Lab2-Report

使用 for 语句进行仿真和综合

```
Adder add1(//对于第一位进行处理
        .a(a[0]),
        .b(b[0]),
        .c_in(c_in),
        .s(sum[0]),
        .c_out(carry_out[0])
    );
    genvar i;
    generate//i 从 1 开始，对于每一个循环，c_in 取前一个循环的
carry_out 输出值，结果保存到 sum[i]中
        for (i = 1; i < LENGTH - 1; i = i + 1) begin
            Adder adder(
                .a(a[i]),
                .b(b[i]),
                .c_in(carry_out[i-1]),
                .s(sum[i]),
                .c_out(carry_out[i])
            );
        end
    endgenerate
    Adder adder(//对最后一位进行处理
```

```
        .a(a[LENGTH-1]),
        .b(b[LENGTH-1]),
        .c_in(carry_out[LENGTH-2]),
        .s(sum[LENGTH-1]),
        .c_out(c_out)
    );
    assign s = sum[LENGTH-1:0];  // 输出和
```

使用$random()函数进行仿真样例生成

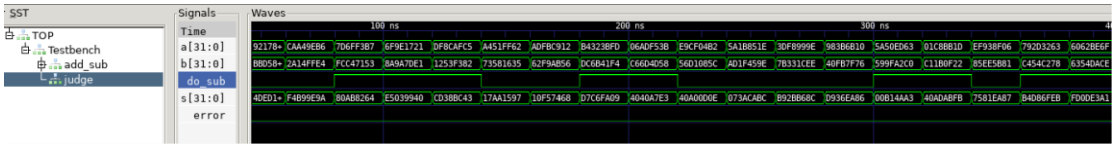
```
integer i;
    initial begin
        for(i = 0; i < 20; i = i + 1) begin
            a = $urandom;//由于 a，b 均为 32 位类型，故无需多做变换
            b = $urandom;
            do_sub = a[0];//do_sub 取随机的 0 或 1，故直接取 a[0]即
可
            #20;//等待 20 个单位
        end
        $finish;
    end
```

综合实现全加减法器

```
wire [LENGTH-1:0] inverted_b;//新建一组 wire 保存 b 的补码
```

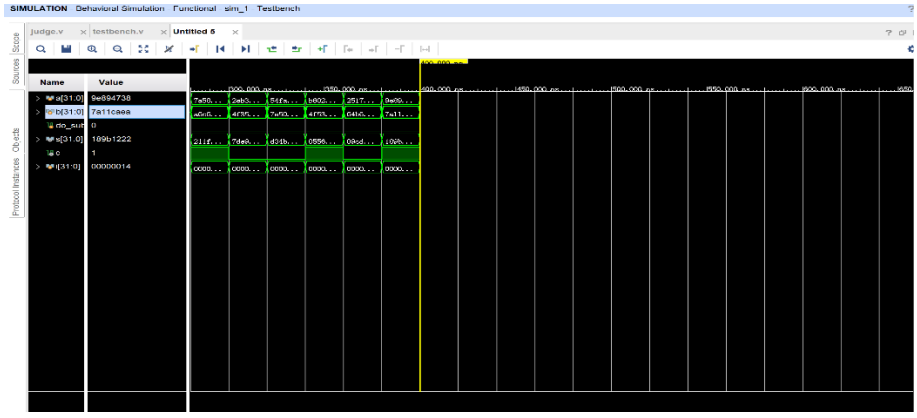
```
    assign inverted_b = (do_sub) ? ~b + 1: b;//若做减法，则相当于 a 与 b 的补码进行加法运算(此处取反操作相当于与全 1 串异或)
    wire tmp;

    Adders #(LENGTH) add_sub (//使用全加器进行加法运算
        .a(a),
        .b(inverted_b),
        .s(s),
        .c_in(0),
        .c_out(tmp)
    );
    assign c = tmp;
```

仿真验证结果



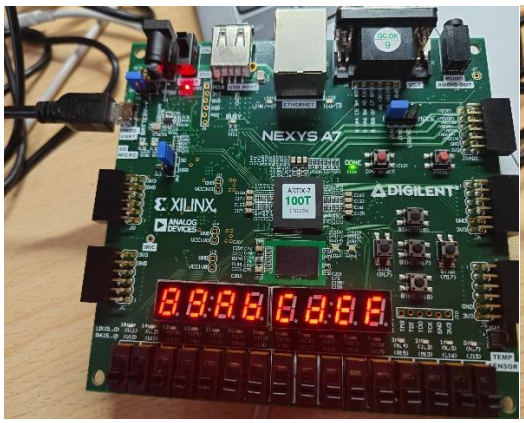
图表 1verilate 仿真

可以看到，error 栏均为 0，说明结果均为正确的。

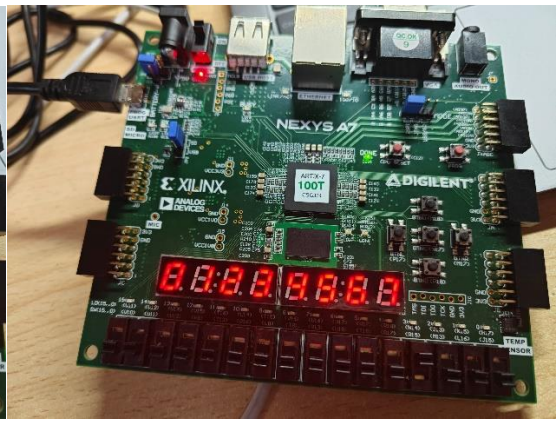


图表 2vivado 仿真

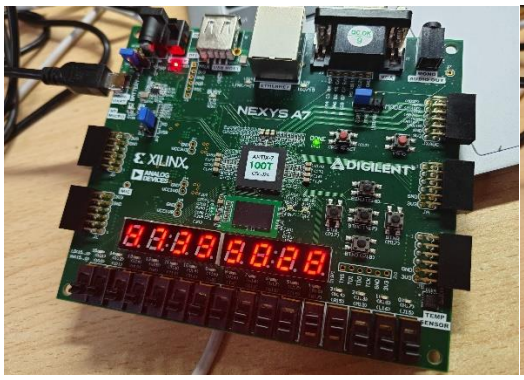
上板验证



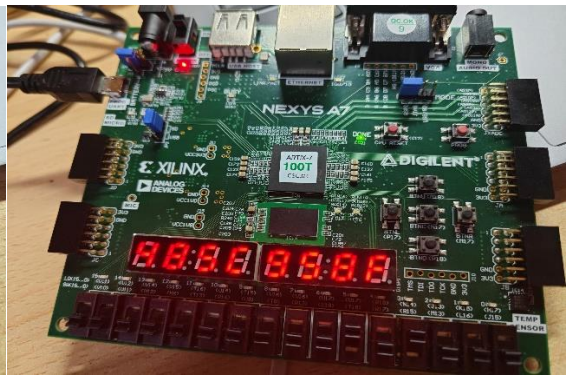
图表 3　SW=0



图表 4　SW=1



图表 5　SW=3

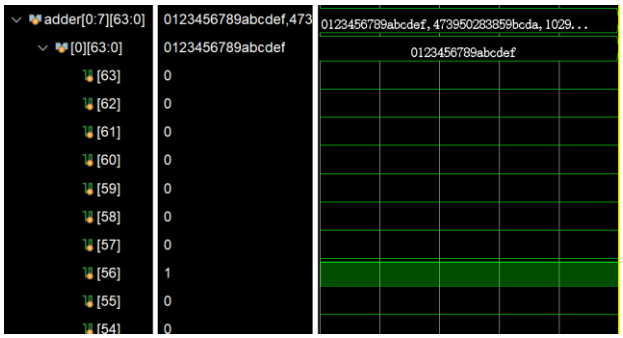


图表 6　SW=5

可以看到，上板结果均正确。

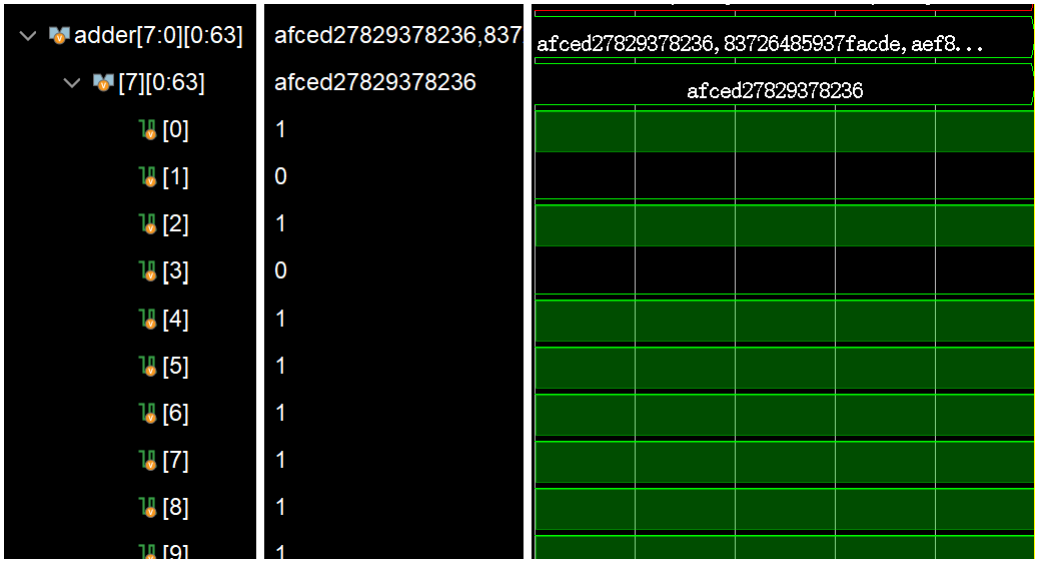
思考的问题

1. 修改 adder 数组后的值



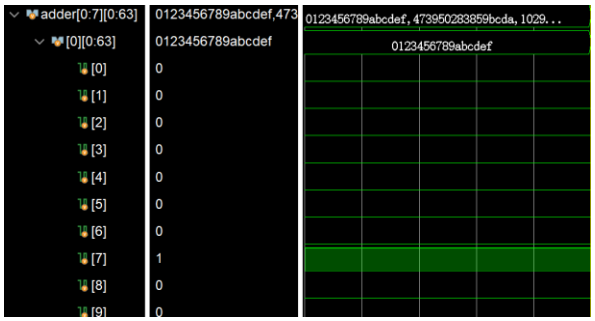
图表 7[63:0] adder [0:7]

且每个元素中的数值从 63 开始存储，比如 adder[0][63:60]即为 0



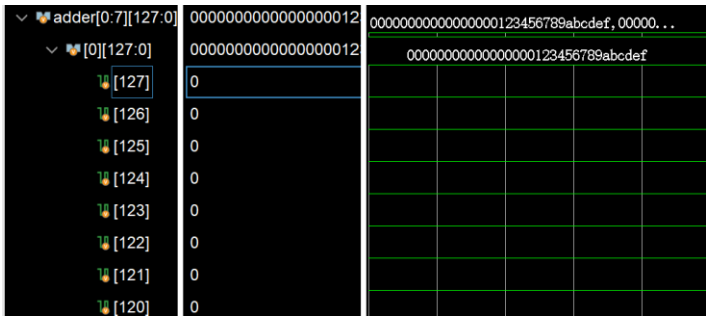
图表 8[0:63] adder [7:0]

从 adder[7]开始，倒序赋值，且每个元素中的数从 0 开始存储，即为 adder[0][0:3]中的内容为 a



图表 9[0:63] adder [0:7]

且每个元素中的数从 0 开始存储，例如 adder[0][0:4]为 0



图表 10[127:0] adder [0:7]

从 adder[0]开始赋值，每个数从高位开始赋值，127:64 位均置为 0，之后与[63:0]的规则相同

赋值方式：从低位开始赋值，如 adder[0] = 0123456789abcdef,如果数组长度超过给定数值，则在高位补 0.

[0;63]或[63:0]改变每个数的存储顺序，即从 63 开始还是从 0 开始

[0:7]或[7:0]改变对于所有数的存储顺序，即从第一个开始还是从最后一个开始

2. 相对于超前进位加法器，行波进位加法器存在的缺点

行波进位加法器的计算延迟比超前进位加法器要高，因为它需要在每个加法位上等待来自前一位的进位。这导致行波进位加法器的运算速度较慢，

因此行波进位加法器不太适合在高速运算环境下使用。在需要高速运算的场景中，通常会选择超前进位加法器或其他更高效的加法器结构。

行波进位加法器对输入的顺序较为敏感。如果输入的数据顺序不恰当，可能会导致延迟增加和性能下降。

3.表示运算溢出：溢出可表示为：overflow = c_out ^ carry_out[LENGTH-2];

如果从第 30 位到第 31 位进位输出和最终的 c_out 不匹配，即原先正数变为负数或者负数变成正数，就意味着发生了溢出。