中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K8009907015　姓名：高鸣驹　专业：计算机科学与技术

实验序号：02　　实验名称：单周期处理器设计

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在本地仓库的主目录下。文件命名规则：学号-prjN.pdf，其中学号中的字母“K”为大写，“-”为英文连字符，“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：2019K8009929000-prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：学号-prj5-projectname.pdf，例如：2019K8009929000-prj5-dma.pdf。具体要求详见实验项目 5 讲义。 注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 GitLab 远程仓库 master 分支（具体命令详见实验报告）。 注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

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）

1. 一些宏定义

```
`define SPECIAL 6'b000000
`define BRANCH 6'b0001
`define ADDI 5'b00100

`define ALUOP_AND 3'b000
`define ALUOP_OR  3'b001
`define ALUOP_ADD 3'b010
`define ALUOP_SUB 3'b110
`define ALUOP_STL 3'b111
`define ALUOP_XOR 3'b100
`define ALUOP_NOR 3'b101
`define ALUOP_SLTU 3'b011

`define SHIFTLEFT   2'b00
`define SHIFTRIGHT  2'b10
`define SHIFTRIGHTA 2'b11

`define TYPEALUIMM 4'b0000
`define TYPEALUR   4'b0001
`define TYPESHIFT  4'b0010
`define TYPESHIFTS 4'b0011
`define TYPEBRANCH 4'b0100
`define TYPEJ      4'b0101
`define TYPEJS     4'b0110
`define TYPELOAD   4'b0111
`define TYPESTORE  4'b1000
`define TYPEMOVE   4'b1111
```

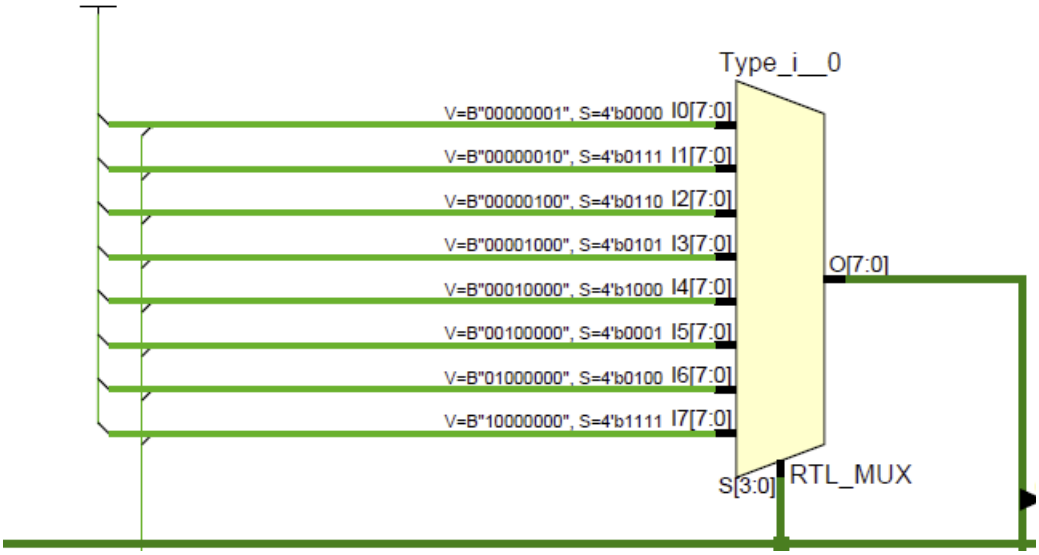
如上图所示，我设计的单周期处理器采用了以上的宏定义。ALUOP 的宏定义和 SHIFTER 的宏定义是照搬的相应模块里的宏定义,最上面的 SPECIAL 宏定义是 opcode 为 SPECIAL 时方便判断（即为 6 位 0），BRANCH 宏定义和 ADDI 宏定义代表 branch 指令和 addi 指令，提高代码的可读性。后面的关于 TYPE 的宏定义在第二条“Type 分类”中会有详细的介绍。

2. Type 分类

```
assign Type = (opcode[5:3] == 3'b001 && opcode != 6'b001111)? `TYPEALUIMM://ALU Immdiate word
              ((opcode == `SPECIAL && funccode[5] == 1)? `TYPEALUR://ALU rs rt rd
              ((opcode == `SPECIAL && funccode[5:2] == 4'b0000)? `TYPESHIFT://Shifter
              ((opcode == `SPECIAL && funccode[5:2] == 4'b0001)? `TYPESHIFTS://SHifter with some special operations
              ((opcode[5:2] == 4'b0001 || (opcode == 6'b000001 && (rt == 5'b00001 || rt == 5'b00000)))? `TYPEBRANCH://BRANCH
              ((opcode[5:1] == 5'b00001)? `TYPEJ://J with instr_index
              ((opcode == `SPECIAL && funccode[5:1] == 5'b00100)? `TYPEJS://J with special operations
              ((opcode[5:3] == 3'b100)? `TYPELOAD://Load
              ((opcode[5:3] == 3'b101)? `TYPESTORE: //Store
              ((opcode == `SPECIAL && funccode[5:1] == 5'b00101)? `TYPEMOVE://Move
              ((opcode == 6'b001111)? 4'b1001: 4'b1010))))))))));//Lui
```

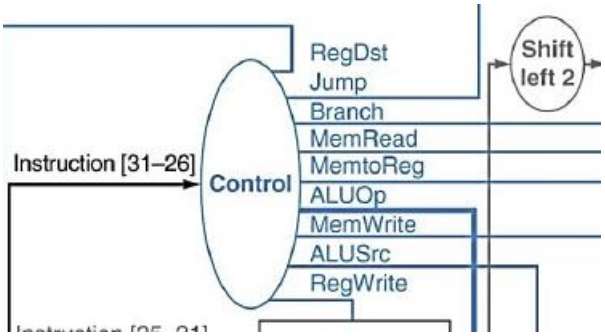
Type 分类代码如上图所示，我把所有的指令很细的分为了 11 类，其实在后

面有些信号很多 Type 可以合并为一类，具体的作用在注释都有写明，其中移位指令我分了两类,第一类是普通的 shifter 指令,即 sa 代表移动的位数，第二类是不太一样的，即 rs 的后五位代表移动的位数。类似的，jump 指令我也分了两类，一种是根据 instr_index 确定跳转地址的，另一种是根据其他方法确定跳转地址的。刚开始这里 Type 的编码没有采用宏定义，于是便很难理解，代码可读性很差，后来根据老师的建议采用了宏定义，代码可读性大大提高了。



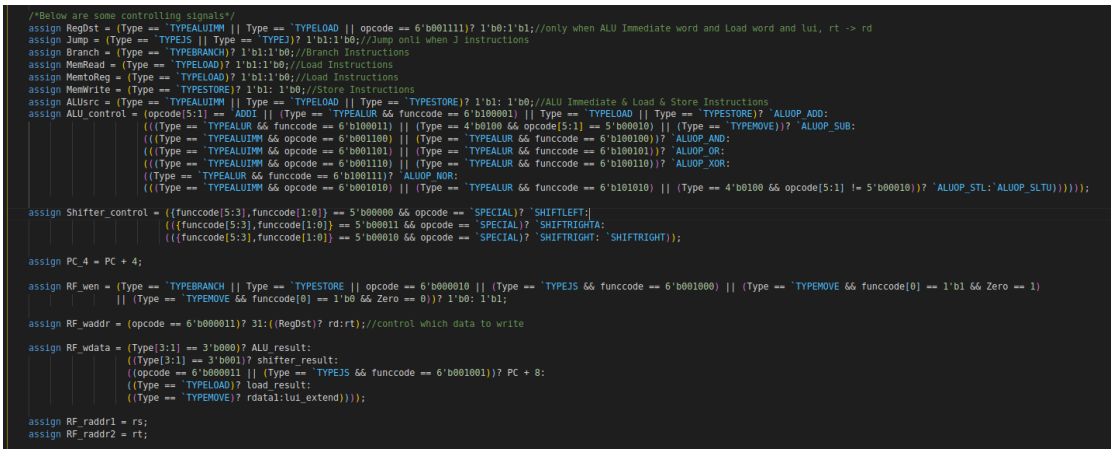
（Type 译码电路）

3. 一些关键的控制信号



（控制信号电路）

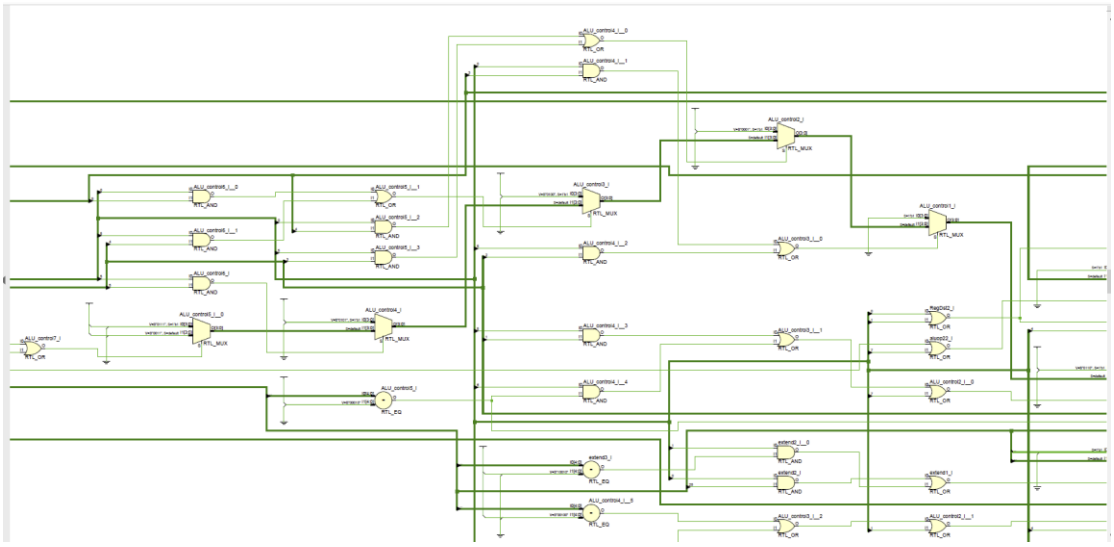
控制信号主要根据理论课所讲设计，具体代码如下：



```
/*Below are some controlling signals*/
assign RegDst = (Type == `TYPEALUIMM || Type == `TYPELOAD || opcode == 6'b001111)? 1'b0:1'b1;//only when ALU Immediate word and Load word and lui, rt -> rd
assign Jump = (Type == `TYPEJS || Type == `TYPEJ)? 1'b1:1'b0;//Jump onli when J instructions
assign Branch = (Type == `TYPEBRANCH)? 1'b1:1'b0;//Branch Instructions
assign MemRead = (Type == `TYPELOAD)? 1'b1:1'b0;//Load Instructions
assign MemtoReg = (Type == `TYPELOAD)? 1'b1:1'b0;//Load Instructions
assign MemWrite = (Type == `TYPESTORE)? 1'b1: 1'b0;//Store Instructions
assign ALUsrc = (Type == `TYPEALUIMM || Type == `TYPELOAD || Type == `TYPESTORE)? 1'b1: 1'b0;//ALU Immediate & Load & Store Instructions
assign ALU_control = (opcode[5:1] == `ADDI || (Type == `TYPEALUR && funccode == 6'b100001) || Type == `TYPELOAD || Type == `TYPESTORE)? `ALUOP_ADD:
                     (((Type == `TYPEALUR && funccode == 6'b100011) || (Type == 4'b0100 && opcode[5:1] == 5'b00010) || (Type == `TYPEMOVE))? `ALUOP_SUB:
                     (((Type == `TYPEALUIMM && opcode == 6'b001100) || (Type == `TYPEALUR && funccode == 6'b100100))? `ALUOP_AND:
                     (((Type == `TYPEALUIMM && opcode == 6'b001101) || (Type == `TYPEALUR && funccode == 6'b100101))? `ALUOP_OR:
                     (((Type == `TYPEALUIMM && opcode == 6'b001110) || (Type == `TYPEALUR && funccode == 6'b100110))? `ALUOP_XOR:
                     ((Type == `TYPEALUR && funccode == 6'b100111)? `ALUOP_NOR:
                     (((Type == `TYPEALUIMM && opcode == 6'b001010) || (Type == `TYPEALUR && funccode == 6'b101010) || (Type == 4'b0100 && opcode[5:1] != 5'b00010))? `ALUOP_STL:`ALUOP_SLTU))))));

assign Shifter_control = ({funccode[5:3],funccode[1:0]} == 5'b00000 && opcode == `SPECIAL)? `SHIFTLEFT:
                         (({funccode[5:3],funccode[1:0]} == 5'b00011 && opcode == `SPECIAL)? `SHIFTRIGHTA:
                         (({funccode[5:3],funccode[1:0]} == 5'b00010 && opcode == `SPECIAL)? `SHIFTRIGHT: `SHIFTRIGHT));

assign PC_4 = PC + 4;

assign RF_wen = (Type == `TYPEBRANCH || Type == `TYPESTORE || opcode == 6'b000010 || (Type == `TYPEJS && funccode == 6'b001000) || (Type == `TYPEMOVE && funccode[0] == 1'b1 && Zero == 1)
                || (Type == `TYPEMOVE && funccode[0] == 1'b0 && Zero == 0))? 1'b0: 1'b1;

assign RF_waddr = (opcode == 6'b000011)? 31:((RegDst)? rd:rt);//control which data to write

assign RF_wdata = (Type[3:1] == 3'b000)? ALU_result:
                  ((Type[3:1] == 3'b001)? shifter_result:
                  ((opcode == 6'b000011 || (Type == `TYPEJS && funccode == 6'b001001))? PC + 8:
                  ((Type == `TYPELOAD)? load_result:
                  ((Type == `TYPEMOVE)? rdata1:lui_extend))));

assign RF_raddr1 = rs;
assign RF_raddr2 = rt;
```

前面的信号在什么时候跳转在注释中均有说明，这里主要说明 ALU_control 信号和 Shifter_control 信号，以及 RF_wen，RF_waddr，RF_wdata 信号

ALU_control：这个信号在我的单周期处理器设计中算是最复杂的了，ALU 模块中可以实现 8 种运算，相对简单的是按位与，按位或，按位异或，按位与非，这些操作只会在 ALU R-Type 和 ALU I-Type 指令中出现，因此在这 45 条指令中，最多只有 2 条指令需要这些操作，找到这两条指令进行译码即可。加法操作用的地方很多，在 ALU R-Type 和 ALU I-Type 指令中有两条，load 和 store 指令也需要加法操作算 memory 的地址；减法操作用的地方就更多了，同样的在 ALU R-Type 指令中有一条，在 Branch 和 move 指令中为了判断是否满足跳转条件或转移条件也需要减法操作；slt 操作除了 ALU R-Type 和 ALU I-Type 指令中出现，也会用在 branch 指令中判断是否满足跳转条件。

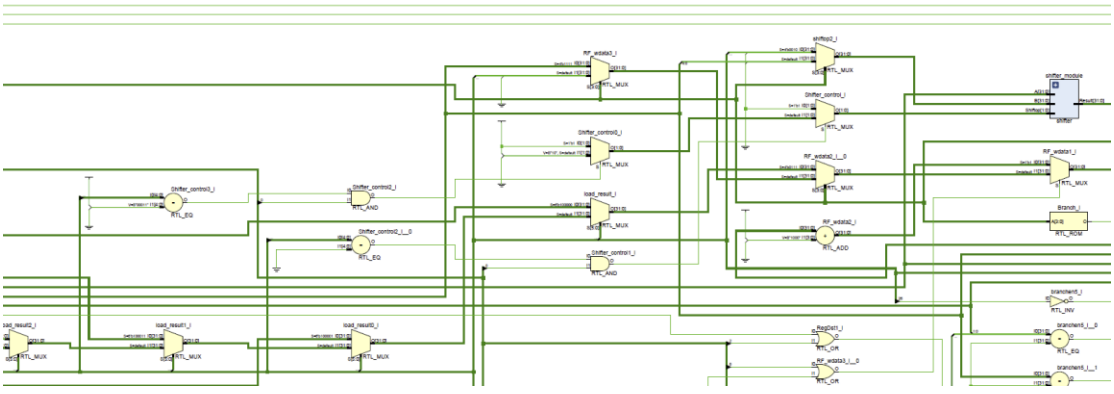


（ALU_control 在最后的原理图中线路较长）

```
assign ALU_control = (opcode[5:1] == `ADDI || (Type == `TYPEALUR && funccode == 6'b100001) || Type == `TYPELOAD || Type == `TYPESTORE)? `ALUOP_ADD:
                     (((Type == `TYPEALUR && funccode == 6'b100011) || (Type == 4'b0100 && opcode[5:1] == 5'b00010) || (Type == `TYPEMOVE))? `ALUOP_SUB:
                     (((Type == `TYPEALUIMM && opcode == 6'b001100) || (Type == `TYPEALUR && funccode == 6'b100100))? `ALUOP_AND:
                     (((Type == `TYPEALUIMM && opcode == 6'b001101) || (Type == `TYPEALUR && funccode == 6'b100101))? `ALUOP_OR:
                     (((Type == `TYPEALUIMM && opcode == 6'b001110) || (Type == `TYPEALUR && funccode == 6'b100110))? `ALUOP_XOR:
                     ((Type == `TYPEALUR && funccode == 6'b100111)? `ALUOP_NOR:
                     (((Type == `TYPEALUIMM && opcode == 6'b001010) || (Type == `TYPEALUR && funccode == 6'b101010) || (Type == 4'b0100 && opcode[5:1] != 5'b00010))? `ALUOP_STL:`ALUOP_SLTU))))));
```

（ALU_control 代码）

Shifter_control: 这个信号相比于 ALU_control 信号就简单了很多，因为移位指令要少的多也比较集中，对于这六条移位指令进行简单的分析便可得到译码（左移，右移，算术右移）。

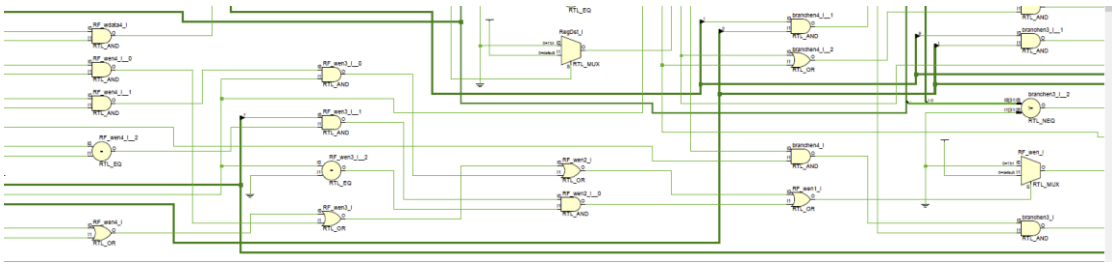


（含有 shifter_control 的原理图）

```
assign Shifter_control = ({funccode[5:3],funccode[1:0]} == 5'b00000 && opcode == `SPECIAL)? `SHIFTLEFT:
                         (({funccode[5:3],funccode[1:0]} == 5'b00011 && opcode == `SPECIAL)? `SHIFTRIGHTA:
                         (({funccode[5:3],funccode[1:0]} == 5'b00010 && opcode == `SPECIAL)? `SHIFTRIGHT: `SHIFTRIGHT));
```

（shifter_control 代码）

RF_wen：寄存器写使能信号，这个信号被拉高的时候比较多，因此我在处理这个信号的时候主要是看它什么时候应该被拉低。当时 Branch，Store，不向 31 号寄存器写数据的 J 指令，不满足 Move 条件的 move 指令时，寄存器堆是不写东西的，其余情况下寄存器堆都要有数据流入，具体的代码和原理图如下：



（含有 RF_wen 的部分）

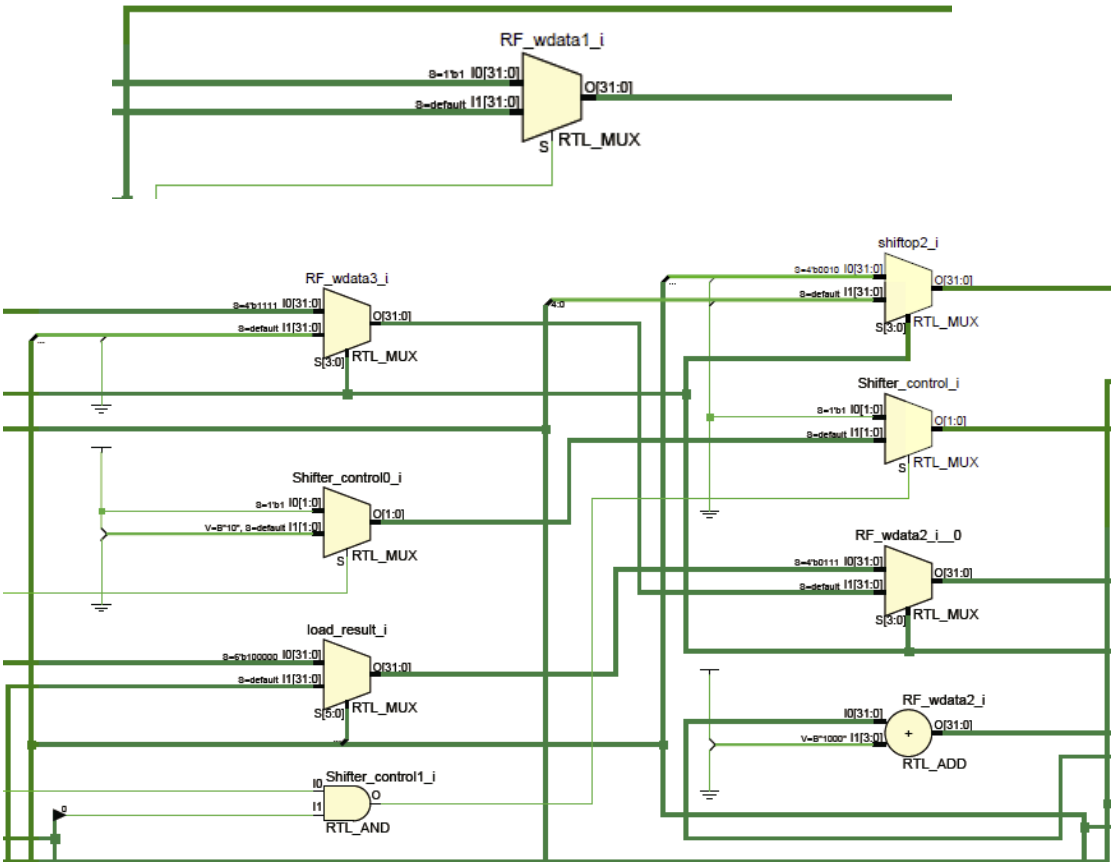
```
assign RF_wen = (Type == `TYPEBRANCH || Type == `TYPESTORE || opcode == 6'b000010 || (Type == `TYPEJS && funccode == 6'b001000) ||
                 (Type == `TYPEMOVE && funccode[0] == 1'b1 && Zero == 1)
                || (Type == `TYPEMOVE && funccode[0] == 1'b0 && Zero == 0))? 1'b0: 1'b1;
```

（含有 RF_wen 的代码）

RF_wdata：写入寄存器的数据分为六种情况，ALU 的结果（为 ALU R-Type 和 ALU I-Type 指令时），shifter 的结果（为 shifter 指令的时候），有一些 J-Type 指令需要把 PC+8 写入 31 号寄存器里，还有 load 型指令，move 型指令，以及 lui，具体的译码逻辑如下面代码所示：

```
assign RF_wdata = (Type[3:1] == 3'b000)? ALU_result:
                  ((Type[3:1] == 3'b001)? shifter_result:
                  ((opcode == 6'b000011 || (Type == `TYPEJS && funccode == 6'b001001))? PC + 8:
                  ((Type == `TYPELOAD)? load_result:
                  ((Type == `TYPEMOVE)? rdata1:lui_extend))));
```

(含有 RF_wdata 的代码)

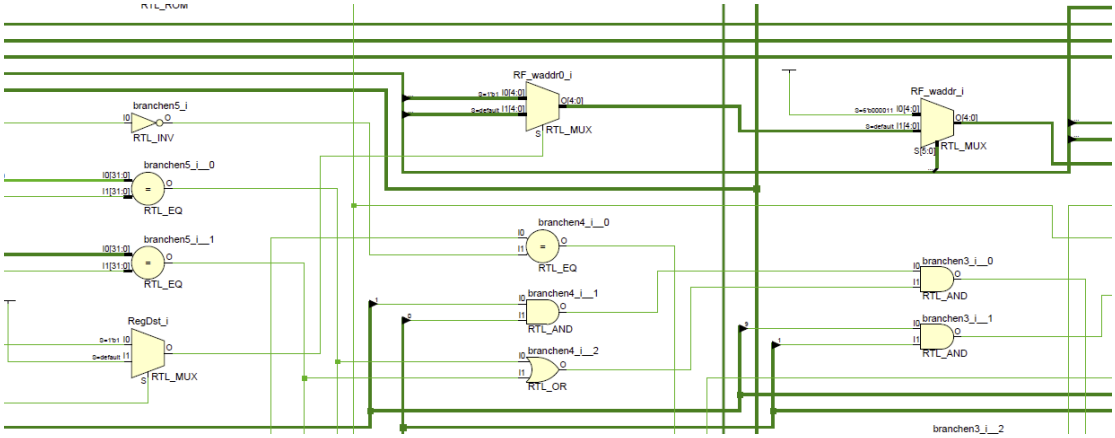


（RF_wdata 的一部分）

RF_waddr：这个比较好确定，只有三种情况，rd，rt，31，可以根据是不是那种特殊的 J-Type 指令确定是不是 31，然后根据 RegDst 确定时 rd 还是 rt，具体的译码逻辑见下：

```
assign RF_waddr = (opcode == 6'b000011)? 31:((RegDst)? rd:rt);//control which data to write
```

（RF_waddr 的译码逻辑）



（含有 RF_waddr 信号的原理图）

4. 扩展

这里的“扩展”，指的是 MIPS 指令集中对于指令中一些立即数的扩展，具体的 Verilog 代码如下：

```
assign sign_extend = {{16{Instruction[15]}},Instruction[15:0]};//sign_extend
assign zero_extend = {{16{1'b0}},Instruction[15:0]};//zero_extend
assign offset_two_extend = {{14{Instruction[15]}},Instruction[15:0],2'b0};//offset_two_extend
assign instr_two_extend = {PC_4[31:28],Instruction[25:0],2'b0};//instr_two_extend
assign lui_extend = {immediate,16'b0};//lui_extend
```

（“扩展”的 Verilog 代码）

具体的扩展方式比较简单，在此不再赘述，根据 MIPS 指令集中的说明进行扩展即可。

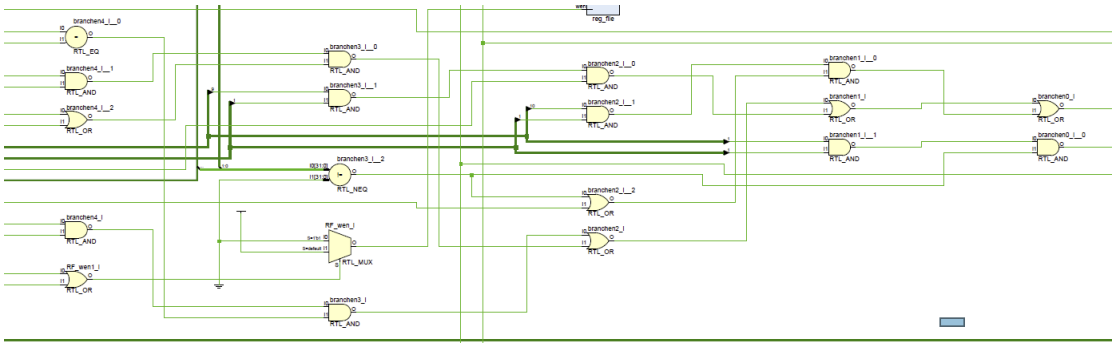
5. Branch 和 Jump

这两个指令有共同的地方，都是要修改 PC 寄存器的值，因此放在一起说。

首先要确定是否 branch，还需要一个 branchen 信号，这个信号什么时候被拉高时 branch 类型指令译码的一个难点，具体的译码逻辑对应的 verilog 代码如下：

```
assign branchen = ((Branch && opcode[5:1] == 5'b00010 && Zero == ~opcode[0]) ||
                   (opcode == 6'b000001 && rt == 5'b00001 && (ALU_result == 0 || rdata1 == 32'b0)) ||
                   (opcode == 6'b000111 && rt == 5'b00000 && ALU_result == 0) ||
                   (opcode == 6'b000110 && rt == 5'b00000 && (ALU_result != 0 || rdata1 == 32'b0))||
                   (opcode == 6'b000001 && rt == 5'b00000 && ALU_result != 0))? 1:0;
```

从代码中我们可以看出，第一行时“等于类型”，Branch 指令的判断，意思时 rs=rt 或者 rs 不等于 rt 的时候跳转，在之前 ALUop 译码的时候，我们知道这个部分 ALUop 使用的时 sub 信号，因此只需要判断相减是不是 0 即可。后面的是“大于小于等于 0”类型的 branch 指令，这里判断的是 ALU_result 是否为 0（因为这里的 ALUop 是 slt，等于 0 或者不等于 0 代表小于或者不小于）。同时考虑特殊情况，slt 只能处理小于 0，不能处理小于等于 0，因此等于 0 的时候需要单独拉出来判断。



（含有 branchen 信号的部分电路）

接下来是 branch 跳转地址的确定，这里需要调用 ALU，具体调用如下

```
alu alu_branch(
    .A(offset_two_extend),
    .B(PC_4),
    .ALUop(`ALUOP_ADD),
    .Result(branch_PC),
    .Overflow(),
    .CarryOut(),
    .Zero()
);
```

（branch_ALU 的调用）

PC_4 是 PC+4，根据课上所讲，得到的跳转地址应该是 PC+4+offset 的扩展，因此调用 ALU 如上图，最终得到 branch_PC，即 branch 指令的跳转

地址。

Jump 指令跳转地址的确定：Jump 指令可分为两种，一种的跳转指令直接存在立即数里，另一种的跳转指令存在寄存器堆里，分好后即可译码，verilog 代码如下：

```
assign Jumpaddr = (opcode[5:1] == 5'b00001)? instr_two_extend:rdata1;
```

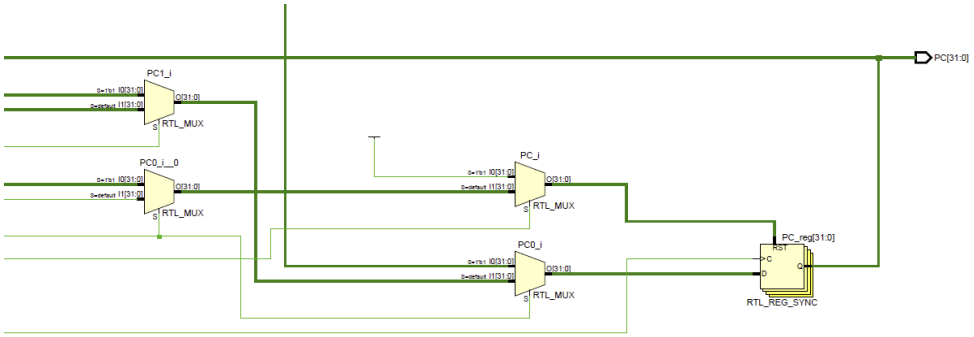
（确定 Jumpaddr 的 verilog 代码）

确定好是否跳转和跳转地址后，就是 PC 的赋值了，由于 PC 是一个寄存器，对其进行赋值需要时序逻辑，具体的 verilog 代码如下：

```
always @(posedge clk) begin
    if(rst) begin
        PC <= 32'b0;
    end else begin
        PC <= Jump? Jumpaddr:((Branch & branchen)? branch_PC: PC_4);
    end
end
```

（PC 赋值的时序逻辑）

当复位信号 rst 来临的时候，PC 置 0，如果复位信号没有来临就根据是 jump 还是 branch，是否满足 branch 指令跳转的要求来确定 PC 的下一个时钟周期的值。



（与 PC 赋值有关的原理图部分，采用 D 触发器）

6. ALU 和 Shifter 的调用

Shifter 的调用比较简单，下面为 shifter 操作数的定义：

```
wire [31:0] shifter_result;
wire [31:0] shiftop1 = rdata2;
wire [31:0] shiftop2 = (Type == `TYPESHIFT)? {{27{1'b0}},sa}: {{27{1'b0}},rdata1[4:0]};//sa or rt
```

我们可以看到，shiftop1 时钟等于 rt，shiftop2 根据两种不同的 shift 命令看是 sa 还是 rs。

ALU 的调用相对复杂，下面为 ALU 操作数的定义

```
assign aluop1 = (Type == 4'b1111)? rdata2:rdata1;
assign aluop2 = (ALUsrc)? extend:
                (((Type == `TYPEBRANCH && opcode[5:1]!= 5'b00010) || Type == `TYPEMOVE)? 32'b0:rdata2);
```

Aluop1 相对简单，如果为 MOVE 类型指令那么就是 rt，其他的情况都为 rs

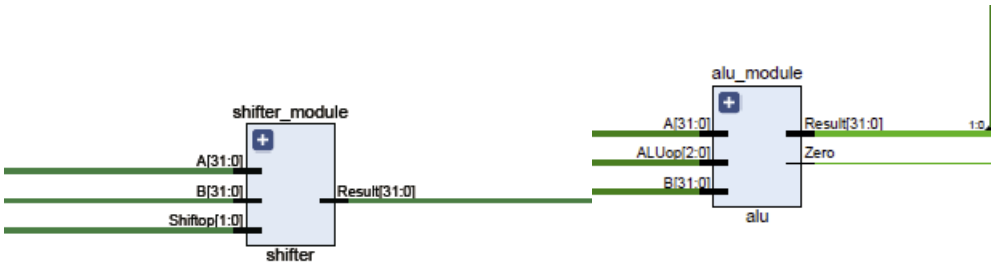
Aluop2 相对复杂，分为三种情况，先看 ALUsrc 控制信号是否拉高，拉高表明有效的是经过扩展的立即数，若没拉高可能我们理所当然地认为它的值就是 rt，但还有一种情况不能忽略，就是部分 branch 指令和 move 指令的条件是和 0 的比较，这时候 ALUop2 要赋值为 0，这种情况也不能忽略。

两者的调用在下面的代码中体现：

```
alu alu_module(
    .A(aluop1),
    .B(aluop2),
    .ALUop(ALU_control),
    .Result(ALU_result),
    .Overflow(),
    .CarryOut(),
    .Zero(Zero)
);
```

```
shifter shifter_module(
    .A(shiftop1),
    .B(shiftop2),
    .Shiftop(Shifter_control),
    .Result(shifter_result)
);
```

对应在原理图的部分如下：



7. Store 和 Load 指令

这两类指令应该是这四十五条指令相对复杂的两类指令了。但两类指令有高度的相似之处，所以报告里只介绍 Load 指令。

```
assign n = ALU_result[1:0];

assign lb_result = (n[1] & n[0])? {{24{Read_data[31]}},Read_data[31:24]}:
                   ((n[1] & ~n[0])? {{24{Read_data[23]}},Read_data[23:16]}:
                   ((~n[1] & n[0])? {{24{Read_data[15]}},Read_data[15:8]}:{{24{Read_data[7]}},Read_data[7:0]}));
assign lbu_result = {{24{1'b0}},lb_result[7:0]};
```

（LB 指令的 Verilog 代码）

我们先获取指令的最后两位，如果为 00，则读取对 Read_data 的最后一个字节进行符号位扩展，如果为 01，则对其 8-15 位进行符号位扩展，依次类推，lbu 的结果是只取 lb 结果的最后一个字节，高位全部补零。

LH 指令和 LHU 指令原理类似。

下面是两个比较麻烦的指令，lwl 和 lwr 指令：

这两个指令相对比较难理解，但是所幸 MIPS 指令集手册有两张非常好的示意图，如下（LWL）：

$vAddr_{1..0}$	Little-endian			
0	L	f	g	h
1	K	L	g	h
2	J	K	L	h
3	I	J	K	L

只需要根据这个表进行译码即可，非常的方便和易于理解，代码如下：

```
assign lwl_result = (n[1] & n[0])? Read_data[31:0]:
                    ((n[1] & ~n[0])? {Read_data[23:0],rdata2[7:0]}:
                    ((~n[1] & n[0])? {Read_data[15:0],rdata2[15:0]}:{Read_data[7:0],rdata2[23:0]}));
```

如图我们可以看出，当 n=3 的时候，Read_data 的全部数据都要读取，n=2 时，寄存器最后一位保留，高三位替换为 Read_data 的第三位，依次类推。Lwr 和此类似。

8. ALU 和 SHIFTER

ALU 做了一些改动，比较主要的是 sltu（无符号数的比较），具体改动代码如下：

```
assign {CarryOut,assresult} = A + ((ALUop[2] | op_sltu)? ~B:B) + (ALUop[2] | op_sltu);
```

```
assign slturesult = {{31{1'b0}},CarryOut};
```

首先，我们知道 sltu 的 ALUop 编码是 011，而这个运算也需要减操作，所以进行对 B 取补码的时候也要考虑这个因素。

如果无符号数比较，A<B，那么必然会产生借位使得 Carryout 为 1，所以 Carryout 成为了判断是否拉高 sltu 的标准。

Shifter 模块较简单，其 verilog 代码如下：

```
`timescale 10 ns / 1 ns

`define SHIFTLEFT   2'b00
`define SHIFTRIGHT  2'b10
`define SHIFTRIGHTA 2'b11

`define DATA_WIDTH 32

module shifter (
    input [`DATA_WIDTH - 1:0] A,
    input [`DATA_WIDTH - 1:0] B,
    input [1:0] Shiftop,
    output [`DATA_WIDTH - 1:0] Result
);

    // TODO: Please add your logic code here

    /*define my shiftop here*/
    wire op_left = Shiftop == `SHIFTLEFT;
    wire op_right = Shiftop == `SHIFTRIGHT;
    wire op_righta = Shiftop == `SHIFTRIGHTA;

    /*define my shifter logic there*/
    wire [63:0] extend = {{32{A[`DATA_WIDTH - 1]}},A};
    wire [63:0] extend_rightaresult = extend >> B;

    wire [31:0] leftresult = A << B;
    wire [31:0] rightresult = A >> B;

    wire [31:0] rightaresult = extend_rightaresult[31:0];


    assign Result = ({32{op_left}} & leftresult) |
                    ({32{op_right}} & rightresult) |
                    ({32{op_righta}} & rightaresult);


endmodule
```

为了处理算数右移，我先对 A 进行了符号位扩展，然后对扩展后的数进行逻辑右移最后取低位得到 A 的算术右移。

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）

1. 语法检查的时候发现有些信号未声明就直接使用，于是之后便把这些信号的声明和定义全部挪到了代码的最前面。

2. 仿真过程中，最麻烦我的便是 alu 模块里的 sltu 信号，刚开始对无符号数

的比较没有什么好的方法，比较结果经常出错，后来自己笔算进行尝试，发现 Carryout 和 sltu 的关系后进行更改，消除了 bug。

3. 关于算数右移，刚开始使用了“>>>”，后来发现会有出错，于是更改成了上部分所说的办法
4. 很多信号，比如说 RF_wen，aluop2 少考虑了一个或者两个情况，后来经过对着波形调试更改过来了
5. 云平台调试过程需要等待时间较长，而且还要顺着波形找到出错的点，这点我认为是调试过程中的难点。但是顺着指令流寻找也不算特别麻烦。

三、 在课后，你花费了大约 7 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）

这次实验相对于 prj1 要复杂很多，首先要花将近半天的时间阅读 MIPS 指令集手册并且给这些指令进行分类，并且记下他们的格式和操作码等信息等等。然后进行译码，刚开始很容易无从下手，只能根据理论课老师展示的单周期 cpu 完整的数据通路先搭好一个大致的框架再进行具体指令细节的完善。Debug 也相对困难了很多，需要找到指令找出出错的信号然后查找指令集手册看看为什么出错等等。但是通过这次实验也大大加深了我对单周期处理器的理解，我觉得这是大有裨益的。这次实验的参考资料相对也比较全，就是云平台仿真速度还是相对较慢，而且还时常发生连接不上

云平台的事情，希望今后能有所完善。