北京航空航天大學

BEIHANG UNIVERSITY

实验二、组合逻辑设计——定、浮点数转换

2018 年 10 月 24 日（版本 v01）

网络空间安全学院

目录

实验二、组合逻辑设计——定、浮点数转换（实验指导书部分）

1 实验指导

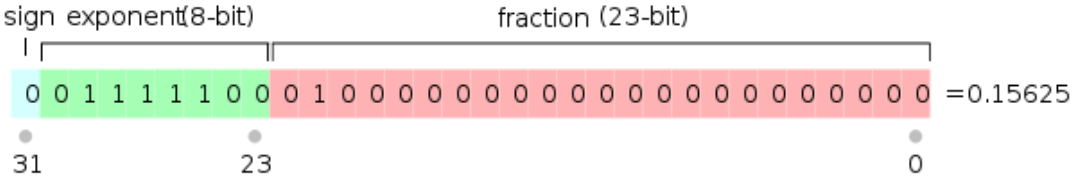
通过一个定点数转换为浮点数的案例，利用 EELAB-FPGACORE2 实验硬件实验平台（以下简称“实验板”）通过 Verilog HDL 语言实现可综合代码，加深对组合逻辑设计理念的认识。

1.1 背景知识——浮点数

浮点数格式是一种计算机数据格式，利用“浮点”（浮动小数点）的方法，可以表示一个范围很大的数值。

1.1.1 IEEE 754 浮点数格式

浮点数是一种二进制的科学计数法（scientific notation），回顾常见的十进制科学计数法，例如：-103.24 = -1.0324 × 10^{+2} ，其中模（或称为“base”）为 10，而计数法中的关键参数是：科学计数绝对值（含一位整数和小数点后的尾数，被称为“mantissa”），科学计数符号，幂指数（“exponent”）的绝对值，幂指数的符号。二进制科学计数法只不过取模为 2，其四种参数与之相同，将该四种参数按照一定的格式以 32 位或 64 位（乃至更长）的二进制编码记录。



说明：本图来源于 wiki 百科，以$(0.15625)_{10}$，即$(0.00101)_2$的单精度浮点数为例

图 1 IEEE754 单精度浮点数的格式

常用的浮点数标准为 IEEE 754，其中单精度浮点数为 32 位，由 1-bit 符号位，8-bit 经过偏移（bias）的指数，以及 23 位尾数组成。其中（如图 1）：

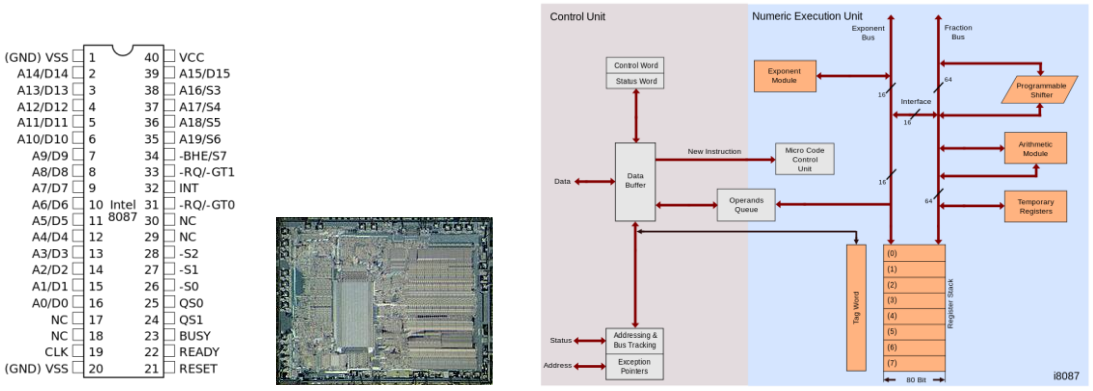
- 符号位：0 表示正数，1 表示负数；
- 指数部分：为了能够表达正指数和负指数，规定偏移量为 127，例如：对于$(0.00101)_2$，可以表达为二进制 1.01×2^{-3}，则指数为 127+(-3)=124，其自然二进制编码为 8’b0111_1100；
- 尾数部分：注意到对于二进制的科学计数值，其整数部分总是 1，所以该整数部分在 IEEE

754 标准中不被存储，只存储尾数的纯小数部分（fraction），例如：对于二进制 1.01×2^{-3}，只用存尾数 01，后面补 0，如果是单精度浮点数，则为 23'b010_0000_0000_0000_0000_0000。

1.1.2 浮点数协处理器

定点数到浮点数，乃至浮点数到定点数的转换，可以编写程序运算，但更高效的方法是采用带有硬件实现的协处理器。在 PC 机发展之初，协处理器芯片和 CPU 芯片是分开的，这也决定了当时电脑的“档次”，例如：Intel 系列的不带协处理器的 8088 和带协处理器的 8086。

协处理器对定、浮点数的转换采用微码的方式实现，不严格地说它也相当于一个功能比较专用的小计算机，是 8087 协处理器的引脚图、模板图和简化的微体系架构模块图。



(a)芯片引脚 (b) 芯片模板 (c) 芯片微体系架构

图 2 8087 协处理器

随着芯片集成工艺的进步，目前将浮点数协处理功能集成到 CPU 芯片中。

微码实现的浮点数协处理器功能强大而灵活，但实际上定点数到浮点数的转换可以完全通过组合逻辑电路实现（反之亦然），只不过对于字长较大的浮点数，资源非常浪费。

1.2 案例：4-bit 无符号定点数转换为浮点数

定点数到浮点数的转换首先要确定定点数中第一个不为 0 的比特的位置，根据该位置确定指数的值，并且根据该位置确定将尾数移位的位数。而位置的确定用优先权编码器就可以胜任。

下面可以通过一个 4-bit 定点数（出于演示的目的，这里暂时只考虑无符号的正整数，指数部分不加偏移量，尾数部分也不去除整数的“1”）的浮点数转换的例题进行启发式说明。

可以采用优先权编码器（priority encoder）和数据选择器（multiplexer），将二进制定点数转换为二进制浮点数；如图 3 所示，以 4-bit 的二进制无符号整数输入为例，用 74148 优先权编码器和 74153

双四选一数据选择器芯片，配合必要的反相器门电路实现。（74148 和 74153 的功能表分别如表 1 和表 2 所示。）

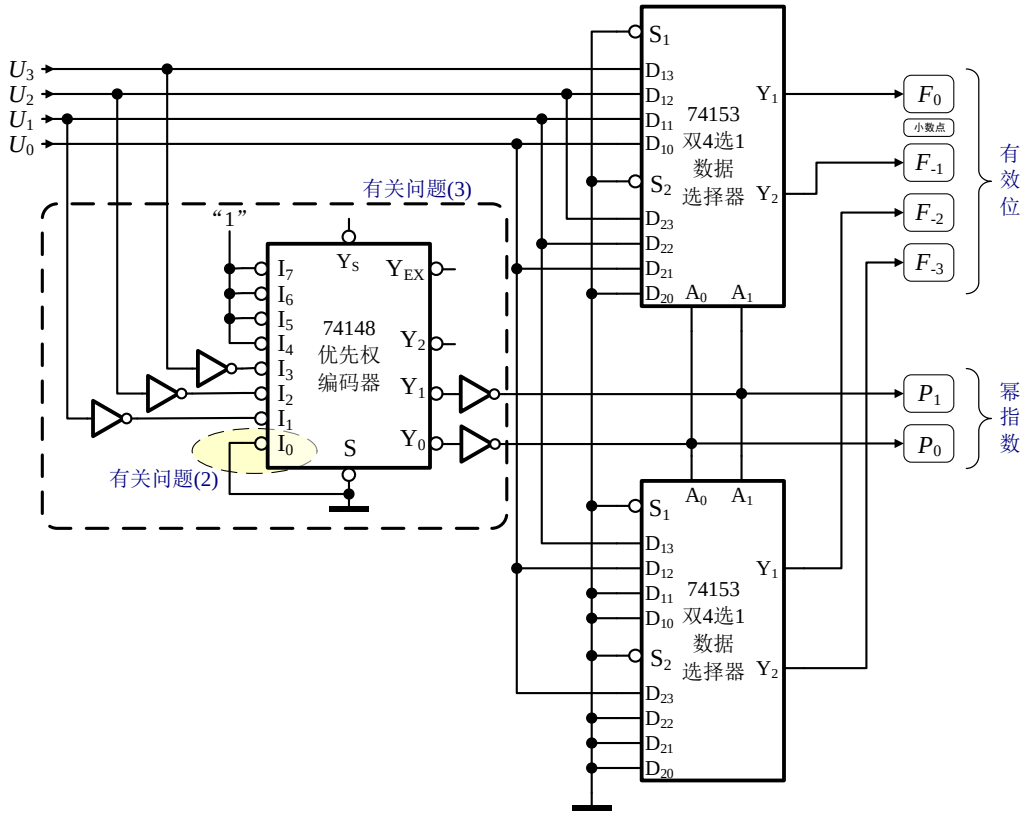


图 3 将 4-bit 的二进制无符号整数转换为二进制浮点数

表 1 优先权编码器 74148 的功能表

输入									输出				
$\overline{S}$	$\overline{I_0}$	$\overline{I_1}$	$\overline{I_2}$	$\overline{I_3}$	$\overline{I_4}$	$\overline{I_5}$	$\overline{I_6}$	$\overline{I_7}$	$\overline{Y_2}$	$\overline{Y_1}$	$\overline{Y_0}$	$\overline{Y_S}$	$\overline{Y_{EX}}$
1	×	×	×	×	×	×	×	×	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	0	1
0	×	×	×	×	×	×	×	0	0	0	0	1	0
0	×	×	×	×	×	×	0	1	0	0	1	1	0
0	×	×	×	×	×	0	1	1	0	1	0	1	0
0	×	×	×	×	0	1	1	1	0	1	1	1	0
0	×	×	×	0	1	1	1	1	1	0	0	1	0
0	×	×	0	1	1	1	1	1	1	0	1	1	0
0	×	0	1	1	1	1	1	1	1	1	0	1	0
0	0	1	1	1	1	1	1	1	1	1	1	1	0

表 2 双四选一数据选择器 74153 的功能表

输入			输出	输入			输出
$\overline{S_1}$	A_1	A_0	Y_1	$\overline{S_2}$	A_1	A_0	Y_2
1	×	×	0	1	×	×	0
0	0	0	D_{10}	0	0	0	D_{20}
0	0	1	D_{11}	0	0	1	D_{21}
0	1	0	D_{12}	0	1	0	D_{22}
0	1	1	D_{13}	0	1	1	D_{23}

请解决如下问题：

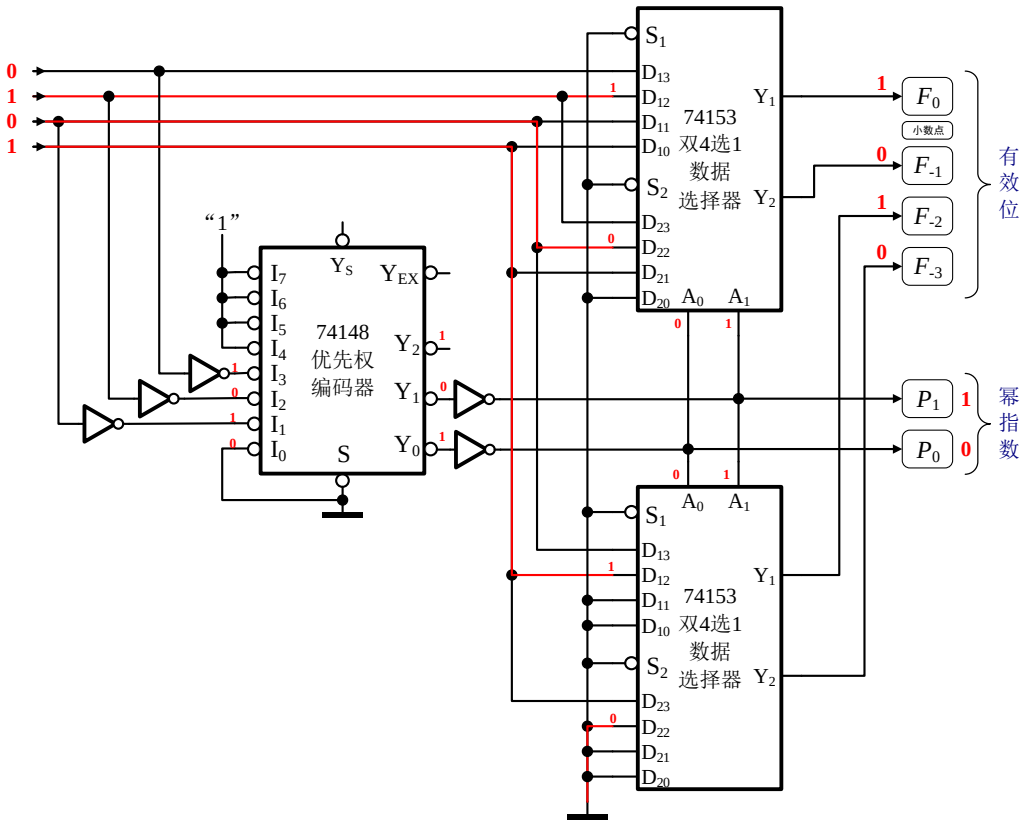
(1) 例如：输入的无符号整数 $U_3U_2U_1U_0$ 为$(0101)_2$，则输出的数字信号 $F_0F_{-1}F_{-2}F_{-3}$ 和 P_1P_0 分别为什么数值？

(2) 图 3 中的椭圆虚线框中所示的 74148 的 $\overline{I_0}$ 端为什么直接接地？（接地相当于逻辑“0”）

(3) 假设采购不到 74148 芯片，则仅依靠 74153 芯片（如必要，还可以用反相器），如何实现图 3 中方形虚线框中电路的功能？请给出设计说明，并绘制这部分功能电路的原理图。

解答：

(1)如果输入的无符号整数 $U_3U_2U_1U_0$ 为$(0101)_2$，则输出的数字信号 $F_0F_{-1}F_{-2}F_{-3}$ 为$(1010)_2$，P_1P_0 为$(10)_2$；电路中的数值如下图所示。



(2)图 3 中的 74148 芯片的 $\overline{I_0}$ 端直接接地，相当于接逻辑“0”，这是因为不论输入为$(0001)_2$还是$(0000)_2$，幂指数均应显示为$(00)_2$，所以使 $\overline{I_0}=0$，不论 U_0 是否为 1。

(3)以 74148 优选权编码器为核心的电路功能是：

（逻辑取值，作答时可以不列出逻辑取值）：

U_3	U_2	U_1	U_0	P_1	P_0
1	×	×	×	1	1
0	1	×	×	1	0
0	0	1	×	0	1
0	0	0	1	0	0

可以分别用“四选一”实现 P_1 和 P_0 的组合逻辑函数。“四选一”的逻辑表达式为：

$Y_i=[(\overline{A_1}\cdot\overline{A_0})\cdot D_{i0}+(\overline{A_1}\cdot A_0)\cdot D_{i1}+(A_1\cdot\overline{A_0})\cdot D_{i2}+(A_1\cdot A_0)\cdot D_{i3}]\cdot S_i$。

逻辑表达式为：

$P_1=U_3+\overline{U_3}\cdot U_2=U_3+U_2$，

$P_0=U_3+\overline{U_3}\cdot\overline{U_2}\cdot U_1=U_3+\overline{U_2}\cdot U_1$

可以取：

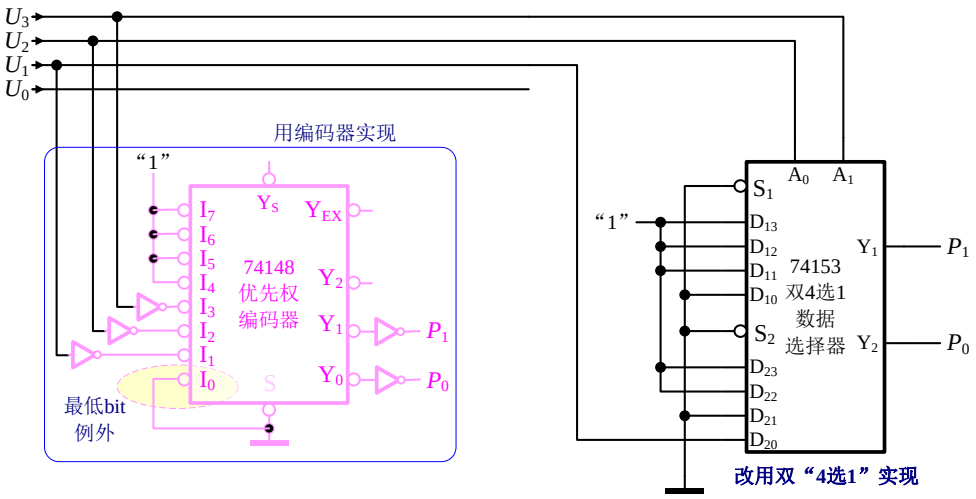
$$\begin{cases} A_1 = U_3 \\ A_0 = U_2 \\ D_{10} = 0 \\ D_{11} = D_{12} = D_{13} = 1 \\ Y_1 = P_1 \\ S_1 = 1 \end{cases}$$ ，使得：

$$P_1 = U_3 + U_2 = \overline{U_3} \cdot \overline{U_2} \cdot 0 + \overline{U_3} \cdot U_2 \cdot 1 + U_3 \cdot \overline{U_2} \cdot 1 + U_3 \cdot U_2 \cdot 1$$

$$\begin{cases} A_1 = U_3 \\ A_0 = U_2 \\ D_{20} = U_1 \\ D_{21} = 0 \\ D_{22} = D_{23} = 1 \\ Y_2 = P_0 \\ S_2 = 1 \end{cases}$$ ，使得：

$$P_0 = U_3 + \overline{U_2} \cdot U_1 = \overline{U_3} \cdot \overline{U_2} \cdot U_1 + \overline{U_3} \cdot U_2 \cdot 0 + U_3 \cdot \overline{U_2} \cdot 1 + U_3 \cdot U_2 \cdot 1$$

绘制电路图如下：



说明：恰好也不需要使用反相器。

1.3 基本实验要求

基本实验要求为：

a) 请采用门级**编程**，实现 4-bit 无符号整数到浮点数转换；

b) 请采用 RTL 级**编程**，实现 4-bit 无符号整数到浮点数转换；

c) 分别对门级编程实现和 RTL 级编程实现的组合逻辑电路进行**功能仿真**；

d) 利用“实验板”对两种 4-bit 无符号整数到浮点数转换电路进行**综合和实现**，设定定点数输入和浮点数输出的人机接口，建议用 4 个 LED 灯表示输入值，操作开关或按动按钮后进行转换，用数码管显示有效位和幂指数；（任何合理的人机接口都是可以接受的）

e) 如果尝试更多位定点数转浮点数的设计，例如：实现 8 位整数定点数转换为浮点数，如果仍采用优先权编码器决定幂指数，请估算所用数据选择器的规模与个数；（可以不用编程或实现，仅在实验报告中回答即可）

f) 不刻意地用优先权编码器或数据选择器的概念，直接用 Verilog HDL 语言 RTL 级**编程并实现** 8-bit 整数定点数转换为浮点数，并合理设计人机接口；（可以仅考虑无符号整数）

g) 观察综合工具中的报告，了解或估算不同规模下不同编程方法下的实现所消耗的可编程逻辑资源，包括：门级 4-bit、RTL 级 4-bit、RTL 级 8-bit。

本实验将采取**实验报告和现场汇报演示相结合**的形式，实验报告上交的具体**截止期限**请注意任课教师的通知，实验报告格式请参考本文档；**现场汇报演示**的时间和分组情况请注意任课老师的通知。特此广而告之。

1.4 提示和扩展要求

提示：

(1) 在门级实现中，建议先模仿优先级编码器 74148 和双四选一多路复用器 74153 的功能实现电路模块，其中优先级编码器电路模块也被 4-bit 和 8-bit 的设计复用；

(2) 但是在资源比较的时候，门级实现建议采用最简组合逻辑函数，或是采用多路复用器实现，避免优先级编码器 74148 的部分逻辑功能未使用造成的资源冗余；

(3) 组合逻辑在程序实现中，请权衡代码的封装和复杂性的代价，灵活地以连续赋值或过程块实现，必要地时候采用模块封装。

扩展的实验要求：（根据完成的程度可酌情提高评价的等级）

h) 进一步熟悉利用“实验板”的资源，即：改造实验中心样例程序中的 Micro USB 构建的 UART（或找到更好的 UART 程序），形成 UART 交互人机接口，进行定点数和浮点数的输入输出；

i) 如果作为一种“偷懒”的转换方式，可以采用查表的方法将 8-bit 整数定点数转换为浮点数，请编写计算机程序（可以用 C 语言）生成并打印 8-bit 编码所对应的 256 字数×11 位存储表

（为什么是 11 位呢？请说明），并加以实现；

j) 思考一下如果定点数的位数加长到 32-bit，使得直接用组合逻辑实现多路复用器消耗资源过大，如何兼顾逻辑运算的速度和资源的复用，添加寄存器进行设计？（可以不用编程或实现，仅在实验报告中回答即可）。

1.5 其它提示

在实验中心提供的样例程序中，UART 的样例源代码具有固定的 9600 波特率、8 位数据位、偶校验、1 位停止位，缺点是一个只能每次读写一个字节的；更好的 UART（注：更稳定可靠、波特率可配置、有 Telnet 等上层协议）需要用户自行开发，或者在网上搜索更好的代码加以改造。

“实验板”的 I/O 接口和人机接口资源请参考《digiC2018 课程实验实验指导书(01)》。

实验二、组合逻辑设计——定、浮点数转换（实验报告部分）

2 实验报告

2.1 实验背景与需求分析

2.1.1 实验背景

浮点数格式是一种计算机数据格式，利用“浮点”（浮动小数点）的方法，可以近似表示任意某个实数。具体的说，这个实数由一个整数或定点数（即尾数）乘以某个基数（计算机中通常是 2）的整数次幂得到，这种表示方法类似于基数为 10 的科学计数法。

2.1.2 需求分析

输入的是一个二进制无符号定点数，需要输出对应浮点数的尾数和幂指数，利用 74148 元件和 74153 元件进行设计，先使用 Logisim 画出电路图，再根据电路图，对这两个器件分别采用门级和 RTL 级描述，实现 4-bit 定浮点数转换。设计人机接口，用 4 个 LED 灯表示输入值，用数码管显示有效位和幂指数。

若不使用 74148 和 74153，直接进行 RTL 级编程实现 8-bit 定浮点数的转换，利用 casex 语句确定第一个有效位的位置，之后便可以进行移位和计算幂指数。若要用查表的方法实现 8-bit 定浮点数转换，只需用 C 语言打印出所对应的 256 字数×11 位查找表，然后在 Verilog 中利用 case 语句直接访问，可以有效减少门电路造成的延迟。

2.2 系统设计

2.2.1 总体设计思路

本实验的主体设计思想是将各种不同的浮点数转换模块综合到开发板上，并且设计合理的人机接口，利用开发板的按键、数码管以及 UART 串口通信使其具有定、浮点数转换的功能。

在门级 4-bit 定浮点数转换模块中，我们分别模仿优先级编码器 74148 和双四选一多路复用器 74153 的功能实现了电路模块，使用组合电路搭建出定浮点数转换器；在 RTL 级 4-bit 定浮点数转换模块中则使用 RTL 级编程重新实现了 74148 和 74153 的功能，再使用相同的电路搭建。在 RTL 级 8-bit 定浮

点数转换模块中，我们通过将多路选择器和移位操作相结合，用较为简单的方式实现了相应功能，最后作为拓展功能，我们用 C 语言打印出了 8-bit 定浮点数转换查询表，用 Look-Up-Table（LUT）的方法实现 8-bit 定浮点数转换功能。

除此之外，本实验还设计了多种人机接口，一方面可以通过开发板上的按键和拨码开关控制定浮点数的转换，并且用数码管结合 LED 灯显示出所需信息，另一方面，我们还修改了实验中心提供的 UART 串口通信样例程序，通过更改状态机的工作方式，完成支持多位数据传输的 UART 通信程序，实现了可以在串口助手中输入无符号整数，然后打印出浮点数的功能，这样可以更加直观地展现出该电路的工作方式。

表 3 已实现的定浮点数转换模块

门级 4-bit		RTL 级 4-bit		RTL 级 8-bit	LUT 查表 8-bit
门级 74148	门级 74153	RTL 级 74148	RTL 级 74153		

2.2.2 接口设计

实验板提供的 IO 分别有系统时钟，复位，拨码开关，按键，数码管管脚，选通端，LED 灯和 UART 的发送端 Tx 以及接收端 Rx，这些接口必须先在顶层模块中定义，由于按键去抖模块和数码管显示模块均需要时钟信号控制，所以也将系统时钟作为他们的输入。除此之外，按键去抖模块还需要按键信号输入，输出去抖后的按键信号，同时也在必要的时候将复位信号分配给相应的模块。

所有的定浮点数转换模块都与系统时钟信号无关，均可以在一个时钟周期内完成，以 4-bit 或 8-bit 无符号整数为输入，输出相应浮点数的尾数和指数。在 4-bit 转换模式，数码管模块以尾数和指数作为输入，LED 显示模块以无符号整数作为输入；在 8-bit 转换模式，预置时数码管模块以无符号整数作为输入，转换时数码管模块以尾数作为输入，LED 显示模块以指数作为输入。数码管模块和 LED 显示模块输出相应的管脚控制信号。

2.2.3 按键预置模块

首先通过第一路拨码开关控制是 4-bit 转换模式还是 8-bit 转换模式，不同模式下按键所控制的寄存器也不同。在 4-bit 转换模式下，两个按键可以对一个 4 位寄存器加一减一，而 8-bit 转换模式下两个按键控制 8 位寄存器的加减。在 8-bit 转换模式下，第二路拨码开关在低位时显示定点数，高位时显示转换后的浮点数。

```
if (switch[1] == 1'b0) begin // 4-bit 转换模式
   if (key_out[1] == 1'b1) // 加一
```

```
            num <= num + 1;
        else if (key_out[0] == 1'b1) // 减一
            num <= num - 1;
        if (switch[0] == 1'b0) begin // 门级电路
            display_1 <= gate_F;
            display_2 <= gate_P;
        end
        else begin // RTL级电路
            display_1 <= rtl_F;
            display_2 <= rtl_P;
        end
        led <= num;
    end
    else begin // 8-bit转换模式
        if (key_out[1] == 1'b1) // 加一
            num_8bit <= num_8bit + 1;
        else if (key_out[0] == 1'b1) // 减一
            num_8bit <= num_8bit - 1;
        if (switch[0] == 1'b0) begin // 预置
            display_1 <= num_8bit[7:4];
            display_2 <= num_8bit[3:0];
            led <= 4'b0;
        end
        else begin // 转换
            display_1 <= F_8bit[7:4];
            display_2 <= F_8bit[3:0];
            led <= P_8bit;
        end
    end
```

2.2.4 门级 4-bit 定浮点数转换模块

由给出的电路图可以得知，门级 4-bit 定浮点数转换模块由 74HC148 优先权编码器和 74HC153 双四选一数据选择器及其他一些门电路构成，则我们首先通过构建这两个电路模块，最后再以组合电路综合即可。我们首先用 Logisim 模拟 74HC148 优先权编码器的内部电路图：

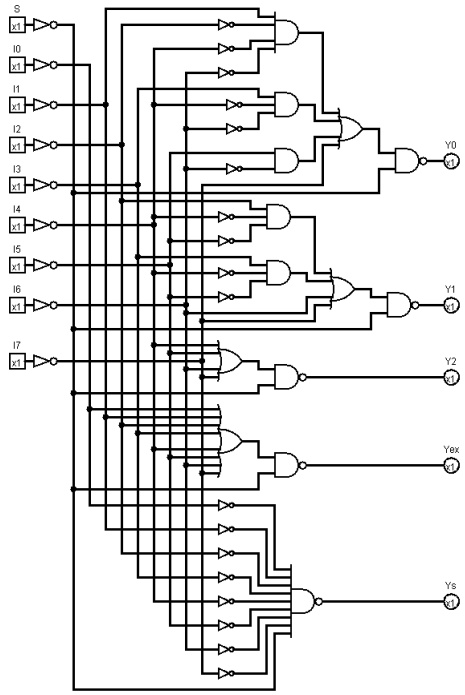


图 4 74HC148 优先权编码器电路图

然后根据电路图依次用逻辑门实现即可：

```
module TTL74148(nS, nIN, Y, Ys, Yex);
    input nS;
    input [7:0] nIN;
    output [2:0] Y;
    output Ys;
    output Yex;

    wire S;
    wire [7:0] IN;
    wire [4:0] aIN;
    wire [2:0] oIN;
    wire [2:0] aS;
    wire aYs, aYex;
    genvar i;

    not(S, nS);

    generate
        for (i = 0; i < 8; i=i+1) begin:loop1
            not(IN[i], nIN[i]);
        end
    endgenerate

    or(oIN[2], IN[4], IN[5], IN[6], IN[7]);
    and(aIN[4], IN[2], nIN[4], nIN[5]);
```

```
    and(aIN[3], IN[3], nIN[4], nIN[5]);
    or(oIN[1], aIN[4], aIN[3], IN[6], IN[7]);
    and(aIN[2], IN[1], nIN[2], nIN[4], nIN[6]);
    and(aIN[1], IN[3], nIN[4], nIN[6]);
    and(aIN[0], IN[5], nIN[6]);
    or(oIN[0], aIN[2], aIN[1], aIN[0], IN[7]);

    generate
        for (i = 0; i < 3; i=i+1) begin:loop2
            and(aS[i], S, oIN[i]);
            not(Y[i], aS[i]);
        end
    endgenerate

    and(aYs, nIN[0], nIN[1], nIN[2], nIN[3], nIN[4], nIN[5], nIN[6], nIN[7], S);
    not(Ys, aYs);
    and(aYex, Ys, S);
    not(Yex, aYex);
endmodule
```

接着模拟 74HC153 双四选一数据选择器的内部电路图：

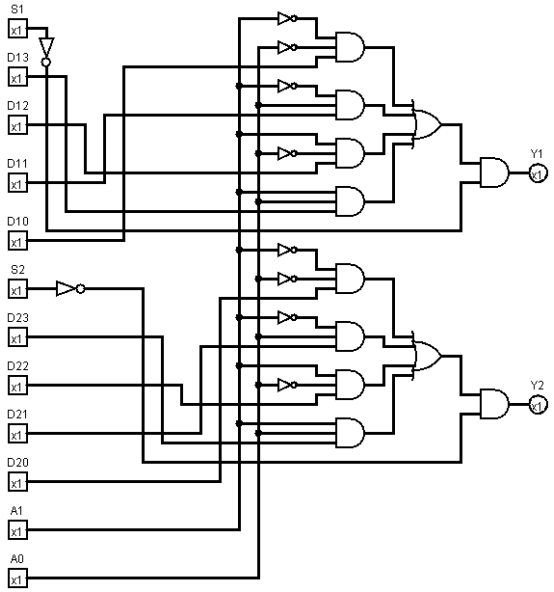


图 5 74HC153 双四选一数据选择器电路图

同理，根据电路图通过 Verilog 自带的门电路模块实现：

```
module TTL74153(nS, IN, A, Y);
    input [1:0] nS;
    input [7:0] IN;
    input [1:0] A;
    output [1:0] Y;
```

```
    wire [1:0] S;
    wire [7:0] aD;
    wire [1:0] nA;
    wire [1:0] oS;

    not(S[1], nS[1]);
    not(S[0], nS[0]);
    not(nA[1], A[1]);
    not(nA[0], A[0]);

    and(aD[7], IN[7], A[1], A[0]);
    and(aD[6], IN[6], A[1], nA[0]);
    and(aD[5], IN[5], nA[1], A[0]);
    and(aD[4], IN[4], nA[1], nA[0]);
    and(aD[3], IN[3], A[1], A[0]);
    and(aD[2], IN[2], A[1], nA[0]);
    and(aD[1], IN[1], nA[1], A[0]);
    and(aD[0], IN[0], nA[1], nA[0]);

    or(oS[1], aD[7], aD[6], aD[5], aD[4]);
    or(oS[0], aD[3], aD[2], aD[1], aD[0]);

    and(Y[1], oS[1], S[1]);
    and(Y[0], oS[0], S[0]);
endmodule
```

最后构建组合电路如下：

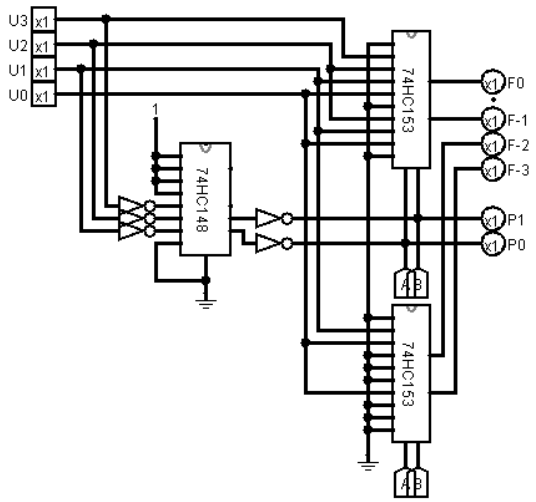


图 6 4-bit 定浮点数转换电路图

用门级编程描述组合电路：

```
module Float(U, F, P);
    input [3:0] U;
```

```
    output [3:0] F;
    output [1:0] P;

    wire [3:0] nU;
    wire [7:0] IN_74153_H;
    wire [7:0] IN_74153_L;
    wire [7:0] IN_74148;
    wire [2:0] Y_74148;
    wire Ys, Yex;

    genvar i;
    generate
        for (i = 0; i < 4; i=i+1) begin:forloop
            not(nU[i], U[i]);
        end
    endgenerate

    assign IN_74148 = {4'b1111, nU[3:1], 1'b0};
    TTL74148 ttl_74148 (
        .nS(1'b0),
        .nIN(IN_74148),
        .Y(Y_74148),
        .Ys(Ys),
        .Yex(Yex)
    );

    not(P[1], Y_74148[1]);
    not(P[0], Y_74148[0]);
    assign IN_74153_H = {U[2:0], 1'b0, U[3:0]};
    assign IN_74153_L = {U[0], 3'b0, U[1:0], 2'b0};

    TTL74153 ttl_74153_H (
        .nS(2'b0),
        .IN(IN_74153_H),
        .A(P),
        .Y({F[2], F[3]})
    );
    TTL74153 ttl_74153_L (
        .nS(2'b0),
        .IN(IN_74153_L),
        .A(P),
        .Y({F[0], F[1]})
    );
endmodule
```

2.2.5 RTL 级 4-bit 定浮点数转换模块

对于 RTL 级 4-bit 定浮点数转换模块的实现，我们可以采用门级编程相同的思路，利用 74148 元件和 74153 元件构建转换模块，此时我们用 RTL 级描述元件功能。

74148 元件的 RTL 级描述如下：

```
module TTL74148_RTL(nS, nIN, Y, Ys, Yex);
   input nS;
   input [7:0] nIN;
   output reg [2:0] Y;
   output reg Ys;
   output reg Yex;

   always @(nS, nIN) begin
      if (nS) begin Y<=3'b111;Yex<=1'b1;Ys<=1'b1; end
      else if (nIN == 8'b11111111) begin Y<=3'b111;Yex<=1'b1;Ys<=1'b0; end
      else if (!nIN[7]) begin Y<=3'b000;Yex<=1'b0;Ys<=1'b1; end
      else if (!nIN[6]) begin Y<=3'b001;Yex<=1'b0;Ys<=1'b1; end
      else if (!nIN[5]) begin Y<=3'b010;Yex<=1'b0;Ys<=1'b1; end
      else if (!nIN[4]) begin Y<=3'b011;Yex<=1'b0;Ys<=1'b1; end
      else if (!nIN[3]) begin Y<=3'b100;Yex<=1'b0;Ys<=1'b1; end
      else if (!nIN[2]) begin Y<=3'b101;Yex<=1'b0;Ys<=1'b1; end
      else if (!nIN[1]) begin Y<=3'b110;Yex<=1'b0;Ys<=1'b1; end
      else begin Y<=3'b111;Yex<=1'b0;Ys<=1'b1; end
   end
endmodule
```

74153 元件的 RTL 级描述如下：

```
module TTL74153_RTL(nS, IN, A, Y);
   input [1:0] nS;
   input [7:0] IN;
   input [1:0] A;
   output reg [1:0] Y;

   always @(nS, IN, A) begin
      if (!nS[1])
         Y[1] <= IN[A+4];
      if (!nS[0])
         Y[0] <= IN[A];
   end
endmodule
```

最终的组合电路实现与门级类似，将 74148 和 74153 元件模块实例化即可，故不再此赘述。

2.2.6 RTL 级 8-bit 定浮点数转换模块

接下来不刻意地用优先权编码器或数据选择器的概念，直接用 RTL 级编程并实现 8-bit 整数定点数转换为浮点数，利用 Verilog 语言的特性，我们使用 casex 语句以及移位的操作，可以容易地实现该模块功能：

```
module Float_8bit(U, F, P);
   input [7:0] U;
   output reg [7:0] F;
   output reg [2:0] P;

   always @(*) begin
      casex (U)
         8'b1???????: begin P <= 3'b111; F <= U; end
         8'b01??????: begin P <= 3'b110; F <= U << 1; end
         8'b001?????: begin P <= 3'b101; F <= U << 2; end
         8'b0001????: begin P <= 3'b100; F <= U << 3; end
         8'b00001???: begin P <= 3'b011; F <= U << 4; end
         8'b000001??: begin P <= 3'b010; F <= U << 5; end
         8'b0000001?: begin P <= 3'b001; F <= U << 6; end
         8'b00000000?: begin P <= 3'b000; F <= U << 7; end
      endcase
   end
endmodule
```

2.2.7 查表法 8-bit 定浮点数转换模块

首先用 C 语言设计编写打印查询表的程序，建立一个 256*11 的二维数组，设置一个计数器 i，而 transfer 函数每次将它转化为二进制，length 函数求出二进制数的长度后进行移位并求出幂指数，最后打印出二维数组即可，我们可以直接打印为 Verilog HDL 语法形式，方便复制到项目中。

```
#include <cstdio>

int transfer(int x)
{
    int p = 1, y = 0, r;
    while (x > 0) {
        r = x & 1;
        x >>= 1;
        y += r * p;
        p *= 10;
    }
    return y;
}
```

```
int length(int n)
{
    int sum = 0;
    while (n) {
        n /= 10;
        sum++;
    }
    return sum;
}

int main()
{
   int binary; // 二进制数
   int L; // 二进制数长度
   int power; // 幂指数
   int table[256][11]; // 储存 8 位二进制数对应表

   for(int i = 0; i < 256; i++){
      binary = transfer(i);
      L = length(binary);
      power = transfer(L-1);

      for(int j = 7; j >= 0; j--){
         table[i][j] = binary % 10;
         binary = binary / 10;
      }
      for(int j = 10; j >= 8; j--){
         table[i][j] = power % 10;
         power = power / 10;
      }
   }
   table[0][10]=0;
   for(int x = 0; x < 256; x++)
   {
      // 输出移位之后的结果
      printf("8\'d%d:\tbegin ", x);
      printf("F <= 8\'b");
      for(int i = 8-length(transfer(x)); i < 8; i++)
         printf("%d",table[x][i]);
      for(int i = 0; i < 8-length(transfer(x)); i++)
         printf("%d",table[x][i]);
      printf("; P <= 3\'b");
      // 输出幂指数
```

```
        for(int i = 8; i < 11; i++)
            printf("%d",table[x][i]);
        printf("; end\n");
    }
}
```

然后在 Verilog 中利用 case 语句查表，可以提高运行速度：

```
module Float_8bit_Table(U, F, P);
    input [7:0] U;
    output reg [7:0] F;
    output reg [2:0] P;

    always @(U) begin
        case (U)
            8'd0:   begin F <= 8'b00000000; P <= 3'b000; end
            8'd1:   begin F <= 8'b10000000; P <= 3'b000; end
            8'd2:   begin F <= 8'b10000000; P <= 3'b001; end
                        ......................
            8'd254: begin F <= 8'b11111110; P <= 3'b111; end
            8'd255: begin F <= 8'b11111111; P <= 3'b111; end
        endcase
    end
endmodule
```

2.2.8 数码管及 LED 灯显示模块

在 4-bit 转换模式下，需要让数码管分别显示浮点数的尾数和指数，LED 灯显示定点数；8-bit 转换模式下，预置时让数码管显示 8 位定点数，转换时让数码管显示浮点数的尾数，LED 灯显示指数。

由于 FPGA 上有两位数码管，我们采用动态扫描刷新，让两个数码管交替刷新显示数字，该模块以需要显示的数字，以及系统时钟和复位作为输入，并且输出数码管的管脚和位段。

首先建立时钟分频模块，新的时钟间隔为 0.02 秒：

```
parameter update_interval = 50000000 / 200 - 1;
always @(posedge clk or negedge rst) begin
    if (!rst) begin
        cnt <= 0;
        sel <= 0;
    end
    else begin
        cnt <= cnt + 1;
        if (cnt == update_interval) begin
            cnt <= 0;
            sel <= ~sel;
        end
```

```
    end
end
```

然后在每次时钟信号改变时，切换选通端，并且将两个输入数据存储在待显示数字的寄存器中，此时数码管刷新频率为 50Hz，满足要求：

```
always @(*) begin
    case (sel)
        1'b0: begin dat <= number1; com <= 2'b01; end
        1'b1: begin dat <= number2; com <= 2'b10; end
    endcase
end
```

最后通过控制管脚的信号让相应数码管显示数字即可：

```
always @(dat) begin
    seg[0] <= 1'b0;
    case (dat)
        4'h0: seg[7:1] <= 7'b1111110;
        4'h1: seg[7:1] <= 7'b0110000;
        4'h2: seg[7:1] <= 7'b1101101;
        4'h3: seg[7:1] <= 7'b1111001;
        4'h4: seg[7:1] <= 7'b0110011;
        4'h5: seg[7:1] <= 7'b1011011;
        4'h6: seg[7:1] <= 7'b1011111;
        4'h7: seg[7:1] <= 7'b1110000;
        4'h8: seg[7:1] <= 7'b1111111;
        4'h9: seg[7:1] <= 7'b1111011;
        4'hA: seg[7:1] <= 7'b1110111;
        4'hB: seg[7:1] <= 7'b0011111;
        4'hC: seg[7:1] <= 7'b1001110;
        4'hD: seg[7:1] <= 7'b0111101;
        4'hE: seg[7:1] <= 7'b1001111;
        4'hF: seg[7:1] <= 7'b1000111;
        default: seg[7:1] <= 7'b0000000;
    endcase
end
```

LED 灯显示模块则直接在不同的模式下将需要显示的值存储到 led 寄存器即可。

2.2.9 UART 串口通信模块

我们改进了实验中心提供的 UART 串口通信模块，实现了多位数据传输功能，主要工作是将原先的状态机工作方式做了一定修改：

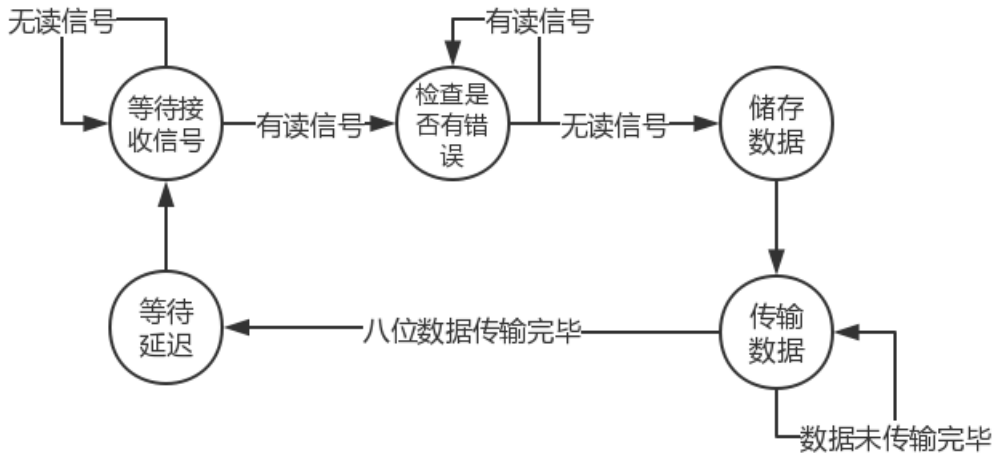


图 7 原状态机的状态转换图

新的状态机工作方式如下：

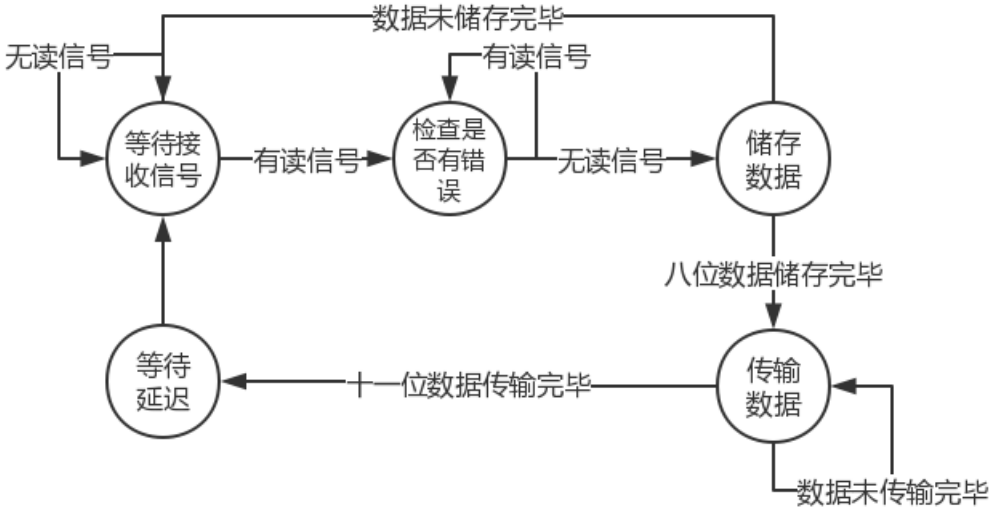


图 8 改进后状态机的状态转换图

我们根据状态转换图，对 Uart_Top 模块做了如下修改：

```
case(State)
   Wait_Rdsig:  begin // 等待接收信号
                   if (!Rdsig)
                       State = Wait_Rdsig;
                   else
                       State = Check_Data;
                end
   Check_Data:  begin // 检查是否有错误
                   if (Rdsig)
                       State = Check_Data;
```

```
                else begin
                    Error_Flag = Rx_DataError_Flag || Rx_FrameError_Flag;
                    State = Save_Data;
                end
            end
Save_Data:  begin // 储存数据
                if (!Error_Flag) begin
                    if (Data_Cnt == 4'd8) begin // 接收完毕
                        Data_Cnt = 4'd0;
                        State = Trans_Data;
                    end
                    else begin
                        State = Wait_Rdsig;
                        Data_Reg[Data_Cnt] = Data_Rx;
                        Data_Cnt = Data_Cnt + 1;
                    end
                end
                else
                    State = Trans_Data;
            end
Trans_Data: begin // 传输数据
                if (!Error_Flag) begin
                    if (Print_Cnt == 4'd11) begin // 输出完毕
                        Print_Cnt = 4'd0;
                        State = State_Delay;
                    end
                    else begin
                        State = Trans_Data;
                        if (cnt == 254) begin // 进一步分频
                            Data_Tx = Print_Reg[Print_Cnt];
                            Wrsig = 1'b1;
                            cnt = 8'd0;
                            Print_Cnt = Print_Cnt + 1;
                        end
                        else begin
                            Wrsig = 1'b0;
                            cnt = cnt + 8'd1;
                        end
                    end
                end
                else begin
                    if (Char_Cnt == 4'd4) begin // 输出完毕
                        Char_Cnt = 4'd0;
                        State = State_Delay;
```

```
                    end
                    else begin
                        State = Trans_Data;
                        if (cnt == 254) begin // 进一步分频
                            Data_Tx = Error_Char[Char_Cnt];
                            Wrsig = 1'b1;
                            cnt = 8'd0;
                            Char_Cnt = Char_Cnt + 1;
                        end
                        else begin
                            Wrsig = 1'b0;
                            cnt = cnt + 8'd1;
                        end
                    end
                end
            end
    State_Delay:begin // 等待延迟
                    if (cnt == 254) begin // 输出回车
                        Data_Tx = 8'd13;
                        Wrsig = 1'b1;
                        cnt = 8'd0;
                        State = Wait_Rdsig;
                    end
                    else begin
                        Wrsig = 1'b0;
                        cnt = cnt + 8'd1;
                    end
                end
endcase
```

而传输和接收信号模块 Uart_Tx 和 Uart_Rx 无需进行修改。

2.3 功能仿真测试

2.3.1 测试程序设计

我们为每个定浮点数转换模块都建立了各自的 test bench，利用 ISE 提供的 ISim 工作台来仿真系统的工作方式，通过观察信号即可分析系统的工作状态。由于不同定浮点数转换模块的测试程序大同小异，我们以 4-bit 定浮点数转换模块的测试程序为例。以下为测试程序的主体内容：

```
initial begin
        // 初始化输入
        U = 0;
```

```
        // 延迟 100ns 以等待全局复位完成
        #100;
        // 添加激励信号
        U = 4'b0001;
        #5;
        U = 4'b0010;
        #5;
        U = 4'b0111;
        #5;
        U = 4'b1010;
        #5;
        U = 4'b1100;
        #5;
        U = 4'b1111;
    end
```

2.3.2 功能仿真过程

通过 ISim 得到的仿真波形图如下：

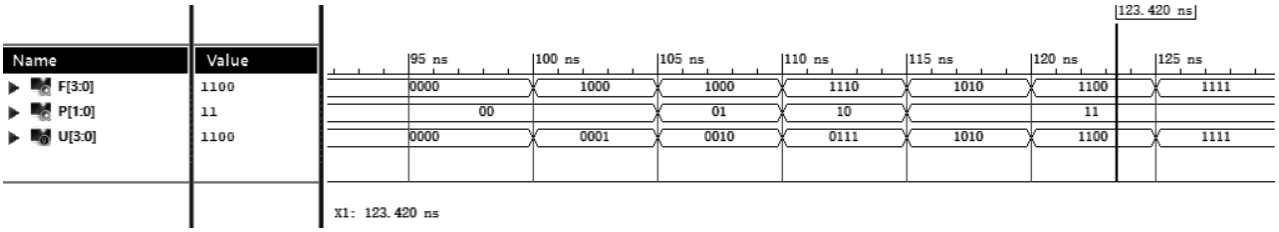


图 9 仿真波形图

2.3.2 实验关键结果及其解释

观察波形图，可以看到当无符号二进制整数 U 的内容改变时，浮点数的尾数 F 和指数 P 改变为相应的值，说明系统工作正常。

2.4 设计实现

2.4.1 综合和下载过程

语法检验通过后，通过 PlanAhead 进行引脚分配。将系统用到的 8 个数码管引脚，2 个数码管片选信号引脚，2 个按键信号引脚，4 个 LED 引脚，2 个拨码开关引脚、UART 通信的 Tx 端、Rx 端和时钟信号、复位信号绑定到实验板上，分配好之后生成 Top.ucf 文件。

Name	Direction	Neg Diff Pair	Site	Fixed
All ports (20)				
com (2)	Output			
com[1]	Output		P27	☑
com[0]	Output		P26	☑
key (2)	Input			
led (4)	Output			
seg (8)	Output			
switch (2)	Input			
Scalar ports (2)				

图 10 引脚分配

然后在 ISE 中进行综合，观察电路原理图：

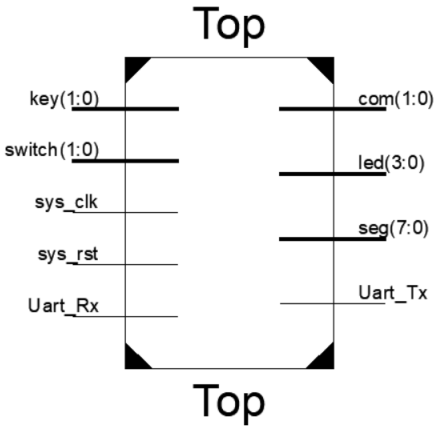


图 11 电路原理图

最后在 iMPACT 中将程序烧录到实验板中：

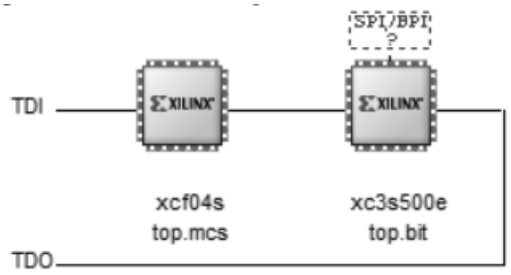


图 12 程序烧录

2.4.2 实验关键结果及其解释

接通电源后，在第一路拨码开关为“0”时，为 4-bit 转换模式，通过两个按键分别进行加减设定数字，这时 LED 灯显示当前四位无符号二进制整数，同时数码管第一位显示浮点数尾数的十六进制表示，第二位显示指数的十六进制表示。将第一路拨码开关打到“1”，这时进入 8-bit 转换模式，第二路拨码开关为“0”时通过两个按键分别加减设定数字，这时两位数码管显示当前八位无符号二进制整数的十六进制表示，将第二路拨码开关达到“1”，两位数码管显示对应浮点数尾数的十六进制表示，LED

灯显示浮点数的指数。

接着我们使用串口通信，打开 AccessPort 连接到实验板，发送信号测试 UART 串口通信以及 8-bit 定浮点数转换的功能，测试结果如下图所示：

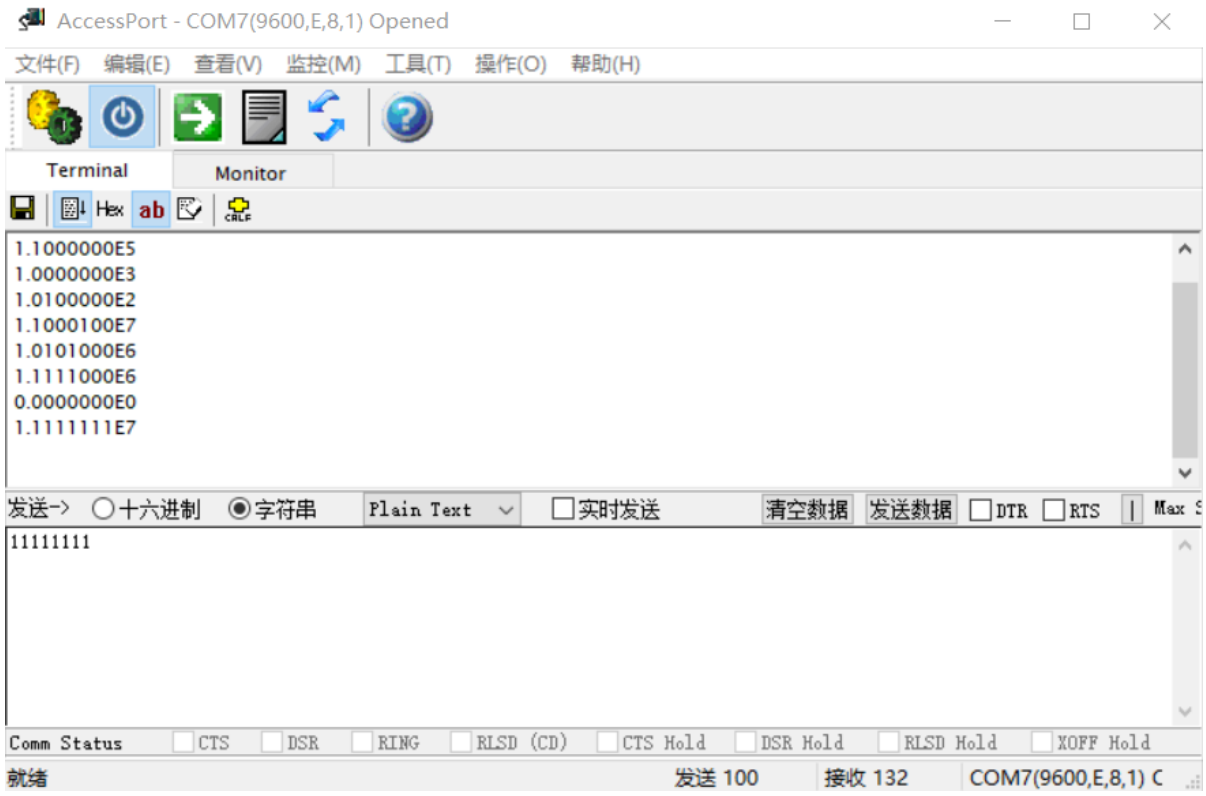


图 13 UART 串口通信测试

2.5 资源估算与电路设计

2.5.1 门级 8-bit 定浮点数转换 MUX 资源估算

我们认为需要双四选一数据选择器 74HC153 **共 6 个**即可实现 8-bit 无符号整数到浮点数的转换。其本质思考模式及方法并没有变化，仅仅是每一位输出需要通过双四选一数据选择器的 8 位输入来决定而非 4 位，当然，最后可以优化一下减少 2 个双四选一数据选择器。这样设计的好处是可以模块化，只要有一个 n 位优先权编码器和多个 n 位数据选择器，则可以实现 n 位符号数转浮点数，方法与上面的类似。

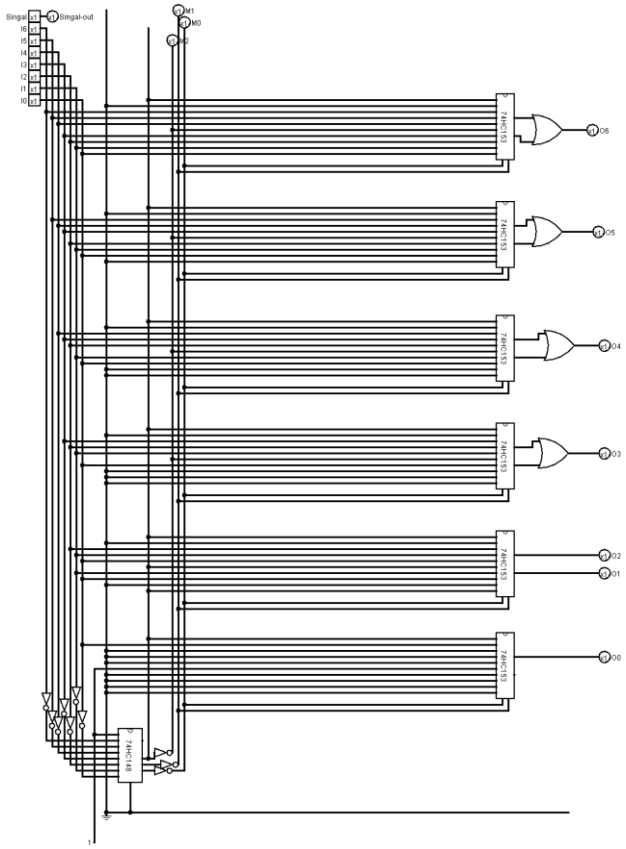


图 14 门级 8-bit 定浮点数转换电路实现

2.5.2 门级 4-bit、RTL 级 4-bit 和 8-bit 定浮点数转换 PLD 资源估算

我们分别对门级实现的 4-bit、RTL 级实现的 4-bit 和 RTL 级实现的 8-bit 定浮点数转换 Verilog 程序在 ISE 中综合，其报告中的部分数据如下：

门级 4-bit 综合报告：

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	86	9,312	1%	
Number of 4 input LUTs	108	9,312	1%	
Number of occupied Slices	70	4,656	1%	
Number of Slices containing only related logic	70	70	100%	
Number of Slices containing unrelated logic	0	70	0%	
Total Number of 4 input LUTs	112	9,312	1%	
Number used as logic	108			
Number used as a route-thru	4			
Number of bonded IOBs	17	66	25%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.41			

图 15 门级 4-bit 定浮点数转换 PLD 硬件资源估算

Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Yes	Autotimespec constraint for clock net sys_clk_BUFGP	SET...	1.304ns	7.518ns	0	0 0

图 16 门级 4-bit 定浮点数转换稳定性分析

```
Design statistics:
   Minimum period:   7.605ns{1}   (Maximum frequency: 131.492MHz)
   Minimum input required time before clock:   3.730ns
   Maximum output delay after clock:   8.849ns
```

图 17 门级 4-bit 定浮点数转换延迟估算

RTL 级 4-bit 综合报告：

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	92	9,312	1%	
Number of 4 input LUTs	113	9,312	1%	
Number of occupied Slices	73	4,656	1%	
Number of Slices containing only related logic	73	73	100%	
Number of Slices containing unrelated logic	0	73	0%	
Total Number of 4 input LUTs	117	9,312	1%	
Number used as logic	113			
Number used as a route-thru	4			
Number of bonded IOBs	17	66	25%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.38			

图 18 RTL 级 4-bit 定浮点数转换 PLD 硬件资源估算

Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Yes	Autotimespec constraint for clock net sys_clk_BUFGP	SET...	1.013ns	7.430ns	0	0 0

图 19 RTL 级 4-bit 定浮点数转换稳定性分析

```
Design statistics:
   Minimum period:   7.318ns{1}   (Maximum frequency: 136.649MHz)
   Minimum input required time before clock:   1.661ns
   Maximum output delay after clock:   9.167ns
```

图 20 RTL 级 4-bit 定浮点数转换延迟估算

RTL 级 8-bit 综合报告：

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	97	9,312	1%	
Number of 4 input LUTs	148	9,312	1%	
Number of occupied Slices	97	4,656	2%	
Number of Slices containing only related logic	97	97	100%	
Number of Slices containing unrelated logic	0	97	0%	
Total Number of 4 input LUTs	152	9,312	1%	
Number used as logic	148			
Number used as a route-thru	4			
Number of bonded IOBs	18	66	27%	
IOB Flip Flops	1			
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.42			

图 21 RTL 级 8-bit 定浮点数转换 PLD 硬件资源估算

Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Yes	Autotimespec constraint for clock net sys_clk_BUFGP	SET...	1.013ns	7.703ns	0	0 0

图 22 RTL 级 8-bit 定浮点数转换 PLD 稳定性分析

```
Design statistics:
   Minimum period:   7.518ns{1}   (Maximum frequency: 133.014MHz)
   Minimum input required time before clock:   4.366ns
   Maximum output delay after clock:  10.492ns
```

图 23 RTL 级 8-bit 定浮点数转换延迟估算

由于 FPGA 运行类似木桶短板，所以 Design statistics 中的 Maximum output delay after clock 是程序运行速度的决定量。

首先比较 4-bit 的门级实现和 RTL 级实现的区别。不难发现，门级编程所占用的 Flip-Flops 和 LUT 都比 RTL 级编程的少，而且效率也高。而且由 Worst Case Slack 和 Best Case Achievable 可以看到数据的稳定性也较好。

再比较 4-bit 和 8-bit 的 RTL 级实现，相应的，8-bit 所需要的 Flip-Flops 和 LUT 都比 4-bit 多，效率更低，数据稳定性也相对差些。

2.5.3 32-bit 定浮点数转换电路设计

分析：实际上定点数转换为浮点数的关键是判断第一位不为 0 的数位，而其本质实现方法是移位。而之所以直接使用组合逻辑实现多路复用器消耗资源过大，最根本的原因在于组合逻辑难以直接保存上一个电路状态——如果直接移位的话，一定是将下一位的状态赋给上一个，而组合逻辑难以直接保存下一位的状态，所以只能用更复杂的办法实现。

而如果允许添加使用寄存器，则可以保存上一个电路状态，无论下一个状态是否改变，这样就极大的简化了电路思路——除非第一位非 0 或全部位为 0，则不停地移位，移位可通过移位寄存器的类似思路来实现，改造 RS 锁存器的驱动方程即可。

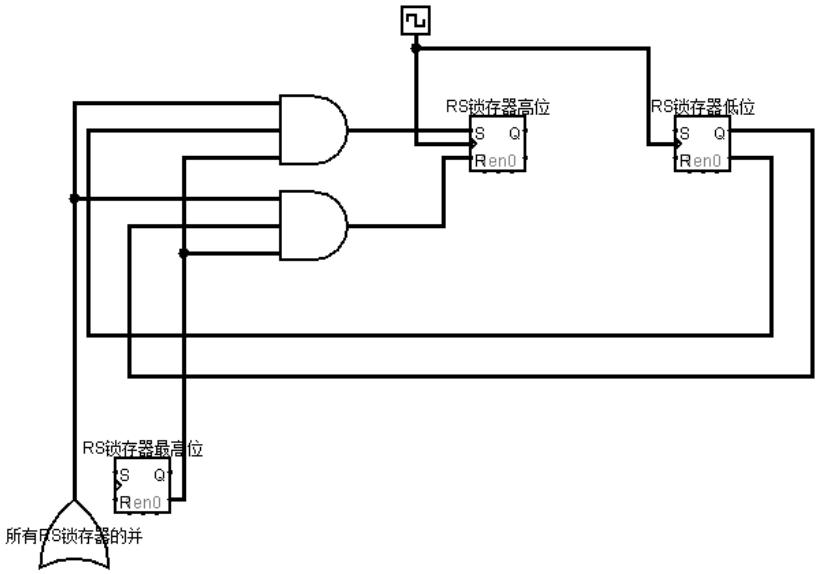


图 24 改造后的移位寄存器单元

这是一个寄存器的实现，可以看出若最高位不为 0 或全部位为 0 时，所有锁存器的 S=R=0，即为保持，否则，仅仅是将低位传给高位，实现了按要求移位。若是要多位并行输入，仅需模仿移位寄存器的多位输入即可。

当然，这是资源消耗最少的方式，但时间会变长，如果要移动 n 位的话，需要等待 n 个时钟信号，所以若要同时兼顾时间和速度，可以将高位部分用寄存器移位的方法得出，低位部分用组合逻辑得出。而低位组合逻辑电路设计思路和 8-bit 定浮点数转换的设计思路相似，高位寄存器的设计则基本是完全一样的。

我们还可以用 Look-Up-Table 的方法实现低位的定浮点数转换，将 32-bit 分为 4 个 8-bit，每个 8-bit 利用查表得出幂指数 power，前三个 8-bit 的 power 与最低位相或，防止 00000001 被认作是无有效数字，之后由四个或门得出的结果即可知道最高位的位置，从而进行移位操作得到幂指数的输出。

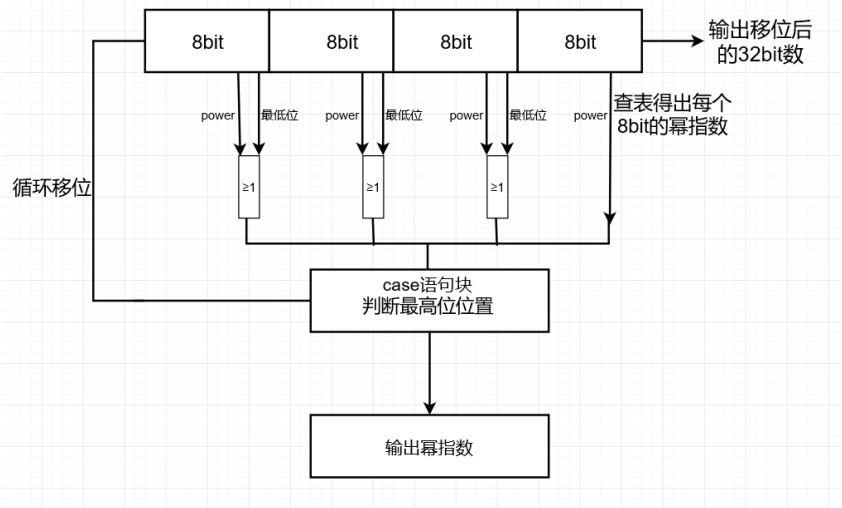


图 25 低位组合电路实现 32-bit 定浮点数转换

2.6 小结

通过此次实验，我们学习到的知识有：

1. 计算机中浮点数的表示原理。
2. 进一步熟悉了 UART 串口通信。
3. 了解了门级编程和 RTL 级编程的概念和区别。
4. 熟练了 ISE 和 Modelsim 的操作。
5. 对书中的一些元器件的应用有了新理解，比如优先权编码器和数据选择器。
6. 如何对资源和速度进行协调分配。

附注：本次实验所用代码已经上传到 Github 仓库：

https://github.com/hiyouga/digiC-experiment/tree/master/ex2