

中国科学技术大学计算机学院
《数字电路实验》报告



实验题目：____综合实验____

学生姓名：____徐奥____

学生学号：____PB20061343____

完成日期：2021 年 12 月 19 日

计算机实验教学中心制

2020 年 09 月

【实验题目】

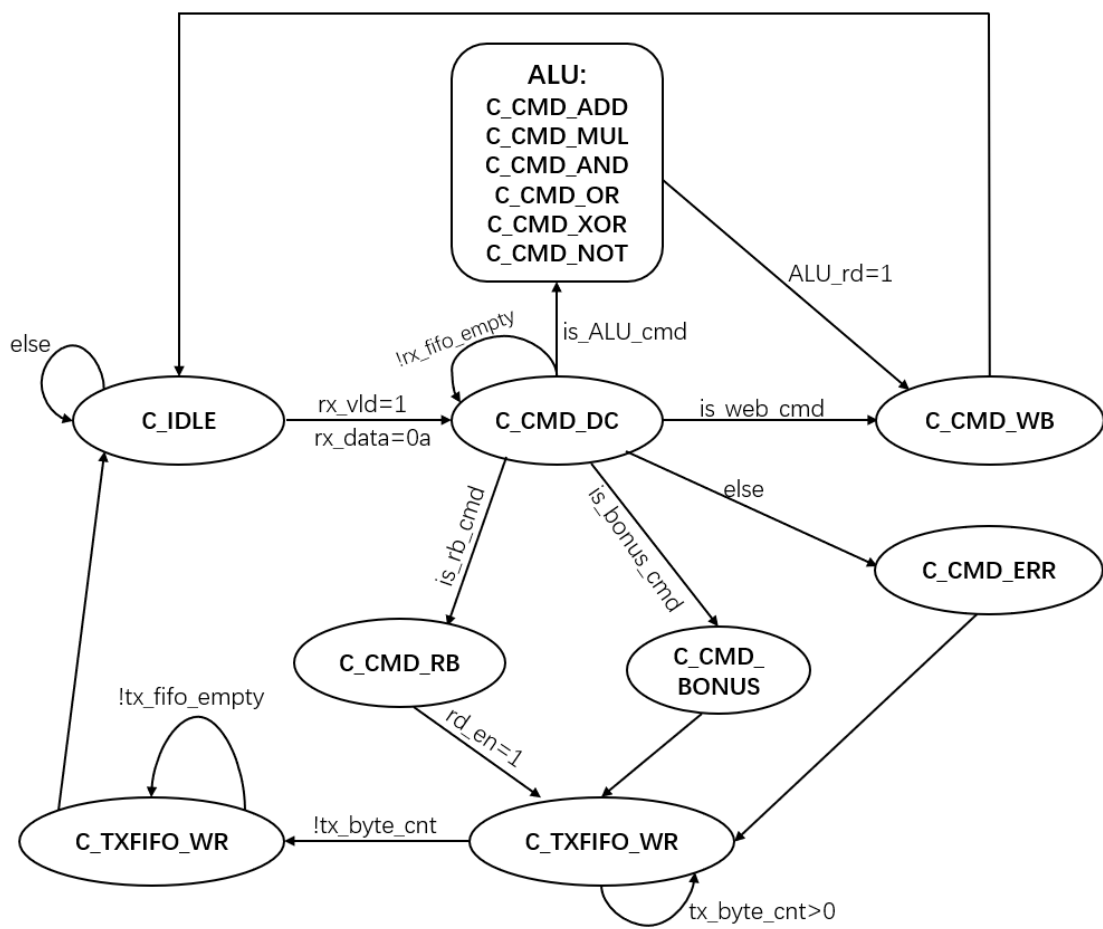
1. 在 FPGA0L 平台上，利用串口终端等外设，实现简单的 Shell 功能，例如：在串口协议基础上，实现一个读写命令解析功能，功能电路接收以 ASCII 码格式发来的命令，并根据命令类型做出合适的响应。
2. 在 Logisim 中或者在 FPGA 开发板上实现逻辑电路，通过 LED 点阵实现汉字的循环显示。要求至少循环显示十个汉字，汉字内容及机内码的形式保存在 ROM 中，控制电路顺序读取数据，完成机内码到区位码的转换，通过查询字库，获取 16*16 的像素数据，最终显示在 LED 点阵上。

【实验练习】

一、在 FPGA0L 平台上实现 Shell 功能

1. 建立状态转换图

本程序会在以下状态中循环。初始状态为 C_IDLE，当读取到来自 Shell 的指令时，进入 C_CMD_DC 状态进行指令解码，根据不同指令进入对应的状态，执行完毕后回到初始状态。状态转换图如下：



2. 命令格式说明

命令功能	格式	举例	说明
写字节	wb [addr] [data]	wb 11 1a	向 11 地址写入字节 1a
读字节	rb [addr]	rb 10	从 10 地址读取一个字节，并以 ASCII 码格式显示在串口终端
加运算	add [addr] [addr] [addr]	add 12 11 10	将地址 11 和 10 的值相加，结果存储在地址 12 中
乘运算	mul [addr] [addr] [addr]	mul 12 11 10	将地址 11 和 10 的值相乘，结果存储在地址 12 中

与运算	and [addr] [addr] [addr]	and 12 11 10	将地址 11 和 10 的值相与，结果存储在地址 12 中
或运算	or [addr] [addr] [addr]	or 12 11 10	将地址 11 和 10 的值相或，结果存储在地址 12 中
非运算	not [addr] [addr]	not 12 11	将地址 11 的值取非，结果存储到地址 12 中
异或运算	xor [addr] [addr] [addr]	xor 12 11 10	将地址 11 和 10 的值相异或，结果存储在地址 12 中
彩蛋	bonus	bonus	在串口终端上显示 "MERRY CHRISTMAS"
其他			无效命令，串口终端打印"ERROR!"字样

说明：本程序对命令格式有严格要求，命令、地址、数据之间有且仅有一个空格，且只有在输入回车后，当前一行命令才会被读入。

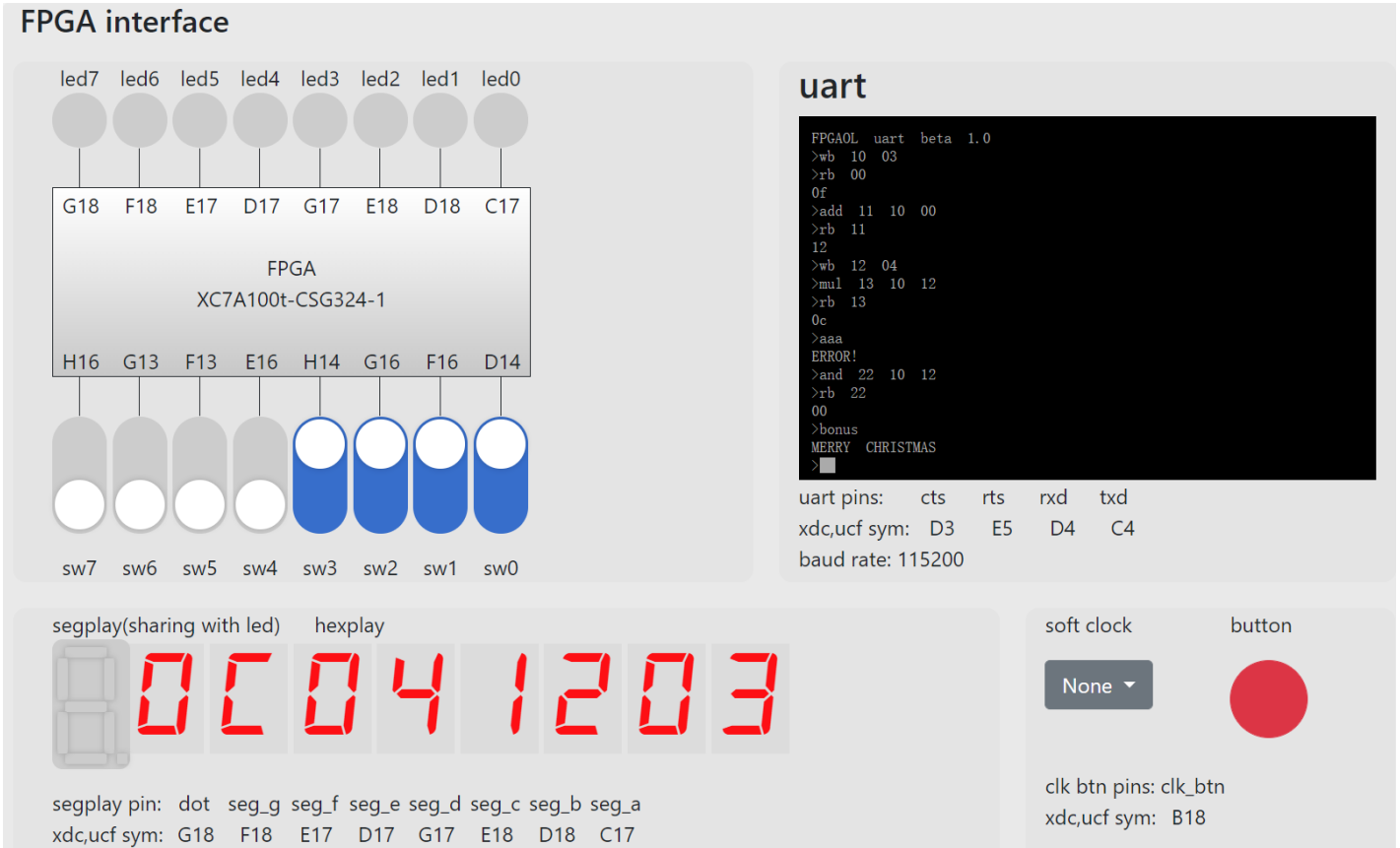
3. 地址空间分配

读地址空间		写地址空间	
00	8 个拨动开关所表示的字节数据	00	8 个 LED 所对应的字节数据
10	数码管 1~0 位所表示的字节数据	10	数码管 1~0 位所表示的字节数据
11	数码管 3~2 位所表示的字节数据	11	数码管 3~2 位所表示的字节数据
12	数码管 5~4 位所表示的字节数据	12	数码管 5~4 位所表示的字节数据
13	数码管 7~6 位所表示的字节数据	13	数码管 7~6 位所表示的字节数据

20~2f	内部存储空间	20~2f	内部存储空间
-------	--------	-------	--------

说明：20 至 2f 共计 16 个地址空间，每个对应的内容空间为 8bits

4. 实测演示截图



5. 扩展部分

- (1) 扩展了指令个数，引入了加、乘、与、或、非、异或操作
- (2) 扩展了内存个数，由原来的 5 个扩展为 21 个

6. 命令实现过程

- (1) 读入并解析来自 Shell 的命令：

通过模块 rx 将来自串口的数据进行转化，约定数据收发频率为

115200, 运用分频计数器对串口接收信号进行采样, 保存到 rx_data, 每读完 8bits 数据, 置读取完成的标志位 rx_vld 为 1.

rx_vld 作为 rx_fifo 写入的使能标志, 当一次 8bits 数据读入后, 调用 fifo 模块将该数据存储到 rx_fifo 中。

当来自串口的信号读取到换行符时, 说明来自 Shell 的一行命令输入完毕, 此时状态机进入命令解码状态。从 rx_fifo 中读取数据并且存入 rx_byte_buff, 更新命令标志变量 is_add_cmd, is_wb_cmd 等.

(2) wb 命令:

若已将 rx_fifo 中的数据读取完毕, 且 is_wb_cmd 标志变量为 1, 则进入写命令状态。根据 rx_byte_buff 更新 wr_addr 和 wr_data, 前者为要写入数据的目标地址, 后者为要写入的数据内容。根据 wr_addr, 更新对应的地址数据。

(3) rb 指令

若已将 rx_fifo 中的数据读取完毕, 且 is_rb_cmd 为 1, 则进入读命令状态。根据 rx_byte_buff 更新 rd_addr, 即要读取数据的地址。然后根据 rd_addr 到对应地址读取数据, 存储到 rd_data.

之后进入 C_TXFIFO_WR 和 C_TXFIFO_WAIT 状态, 将要显示到串口终端的数据写入 tx_fifo, 并逐个 tx_fifo 转化为 tx_data, 再将 tx_data 通过调用 tx 模块转化为输出到串口的数据。每当 tx 模块转化完其接收到的数据, 则置 tx_fifo 的读使能有效, 即再从 tx_fifo 中读取数据送入 tx 模块。

tx 模块，实现将读入的 8bits 数据按照数据收发频率 115200 转化为输出数据，输出到串口终端。

(4) add 命令

若已将 rx_fifo 中的数据读取完毕，且 is_add_cmd 标志位为 1，那么进入加命令状态。根据 rx_byte_buff 更新 ALU_addr_0, ALU_addr_1, ALU_addr_2，第一个为结果存储的目标地址，后两个为进行加法运算的操作数地址。

根据 ALU_addr_1 和 ALU_addr_2 获取对应位置的操作数，存储到 ALU_operand_1 和 ALU_operand_2，调用 8bits 加法器，求得加法结果，存入 ALU_result。

用 ALU_addr_0 和 ALU_result 分别更新 wr_addr 和 wr_data，进入写命令状态，根据 wr_addr，用 wr_data 更新目标地址的值。

(5) mul 命令

若已将 rx_fifo 中的数据读取完毕，且 is_mul_cmd 标志位为 1，那么进入乘命令状态。根据 rx_byte_buff 更新 ALU_addr_0, ALU_addr_1, ALU_addr_2，第一个为结果存储的目标地址，后两个为进行乘法运算的操作数地址。

根据 ALU_addr_1 和 ALU_addr_2 获取对应位置的操作数，存储到 ALU_operand_1 和 ALU_operand_2，调用 8bits 乘法器，求得乘法结果，存入 ALU_result。

用 ALU_addr_0 和 ALU_result 分别更新 wr_addr 和 wr_data，进入写命令状态，根据 wr_addr，用 wr_data 更新目标地址的值。

（6）逻辑运算命令

本程序实现的逻辑运算包括与、或、非、异或，所有逻辑运算执行过程类似，只是在部分细节上存在微小差异，故归到一类来说明。

若已将 rx_fifo 中的数据读取完毕，且根据更新后的逻辑运算命令标志位，判定为逻辑运算命令，则进入对应的逻辑运算命令执行状态。

根据 rx_byte_buff 更新 ALU_addr_0, ALU_addr_1, ALU_addr_2, 第一个为结果存储的目标地址，后两个为进行逻辑运算的操作数地址。

根据 ALU_addr_1 和 ALU_addr_2 获取对应位置的操作数，存储到 ALU_operand_1 和 ALU_operand_2, 进行逻辑运算，将结果存储到 ALU_result, 用 ALU_addr_0 和 ALU_result 分别更新 wr_addr 和 wr_data, 进入写命令状态，根据 wr_addr, 用 wr_data 更新目标地址的值。

注：非运算只有一个操作数，故非运算在上述执行过程中只会获得一个操作数地址，相应的，只会获得一个有效操作数。

（7）彩蛋

若已将 rx_fifo 中的数据读取完毕，且彩蛋命令标志位为 1，则进入彩蛋命令处理状态。

置 tx_byte_buff 为 "MERRY CHRISTMAS\n", 进入 C_TXFIFO_WR 和 C_TXFIFO_WAIT 状态，将要 tx_byte_buff 写入 tx_fifo, 并逐个 tx_fifo 转化为 tx_data, 再将 tx_data 通过调用 tx 模块转化为输出到串口的数据。

(8) 命令不合法，输出 REEOR 信息

若已将 rx_fifo 中的数据读取完毕，且更新后的各命令标志位均为零，即读取到的命令非法，则进入错误命令处理状态。

置 tx_byte_buff 为 "ERROR!\n"，进入 C_TXFIFO_WR 和 C_TXFIFO_WAIT 状态，将要 tx_byte_buff 写入 tx_fifo，并逐个 tx_fifo 转化为 tx_data，再将 tx_data 通过调用 tx 模块转化为输出到串口的数据。

(9) 将部分地址对应的数据显示到 LED 或数码管上

将对应的数据轮换存储到 hexplay_data，利用刷新，显示到 FPGA 上。

7. 关键代码演示说明

(1) tx 模块

此模块要实现的功能为将 8bits input 数据转化为串口数据，即一个信号会持续 868 个时钟周期，再加上开头一位起始低位，结尾一位停止高位。

建立状态机：共包含两个状态，C_IDLE 和 C_TX，前者为空闲状态，后者为输出状态，即将 tx_data 的每一位转化为 868 时钟周期的输出信号 tx。当 tx_fifo 非空时，状态机进入输出状态。当数据输出数目达到 10*868 后，说明当前 8bits 数目输出完毕，进入空闲状态。

div_cnt 记录当前 bit 的数据已经输出了多少位，tx_cnt 记录当

前输出是第几位数据。更新次态的代码为：

```
always@(*)
begin
    case(curr_state)
        C_IDLE:
            if(tx_ready==1'b1)
                next_state = C_TX;
            else
                next_state = C_IDLE;
        C_TX:
            if((div_cnt==DIV_CNT)&&(tx_cnt>=TX_CNT))
                next_state = C_IDLE;
            else
                next_state = C_TX;
    endcase
end
```

转化输出信号 tx 的代码为：

```
always@(posedge clk or posedge rst)
begin
    if(rst)
        tx <= 1'b1; //空闲
    else if(curr_state==C_IDLE)
        tx <= 1'b1; //空闲
    else if(div_cnt==10'h0) //输出状态
    begin
        case(tx_cnt)
            4'h0: tx <= 1'b0; //第一个低位数据
            4'h1: tx <= tx_reg[0];
            4'h2: tx <= tx_reg[1];
            4'h3: tx <= tx_reg[2];
            4'h4: tx <= tx_reg[3];
            4'h5: tx <= tx_reg[4];
            4'h6: tx <= tx_reg[5];
            4'h7: tx <= tx_reg[6];
            4'h8: tx <= tx_reg[7];
            4'h9: tx <= 1'b1;
        endcase
    end
end
```

(2) rx 模块

本模块实现了串口数据的输入，并将输入的数据转化为 8bits 的 rx_data。与 tx 模块实现过程类似，只不过数据传输方向相反。

状态机有两个状态，空闲状态和输入状态。由于数据采样是在 868bits 信号的中间，所以当第一个 868 时钟周期的低位开始信号输入一半时，状态由空闲装填转化为输入状态。更新次态的代码如下：

```
always@(*)
begin
    case(curr_state)
        C_IDLE:
            if(div_cnt==HDIV_CNT)//低电平进入，下一个状态为接收状态
                next_state = C_RX;
            else
                next_state = C_IDLE;
        C_RX:
            if((div_cnt==DIV_CNT)&&(rx_cnt>=RX_CNT))//8位数据接收完毕
                next_state = C_IDLE;
            else
                next_state = C_RX;
    endcase
end
```

数据采样代码如下：

```

always@(posedge clk or posedge rst)
begin
    if(rst)
        div_cnt <= 10'h0;
    else if(curr_state == C_IDLE)//在空闲状态
    begin
        if(rx==1'b1)//空闲帧
            div_cnt <= 10'h0;
        else if(div_cnt < HDIV_CNT)//不在空闲帧，即第一个低位检测到，开始输入数据
            div_cnt <= div_cnt + 10'h1;
        else//第一个开始计数的低位检测完毕
            div_cnt <= 10'h0;
    end
    else if(curr_state == C_RX)//在接收状态。计数器在0~867之间循环
    begin
        if(div_cnt >= DIV_CNT)
            div_cnt <= 10'h0;
        else
            div_cnt <= div_cnt + 10'h1;
    end
end
end

```

```

always@(posedge clk or posedge rst)
begin
    if(rst)
        rx_cnt <= 4'h0;
    else if(curr_state == C_IDLE)
        rx_cnt <= 4'h0;
    else if((div_cnt == DIV_CNT)&&(rx_cnt<4'hF))//在接收状态并且当前新读入一个数据
        rx_cnt <= rx_cnt + 1'b1;
end

assign rx_pulse = (curr_state==C_RX)&&(div_cnt==DIV_CNT);//读新数据的一个时钟周期

always@(posedge clk)
begin
    if(rx_pulse)
    begin
        case(rx_cnt)
            4'h0: rx_reg_0 <= rx;
            4'h1: rx_reg_1 <= rx;
            4'h2: rx_reg_2 <= rx;
            4'h3: rx_reg_3 <= rx;
            4'h4: rx_reg_4 <= rx;
            4'h5: rx_reg_5 <= rx;
            4'h6: rx_reg_6 <= rx;
            4'h7: rx_reg_7 <= rx;
        endcase//根据已读的数据数量，将当前数据读入
    end
end
end

```

(3) rx_fifo

fifo 这个 ip 核实现了先入先出的数据存储和读取，rx_fifo 实现了将原本 868 时钟周期才读入 1bit 的输入数据暂存，然后在获得完整的来自 Shell 的一行指令后，再依次输出，实现了对时钟不同步的输入信号的同步化处理。

它的调用代码为：

```
fifo_32x8bit_0 rx_fifo(  
  .clk          (clk),  
  .rst          (rst),  
  .din          (rx_data),  
  .wr_en        (rx_vld),  
  .rd_en        (rx_fifo_en),  
  .dout         (rx_fifo_data),  
  .full         (),  
  .empty        (rx_fifo_empty)  
);
```

向 rx_fifo 写入数据时：写使能为 rx_vld，写使能为 1 当且仅当 rx 模块刚读取完 8bit 数据。

从 tx_fifo 读取数据：读使能为 rx_fifo_en，在命令解码状态会置这个标志位为 1

(4) tx_fifo

tx_fifo 实现了将要输出的数据，根据约定好的串口数据收发频率进行转化输出，将每一位要输出的数据转化为 868 时钟周期的信号。

调用代码为：

```

fifo_32x8bit_0    tx_fifo(
.clk              (clk),
.rst              (rst),
.din              (tx_fifo_din),
.wr_en            (tx_fifo_wr_en),
.rd_en            (tx_rd), //读使能有效,
.dout             (tx_data),
.full             (tx_fifo_full),
.empty            (tx_fifo_empty)
);

```

写入数据：将要输入的数据赋值给 tx_fifo_din，写使能为 tx_fifo_wr_en，在 C_TXFIFO_WR 状态会置此标志位为 1。

读取数据：读使能为 tx_rd，这个标志位为 1 当且仅当 tx_fifo 非空且 tx 模块输出完上一个 8bits 数据。

(5) 命令解码状态 C_CMD_DC

将来自 rx_fifo 的数据存储到 rx_byte_buff，根据 rx_byte_buff 更新命令标志位，例如判断是否为乘法命令的标志位：（其他标志位基本类似）

```

assign is_mul_cmd = (curr_state==C_CMD_DC)
&&(rx_byte_buff_0=="m")&&(rx_byte_buff_1=="u")&&(rx_byte_buff_2=="1")
&&(rx_byte_buff_3==" ")
&&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f")))
&&(((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))||((rx_byte_buff_5>="a")&&(rx_byte_buff_5<="f")))
&&(rx_byte_buff_6==" ")
&&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f")))
&&(((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))||((rx_byte_buff_8>="a")&&(rx_byte_buff_8<="f")))
&&(rx_byte_buff_9==" ")
&&(((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))||((rx_byte_buff_10>="a")&&(rx_byte_buff_10<="f")))
&&(((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))||((rx_byte_buff_11>="a")&&(rx_byte_buff_11<="f"))));

```

之后会根据各种标志位进行状态转移。

(6) 加法命令

根据处理好的两个操作数，调用加法模块，计算结果。8bits 加

法器代码为：

```
module add_8(  
    output [7:0] s,  
    output      cout,  
    input  [7:0] a,b,  
    input      cin  
);  
wire [6:0] carry;  
  
add    add0(s[0], carry[0], a[0], b[0], cin);  
add    add1(s[1], carry[1], a[1], b[1], carry[0]);  
add    add2(s[2], carry[2], a[2], b[2], carry[1]);  
add    add3(s[3], carry[3], a[3], b[3], carry[2]);  
add    add4(s[4], carry[4], a[4], b[4], carry[3]);  
add    add5(s[5], carry[5], a[5], b[5], carry[4]);  
add    add6(s[6], carry[6], a[6], b[6], carry[5]);  
add    add7(s[7], cout,    a[7], b[7], carry[6]);  
endmodule
```

```
module add(  
    output s, cout,  
    input a, b, cin  
);  
  
assign s = a ^ b ^ cin;  
assign cout = (a & b) | (a & cin) | (b & cin);  
endmodule
```

（7）乘法命令

8bits 乘法器的实现是运用了列竖式的思想,进行二进制数相乘,判断乘数的每个 bit 是否是 1,若是,则将被乘数移位后加入结果.

代码如下：

```

module mul(
input  [7:0]  a,b,
output [7:0]  out
);

wire  [7:0]  mul_1;
wire  [7:0]  mul_2;
wire  [7:0]  mul_3;
wire  [7:0]  mul_4;
wire  [7:0]  mul_5;
wire  [7:0]  mul_6;
wire  [7:0]  mul_7;
wire  [7:0]  mul_8;

assign mul_1 = (b[0]==1'b1)?(a):(8'h0);
assign mul_2 = (b[1]==1'b1)?({a[6:0],1'b0}):(8'h0);
assign mul_3 = (b[2]==1'b1)?({a[5:0],2'b0}):(8'h0);
assign mul_4 = (b[3]==1'b1)?({a[4:0],3'b0}):(8'h0);
assign mul_5 = (b[4]==1'b1)?({a[3:0],4'b0}):(8'h0);
assign mul_6 = (b[5]==1'b1)?({a[2:0],5'b0}):(8'h0);
assign mul_7 = (b[6]==1'b1)?({a[1:0],6'b0}):(8'h0);
assign mul_8 = (b[7]==1'b1)?({a[0],7'b0}):(8'h0);

wire  [7:0]  tmp_1;
wire  [7:0]  tmp_2;
wire  [7:0]  tmp_3;
wire  [7:0]  tmp_4;
wire  [7:0]  tmp_5;
wire  [7:0]  tmp_6;

add_8  add1(.s(tmp_1),.cout(),.a(mul_1),.b(mul_2),.cin(1'b0));
add_8  add2(.s(tmp_2),.cout(),.a(mul_3),.b(mul_4),.cin(1'b0));
add_8  add3(.s(tmp_3),.cout(),.a(mul_5),.b(mul_6),.cin(1'b0));
add_8  add4(.s(tmp_4),.cout(),.a(mul_7),.b(mul_8),.cin(1'b0));
add_8  add5(.s(tmp_5),.cout(),.a(tmp_1),.b(tmp_2),.cin(1'b0));
add_8  add6(.s(tmp_6),.cout(),.a(tmp_3),.b(tmp_4),.cin(1'b0));
add_8  add7(.s(out),.cout(),.a(tmp_5),.b(tmp_6),.cin(1'b0));

endmodule

```


(8) ALU

ALU 包含加、乘、与、或、非、异或。

1. 根据 rx_byte_buff 获得操作数地址和结果地址的代码如下：

(以加法指令为例，其他指令如此类似，在具体细节上存在细微差别)

```
if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
    ALU_addr_0[7:4] <= rx_byte_buff_4[3:0];
else
    ALU_addr_0[7:4] <= rx_byte_buff_4[2:0] + 4'h9;
if((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))
    ALU_addr_0[3:0] <= rx_byte_buff_5[3:0];
else
    ALU_addr_0[3:0] <= rx_byte_buff_5[2:0] + 4'h9;
if((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))
    ALU_addr_1[7:4] <= rx_byte_buff_7[3:0];
else
    ALU_addr_1[7:4] <= rx_byte_buff_7[2:0] + 4'h9;
if((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))
    ALU_addr_1[3:0] <= rx_byte_buff_8[3:0];
else
    ALU_addr_1[3:0] <= rx_byte_buff_8[2:0] + 4'h9;
if((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))
    ALU_addr_2[7:4] <= rx_byte_buff_10[3:0];
else
    ALU_addr_2[7:4] <= rx_byte_buff_10[2:0] + 4'h9;
if((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))
    ALU_addr_2[3:0] <= rx_byte_buff_11[3:0];
else
    ALU_addr_2[3:0] <= rx_byte_buff_11[2:0] + 4'h9;
```

2. 根据操作数，求结果的代码如下：

```

//ALU_operand
always @(*) begin
    if(ALU_rd_2) begin
        case(ALU_addr_1)
            8'h0:   ALU_operand_1 = sw;
            8'h10:  ALU_operand_1 = hexplay_buff[7:0];
            8'h11:  ALU_operand_1 = hexplay_buff[15:8];
            8'h12:  ALU_operand_1 = hexplay_buff[23:16];
            8'h13:  ALU_operand_1 = hexplay_buff[31:24];
            8'h20:  ALU_operand_1 = store_buff[7:0];
            8'h21:  ALU_operand_1 = store_buff[15:8];
            8'h22:  ALU_operand_1 = store_buff[23:16];
            8'h23:  ALU_operand_1 = store_buff[31:24];
            8'h24:  ALU_operand_1 = store_buff[39:32];
            8'h25:  ALU_operand_1 = store_buff[47:40];
            8'h26:  ALU_operand_1 = store_buff[55:48];
            8'h27:  ALU_operand_1 = store_buff[63:56];
            8'h28:  ALU_operand_1 = store_buff[71:64];
            8'h29:  ALU_operand_1 = store_buff[79:72];
            8'h2a:  ALU_operand_1 = store_buff[87:80];
            8'h2b:  ALU_operand_1 = store_buff[95:88];
            8'h2c:  ALU_operand_1 = store_buff[103:96];
            8'h2d:  ALU_operand_1 = store_buff[111:104];
            8'h2e:  ALU_operand_1 = store_buff[119:112];
            8'h2f:  ALU_operand_1 = store_buff[127:120];
            default:ALU_operand_1 = 8'h0;
        endcase
    end
end

```

```

        case(ALU_addr_2)
            8'h0:   ALU_operand_2 = sw;
            8'h10:  ALU_operand_2 = hexplay_buff[7:0];
            8'h11:  ALU_operand_2 = hexplay_buff[15:8];
            8'h12:  ALU_operand_2 = hexplay_buff[23:16];
            8'h13:  ALU_operand_2 = hexplay_buff[31:24];
            8'h20:  ALU_operand_2 = store_buff[7:0];
            8'h21:  ALU_operand_2 = store_buff[15:8];
            8'h22:  ALU_operand_2 = store_buff[23:16];
            8'h23:  ALU_operand_2 = store_buff[31:24];
            8'h24:  ALU_operand_2 = store_buff[39:32];
            8'h25:  ALU_operand_2 = store_buff[47:40];
            8'h26:  ALU_operand_2 = store_buff[55:48];
            8'h27:  ALU_operand_2 = store_buff[63:56];
            8'h28:  ALU_operand_2 = store_buff[71:64];
            8'h29:  ALU_operand_2 = store_buff[79:72];
            8'h2a:  ALU_operand_2 = store_buff[87:80];
            8'h2b:  ALU_operand_2 = store_buff[95:88];
            8'h2c:  ALU_operand_2 = store_buff[103:96];
            8'h2d:  ALU_operand_2 = store_buff[111:104];
            8'h2e:  ALU_operand_2 = store_buff[119:112];
            8'h2f:  ALU_operand_2 = store_buff[127:120];
            default:ALU_operand_2 = 8'h0;
        endcase
    end
end

```

```

wire [7:0] ALU_ADD_result; //加法器
add_8 add(.s(ALU_ADD_result),.cout(),.a(ALU_operand_1),.b(ALU_operand_2),.cin(1'b0));
wire [7:0] ALU_MUL_result; //乘法器
mul mul(.a(ALU_operand_1),.b(ALU_operand_2),.out(ALU_MUL_result));

always @(*) begin
    if(ALU_rd) begin
        if(curr_state==C_CMD_ADD)
            ALU_result = ALU_ADD_result;
        else if(curr_state==C_CMD_AND)
            ALU_result = ALU_operand_1 & ALU_operand_2;
        else if(curr_state==C_CMD_OR)
            ALU_result = ALU_operand_1 | ALU_operand_2;
        else if(curr_state==C_CMD_NOT)
            ALU_result = ~ ALU_operand_1;
        else if(curr_state==C_CMD_XOR)
            ALU_result = ALU_operand_1 ^ ALU_operand_2;
        else if(curr_state==C_CMD_MUL)
            ALU_result = ALU_MUL_result;
    end
end

```

3. 比较重要的一点是，ALU 运算过程是有先后过程，故设计为三个时钟周期。第一个时钟周期完成根据 rx_byte_buff 更新 ALU_addr，第二个时钟周期内根据 ALU_addr 取出相应的操作数，第三个时钟周期，将操作数进行运算，获得 ALU_result。

两个时钟周期依靠标志位 ALU_rd_2 和 ALU_rd 实现，当且仅当已经进入 ALU 运算的状态后，置 ALU_rd_2 为 1，在检测到 ALU_rd_2 为 1 的那个时钟周期，置 ALU_rd 为 1，在检测到 ALU_rd 为 1 的那个时钟周期，置次态为写操作状态：

```

C_CMD_ADD:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_ADD;

```

也即在进入加法运算状态的第一个时钟周期内，因 ALU_rd 此时为 0，故次态仍为加法状态。进入第二个时钟周期，ALU_rd_2 已被置为 1，ALU_rd 仍为 0，次态仍是加法状态。进入第三个时钟周期，ALU_rd 已被置为 1，此时次态是写操作状态，所以三个时钟周期后，加法运

算结束。

(9) 写命令

写命令的实行过程包括：更新要写入的目标地址 `wr_addr`，获得要写入的数据；根据 `wr_addr` 将 `wr_data` 写入相应的地址。

需要特殊处理的是，在扩展了 ALU 指令后，进入到写操作状态有两种情况。一是来自 Shell 的命令就是写命令，二是来自 Shell 的命令是运算指令，运算过程执行完后，需要将运算结果存储到目标地址时，也会进入写操作状态。

为区分这两个不同情况，设置标志位 `ALU_wd`，`ALU_wd` 为 1，表示上一个状态是运算状态，此时要写入的数据存储在 `ALU_result` 中，要写入的目标地址存储在 `ALU_addr_0` 中。`ALU_wd=0`，表示当前命令就是单纯的写命令，要写入的目标地址和数据均来自 Shell，现存储在 `rx_byte_buff` 中。

所以在写操作状态中，更新 `wr_addr` 和 `wr_data` 的操作如下：

```

else if(curr_state == C_CMD_WB) begin
    wr_en    <= 1'b1;
    if(ALU_wd==1'b1) begin//上一个状态是进行运算
        wr_addr <= ALU_addr_0;
        wr_data <= ALU_result;
    end
    else begin//直接写的操作
        if((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))
            wr_addr[7:4] <= rx_byte_buff_3[3:0];
        else
            wr_addr[7:4] <= rx_byte_buff_3[2:0] + 4'h9;
        if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
            wr_addr[3:0] <= rx_byte_buff_4[3:0];
        else
            wr_addr[3:0] <= rx_byte_buff_4[2:0] + 4'h9;
        if((rx_byte_buff_6>="0")&&(rx_byte_buff_6<="9"))
            wr_data[7:4] <= rx_byte_buff_6[3:0];
        else
            wr_data[7:4] <= rx_byte_buff_6[2:0] + 4'h9;
        if((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))
            wr_data[3:0] <= rx_byte_buff_7[3:0];
        else
            wr_data[3:0] <= rx_byte_buff_7[2:0] + 4'h9;
    end
end
end

```

(10) 为向 Shell 输出做准备

要输出的数据个数存储到 tx_byte_cnt 中，要输出的数据存储到 tx_byte_buff 中。这个过程在进入 C_TXFIFO_WR 之前完成。

代码如下：

```

//tx_byte_cnt, 要向shell输出的数据块数
always@(posedge clk or posedge rst)
begin
    if(rst)
        tx_byte_cnt <= 8'h0;
    else if(curr_state==C_IDLE)
        tx_byte_cnt <= 8'h0;
    else if(curr_state==C_CMD_RB)
        tx_byte_cnt <= 8'h2;
    else if(curr_state==C_CMD_ERR)
        tx_byte_cnt <= 8'h6;
    else if(curr_state==C_CMD_BONUS)
        tx_byte_cnt <= 8'hf;
    else if(curr_state==C_TXFIFO_WR)
    begin
        if(tx_byte_cnt!=8'h0)
            tx_byte_cnt <= tx_byte_cnt - 8'h1;
    end
end
end

```

```

//tx_byte_buff, 即将向shell输出的数据
//在C_TXFIFO_WR和C_TCFIFO_WAIT状态下, 进行转化输出
always@(posedge clk or posedge rst)
begin
    if(rst) begin
        tx_byte_buff_0 <= 8'h0;
        tx_byte_buff_1 <= 8'h0;
        tx_byte_buff_2 <= 8'h0;
        tx_byte_buff_3 <= 8'h0;
        tx_byte_buff_4 <= 8'h0;
        tx_byte_buff_5 <= 8'h0;
        tx_byte_buff_6 <= 8'h0;
        tx_byte_buff_7 <= 8'h0;
    end
    else if(curr_state==C_IDLE) begin
        tx_byte_buff_0 <= 8'h0;
        tx_byte_buff_1 <= 8'h0;
        tx_byte_buff_2 <= 8'h0;
        tx_byte_buff_3 <= 8'h0;
        tx_byte_buff_4 <= 8'h0;
        tx_byte_buff_5 <= 8'h0;
        tx_byte_buff_6 <= 8'h0;
        tx_byte_buff_7 <= 8'h0;
    end
end
end

```

```

else if(curr_state==C_CMD_RB) begin
    tx_byte_buff_0 <= "\n";
    if(rd_data[7:4]<=4'h9)//0~9
        tx_byte_buff_2 <= {4'h3,rd_data[7:4]};
    else
        tx_byte_buff_2 <= rd_data[7:4] - 4'ha + "a";
    if(rd_data[3:0]<=4'h9)//0~9
        tx_byte_buff_1 <= {4'h3,rd_data[3:0]};
    else
        tx_byte_buff_1 <= rd_data[3:0] - 4'ha + "a";
end
else if(curr_state==C_CMD_ERR) begin
    tx_byte_buff_6 <= "E";
    tx_byte_buff_5 <= "R";
    tx_byte_buff_4 <= "R";
    tx_byte_buff_3 <= "O";
    tx_byte_buff_2 <= "R";
    tx_byte_buff_1 <= "!";
    tx_byte_buff_0 <= "\n";
end
else if(curr_state==C_CMD_BONUS) begin
    tx_byte_buff_15 <= "M";
    tx_byte_buff_14 <= "E";
    tx_byte_buff_13 <= "R";
    tx_byte_buff_12 <= "R";
    tx_byte_buff_11 <= "Y";
    tx_byte_buff_10 <= " ";
    tx_byte_buff_9 <= "C";
    tx_byte_buff_8 <= "H";
    tx_byte_buff_7 <= "R";
    tx_byte_buff_6 <= "I";
    tx_byte_buff_5 <= "S";
    tx_byte_buff_4 <= "T";
    tx_byte_buff_3 <= "M";
    tx_byte_buff_2 <= "A";
    tx_byte_buff_1 <= "S";
    tx_byte_buff_0 <= "\n";
end
end
end

```

进入 C_TXFIFO_WR 状态后，tx_byte_cnt 每个时钟周期减一，实现将 tx_byte_buff 逐个写入 tx_fifo 中。每写完一个就会开始输出，将输出送入 tx 模块转化为串口支持的数据。

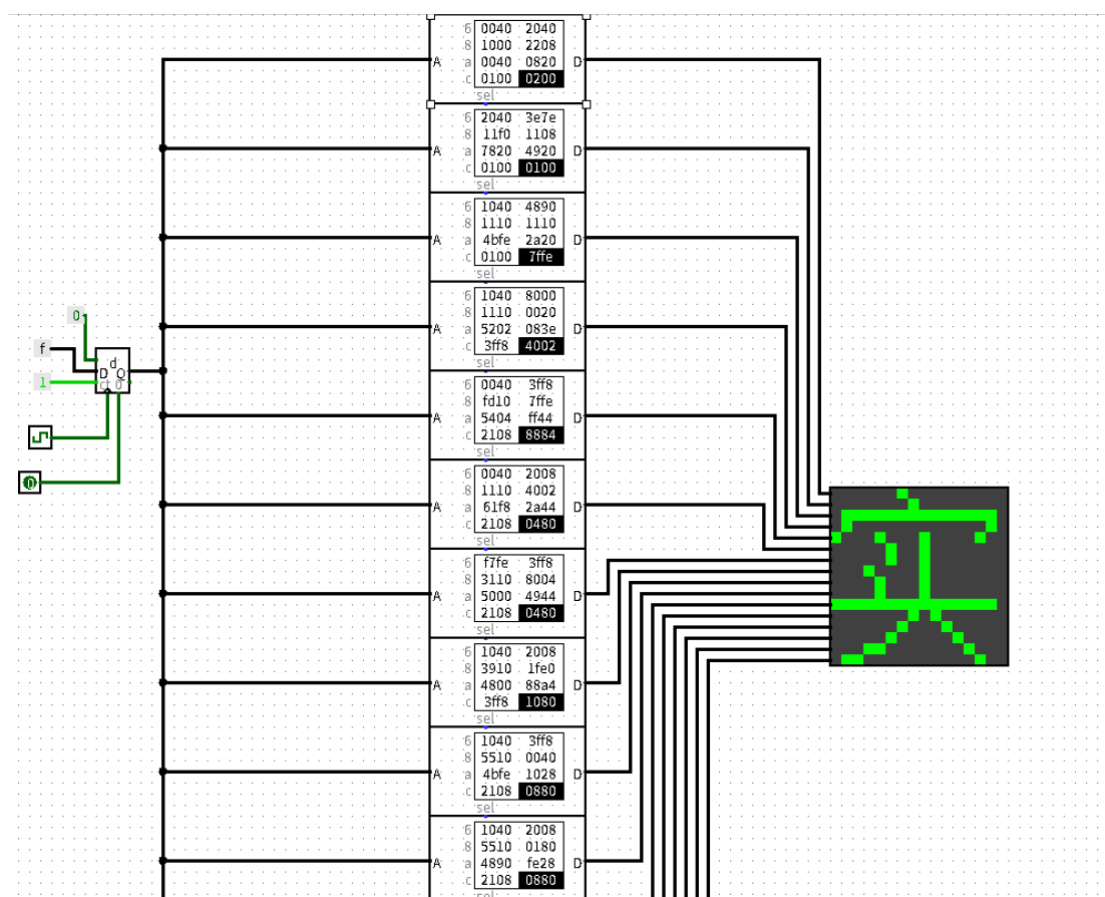
最后在 tx_byte_cnt=0 但 tx_fifo 中的数据仍没有全部输出时，状态进入 C_TXFIFO_WAIT 状态，将 tx_fifo 剩下的数据全部输出。

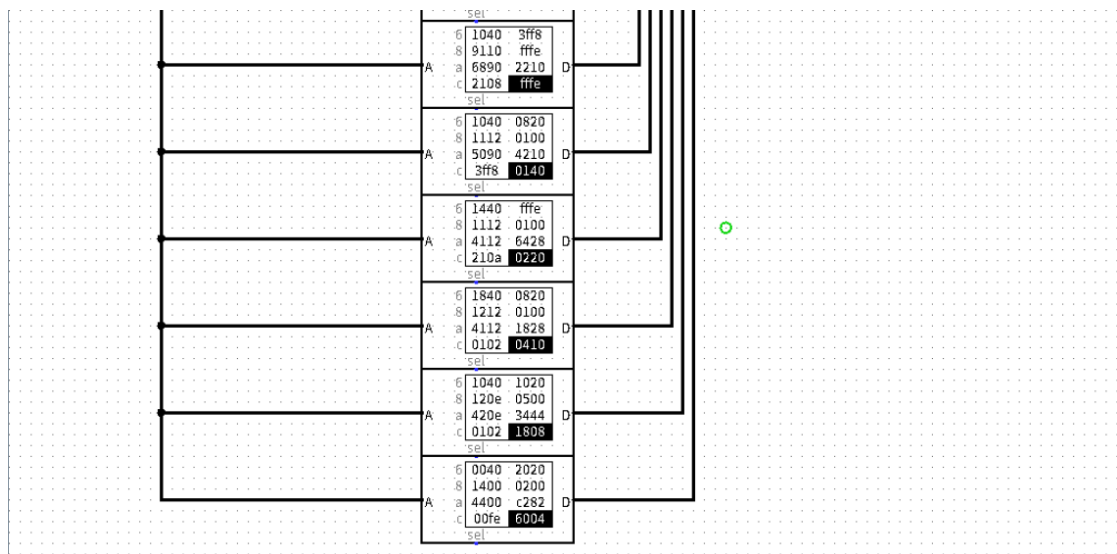
二、在 Logisim 中通过 LED 点阵实现汉字的循环显示

要在 16*16 的 LED 点阵中显示一个汉字，需要 16 根 16bits 线作为输入。而要实现循环输出 16 个汉字，则需要每个 16bits 的线可以循环输出 16 个 16bits 数据。故使用 ROM 存储 16*16bits 的数据，共需要 16 个 ROM。而汉字循环过程用计数器实现。

在汉字转换成 16*16 点阵的在线转换器上将要显示的汉字转化成 16 进制数，然后每一行存储到不同的 ROM 中。

引入计数器，进行 0~15 循环计数，实现每个 ROM 的 16 个数据的循环输出。如下图：





向计数器引入时钟信号，在合适的频率下，可实现速度适中的汉字循环显示效果。

【总结与思考】

1. 本次实验，学习到了如何通过串口实现与 FPGA 实现数据交互，如何在没有同步时钟信号的情况下，实现数据的采样。
2. 学习了 FIFO IP 核的使用，它存储的数据的特点是先入先出，实现了无法同步处理的数据的暂存。
3. 学习了在较大规模的 Verilog 程序中，状态机的使用，本次实验的核心实现逻辑在于状态机的设计与实现，根据不同条件实现状态的转移，在不同状态内完成相应的操作，实现了将一个大的任务拆分成容易实现的子任务。
4. 在本次实验，对 Verilog 语言的使用进行了大量的联系，对于 Verilog 与计算机高级语言的区别有了更深入的认识，其中最重要的一点就是它的不同 Always 块是并行执行的，代码书写的先后顺序并

不影响执行的并行性。如果需要引入先后顺序，则需要加入标志变量或者拆分为不同的状态。

5. 本次实验在任务量和代码量都远超过去每一次实验，难度也比较大，作为综合实验也是具有一定的挑战性。

6. 建议在今后的课程中，加入对 IP 核使用的介绍，并且讲解一下综合实验。

附录：代码

lab10_v.v

```
//程序的核心模块
//对时钟周期的认识，每个状态占用周期数需要仔细规划，always是选择在时钟上升沿还是*

module lab10_v(
    input                clk,
    input                rst,
    input                rx,//输入信号
    output               tx,//输出信号，以上两个信号是与FPGA的shell的串口数据,需要特殊处理
    output reg [7:0]     led,//输出到LED
    input [7:0]          sw,//通过开关读入
    output wire [2:0]     hexplay_an,
    output reg [3:0]      hexplay_data
);

parameter C_IDLE        = 4'b0000;
parameter C_CMD_DC      = 4'b0010;
parameter C_CMD_WB      = 4'b0011;
parameter C_CMD_RB      = 4'b0100;
parameter C_CMD_ERR     = 4'b0111;
parameter C_TXFIFO_WR   = 4'b0101;
parameter C_TXFIFO_WAIT = 4'b0110;
parameter C_CMD_ADD     = 4'b1000;
parameter C_CMD_AND     = 4'b1001;
parameter C_CMD_OR      = 4'b1010;
parameter C_CMD_NOT     = 4'b1011;
parameter C_CMD_XOR     = 4'b1100;
parameter C_CMD_MUL     = 4'b1101;
parameter C_CMD_BONUS   = 4'b1110;

reg [7:0] ALU_operand_1;
reg [7:0] ALU_operand_2;
reg [7:0] ALU_result;

reg [7:0] ALU_addr_0;
reg [7:0] ALU_addr_1;
reg [7:0] ALU_addr_2;

reg ALU_rd;
reg ALU_wd;

wire [7:0] tx_data;
wire [7:0] rx_data;

reg [3:0] curr_state;
reg [3:0] next_state;

wire is_wb_cmd;
wire is_rb_cmd;
wire is_add_cmd;
wire is_and_cmd;
wire is_or_cmd;
```

```

wire          is_not_cmd;
wire          is_xor_cmd;
wire          is_mul_cmd;
wire          is_bonus_cmd;

reg           [7:0] tx_byte_cnt;
reg           [3:0] rx_byte_cnt;

reg           [7:0] rx_byte_buff_0;
reg           [7:0] rx_byte_buff_1;
reg           [7:0] rx_byte_buff_2;
reg           [7:0] rx_byte_buff_3;
reg           [7:0] rx_byte_buff_4;
reg           [7:0] rx_byte_buff_5;
reg           [7:0] rx_byte_buff_6;
reg           [7:0] rx_byte_buff_7;
reg           [7:0] rx_byte_buff_8;
reg           [7:0] rx_byte_buff_9;
reg           [7:0] rx_byte_buff_10;
reg           [7:0] rx_byte_buff_11;

reg           [7:0] tx_byte_buff_0;
reg           [7:0] tx_byte_buff_1;
reg           [7:0] tx_byte_buff_2;
reg           [7:0] tx_byte_buff_3;
reg           [7:0] tx_byte_buff_4;
reg           [7:0] tx_byte_buff_5;
reg           [7:0] tx_byte_buff_6;
reg           [7:0] tx_byte_buff_7;
reg           [7:0] tx_byte_buff_8;
reg           [7:0] tx_byte_buff_9;
reg           [7:0] tx_byte_buff_10;
reg           [7:0] tx_byte_buff_11;
reg           [7:0] tx_byte_buff_12;
reg           [7:0] tx_byte_buff_13;
reg           [7:0] tx_byte_buff_14;
reg           [7:0] tx_byte_buff_15;

reg           [31:0] hexplay_buff; //存储数据的作用
reg           [127:0] store_buff;

reg           rx_fifo_en;
wire          [7:0] rx_fifo_data;
wire          rx_fifo_empty;

reg           wr_en;
reg           [7:0] wr_addr;
reg           [7:0] wr_data;

reg           rd_en;
reg           [7:0] rd_addr;
reg           [7:0] rd_data;

reg           [7:0] tx_fifo_din;
reg           tx_fifo_wr_en;
wire          tx_fifo_full;
wire          tx_fifo_empty;

```

```

rx            rx_inst(
.clk          (clk),
.rst          (rst),
.rx           (rx),
.rx_vld       (rx_vld),
.rx_data      (rx_data)
);
//串行输入信号为rx，取完所有数据的标志为rx_vld，获得的数据存储在rx_data

tx            tx_inst(
.clk          (clk),
.rst          (rst),
.tx           (tx ),
.tx_ready     (~tx_fifo_empty),
.tx_rd        (tx_rd), //从tx_fifo再读一次数据的标志位
.tx_data      (tx_data)
);
//要输出的数据为tx_data，若tx_fifo不为空，则开始串行输出数据，输出信号为tx，输出完的标志位为
tx_rd

fifo_32x8bit_0 rx_fifo(
.clk          (clk),
.rst          (rst),
.din          (rx_data),
.wr_en        (rx_vld),
.rd_en        (rx_fifo_en),
.dout         (rx_fifo_data),
.full         (),
.empty        (rx_fifo_empty)
);

fifo_32x8bit_0 tx_fifo(
.clk          (clk),
.rst          (rst),
.din          (tx_fifo_din),
.wr_en        (tx_fifo_wr_en),
.rd_en        (tx_rd), //读使能有效，当且仅当tx_fifo非空，且需要从tx_fifo读出下一个
8bits
.dout         (tx_data),
.full         (tx_fifo_full),
.empty        (tx_fifo_empty)
);

always@(*)
begin
    case(curr_state)
        C_IDLE:
            if((rx_vld==1'b1)&&(rx_data==8'h0a)) //读完一行shell指令，读到了换行符
                next_state = C_CMD_DC;
            else
                next_state = C_IDLE;
        C_CMD_DC:
            if(rx_fifo_empty)
            begin
                if(is_wb_cmd)
                    next_state = C_CMD_WB;
                else if(is_rb_cmd)
                    next_state = C_CMD_RB;
            end
    endcase
end

```

```

        else if(is_add_cmd)
            next_state = C_CMD_ADD;
        else if(is_and_cmd)
            next_state = C_CMD_AND;
        else if(is_or_cmd)
            next_state = C_CMD_OR;
        else if(is_not_cmd)
            next_state = C_CMD_NOT;
        else if(is_xor_cmd)
            next_state = C_CMD_XOR;
        else if(is_mul_cmd)
            next_state = C_CMD_MUL;
        else if(is_bonus_cmd)
            next_state = C_CMD_BONUS;
        else
            next_state = C_CMD_ERR;
    end
else
    next_state = C_CMD_DC;
C_CMD_MUL:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_MUL;
C_CMD_XOR:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_XOR;
C_CMD_NOT:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_NOT;
C_CMD_OR:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_OR;
C_CMD_WB:
    next_state = C_IDLE;
C_CMD_RB://需要两个时钟周期完成
    if(rd_en==1'b1)
        next_state = C_TXFIFO_WR;
    else
        next_state = C_CMD_RB;
C_CMD_ADD:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_ADD;
C_CMD_AND:
    if(ALU_rd==1'b1)
        next_state = C_CMD_WB;
    else
        next_state = C_CMD_AND;
C_CMD_BONUS:
    next_state = C_TXFIFO_WR;

```

```

        C_CMD_ERR:
            next_state = C_TXFIFO_WR;
        C_TXFIFO_WR:
            if(tx_byte_cnt==4'h0)
                next_state = C_TXFIFO_WAIT;
            else
                next_state = C_TXFIFO_WR;
        C_TXFIFO_WAIT:
            if(tx_fifo_empty)
                next_state = C_IDLE;
            else
                next_state = C_TXFIFO_WAIT;
        default:
            next_state = C_IDLE;
    endcase
end

always@(posedge clk or posedge rst)//状态转移
begin
    if(rst)
        curr_state <= C_IDLE;
    else
        curr_state <= next_state;
end

//tx_byte_cnt, 要向shell输出的数据块数
always@(posedge clk or posedge rst)
begin
    if(rst)
        tx_byte_cnt <= 8'h0;
    else if(curr_state==C_IDLE)
        tx_byte_cnt <= 8'h0;
    else if(curr_state==C_CMD_RB)
        tx_byte_cnt <= 8'h2;
    else if(curr_state==C_CMD_ERR)
        tx_byte_cnt <= 8'h6;
    else if(curr_state==C_CMD_BONUS)
        tx_byte_cnt <= 8'hf;
    else if(curr_state==C_TXFIFO_WR)
        begin
            if(tx_byte_cnt!=8'h0)
                tx_byte_cnt <= tx_byte_cnt - 8'h1;
        end
    end
end

//tx_byte_buff, 即将向shell输出的数据
//在C_TXFIFO_WR和C_TCFIFO_WAIT状态下, 进行转化输出
always@(posedge clk or posedge rst)
begin
    if(rst) begin
        tx_byte_buff_0 <= 8'h0;
        tx_byte_buff_1 <= 8'h0;
        tx_byte_buff_2 <= 8'h0;
        tx_byte_buff_3 <= 8'h0;
        tx_byte_buff_4 <= 8'h0;
        tx_byte_buff_5 <= 8'h0;
        tx_byte_buff_6 <= 8'h0;
        tx_byte_buff_7 <= 8'h0;
    end
end

```

```

end
else if(curr_state==C_IDLE) begin
    tx_byte_buff_0 <= 8'h0;
    tx_byte_buff_1 <= 8'h0;
    tx_byte_buff_2 <= 8'h0;
    tx_byte_buff_3 <= 8'h0;
    tx_byte_buff_4 <= 8'h0;
    tx_byte_buff_5 <= 8'h0;
    tx_byte_buff_6 <= 8'h0;
    tx_byte_buff_7 <= 8'h0;
end
else if(curr_state==C_CMD_RB) begin
    tx_byte_buff_0 <= "\n";
    if(rd_data[7:4]<=4'h9)//0~9
        tx_byte_buff_2 <= {4'h3,rd_data[7:4]};
    else
        tx_byte_buff_2 <= rd_data[7:4] - 4'ha + "a";
    if(rd_data[3:0]<=4'h9)//0~9
        tx_byte_buff_1 <= {4'h3,rd_data[3:0]};
    else
        tx_byte_buff_1 <= rd_data[3:0] - 4'ha + "a";
end
else if(curr_state==C_CMD_ERR) begin
    tx_byte_buff_6 <= "E";
    tx_byte_buff_5 <= "R";
    tx_byte_buff_4 <= "R";
    tx_byte_buff_3 <= "O";
    tx_byte_buff_2 <= "R";
    tx_byte_buff_1 <= "!";
    tx_byte_buff_0 <= "\n";
end
else if(curr_state==C_CMD_BONUS) begin
    tx_byte_buff_15 <= "M";
    tx_byte_buff_14 <= "E";
    tx_byte_buff_13 <= "R";
    tx_byte_buff_12 <= "R";
    tx_byte_buff_11 <= "Y";
    tx_byte_buff_10 <= " ";
    tx_byte_buff_9 <= "C";
    tx_byte_buff_8 <= "H";
    tx_byte_buff_7 <= "R";
    tx_byte_buff_6 <= "I";
    tx_byte_buff_5 <= "S";
    tx_byte_buff_4 <= "T";
    tx_byte_buff_3 <= "M";
    tx_byte_buff_2 <= "A";
    tx_byte_buff_1 <= "S";
    tx_byte_buff_0 <= "\n";
end
end
end

//开始执行C_CMD_DC中的操作
//DECODE 来自shell的输入, 已经由模块rx和rx_fifo存在rx_fifo中
//在C_CMD_DC状态, 置rx_fifo_en为1, 从rx_fifo获得rx_fifo_data
//存储到rx_byte_buff, 准备在后续状态中使用
//更新 is_wb_cmd 和 is_rb_cmd

//rx_fifo_en

```



```

always@(posedge clk or posedge rst)
begin
    if(rst)
        rx_fifo_en  <= 1'b0;
    else if(curr_state==C_CMD_DC)
        rx_fifo_en  <= 1'b1;
    else
        rx_fifo_en  <= 1'b0;
end

//rx_byte_cnt,记录从rx_fifo中读取的数据块
always@(posedge clk or posedge rst)
begin
    if(rst)
        rx_byte_cnt <= 4'h0;
    else if(curr_state==C_CMD_DC)
    begin
        if((rx_fifo_en)&&(rx_fifo_empty==1'b0)&&(rx_byte_cnt<4'hf))//最多读16个
8bit数据
            rx_byte_cnt <= rx_byte_cnt + 4'b1;
        end
    else
        rx_byte_cnt <= 4'h0;
    end
end

//rx_byte_buff,将rx_fifo中的数据存到rx_byte_buff中
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        rx_byte_buff_0  <= 8'h0;
        rx_byte_buff_1  <= 8'h0;
        rx_byte_buff_2  <= 8'h0;
        rx_byte_buff_3  <= 8'h0;
        rx_byte_buff_4  <= 8'h0;
        rx_byte_buff_5  <= 8'h0;
        rx_byte_buff_6  <= 8'h0;
        rx_byte_buff_7  <= 8'h0;
        rx_byte_buff_8  <= 8'h0;
        rx_byte_buff_9  <= 8'h0;
        rx_byte_buff_10 <= 8'h0;
        rx_byte_buff_11 <= 8'h0;
    end
    else if(curr_state==C_IDLE)
    begin
        rx_byte_buff_0  <= 8'h0;
        rx_byte_buff_1  <= 8'h0;
        rx_byte_buff_2  <= 8'h0;
        rx_byte_buff_3  <= 8'h0;
        rx_byte_buff_4  <= 8'h0;
        rx_byte_buff_5  <= 8'h0;
        rx_byte_buff_6  <= 8'h0;
        rx_byte_buff_7  <= 8'h0;
        rx_byte_buff_8  <= 8'h0;
        rx_byte_buff_9  <= 8'h0;
        rx_byte_buff_10 <= 8'h0;
        rx_byte_buff_11 <= 8'h0;
    end
end

```

```

else if(curr_state==C_CMD_DC)
begin
    case(rx_byte_cnt)
        4'h0:    rx_byte_buff_0 <= rx_fifo_data;
        4'h1:    rx_byte_buff_1 <= rx_fifo_data;
        4'h2:    rx_byte_buff_2 <= rx_fifo_data;
        4'h3:    rx_byte_buff_3 <= rx_fifo_data;
        4'h4:    rx_byte_buff_4 <= rx_fifo_data;
        4'h5:    rx_byte_buff_5 <= rx_fifo_data;
        4'h6:    rx_byte_buff_6 <= rx_fifo_data;
        4'h7:    rx_byte_buff_7 <= rx_fifo_data;
        4'h8:    rx_byte_buff_8 <= rx_fifo_data;
        4'h9:    rx_byte_buff_9 <= rx_fifo_data;
        4'ha:    rx_byte_buff_10 <= rx_fifo_data;
        4'hb:    rx_byte_buff_11 <= rx_fifo_data;
    endcase
end
end

assign is_wb_cmd = (curr_state==C_CMD_DC)
                &&(rx_byte_buff_0=="w")&&(rx_byte_buff_1=="b")
                &&(rx_byte_buff_2==" ")&&(rx_byte_buff_5==" ")
                &&(((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))||
((rx_byte_buff_3>="a")&&(rx_byte_buff_3<="f"))
                &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))
                &&(((rx_byte_buff_6>="0")&&(rx_byte_buff_6<="9"))||
((rx_byte_buff_6>="a")&&(rx_byte_buff_6<="f"))
                &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f")));
assign is_rb_cmd = (curr_state==C_CMD_DC)
                &&(rx_byte_buff_0=="r")&&(rx_byte_buff_1=="b")
                &&(rx_byte_buff_2==" ")
                &&(((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))||
((rx_byte_buff_3>="a")&&(rx_byte_buff_3<="f"))
                &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f")));

assign is_add_cmd = (curr_state==C_CMD_DC)
                &&(rx_byte_buff_0=="a")&&(rx_byte_buff_1=="d")&&
(rx_byte_buff_2=="d")
                &&(rx_byte_buff_3==" ")
                &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))
                &&(((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))||
((rx_byte_buff_5>="a")&&(rx_byte_buff_5<="f"))
                &&(rx_byte_buff_6==" ")
                &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f"))
                &&(((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))||
((rx_byte_buff_8>="a")&&(rx_byte_buff_8<="f"))
                &&(rx_byte_buff_9==" ")
                &&(((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))||
((rx_byte_buff_10>="a")&&(rx_byte_buff_10<="f"))
                &&(((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))||
((rx_byte_buff_11>="a")&&(rx_byte_buff_11<="f")));

assign is_and_cmd = (curr_state==C_CMD_DC)

```

```

        &&(rx_byte_buff_0=="a")&&(rx_byte_buff_1=="n")&&
(rx_byte_buff_2=="d")
        &&(rx_byte_buff_3==" ")
        &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))))
        &&(((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))||
((rx_byte_buff_5>="a")&&(rx_byte_buff_5<="f"))))
        &&(rx_byte_buff_6==" ")
        &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f"))))
        &&(((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))||
((rx_byte_buff_8>="a")&&(rx_byte_buff_8<="f"))))
        &&(rx_byte_buff_9==" ")
        &&(((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))||
((rx_byte_buff_10>="a")&&(rx_byte_buff_10<="f"))))
        &&(((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))||
((rx_byte_buff_11>="a")&&(rx_byte_buff_11<="f"))));

assign is_or_cmd = (curr_state==C_CMD_DC)
        &&(rx_byte_buff_0=="o")&&(rx_byte_buff_1=="r")
        &&(rx_byte_buff_2==" ")
        &&(((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))||
((rx_byte_buff_3>="a")&&(rx_byte_buff_3<="f"))))
        &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))))
        &&(rx_byte_buff_5==" ")
        &&(((rx_byte_buff_6>="0")&&(rx_byte_buff_6<="9"))||
((rx_byte_buff_6>="a")&&(rx_byte_buff_6<="f"))))
        &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f"))))
        &&(rx_byte_buff_8==" ")
        &&(((rx_byte_buff_9>="0")&&(rx_byte_buff_9<="9"))||
((rx_byte_buff_9>="a")&&(rx_byte_buff_9<="f"))))
        &&(((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))||
((rx_byte_buff_10>="a")&&(rx_byte_buff_10<="f"))));

assign is_not_cmd = (curr_state==C_CMD_DC)
        &&(rx_byte_buff_0=="n")&&(rx_byte_buff_1=="o")&&
(rx_byte_buff_2=="t")
        &&(rx_byte_buff_3==" ")
        &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))))
        &&(((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))||
((rx_byte_buff_5>="a")&&(rx_byte_buff_5<="f"))))
        &&(rx_byte_buff_6==" ")
        &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f"))))
        &&(((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))||
((rx_byte_buff_8>="a")&&(rx_byte_buff_8<="f"))));

assign is_xor_cmd = (curr_state==C_CMD_DC)
        &&(rx_byte_buff_0=="x")&&(rx_byte_buff_1=="o")&&
(rx_byte_buff_2=="r")
        &&(rx_byte_buff_3==" ")
        &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))))
        &&(((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))||
((rx_byte_buff_5>="a")&&(rx_byte_buff_5<="f"))))

```

```

        &&(rx_byte_buff_6==" ")
        &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f"))))
        &&(((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))||
((rx_byte_buff_8>="a")&&(rx_byte_buff_8<="f"))))
        &&(rx_byte_buff_9==" ")
        &&(((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))||
((rx_byte_buff_10>="a")&&(rx_byte_buff_10<="f"))))
        &&(((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))||
((rx_byte_buff_11>="a")&&(rx_byte_buff_11<="f"))));

assign is_mul_cmd = (curr_state==C_CMD_DC)
        &&(rx_byte_buff_0=="m")&&(rx_byte_buff_1=="u")&&
(rx_byte_buff_2=="1")
        &&(rx_byte_buff_3==" ")
        &&(((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))||
((rx_byte_buff_4>="a")&&(rx_byte_buff_4<="f"))))
        &&(((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))||
((rx_byte_buff_5>="a")&&(rx_byte_buff_5<="f"))))
        &&(rx_byte_buff_6==" ")
        &&(((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))||
((rx_byte_buff_7>="a")&&(rx_byte_buff_7<="f"))))
        &&(((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))||
((rx_byte_buff_8>="a")&&(rx_byte_buff_8<="f"))))
        &&(rx_byte_buff_9==" ")
        &&(((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))||
((rx_byte_buff_10>="a")&&(rx_byte_buff_10<="f"))))
        &&(((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))||
((rx_byte_buff_11>="a")&&(rx_byte_buff_11<="f"))));

assign is_bonus_cmd = (curr_state==C_CMD_DC)
        &&(rx_byte_buff_0=="b")&&(rx_byte_buff_1=="o")&&
(rx_byte_buff_2=="n")
        &&(rx_byte_buff_3=="u")&&(rx_byte_buff_4=="s");

//至此，完成C_CMD_DC状态中的全部操作

//执行ALU:C_CMD_ADD,C_CMD_AND,C_CMD_OR,C_CMD_NOT,C_CMD_XOR,C_CMD_MUL
//先根据rx_fifo_buff更新ALU_addr
//再根据ALU_addr更新ALU_operand
//对ALU_operand进行add运算，存储在ALU_result中
//根据ALU_addr，将ALU_result存回地址中

reg        ALU_rd_2;
//ALU_addr,ALU_rd
always@(posedge c1k or posedge rst) begin
    if(rst) begin
        ALU_rd_2    <= 1'b0;
        ALU_addr_0   <= 8'h0;//结果存储的地址
        ALU_addr_1   <= 8'h0;
        ALU_addr_2   <= 8'h0;
    end
    else if(curr_state == C_CMD_ADD||curr_state == C_CMD_AND||curr_state ==
C_CMD_XOR||curr_state == C_CMD_MUL) begin
        ALU_rd_2    <= 1'b1;
        if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
            ALU_addr_0[7:4] <= rx_byte_buff_4[3:0];
        else

```

```

        ALU_addr_0[7:4] <= rx_byte_buff_4[2:0] + 4'h9;
    if((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))
        ALU_addr_0[3:0] <= rx_byte_buff_5[3:0];
    else
        ALU_addr_0[3:0] <= rx_byte_buff_5[2:0] + 4'h9;
    if((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))
        ALU_addr_1[7:4] <= rx_byte_buff_7[3:0];
    else
        ALU_addr_1[7:4] <= rx_byte_buff_7[2:0] + 4'h9;
    if((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))
        ALU_addr_1[3:0] <= rx_byte_buff_8[3:0];
    else
        ALU_addr_1[3:0] <= rx_byte_buff_8[2:0] + 4'h9;
    if((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))
        ALU_addr_2[7:4] <= rx_byte_buff_10[3:0];
    else
        ALU_addr_2[7:4] <= rx_byte_buff_10[2:0] + 4'h9;
    if((rx_byte_buff_11>="0")&&(rx_byte_buff_11<="9"))
        ALU_addr_2[3:0] <= rx_byte_buff_11[3:0];
    else
        ALU_addr_2[3:0] <= rx_byte_buff_11[2:0] + 4'h9;
end
else if(curr_state == C_CMD_OR) begin
    ALU_rd_2    <= 1'b1;
    if((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))
        ALU_addr_0[7:4] <= rx_byte_buff_3[3:0];
    else
        ALU_addr_0[7:4] <= rx_byte_buff_3[2:0] + 4'h9;
    if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
        ALU_addr_0[3:0] <= rx_byte_buff_4[3:0];
    else
        ALU_addr_0[3:0] <= rx_byte_buff_4[2:0] + 4'h9;
    if((rx_byte_buff_6>="0")&&(rx_byte_buff_6<="9"))
        ALU_addr_1[7:4] <= rx_byte_buff_6[3:0];
    else
        ALU_addr_1[7:4] <= rx_byte_buff_6[2:0] + 4'h9;
    if((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))
        ALU_addr_1[3:0] <= rx_byte_buff_7[3:0];
    else
        ALU_addr_1[3:0] <= rx_byte_buff_7[2:0] + 4'h9;
    if((rx_byte_buff_9>="0")&&(rx_byte_buff_9<="9"))
        ALU_addr_2[7:4] <= rx_byte_buff_9[3:0];
    else
        ALU_addr_2[7:4] <= rx_byte_buff_9[2:0] + 4'h9;
    if((rx_byte_buff_10>="0")&&(rx_byte_buff_10<="9"))
        ALU_addr_2[3:0] <= rx_byte_buff_10[3:0];
    else
        ALU_addr_2[3:0] <= rx_byte_buff_10[2:0] + 4'h9;
end
else if(curr_state == C_CMD_NOT) begin
    ALU_rd_2    <= 1'b1;
    if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
        ALU_addr_0[7:4] <= rx_byte_buff_4[3:0];
    else
        ALU_addr_0[7:4] <= rx_byte_buff_4[2:0] + 4'h9;
    if((rx_byte_buff_5>="0")&&(rx_byte_buff_5<="9"))
        ALU_addr_0[3:0] <= rx_byte_buff_5[3:0];
    else

```

```

        ALU_addr_0[3:0] <= rx_byte_buff_5[2:0] + 4'h9;
    if((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))
        ALU_addr_1[7:4] <= rx_byte_buff_7[3:0];
    else
        ALU_addr_1[7:4] <= rx_byte_buff_7[2:0] + 4'h9;
    if((rx_byte_buff_8>="0")&&(rx_byte_buff_8<="9"))
        ALU_addr_1[3:0] <= rx_byte_buff_8[3:0];
    else
        ALU_addr_1[3:0] <= rx_byte_buff_8[2:0] + 4'h9;
end
else begin
    ALU_rd_2 <= 1'b0;
    ALU_addr_0 <= 8'h0; //结果存储的地址
    ALU_addr_1 <= 8'h0;
    ALU_addr_2 <= 8'h0;
end
end

always @(posedge clk or posedge rst) begin
    if(rst) ALU_rd <= 1'b0;
    else if(ALU_rd_2==1'b1) ALU_rd <= 1'b1;
    else ALU_rd <= 1'b0;
end

//ALU_wd
always @(posedge clk or posedge rst) begin
    if(rst)
        ALU_wd <= 1'b0;
    else if(((curr_state==C_CMD_ADD)|| (curr_state==C_CMD_AND)||
(curr_state==C_CMD_OR)|| (curr_state==C_CMD_NOT)
|| (curr_state==C_CMD_XOR)|| (curr_state==C_CMD_MUL)))&&ALU_rd)
        ALU_wd <= 1'b1; //通过这个标志位，使得写比读晚一个周期
    else
        ALU_wd <= 1'b0;
end

//ALU_operand
always @(*) begin
    if(ALU_rd_2) begin
        case(ALU_addr_1)
            8'h0: ALU_operand_1 = sw;
            8'h10: ALU_operand_1 = hexplay_buff[7:0];
            8'h11: ALU_operand_1 = hexplay_buff[15:8];
            8'h12: ALU_operand_1 = hexplay_buff[23:16];
            8'h13: ALU_operand_1 = hexplay_buff[31:24];
            8'h20: ALU_operand_1 = store_buff[7:0];
            8'h21: ALU_operand_1 = store_buff[15:8];
            8'h22: ALU_operand_1 = store_buff[23:16];
            8'h23: ALU_operand_1 = store_buff[31:24];
            8'h24: ALU_operand_1 = store_buff[39:32];
            8'h25: ALU_operand_1 = store_buff[47:40];
            8'h26: ALU_operand_1 = store_buff[55:48];
            8'h27: ALU_operand_1 = store_buff[63:56];
            8'h28: ALU_operand_1 = store_buff[71:64];
            8'h29: ALU_operand_1 = store_buff[79:72];
            8'h2a: ALU_operand_1 = store_buff[87:80];

```

```

        8'h2b: ALU_operand_1 = store_buff[95:88];
        8'h2c: ALU_operand_1 = store_buff[103:96];
        8'h2d: ALU_operand_1 = store_buff[111:104];
        8'h2e: ALU_operand_1 = store_buff[119:112];
        8'h2f: ALU_operand_1 = store_buff[127:120];
        default:ALU_operand_1 = 8'h0;
    endcase
    case(ALU_addr_2)
        8'h0: ALU_operand_2 = sw;
        8'h10: ALU_operand_2 = hexplay_buff[7:0];
        8'h11: ALU_operand_2 = hexplay_buff[15:8];
        8'h12: ALU_operand_2 = hexplay_buff[23:16];
        8'h13: ALU_operand_2 = hexplay_buff[31:24];
        8'h20: ALU_operand_2 = store_buff[7:0];
        8'h21: ALU_operand_2 = store_buff[15:8];
        8'h22: ALU_operand_2 = store_buff[23:16];
        8'h23: ALU_operand_2 = store_buff[31:24];
        8'h24: ALU_operand_2 = store_buff[39:32];
        8'h25: ALU_operand_2 = store_buff[47:40];
        8'h26: ALU_operand_2 = store_buff[55:48];
        8'h27: ALU_operand_2 = store_buff[63:56];
        8'h28: ALU_operand_2 = store_buff[71:64];
        8'h29: ALU_operand_2 = store_buff[79:72];
        8'h2a: ALU_operand_2 = store_buff[87:80];
        8'h2b: ALU_operand_2 = store_buff[95:88];
        8'h2c: ALU_operand_2 = store_buff[103:96];
        8'h2d: ALU_operand_2 = store_buff[111:104];
        8'h2e: ALU_operand_2 = store_buff[119:112];
        8'h2f: ALU_operand_2 = store_buff[127:120];
        default:ALU_operand_2 = 8'h0;
    endcase
end
end

wire [7:0] ALU_ADD_result; //加法器
add_8
add(.s(ALU_ADD_result),.cout(),.a(ALU_operand_1),.b(ALU_operand_2),.cin(1'b0));
wire [7:0] ALU_MUL_result; //乘法器
mul mul(.a(ALU_operand_1),.b(ALU_operand_2),.out(ALU_MUL_result));

always @(*) begin
    if(ALU_rd) begin
        if(curr_state==C_CMD_ADD)
            ALU_result = ALU_ADD_result;
        else if(curr_state==C_CMD_AND)
            ALU_result = ALU_operand_1 & ALU_operand_2;
        else if(curr_state==C_CMD_OR)
            ALU_result = ALU_operand_1 | ALU_operand_2;
        else if(curr_state==C_CMD_NOT)
            ALU_result = ~ ALU_operand_1;
        else if(curr_state==C_CMD_XOR)
            ALU_result = ALU_operand_1 ^ ALU_operand_2;
        else if(curr_state==C_CMD_MUL)
            ALU_result = ALU_MUL_result;
    end
end
end

```

```

//ALU_result
//always @(*) begin
//    ALU_result = ALU_operand_1 + ALU_operand_2;
//end

//执行C_CMD_WB指令
//根据rx_fifo_buff_3和4 更新 wr_addr,根据rx_fifo_buff_6和7 更新wr_data
//置wr_en=1,根据 wr_data, 将wr_data 更新到 led 或者 hexplay_buff准备输出到数码管

//wr_addr,wr_data,wr_en
always@(posedge clk or posedge rst)
begin
    if(rst) begin
        wr_en    <= 1'b0;
        wr_addr[7:4] <= 4'h0;
        wr_addr[3:0] <= 4'h0;
        wr_data[7:4] <= 4'h0;
        wr_data[3:0] <= 4'h0;
    end
    else if(curr_state == C_CMD_WB) begin
        wr_en    <= 1'b1;
        if(ALU_wd==1'b1) begin//上一个状态是进行运算
            wr_addr <= ALU_addr_0;
            wr_data <= ALU_result;
        end
        else begin//直接写的操作
            if((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))
                wr_addr[7:4] <= rx_byte_buff_3[3:0];
            else
                wr_addr[7:4] <= rx_byte_buff_3[2:0] + 4'h9;
            if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
                wr_addr[3:0] <= rx_byte_buff_4[3:0];
            else
                wr_addr[3:0] <= rx_byte_buff_4[2:0] + 4'h9;
            if((rx_byte_buff_6>="0")&&(rx_byte_buff_6<="9"))
                wr_data[7:4] <= rx_byte_buff_6[3:0];
            else
                wr_data[7:4] <= rx_byte_buff_6[2:0] + 4'h9;
            if((rx_byte_buff_7>="0")&&(rx_byte_buff_7<="9"))
                wr_data[3:0] <= rx_byte_buff_7[3:0];
            else
                wr_data[3:0] <= rx_byte_buff_7[2:0] + 4'h9;
        end
    end
end
else begin
    wr_en    <= 1'b0;
    wr_addr[7:4] <= 4'h0;
    wr_addr[3:0] <= 4'h0;
    wr_data[7:4] <= 4'h0;
    wr_data[3:0] <= 4'h0;
end
end

//led,hexplay_buff
//这个就作为程序内部的存储
always@(posedge clk or posedge rst)
begin

```



```

if(rst)
begin
    led          <= 8'h0;
    hexplay_buff <= 32'h0;
    store_buff   <= 128'h0;
end
else if(wr_en)
begin
    case(wr_addr)
        8'h0:   led          <= wr_data;
        8'h10:  hexplay_buff[7:0] <= wr_data;
        8'h11:  hexplay_buff[15:8] <= wr_data;
        8'h12:  hexplay_buff[23:16] <= wr_data;
        8'h13:  hexplay_buff[31:24] <= wr_data;
        8'h20:  store_buff[7:0] <= wr_data;
        8'h21:  store_buff[15:8] <= wr_data;
        8'h22:  store_buff[23:16] <= wr_data;
        8'h23:  store_buff[31:24] <= wr_data;
        8'h24:  store_buff[39:32] <= wr_data;
        8'h25:  store_buff[47:40] <= wr_data;
        8'h26:  store_buff[55:48] <= wr_data;
        8'h27:  store_buff[63:56] <= wr_data;
        8'h28:  store_buff[71:64] <= wr_data;
        8'h29:  store_buff[79:72] <= wr_data;
        8'h2a:  store_buff[87:80] <= wr_data;
        8'h2b:  store_buff[95:88] <= wr_data;
        8'h2c:  store_buff[103:96] <= wr_data;
        8'h2d:  store_buff[111:104] <= wr_data;
        8'h2e:  store_buff[119:112] <= wr_data;
        8'h2f:  store_buff[127:120] <= wr_data;
    endcase
end
end

//以上，处理完C_CMD_WB状态

//开始处理C_CMD_RB状态，这个状态会转移到向shell输出的C_TXFIFO_WR状态
//根据rx_byte_buff_3和4更新rd_addr
//置tx_byte_cnt=2，会向shell输出2byte数据
//置rd_en=1,根据rd_addr,由sw或对应的hexplay_buff 更新rd_data
//根据rd_data，更新tx_fifo_buff，为向shell输出做准备

//rd_addr,rd_data,rd_en
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        rd_en          <= 1'b0;
        rd_addr[7:4] <= 4'h0;
        rd_addr[3:0] <= 4'h0;
    end
    else if(curr_state==C_CMD_RB)//当前指令是rb，处理地址，将ASCII码转化为数字
    begin
        rd_en    <= 1'b1;
        // '1'的ASCII码为：00110001
        // 'a'的ASCII码为：01100001
        if((rx_byte_buff_3>="0")&&(rx_byte_buff_3<="9"))
            rd_addr[7:4] <= rx_byte_buff_3[3:0];
    end
end

```

```

        else
            rd_addr[7:4] <= rx_byte_buff_3[2:0] + 4'h9;
            if((rx_byte_buff_4>="0")&&(rx_byte_buff_4<="9"))
                rd_addr[3:0] <= rx_byte_buff_4[3:0];
            else
                rd_addr[3:0] <= rx_byte_buff_4[2:0] + 4'h9;
            end
        else
            begin
                rd_en    <= 1'b0;
                rd_addr[7:4] <= 4'h0;
                rd_addr[3:0] <= 4'h0;
            end
        end

//rd_data
always@(*)//不要设置为时钟周期上升沿
begin
    if(rd_en)
        begin
            case(rd_addr)
                8'h0:   rd_data = sw;
                8'h10:  rd_data = hexplay_buff[7:0];
                8'h11:  rd_data = hexplay_buff[15:8];
                8'h12:  rd_data = hexplay_buff[23:16];
                8'h13:  rd_data = hexplay_buff[31:24];
                8'h20:  rd_data = store_buff[7:0];
                8'h21:  rd_data = store_buff[15:8];
                8'h22:  rd_data = store_buff[23:16];
                8'h23:  rd_data = store_buff[31:24];
                8'h24:  rd_data = store_buff[39:32];
                8'h25:  rd_data = store_buff[47:40];
                8'h26:  rd_data = store_buff[55:48];
                8'h27:  rd_data = store_buff[63:56];
                8'h28:  rd_data = store_buff[71:64];
                8'h29:  rd_data = store_buff[79:72];
                8'h2a:  rd_data = store_buff[87:80];
                8'h2b:  rd_data = store_buff[95:88];
                8'h2c:  rd_data = store_buff[103:96];
                8'h2d:  rd_data = store_buff[111:104];
                8'h2e:  rd_data = store_buff[119:112];
                8'h2f:  rd_data = store_buff[127:120];
                default:rd_data = 8'h0;
            endcase
        end
    else
        rd_data = 8'h0;
    end
end

//以上，处理完C_CMD_WR状态

//C_TXFIFO_WR和C_TXFIFO_WIAT状态中：
//首先根据tx_byte_cnt，用tx_byte_buff 更新tx_fifo_din
//置tx_fifo的写使能 tx_fifo_wr_en为1，将tx_fifo_din存入tx_fifo
//每clk,tx_byte_cnt减一
//tx_fifo输出为tx_data,通过tx模块转化为tx信号，通过串口输出到FPGA的shell

//tx_fifo_din,tx_fifo_wr_en

```

```

always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        tx_fifo_wr_en    <= 1'b0;
        tx_fifo_din      <= 8'h0;
    end
    else if(curr_state==C_TXFIFO_WR)
    begin
        tx_fifo_wr_en    <= 1'b1;
        case(tx_byte_cnt)
            8'hf: tx_fifo_din <= tx_byte_buff_15;
            8'he: tx_fifo_din <= tx_byte_buff_14;
            8'hd: tx_fifo_din <= tx_byte_buff_13;
            8'hc: tx_fifo_din <= tx_byte_buff_12;
            8'hb: tx_fifo_din <= tx_byte_buff_11;
            8'ha: tx_fifo_din <= tx_byte_buff_10;
            8'h9: tx_fifo_din <= tx_byte_buff_9;
            8'h8: tx_fifo_din <= tx_byte_buff_8;
            8'h7: tx_fifo_din <= tx_byte_buff_7;
            8'h6: tx_fifo_din <= tx_byte_buff_6;
            8'h5: tx_fifo_din <= tx_byte_buff_5;
            8'h4: tx_fifo_din <= tx_byte_buff_4;
            8'h3: tx_fifo_din <= tx_byte_buff_3;
            8'h2: tx_fifo_din <= tx_byte_buff_2;
            8'h1: tx_fifo_din <= tx_byte_buff_1;
            8'h0: tx_fifo_din <= tx_byte_buff_0;
            default:tx_fifo_din <= 8'h0;
        endcase
    end
    else
    begin
        tx_fifo_wr_en    <= 1'b0;
        tx_fifo_din      <= 8'h0;
    end
end

```

//刷新输出到FPGA led以及数码管

```

reg [20:0] hexplay_cnt;
always @(posedge clk or posedge rst) begin
    if(rst) hexplay_cnt=0;
    else hexplay_cnt<=hexplay_cnt+21'd1;
end

```

```

assign hexplay_an=hexplay_cnt[20:18];

```

```

always @(*) begin
    case (hexplay_an)
        3'd0: hexplay_data<=hexplay_buff[3:0];
        3'd1: hexplay_data<=hexplay_buff[7:4];
        3'd2: hexplay_data<=hexplay_buff[11:8];
        3'd3: hexplay_data<=hexplay_buff[15:12];
        3'd4: hexplay_data<=hexplay_buff[19:16];
        3'd5: hexplay_data<=hexplay_buff[23:20];
        3'd6: hexplay_data<=hexplay_buff[27:24];
        3'd7: hexplay_data<=hexplay_buff[31:28];
    endcase

```

```
end
```

```
endmodule
```

tx.v

```
//本模块实现了串口数据的输出
//当输出开始,信号 tx_ready 为 1 时,开始将 tx_data 转化为协议规定好的 tx 进行输出
//一次发送8bits数据, tx_rd关联tx模块与tx_fifo模块
//当tx_fifo非空, 并且tx模块还没有开始转化输出时, tx_rd=1
//此时这个信号控制tx_fifo读取一个8bit的数据, 送入tx模块的tx_data

module tx(
    input          clk,rst,
    output reg     tx,//输出信号
    input          tx_ready,//当tx_fifo非空时, 这个标志位为1, 开始转化输出
    output reg     tx_rd,
    input [7:0]    tx_data
);
//输入是时钟信号, 复位信号。要输出的8bit数据, 以及开始输出数据的标志信号
//当标志信号为1, 则按照868为周期输出10个信号, 包括起始的低位信号, 和结束的高位信号
parameter DIV_CNT    = 10'd867;
parameter TX_CNT     = 4'h9;//已发送数据的上限
parameter C_IDLE     = 1'b0;//空闲状态
parameter C_TX       = 1'b1;//发送状态

reg          curr_state,next_state;
reg [9:0]    div_cnt;//时钟周期数
reg [4:0]    tx_cnt;//已发送的数据数目
reg [7:0]    tx_reg;//要发送的数据

always@(posedge clk or posedge rst)
begin
    if(rst)
        curr_state <= C_IDLE;
    else
        curr_state <= next_state;
end

always@(*)
begin
    case(curr_state)
        C_IDLE:
            if(tx_ready==1'b1)
                next_state = C_TX;
            else
                next_state = C_IDLE;
        C_TX:
            if((div_cnt==DIV_CNT)&&(tx_cnt>=TX_CNT))//数据输出数目达到上限
                next_state = C_IDLE;
            else
                next_state = C_TX;
    endcase
end
```

```

always@(posedge clk or posedge rst)
begin
    if(rst)
        div_cnt <= 10'h0;
    else if(curr_state==C_TX)//当前状态是输出状态
    begin
        if(div_cnt>=DIV_CNT)
            div_cnt <= 10'h0;
        else
            div_cnt <= div_cnt + 10'h1;
    end
    else//当前状态是空闲状态
        div_cnt <= 10'h0;
end

always@(posedge clk or posedge rst)
begin
    if(rst)
        tx_cnt <= 4'h0;
    else if(curr_state==C_TX)//当前状态是输出状态
    begin
        if(div_cnt==DIV_CNT)//输出一个可读入的数据
            tx_cnt <= tx_cnt + 1'b1;
        end
    else
        tx_cnt <= 4'h0;
end

//tx_rd,即将开始输出标志,或者获取tx_data模块
always@(posedge clk or posedge rst)
begin
    if(rst)
        tx_rd <= 1'b0;
    else if((curr_state==C_IDLE)&&(tx_ready==1'b1))//输出完成
        tx_rd <= 1'b1;
    else
        tx_rd <= 1'b0;
end

//此时由tx_data更新tx_reg
always@(posedge clk or posedge rst)
begin
    if(rst)
        tx_reg <= 8'b0;
    else if((curr_state==C_IDLE)&&(tx_ready==1'b1))
        tx_reg <= tx_data;
end

always@(posedge clk or posedge rst)
begin
    if(rst)
        tx <= 1'b1;//空闲
    else if(curr_state==C_IDLE)
        tx <= 1'b1;//空闲
    else if(div_cnt==10'h0)//输出状态
    begin
        case(tx_cnt)
            4'h0: tx <= 1'b0;//第一个低位数据

```

```

        4'h1: tx <= tx_reg[0];
        4'h2: tx <= tx_reg[1];
        4'h3: tx <= tx_reg[2];
        4'h4: tx <= tx_reg[3];
        4'h5: tx <= tx_reg[4];
        4'h6: tx <= tx_reg[5];
        4'h7: tx <= tx_reg[6];
        4'h8: tx <= tx_reg[7];
        4'h9: tx <= 1'b1;
    endcase
end
end
endmodule

```

rx.v

```

//本模块实现了串口数据的输入
//开始将 rx 转化为协议规定好的 rx_data 进行输入
//一次接收8bits数据，存储到rx_data,需及时使用
//完成输入8bit的数据后，rx_vld=1

module rx(
    input                clk,rst,
    input                rx,//输入的数据
    output reg           rx_vld,//读完所有数据的标志位
    output reg [7:0]     rx_data//存储的读入数据
);

parameter DIV_CNT      = 10'd867;
parameter HDIV_CNT     = 10'd433;
parameter RX_CNT       = 4'h8;
parameter C_IDLE       = 1'b0;//空闲状态
parameter C_RX         = 1'b1;//接收状态

reg curr_state;
reg next_state;
reg [9:0] div_cnt;
reg [3:0] rx_cnt;//记录已经接收了多少数据
reg
rx_reg_0,rx_reg_1,rx_reg_2,rx_reg_3,rx_reg_4,rx_reg_5,rx_reg_6,rx_reg_7;
wire rx_pulse;

always@(posedge clk or posedge rst)
begin
    if(rst)
        curr_state <= C_IDLE;
    else
        curr_state <= next_state;
end

always@(*)
begin
    case(curr_state)
        C_IDLE:
            if(div_cnt==HDIV_CNT)//低电平进入，下一个状态为接收状态

```

```

        next_state = C_RX;
    else
        next_state = C_IDLE;
    C_RX:
        if((div_cnt==DIV_CNT)&&(rx_cnt>=RX_CNT))//8位数据接收完毕
            next_state = C_IDLE;
        else
            next_state = C_RX;
    endcase
end

always@(posedge clk or posedge rst)
begin
    if(rst)
        div_cnt <= 10'h0;
    else if(curr_state == C_IDLE)//在空闲状态
    begin
        if(rx==1'b1)//空闲帧
            div_cnt <= 10'h0;
        else if(div_cnt < HDIV_CNT)//不在空闲帧，即第一个低位检测到，开始输入数据
            div_cnt <= div_cnt + 10'h1;
        else//第一个开始计数的低位检测完毕
            div_cnt <= 10'h0;
        end
    else if(curr_state == C_RX)//在接收状态。计数器在0~867之间循环
    begin
        if(div_cnt >= DIV_CNT)
            div_cnt <= 10'h0;
        else
            div_cnt <= div_cnt + 10'h1;
        end
    end

always@(posedge clk or posedge rst)
begin
    if(rst)
        rx_cnt <= 4'h0;
    else if(curr_state == C_IDLE)
        rx_cnt <= 4'h0;
    else if((div_cnt == DIV_CNT)&&(rx_cnt<4'hF))//在接收状态并且当前新读入一个数据
        rx_cnt <= rx_cnt + 1'b1;
    end

assign rx_pulse = (curr_state==C_RX)&&(div_cnt==DIV_CNT);//读新数据的一个时钟周期

always@(posedge clk)
begin
    if(rx_pulse)
    begin
        case(rx_cnt)
            4'h0: rx_reg_0 <= rx;
            4'h1: rx_reg_1 <= rx;
            4'h2: rx_reg_2 <= rx;
            4'h3: rx_reg_3 <= rx;
            4'h4: rx_reg_4 <= rx;
            4'h5: rx_reg_5 <= rx;
            4'h6: rx_reg_6 <= rx;
            4'h7: rx_reg_7 <= rx;

```

```

        endcase//根据已读的数据数量，将当前数据读入
    end
end

always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        rx_vld <= 1'b0;
        rx_data <= 8'h55;
    end
    else if((curr_state==C_RX)&&(next_state==C_IDLE))//已经读完所有数据
    begin
        rx_vld <= 1'b1;
        rx_data <=
{rx_reg_7,rx_reg_6,rx_reg_5,rx_reg_4,rx_reg_3,rx_reg_2,rx_reg_1,rx_reg_0};
    end
    else
        rx_vld <= 1'b0;
end
endmodule

```

add_8.v

```

// 8bits 加法器
module add_8(
output [7:0] s,
output cout,
input [7:0] a,b,
input cin
);
wire [6:0] carry;

add add0(s[0], carry[0], a[0], b[0], cin);
add add1(s[1], carry[1], a[1], b[1], carry[0]);
add add2(s[2], carry[2], a[2], b[2], carry[1]);
add add3(s[3], carry[3], a[3], b[3], carry[2]);
add add4(s[4], carry[4], a[4], b[4], carry[3]);
add add5(s[5], carry[5], a[5], b[5], carry[4]);
add add6(s[6], carry[6], a[6], b[6], carry[5]);
add add7(s[7], cout, a[7], b[7], carry[6]);
endmodule

```

add.v


```
//一位全加器
module add(
output s, cout,
input a, b, cin
);

assign s = a ^ b ^ cin;
assign cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

mul.v

```
//8bits 乘法器
module mul(
input  [7:0]  a,b,
output [7:0]  out
);

wire  [7:0]  mul_1;
wire  [7:0]  mul_2;
wire  [7:0]  mul_3;
wire  [7:0]  mul_4;
wire  [7:0]  mul_5;
wire  [7:0]  mul_6;
wire  [7:0]  mul_7;
wire  [7:0]  mul_8;

assign mul_1 = (b[0]==1'b1)?(a):(8'h0);
assign mul_2 = (b[1]==1'b1)?({a[6:0],1'b0}):(8'h0);
assign mul_3 = (b[2]==1'b1)?({a[5:0],2'b0}):(8'h0);
assign mul_4 = (b[3]==1'b1)?({a[4:0],3'b0}):(8'h0);
assign mul_5 = (b[4]==1'b1)?({a[3:0],4'b0}):(8'h0);
assign mul_6 = (b[5]==1'b1)?({a[2:0],5'b0}):(8'h0);
assign mul_7 = (b[6]==1'b1)?({a[1:0],6'b0}):(8'h0);
assign mul_8 = (b[7]==1'b1)?({a[0],7'b0}):(8'h0);

wire  [7:0]  tmp_1;
wire  [7:0]  tmp_2;
wire  [7:0]  tmp_3;
wire  [7:0]  tmp_4;
wire  [7:0]  tmp_5;
wire  [7:0]  tmp_6;

add_8  add1(.s(tmp_1),.cout(),.a(mul_1),.b(mul_2),.cin(1'b0));
add_8  add2(.s(tmp_2),.cout(),.a(mul_3),.b(mul_4),.cin(1'b0));
add_8  add3(.s(tmp_3),.cout(),.a(mul_5),.b(mul_6),.cin(1'b0));
add_8  add4(.s(tmp_4),.cout(),.a(mul_7),.b(mul_8),.cin(1'b0));
add_8  add5(.s(tmp_5),.cout(),.a(tmp_1),.b(tmp_2),.cin(1'b0));
add_8  add6(.s(tmp_6),.cout(),.a(tmp_3),.b(tmp_4),.cin(1'b0));
add_8  add7(.s(out),.cout(),.a(tmp_5),.b(tmp_6),.cin(1'b0));

endmodule
```

lab10_c.xdc

//引脚约束文件

```
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clk
}];
set_property -dict { PACKAGE_PIN N17     IOSTANDARD LVCMOS33 } [get_ports { rst
}];
#uart
set_property -dict { PACKAGE_PIN C4      IOSTANDARD LVCMOS33 } [get_ports { rx }];
set_property -dict { PACKAGE_PIN D4      IOSTANDARD LVCMOS33 } [get_ports { tx }];

#Switch
set_property -dict { PACKAGE_PIN H16     IOSTANDARD LVCMOS33 } [get_ports { sw[7]
}];
set_property -dict { PACKAGE_PIN G13     IOSTANDARD LVCMOS33 } [get_ports { sw[6]
}];
set_property -dict { PACKAGE_PIN F13     IOSTANDARD LVCMOS33 } [get_ports { sw[5]
}];
set_property -dict { PACKAGE_PIN E16     IOSTANDARD LVCMOS33 } [get_ports { sw[4]
}];
set_property -dict { PACKAGE_PIN H14     IOSTANDARD LVCMOS33 } [get_ports { sw[3]
}];
set_property -dict { PACKAGE_PIN G16     IOSTANDARD LVCMOS33 } [get_ports { sw[2]
}];
set_property -dict { PACKAGE_PIN F16     IOSTANDARD LVCMOS33 } [get_ports { sw[1]
}];
set_property -dict { PACKAGE_PIN D14     IOSTANDARD LVCMOS33 } [get_ports { sw[0]
}];
#Led
set_property -dict { PACKAGE_PIN G18     IOSTANDARD LVCMOS33 } [get_ports { led[7]
}];
set_property -dict { PACKAGE_PIN F18     IOSTANDARD LVCMOS33 } [get_ports { led[6]
}];
set_property -dict { PACKAGE_PIN E17     IOSTANDARD LVCMOS33 } [get_ports { led[5]
}];
set_property -dict { PACKAGE_PIN D17     IOSTANDARD LVCMOS33 } [get_ports { led[4]
}];
set_property -dict { PACKAGE_PIN G17     IOSTANDARD LVCMOS33 } [get_ports { led[3]
}];
set_property -dict { PACKAGE_PIN E18     IOSTANDARD LVCMOS33 } [get_ports { led[2]
}];
set_property -dict { PACKAGE_PIN D18     IOSTANDARD LVCMOS33 } [get_ports { led[1]
}];
set_property -dict { PACKAGE_PIN C17     IOSTANDARD LVCMOS33 } [get_ports { led[0]
}];
#HEX_SEG
set_property -dict { PACKAGE_PIN A18     IOSTANDARD LVCMOS33 } [get_ports {
hexplay_an[2] }];
set_property -dict { PACKAGE_PIN B16     IOSTANDARD LVCMOS33 } [get_ports {
hexplay_an[1] }];
set_property -dict { PACKAGE_PIN B17     IOSTANDARD LVCMOS33 } [get_ports {
hexplay_an[0] }];
set_property -dict { PACKAGE_PIN A15     IOSTANDARD LVCMOS33 } [get_ports {
hexplay_data[3] }];
```

```
set_property -dict { PACKAGE_PIN A16    IOSTANDARD LVCMOS33 } [get_ports {  
hexplay_data[2] }];  
set_property -dict { PACKAGE_PIN A13    IOSTANDARD LVCMOS33 } [get_ports {  
hexplay_data[1] }];  
set_property -dict { PACKAGE_PIN A14    IOSTANDARD LVCMOS33 } [get_ports {  
hexplay_data[0] }];
```