姓名： Epi 实验日期：2024 年 9 月 29 日

实验名称：实验一、基础练习

实验目的：熟悉实验环境，学习如何安装 Vivado、如何使用Vivado 2018 创建工程、代码编辑、RTL 分析、仿真等设计流程。

实验内容：

（1）学习所有视频以及“lab0.pdf”，了解 Vivado 设计流程和功能

（2）按照“lab1.pdf”完成数码管和加法器实验；

（3）Vivado 代码编辑和 RTL 分析；

实验结果与分析：

（1）观看提供的所有视频资料；学习：Vivado 设计流程中的基本概念.pdf、约束文件.ppt。回答以下问题：

- 描述 Vivado 的设计流程

1. 项目创建：在 vivado 中创建新的项目，指定 FPGA 型号和开发板型号
2. RTL 设计：用 HDL 语言（如 verilog）编写 RTL 代码
3. 综合：HDL 代码转换为逻辑门级别的网络表达，综合成网表文件
4. 时序约束：确保电路按照规定的时序运行
5. 实现：将网表映射到真实的 FPGA 器件上，并生成 bitstream
6. 仿真调试：对设计进行仿真调试，确保设计按照预期工作

- 什么是网表

网表是对电路设计逻辑门级别的描述

- Vivado 设计流程中，Synthesis 的作用是什么？

将 HDL 代码转换为逻辑门级别的网络表达，综合成网表文件

- Vivado 设计流程中，Implementation 的作用是什么？

将门级网表映射到 FPGA 的物理资源上的过程

（2）按照“lab1.pdf”完成数码管实验；

- 给出数码管仿真结果截图，对波形进行简要解释：

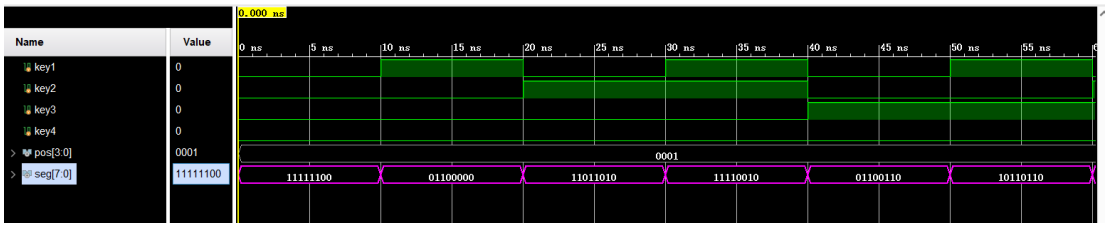


图 1 七段数码管的数码仿真结果

如图，{key1-key4}是模拟的测试用例，在 0-10ns 为{0000}，10-20ns 为{1000}等等

pos[3:0]由于写程序的时候就固定为常数了所以没有波动。

seg[7:0]给出了模拟出的对应输出（源文件写的与 lab1.pdf 稍有区别，所以刚好二进制的位是反过来的，当然调整接口顺序还是可以正常显示的）

可以看到随着{key1-key4}的变化，seg[7:0]也相应地发生了变化。而值与我们的源码时一致的，这说明程序确实按照我们的预想实现了！

```
always @(data) begin
    pos = 4'b0001;
    case (data)
        4'b0000: seg = 8'b11111100; // 0
        4'b0001: seg = 8'b01100000; // 1
        4'b0010: seg = 8'b11011010; // 2
        4'b0011: seg = 8'b11110010; // 3
        4'b0100: seg = 8'b01100110; // 4
        4'b0101: seg = 8'b10110110; // 5
        4'b0110: seg = 8'b10111110; // 6
        4'b0111: seg = 8'b11100000; // 7
        4'b1000: seg = 8'b11111110; // 8
        4'b1001: seg = 8'b11110110; // 9
        default: seg = 8'b00000000; // 默认关闭
```

图 2 数码管的部分源码实现

- 给出板子运行结果照片，以及你的操作过程：

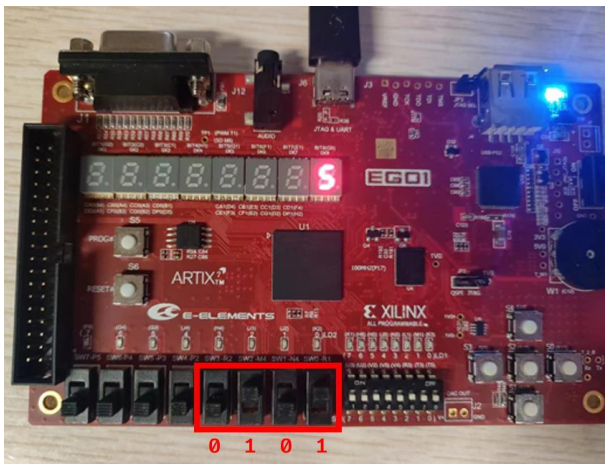


图 3 测试结果 1

如上图，后四个开关依次对应{key1-key4}，输入 0101 时，数码管显示“5”。

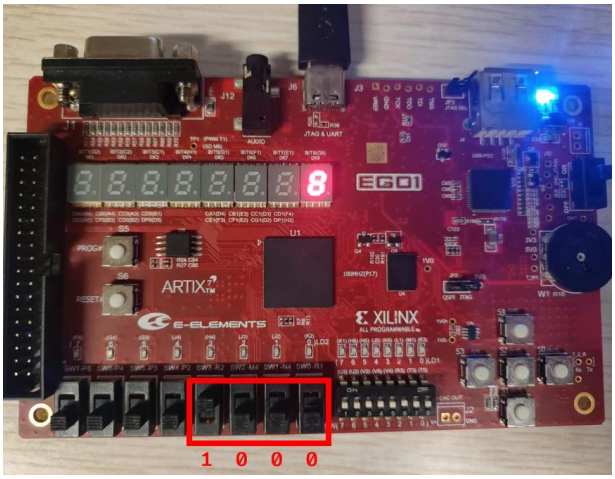


图 4 测试结果 2

如上图，输入 1000 时，数码管显示“8”。

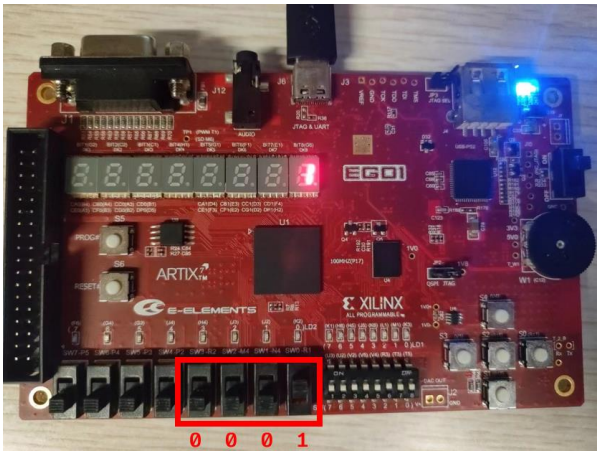


图 5 测试结果 3

如上图，输入 0001 时，数码管显示“1”。

- segMsg 的输入信号的作用都是什么？

~~（真不问输出信号吗？）~~输入信号以二进制的形式区分出 4bit 的信息，以此区分不同的数字。

~~（还是答一下输出信号）~~pos 可以认为是数码管的共阴极，本题中它控制着右侧四个数码管中只有最右侧的亮，seg 的各位对应着数码管上的 8 段（包括右下角的点）。

（3）按照“lab1.pdf”完成加法器实验；

- 给出两位半加器源代码，并进行简要解释：

两位半加器源代码

```
module full_adder (
    input wire A,
    input wire B,
    input wire Cin,
    output wire Sum,
```

```
    output wire Carry
);
    assign Sum = A ^ B ^ Cin;
    assign Carry = (A & B) | (Cin & (A ^ B));
endmodule


module two_bit_half_adder (
    input wire [1:0] A,
    input wire [1:0] B,
    output wire [1:0] Sum,
    output wire Carry
);
    wire Sum0;
    wire Carry0;
    wire Sum1;
    full_adder FA0 (
        .A(A[0]),
        .B(B[0]),
        .Cin(1'b0),
        .Sum(Sum0),
        .Carry(Carry0)
    );
    full_adder FA1 (
        .A(A[1]),
        .B(B[1]),
        .Cin(Carry0),
        .Sum(Sum1),
        .Carry(Carry)
    );
    assign Sum = {Sum1, Sum0};
endmodule
```

上述代码是完整的源代码，解释如下：

首先，实现一位全加器模块，输入（A，B，Cin），输出（Sum，Carry）。Sum 的逻辑很好理解，而 Carry 的逻辑分解为两部分：A & B：当 A 和 B 同时为 1 时，会产生进位；Cin & (A ^ B)：如果进位输入 Cin 为 1，且 A 和 B 不同，也会导致进位产生。

有了一位全加器，就可以实现二位全加器了，逻辑如下：

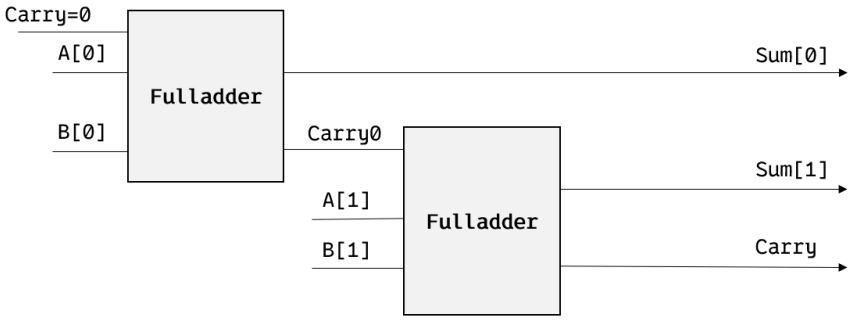
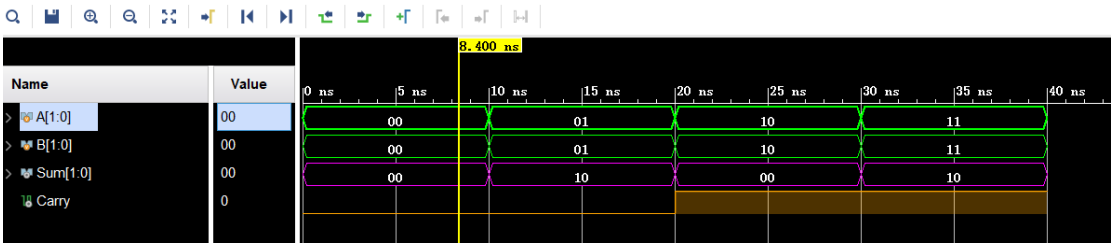


图 6 两位半加器的实现逻辑

➢ 给出两位半加器仿真结果截图，对波形进行简要解释：



A，B 代表输入的两位二进制数，Sum 给出了输出结果，Carry 给出了进位

如 20-30ns 中的仿真结果显示了 10+10=(1)00 的结果

➢ 给出板子运行结果照片，以及你的操作过程：（如果没有可以删除）

这里没有直接继续，而是进一步做了数码管显示模块，代码如下

```
module adder_with_seven_segment (
    input [1:0] A,
    input [1:0] B,
    output reg pos,
    output reg [6:0] seg,
    output Carry
);
wire [2:0] Sum;
wire c1;
assign Sum[0] = A[0] ^ B[0];
assign c1 = A[0] & B[0];
assign Sum[1] = A[1] ^ B[1] ^ c1;
assign Carry = (A[1] & B[1]) | (c1 & (A[1] ^ B[1]));
assign Sum[2] = (A[1] & B[1]) | (c1 & (A[1] ^ B[1]));
always @(Sum) begin
    pos = 1;
    case(Sum)
        3'b000: seg = 7'b0111111;
        3'b001: seg = 7'b0000110;
        3'b010: seg = 7'b1011011;
```

```
        3'b011: seg = 7'b1001111;
        3'b100: seg = 7'b1100110;
        3'b101: seg = 7'b1101101;
        3'b110: seg = 7'b1111101;
        3'b111: seg = 7'b0000111;
        default: seg = 7'b0000000;
    endcase
end
endmodule
```

在两位半加器的基础上，直接简化了逻辑，并添加了 Sum[2]（保留进位而不是截断），并添加了数码管输出模块。板子测试结果如下：

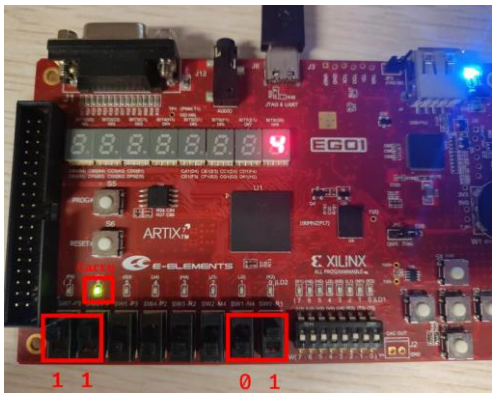


图 7 两位半加器测试结果 1

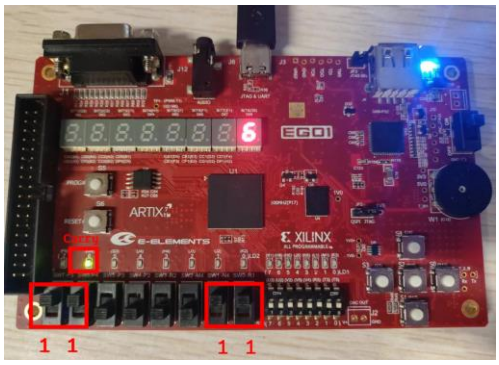


图 8 两位半加器测试结果

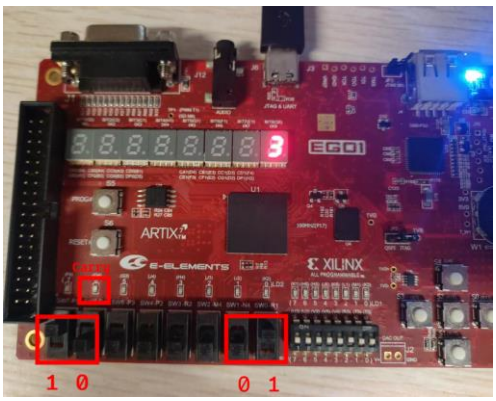


图 9 两位半加器测试结果 3

与预期完全一致！

➢ 如果要实现两位全加器，需要在两位半加器的基础上做什么改动？

只要添加进位输入，即：

```
module two_bit_full_adder (
    input wire [1:0] A,
    input wire [1:0] B,
    input Cin,
    output wire [1:0] Sum,
    output wire Carry
);
```

具体实现只要让 Cin 成为第一个全加器的输入即可，即：

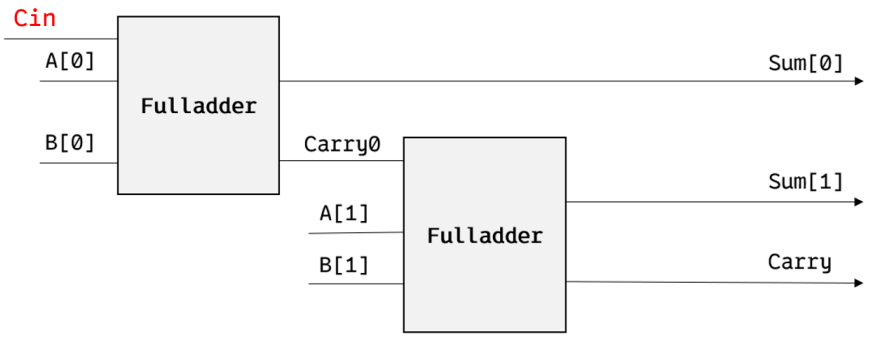


图 10 两位全加器的实现逻辑

（4）创建一个工程，自己指定工程位置和工程名称，新建空白源程序文件，依次完成下面代码编辑和 RTL 分析：

➢ 分别编写教材图 2.37、2.38 例子，观察 vivado 工具的 RTL 分析结果，截图如下；

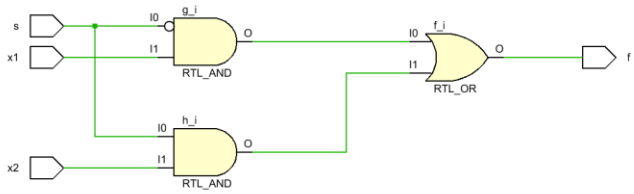


图 2.37 对应的 RTL 分析结果

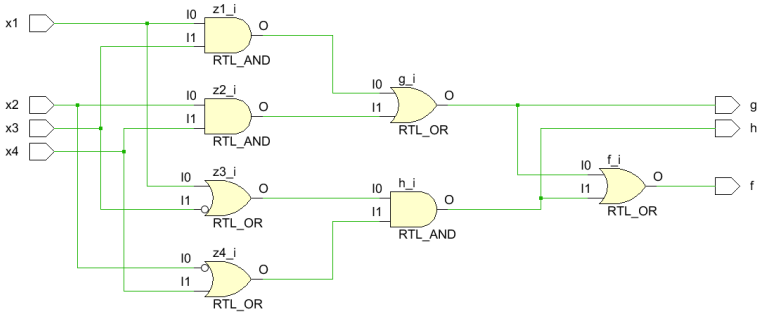


图 2.38 对应的 RTL 分析结果

- 分别编写教材图 2.40、2.41 例子，观察 vivado 工具的 RTL 分析结果，截图如下；

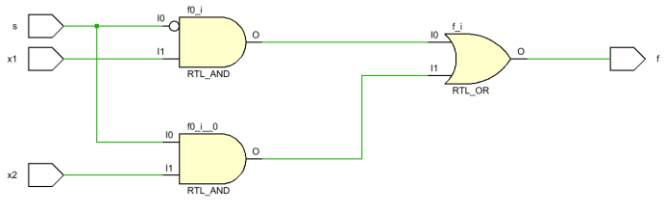


图 2.40 对应的 RTL 分析结果

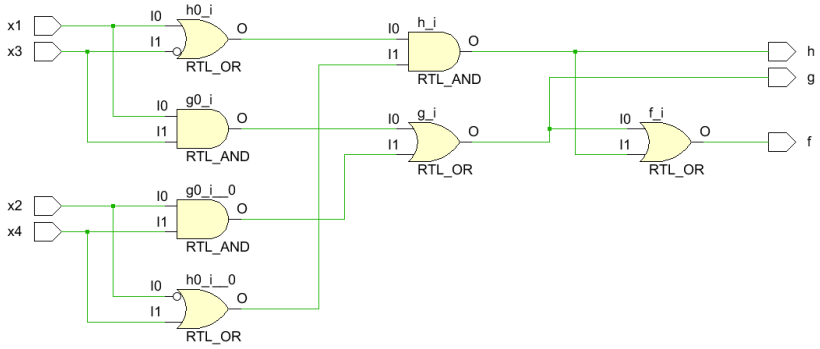


图 2.41 对应的 RTL 分析结果

- 分别编写教材图 3.18、3.20 例子，对比 vivado 工具的 RTL 分析结果，截图并给出你对结果的理解；

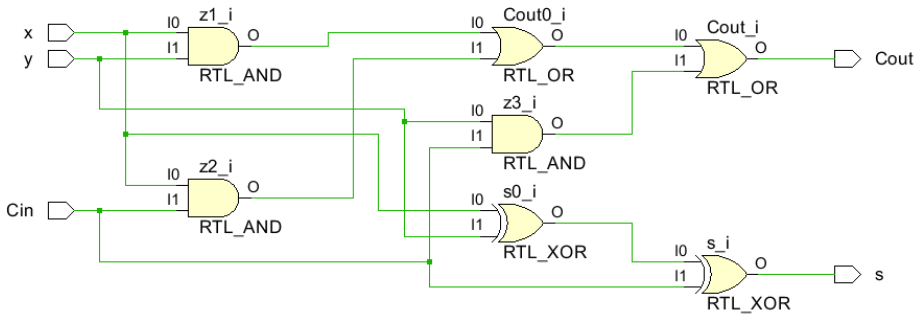


图 3.18 对应的 RTL 分析结果

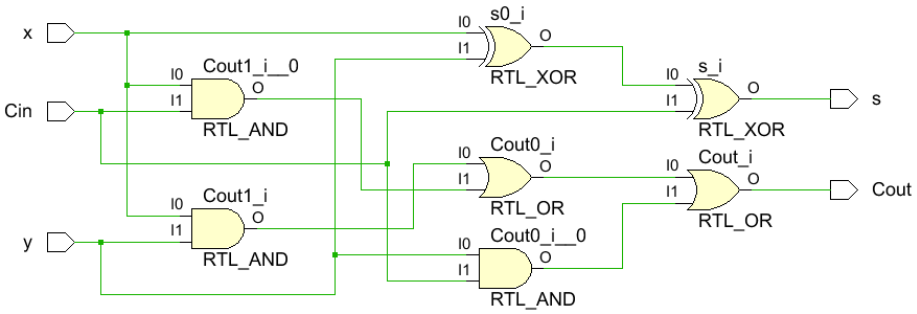
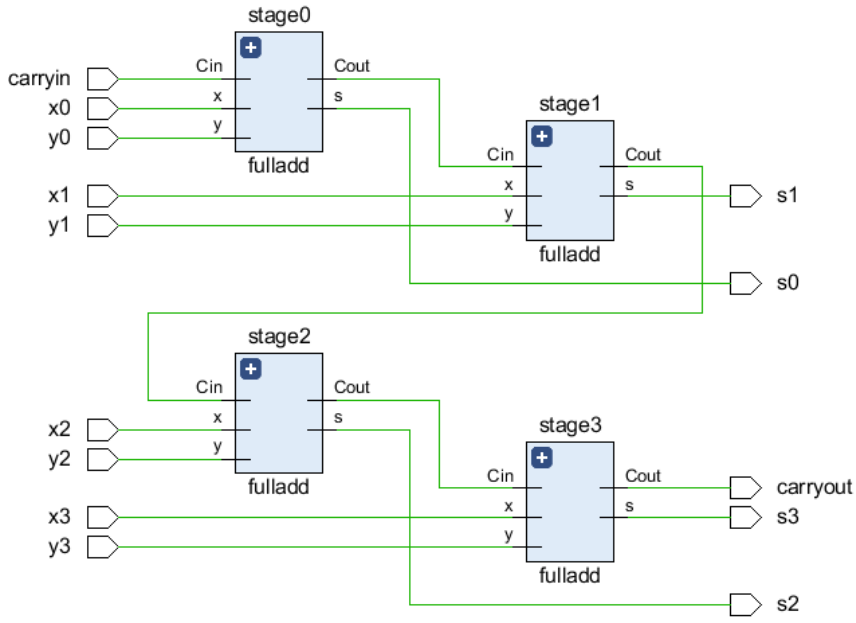


图 3.20 对应的 RTL 分析结果

他们都是在描述全加器，s 是三个输入做异或的结果，Cout 只要有至少两个输入为 1 即为 1。就 RTL 分析结果来看，它们都是由两个一位半加器组合而成的。

- 编写教材图 3.22 例子，观察 vivado 工具的 RTL 分析结果，截图并给出你对结果的理解；



这实现了一个四位全加器模块，其中 fulladd 是一位全加器模块

功能上，它计算了 `x0x1x2x3 + y0y1y2y3 + Carry = carryout + s0s1s2s3`

（5）实验中遇到哪些问题，是如何解决的。（如果没遇到问题可以不写）

不了解七段数码管的引脚位置，查询文档后解决。

使用 vivado 仿真时，没注意时间尺度以为没仿真出结果，调整尺度后解决。

（6）本次实验的感受及建议（如没有体会和想法可以不写）。

体会：lab 文档很详细，跟着做很快就搞明白了 vivado 的流程。然后就是 vivado 的有些流程有点缓慢（甚至中途去换了 vscode+iverilog+gtkwave 的组合，感觉还挺简便的）

建议：感觉对于半加器和全加器的区分可以明确提出来，还有就是可以明确的说明一下数码管的原理（比如什么是共阴极）。