CS202-Project-CPU

开发者说明

成员

学号	姓名	贡献比
12210652	张伟祎	33.3%
12210723	王禀钦	33.3%
12210360	黄奕凡	33.3%

分工

	top	uart	asm1	asm2	模块实现	模块测试	MMIO
张伟祎	√	√	√		√	√	√
王禀钦			√	√	√	√	√
黄奕凡		√	√		√	√	√

开发计划日程安排

5.1-5.7

- project放出第一周，开始阅读项目要求，各自设计CPU部分

5.8-5.14

- 完成各个模块的初步设计，包括`PC, Controller, GenImm, ALU, inst_mem, data_mem`
- 一些输出和显示模块：`led`
- 用Java实现txt转coe小工具

5.15 - 5.21

- 连接uart
- 连接5个阶段内部：`IF, ID, EX, MEM, WB`
- 连接顶层`top`

5.22 - 5.26

- 每个模块单元测试
- 顶层测试

5.27 - 6.3

- 修改并测试 test1，test2 的 asm

CPU架构设计说明

CPU 特性

ISA（指令集架构）

- **参考的ISA**: RISC-V
- **寄存器**: 32bit寄存器，包括通用寄存器、程序计数器（PC）、栈
- **异常处理**: 当RISC-V程序里给x0寄存器赋值时，无法写入

实现指令集

（参考白老师bb站点上“RV32-reference_card”，除了`ecall, ebreak, sb, sh`其余指令均实现）

- R: `add, sub, xor, or, and, sll, srl, sra, slt,sltu`
- I: `addi,xori,ori,andi,slli,srli,srai,slti,sltiu`
- Load: `lb, lbu, lw, lh, lhu`
- Store: `sw`
- B: `beq,bne,blt,bge,bltu,bgeu`
- Jump: `jal,jalr`
- U: `auipc,lui`

时钟与性能

- **CPU时钟**:
    - FPGA (clk): 100MHz
    - Uart (uart_clk): 10MHz
    - CPU (cpu_clk): 10MHz
- **CPI**: 1
- **CPU类型**: 单周期（Single-Cycle）

寻址空间设计

- **结构类型**: 哈佛结构
- **寻址单位**: 以一个 word，32bit 为单位
- **指令空间与数据空间**: 都为 `16384*32` bit
- **栈空间**:
    - 栈空间的基地址 `0x7fff_effc`
    - 大小：和内存等大 `16384*32` bit

对外设IO的支持

- **MMIO** CPU通过内存映射IO（MMIO）访问外设
- 地址映射表

输入输出设备	对应地址
拨码开关[16个]	0xFFFFFC50
led灯	0xFFFFFC60
七段数码显示管	0xFFFFFC70
按钮	0xFFFFFC90

- **IO访问方式**: CPU是采用轮询的方法

CPU 接口

时钟 (Clock)

- 系统时钟 `clk` 是所有同步电路的时序信号来源。
- 时钟输入引脚定义和约束如下：

```
//clk
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property PACKAGE_PIN P17 [get_ports clk]
```

- **时钟源**：系统时钟由外部晶振提供，连接到FPGA的P17引脚。
- **时钟频率**：时钟频率应在硬件规格范围内，根据项目需求设置 `cpu_clk`(10MHz）和 `uart_clk`(10MHz）。
- **配置方法**：`upg_clk`分发给Uart模块、DMemory32模块，其余模块传入`cpu_clk`

复位 (Reset)

- 复位信号 `fpga_rst` 确保系统从已知状态启动。
- 复位输入引脚定义和约束如下：

约束文件内容:

```
//rst
set_property IOSTANDARD LVCMOS33 [get_ports fpga_rst]
set_property PACKAGE_PIN R15 [get_ports fpga_rst]
```

- **复位源**：外部复位信号连接到FPGA的R15引脚。
- **复位类型**：硬复位信号 `fpga_rst` （控制cpu进入工作模式）和软复位信号 `upg_rst` （控制cpu进入通信模式），Uart传输结束后进入正常模式
- **复位流程**：使用 `BUFG` 将复位信号分发至所有模块，确保系统初始化一致。
- **复位配置**：通过 `assign rst = fpga_rst | !upg_rst` 确定各模块复位信号。

```
wire spg_bufg;
   // 分发复位信号，确保其在整个芯片上同步
```

```
    BUFG U1(.I(start_pg), .O(spg_bufg));

    reg upg_rst; //active high
    always @ (posedge clk) begin
            if (spg_bufg) upg_rst <= 0;
            if (fpga_rst) upg_rst <= 1;
        end

    wire rst;
    assign rst = fpga_rst | !upg_rst; // reset

    wire kickOff = upg_rst | (~upg_rst & upg_done_o);
```

UART 接口

- UART用于串行通信，包含接收 `rx` 和发送 `tx` 信号。
- UART输入/输出引脚定义和约束如下：

```
//uart
set_property IOSTANDARD LVCMOS33 [get_ports rx]
set_property PACKAGE_PIN N5 [get_ports rx] //接收数据
set_property IOSTANDARD LVCMOS33 [get_ports tx]
set_property PACKAGE_PIN T4 [get_ports tx] //发送数据
```

- **波特率**：配置在 `uart_bmpg_0` 模块中。
- **配置方法**：通过 `uart_bmpg_0` 模块初始化UART接口。
- **数据格式**：默认数据格式，包括起始位、数据位、校验位和停止位。
- **通信协议**：符合标准的UART通信协议。

```
uart_bmpg_0 uart_inst(
    .upg_clk_i(upg_clk),
    .upg_rst_i(upg_rst),
    .upg_rx_i(rx),
    .upg_clk_o(upg_clk_o),
    .upg_wen_o(upg_wen_o),
    .upg_adr_o(upg_adr_o),
    .upg_dat_o(upg_dat_o),
    .upg_done_o(upg_done_o),
    .upg_tx_o(tx)
);
```

其他 IO 接口

按钮 (Button)

- 按钮输入信号包括 `start_pg` [V1]、`confirm` [R11]、`up` [U4]、`down` [R17]。
-
    - `start_pg` (V1): 启动
    - `confirm` (R11): 确认
    - `up` (U4) `down` (R17): 状态切换

```
button button_inst(
    .clk(cpu_clk),
    .rst(rst),
    .IORead(IORead),
    .ButtonCtrl(ButtonCtrl),
    .buttonaddr(addr_out[1:0]),
    .confirm(confirm),
    .up(up),
    .down(down),
    .all_button(all_button)
);
```

开关 (Switch)

- 开关信号通过 `switchs` 模块处理。

```
switchs switchs_inst(
    .clk(cpu_clk),
    .rst(rst),
    .IORead(IORead),
    .SwitchCtrl(SwitchCtrl),
    .switchaddr(addr_out[1:0]),
    .switch(switch),
    .switchrdata(switchrdata)
);
```

LED

- LED信号通过 `leds` 模块控制。

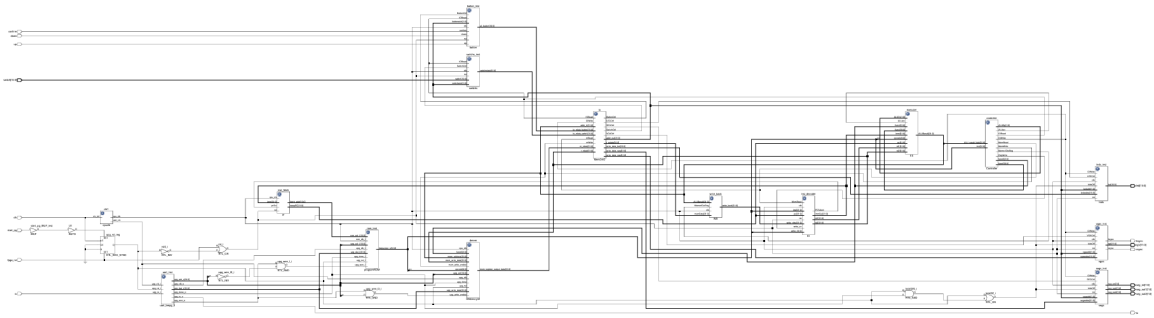
```
verilog leds leds_inst(
    .clk(cpu_clk),
    .rst(rst),
    .kickOff(kickOff),
    .IOWrite(IOWrite),
    .LEDCtrl(LEDCtrl),
    .ledaddr(addr_out[1:0]),
    .ledwdata(write_data_led),
    .led(led)
);
```

数码管

- 数码管信号通过 `segs` 模块控制。

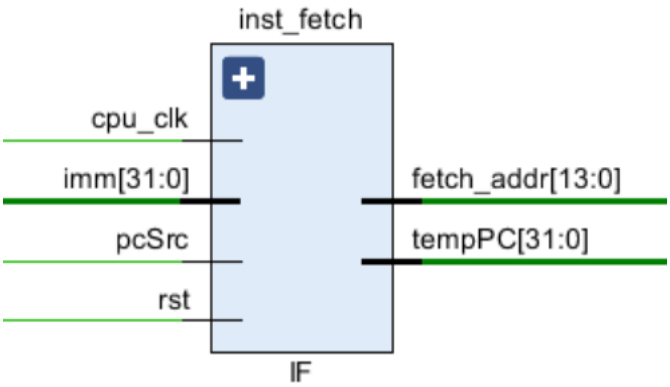
```
verilog segs segs_inst(
    .clk(cpu_clk),
    .rst(rst),
    .kickOff(kickOff),
    .IOWrite(IOWrite),
    .SEGCtrl(SEGCtrl),
    .segaddr(addr_out[1:0]),
    .segwdata(write_data_seg),
    .seg_en(seg_en),
    .seg_out1(seg_out1),
    .seg_out2(seg_out2)
);
```

CPU 内部结构



IF

- 结构图



- input:
    - `cpu_clk`
    - `rst`
    - `pcSrc`: 选择信号，筛选pc是否跳转
    - `imm`: 从ID来若pc跳转，需要加上的立即数
- output:
    - `curPC`
    - `tempPC`
    - `fetch_addr`
- 主要结构:
- IF

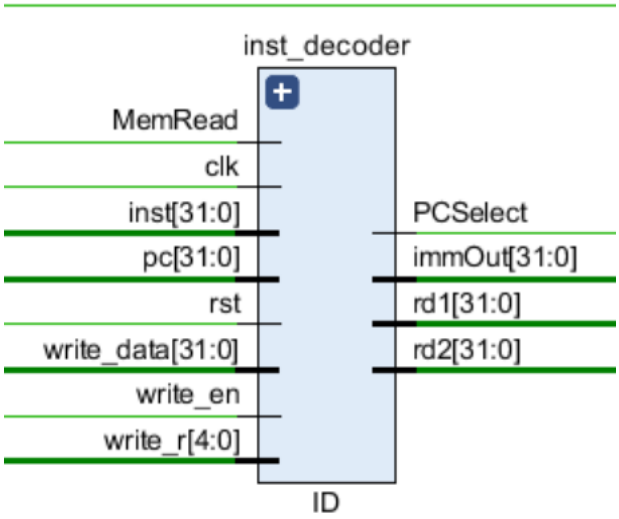
    - pc
- 功能：
    - CPU时钟下降沿时根据跳转选择信号pcSrc更新pc，
    - 时钟上升沿将pc的值赋给 `tempPC`

programROM

- 用一个信号控制是否使用uart。如果不用uart，直接根据读入的内存地址读取出需要的指令地址值；如果要使用uart，与要使用uart时钟，和一个 active high 的 reset 控制 uart 的开关，并且需要读取指令，就把指令放在对应的位置。

ID

- 结构图



主要结构：

- ID
    - Register
    - GenImm
    - PCSelect
        - compare
- 将Branch相关指令的跳转判断放在ID模块执行，决定是否跳转所需时间更短，且不需要经过后面三个阶段，效率更高，因此增添 `PCSelect` 模块，输出 `PCSrc` 返回IF阶段，实现pc跳转的选择，模块结构如下：
  （图,最后再加）
- 关键代码：

```
always@* begin
    if (opcode == `opB) begin
        case(funct3)
        3'h0: PCSel = (compResult_s == `EQUAL)? `PCSEL_JUMP: `PCSEL_PC;
        3'h1: PCSel = (compResult_s != `EQUAL)? `PCSEL_JUMP: `PCSEL_PC;
        3'h4: PCSel = (compResult_s == `LESS)? `PCSEL_JUMP: `PCSEL_PC;
        3'h6: PCSel = (compResult_u == `LESS)? `PCSEL_JUMP: `PCSEL_PC;
        3'h5: PCSel = (compResult_s == `GREATER_EQ || compResult_s ==
`EQUAL)? `PCSEL_JUMP: `PCSEL_PC;
```

```
        3'h7: PCSel = (compResult_u == `GREATER_EQ || compResult_u == `EQUAL)? `PCSEL_JUMP: `PCSEL_PC;
        default PCSel = `PCSEL_PC;
        endcase
    end else if (opcode == `opJ || opcode == `opIJ) PCSel = `PCSEL_JUMP;
    else PCSel = `PCSEL_PC;
end
```

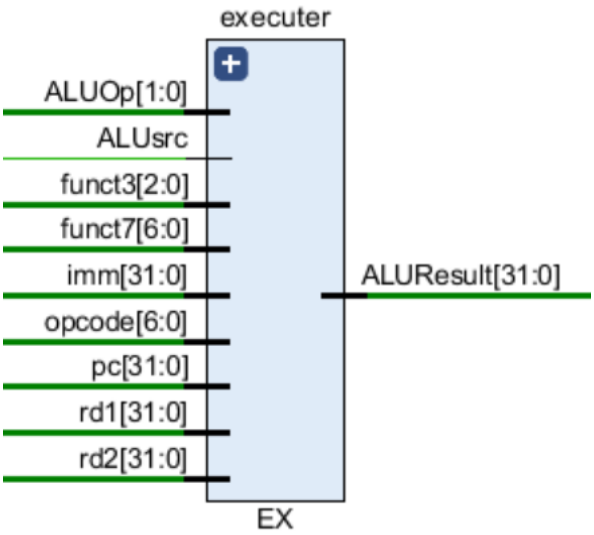
GenImm

- 根据不同指令类型生成不同的立即数
- 核心代码如下：

```
always @* begin
    case(opcode)
        `opI,`opIL,`opIE: out = {{20{inst[31]}},inst[31:20]}; // I
        `opIJ: out = {{20{inst[31]}},inst[31:20]}; // jalr
        `opS: out = {{20{inst[31]}},inst[31:25],inst[11:7]}; // S
        `opB: out = {{20{inst[31]}},inst[7],inst[30:25],inst[11:8],1'b0}; //B
        `opJ: out = {{12{inst[31]}}, inst[19:12],inst[20],inst[30:21],1'b0}; //J
        `opU,`opAU: out = {inst[31:12]}; //U
        default: out = 0;
    endcase
end
```

EX

- 结构图

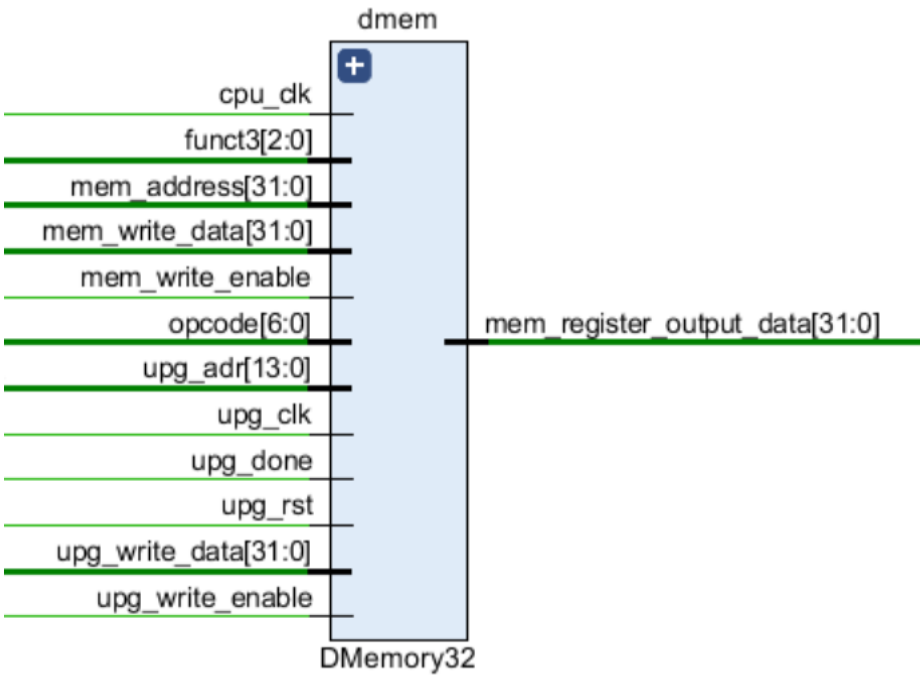


- Input:
    - `rd1`: 从寄存器中取出的数
    - `rd2`: 从寄存器中取出的数
    - `imm`: ID传出的立即数
    - `pc`
    - `ALUsrc`: 来自controller，用于筛选第二个操作数来自寄存器还是立即数
    - `funct3`

    - `funct7`
    - `ALUOp`
    - `opcode`
- output:
    - `ALUResult`: ALU计算结果
- 在EX阶段的ALU模块里，通过ALUOp与funct3、funct7共同决定，确定运算符

DMemory

- 结构图



- Input:
    - `cpu_clk`
    - `mem_write_enable`
    - `mem_address`
    - `mem_write_data`
    - `opcode`
    - `funct3`
    - `upg_rst`
    - `upg_clk`
    - `upg_write_enable`
    - `upg_adr`
    - `upg_write_data`
    - `upg_done`

  Output:
    - `mem_register_output_data`: 从内存中取出的数据

  Data Memory的内部有一个ip核，可以区分是uart访问还是程序访问

  不过这个IP核在读取数据的时候，只能一次读出32bit，因此传入的地址是`mem_address[15:2]`，所以要是想实现`lb,lbu,lh,lhu`需要对取出的数据作一定的修改，我们定义变量`offset[1:0]`来对最后两位进行计算。

  逻辑如下：

```
wire [1:0] offset = mem_address[1:0];

always@* begin
    if(opcode == `opIL) begin
        case(funct3)
            3'h0: mem_o =   (offset == 2'b01) ? {{24{out[15]}},out[15:8]}:
                            (offset == 2'b10) ? {{24{out[23]}},out[23:16]}:
                            (offset == 2'b11) ? {{24{out[31]}},out[31:24]}:

                {{24{out[7]}},out[7:0]}; // lb
            3'h1: mem_o =   (offset == 2'b10) ? {{16{out[31]}},out[31:16]} :

                {{16{out[15]}},out[15:0]}; // lh
            3'h4: mem_o =   (offset == 2'b01) ? {24'b0,out[15:8]}:
                            (offset == 2'b10) ? {24'b0,out[23:16]}:
                            (offset == 2'b11) ? {24'b0,out[31:24]}:
                                                {24'b0,out[7:0]}; // lbu
            3'h5: mem_o =   (offset == 2'b10) ? {16'b0,out[31:16]} : {16'b0,out[15:0]}; // lhu
            default: mem_o = out;
        endcase
    end else mem_o = out;
end
```

WB

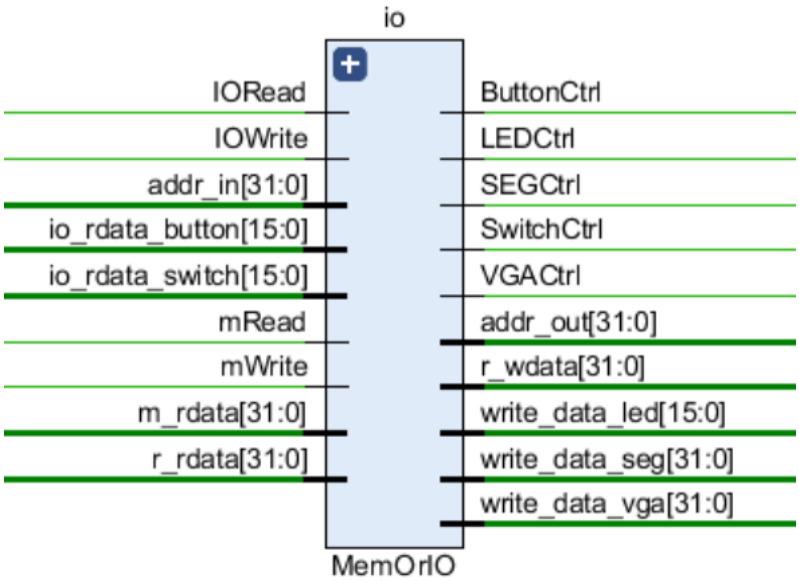
- 结构图

- 多路选择器，用来筛选写回寄存器的数据是来自data memory还是ALU。
- Input:
    - `clk`: cpu clock
    - `memData`: 内存中取出的数据
    - `MemorIOtoReg`: control bit
    - `ALUResult`: ALU计算结果
- Output:
    - `write_back`: 写回寄存器中的数据
- 主要代码如下:

```
always@* begin
    case(MemorIOtoReg)
        1'b1: write_back = memData;
        default: write_back = ALUResult;
    endcase
end
```

MemOrIO

- 结构图



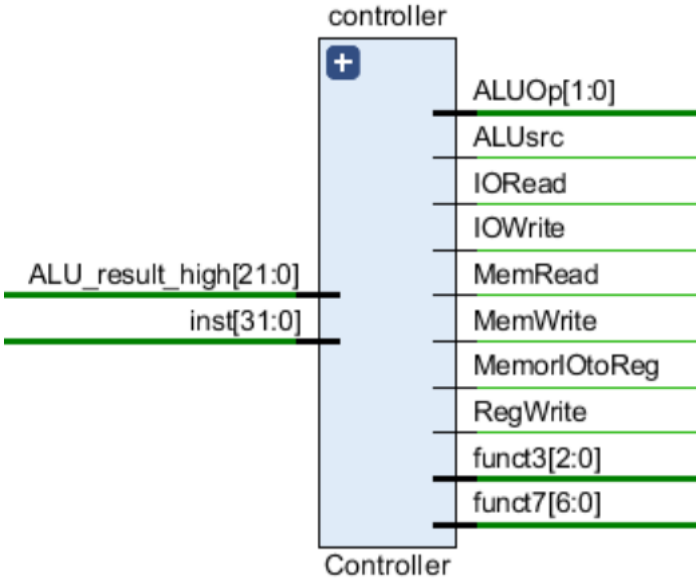
- 在asm中`lw`有两个作用，一个是从内存中读取数据，另一个是从开发板中读取数据，`sw`同理。因此，这个模块主要用于区分数据读入的来源和输出的去向

```
reg [15:0] io_rdata;
always @* begin
    if(ButtonCtrl) io_rdata = io_rdata_button;
    else if (SwitchCtrl) io_rdata = io_rdata_switch;
end

assign addr_out   = addr_in;
assign r_wdata    = (IORead  == 1'b1)?{16'h0000,io_rdata}:m_rdata; // may from memory or I/O, if from I/O, it is rdata's lower 16bit
assign ButtonCtrl = (IORead == 1'b1 && addr_in[7:4] == 4'b1001)?1'b1:1'b0; //buttonCtrl, 1 is effective   ? 0xFFFF_FC90
assign SwitchCtrl = (IORead == 1'b1 && addr_in[7:4] == 4'b0101)?1'b1:1'b0; //switchCtrl, 1 is effective   ? 0xFFFF_FC50
assign LEDCtrl    = (IOWrite == 1'b1 && addr_in[7:4] == 4'b0110)?1'b1:1'b0; // ledCtrl, 1 is effective   ? 0xFFFF_FC60
assign SEGCtrl    = (IOWrite == 1'b1 && addr_in[7:4] == 4'b0111)?1'b1:1'b0; //segCtrl, 1 is effective   ? 0xFFFF_FC70
```

Controller

- 结构图



- controller是CPU核心部分，主要用于输出各个模块的control bits，包含`Branch, LUOp, ALUsrc, MemorIOtoReg, RegWrite, MemRead, MemWrite, IORead, IOWrite,`
- 主要代码如下

```
// Branch
assign Branch = (instType == `typeB)? 1'b1:1'b0;

// ALUOp
// load/store 00; Branch 01; R-type 10
always @* begin
    case(opcode)
        `opIL,`opS: ALUOp = 2'b00;
        `opI,`opR: ALUOp = 2'b10;
        `opU,`opAU: ALUOp = 2'b11;
        default: ALUOp = 2'b01;
    endcase
end

wire src = (ALU_result_high[21:0] == 22'b1111111111111111111111)? 1'b1:1'b0;

// ALUsrc
assign ALUsrc       = (instType == `typeB || instType == `typeR)? 1'b0 : 1'b1;
assign RegWrite     = (instType == `typeB || instType == `typeS)? 1'b0 : 1'b1;
assign MemWrite     = (instType == `typeS && src == 1'b0       )? 1'b1 : 1'b0;
assign MemRead      = (opcode == `opIL    && src == 1'b0       )? 1'b1 : 1'b0;
assign IORead       = (opcode == `opIL    && src == 1'b1       )? 1'b1 : 1'b0;
assign IOWrite      = (instType == `typeS && src == 1'b1       )? 1'b1 : 1'b0;
```

```
// Read operations require reading data from memory or I/O to write to the register
assign MemorIOtoReg = IORead || MemRead;
```

Bonus 部分

Bonus 共包含以下三个部分：

- Uart
- lui，auipc
- coe文件创作工具

Uart

- 用 uart 进行文件传输，将 instruction 和 data memory 合成 out.txt 文件之后通过 uart 传输
- 核心代码

```
//  Uart
    uart_bmpg_0 uart_inst(
    .upg_clk_i(upg_clk),
    .upg_rst_i(upg_rst),
    .upg_rx_i(rx),
    .upg_clk_o(upg_clk_o),
    .upg_wen_o(upg_wen_o),
    .upg_adr_o(upg_adr_o),
    .upg_dat_o(upg_dat_o),
    .upg_done_o(upg_done_o),
    .upg_tx_o(tx)
    );
```

`lui, auipc`

设计说明

`lui`是将立即数左移12位存入寄存器中

`auipc`是将立即数左移12位之后加上pc存入寄存器中，其中需要注意的是我们程序中pc起始地址是0，而risc-v中起始地址为0x00400000，内存起始地址为0x10010000，因此要减去偏移量。

这两条指令只需要在ALU部分多加一个case进行运算

```
always@* begin
    if(ALUOp==2'b00) ALUControl = 4'h0;
    ......
    else if(ALUOp == 2'b11 && opcode == `opU) ALUControl = 4'hb; // lui
    else if(ALUOp == 2'b11 && opcode == `opAU) ALUControl = 4'hc; // auipc
    ......
end

always@* begin
    case(ALUControl)
        ......
        4'hB: ALUResult = operand2 << 12; // lui
```

```
        4'hC: ALUResult = pc + (operand2 << 12) + 32'h00400000-32'h10010000;
// auipc
        ......
        default: ALUResult = 32'h0;
    endcase
end
```

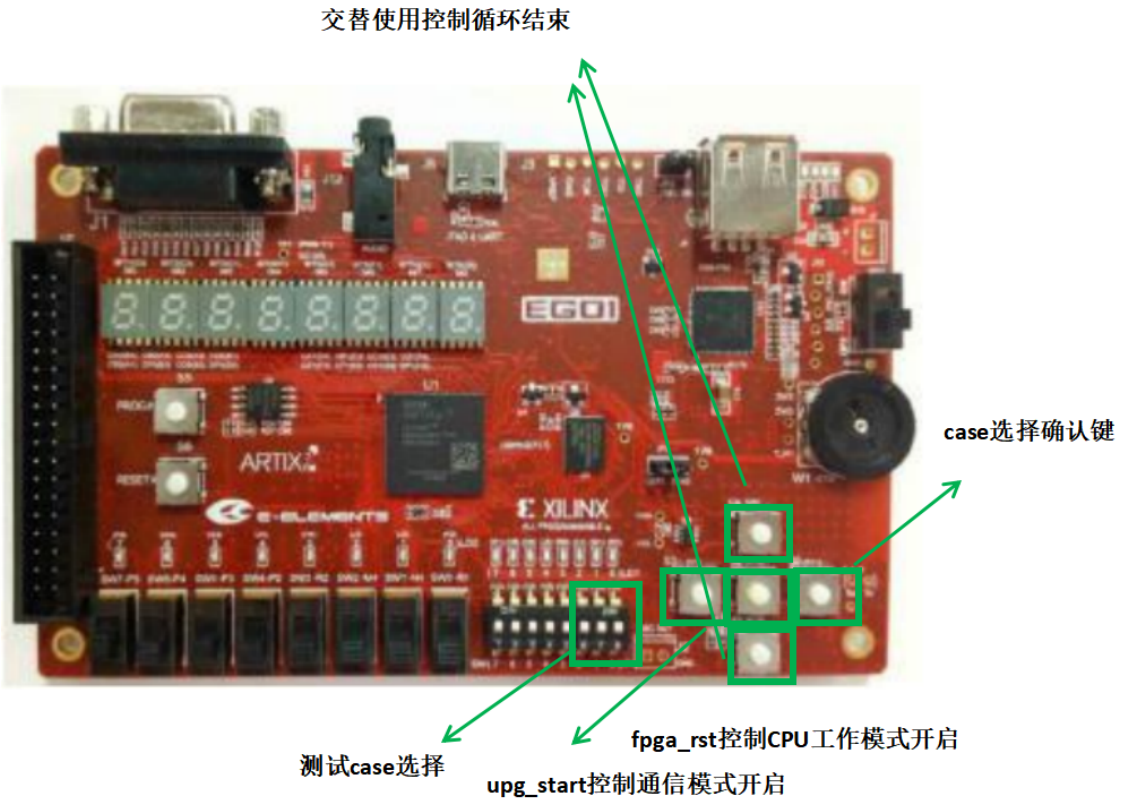
coe文件创作工具

- 实现 RISC - V 生成的 txt 文件转换为可以直接被 verilog 识别的 coe 文件
    - 使用方法
        - 输入读入文件的路径
        - 输入输出文件的路径
    - 特性
        - 路径错误会提示路径错误
        - 检测每一行是否为八个字符
        - 检测是否为十六进制数

```
for (int i = 0; i < line.length(); i++) {
    int index = line.charAt(i);
    if (!((index >= 48 && index <= 57) || (index >= 65 && index <= 70)
          || (index >= 97 && index <= 102))) {
        System.err.print("wrong content error in " + (counter + 1) +
                         " line " + (i + 1) + " column: " + (char) index + "
("  + index + ")");
        return;
        }
    }
    if (line.length() != 8) {
        System.err.print("length error" + (counter + 1) + " line ");
        return;
    }
}
```

系统上板使用说明

- 点击按钮 left [V1] 开启通信模式，所有led灯全亮，传输结束后led灯回熄灭
- 点击 center [R15] 进入CPU工作模式
- 使用后三位拨码开关选择case 选择的case会实时显示在七段数码显示管上
- 点击 right [R11]确认case选择
- 在每个case交替点击 up[U4]， down [R17] 确定输入或结束显示（结束循环）



自测说明

总体测试

- 出现 timing 的 critical warning
- 解决：将uart的控制时钟(upg_clk)由 23MHz 换为 10MHz
- 出现reset信号逻辑错误
    - 解决：使用按键 V1 R15 均为低电平有效 明确不同reset信号的功能

```
input start_pg
input fpga_rst, // 用于开启cpu工作模式
input start_pg, // 用于控制uart进入通信模式
wire spg_bufg;
// 分发复位信号，确保其在整个芯片上同步
    BUFG U1(.I(start_pg), .O(spg_bufg));

    reg upg_rst; //active high
    always @ (posedge clk) begin
            if (spg_bufg) upg_rst <= 0;
            if (fpga_rst) upg_rst <= 1;
        end

 wire rst = fpga_rst|!upg_rst // 用于传入各个模块进行reset
 wire kickOff = upg_rst | (~upg_rst & upg_done_o);
```

单元测试

Uart

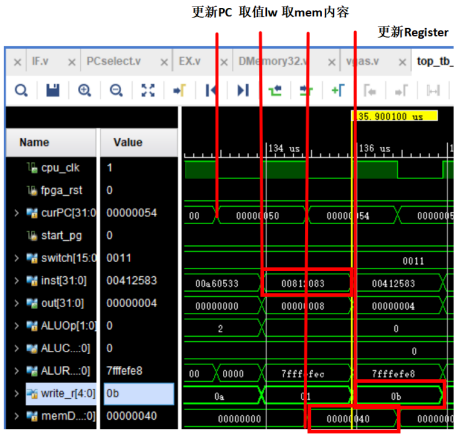
测试方法	问题描述	解决方案	测试结果
仿真	upg_rst和fpga_rst之间的逻辑错误，无法正确开启通信模式	项目所用按钮按下为高电平，upg_rst用于控制uart，使用按键V1；fpga_rst用于进入工作模式，使用按键R15	通过
上板	使用串口传输助手发送，出现了先接收后发送的问题	原来换个板子就好了：）	通过
上板	读内存时无法读出正确的内容，最后输出总是乱码	发现给inst内存和data memory的使能信号有问题，导致把inst写入data memory里，因此读出的“乱码”其实是machine code	通过

IF模块

测试方法	测试用例描述	测试结果
仿真	是否能正确的pc+4	通过
仿真	测试branch指令要跳转时，是否能pc+imm	通过
仿真	测试`tempPC`是否能比pc滞后半个周期	通过

ID模块

1. 计算`pc`和计算`ALUResult`都通过`ALU`模块，会产生冲突。基于**下降沿更新`pc`、上升沿取指、下降沿写入Mem**的设计，最初在时钟下降沿写回`Register`。如图所示为`lw`指令，会将Mem取回的值错误的存在下一条指令对应的寄存器。



- 解决：在最初的解决方式中我们尝试了对`ALUResult`进行延迟处理，发现行不通后又继续往前对很多传递途径中涉及的信号进行了延迟，时钟没有解决问题。
- 最终我们发现一条指令的处理周期不能超过两个时钟周期，因此我们改为在**时钟上升沿写回`Register`**，解决了`lw`指令的问题，并进一步对`pc`进行半个周期的`latch`，以服务于指令跳转。

EX模块

`EX` 模块最为主要的是`ALU`部分

测试方法	测试用例描述	测试结果
仿真	测试`load`,`store`指令时`ALU`计算情况（`ALUOp=00`）	通过
仿真	测试`I`,`R`两种类型指令在不同`funct3`，`funct7`影响下计算结果	通过
仿真	测试`auipc`	通过
仿真	测试`jal`,`jalr`能否正确存储`pc + 4`	通过

MMIO

测试方法	测试用例描述	测试结果
上板	通过拨码开关与按钮输入，通过LED和七段数码显示管输出	通过

测试用例1

用例编号	测试方法	测试用例描述	测试结果
0	上板	输入测试数a, 输入测试数b，在输出设备（led）上展示8bit的a和b的值	通过
1	上板	输入测试数a ，以lb的方式放入某个寄存器，将该32位的寄存器的值以十六进制的方式展示在输出设备上（数码管或者VGA)，并将该 数保存到memory中（在3’b011-3’b111用例中，将通过lw 指令从该memory单元中读取a的值进行比较）	通过
2	上板	输入测试数b，以 lbu 的方式存入某个寄存器，将该32位寄存器的值以十六进制的方式展示在输出设备上（数码管或者VGA)，并将该 数保存到memory中（在3’b011-3’b111用例中，将通过lw 指令从该memory单元中读取a的值进行比较）	通过
3	上板	用 beq 比较 测试数 a 和 测试数 b（来自于用例1和用例2），如果关系成立，点亮led，关系不成立，led熄灭	通过
4	上板	用 blt 比较 测试数 a 和 测试数 b（来自于用例1和用例2），如果关系成立，点亮led，关系不成立，led熄灭	通过
5	上板	用 bge 比较 测试数 a 和 测试数 b（来自于用例1和用例2），如果关系成立，点亮led，关系不成立，led熄灭	通过
6	上板	用 bltu 比较 测试数 a 和 测试数 b（来自于用例1和用例2），如果关系成立，点亮led，关系不成立，led熄灭	通过

用例编号	测试方法	测试用例描述	测试结果
7	上板	用 bgeu 比较 测试数 a 和 测试数 b（来自于用例1和用例2），如果关系成立，点亮led，关系不成立，led熄灭	通过

测试用例2

用例编号	测试方法	测试用例描述	测试结果
0	上板	输入一个8bit的数，计算并输出其前导零的个数	通过
1	上板	输入16bit位宽的IEEE754编码的半字浮点数，对其进行向上取整，输出取整后的结果	通过
2	上板	输入16bit位宽的IEEE754编码的半字浮点数，对其进行向下取整，输出取整后的结果	通过
3	上板	输入16bit位宽的IEEE754编码的半字浮点数，对其进行四舍五入取整，输出取整后的结果	通过
4	上板	分两次输入两个8bit的数a和b，对a，b做加法运算，如果相加和超过8bit，将高位取出，累加到相加和中，对相加和取反后输出	通过
5	上板	输入12bit的数据，以小端模式从拨码开关输入，以大端的方式呈现在输出设备上	通过
6	上板	以递归的方式计算小于输入数据的斐波拉契数字的数目，记录本次入栈和出栈次数，在输出设备上显示入栈和出栈的次数之和	通过
7	上板	以递归的方式计算小于输入数据的斐波拉契数字的数目，记录入栈和出栈的数据，在输出设备上显示入栈的参数，每一个入栈的参数显示停 留2-3秒 （说明，此处的输出不关注ra的入栈和出栈）	通过

遇到的问题及解决方案：

1. 输入输出无法交互，输出设备无显示
    - 解决：所有输入设备的读入和更新、所有输出设备的更新均使用组合逻辑，避免因为时钟沿更新导致的冲突和错误
2. 用按钮实现确定、跳转等功能时，因为按钮0-1状态不稳定，而读取输入和显示输出又需要在loop里反复执行，一个确认按钮很容易导致一次跳转多个循环
    - 解决：使用两个按钮交替作为确认按键，通过规定常数（001，010，100）来表示不同的按钮。

```
addi a5, zero, 0x4 #confrm--100--for scene choose
```

```
addi a6, zero, 0x2 #up--010
addi a7, zero, 0x1 #down--001

##场景选择##

case0:
    sw zero, 0x70(s11)
    sw zero, 0x60(s11)
loop1:
        lw a0, 0x50(s11)
        lw a1, 0x90(s11)
        bne a1, a6, loop1 #a6

loop2:
        sw a0, 0x60(s11)
        lw a1, 0x90(s11)
        bne a1, a7, loop2 #a7

loop3:
    sw zero, 0x60(s11)
        lw a0, 0x50(s11)
    lw a1, 0x90(s11)
        bne a1, a6, loop3 #a6

loop4:
        sw a0, 0x60(s11)
        lw a1, 0x90(s11)
        bne a1, a7, loop4 #a7

loop5:
    sw zero, 0x60(s11)
    lw a1, 0x90(s11)
    bne a1,a6,loop5 #a6
    beq zero,zero,choose

```

3. 数码管出现闪烁问题，且显示内容与预期不符，发现原因是在同一个loop中对led灯和七段数码显示管赋不同的值，会导致硬件中ALUResult的结果反复变化，存入输出设备的值也在循环变化。
    - 解决: 每个循环中如果led和七段数码显示管同时需要展示结果，则展示相同的结果，且每次进入一个case的时候先对数码管和led灯进行清空，清除上一个case中可能存储的脏数据。例如：

```
sw zero, 0x70(s11)
```

4. Type Error 寄存器用混导致输出结果不正确
    - 解决：以测试场景1为例，以a0专门存储输出结果，以a1专门存储按钮的结果。

```
        add  a0, x0, x0
loop62:
        sw  a0,  0x60(s11)
        lw  a1,  0x90(s11)
        bne  a1, a7, loop62  #a7
        beq  x0, x0, choose

loop61:
        addi  a0, x0, 1
loop63:
        sw  a0,  0x60(s11)
        lw  a1,  0x90(s11)
        bne  a1, a7, loop63  #a7
        beq  x0, x0, choose
```

5. 输出结果与 RISC-V 结果与数码管显示的内容不同
    - 解决：由于在汇编中插入大量死循环，直接仿真难以发现问题，需要上板具体解决问题。首先检测每一个测试用例的逻辑是否正确，确认逻辑无误后通过在汇编文件中插入死循环并输出结果的方式找到具体的错误位置，从而发现一些问题。有时会遇到一些很奇怪的输出无法解决，重新生成 out 文件就可以解决，最后一个实在没办法了，就重写里一遍汇编代码，结果就能正常显示了😀。
6. 汇编的 IO 方式与 CPU 中实现的 IO 方式不同
    - 解决：及时与写 IO 的同学沟通，单独改动一个方面不能真正解决问题，一个按键不好实现输入就用两个按键分别控制输入的进行和终止。
7. 最麻烦的是汇编代码和 CPU 同时出现 bug
    - 解决：需要非常冷静地分析问题，同时仿真和汇编，逐步比对，发现问题。
8. 进行 IO 时不知道当前的进行的步骤
    - 解决：根据不同用例，不同阶段在数码管上显示不同的内容，同时不能与输出结果相冲突。

思考与总结

- 写不同模块时可以相互检验其他部分的代码逻辑是否正确，单独进行仿真或者直接上板难以直接确定问题。
- 每个人只了解自己写过的代码，不够了解其他模块，导致 debug 只能从自己写过的部分找问题。后面了解了整个代码的框架之后，会快速找到问题。
- CPU 是由硬件控制执行逻辑，软件控制执行的内容。与用户交互的部分需要根据实际情况软硬件相结合共同控制，需要写不同模块的同学共同讨论出合适的解决方案，相互协商共同解决问题。