|  |  |
| --- | --- |
| logo[1] | 杭州电子科技大学 |

RISC-V设计报告

|  |  |  |
| --- | --- | --- |
| **实验课程名称** | **数字系统与处理器** | |
| **实验序号** | 9 | |
| **实验内容** | 简单32位RISC CPU设计 | |
| **班级** | 18184111 |  |
| **姓名** | 杨臻 |  |
| **学号** | 18052032 |  |
| **指导教师** | 黄继业 | |

二○一九 年 十二 月 十三 日

## 设计目的与要求

设计内容：由汇编语言控制的音乐播放器，可实现VGA显示歌曲名，数码管和LED显示当前音调

## 设计原理

使用UTSC-RVSoc为基础进行构建，加入了蜂鸣器驱动模块、LED驱动模块和数码管驱动模块。通过汇编语言向寄存器中写入单个音符数据，持续一段时间后变更为另一个音符数据。蜂鸣器、LED、数码管均实时读取蜂鸣器数据进行操作。

## Verilog设计内容

设计步骤：

1. 开启老师修改好的工程文件
2. 第一次测试
3. 新增蜂鸣器驱动代码（使用coe方式）
4. 第二次测试
5. 修改蜂鸣器驱动、LED驱动代码，删除coe，新增从寄存器读取数据模块
6. 编写汇编代码，将音调数据和持续时间写入寄存器
7. 第三次测试
8. 修改数码管驱动代码，使其从寄存器中读取并显示音调数据
9. 第四次测试
10. 完善汇编代码
11. 最终测试

关键源程序（含注释）：

soc\_top.sv新增部分：

assign rd\_data = bus\_slaves[2].rd\_data ; //原本为bus\_slaves[5]，

assign rd\_gnt = bus\_slaves[2].rd\_gnt ; //此处改为[2]，

assign wr\_req = bus\_slaves[2].wr\_req ; //为ram地址

assign wr\_addr = bus\_slaves[2].wr\_addr ; //0x00010

assign wr\_data = bus\_slaves[2].wr\_data ;

assign wr\_gnt = bus\_slaves[2].wr\_gnt ;

SPEAKER #(

.SPK\_CLK ( SPK\_CLK )

)spk\_ram(

.CLK0 ( clk ),

.RST0 ( rst\_n ),

.LED ( LED ),

.bus ( bus\_slaves[2]),

.H ( H ),

.SPK\_KX ( SPK\_KX )

);

SPEAKER.sv：

module SPEAKER #(

parameter SPK\_CLK = 1)

(

input logic CLK0,RST0,

output logic SPK\_KX,

output logic [3:0] LED,

output logic H,

naive\_bus.slave bus //接入bus总线

);

reg [7:0] CNT8;

logic [7:0] Q;

reg CLK,CLK1M;

wire [10:0] T;

reg [29:0] Q1,Q2;

wire SPK;

reg SPK\_KX1;

assign bus.rd\_gnt = bus.rd\_req;

assign bus.wr\_gnt = bus.wr\_req;

assign bus.rd\_data = Q; //将bus.rd\_data中的值传递给Q

assign SPK\_KX=SPK\_KX1;

always @(posedge CLK0 or negedge RST0) begin

if (!RST0)

begin

Q2<=0;

CLK1M<=1'b0;

end

else

begin

Q2<=Q2+1;

if (Q2==24)

begin

CLK1M=~CLK1M;

Q2<=0;

end

end

end

F\_CODE FC(

.INX(Q),

.CODE(LED),

.TO(T),

.H(H)

);

SPKER SP(

.CLK(CLK1M),

.RST(RST0),

.TN(T),

.SPKS(SPK)

);

always @(posedge SPK or negedge RST0)

begin

if (!RST0) SPK\_KX1<=0;

else

SPK\_KX1<=~SPK\_KX1;

end

endmodule

pout\_seg.v新增部分：

assign wdata = rd\_data; //令wdata的值始终等于0x00010地址中的值

seg.v：

module seg(

input clk,

input [31:0] seg\_data,

output reg [7:0] DIG,

output reg [7:0] SEL

);

reg [18:0] cnt;

wire [3:0] num;

reg clock\_25m;

always@(posedge clk)

begin

clock\_25m <= ~clock\_25m;

end

assign num = seg\_data[3:0]; //num始终等于0x00010地址中的值

always@(posedge clock\_25m)begin

if(cnt == 19'b111\_1111\_1111\_1111\_1111)

cnt <= 0;

else

cnt <= cnt + 1;

end

always@(posedge clock\_25m)begin

case(cnt[18:16])

3'b000:

begin SEL <= 8'b1111\_1110;end

3'b001:

begin SEL <= 8'b1111\_1101;end

3'b010:

begin SEL <= 8'b1111\_1011;end

3'b011:

begin SEL <= 8'b1111\_0111;end

3'b100:

begin SEL <= 8'b1110\_1111;end

3'b101:

begin SEL <= 8'b1101\_1111;end

3'b110:

begin SEL <= 8'b1011\_1111;end

3'b111:

begin SEL <= 8'b0111\_1111;end

endcase

end

always@(posedge clock\_25m)begin

case(num)

4'h0 : DIG <= 8'b11000000; //0

4'h1 : DIG <= 8'b11111001; //1

4'h2 : DIG <= 8'b10100100; //2

4'h3 : DIG <= 8'b10110000; //3

4'h4 : DIG <= 8'b10011001; //4

4'h5 : DIG <= 8'b10010010; //5

4'h6 : DIG <= 8'b10000010; //6

4'h7 : DIG <= 8'b11111000; //7

4'h8 : DIG <= 8'b10000000; //8

4'h9 : DIG <= 8'b10010000; //9

4'hA : DIG <= 8'b10001000; //a

4'hB : DIG <= 8'b10000011; //b

4'hC : DIG <= 8'b11000110; //c

4'hD : DIG <= 8'b10100001; //d

4'hE : DIG <= 8'b10000110; //e

4'hF : DIG <= 8'b10001110; //f

default : DIG <= 8'b11000000; //0

endcase

end

endmodule

汇编代码：

.org 0x0

.global \_start

\_start:

or t3, zero,zero # t3 清零

print\_vga:

# 第一步：令t0寄存器=0x00020000，即显示RAM的地址

or t0, zero,zero # t0 清零

lui t0, 0x00020 # t0 寄存器的高20bit=0x00020

# 第二步：将Opening逐个字符写入user\_uart外设，即打印Opening到uart

ori t1, zero, 0x04F # t1='O'的ASCII码

sb t1, (t0) # t1写入t0地址

addi t0, t0 , 0x001 # t0+1

ori t1, zero, 0x070 # t1='p'的ASCII码

sb t1, (t0) # t1写入t0地址

addi t0, t0 , 0x001 # t0+1

ori t1, zero, 0x065 # t1='e'的ASCII码

sb t1, (t0) # t1写入t0地址

addi t0, t0 , 0x001 # t0+1

ori t1, zero, 0x06e # t1='n'的ASCII码

sb t1, (t0) # t1写入t0地址

addi t0, t0 , 0x001 # t0+1

ori t1, zero, 0x069 # t1='i'的ASCII码

sb t1, (t0) # t1写入t0地址

addi t0, t0 , 0x001 # t0+1

ori t1, zero, 0x06e # t1='n'的ASCII码

sb t1, (t0) # t1写入t0地址

addi t0, t0 , 0x001 # t0+1

ori t1, zero, 0x067 # t1='g'的ASCII码

sb t1, (t0) # t1写入t0地址

print\_hello:

# 第一步：令t0寄存器=0x00010000，即数据RAM的地址

or t0, zero,zero # t0 清零

lui t0, 0x00010 # t0 指向内存地址0x00010

# 第二步：将逐个音符写入Data\_RAM

ori t1, zero, 2 # t1=2

sb t1, (t0) # t1写入t0地址

lui t2, 0x00300 # 令t2初值为0x00300

loop1: # 通过空循环实现音符延迟

addi t2, t2, -1

bne t2, zero, loop1

ori t1, zero, 0 # t1=0

sb t1, (t0) # t1写入t0地址，覆盖原音符数据，即寄存器中只放一个音符数据

lui t2, 0x00300

loop2:

addi t2, t2, -1

bne t2, zero, loop2

ori t1, zero, 2 # t1=2

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop3:

addi t2, t2, -1

bne t2, zero, loop3

ori t1, zero, 4 # t1=4

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop4:

addi t2, t2, -1

bne t2, zero, loop4

ori t1, zero, 5 # t1=5

sb t1, (t0) # t1写入t0地址

lui t2, 0x00200

loop5:

addi t2, t2, -1

bne t2, zero, loop5

ori t1, zero, 0 # t1=5

sb t1, (t0) # t1写入t0地址

lui t2, 0x000ff

loop51:

addi t2, t2, -1

bne t2, zero, loop51

ori t1, zero, 5 # t1=5

sb t1, (t0) # t1写入t0地址

lui t2, 0x00200

loop52:

addi t2, t2, -1

bne t2, zero, loop52

ori t1, zero, 4 # t1=4

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop6:

addi t2, t2, -1

bne t2, zero, loop6

ori t1, zero, 2 # t1=2

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop7:

addi t2, t2, -1

bne t2, zero, loop7

ori t1, zero, 5 # t1=5

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop8:

addi t2, t2, -1

bne t2, zero, loop8

ori t1, zero, 1 # t1=1

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop9:

addi t2, t2, -1

bne t2, zero, loop9

ori t1, zero, 5 # t1=5

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop10:

addi t2, t2, -1

bne t2, zero, loop10

ori t1, zero, 4 # t1=4

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop11:

addi t2, t2, -1

bne t2, zero, loop11

ori t1, zero, 5 # t1=5

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop12:

addi t2, t2, -1

bne t2, zero, loop12

ori t1, zero, 1 # t1=1

sb t1, (t0) # t1写入t0地址

lui t2, 0x00400

loop13:

addi t2, t2, -1

bne t2, zero, loop13

ori t1, zero, 0 # t1=1

sb t1, (t0) # t1写入t0地址

lui t2, 0x00f00

loop14:

addi t2, t2, -1

bne t2, zero, loop14

jal zero, print\_hello # 大循环结束，跳到print\_hello，重复打印

设计框图（可用Visio绘制）：

详细描述：

REV1.0：

将之前的音乐播放器代码移植入RISCV中，放置于soc\_top之下，使用IP核调用coe文件实现音乐播放。修改汇编文件使屏幕显示“Opening”（开场）。

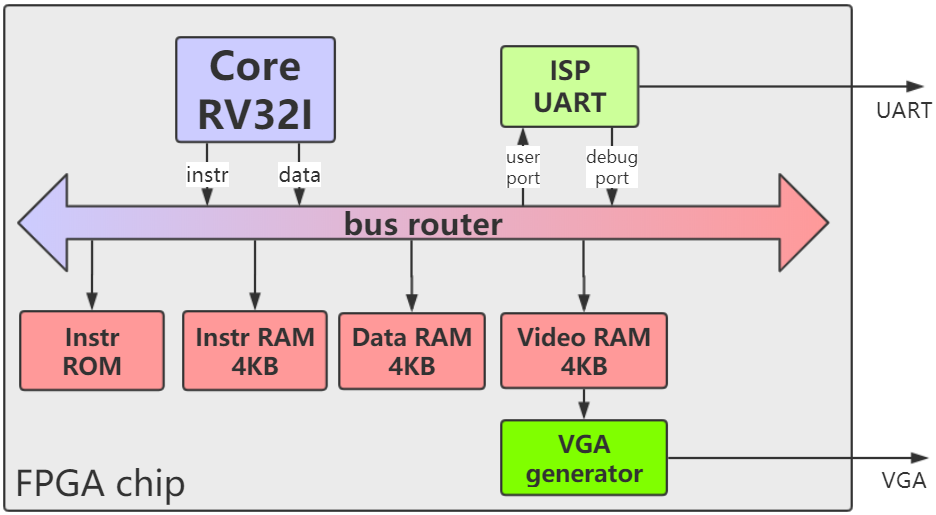
REV2.0：

从代码中了解到数据从汇编语言写入到用户UART中，即内存地址0x00030，bus总线slaves[5]。因此尝试让SPEAKER模块读取slaves[5]中的内容。创建bit文件时出错，提示数组越界。

REV2.1：

从原作者的Github项目中了解到用户UART中保存的是需要输出到串口的内容，汇编样例代码中负责串口输出所用的地址也是slaves[5]。随即放弃slaves[5]，改用数据RAM，即slaves[2]进行写入。成功使蜂鸣器发声、LED灯闪烁。





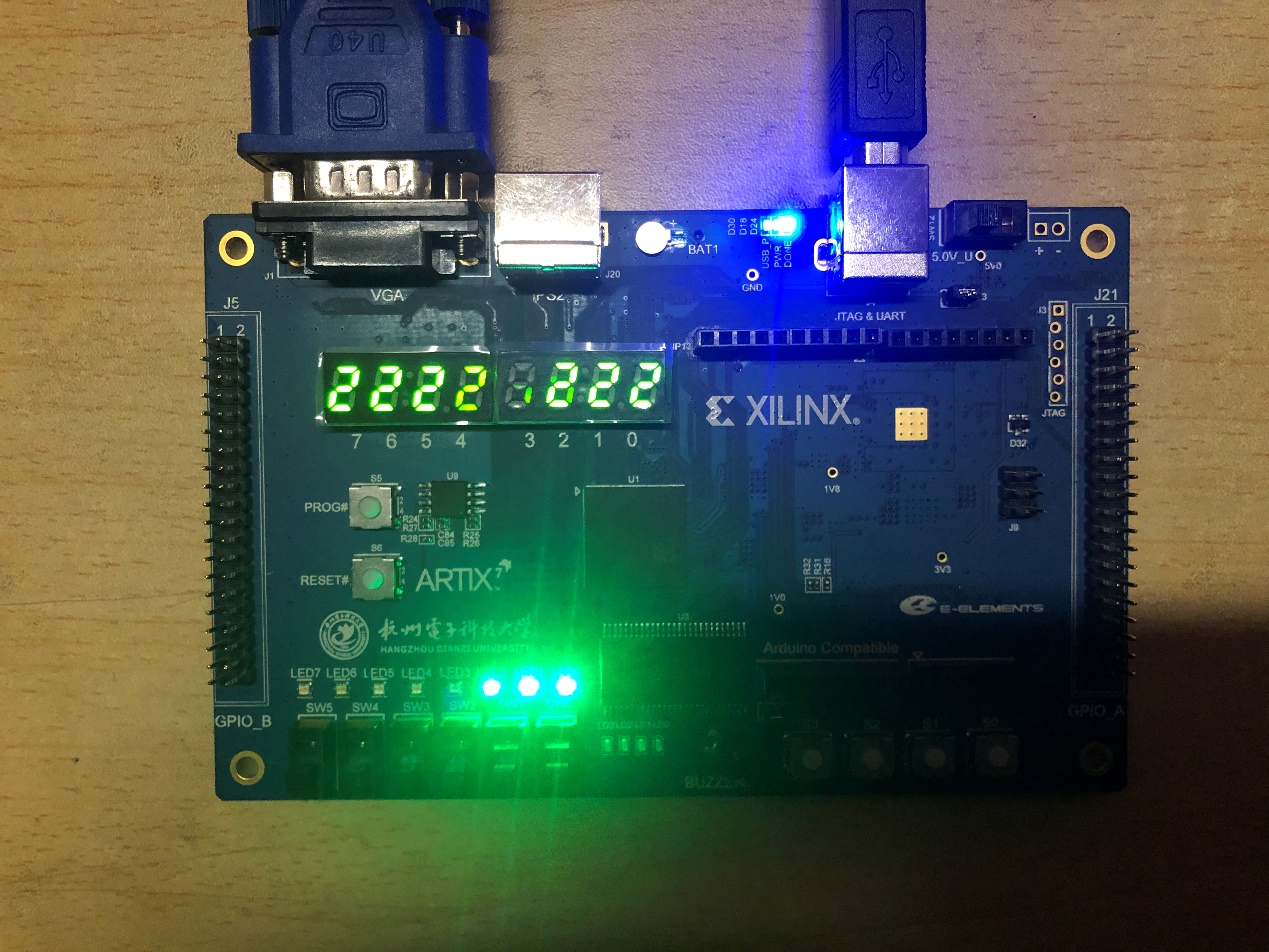
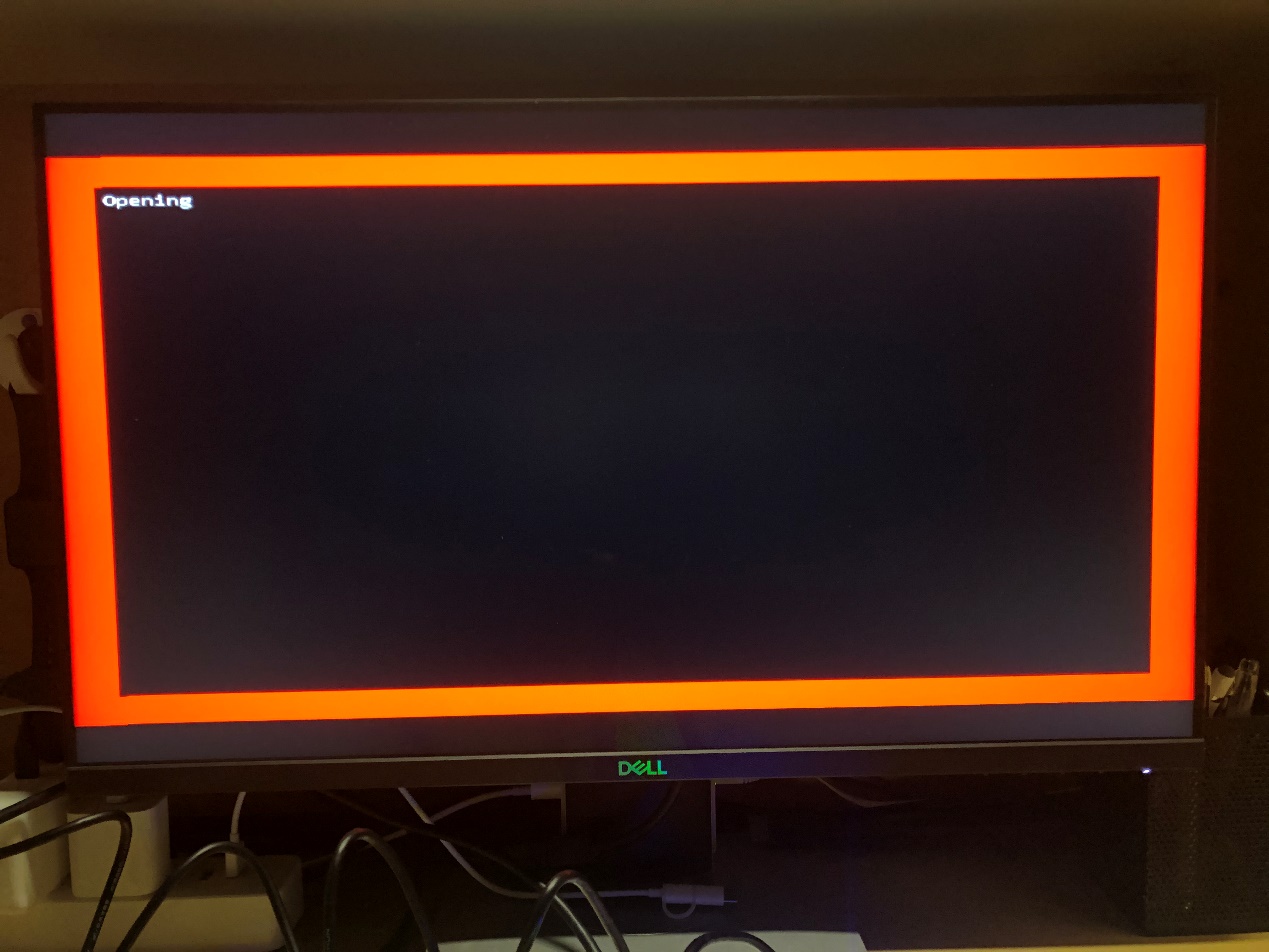
REV2.2：

完善汇编代码，使得音乐更加完整流畅。

REV3.0：

修改pout\_seg.v中的内容。操作时发现所给的pout\_seg.v为Verilog代码，并非SystemVerilog，不支持直接将bus总线引入，观察代码后发现已在soc\_top中将slaves[5]的数据单独取出，并传递至pout\_seg中，但pout\_seg中没有使用这些数据。因此将slaves[5]改成slaves[2]，并令其中的数字显示控制变量wdata始终等于slaves[2].rd\_data。再修改seg.v中的一些代码，使得数码管显示一排一样的数字。

硬件演示照片：

## 总结与体会

这次大作业完成的时候经历了许多困难，也接触了很多之前没接触过的功能。花费了将近两天才完成开发，可以算是耗时最长的项目了，不过几乎用上了开发板上所有的输出部件，包括VGA、LED、数码管、蜂鸣器，并且完全通过汇编来控制。总的来说还是比较成功的。