lab 2-1：64 位全加减法器的实现

1. 实验环境

1. EDA 工具：Ngspice, Logisim-evolution
2. 操作系统：Ubuntu 24.04
3. VHDL：Verilog

2. 实验目的

1. 强化使用 Verilog 语言进行电路设计的方法
2. 掌握行波进位加法器和全加减法器的原理
3. 设计实现 64 位超前进位加法器

3. 实验报告 100%

3.1 每一步的过程&截图和解释 50%

本次实验按“先小模块、再级联、最后系统验证”的顺序完成。先写 1-bit 基本加法单元并单独检查真值，再用 `for + generate` 扩展为多位结构，最后加入加/减控制位形成 `AddSubers`，并用随机激励和下板结果做双重验证。

3.1.1 1-bit 五门全加器 `adder.v`

```
module Adder(
    input a,
    input b,
    input c_in,
    output s,
    output c_out
);
    wire sum0, carry0, carry1;
    half_adder ha1(.a(a), .b(b), .s(sum0), .c_out(carry0));
    //第一位加法器
    half_adder ha2(.a(sum0), .b(c_in), .s(s), .c_out(carry1));
    //第二位加法器，sum与c_in相加
    or or1(c_out, carry0, carry1);
endmodule

module half_adder(
    input a,
    input b,
    output s,
    output c_out
);
    xor xor1(s, a, b); //sum = A XOR B
    and and1(c_out, a, b); //carry = A AND B
endmodule
```

这一步先把 1-bit 全加器搭好：本质是两个半加器再加一个或门。拆成小模块后，结构更清楚，也方便后续在多位加法器里重复调用。

3.1.2 流水线（行波进位）加法器 `adders.v`

- 使用 `for` 语句进行仿真和综合 15%

```
module Adders #(parameter LENGTH = 32)(
    input [LENGTH-1:0] a,
    input [LENGTH-1:0] b,
    input c_in,
    output [LENGTH-1:0] s,
    output c_out
);
    wire [LENGTH:0] c; //carry
    assign c[0] = c_in;
    genvar i;
    generate
        for(i=0; i<LENGTH; i=i+1)begin
            Adder adder(.a(a[i]),.b(b[i]),.c_in(c[i]),
                        .s(s[i]),.c_out(c[i+1]));
        end
    endgenerate
    assign c_out = c[LENGTH];
endmodule
```

这里用行波进位思路，把低位产生的进位依次传到高位。`for` 循环让代码可以直接扩展到任意位宽，仿真和综合都能复用同一套写法。

3.1.3 全加减法器 `add_subs.v`

- 使用 `for` 语句进行仿真和综合 15%
- 综合实现全加减法器 20%

```
module AddSubers #(parameter LENGTH = 32)(
    input [LENGTH-1:0] a,
    input [LENGTH-1:0] b,
    input do_sub,
    output [LENGTH-1:0] s,
    output c
);
    wire [LENGTH-1:0] b_xor_do_sub;
    genvar i;
    generate
        for(i=0; i<LENGTH; i=i+1)begin
            assign b_xor_do_sub[i] = b[i]^do_sub;
        end
    endgenerate
    Adders #(.LENGTH(LENGTH)) adders(
        .a(a),
        .b(b_xor_do_sub/*对应式中的"xor c"*/),
```

```
        .c_in(do_sub/*对应式中的"+c"*/),
        .s(s),
        .c_out(c)
    );
endmodule
```

这一层通过 `b ^ do_sub` 和 `c_in = do_sub` 统一加减法：`do_sub=0` 时做加法，`do_sub=1` 时等效做补码减法。优点是硬件结构统一，不需要分两套电路。

3.1.4 部分 `Testbench.v`

1. 使用 `$random()` 函数进行仿真样例生成 15%
2. 使用 `for` 语句进行仿真和综合 15%

```
integer i;
    initial begin
        //add test sample
        /*Verilog 提供了函数 $random() 来提供宽度为 32 位的随机数*/
        for(i=0; i<10; i=i+1) begin
            a = $random();//固定种子，后续将改进，见下文
            b = $random();
            do_sub = 1'b0;
            #20;
        end
        for(i=0; i<10; i=i+1) begin
            a = $random();
            b = $random();
            do_sub = 1'b1;
            #20;
        end
        $finish;
    end

```

- 发现每次仿真时的**随机数序列相同**，可能是默认种子是固定值导致。
- 查询资料后在 `initial` 代码块下加入时间种子可以解决该问题：

```
reg [31:0] a;
integer seed;
initial begin
    seed = $stime;
    a = $random(seed);
end
```

- 经多次尝试，时间种子有效，每次仿真的**随机数序列不同**，且 `error = 0`。
- 完整代码将提交在 `lab2-code` 中。

3.1.5 仿真验证

仿真验证结果如下图所示。

测试部分分两轮随机激励：第一轮测加法，第二轮测减法（对于前十个测试点，`do_sub = 0`；对于后十个测试点 `do_sub = 1`）。再用 `Judge` 对输出自动比对。

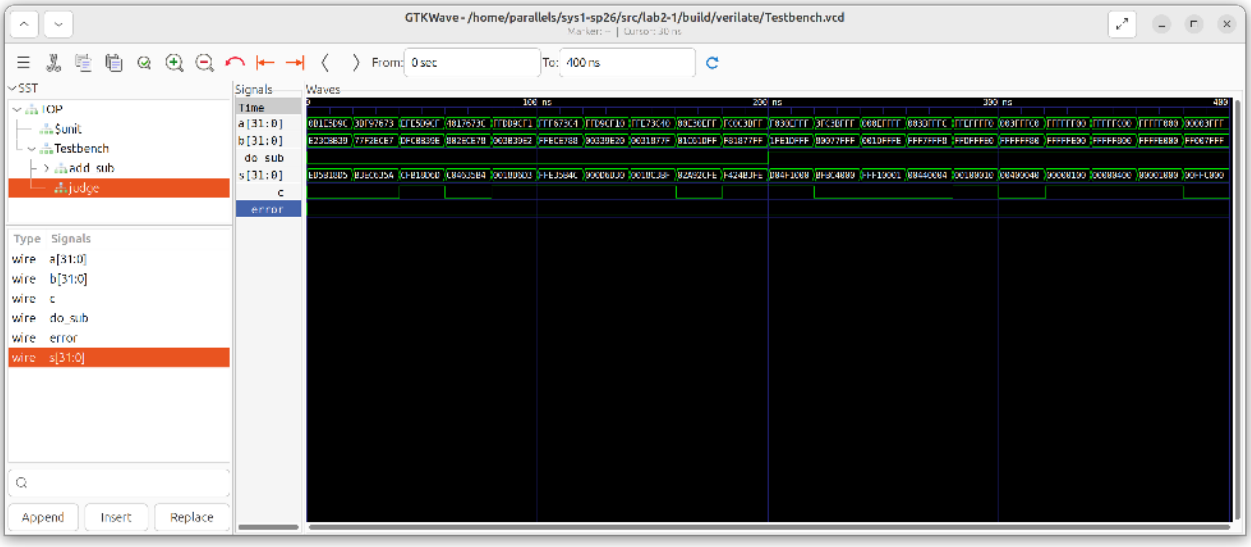

Image 1

3.1.6 下板验证

> **Note**
>
> 取第一组数据 `a = 0123456789abcdef`，`b = 473950283859bcda`。

3.1.6.1 ADD

sum：`s = 0x485C958FC2058AC9`，carry：`c = 0`（无进位），经验证，计算正确。

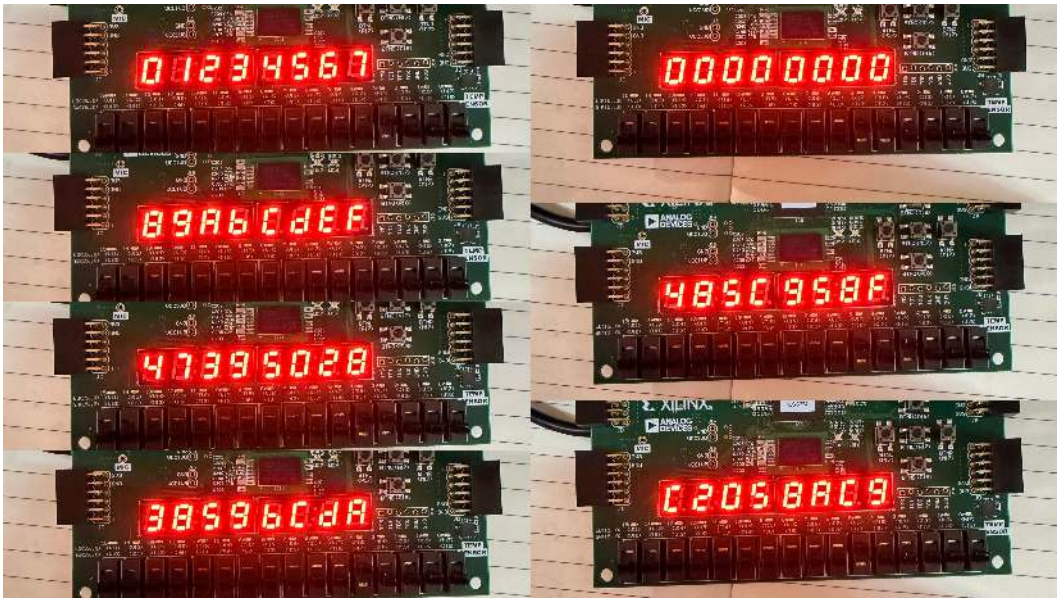

Image 2

3.1.6.2 SUB

sum: `s = 0xB9E9F53F51521115`, carry: `c = 0`（有借位），经验证，计算正确。

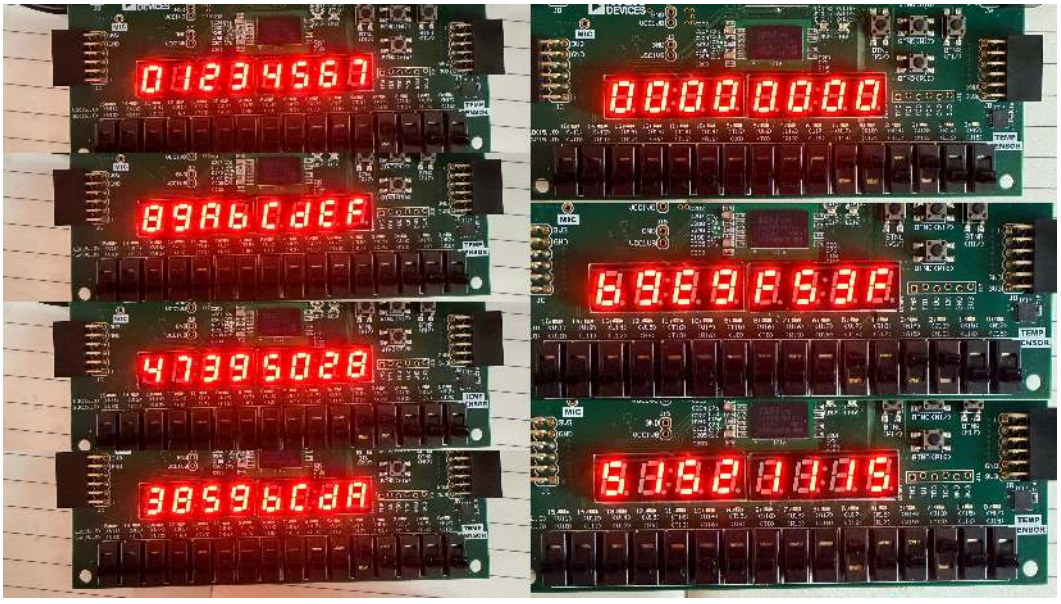

Image 3

4. 思考题

1. 对于 `repo/sys-project/lab2-1/syn/top.v` 中的 `reg [63:0] adder [0:7]`，通过仿真或者综合下板观察 adder 的每个 bit 会被初始化为何值？请尝试对对 `readmemh` 如何初始化一维向量数组的每个 bit 给出结论。15%

> `$readmemh` 将文件第 1 行给 `adder[0]`，第 2 行给 `adder[1]`，依次类推。每行的十六进制数会按位填到对应的 64 位 `reg` 里（高位对高位、低位对低位）；位数不够会在高位补 0，位数太多会截断高位。

根据下板结果：

`sw[5:3] = {0, 0, 0}` 时，`adder[0]` 的低32位 `a = 89abcdef`。

`sw[5:3] = {0, 0, 1}` 时，`adder[0]` 的高32位 `b = 01234567`。

2. 请总结超前进位加法器和行波进位加法器的优缺点？20%

1. 行波进位加法器
   (a) 优点是结构简单、面积小、容易编写；
   (b) 缺点是位宽较大时速度会明显变慢。
2. 超前进位加法器
   (a) 优点是速度快；
   (b) 缺点是逻辑更复杂、资源开销更大。

3. 如何表示运算溢出？请给出溢出表示（用 `AddSubers` 的输入与输出以及各位进位 c_i 计算）。15%

- 无符号加法：看最高位进位，`c=1` 说明溢出。
- 无符号减法：看借位。用补码减法实现时，`c=0` 说明有借位（可理解为溢出）。
- 有符号加法：同号相加结果变号则溢出，
  `a[LENGTH-1] == b[LENGTH-1]` 且 `s[LENGTH-1] != a[LENGTH-1]`。
- 有符号减法：异号相减结果变号则溢出，
  `a[LENGTH-1] != b[LENGTH-1]` 且 `s[LENGTH-1] != a[LENGTH-1]`。
- **有符号溢出也可写成**：
  `overflow = c[LENGTH] ^ c[LENGTH-1]`。
  其中 `c[LENGTH]` 为符号位向后的进位，即 `c_out`。

验证：

1. 同号相加
   (a) 正+正：符号位 `0+0+x = x`，`c[LENGTH] = 0`
      i. 不溢出 `x = 0`：数值位无进位，`c[LENGTH-1] = 0`，异或为 0。
      ii. 溢出 `x = 1`：数值位有进位，`c[LENGTH-1] = 1`，异或为 1。
   (b) 负+负：符号位 `1+1+x = x`，`c[LENGTH] = 1`
      i. 不溢出 `x = 1`：数值位有进位，`c[LENGTH-1] = 1`，异或为 0。
      ii. 溢出 `x = 0`：数值位无进位，`c[LENGTH-1] = 0`，异或为 1。
2. 异号相减：与同号相加相同