

（深圳）

实验报告

开课学期： 2024秋季

课程名称：数字逻辑设计（实验）

实验名称： 综合实验

实验性质： 综合设计型

实验学时： 6 地点： T2612

学生班级： 7班

学生学号： 2023311709

学生姓名： 宁中昊

评阅教师：

报告成绩：

实验与创新实践教育中心制

2024年10月

注：本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明设计的成果和特色。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

|  |
| --- |
| 设计的功能描述 |
| 概述基本功能  主要分为两个功能：   1. 数据发送（uart\_send）   使用拨码开关SW7-SW0，输入十六进制ASCII码，通过串口软件Supercom发送到电脑端。   1. 数据接收（uart\_recv）   使用串口软件Supercom发送十六进制数据（不带换行），用开发板接收，并将电脑端发送的数据显示在数码管上。 |
| 系统设计 |
| 用硬件框图描述系统主要功能及各模块之间的相互关系  一、系统设计的硬件框图  框图  二、文字说明  各模块功能：   1. 顶层模块   分为三个部分，分别对应于接收数据、发送数据与最顶层的功能连接。一开始的构思是两个功能共用一个顶层模块，但是由于接收与发送两个功能相对独立，且都其顶层设计都需要两个模块相连，因此将两者分开，先在对应的顶层模块连接好实现功能再一起连接到最顶层。  第一部分top\_design：最顶层模块，连接uart\_recv\_top模块与uart\_send\_top模块，将两部分功能的模块连接到一起。这两个部分是相对独立的，uart\_top向top\_design传入LED控制的相关数据led\_en与led\_cx并由top\_desgin连接到对应数码管的管脚，uart\_send通过tx向top\_desgin传入待发送数据，由top\_design连接到UART功能的发送管脚。  第二部分uart\_recv\_top：接收功能的顶层模块，连接实现接收功能的两个模块uart\_recv与led\_ctrl，将UART接收端口的信号rx接入uart\_recv模块的输入信号din，再将其产生的输出信号data与有效信号valid输入led\_ctrl模块中，输出对应的LED控制信号led\_en与led\_cx。  第三部分uart\_send\_top：发送功能的顶层模块，将拨码开关对应的8位数据switch与按钮S3的信号button输入uart\_send，再将button\_ctrl模块产生的输出信号dout与有效信号valid接到uart\_send。   1. 接收功能模块   总体上分为两部分：接收实现uart\_recv与数码管控制led\_ctrl，而在led\_ctrl中，对数码管显示数字0-9与字母A-F对应的led\_cx赋值的操作经常重复使用，且涉及到大量常量的赋值，多次重复书写没有意义，因此单独将其提取出来成为一个新的模块led\_display，由于其未涉及功能实现没有将其纳入框图之中。  uart\_recv：接收信号，并将其转换为8位信号data，每一次接收8位信号完毕时拉高有效信号valid。  led\_ctrl：将data中表示的数据显示在数码管上，只显示最近接收的8个字符。不足8个字符高位不显示。  led\_display：根据输入其中的信号产生对应数据为显示在数码管上的管脚控制信号。   1. 发送功能模块   分为两个部分：发送实现uart\_send与按钮控制button\_ctrl，其中uart\_send模块与实验4相同，button\_ctrl中的大部分功能与实验3中相同。  uart\_send：button\_ctrl每给到一次高位有效信号，就将拨码开关表示的十六进制ASCII码对应的数据发送出去。  button\_ctrl：每次按下按钮S3就拉高一次有效信号valid。 |
| 模块设计与实现 |
| 包括各子模块设计思路，输入、输出端口及关键代码  一、顶层模块RTL分析图   1. RTL_top_designtop\_design 2. uart\_recv\_top   RTL_uart_recv_top   1. RTL_uart_send_topuart\_send\_top   总体显示：   1. RTL_whole子模块设计思路、输入输出端口与关键代码   （1）uart\_recv  设计思路：三段式实现状态机，重难点在于描述好状态转移条件，以控制采样时间在信号中段。  使用一个计数器来控制波特率，在起始状态达到计数到波特率周期的一半的时候进入数据状态，接着以一个波特率周期为时间间隔进行采样，如此则可以控制所有采样点在信号中段。  输入端口：clk、rst、din  输出端口：valid、data  关键代码：  //\* 三段式实现状态机      // 第1个always块，描述次态迁移到现态      always @(posedge clk or posedge rst) begin          if (rst) current\_state <= IDLE;          else     current\_state <= next\_state;      end      // 第2个always块，描述状态转移条件判断      always @(\*) begin          case (current\_state)              // 空闲状态: 接收到起始信号 din == 0 后进入起始状态              IDLE: begin                  if(din == 0) begin                      next\_state = START;                  end else next\_state = IDLE;              end              // 起始状态: 持续半个波特率, 后进入数据状态              // 保证之后采样间隔为一个波特率的同时, 在信号的中段采样              START: begin                  if(baud\_counter == BAUD\_HALF) begin                      next\_state = DATA;                  end else next\_state = START;              end              // 数据状态: 传入 8 位数据后进入停止状态              DATA: begin                  if (bit\_index == 8) begin                      next\_state = STOP;                  end else next\_state = DATA;              end              // 停止状态: 继续采样, 接受到停止信号 din == 1 后转入空闲状态              STOP: begin                  if(baud\_counter == BAUD\_END) begin                      next\_state = IDLE;                  end else next\_state = STOP;              end              default: next\_state = IDLE;          endcase      end      // 第3个always块，描述输出逻辑      always @(posedge clk or posedge rst) begin          if(rst) begin              data <= 0;              data\_saved <= 0;              valid <= 0;              baud\_counter <= 0;          end else begin              case(current\_state)                  // 空闲状态: 复位计数器与有效信号                  IDLE: begin                      valid <= 0;                      baud\_counter <= 0;                  end                  // 起始状态: 计数半个波特率, 完成后复位波特率计数器与位计数器                  START: begin                      if(baud\_counter == BAUD\_HALF) begin                          baud\_counter <= 0;                          bit\_index <= 0;                      end else begin                          baud\_counter <= baud\_counter + 1;                      end                  end                  // 数据状态: 以波特率为间隔采样, 逐位传入数据                  DATA: begin                      if(baud\_counter == BAUD\_END) begin                          data\_saved[bit\_index] <= din;                          baud\_counter <= 0;                          if(bit\_index == 8) begin                              baud\_counter <= 0;                          end else begin                              bit\_index <= bit\_index + 1;                          end                      end else begin                          baud\_counter <= baud\_counter + 1;                      end                  end                  // 停止状态: 继续采样, 接收到停止信号 din == 1 后将 valid 拉高, 并将暂存的数据赋值给输出端口                  STOP: begin                      if(baud\_counter == BAUD\_END) begin                          if(din == 1) begin                              baud\_counter <= 0;                              data <= data\_saved;                              valid <= 1;                          end else baud\_counter <= 0;                      end else baud\_counter = baud\_counter + 1;                  end                  default: begin                      valid <= 0;                      data <= 0;                  end              endcase          end  end  状态转移图：  状态转移图  （2）led\_ctrl  设计思路：基本思路是LED轮询工作，但是实现过程中还是有一些重难点。  重点：串口软件Supercom只能发送十六位数据，而十六进制包含了两个八位数据，因此需要两个数据两个数据地保存更新。解决方案：将接收到的数据拆分为两段，分别保存。  难点：会出现大量重复且没有意义的赋值代码，很麻烦，且会导致按钮按下两次才能正确地在数码管上显示。解决方案：使用循环，将重复的赋值语句拆分为新的led\_display模块。  输入端口：clk、rst、data、valid  输出端口：led\_en、led\_cx  关键代码：      // 串口软件只能发送十六进制信号, 也就是一次会发送两个信号, 所以需要将一次接收两个信号      always @(posedge clk or posedge rst) begin          if (rst) begin              flag <= 0;              data\_former <= 5'h1f;              data\_latter <= 5'h1f;              buffer\_cnt <= 0;              for(i=0; i<8; i=i+1) begin                  buffer[i] <= 5'h1f;              end          end else if (valid) begin              data\_former <= data[7:4];              data\_latter <= data[3:0];              flag <= 1;          end else if (flag) begin              for (n=7; n>1; n=n-1) begin                  buffer[n] <= buffer[n-2];              end              buffer\_cnt <= buffer\_cnt +1;              buffer[1] <= data\_former;              buffer[0] <= data\_latter;              flag <= 0;          end      end      // 循环实例化 led\_display 模块, 将数据转化为 led\_cx      genvar q;      generate          for(q=0; q<8; q=q+1) begin: u\_led\_display              led\_display u\_led\_display(                  .data       (buffer  [q]),                  .led\_ctrl\_cx(data\_to\_led[q])          );        end      endgenerate      // LED 灯轮询工作      always @(posedge clk or posedge rst) begin          if (rst) begin              led\_cnt <= 0;              led\_index <= 0;          end else begin              if (led\_cnt == REFRESH\_RATE) begin                  led\_cnt <= 0;                  if(led\_index == 7) begin                      led\_index <= 0;                  end else led\_index <= led\_index + 1;              end else begin                  led\_cnt = led\_cnt + 1;            end          end      end      always @(posedge clk or posedge rst) begin          if (rst) begin              led\_cx <= 8'b11111111;          end else begin              led\_cx <= data\_to\_led[led\_index];          end      end      always @(posedge clk or posedge rst) begin          if (rst) begin              led\_en <= 8'b11111111;          end else begin              case(led\_index)                  3'd0: led\_en <= 8'b11111110;                  3'd1: led\_en <= 8'b11111101;                  3'd2: led\_en <= 8'b11111011;                  3'd3: led\_en <= 8'b11110111;                  3'd4: led\_en <= 8'b11101111;                  3'd5: led\_en <= 8'b11011111;                  3'd6: led\_en <= 8'b10111111;                  3'd7: led\_en <= 8'b01111111;              endcase          end      end  （3）led\_display  设计思路：将循环的赋值语句拆分出来。  输入端口：data  输出端口：led\_ctrl\_cx  关键代码：      always @(\*) begin          if (data == 5'h1f) led\_ctrl\_cx = 8'b11111111; // 不显示          else begin              case (data)                  4'h0:    led\_ctrl\_cx = 8'b00000011; // 0                  4'h1:    led\_ctrl\_cx = 8'b10011111; // 1                  4'h2:    led\_ctrl\_cx = 8'b00100101; // 2                  4'h3:    led\_ctrl\_cx = 8'b00001101; // 3                  4'h4:    led\_ctrl\_cx = 8'b10011001; // 4                  4'h5:    led\_ctrl\_cx = 8'b01001001; // 5                  4'h6:    led\_ctrl\_cx = 8'b01000001; // 6                  4'h7:    led\_ctrl\_cx = 8'b00011111; // 7                  4'h8:    led\_ctrl\_cx = 8'b00000001; // 8                  4'h9:    led\_ctrl\_cx = 8'b00001001; // 9                  4'ha:    led\_ctrl\_cx = 8'b00010001; // A                  4'hb:    led\_ctrl\_cx = 8'b11000001; // B                  4'hc:    led\_ctrl\_cx = 8'b11100101; // C                  4'hd:    led\_ctrl\_cx = 8'b10000101; // D                  4'he:    led\_ctrl\_cx = 8'b01100001; // E                  4'hf:    led\_ctrl\_cx = 8'b01110001; // F                  default: led\_ctrl\_cx = 8'b11111111; // 不显示              endcase          end      end  （4）uart\_send  设计思路：三段式实现状态机，与实验4相同。  输入端口：clk、rst、valid、data  输出端口：dout  关键代码：      // \* 三段式实现状态机      // 第1个always块, 描述次态迁移到现态      always @(posedge clk or posedge rst) begin          if (rst) current\_state <= IDLE;          else current\_state <= next\_state;      end      // 第2个always块, 描述状态转移条件判断      always @(\*) begin          if (baud\_check) begin              if (start\_flag) next\_state = START;              else begin                  case (next\_state)                  IDLE: begin                      next\_state = IDLE;                      // 空闲状态 => 空闲状态                  end                  START: begin                      next\_state = DATA;                      // 起始状态 => 数据状态                  end                  DATA: begin                      if (bit\_index == 7) next\_state = STOP;  // 数据状态 => 停止状态                      else next\_state = DATA;                 // 否则继续发送数据位                  end                  STOP: begin                      next\_state = IDLE;                      // 停止状态 => 空闲状态                  end                  default: next\_state = IDLE;                 // 默认: 空闲                  endcase              end          end      end      // 逐位输出数据      always @(posedge clk or posedge rst) begin          if (rst) begin              bit\_index <= 3'b0;          end else if (valid) bit\_index =0;          else if (baud\_check) begin              if (current\_state == DATA)                  bit\_index <= bit\_index + 1;                 // 数据位计数器加一          end      end      // 第3个always块, 描述输出逻辑      always @(posedge clk or posedge rst) begin          if (rst) begin              dout <= 1'b1;          end else begin              case (current\_state)              IDLE:    dout <= 1'b1;                          // 空闲状态发送持续的高电平              START:   dout <= 1'b0;                          // 起始状态发送一位低电平              DATA:    dout <= data\_saved[bit\_index];         // 数据状态逐位发送数据              STOP:    dout <= 1'b1;                          // 停止状态发送一位高电平              default: dout <= 1'b1;                          // 默认: 空闲状态              endcase          end      end  （5）button\_ctrl  设计思路：进行按键消抖，与上升沿检测。  输入端口：clk、rst、button、switch  输出端口：valid、data  关键代码：      // button 按键去抖      always @(posedge clk) begin        debounce\_shift\_reg <= {debounce\_shift\_reg[14:0], button};      end      assign button\_status = (debounce\_shift\_reg == 16'hFFFF) ? 1'b1 : 1'b0;      // button 上升沿检测      always @ (posedge clk or posedge rst) begin          if(rst) sig\_r0 <= 1'b0;          else    sig\_r0 <= button\_status;      end      always @ (posedge clk or posedge rst) begin          if(rst) sig\_r1 <= 1'b0;          else    sig\_r1 <= sig\_r0;      end      always @ (posedge clk or posedge rst) begin          if(rst) sig\_r2 <= 1'b0;          else    sig\_r2 <= sig\_r1;      end  assign button\_edge\_detect = ~sig\_r2 & sig\_r0; |
| 调试报告 |
| 仿真波形截图及仿真分析  仿真波形截图：    如图，通过全部的十组仿真数据测试。  仿真分析：  如图，在每个数据测试中，接收模块在IDLE状态直接进入START状态，在这里经过半个波特率周期采样低电平信号din==0，接着进入DATA状态，每隔一个波特率周期采样一次，刚好在信号中段采样。经过8位数据的采样后，进入STOP状态，停留一个波特率周期采样到高电平信号din==1，同时拉高有效信号valid两个周期。重新进入IDLE状态，进入下一次数据的测试。 |
| 设计过程中遇到的问题及解决方法 |
| 现象：  原本正常通过测试，但是在修改了代码格式、添加注释与改动了由现态转移到次态的条件之后，第一个数据正常，第二个数据以后由于valid信号没有正常拉高，不能正常无法通过仿真测试。  分析过程：  正常通过测试表明基本逻辑没有问题，应当是转移条件有误。由于查看仿真波形图valid没有正常拉高，且第一个数据没有问题，首先猜测是进入STOP状态的条件有误，检查无果。  接着猜测进入STOP状态后没有正常进入IDLE状态，故意做出一些错误修改以作为调试，发现错误的修改对仿真结果没有影响，排除。  然后由于第一个信号正常，此时想到可能是DATA状态采样后没有将bit\_index复位，发现的确没有，但是仍然无法解决。  最后发现DATA数据在bit\_index==7的时候就进行了复位，导致数据一直不足8位因而无法进入STOP状态拉高valid，问题解决。  错误原因：   1. DATA状态采样后没有将bit\_index复位； 2. DATA数据在bit\_index==7的时候就进行了复位，导致数据一直不足8位因而无法进入STOP状态拉高valid。   //\* 三段式实现状态机      // 第1个always块，描述次态迁移到现态      always @(posedge clk or posedge rst) begin          if (rst) current\_state <= IDLE;          else     current\_state <= next\_state;      end      // 第2个always块，描述状态转移条件判断      always @(\*) begin          case (current\_state)              // 空闲状态: 接收到起始信号 din == 0 后进入起始状态              IDLE: begin                  if(din == 0) begin                      next\_state = START;                  end else next\_state = IDLE;              end              // 起始状态: 持续半个波特率, 后进入数据状态              // 保证之后采样间隔为一个波特率的同时, 在信号的中段采样              START: begin                  if(baud\_counter == BAUD\_HALF) begin                      next\_state = DATA;                  end else next\_state = START;              end              // 数据状态: 传入 8 位数据后进入停止状态              DATA: begin                  if (bit\_index == 8) begin                      next\_state = STOP;                  end else next\_state = DATA;              end              // 停止状态: 继续采样, 接受到停止信号 din == 1 后转入空闲状态              STOP: begin                  if(baud\_counter == BAUD\_END) begin                      next\_state = IDLE;                  end else next\_state = STOP;              end              default: next\_state = IDLE;          endcase      end      // 第3个always块，描述输出逻辑      always @(posedge clk or posedge rst) begin          if(rst) begin              data <= 0;              data\_saved <= 0;              valid <= 0;              baud\_counter <= 0;          end else begin              case(current\_state)                  // 空闲状态: 复位计数器与有效信号                  IDLE: begin                      valid <= 0;                      baud\_counter <= 0;                  end                  // 起始状态: 计数半个波特率, 完成后复位波特率计数器与位计数器                  START: begin                      if(baud\_counter == BAUD\_HALF) begin                          baud\_counter <= 0;                      end else begin                          baud\_counter <= baud\_counter + 1;                      end                  end                  // 数据状态: 以波特率为间隔采样, 逐位传入数据                  DATA: begin                      if(baud\_counter == BAUD\_END) begin                          data\_saved[bit\_index] <= din;                          baud\_counter <= 0;                          if(bit\_index == 7) begin                              baud\_counter <= 0;                          end else begin                              bit\_index <= bit\_index + 1;                          end                      end else begin                          baud\_counter <= baud\_counter + 1;                      end                  end                  // 停止状态: 继续采样, 接收到停止信号 din == 1 后将 valid 拉高, 并将暂存的数据赋值给输出端口                  STOP: begin                      if(baud\_counter == BAUD\_END) begin                          if(din == 1) begin                              baud\_counter <= 0;                              data <= data\_saved;                              valid <= 1;                          end else baud\_counter <= 0;                      end else baud\_counter = baud\_counter + 1;                  end                  default: begin                      valid <= 0;                      data <= 0;                  end              endcase          end  end  解决方案：   1. 在START状态复位bit\_index； 2. 将条件bit\_index==7改为bit\_index==8。 |
| 课程设计总结 |
| 实验共用12小时，期中8小时写代码，4小时写报告。  课程收获：  学会了使用Verilog来进行时序逻辑电路与组合逻辑电路的电路，理解了模块化设计电路的  总结：在写代码之前应该首先进行模块设计，保证思路清晰。  建议：实验开课时间和考试周重复了，重心一定会优先考试。 |