浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	颜晗
学　　院：	计算机科学与技术学院
专　　业：	计算机
指导教师：	
报告日期：	2022　年 6 月 1　日

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Lab04-0 CPU 核集成设计

学生姓名：3200105515 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

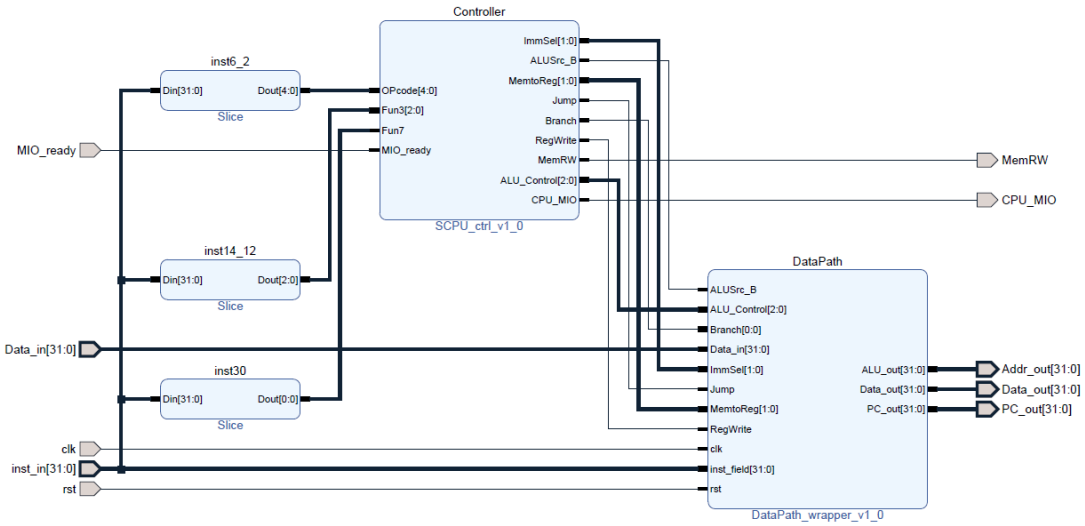
1. 本次实验使用老师所给的来两个模块重新构建 cpu 并形成完整的测试电路，得到和 Lab2 相同的结果

2. 根据原理图，我们可以看出原本整体的 cpu 被分为两个部分，一部分产生控制信号，另一部分包括了除内存外的其余部件，完成实际的计算。

对于控制模块来说，我们需要指令的 opcode、funt3、funt7 信号进行译码，opcode 可以区分指令的类型——R 型、S 型、I 型等，对于选择、跳转控制信号，同一类型的指令都是相同的，而对于运算的选择，我们还需要 funt3、funt7 信号来区分。

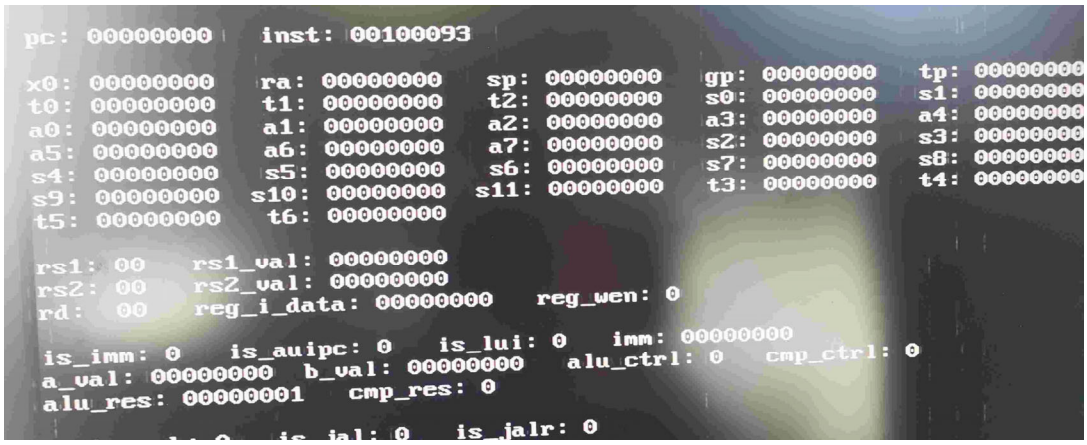
对于 Datapath，即利用得到的控制信号控制寄存器乃至内存的读写，以及决定下一步执行的指令为哪一条。

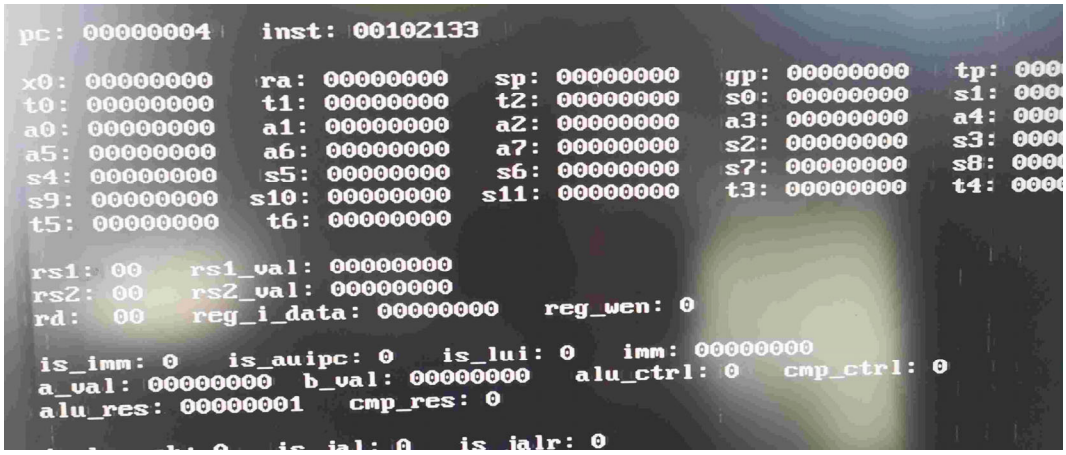
本次实验原理图相当简单，利用 verilog 实现即可。

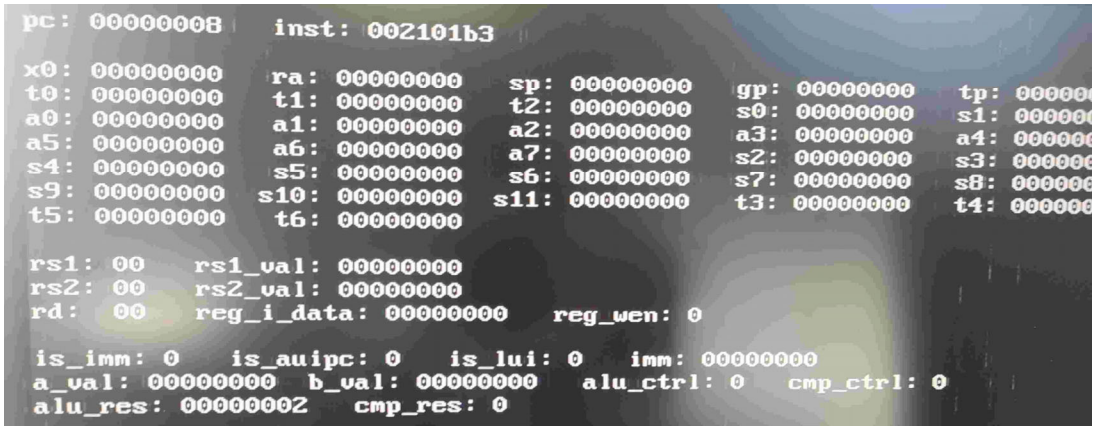


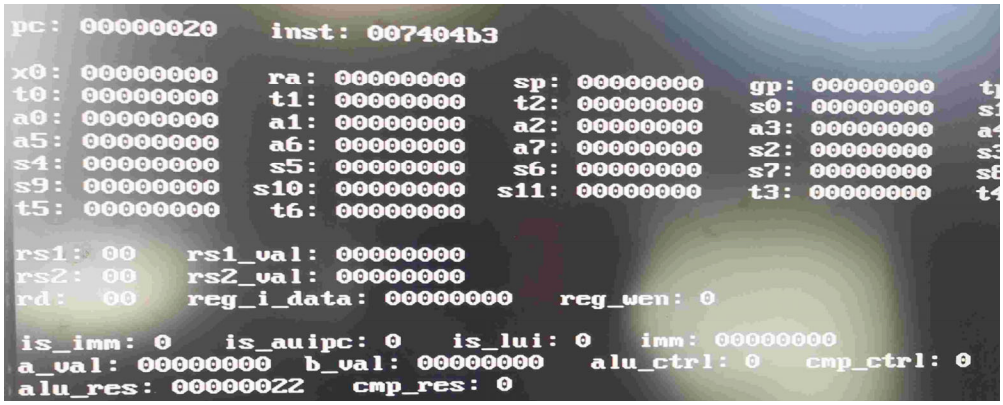
二、实验结果与分析

下板，斐波那契验证：


pc: 00000000 inst: 00100093
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000 tp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000 s1: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000 a4: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000 s3: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000 s8: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000 t4: 00000000
t5: 00000000 t6: 00000000
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0
is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0


pc: 00000004 inst: 00102133
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000
t5: 00000000 t6: 00000000
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0
is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000001 cmp_res: 0


pc: 00000008 inst: 002101b3
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000
t5: 00000000 t6: 00000000
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0
is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000002 cmp_res: 0


pc: 00000020 inst: 007404b3
x0: 00000000 ra: 00000000 sp: 00000000 gp: 00000000
t0: 00000000 t1: 00000000 t2: 00000000 s0: 00000000
a0: 00000000 a1: 00000000 a2: 00000000 a3: 00000000
a5: 00000000 a6: 00000000 a7: 00000000 s2: 00000000
s4: 00000000 s5: 00000000 s6: 00000000 s7: 00000000
s9: 00000000 s10: 00000000 s11: 00000000 t3: 00000000
t5: 00000000 t6: 00000000
rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000 reg_wen: 0
is_imm: 0 is_auipc: 0 is_lui: 0 imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 0 cmp_ctrl: 0
alu_res: 00000022 cmp_res: 0

斐波那契数列：1，1，2，3，5，8，13，21，34
对于前三个即 1，1，2，pc 等于 20 时，运行到第 9 个 34，即十六进制 22。

三、讨论与心得

本次实验简单了解 cpu 的分模块组成，为之后的实验打下基础。

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Lab04-1 CPU 设计之数据通路

学生姓名：3200105515 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

本次实验我们需要自己实现 Datapath 并替换老师给定的模块。根据老师给定的原理图，整个 Datapath 可分为两部分——跳转部分与计算部分。

跳转部分主要针对目前实现的 beq 与 jal 指令。首先是 PC 寄存器，利用一个触发器接受输入的下一个 PC 地址并输出当前 PC 地址。而当前 PC 地址将分别与 4 和产生的立即数进行加法运算并接连进入两个 mux 中，两个 mux 分别用于判断 beq 指令与 jal 指令是否跳转或存在（beq 需要产生的 branch 信号与 ALU 减法计算结果 zero 是否为零相与判断是否跳转，而 jal 是无条件跳转语句），跳转的话，PC 值加立即数便会被选择出成为下一时钟周期 PC 值。

计算部分即利用寄存器的值或立即数通过 ALU 计算出即将存入寄存器中的值或内存地址。立即数产生器目前只有 4 种情况，对应 I、S、B、J 型四种指令立即数，利用{}连接符将指令中属于立即数的部分进行拼接即可，注意 jal 和 beq 指令立即数对应的是指令地址，最低位默认为 0。ALU 模块和 regfile 之前已有实现。Regfile 的输入通过 memtoreg 信号与一个四选一（三选一）mux 来选择，可以来自 PC+4，内存数据以及 ALU 结果。

原理图非常详细且本次实验并无相关拓展，只需照原理图进行 verilog 代码编写即可。

二、实验结果与分析

参见 Lab4-3.

三、讨论与心得

本次实验参照原理图并不难。对于实验中的部分设计，可能是为了后面的拓展或系统稳定性，老师的部分设计比较冗余，比如 PC 值的跳转只有 PC+4 和 PC+imm 两种，其实只需要一个模块，beq 和 jal 信号完全可以通过或门合并起来，个人认为相比可能更加简洁点。

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Lab04-2 CPU 设计之控制器

学生姓名：3200105515 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

控制器产生 cpu 需要的各种信号：包括：

1. ImmSel，立即数选择信号，当前只需 2 位四种信号
2. ALUSrc_B，ALU 第二个操作数的选择，ALU 第二个操作数有两个来源，一个是 2 号源寄存器，另一个是立即数分别对应 R 型指令与 I 型的计算指令。
3. ALUOP、ALU_control，ALUOP 为根据指令类型决定的信号，同种类型指令根据 fun3 与 fun7 还会有不同的运算。通过多级译码使代码更具层次感。
4. RegWrite，决定是否向寄存器组写入数据，像 R 型与 I 型都计算完后要向寄存器写回计算得到的数据的。
5. MemRW，内存的使用方法，0 为读，1 为写。
6. Branch，jump，控制 PC 的跳转，注意 jal 产生的 Jump 信号是无条件跳转，而 branch 只是一个跳转的可能，是否跳转还需要在 DataPtath 中进行一些计算才能最终决定。

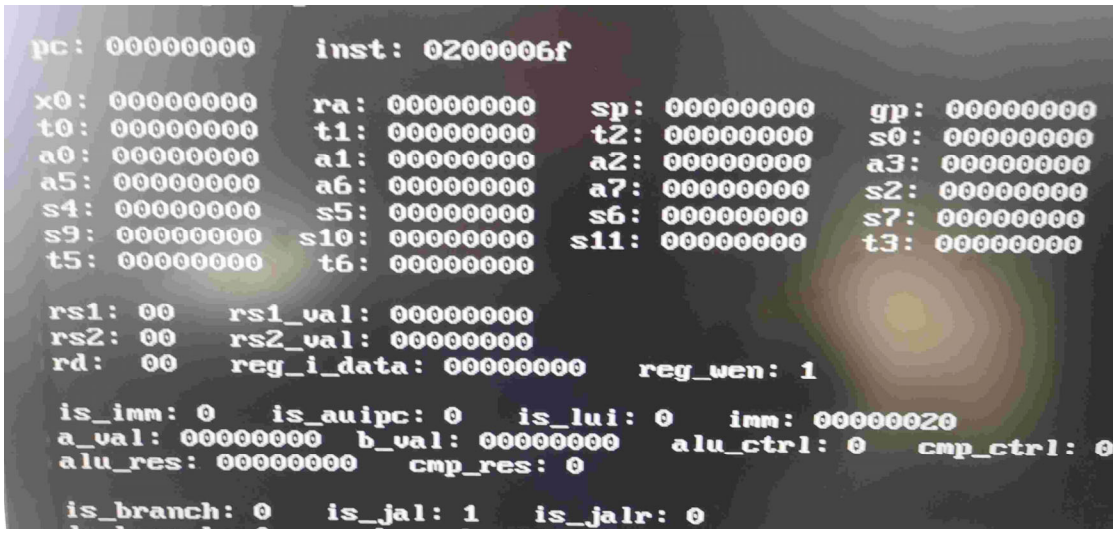
知道了各种信号的功能和各种类型指令的功能，我们即可写出对应代码，使用 case 语句或 If-else 语句都可以分多种情况对 reg 进行赋值。部分代码如下：

```
always @* begin
        case(OPcode)
        5'b01100:begin
        //R-type add
            ALUSrc_B = 1'b0;
            MemtoReg = 2'b00;
            MemRW    = 1'b0;
            RegWrite = 1'b1;
            Branch   = 1'b0;
            Jump     = 1'b0;
            ALUop    = 2'b10;
            ImmSel   = 2'b00;
            End
//R 型指令的信号一览
```

```
always @* begin
   case(ALUop)
     2'b00:ALU_Control=3'h2;//lw,sw
     2'b01:ALU_Control=3'h6;//beq
     2'b10://R 型指令
         case(Fun)
             4'b0000:ALU_Control=3'h2;
             4'b0001:ALU_Control=3'h6;
             4'b1110:ALU_Control=3'h0;
             4'b1100:ALU_Control=3'h1;
             4'b0100:ALU_Control=3'h7;
             4'b1010:ALU_Control=3'h5;
             4'b1000:ALU_Control=3'h3;
             default: ALU_Control=4;
```

二、实验结果与分析

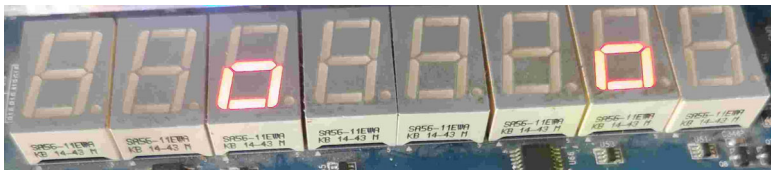
1. 下板验证

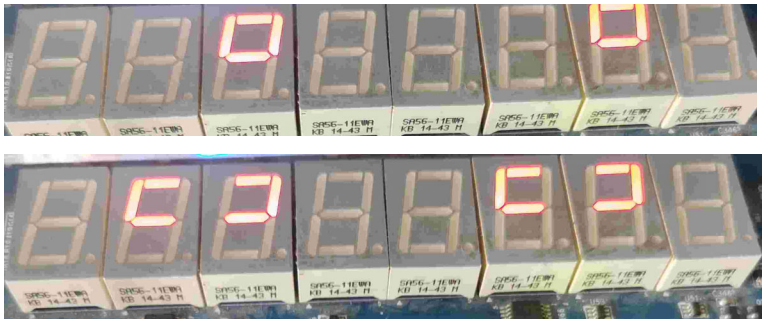


第一条指令 0200006f ，即 jal x0，32，reg_wen 为 1 是由于要向寄存器写入 PC+4，但由于是 x0 寄存器，写入无效，而 is_jal 显示出确实判断出了跳转指令。

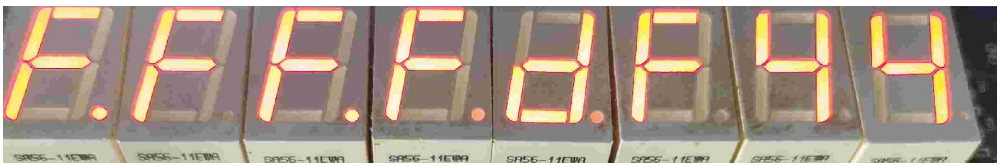
开机动画：

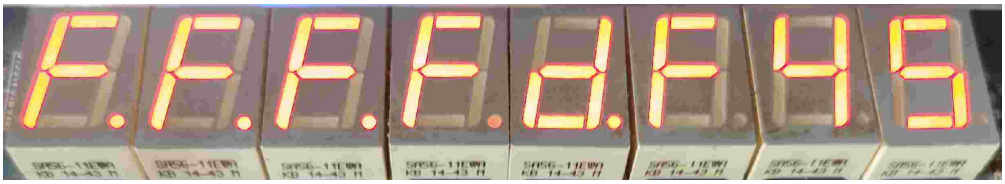
矩形动画：



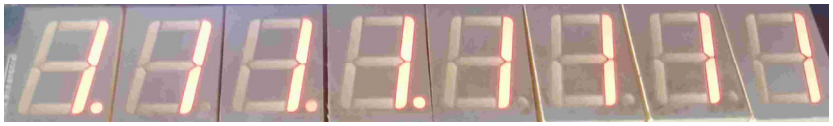


计数器：





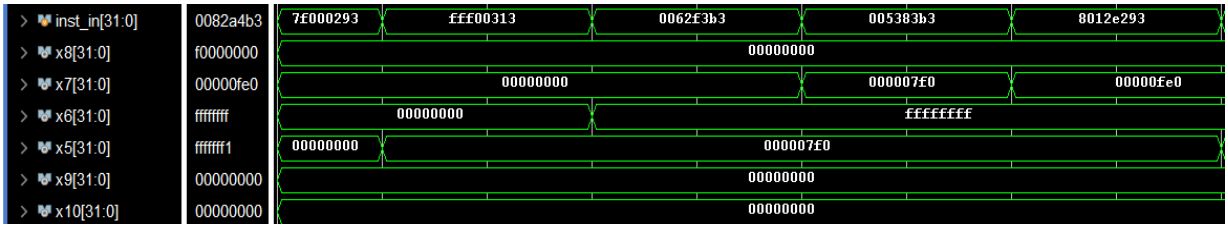
学号 3200105515：



2. 仿真验证

addi x5,x0,0x7f0	7F000293,
addi x6,x0,-1	FFF00313,
and x7,x5,x6	0062F3B3,
add x7,x7,x5	005383B3,
ori x5,x5,-2047	8012E293,
lw x8,(0)x0	00002403,
slt x9,x5,x8	0082A4B3,
xor x9,x5,x8	0082C4B3,
srli x5,x5,2	0022D293,
srl x7,x7,x5	0053D3B3,
sub x7,x7,x5	405383B3,

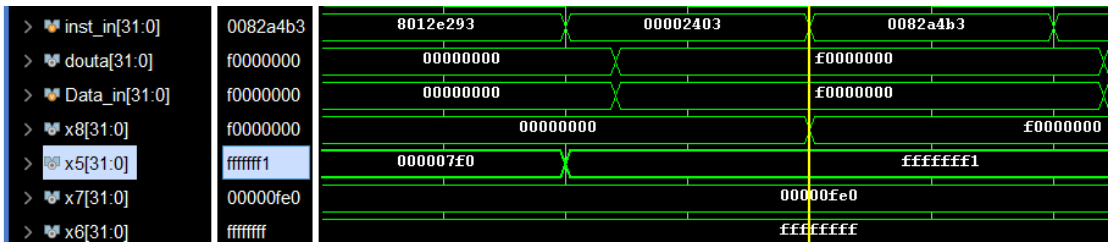
slti x9,x9,0
beq x9,x0,out
loop:
add x0,x0,x0
xor x0,x0,x0
addi x0,x0,2
jal x0,loop
out:
sw x8,(8)x0
lw x10,(8)x0
xori x7,x7,0x10d
or x4,x8,x5
jal x1,loop
	0004A493,
	00048A63,
	00000033,
	00004033,
	00200013,
	FF5FF06F,
	00802423,
	00802503,
	10D3C393,
	00546233,
	FE1FF0EF;



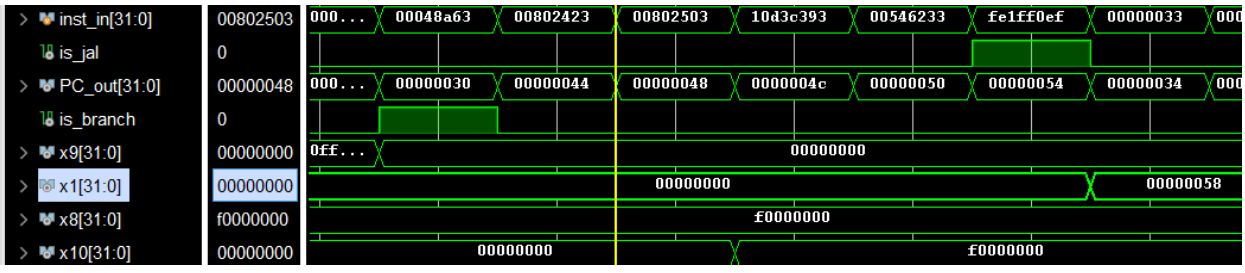
这一部分主要是 R 型以及 I 型指令的计算仿真，我们可以根据寄存器的值进行验证，

第一条 addi x5,x0,0x7f0 将寄存器 x5 的值置为 0+7f0 = 000007f0

第五条 ori x5,x5,-2047 将 x5 的值与 -2047 异或，最终结果为 fffffff1，由于改变延迟一个周期，可以在下一张图中看到



计算指令不再一一介绍，对于该图第二条指令 00002403 为一条 lw 指令，将第一个数据 f0000000 存入 x8 中，从信号可以看出，在指令执行中数据被顺利取出并在上升沿写入寄存器中。



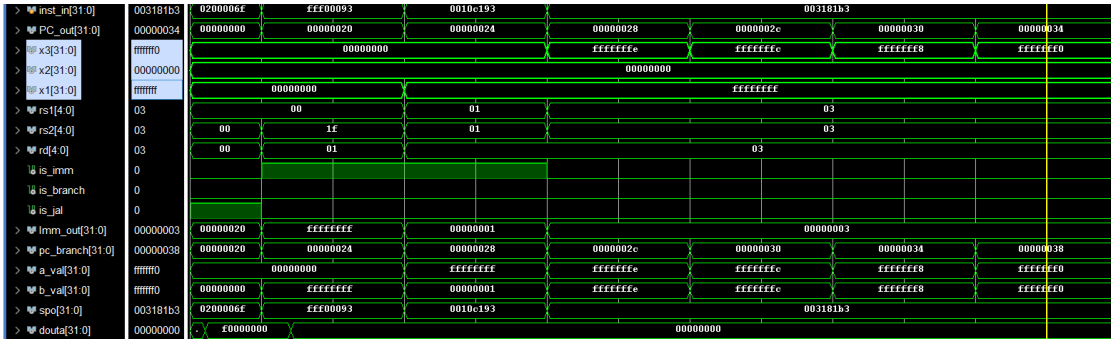
该图中首先是跳转指令

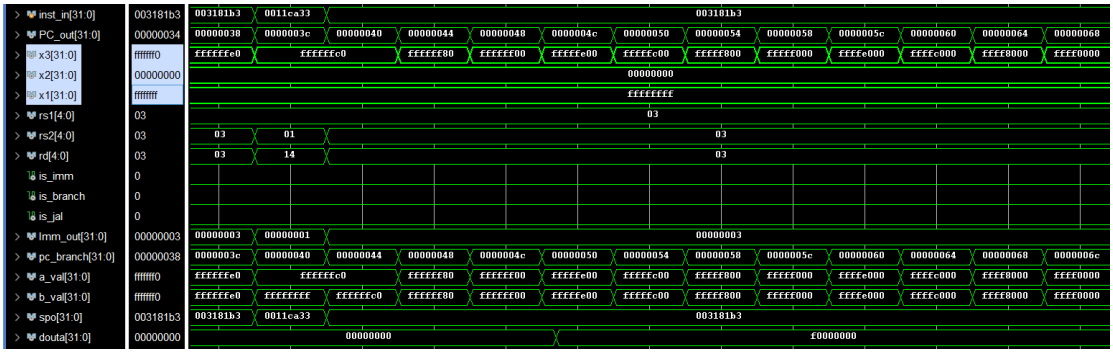
第一条 00048a63 beq x9,x0,out （地址 30）可以看出 x9=x0=0，因此下一条指令跳转至了 44；

倒数第二条 fe1ff0ef jal x1,loop 从 54 跳至 34 且寄存器 x1 的值被更新为 58.

其次是 store 指令，00802423　sw x8,8(x0)将 f0000000 存入地址 8 中，然后我们利用 00802503 (lw x10,8(x0))取出放置在 x10 中，可以看出数据确实被放进了对应位置。

助教代码





三、讨论与心得

Lab4-12 主要是实现课堂内讲解过的 cpu 结构，注意不同信号间长度对应、仔细连线基本无问题。指令虽然多但是 R 型和 I 型计算的指令除了 ALU 计算不同外，其余基本相同，且老师 ppt 很完善了，问题不大。

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Lab04-3 CPU 设计之指令扩展

学生姓名：3200105515 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

本次实验我们要求实现基本所有指令，自行设计并添加信号或模块。

1. 对于 R 型以及 I 型计算指令

   由于计算的种类增多了，需要四位的 ALU 控制信号才能对 alu 操作进行控制，对于 R 型依旧是通过 Fun3 与 fun7 进行选择即可，而对于 I 型指令，注意三个移位运算符的特殊性，特别是右移包括逻辑和算数，需要额外进行区分。

```
3'b001:ALU_Control=4'he;//slli
3'b101:
    case(Fun7)
        1'b0:ALU_Control = 4'hd;//srli
        1'b1:ALU_Control = 4'hf;//srai
    endcase
```

2. Branch 指令

   所有的 branch 指令都被添加进来了，可将 branch 信号拓展至六位对应六条指令，一条指令对应一位，置 1 说明目前为该种指令，同时可看出六条指令可两两成对使用一种运算的结果（sub、slt、sltu），将 ALU 运算的第一位连接与 branch 信号相与可以判断是否真的跳转。

```
//拓展
assign and_2_res = |(Branch[5:0] & {{~ALU_out[0]},ALU_out[0],{~ALU_out[0]},ALU_out[0],{~zero},zero});
```

3. Jalr 指令

   该指令也是无条件跳转指令，但是是一条 I 型指令，需要 rs1+imm 的结果作为跳转地址。同 branch 类似，我们将 jump 信号拓展至两位，junp[1]信号来判断 jalr 信号的存在。另外将 alu 结果引出一条线到 PCjump 判断的 mux 中（mux 拓展至 3 选 1），该指令的添加基本完成。

```
assign mux2T1_o3 = Jump[1] ?ALU_out :(Jump[0]? add_c1 : mux2T1_o1);//is_jump? 1
```

4. Load 与 store

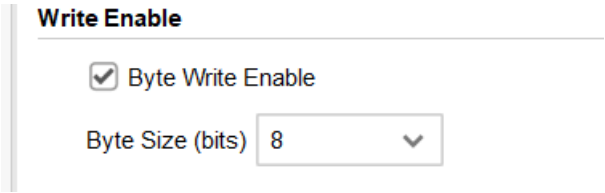
   这两种指令与内存有关，alu 的计算即加法计算地址，其余主要是对数据存储位数的判

断。两者都是通过 fun3 来进行指令判断的，首先我们要引出 fun3 信号（直接从指令截取）。

对于 load 类型较好操作，只需通过 fun3 来对从内存输入的数据进行截取和（无）符号拓展即可。

```
case(Ltype)
            3'b000:begin
                Data_reg = {{24{Data_in[7]}},Data_in[7:0]};
            end
            3'b001:begin
                Data_reg = {{16{Data_in[15]}},Data_in[15:0]};
            end
            3'b010:begin
                Data_reg = Data_in;
            end
            3'b100:begin
                Data_reg = {24'h000000,Data_in[7:0]};
            end
            3'b101:begin
                Data_reg = {16'h0000,Data_in[15:0]};
            end
        endcase
```

而对于 store 指令，我们首先需要修改 ram 的配置，使其能按字节写。



而后我们跳到 top 层模块，通过 vga 信号的输出可以发现我们计算的地址都会被其余模块统一为 4 的倍数输入到 ram 获取数据，而数据内存的 wea 允许我们对对应位进行写入。我们输入的数据都是从低位开始的，因此想要写数据我们还需要对我们的数据进行移位使其真的能在对应位置被写入。实际即用我们计算出的地址对 4 除余（取低 2 位即可）再向左移结果的 8 倍。通过 fun3 判断 wea 使能的位置即可。

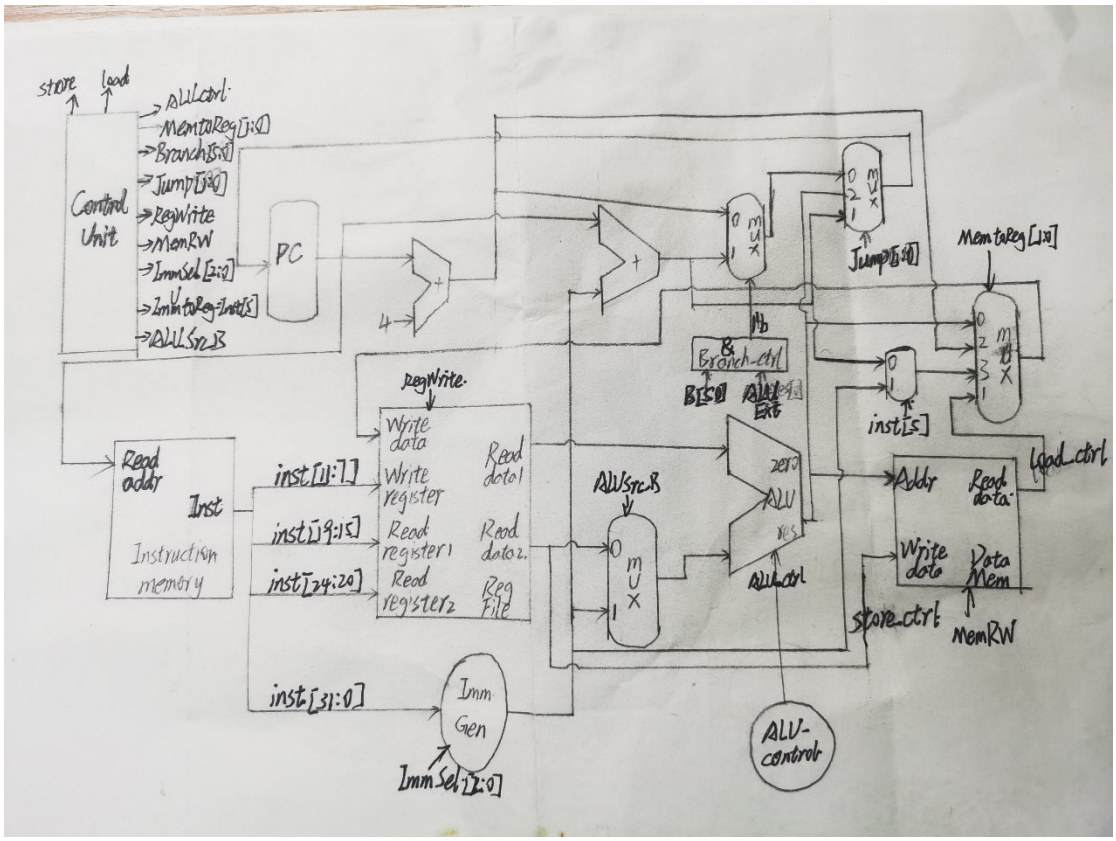
```
wire [1:0] pos = source_addr[1:0];
    always @* begin
        data_out = data_in << (pos*8);
        case(data_ram_we)
            1'b0: wea = 4'b0000;
            1'b1:
                case(Stype)
                    3'b000:
                    case(pos)
                        2'b00: wea = 4'b0001;
                        2'b01: wea = 4'b0010;
                        2'b10: wea = 4'b0100;
```

```
                    2'b11: wea = 4'b1000;
                endcase
                3'b001:
                case(pos)
                    2'b00: wea = 4'b0011;
                    2'b01: wea = 4'b0110;
                    2'b10: wea = 4'b1100;
                endcase
                3'b010:wea = 4'b1111;
            endcase
        endcase
    end
```

5. U 型指令

   还增加了两条 U 型指令 lui 与 auipc，首先是新增一种立即数，我们要拓展立即数选择信号，然后它们都是要向寄存器组写入的，而且写入的数据还有所不同因此我们需要对选择写入寄存器的 mux 进行信号增加，而由于两种指令在 fun7 是不同的，我们可以将通过 2 选 1mux 选出对应的值（纯立即数或者 PC+imm），再将该信号传入原本四选一 mux 空出的一个信号位，就不要修改所有指令的 MemtoReg 信号了。

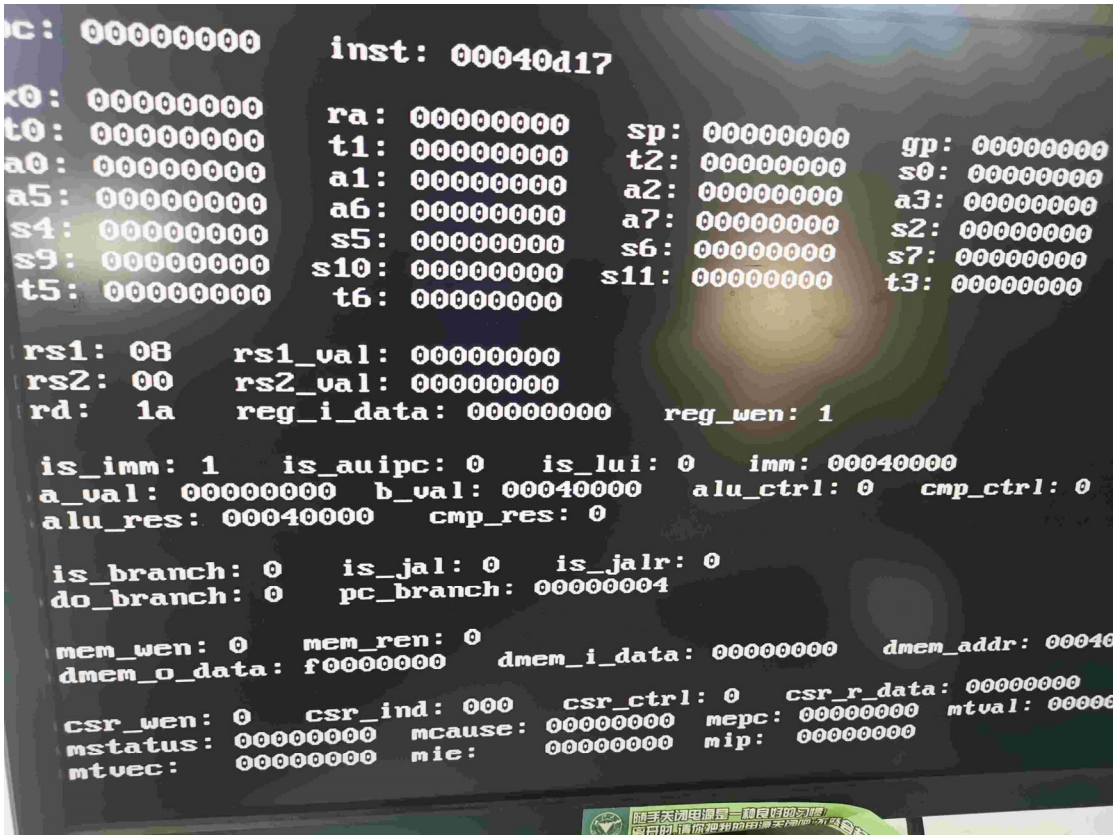
6. 实现设计图



二、实验结果与分析

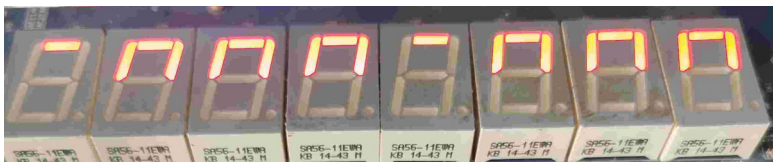
1. 下板验证

第一条指令：

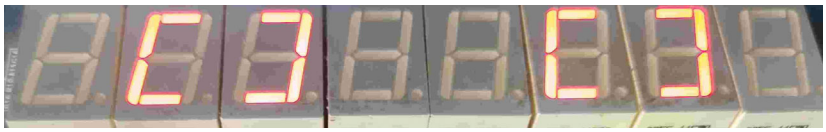


第一条指令 00040d17　　auipc x26, 0x40,将 PC 高五位加上 0x00040 后写入 x26 寄存器。

开机动画正常



矩形动画：

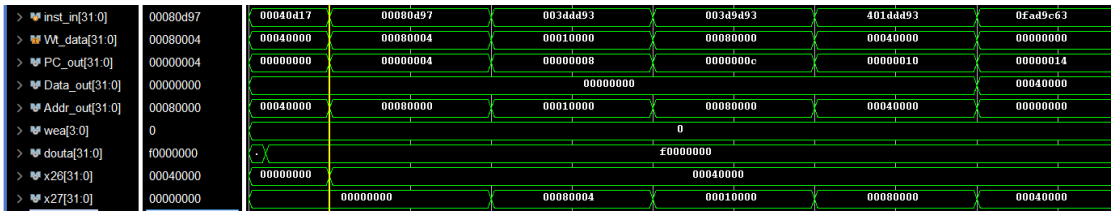


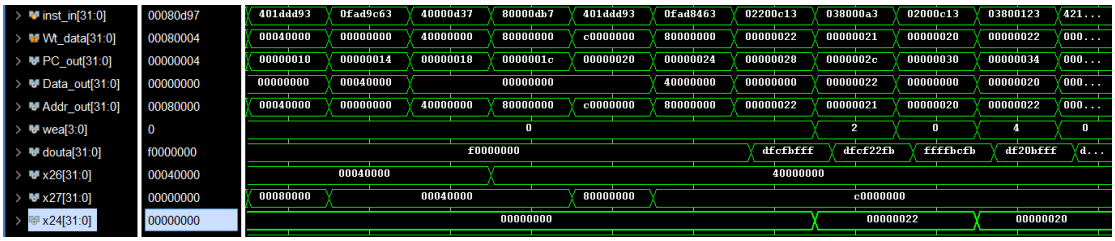
学号（前两组数据有改变 通过两次 sb 指令改变了 33333333 ，通过 sh 指令改变了 22222222 ）

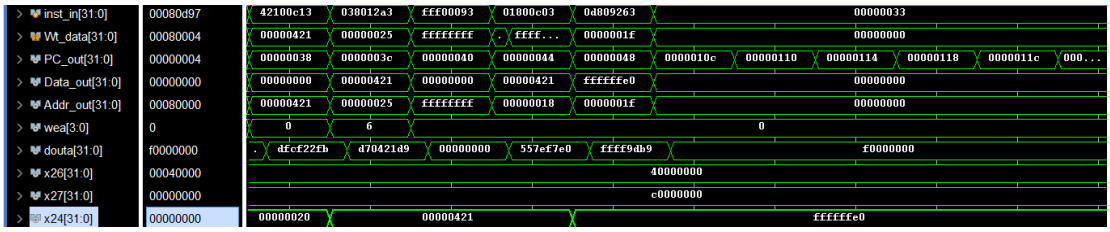
计数器

仿真验证

助教代码







自己代码
无……

三、讨论与心得

本次实验在思路上花了很久，因此实现方面大问题基本没有，主要是对于算数移位最初不太了解如何实现，查找资料后发现还要“显式”声明为符号数。另外对于 branch 指令在拓展使用时不太仔细，一方面对于 ALU 的运算结果没有截取，导致信号位数不匹配，另外复制粘贴过程也遗漏了部分“非”逻辑运算，导致 branch 无法正常执行。

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Lab04-4 CPU 设计之中断

学生姓名：3200105515 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

本次实验我们将实现对 cpu 中断的处理，主要是三种情况，ecall 指令、非法指令、外部中断。ecall 指令为 riscV 本身的一条指令可用于系统调用，跳转到对应区域处理；而遇到非法指令（即不存在的 opcode 之类的）同样需要进行处理；而此次实验中，我们还会通过一个 swich 开关来进行外部中断。当我们触发中断时，程序跳转到中断处理程序进行处理并在结束后跳转回正常程序中继续运行下去。

Ecall 指令是 riscV 内置的一条指令，因此我们需要在控制模块添加新的 opcode 处理模块使其遇到 ecall 能产生正确信号。另外对于非法指令我们需要完善控制模块的 default 或 else 的部分使其遇到未知指令时进入非法处理的部分。而中断部分我们通过为 CPU 添加一个外部输入的中断 INT 信号使其能影响 CPU 的执行。而对于所有的中断处理程序，我们还需要额外的 mret 指令来使其返回到正常的程序运行中去，即利用原本存储的 PC 值恢复 CPU 的正常执行指令。

```
5'b11100:begin//ecall, mret
            ALUSrc_B = 1'b1;
            MemtoReg = 2'b00;
            MemRW    = 1'b0;
            RegWrite = 1'b0;
            Branch   = 6'b000000;
            Jump     = 2'b00;
            ALUop    = 2'b11;
            ImmSel   = 3'b000;
            case(inst29)
                1'b0:begin
                    ecall = 1;
                    mret = 0;
                end
                1'b1:begin
                    ecall = 0;
                    mret = 1;
```

```
        default:begin //illegal
            ALUSrc_B = 1'b1;
            MemtoReg = 2'b00;
            MemRW    = 1'b0;
            RegWrite = 1'b0;
            Branch   = 6'b000000;
            Jump     = 2'b00;
            ALUop    = 2'b11;
            ImmSel   = 3'b000;
            ecall    = 0;
            mret     = 0;
            ill_instr= 1;
            end
        endcase
    end
```

```
            end
        endcase
        ill_instr= 0;
    end
```

而对于中断处理我们还需要额外的模块来控制 PC 值，也即根据输入的中断信号来决定下一周期 PC 值。由于我们实现的中断较为简单，模块中的寄存器有重要意义的仅一个，即 mepc，用于存储我们返回正常程序的 PC 值，而下一周期 PC(pc 被设为中断服务的地址)，当 mret 时，我们取出 mepc 中的 PC 值并清空所有中断相关寄存器。使用 case 或 if-else 语句即可实现分别处理功能。

```
assign type = {INT,ecall,mret,ill_instr};
    always @* begin
        if(reset==1)begin
            mstatus = 0;
            mcause = 0;
            mepc = 0;
            mtval = 0;
            mtvec = 0;
            //pc = 0;
        end
        else begin
            case(type)
                4'b0000:begin//type={INT,ecall,mret,ill_instr};
                    mstatus   = mstatus;
                    mcause    = mcause;
                    mepc      = mepc;
                    mtval     = mtval;
                    mtvec     = mtvec;
                    pc  = pc_next;
                    end
                4'b0001:begin//type={INT,ecall,mret,ill_instr};
                    mstatus = 32'h00000001;
                    mcause  = 32'h03333333;
                    mepc    = pc_next;
                    mtval   = inst ;

                    mtvec   = 32'h0000027c;
                    pc      = mtvec;
                    end
                4'b0010:begin//type<={INT,ecall,mret,ill_instr};
                    pc = mepc;
                    mstatus = 32'h00000000;
                    mcause  = 32'h00000000;
                    mepc    = 0;
```

```
                mtval   = 0;

                mtvec   = 32'h0000000;
                end
            4'b0100:begin//type<={INT,ecall,mret,ill_instr};
                mstatus = 32'h00000100;
                mcause  = 32'h01111111;
                mepc    = pc_next;
                mtval   = pc_next-4;

                mtvec   = 32'h00000264;
                pc      = mtvec;
                end
            4'b1000:begin//type<={INT,ecall,mret,ill_instr};
            //if(mstatus==0)begin
                mstatus = 32'h00001000;
                mcause  = 32'h80000000;
                mepc    = pc_next;
                mtval   = 32'h12345678;

                mtvec   = 32'h00000294;
                pc      = mtvec;
            //end
            //else pc<=pc_next;
                end
            default:begin
                mstatus = 32'h00000000;
                mcause  = 32'h00000000;
                mepc    = 0;
                mtval   = 0;
                mtvec   = 0;
                pc      = pc_next;
            end

        endcase
    end
end
```

修改下板代码：

为尽可能减少对助教代码的影响，减少 debug 时间，所有的代码都添加至末尾，具体逻辑为在 L7 部分读取 GPIO 端口后跳转至 check 部分检查对应模式并跳转至对应部分，INT 中断由外部控制并在 Top 模块设计中添加约束，仅在矩形动画下能产生 int 信号。而为了让显示

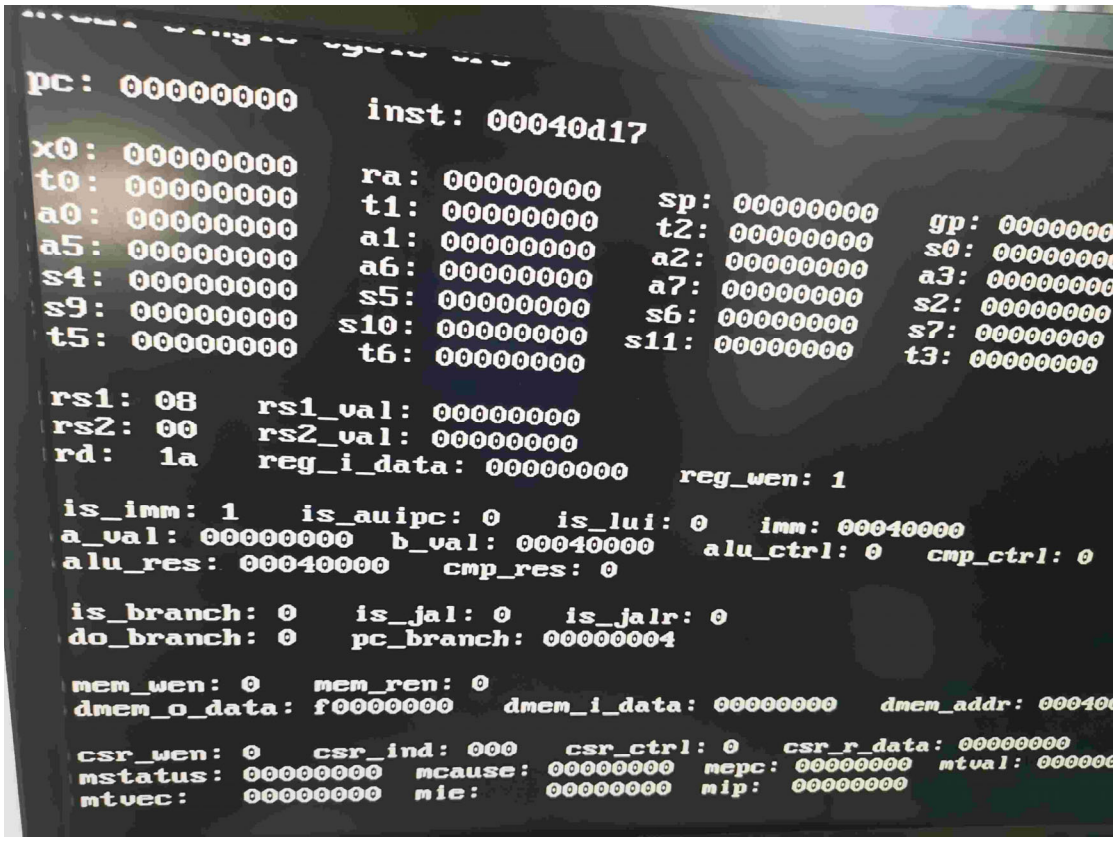
的动画较长时间存在，我们可以主动在程序中添加自增器，逐步增 1 直至溢出，这样就可以使数码管的显示持续一段时间。

```
L7:
lw x5, 0x0(x3) # 读 GPIO 端口 F0000000 状态
jal x0,check
Ck_ret:add x11, x5, x5
slli x11, x11, 1 # 左移 2 位将 SW 与 LED 对齐，同时 D1D0 置 00，选择计数器通道 0
sw x11, 0x0(x3) # x5 输出到 GPIO 端口 F0000000，计数器通道 counter_set=00 端口不变
sw x6, 0x4(x3) # 计数器端口：F0000004，送计数常数 x6=F8000000
bge zero, x4, l_next


check:
add x18, x14, x14           # x14=4,x18=8
add x22, x18, x18           # x22=00000010
add x18, x18, x22           # x18=00000018
and x11, x5, x18            # x11 取真实 sw[4:3]
add x18, x14, x14           # x18  sw[4:3] 01 学号
bne x11, x18, 12                # 符合不跳转执行 ecall
ecall
jal x0, Ck_ret
slli x18,x18,1              #sw[4:3] 10 计数器  illegal
bne x11,x18,8               #不执行非法，跳回
0x0000007f  #非法指令，无法编译，需后期添加入 coe
jal x0, Ck_ret

E_vec:
lw x28, 0xac(x0)            #5d615511
sw x28,0x0(x4)              #显示
lui x28, 0xff800
addi x28,x28,1
bne x28,x0,-4
mret

ill_vec:
lw  x28,0xb0(x0)
sw x28,0x0(x4)              #显示
lui x28, 0xff800
addi x28,x28,1
bne x28,x0,-4
mret
```

```
INT:
lw  x28,0xb4(x0)
sw x28,0x0(x4)              #显示
lui x28, 0xff800
addi x28,x28,1
bne x28,x0,-4
mret
```

二、实验结果与分析

下板验证：

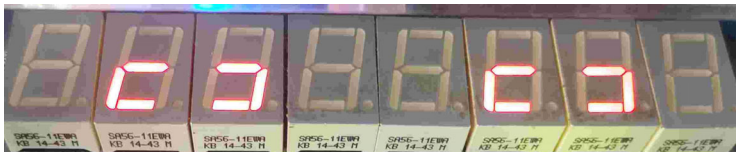


开机动画正常

学号对应 ecall 显示

计数器对应 illegal 显示

矩形动画，拨动一次开关 SW[1]进入一次中断，之后正常运行



三、讨论与心得

本次实验实现了中断，主要问题在于对于实验的原理不熟悉，开始看助教代码很吃力，后面想添加的代码放哪也花了点时间。然后原本是想通过助教的七段管译码程序写一个逆程序方便得到对应的数据，但结果写了几次出现各种问题，还以为自己 verilog 代码错了，最后还是自己一个个直接对应得到了七段管显示对应的数。而中断处理的硬件实现搞明白后反而比较简单，主要是 PC 值的存储和恢复，只有在 mret 时才需要将所有中断对应寄存器清 0，对于 cpu 来说中断程序内部的代码其实也是普通代码。另外，此次实验中挑选的是未使用的寄存器，因此没有进行寄存器值的保存，在实际运用中，我们要在中断程序中将我们使用的寄存器的值进行保存（存入内存中某一区域），并在程序结束时取出恢复寄存器值。