add_tc_4_4
add_tc_4_4_inst1_4(.p(P[8:5]),.g(G[8:5]),.cin(c[16]),.P(P[9]),.G(G[9]),.cout({c[28],c[24],c[20]
}));

    add_tc_4_4
add_tc_4_4_inst2(.p({2'b00,P[9],P[4]}),.g({2'b00,G[9],G[4]}),.cin(c[0]),.P(),.G(),.cout({c_x,Co
ut,c[16]}));

    assign Sum[31:0] = p^c;

endmodule
```

Testbench 代码

```
`timescale 1ns/1ps
module ripple_carry_tb;
    // 输入信号定义
    reg [31:0] a;                // 32 位输入 a
    reg [31:0] b;                // 32 位输入 b
    reg c_in;                    // 进位输入
    // 输出信号定义
    wire [31:0] sum;             // 32 位和
    wire c_out;                  // 进位输出
    // 实例化 ripple_carry 模块
    carry_lookahead_adder uut (
        .A(a),
        .B(b),
        .Cin(c_in),
        .Sum(sum),
        .Cout(c_out)
    );
    // 测试用例
    initial begin
        // 初始化信号
        a = 32'b0;               // 默认 a 为 0
        b = 32'b0;               // 默认 b 为 0
        c_in = 1'b0;             // 默认进位输入为 0

        // 等待一段时间后进行初始化输出显示
        #10;
        $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);
```

```
        // Test case 1: a = 1, b = 1, c_in = 0
        a = 32'b00000000000000000000000000000001;
        b = 32'b00000000000000000000000000000001;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

        // Test case 2: a = 2, b = 3, c_in = 0
        a = 32'b00000000000000000000000000000010;
        b = 32'b00000000000000000000000000000011;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

        // Test case 3: a = 10, b = 5, c_in = 0
        a = 32'b00000000000000000000000000001010;
        b = 32'b00000000000000000000000000000101;
        c_in = 1'b0;
        #10;
        $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

        // Test case 4: a = 15, b = 20, c_in = 1 (with carry in)
        a = 32'b00000000000000000000000000001111;
        b = 32'b00000000000000000000000000010100;
        c_in = 1'b1;
        #10;
        $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

        // Test case 5: a = MAX, b = MAX, c_in = 1 (edge case)
        a = 32'b11111111111111111111111111111111;
        b = 32'b11111111111111111111111111111111;
        c_in = 1'b1;
        #10;
        $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

        // Test case 6: a = 0, b = 0, c_in = 1 (edge case)
        a = 32'b00000000000000000000000000000000;
```

```
            b = 32'b00000000000000000000000000000000;
            c_in = 1'b1;
            #10;
            $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

            // Test case 7: a = 255, b = 255, c_in = 0
            a = 32'b00000000000000000000000011111111;
            b = 32'b00000000000000000000000011111111;
            c_in = 1'b0;
            #10;
            $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

            // Test case 8: a = 100, b = 150, c_in = 0
            a = 32'b00000000000000000000000001100100;
            b = 32'b00000000000000000000000010010110;
            c_in = 1'b0;
            #10;
            $display("Time = %0t, a = %h, b = %h, c_in = %h, sum = %h, c_out = %h", $time,
a, b, c_in, sum, c_out);

            // Finish simulation
            $finish;
      end
initial begin
$fsdbDumpfile("a.fsdb");
$fsdbDumpvars();
end
endmodule
```

三、仿真验证

（1）32 位行波进位加法器

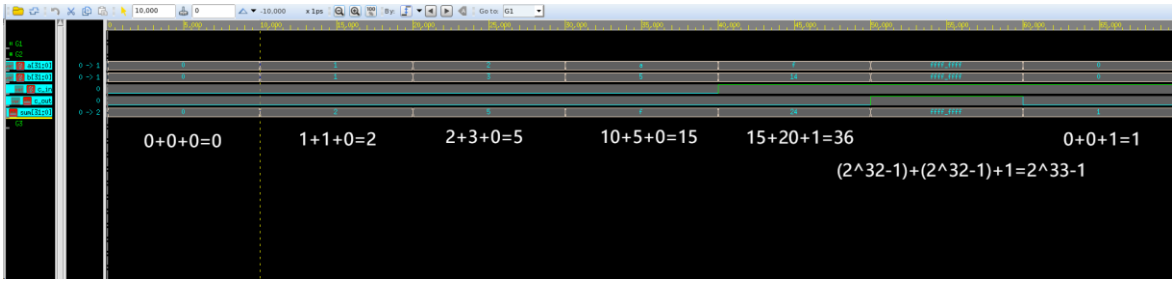
基本**加法**测试：测试两个相同大小的数相加，检查 sum 和 c_out 是否正确。

- 测试：a = 1，b = 1，c_in = 0
  预期输出：sum = 2，c_out = 0
- 示例：a = 2，b = 3，c_in = 0

预期输出：sum = 5，c_out = 0

边界值测试，测试输入值为零和两个最大值相加时的进位处理。

- 示例：a = 0，b = 0，c_in = 0
  预期输出：sum = 0，c_out = 0
- 示例：a = 0xffff_ffff，b = 0xffff_ffff，c_in = 1（最大 32 位无符号整数）
  预期输出：sum = 0xffff_ffff，c_out = 1
- 示例：a = 0，b = 0，c_in = 1
  预期输出：sum = 1，c_out = 0



由仿真结果可知，加法器逻辑正确。

（2）四级流水线加法器

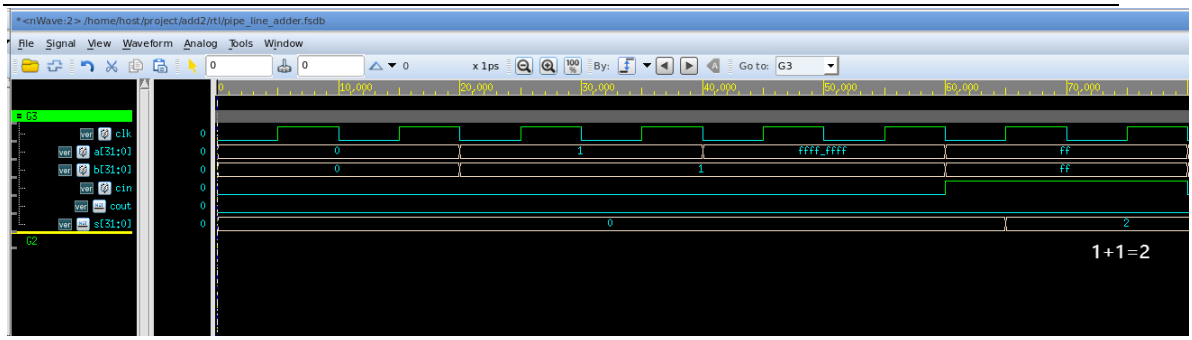
基本**加法**测试：测试两个相同大小的数相加，检查 sum 和 c_out 是否正确。

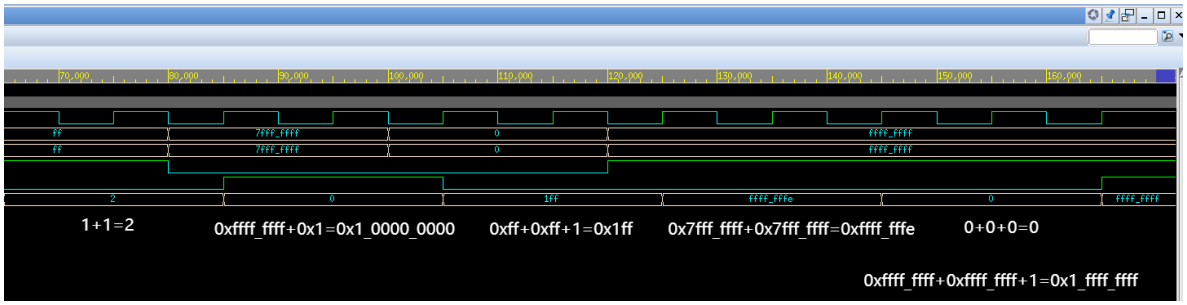
- 测试：a = 1，b = 1，c_in = 0
  预期输出：sum = 2，c_out = 0
- 示例：a = 0xffff_ffff，b = 1，c_in = 0
  预期输出：sum = 0，c_out = 1

边界值测试，测试输入值为零和两个最大值相加时的进位处理。

- 示例：a = 0，b = 0，c_in = 0
  预期输出：sum = 0，c_out = 0
- 示例：a = 0xffff_ffff，b = 0xffff_ffff，c_in = 1（最大 32 位无符号整数）
  预期输出：sum = 0xffff_ffff，c_out = 1

课程名称：数字集成电路设计　　开课学院：微电子学院
姓名：xxx　　学号：xxxxxx
报告提交日期：2024.1.19





由于采用流水线，加法器输出与输入信号相比延时四个时钟信号，由仿真结果可知，加法器逻辑正确。

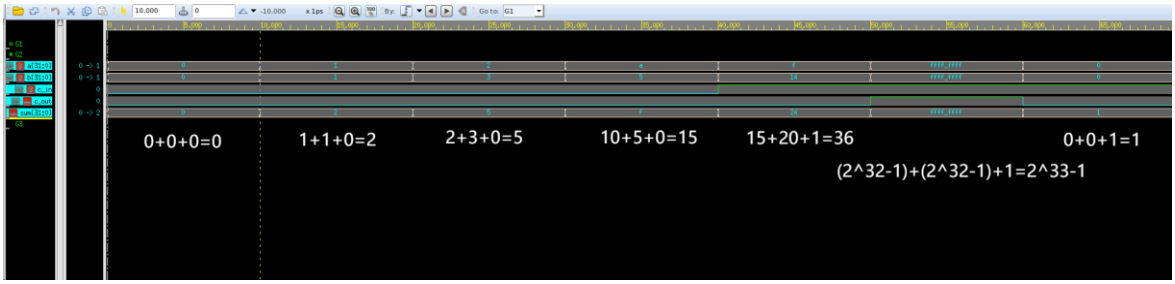
（3）超前进位加法器

基本**加法**测试：测试两个相同大小的数相加，检查 sum 和 c_out 是否正确。

- 测试：a = 1，b = 1，c_in = 0
  预期输出：sum = 2，c_out = 0
- 示例：a = 2，b = 3，c_in = 0
  预期输出：sum = 5，c_out = 0

边界值测试，测试输入值为零和两个最大值相加时的进位处理。

- 示例：a = 0，b = 0，c_in = 0
  预期输出：sum = 0，c_out = 0
- 示例：a = 0xffff_ffff，b = 0xffff_ffff，c_in = 1（最大 32 位无符号整数）
  预期输出：sum = 0xffff_ffff，c_out = 1
- 示例：a = 0，b = 0，c_in = 1
  预期输出：sum = 1，c_out = 0



由仿真结果可知，加法器逻辑正确。

四、综合

（1）32 位行波进位加法器

4.1.1 约束条件：

虚拟时钟 virtual_clk，其周期为 40ns（即频率为 25MHz）,无时钟抖动。

输入信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

输出信号相对于 virtual_clk 时钟的最大延迟为 2.5ns，最小延迟为 0ns。

无面积、功耗约束。

4.1.2 综合结果与分析：

能达到的最大频率

a[0] (in)	0.00	23.00 f
.....		
data arrival time		35.70 f

通过行波进位加法器的最长延时为 12.7ns，故加法器能达到的最大频率约为 78MHz（忽略保持时间和建立时间）。

面积

Number of ports:	258
Number of nets:	353
Number of cells:	160
Number of combinational cells:	128

Number of sequential cells:	0
Number of macros/black boxes:	0
Number of buf/inv:	32
Number of references:	32
Total cell area:	**1403.967972**

行波进位加法器需要总的逻辑门面积约为 1404 单位。

（2）四级流水线加法器

4.2.1 约束条件：

时钟 clk，其周期为 20ns（即频率为 50MHz），无时钟抖动。

输入信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

输出信号相对于 virtual_clk 时钟的最大延迟为 2.5ns，最小延迟为 0ns。

无面积、功耗约束。

4.2.2 综合结果与分析：

能达到的最大频率

a_1_reg[0]/CK (FFDQRHD1X)	0.00	0.00 f
……		
s_1_reg[7]/D (FFDQRHD1X)	0.00	1.13 f
Data arrival time		1.13
……		
library hold time	**-0.02**	**-0.02**
……		

四级流水线加法器中最长延时为 1.13ns，是从第一级输入寄存器（a_1_reg[0]）到第一级输出寄存器（s_1_reg[0]）之间的延时，考虑到**寄存器建立时间为 0.02ns**，故加法器能达到的最大频率约为 884MHz。

四级流水线加法器的最大频率能如此大是因为流水线利用寄存器保存输出，使得一个加法过程被拆分成四个过程，降低单个过程所需要的时钟

周期。

同时，我在 rtl 代码中利用 `+` 实现 8 位无符号数的相加，而不是利用行波进位加法器。`+`综合后的加法器电路结构由编译器决定，大概率综合后的加法器性能远大于八位行波进位加法器，故单个过程所需要的时钟周期比 32 位行波进位加法器小不止 4 倍。

面积

Number of ports:	216
Number of nets:	514
Number of cells:	315
Number of combinational cells:	62
Number of sequential cells:	249
Number of macros/black boxes:	0
Number of buf/inv:	30
Number of references:	6
Total cell area:	10242.424293

行波进位加法器需要总的逻辑门面积约为 10242 单位，远大于行波进位加法器，可能的原因是流水线加法器在行波进位加法器的基础上多了 8 个寄存器，以及一些连接节点。

（3）超前进位加法器

4.3.1 约束条件：

虚拟时钟 virtual_clk，其周期为 40ns（即频率为 25MHz）。

输入信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

输出信号相对于 virtual_clk 时钟的最大延迟为 2.5ns，最小延迟为 0ns。

无面积、功耗约束。

4.3.2 综合结果与分析：

能达到的最大频率

```
A[5] (in)                                0.00       23.00 f
……
Sum[31] (out)                            0.00       27.64 f
```

通过超前进位加法器的最长延时为 4.64ns，故加法器能达到的最大频率约为 215MHz（忽略保持时间和建立时间）。超前进位加法器的速度远远大于行波进位加法器，得利于它四位超前进位加法器单元加法器，和各组之间的超前进位，输入不再非常依赖前一级输出。

面积

```
Number of ports:                          252
Number of nets:                           445
Number of cells:                          240
Number of combinational cells:            228
Number of sequential cells:                 1
Number of macros/black boxes:               0
Number of buf/inv:                         44
Number of references:                      13



Total cell area:                     2092.305993
```

超前进位加法器需要总的逻辑门面积约为 2092 单位，大约为行波进位加法器的两倍，这是因为超前进位加法器有更复杂的逻辑关系，利用面积换取速度。

五、静态时序分析

（1）32 位行波进位加法器

由于行波进位加法器是组合电路，故没有静态时序分析。

（2）四级流水线加法器

约束条件：

虚拟时钟 virtual_clk，其周期为 40ns（即频率为 25MHz）。

输入信号相对于 virtual_clk 时钟的最大延迟为 3ns，最小延迟为 0ns。

输出信号相对于 virtual_clk 时钟的最大延迟为 2.5ns，最小延迟为 0ns。

时序分析结果：

维持时间分析：

```
Startpoint: a_1_reg[8]
Endpoint: a_2_reg[0]
Path Group: clk
Path Type: min

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
pipe_line_adder    reference_area_20000  smic13_ff

Point                                    Incr       Path
-----------------------------------------------------------
clock clk (rise edge)                    0.00       0.00
clock network delay (ideal)              0.00       0.00
a_1_reg[8]/CK (FFDQRHD1X)                0.00       0.00 r
a_1_reg[8]/Q (FFDQRHD1X)                 0.15       0.15 f
a_2_reg[0]/D (FFDQRHD1X)                 0.00       0.15 f
data arrival time                                   0.15

clock clk (rise edge)                    0.00       0.00
clock network delay (ideal)              0.00       0.00
a_2_reg[0]/CK (FFDQRHD1X)                0.00       0.00 r
library hold time                       -0.02      -0.02
data required time                                 -0.02
-----------------------------------------------------------
data required time                                 -0.02
data arrival time                                  -0.15
-----------------------------------------------------------
slack (MET)                                         0.17
```

由 hold_time_report 我们可知，a_1_reg[8]寄存器的输出最短延时为 0.15ns，同时 a_1_reg[8] 寄存器的输出与 a_2_reg[0] 寄存器直接相连，相当于逻辑组合延时为 0ns，新数据到达 a_2_reg[0] 寄存器的最短时间为 0.15ns。而 a_2_reg[0] 寄存器的 hold time 为 0.02ns，小于 0.15ns，

因此维持时间要求满足。

a_1_reg[8]寄存器与a_2_reg[0]寄存器之间路径延时最短，只要它们满足维持时间要求，电路其它部分也满足维持时间要求。

保持时间分析：

```
Startpoint: a_1_reg[0]
Endpoint: s_1_reg[7]
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
pipe_line_adder    reference_area_20000  smic13_ff
pipe_line_adder_DW01_add_3
                   reference_area_20000  smic13_ff

Point                                                   Incr       Path
-----------------------------------------------------------------------------
clock clk (rise edge)                                   0.00       0.00
clock network delay (ideal)                             0.00       0.00
a_1_reg[0]/CK (FFDQRHD1X)                               0.00       0.00 r
a_1_reg[0]/Q (FFDQRHD1X)                                0.16       0.16 f
......
s_1_reg[7]/D (FFDQRHD1X)                                0.00       1.13 f
data arrival time                                                  1.13

clock clk (rise edge)                                  20.00      20.00
clock network delay (ideal)                             0.00      20.00
s_1_reg[7]/CK (FFDQRHD1X)                               0.00      20.00
library setup time                                     -0.11      19.89
data required time                                                19.89
-----------------------------------------------------------------------------
data required time                                                19.89
data arrival time                                                 -1.13
-----------------------------------------------------------------------------
slack (MET)                                                       18.76
```

由set_up_time report我们可知，s_1_reg[7]寄存器的输入最慢延时为1.13ns，而s_1_reg[7]寄存器的setup time 为0.11ns，显然满足建立时间要求。

a_1_reg[0]寄存器与s_1_reg[7]寄存器之间路径延时最长，只要它们

满足建立时间要求，电路其它部分也满足建立时间要求。

（3）超前进位加法器

由于超前进位加法器是组合电路，故没有静态时序分析。

六、最大延时路径

（1）32 位行波进位加法器

最大延时路径为 a[0]至 sum[31]，延时为 12.7ns。因为行波进位加法器是全加器的级联，所以前一级计算完毕之后下一级才能开始计算，延时较大。

行波进位加法器的优化方向是改为流水线，避免加法器的闲置状态，或者改为超前进位加法器，使得后一级加法运算不依赖于前一级输出。

（2）四级流水线加法器

最大延时路径为 a_1_reg[0]寄存器与 s_1_reg[7]寄存器、a_2_reg[0]寄存器与 s_2_reg[7]寄存器、a_3_reg[0]寄存器与 s_3_reg[7]寄存器、a_4_reg[0]寄存器与 s_4_reg[7]寄存器之间的延时，延时都为 1.13ns。由于各级寄存器之间的组合逻辑都为同一个加法器，所以组合逻辑的最大延时都相同，中的延时也相同。

四级流水线加法器的优化方向是降低单个放大器的延时，比如改为超前进位加法器，进一步提高最大时钟频率。同时，流水线加法器可以尝试改变流水线级数，在面积、功耗与延时时间之间找到最优平衡点。

（3）超前进位加法器

最大延时路径为 A[5]至 sum[31]，延时为 4.64ns。这条路径延时最大的原因是输入 A 经过三次逻辑运算，依次变为 pg，P*G*，P** G**，得到 P** G**后，经过第三层四位超前进位加法器得到 c_16，c_16 再输入第二层四位超前进位加法器得到 c_31，c_31 最后 p_31 异或得到 sum_31。

超前进位加法器的优化方向是优化电路实现面积，修改逻辑运算电路

的拓扑结构，降低晶体管个数，降低功耗。

七、实验总结

本实验围绕 32 位加法器展开，分别设计并实现了行波进位加法器、四级流水线加法器和超前进位加法器，通过仿真验证、综合及时序分析等环节对各加法器性能进行了全面评估，具体内容如下：

在设计与实现方面，行波进位加法器由 32 个全加器级联而成，结构简单，但进位链过长限制了性能。四级流水线加法器将加法操作分为四级，利用寄存器暂存中间结果，实现了各阶段并行计算，有效提高了运算效率。超前进位加法器通过生成进位生成函数和进位传递函数，减少了进位传递延迟，但逻辑复杂度较高。

仿真验证结果表明，三种加法器在基本加法测试和边界值测试中均能得出正确结果，逻辑功能得以验证。综合分析显示，行波进位加法器最大频率约 78MHz，逻辑门面积约 1404 单位；四级流水线加法器最大频率达 884MHz，但逻辑门面积约 10242 单位；超前进位加法器最大频率约 215MHz，逻辑门面积约 2092 单位。由此可见，流水线加法器和超前进位加法器在速度上相较于行波进位加法器有显著提升，但面积也有所增加，体现了速度与面积的权衡关系。

在最大延时路径方面，行波进位加法器为 a [0] 至 sum [31]，延时 12.7ns；四级流水线加法器是各级加法器输入输出寄存器，延时 1.13ns；超前进位加法器为 A [5] 至 sum [31]，延时 4.64ns。针对不同加法器的特点，分别提出了相应优化方向，行波进位加法器可向流水线或超前进位方式改进，流水线加法器可调整级数、优化单级放大器，超前进位加法器可优化电路拓扑结构以降低功耗和面积。

本实验深入探究了不同类型 32 位加法器的设计原理、性能表现及优化途径，为数字电路设计中加法器的选型与优化提供了重要参考依据，有助于在实际应用中根据具体需求平衡速度、面积和功耗等因素，设计出更高效的数字系统。