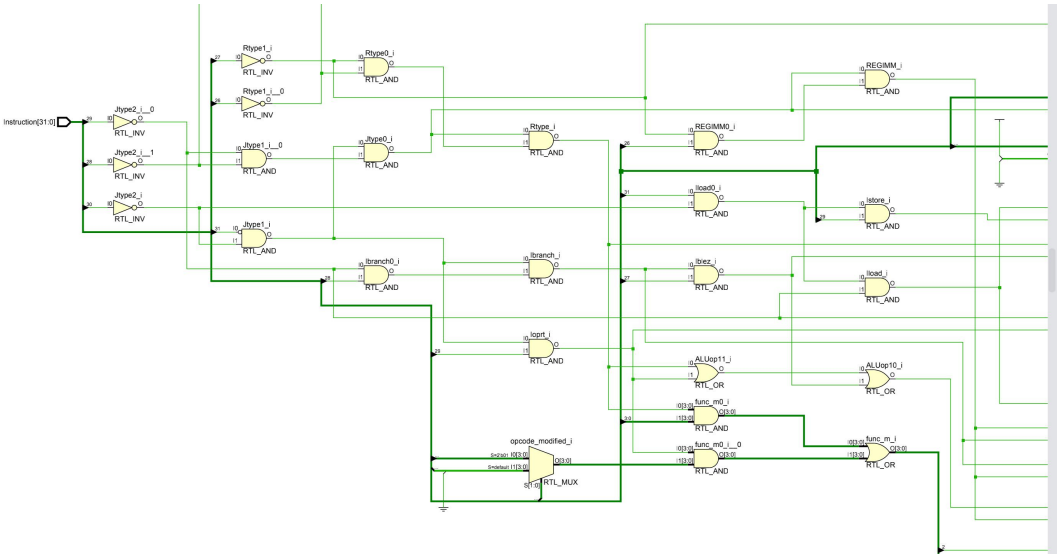
中国科学院大学计算机组成原理实验课

实 验 报 告

学号：_2019K8009908002_ 姓名：__何之粼__ 专业：___计算机科学与技术___
实验序号：_2_ 实验名称：_____单周期处理器设计_____

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

//instruction decode



```
assign Rtype = (~Opcode[5] & ~Opcode[4]) & (~Opcode[3] & ~Opcode[2]) & (~Opcode[1] &
~Opcode[0]);//6'b000000
assign REGIMM = (~Opcode[5] & ~Opcode[4]) & (~Opcode[3] & ~Opcode[2]) & (~Opcode[1] &
Opcode[0]);//6'b000001
assign Jtype = (~Opcode[5] & ~Opcode[4]) & (~Opcode[3] & ~Opcode[2]) & Opcode[1];//5'b000001
assign Ibranch = (~Opcode[5] & ~Opcode[4]) & (~Opcode[3] & Opcode[2]);//4'b0001
assign Ioprt = (~Opcode[5] & ~Opcode[4]) & Opcode[3];//3'b001
assign Iload = ( Opcode[5] & ~Opcode[4]) & ~Opcode[3];//3'b100
assign Istore = ( Opcode[5] & ~Opcode[4]) & Opcode[3];//3'b101

assign Ibeq = Ibranch & ~Opcode[1];
assign Iblez = Ibranch & Opcode[1];
assign op_shift = Rtype & Func[5:3]==3'b000;
assign op_jump = Rtype & {Func[5:3], Func[1]} == 4'b0010;
assign op_mov = Rtype & {Func[5:3],Func[1]} == 4'b0011;
assign jumpal = (op_jump & Func[0]) | (Jtype & Opcode[0]);//jal & jalr
assign jr = op_jump & ~Func[0];
assign jal = Jtype & Opcode[0];
assign lui = Ioprt & Opcode[2:0]==3'b111;
```

译码部分逐级译码，前面的大类采取非再与的写法，并用括号两两分组、提高效率。之后在大类基础上译出一些后面要用到的特殊指令，这样在写的时候不用再根据具体opcode，增加可读性、复用性和效率。

//control unit

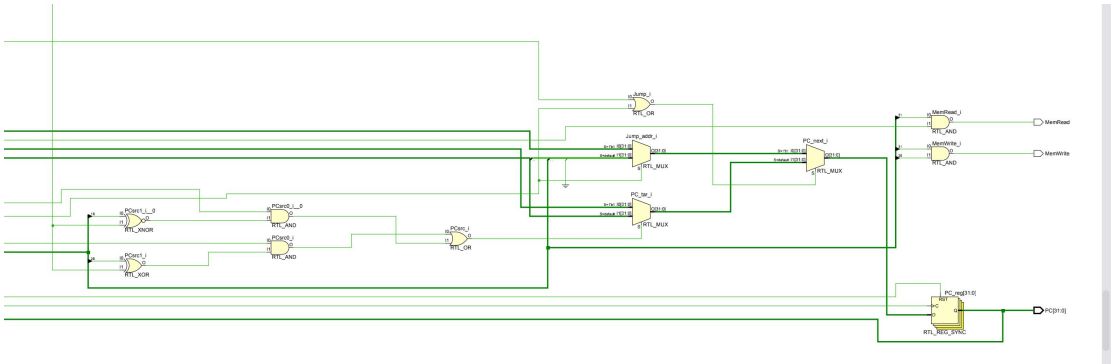
```
assign MemRead = Opcode[5] & (~Opcode[3]);
assign MemWrite = Opcode[5] & Opcode[3];
assign RegDst = Rtype;
assign Jump = Jtype | op_jump;
assign ALUsrc = Iload | Istore | Ioprt;
assign Mem2Reg = Iload;
assign RegWrite = Rtype | Iload | Ioprt | jal;
assign ALUop1 = Rtype | Ioprt | Iblez | REGIMM;
assign ALUop0 = Ibranch | REGIMM ;
assign PCsrc = (Ibranch & (Opcode[0] ^ ALU_zero)) | (REGIMM & (rt[0] ^~ ALU_zero));
```

控制信号根据各类指令的数据通路区别指定即可。

其中 PCsrc 需要额外依赖 zero 信号结合具体指令类型，一开始如下设置了一个branch_judge，根据多个译码的具体指令得出判断结果，后来发现略显多余，寻找规律简化后如上直接写到了 PCsrc 中。

```
assign branch_judge = (zero & (bgez | beq | blez)) | (~zero & (bltz | bne | bgtz));
assign PCsrc = Branch & branch_judge; //Branch=bgez|beq|... , meaningless sentence
```

//PC



```
assign PC_plus4 = PC + 32'd4;
assign PC_add = jumpal ? 32'd4 : shift_ext;
assign PC_result = PC_plus4 + PC_add;
assign Jump_tar = {PC_plus4[31:28],Instruction[25:0],2'b00};
assign Jump_addr = op_jump? rdata1 : Jump_tar;
assign PC_tar = PCsrc? PC_result : PC_plus4;
assign PC_next = Jump? Jump_addr : PC_tar;
always @(posedge clk) begin
    if(rst) PC<=32'd0;
    else PC <= PC_next;
end
```

PC 中要考虑是下一条还是分支，以及 jump 和特殊的 jal 和 jalr。

一开始的写法是在 RF 写入的端口直接写 PC+8：

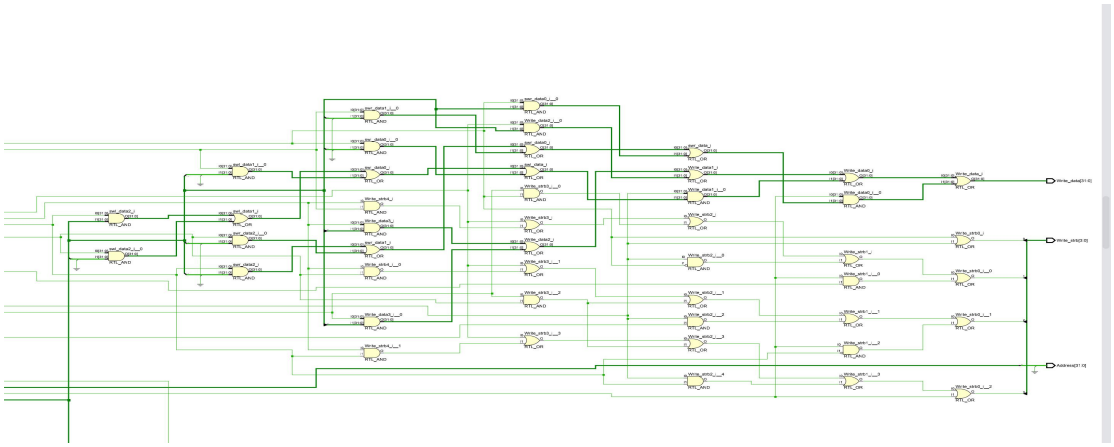
```
assign PC_plus4 = PC + 4;
assign PC_add = PC_plus4 + shift_ext;
assign PC_result = PCsrc? PC_add : PC_plus4;
...
assign RF_wdata = Mem2Reg? Load_data:
                        (jumpal? PC+8: //...
```

观察综合报告的时候，发现这样会多用一个加法器，于是修改后统一用 PC_result 来供写入时使用，用一个 MUX 换一个加法器。

```
250 Module mips_cpu
251 Detailed RTL Component Info :
252 +---Adders :
253           2 Input     32 Bit       Adders := 3
254 +---XORs :
255           2 Input      1 Bit         XORs := 3
256 +---Registers :
257                       32 Bit    Registers := 1
258 +---Muxes :
259           2 Input     32 Bit        Muxes := 10
260           2 Input      5 Bit        Muxes := 3
261           2 Input      4 Bit        Muxes := 1
```

```
250 Module mips_cpu
251 Detailed RTL Component Info :
252 +---Adders :
253           2 Input     32 Bit       Adders := 2
254 +---XORs :
255           2 Input      1 Bit         XORs := 3
256 +---Registers :
257                       32 Bit    Registers := 1
258 +---Muxes :
259           2 Input     32 Bit        Muxes := 11
260           2 Input      5 Bit        Muxes := 3
261           2 Input      4 Bit        Muxes := 1
```

//load & store



load 和 store 部分由于 32 位数据和各种情况显得规模较大。

首先先将不同类型由 Opcode[2:0]译码，除了 lhu 和 lbu，load 和 store 的其他可以共用。

接下来根据 ALU_result 后两位判断地址类型，并用一个独热编码表示。

```
assign sb = ~Opcode[2] & ~Opcode[1] & ~Opcode[0];//Opcode[2:0]==3'b000;
...
assign addrtype[0] = ~ALU_result[1] & ~ALU_result[0];//2'b00;
assign addrtype[1] = ~ALU_result[1] & ALU_result[0];//2'b01;
assign addrtype[2] = ALU_result[1] & ~ALU_result[0];//2'b10;
assign addrtype[3] = ALU_result[1] & ALU_result[0];//2'b11;
```

Write_srtb 的每一位根据不同的指令和地址类型分不同的情况，中间用或连接。

```
assign Write_strb[3] = sw | sb & addrtype[3] | sh & addrtype[2] | swl & addrtype[3] | swr;
```

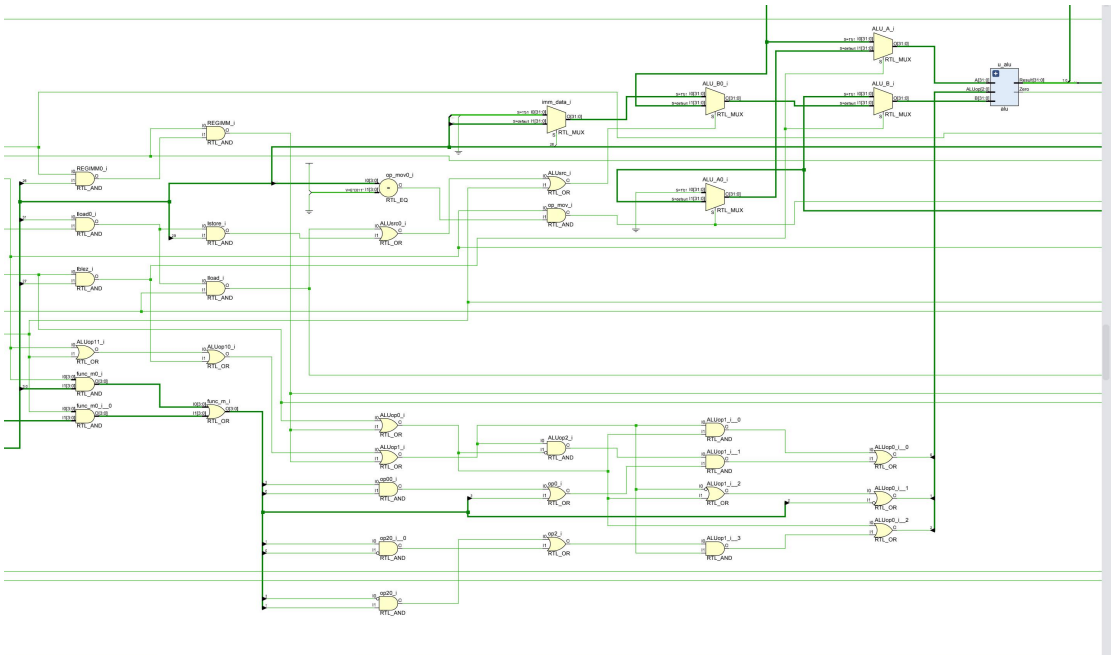
```
assign Write_strb[2] = sw | sb & addrtype[2] | sh & addrtype[2] | swl & (addrtype[3] | addrtype[2])
| swr & ~addrtype[3];
assign Write_strb[1] = sw | sb & addrtype[1] | sh & addrtype[0] | swl & ~addrtype[0] | swr &
(addrtype[0] | addrtype[1]);
assign Write_strb[0] = addrtype[0] | swl;
```

swr 和 swl 要根据地址类型预先处理好数据，最后用一个 MUX 选择输出。

```
assign swr_data = ({32{addrtype[3]}} & {rdata2[ 7:0],24'd0})
                | ({32{addrtype[2]}} & {rdata2[15:0],16'd0})
                | ({32{addrtype[1]}} & {rdata2[23:0], 8'd0})
                | ({32{addrtype[0]}} & rdata2);
assign swl_data = ({32{addrtype[3]}} & rdata2)
                | ({32{addrtype[2]}} & { 8'd0,rdata2[31:8]})
                | ({32{addrtype[1]}} & {16'd0,rdata2[31:16]})
                | ({32{addrtype[0]}} & {24'd0,rdata2[31:24]});
assign Write_data = ({32{sb}} & {4{rdata2[7:0]}})
                  | ({32{sh}} & {2{rdata2[15:0]}})
                  | ({32{sw}} & rdata2)
                  | ({32{swl}} & swl_data)
                  | ({32{swr}} & swr_data);
```

load 的思路较为类似，不再贴具体代码，也可以考虑一字节一字节处理而不是 32 位一起。

//ALU



```
assign opcode_modified = (Opcode[2:1]==2'b01)? Opcode[3:0] : {1'b0,Opcode[2:0]};
assign func_m = ({4{Rtype}} & Instruction [3:0]) | ({4{Ioprt}} & opcode_modified);
assign op2 = (~func_m[3] & func_m[1]) | (func_m[1] & ~func_m[0]);
assign op1 = ~func_m[2];
assign op0 = (func_m[2] & func_m[0]) | func_m[3];
```

```
assign ALUop[2] = ALUop0 | (ALUop1 & op2);
assign ALUop[1] = ~ALUop1 | ALUop0 | op1;
assign ALUop[0] = (ALUop1 & ALUop0) | (ALUop1 & ~ALUop0 & op0);
assign ALU_A = Iblez? rdata2: op_mov? 32'b0 :rdata1;
assign ALU_B = Iblez? rdata1: ALUsrc? imm_data : rdata2;
```

3 位 ALUop 的生成没有参考实验课讲义，而是参照的课堂方法：不用知道所有的 Opcode，根据 2 位 ALUop0 和 ALUop1 来决定最后 ALU 的 3 位 ALUop。这样多层解码减小主控单元规模，提高控制速度，不过可读性降低。

opcode_modified 和 func_m 是根据 Rtype 的 funcode 编码规则，将 Itype 的 opcode 稍作改动统一而得到的。

op012 是为了化简方便引入的，当 ALUop1 和 ALUop0 为 10（Rtype 和 Itype）时，刚好对应的是三位 ALUop。

具体的化简过程见后附。

//rf 接口

```
assign mov_judge = op_mov & (Func[0] ^~ rdata2==0);
assign lui_data = {Instruction[15:0],16'd0};
assign raddr1 = rs;
assign raddr2 = REGIMM? 0:rt;
assign RF_wen = (jr | mov_judge)? 0:RegWrite;
assign RF_waddr = jal? 6'd31 :RegDst? rd:rt;
assign Data_result = ({32{jumpal}} & PC_result)
                    |({32{lui}} & lui_data)
                    |({32{op_mov}} & rdata1)
                    |({32{op_shift}} & Shift_result)
                    |({32{~jumpal & ~lui & ~op_mov & ~op_shift}} & ALU_result);
assign RF_wdata = Mem2Reg? Load_data: Data_result;
assign Address = {ALU_result[31:2], 2'b00};
```

主要需要考虑一些 wen 和 waddr 的特殊情况，最后 MUX 选出要写入的数据。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，仿真、云平台调试过程中的难点等）

一开始顾虑于过多信号仿真会变慢，没有将所有信号添加，后来权衡重新添加信号跑的时间，还是基本将所有信号都添加了。

主要问题：

1.连线问题导致不定态。

```
assign alusft_result = Func[5]? ALU_result:Shift_result;
assign RF_wdata = Mem2Reg? Read_data: (jumpal? PC+8: alusft_result);
```

这样只用 Func[5]来作为选择条件看似简洁，但是由组合逻辑特性，实际上导致其他指令时 alusft_result 也被赋值，而 shifte_result 此时可能因为没有输入而处在不定态，因此 wdata 也会是不定态。

2.shifter 算术右移的结果仍然是逻辑右移。

```
assign Result = ({`DATA_WIDTH{shift_left}}         & (A << B[4:0])        )
              | ({`DATA_WIDTH{shifta_right}}   & ($signed(A) >>> B[4:0]))
              | ({`DATA_WIDTH{shiftl_right}}   & (A >> B[4:0])
```

一开始写成如上形式，括号出了一个微妙的错误。但是改正后仍然不行，最后改为

```
assign sra_result = ($signed(A)) >>> B[4:0];
assign Result = ({`DATA_WIDTH{sra}} & sra_result)...
```

才正确，猜测可能与 verilog 本身有关。

其他都是一些不仔细产生的小错误，如 raddr1 和 2 写反，三元判断里两项结果写反，有信号没有定义，名称错误等。在遇到这种错误的时候，很多时候我就根据 Instruction 字段去汇编里搜索对应的指令，再顺着该指令数据通路排查即可。

因此建议云平台可以显示错误处相应指令的内容，在有时避免打开波形查看。

三、 在课后，你花费了大约_____8_____小时完成此次实验。

四、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

//缺少的信息资料：

1.verilog 代码的规范指导。从 alu 和 reg_file 小部件模块的编写一下子跨越到 cpu，代码量陡然增加，却没有一些代码规范的指导（之前数电课也并没有，大概这也是 missing course of CS 吧）。比如声明和例化应该放在代码什么部分、命名规则、代码对齐、优良和不良的写法对比等。感觉注重这种细节，一能在初期养成好习惯，二能提高设计效率（曾有一个 bug 就是因为 zero 命名的大小写问题），三还能提高代码可读性、减少助教工作量。

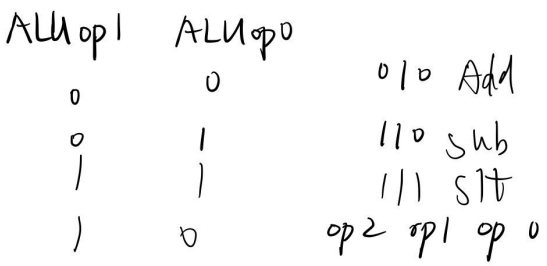
2.对 load&store 的讲解较为简略。特别是具体指令类型的要求和处理，以至于这块成为了一个设计上的难点。感谢与我一起讨论的同学，帮助我有了思路。

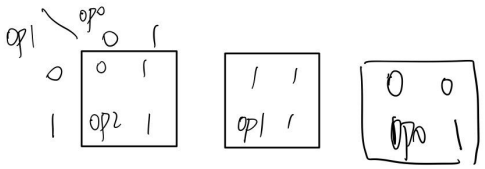
//心得感受：

在写的初期一直试图注重每个细节，并且尽量优化整体的设计，导致有点瞻前顾后、焦头烂额。比如添加新功能后，很多已经化简的又得修改；功能都写得差不多了再 push，一跑还是连第一个 benchmark 都没过。后来与同学老师交流，觉得还是应该先保证基本功能的实现，然后再进一步优化，何况当前情况下有些优化的实际作用也很有限。

不过矛盾的是，在实现以后思维和框架趋向固定，再要改动也很困难，找到这二者的平衡属实不易。

附：ALU control 卡诺图化简过程

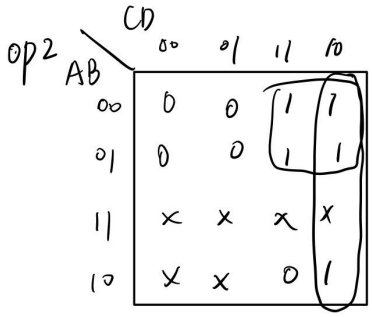




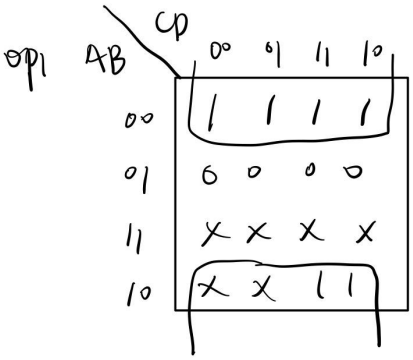
① ALUop0 | (ALUop1 & op2)

② ~ALUop1 | ALUop2 | op1

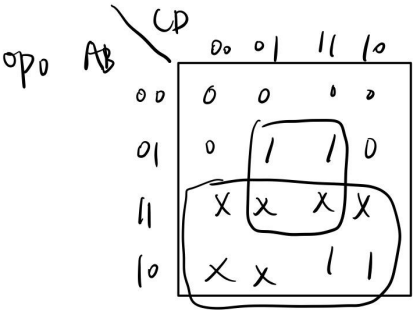
③ (ALUop1 & ALUop2) |
(ALUop1 & ~ALUop2 & op0)



$op_2 = (\sim func[3] \,\&\, func[1]) \mid (func[1] \,\&\, func[0])$



$op_1 = \sim func[2]$



$op_0 = func[1] \mid (func[2] \,\&\, func[0])$