中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 宋俊仪 学号 2023K8009929018 专业 计算机科学与技术

实验项目编号 2 实验名称 单周期 CPU 设计

注 1:	撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下(注意:reports 全部小写)。文件命名规则:`prjN.pdf`,其中 `prj` 和后缀名 `pdf` 为小写,`N` 为 1 至 4 的阿拉伯数字。例如:`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:`prj5-projectname.pdf`,其中 "-" 为英文标点符号的短横线。文件命名举例:`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` 推送到实验课 SERVE 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、 设计单周期 CPU 的准备工作

- **逻辑电路的设计及初步分析**

  讲义中的逻辑电路设计如下:(见图 1)

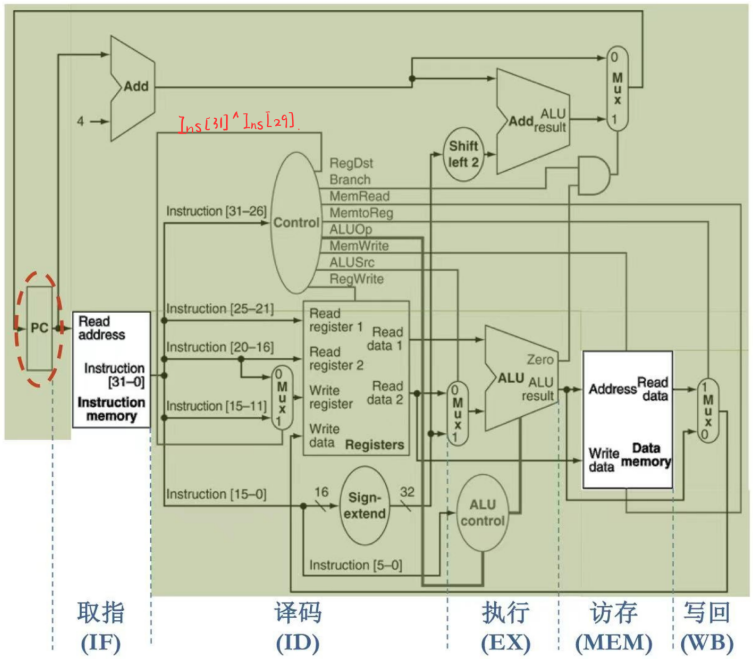

图 1: 讲义单周期 CPU 逻辑电路设计

对图 1 进行分析,设计思路如下:

1. **取指阶段 (IF)**

   - **PC (Program Counter - 程序计数器)**
     - **作用:** 存储当前正在执行指令的下一条指令的内存地址。CPU 总是根据 PC 的值去取指令。

- **工作方式:** 每个时钟周期结束后,PC 会更新为下一条指令的地址。通常是 `PC + 4`(因为 MIPS 指令是 32 位,即 4 字节,内存按字节编址),但在分支或跳转指令成功时,会被更新为目标地址。图中虚线框代表 PC 寄存器本身,它的输出连接到指令存储器的地址输入端。

- **Instruction Memory (指令存储器)**
  - **作用:** 存储程序的机器指令。
  - **工作方式:** 它是一个只读存储器(或行为类似只读)。接收来自 PC 的地址,输出该地址对应的 32 位指令 `Instruction [31-0]`。

- **Add (上方的加法器)**
  - **作用:** 计算顺序执行时的下一条指令地址。
  - **工作方式:** 将 PC 的输出值加 4。其结果是 `PC + 4`,是潜在的下一个 PC 值之一(另一个是分支/跳转目标地址)。

2. **译码阶段 (ID)**

- **Instruction Split (指令字段拆分)**
  - **作用:** 将从指令存储器读出的 32 位指令拆分成不同的字段,供后续单元使用。
  - **字段:**
    - `Instruction [31-26]` (Opcode): 送到主控制单元。
    - `Instruction [25-21]` (rs): 作为第一个源寄存器地址(Read register 1)。
    - `Instruction [20-16]` (rt): 作为第二个源寄存器地址(Read register 2),也可能作为目标寄存器地址(对于某些 I 型指令如 `lw`)。
    - `Instruction [15-11]` (rd): 作为 R 型指令的目标寄存器地址。
    - `Instruction [15-0]` (Immediate): 16 位立即数,送往符号扩展单元。
    - `Instruction [5-0]` (Funct): 功能码(对于 R 型指令),送到 ALU 控制单元。

- **Control (主控制单元)**
  - **作用:** CPU 的“大脑”。根据指令的操作码 `Instruction [31-26]` 生成控制整个数据通路行为的信号。
  - **输出信号示例:**
    - `RegDst`: 控制写入哪个目标寄存器(rt 还是 rd)。
    - `Branch`: 指示是否是分支指令。
    - `MemRead`: 控制数据存储器是否进行读操作。
    - `MemtoReg`: 控制写回寄存器的数据来源(ALU 结果还是内存数据)。
    - `ALUOp`: 发给 ALU 控制单元,指示 ALU 操作的大类。
    - `MemWrite`: 控制数据存储器是否进行写操作。
    - `ALUSrc`: 控制 ALU 的第二个操作数来源(寄存器数据还是立即数)。
    - `RegWrite`: 控制是否将结果写入寄存器堆。

- **Registers (寄存器堆)**
  - **作用:** 存储 CPU 的通用寄存器(MIPS 通常有 32 个)。提供快速的数据存取。
  - **接口:** 有两个读端口(`Read register 1`, `Read register 2` 指定地址,输出 `Read data 1`, `Read data 2`)和一个写端口(`Write register` 指定地址,`Write data` 写入数据,`RegWrite` 控制是否写入)。

- **工作方式:** 根据 rs 和 rt 字段指定的地址,并行读出两个寄存器的值。根据 RegWrite 信号和写地址、写数据,在时钟边沿写入数据。

- **Mux (目标寄存器地址选择)**
  - **作用:** 根据 RegDst 信号,选择写入寄存器堆的目标寄存器地址。
  - **工作方式:** 如果 RegDst=0 (例如 lw 指令),选择 rt (Instruction [20-16]) 作为写地址;如果 RegDst=1 (R 型指令),选择 rd (Instruction [15-11]) 作为写地址。
- **Sign-extend (符号扩展单元)**
  - **作用:** 将指令中的 16 位立即数 Instruction [15-0] 扩展为 32 位。
  - **工作方式:** 如果立即数是用于算术运算或地址偏移,需要进行符号扩展(将最高位复制到扩展出的高 16 位),以保持其表示的有符号数值不变。如果用于逻辑运算,则进行零扩展(高 16 位补 0)。图中通常指符号扩展。

3. **执行阶段 (EX)**

- **Mux (ALU 第二个操作数选择)**
  - **作用:** 根据 ALUSrc 信号,选择送入 ALU 的第二个操作数。
  - **工作方式:** 如果 ALUSrc=0 (R 型指令),选择从寄存器堆读出的 Read data 2;如果 ALUSrc=1 (I 型指令),选择经过符号扩展后的 32 位立即数。
- **ALU control (ALU 控制单元)**
  - **作用:** 根据主控制单元传来的 ALUOp 信号和指令的功能码 Instruction [5-0](仅对 R 型指令),生成具体的控制信号给 ALU,告诉 ALU 执行哪种运算(如加、减、与、或、置小于等)。
- **ALU (Arithmetic Logic Unit - 算术逻辑单元)**
  - **作用:** 执行算术运算(加、减)或逻辑运算(与、或、非等)。
  - **工作方式:** 接收两个 32 位操作数和 ALU 控制信号,输出 32 位运算结果 ALU result 和一个 Zero 标志位。Zero 标志位在结果为 0 时置 1,用于判断分支条件(如 beq 指令)。
- **Shift left 2 (左移 2 位单元)**
  - **作用:** 将符号扩展后的立即数(通常用作分支偏移量)左移两位。
  - **工作方式:** 相当于乘以 4。因为分支指令的偏移量是相对于 PC+4 的字数,而 PC 是按字节增加的,所以要乘以 4 转换为字节偏移量。
- **Add (分支目标地址加法器)**
  - **作用:** 计算分支指令跳转的目标地址。
  - **工作方式:** 将 PC+4 (来自 IF 阶段的加法器) 与左移两位后的立即数相加。
- **AND Gate (与门)**
  - **作用:** 判断分支条件是否满足(特指 beq 指令)。
  - **工作方式:** 当主控制单元发出 Branch 信号(表示当前是分支指令)且 ALU 的 Zero 标志位为 1(表示比较的两个寄存器相等)时,与门输出 1,表示分支条件满足,需要跳转。这个信号会控制更新 PC 值的 Mux(图中未完全画出 PC 更新逻辑)。

4. **访存阶段 (MEM)**

- **Data Memory (数据存储器)**
  - **作用:** 存储程序运行过程中需要读写的数据。

- **工作方式:**
  - **地址 (Address):** 对于 `lw` 和 `sw` 指令,地址由 ALU 计算得到(基址 + 偏移量)。
  - **写数据 (Write data):** 对于 `sw` 指令,要写入的数据来自寄存器堆的 `Read data 2` 输出。
  - **控制信号:** `MemRead` 为 1 时执行读操作,`MemWrite` 为 1 时执行写操作(两者通常互斥)。
  - **读数据 (Read data):** 对于 `lw` 指令,从存储器读出的数据会送到下一阶段的 Mux。

5. **写回阶段 (WB)**

- **Mux (写回数据选择)**
  - **作用:** 根据 `MemtoReg` 信号,选择最终要写回寄存器堆的数据。
  - **工作方式:** 如果 `MemtoReg=0` (R 型或 I 型算术/逻辑指令),选择 ALU 的运算结果 `ALU result`;如果 `MemtoReg=1` (`lw` 指令),选择从数据存储器读出的 `Read data`。
- **Data Path Back to Registers (写回数据通路)**
  - **作用:** 将选定的结果数据通过数据通路送回到寄存器堆的 `Write data` 输入端口。配合正确的 `Write register` 地址和有效的 `RegWrite` 信号,在时钟边沿将结果写入目标寄存器。

根据讲义中的逻辑电路设计,结合实验要求,完善设计思路,设计出我自己的 cpu 逻辑电路,主要是对 PC 模块做了一些修改。(见图 2)

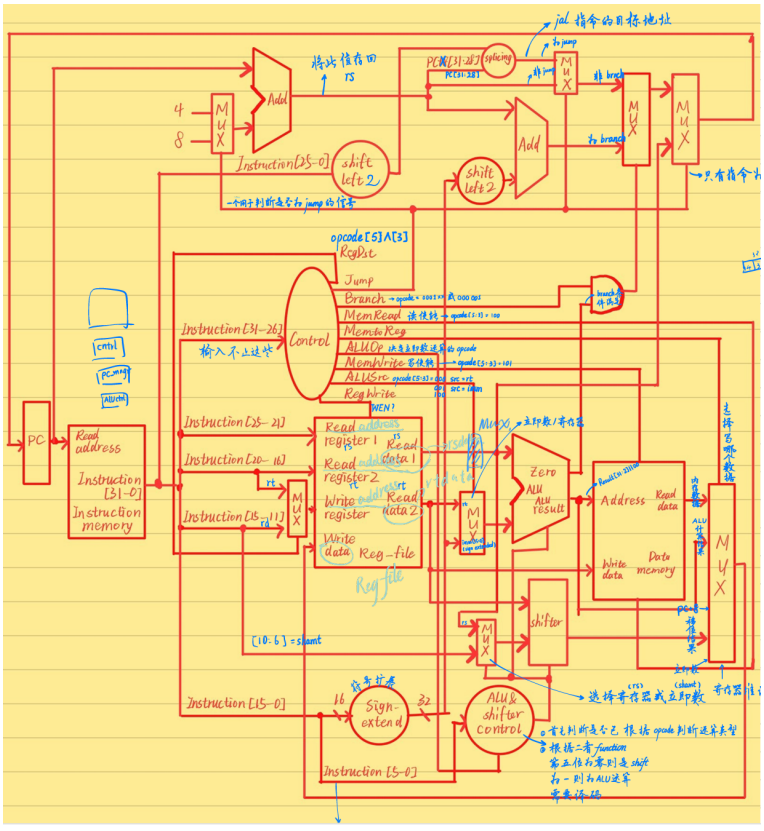

图 2: 改进后单周期 CPU 逻辑电路设计

1. 因为 MIPS 指令中跳转指令有 jump 和 branch 两种,所以我在 PC 模块中添加了两个选择器,在图中也有所体现,可以根据 instruction 产生的 jump 和 branch 信号来控制 PC 的操作。

2. 原讲义中的设计，只有 PC=PC+4（无跳转）或者是 PC=PC+4+ 立即数（有跳转），但是在 j 型指令中，我们还需要将 PC+8 的值，存入寄存器中，因此我还计算了 PC+8，并将它接在最后 WB 阶段的 MUX 上。

3. 最后的 MUX 选择器，原讲义中只有 ALU 计算结果和内存访问结果两种选择，但是这并不全面于是在我设计的电路图中（蓝色部分），我还完善了最后的选择，它还有可能是位移指令计算结果 PC+8 的值，以及立即数等。

- **学习 MIPS 指令集，完成译码表的绘制。**

1. 由于英文指令集手册阅读起来比较麻烦，所以我参考了中文的指令集手册以及向 AI 询问学习了相关指令的含义。

2. 我将指令按照其功能进行了划分，例如运算类指令、跳转类指令、访存类指令等。这样划分的好处是，同一类操作的指令可以放在一起，便于理解和记忆，也便于纵向对比。都是也有一些不够完美的地方，因为我没有按照指令类型进行划分，如 R 型指令、I 型指令、J 型指令等。这对之后多周期的实现产生了一些麻烦，因为有些指令在多周期中需要分开处理。

3. 我将输入的 instruction 分成了几个部分，更加直观的反映了不同指令的指令码之间的异同，为后续产生相应的控制信号提供了便利。

译码表如下：（见图 3）

指令分类	序号	指令名	指令码								指令含义	具体解释	imm扩展
			opcode	rs	rt	rd	shamt	func	imm	tar			
运算指令	1	addu	000 000	rs 5bit操作数	rt 5bit操作数	rd 5bit结果	00000	100 001			无符号数加	rs与rt运算--->rd	
	2	subu						100 011			无符号数减		
	3	and						100 100			按位与		
	4	nor						100 111			按位同或		
	5	or						100 101			按位或		
	6	xor						100 110			按位异或		
	7	slt						101 010			有符号数比较		
	8	sltu						101 011			无符号数比较		
	9	addiu	001 001	5bit操作数	5bit结果				16bit立即数		无符号数加	rs与立即数 运算--->rd	SE
	10	andi	001 100								按位与		0E
	11	ori	001 101								按位或		0E
	12	xori	001 110								按位异或		0E
	13	slti	001 010								有符号数比较		SE
	14	sltiu	001 011								无符号数比较		SE
移位指令	15	sll	000000	00000	rt 5bit操作数	rd 5bit结果	5bit操作数	000 000			左移	把rt中的数 移shamt位 --->rd	
	16	sra						000 011			算术右移		
	17	srl						000 010			逻辑右移		
	18	sllv		rs 5bit操作数			00000	000 100			左移	把rt中的数 移rs（低5位）位 --->rd	
	19	srav						000 111			算术右移		
	20	srlv						000 110			逻辑右移		
跳转指令	21	beq	000 100	rs 5bit	rt 5bit				16bit立即数offset		等于时跳转	当前指令的下一条指令地址+offset（有延迟槽）	SE|00
	22	bne	000 101		00000						不等于时跳转		
	23	blez	000 110								<=0跳转		
	24	bgtz	000 111		00000						>0跳转		
	25	bltz	000 001								<0跳转		
	26	bgez			00001						>=0跳转		
	27	j	000 010							26bit操作数	无条件跳转	跳到tar指令地址	
	28	jal	000 011								保存位置跳转	跳到tar指令地址，并把原来的PC+8写入11111寄存器	
	29	jr	000 000	rs 5bit	00000	00000	5bit操作数	001 000			跳转回	跳转到rs的值的指令地址	
	30	jalr				rd 5bit		001 001			保存位置跳转	跳转到rs的值的指令地址，并把PC+8写入rd	
访存指令	31	lb	100 000	rs 5bit	rt 5bit				16bit立即数		读有符号数1B	Result=rs+imm,Result<<2作为内存地址，rt为要写入的寄存器地址要写入rt的值，较为复杂，由Result=rs+imm对齐要求，写入不同的Read_data值	SE
	32	lbu	100 100								读无符号数1B		
	33	lh	100 001								读有符号数2B		
	34	lhu	100 101								读无符号数2B		
	35	lw	100 011								读有符号数4B		
	36	lwl	100 010								读无符号数（左对齐）		
	37	lwr	100 110								读无符号数（右对齐）		
	38	sb	101 000								存1B数据	Result=rs+imm,Result<<2作为内存地址，rt为要写入的内存的数值具体要写入rt的那一部分，也需要根据Result=rs+imm的对齐情况，进行不同的选择	SE
	39	sh	101 001								存2B数据		
	40	sw	101 011								存4B数据		
	41	swl	101 010								存4B数据（左对齐）		
	42	swr	101 110								存4B数据（右对齐）		
数据移动指令	43	movn	000 000	rs	rt	rd	00000	001 011				当rt的值不是0(用rt和0与)，那么把rs的值写到rd	
	44	movz						001 010				当是0，进行上面的操作	
	45	lui	001 111	00000	rt				16bit立即数			把立即数左移16位，放入rt	

图 3: 译码表

二、 各模块实现及完整代码分析

- **ALU 模块的完善**

根据 45 条 MIPS 指令的要求，ALU 模块需要实现加法、减法、按位与、按位或、按位异或，有符号数和无符号数的比较操作。因此，在原来的 ALU 模块基础上，我增加了一些信号：

1. 增加了原来没有的无符号比较和按位同或的运算信号（见图 4）

2. 增加了无符号比较和按位同或等运算的逻辑（见图 5）

```
module alu(
    input  [31:0]  A,
    input  [31:0]  B,
    input  [2: 0]  ALUop,
    output         Zero,
    output [31:0]  Result

);
    wire [31:0]    and_result;
    wire [31:0]    or_result;
    wire [32:0]    add_result;
    wire [31:0]    xor_result;
    wire [31:0]    nor_result;
    wire           slt_result;
    wire           sltu_result;
    wire           Overflow;
    wire           CarryOut;
```

图 4: ALU 模块–增加的信号

```
assign and_result  = A&B;
assign or_result   = A|B;
assign xor_result  = A^B;
assign nor_result  = A~^B;
assign add_result  = a+b+(ALUop[2]|ALUop[0]);
assign slt_result  = (A[31] == 1'b1 && B[31] == 1'b0) ? 1'b1 :
                     (A[31] == 1'b0 && B[31] == 1'b1) ? 1'b0 :
                     (add_result[31] == 1'b1)         ? 1'b1 :
                                                        1'b0 ;
assign sltu_result = add_result[32] == 1'b1           ? 1'b1 :
                                                        1'b0 ;

assign Result = (ALUop == 3'b000)                 ? and_result                    :
                (ALUop == 3'b001)                 ? or_result                     :
                (ALUop == 3'b010||ALUop == 3'b110) ? add_result[31:0]             :
                (ALUop == 3'b100)                 ? xor_result                    :
                (ALUop == 3'b101)                 ? nor_result                    :
                (ALUop == 3'b111)                 ? (slt_result ? 32'b1 : 32'b0)  :
                                                    (sltu_result ? 32'b1 : 32'b0) ;
assign Zero = (Result == 32'b0) ? 1'b1 : 1'b0;
```

图 5: ALU 模块–增加的运算逻辑

- **移位运算模块设计。**

  设计 shifter 时遇到了以下的一些问题:(主要是 verilog 语法的问题,本身移位运算是很好理解的)

1. 左移有直接的逻辑运算符 «,但是右移有两种方式,分别是算术右移和逻辑右移。逻辑右移是直接使用 »,而算数右移需要使用一个组合逻辑电路来实现。

2. 在算数右移中，直接使用位拼接，是我一开始就想到的最简单的实现方式。但是在 verilog 中，使用位拼接时，需要拼接的符号位的位数是不确定的，是一个变量。在 system verilog 中，拼接的位数是可以是一个动态的，不确定的变量，但是在 verilog 中，拼接的位数是一个常量，所以我需要使用一个组合逻辑电路来实现。这个语法问题困扰了我很久，之后借助 AI，我学习了解到了这个细节。因此，我改用了一个组合逻辑电路来实现算数右移。

3. 具体的 shifter 模块设计如下:（见图 6）

```
module shifter (
    input  [`DATA_WIDTH - 1:0] A,      // 操作数
    input  [4:0] B,                     // 要移动的位数
    input  [1:0] Shiftop,               // 移位操作类型：00 左移，11 算术右移，10 逻辑右移
    output [`DATA_WIDTH - 1:0] Result   // 移位结果
);
    wire [`DATA_WIDTH - 1:0] sign = {`DATA_WIDTH{A[`DATA_WIDTH - 1]}};
    wire [`DATA_WIDTH - 1:0] lr;
    assign lr = A >> B;
    wire [`DATA_WIDTH - 1:0] ar;
    wire [`DATA_WIDTH - 1:0] ar_sign;
    assign ar_sign = sign << (`DATA_WIDTH - B);
    assign ar = ar_sign | lr ;
    // 使用 assign 和条件运算符 ?: 实现不同类型的移位
    assign Result = (Shiftop == 2'b00) ? (A << B) :                 // 左移
                    (Shiftop == 2'b11) ? ar       :                 // 算术右移
                    (Shiftop == 2'b10) ? lr       :                 // 逻辑右移
                                         `DATA_WIDTH'b0;            // 默认情况

endmodule
```

图 6: shifter 模块设计代码

- **完整代码实现 CPU**

按照我自己设计的译码表，分析实现这 45 条 MPIPS 指令需要用到哪些变量，哪些模块，哪些控制信号等。然后在代码中实现这些变量和模块。

1. 根据译码表，设计了一个控制信号的模块，控制信号的模块是一个组合逻辑电路，输入是指令的操作码，输出是控制信号。具体的控制信号的作用，见代码注释:(见图 7)

```
//会用到的instruction的各段
wire [5:0] Opcode = Instruction[31:26];
wire [4:0] rs     = Instruction[25:21];
wire [4:0] rt     = Instruction[20:16];
wire [4:0] rd     = Instruction[15:11];
wire [5:0] shamt  = Instruction[10: 6];
wire [6:0] func   = Instruction[ 5: 0];
wire [15:0]imm    = Instruction[15: 0];
wire [25:0]tar    = Instruction[25: 0];
//会用到的控制信号
wire          RegDst;          //选择读写寄存器的地址
wire          jump;
wire          branch;
wire          move;            //判断是不是move指令
wire [2:0]    ALUSrc;          //选择寄存器还是立即数，读入alu计算,高两位判断alu操作数1的选择，低两位判断alu操作数2的选择
wire [31:0]   rsdata;
wire [31:0]   rtdata;
wire [31:0]   rddata;          //移位结果
wire [4:0]    shiftlength;     //移位位数
wire          zero;
wire [31:0]   Result;          //alu结果
wire [31:0]   imm_32;          //拓展到32位的立即数 即imm + shifter -----> imm_32
```

图 7: 定义控制信号代码

2. 设计了一个 ALU 控制信号的模块,即 ALUop 的计算模块。ALU 控制信号的模块是一个组合逻辑电路,输入是指令的操作码和功能码,输出是 ALUop 信号:(见图 8)

```
assign ALUop = (Opcode == 6'b000000) ? (
               (func   == 6'b100001) ? 3'b010 :
               (func   == 6'b100011) ? 3'b110 :
               (func   == 6'b100100) ? 3'b000 :
               (func   == 6'b100111) ? 3'b101 :
               (func   == 6'b100101) ? 3'b001 :
               (func   == 6'b100110) ? 3'b100 :
               (func   == 6'b101010) ? 3'b111 :
               (func   == 6'b101011) ? 3'b011 :
                                       3'b001):
               (Opcode == 6'b000001) ? 3'b111 :
               (Opcode[5:3] == 3'b000)? 3'b110:
               (Opcode[5:3] == 3'b101 || Opcode[5:3] == 3'b100)? 3'b010 :
               (Opcode == 6'b001001) ? 3'b010 :
               (Opcode == 6'b001100) ? 3'b000 :
               (Opcode == 6'b001101) ? 3'b001 :
               (Opcode == 6'b001110) ? 3'b100 :
               (Opcode == 6'b001010) ? 3'b111 :
                                       3'b011 ;
```

图 8: ALUop 计算模块代码

3. 为了使代码可读性更强,避免后续使用不同的 imm 时产生混乱,于是我干脆定义了一个 imm 的模块,通过我的译码表,我分析出了之后会用到哪些 imm 的变形,如符号扩展,零扩展,左移 2 位等。于是我定义了一个 imm 模块,输入是指令的操作码和功能码,输出是 imm 的值。具体的 imm 模块代码如下:(见图 9)

```
//imm_32的计算
    wire [31:0] imm_32_0E;
    wire [31:0] imm_32_SE;
    wire [31:0] imm_32_SE00;//用于跳转指令的imm(SE)||00
    wire [31:0] imm_32_1600;//用于lui指令
    wire        imm_sign = imm[15];

    assign imm_32_SE00 = {{14{imm_sign}},imm,2'b00};
    assign imm_32_0E   = {{16{1'b0}},imm} ;
    assign imm_32_SE   = {{16{imm_sign}},imm} ;
    assign imm_32_1600 = {imm,{16{1'b0}}};

    assign imm_32 = (Opcode == 6'b001100 || Opcode == 6'b001101 || Opcode == 6'b001110) ? imm_32_0E  :
                    Opcode[5:3] == 3'b000                                               ? imm_32_SE00:
                    Opcode[5:0] == 6'b001111                                            ? imm_32_1600:
                                                                                          imm_32_SE  ;
```

图 9: imm 模块代码

4. 在第一步中，我画出了 CPU 的逻辑电路图，其中和讲义中的设计有一些不同之处，而 RF_wdata 的选择也就是最后的 MUX 选择器的选择。根据我的设计，RF_wdata 的选择有五种可能性，分别是 ALU 计算结果，内存访问结果，位移指令计算结果和 PC+8 以及 imm32 的值。具体的 RF_wdata 选择代码如下：（见图 10）

```
//RF_wdata的选择
    assign RF_wdata = Opcode == 6'b0 ? (
                      func   == 6'b001000 || func == 6'b001001 ? PC+8    :
                      func   == 6'b001011 || func == 6'b001010 ? rsdata  :
                      func[5]== 1'b0                           ? rddata  :
                                                                 Result) :
                      Opcode      == 6'b001111 ? imm_32     :
                      Opcode[5:3] == 3'b001    ? Result     :
                      Opcode[5:3] == 3'b100    ? mem_data   :  PC+8       ;
```

图 10: RF_wdata 选择代码

5. 在内存访存指令中，又一个很重要的信号输出，那就是写掩码 strb 信号。这个信号是一个组合逻辑电路，输入是指令的操作码和功能码，输出是写掩码 strb 信号。具体的 strb 信号输出代码如下：（见图 11）

```
wire [3:0] strb_1;
assign strb_1[0] = (Result[1:0] == 2'b00);
assign strb_1[1] = (Result[1:0] == 2'b01);
assign strb_1[2] = (Result[1:0] == 2'b10);
assign strb_1[3] = (Result[1:0] == 2'b11);
wire [3:0] strb_2;
assign strb_2[1:0] = {2{(Result[1] == 1'b0)}};
assign strb_2[3:2] = {2{(Result[1] == 1'b1)}};
wire [3:0] strb_3 = 4'b1111;
wire [3:0] strb_4;
assign strb_4 = 4'b1111 >> (3-Result[1:0]);
wire [3:0] strb_5;
assign strb_5 = 4'b1111 << Result[1:0];
assign Write_strb = Opcode[2:0] == 3'b000 ? strb_1 :
                    Opcode[2:0] == 3'b001 ? strb_2 :
                    Opcode[2:0] == 3'b011 ? strb_3 :
                    Opcode[2:0] == 3'b010 ? strb_4 :
                                            strb_5 ;
```

图 11: strb 信号输出代码

6. 在设计 PC 模块时，为了可以实现时序逻辑电路，我定义了一个 reg 型的变量 PC_reg，表示当前的 PC 值。这个变量在时钟上升沿时更新为 PC+4 或者是 next_PC 的值，这里的 next_PC 是通过指令码计算而来的，也就是 branch 指令和 jump 指令时，跳转到的 PC 位置。具体的 PC 模块代码如下：（见图 12）

```
reg [31:0] reg_pc;
always @(posedge clk) begin
    if(rst) begin
        reg_pc <= 32'b0;
    end
    else begin
        if(is_jorb) begin
            reg_pc <= next_pc;
        end
        else begin
            reg_pc <= pc_add4;
        end
    end
end
assign PC = reg_pc;
```

图 12: PC 模块代码

三、 实验中的难点及 debug 过程

- **逻辑过于复杂**

在实验中，有很多指令的实现是比较复杂的，尤其是访存类指令和跳转类指令。因为这两类指令需要对 PC 进行修改，而 PC 的修改又会影响到其他指令的执行。而在计算这些指令的一些变量时，如果全部用指令码来进行最原始化的逻辑运算，很容易就会把自己绕进去，最后 debug 时，再看自己的代码也会觉得很复杂。

因此，我在设计时，尽量将指令码进行拆分，分成几个部分进行逻辑运算，这样可以避免一些不必要的麻烦。例如：

1. 跳转到新的 PC 时，计算这个 next_PC 需要用到不同的立即数，有零拓展，有符号拓展，有左移 2 位等。所以我就先单独出来一个模块进行这方面的计算，之后根据指令码选择不同类型的 imm，这样只 = 之后我在使用 imm 时只需要用到 imm 这个变量就可以了。

2. 在计算 PC_reg 时，是否要跳转，需要用到指令码来进行判断，但是在 always 模块中，写很长的 if 语句会让人觉得很复杂。所以我就将指令码拆分成几个部分，在写 always 模块前，就先行计算出了 branch 和 jump 信号用来判断是否是跳转指令并生成 is_jump 信号，最终在 always 模块中我就只需要使用到 is_jump 信号即可，这样可以避免一些不必要的麻烦。具体的代码如下：（见图 13、图 14）

```
//branch计算
    assign branch = Opcode[5:2] == 1'h1 | Opcode == 6'b000001;

//jump计算
    assign jump = Opcode == 6'b000010 || Opcode == 6'b000011 || (Opcode == 6'b000000 && (func == 6'b001000 || func == 6'b001001));
```

图 13: branch 和 jump 信号计算代码

```
wire        is_jorb;    //记录是否跳转
wire [31:0] next_pc;    //跳转地址
assign is_jorb = jump   ? 1 :
                 branch ? (
                     Opcode[2:1] == 2'b10 ? Opcode[0]^zero               :
                     Opcode[2:1] == 2'b11 ? Opcode[0]^(Result[31]||zero) :
                                            rt[0]^(Result == 1'b1)
                 ): 0;
```

图 14: is_jump 计算模块代码

- **对齐问题**

在实现访存类指令时，内存的对齐问题是一个很重要的问题。因为在 MIPS 中，内存是按字节对齐的，而访存类指令需要访问的是一个字（4 个字节），所以在计算地址时，需要将地址进行对齐。尤其是 lb lwr 那 7 条指令，它们需要将内存中的数写入到寄存器中，而这个数的大小是一个字节，所以在计算地址时，需要将地址进行对齐。而对齐方式我最开始就理解错误了，经过本地运行后，根据报错我发现了这里的问题。最后我画图理解了是如何写入。以 lb 为例（见图 15、图 16）以 lwl 为例（见图 17、图 18）

```
assign lb_mem_data = Result[1:0] == 2'b00 ? {{24{Read_data[7]}},Read_data[7:0]}    :
                     Result[1:0] == 2'b01 ? {{24{Read_data[15]}},Read_data[15:8]}   :
                     Result[1:0] == 2'b10 ? {{24{Read_data[23]}},Read_data[23:16]}  :
                                            {{24{Read_data[31]}},Read_data[31:24]}  ;
```

图 15: lb 指令对齐方式代码实现

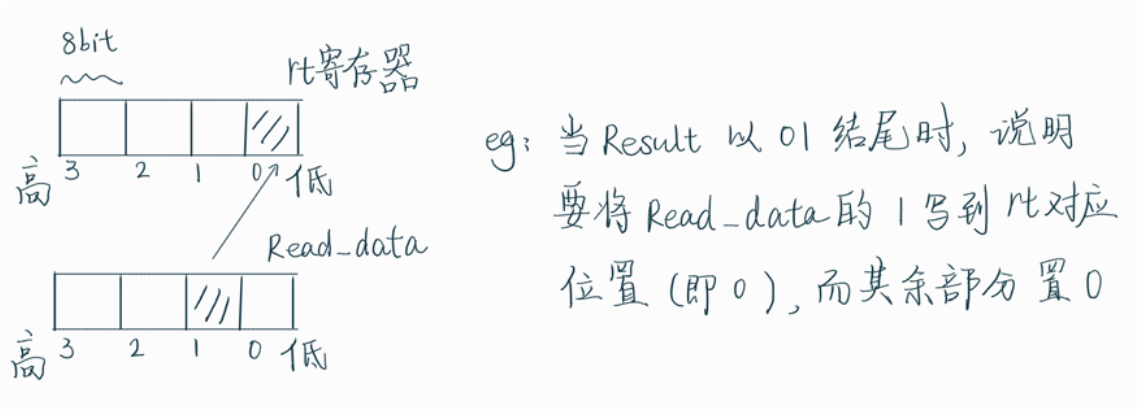

图 16: lb 指令对齐方式图示

```
assign lwl_data = Result[1:0] == 2'b00  ? {Read_data[7:0],rtdata[23:0] }    :
                  Result[1:0] == 2'b01  ? {Read_data[15:0],rtdata[15:0]}    :
                  Result[1:0] == 2'b10  ? {Read_data[23:0],rtdata[7:0] }    :
                                          Read_data                         ;
```

图 17: lwl 指令对齐方式代码实现

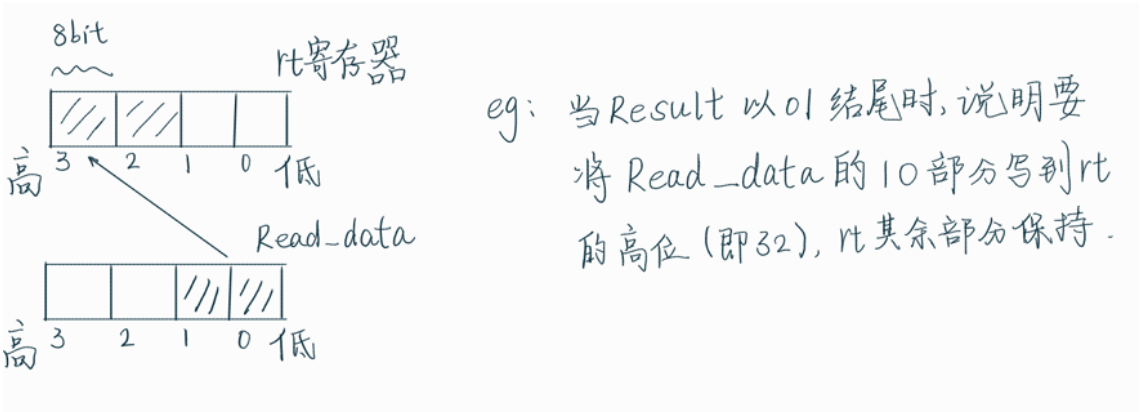

图 18: lwl 指令对齐方式图示

- **debug 过程**

在 debug 初期，最先遇到的一定是语法错误，虽然语法错误是最容易解决的，只要仔细检查代码就可以了，但是我们的平台上不会显示语法错误提示，我们只能通过本地运行来查看语法错误。但是这样效率太低，于是我想到了一个好办法就是使用以前使用过的 vivado 平台，它可以显示语法错误提示。（这里没有代码试例，因为我 debug 时忘记了留档，现在的代码已经是最终版）通过 vivado 的语法检查，我发现了很多语法错误，大致有以下几类：

1. 疏漏错误：例如忘记最后的分号，或者是 if 语句的 else 部分没有写等。
2. 拼写错误：例如变量名拼写错误，或者是模块名拼写错误等。
3. 括号错误：例如 if 语句的括号没有配对，或者是模块的括号没有配对等。
4. 汉语符号错误：例如中文的逗号，句号等。

5. 语法使用错误：例如使用了不支持的语法，或者是使用了不支持的模块等。（例如前面我写到的位拼接不支持变长度拼接）

举例说明：使用本地测试后的波形图进行 debug

1. 先根据测试用例，运行代码，查看报错信息，下载波形图，并根据报错信息定位波形图时刻（见图 19，图 20）

```
================================================
ERROR: at                    410ns.
Yours:     PC = 0x00000068
Reference: PC = 0x0000007c
Please check assignment of PC at previous cycle.
------------------------------------------------
```

图 19: 报错信息

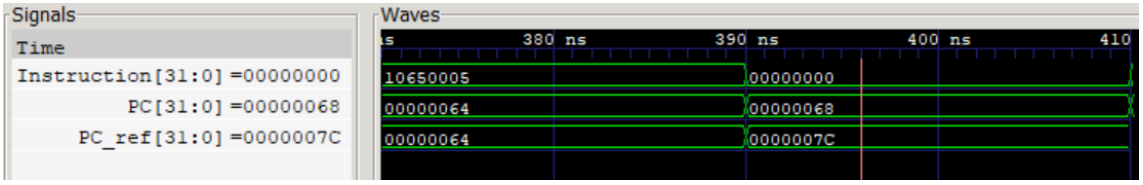

图 20: 波形图

2. 最初我以为出错的指令是当前时刻指令，但是经过分析后发现，出错的指令是前一时刻的指令。（只有和 PC 有关的指令是这样的，因为 PC 更新是一个时序电路，要下一个时钟周期上升沿更新）所以我可以找到前一时刻指令并进行分析（见图 21）

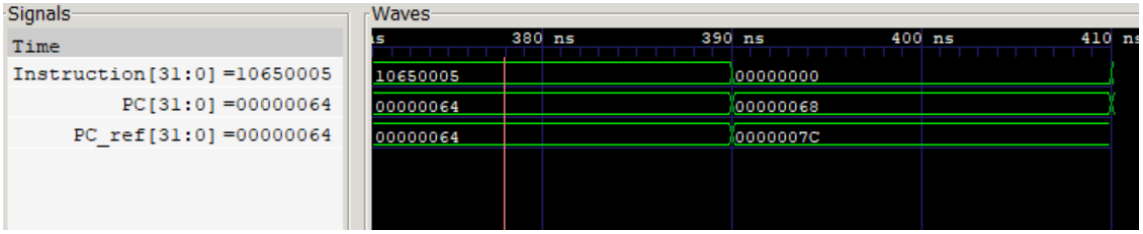

图 21: 波形图分析

3. 于是我得到了出错指令为：10650005；再借助 MIPS Converter 工具，将其转换为汇编指令，得到出错指令为：（见图 22）

BEQ $v1 $a1 0x0005

Binary: 00010000011001010000000000000101

Hex: 0x10650005

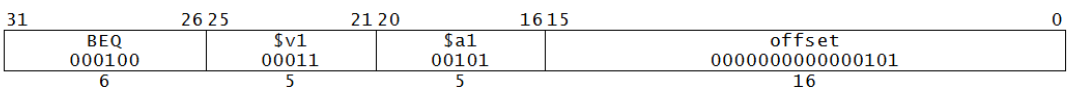

图 22: 指令分析

4. 经过分析，发现此处是一个 branch 型指令，而且经过对比 PC_ref 和我的 PC 的值，再结合指令码分析，可以得出这里出错的原因是我并没有完成跳转，而是进行了 PC+4 的操作。而跳转是否进行又由 is_jump 信号控制，于是我在波形图中找到了 is_jump 信号的值，发现它的值是 X（见图 23）

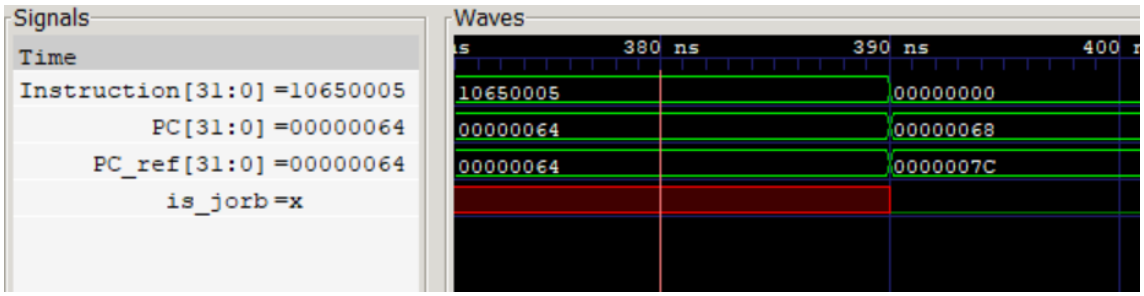

图 23: is_jump 信号值

5. 再返回代码页面进行分析，找出我的 is_jump 信息是由哪些变量产生的，再找出它们的波形图查看是否出错，经过分析发现是 zero 信号出错（见图 24）

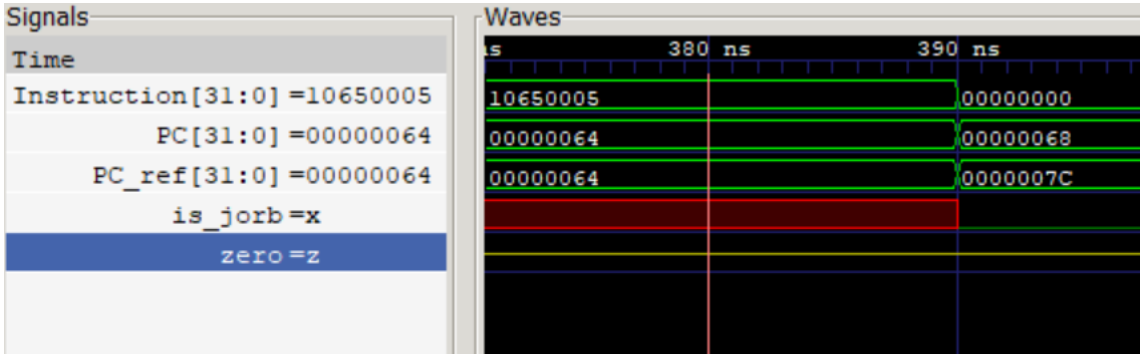

图 24: zero 信号值

6. zero 信号是 ALU 模块的输出信号，所以定位到应该是 ALU 模块出错，经过检查，是因为之前完善 ALU 时，忘记了 Zero 是输出没有写 Zero 的计算逻辑，没有产生 Zero 输出，改正后成功运行

四、 思考题

- **ALUop 的编码有什么规律？表格中的 ALUop 编码是否还有优化空间？**

我们首先回顾表格中 R-Type 指令的 `func` 字段和对应的 `ALUop` 编码：

R-Type 指令	func (6-bit)	ALUop (3-bit)	备注
add	10 00 00	010	prj1-ALU 已实现
addu	10 00 01	010	prj1-ALU 已实现
sub	10 00 10	110	prj1-ALU 已实现
subu	10 00 11	110	prj1-ALU 已实现
and	10 01 00	000	prj1-ALU 已实现
or	10 01 01	001	prj1-ALU 已实现
xor	10 01 10	100	新增
nor	10 01 11	101	新增
slt	10 10 10	111	prj1-ALU 已实现
sltu	10 10 11	011	新增

ALUop 的编码规律分析

ALUop (3-bit) 的生成主要依赖于 R-Type 指令的 func 字段 (6-bit)，特别是其低 4 位 func[3:0]，因为所有这些指令的 func[5:4] 均为 10 。

- **算术运算组 (add, sub):** 当 func[5:3] = 100 (func = 1000xx)
  - 若 func[1] = 0 (add, addu)，则 ALUop = 010 。
  - 若 func[1] = 1 (sub, subu)，则 ALUop = 110 。
  - ALUop[2] 似乎与 func[1] 相关（同为 1 时指示减法类）。
- **逻辑运算组 (and, or, xor, nor):** 当 func[5:4]=10 且 func[3]=1 (func = 1001xx)
  - and (func[2:0] = 100) → ALUop = 000
  - or (func[2:0] = 101) → ALUop = 001
  - xor (func[2:0] = 110) → ALUop = 100
  - nor (func[2:0] = 111) → ALUop = 101
  - 此组解码逻辑可简化为：
    - ALUop[2] = func[2]
    - ALUop[1] = 0
    - ALUop[0] = func[0]
- **比较运算组 (slt, sltu):** 当 func[5:4]=10 , func[3]=0, func[2]=1 (func = 1010xx)
  - slt (func[1:0] = 10) → ALUop = 111
  - sltu (func[1:0] = 11) → ALUop = 011
  - ALUop[1:0] 似乎固定为 11 ，而 ALUop[2] 用于区分有符号/无符号。

总结规律：ALUop 的生成逻辑根据 func 字段的不同位组合区分操作大类，并在类内进一步细化。逻辑运算组的解码尤为规整。

ALUop 编码的优化空间分析

前提条件：不允许改变实验项目 1 中已实现的 5 种操作编码。这 5 种固定编码为：add/addu → 010, sub/subu → 110, and → 000, or → 001, slt → 111。

这些编码已占用了 3 位 ALUop 的 8 个可能值中的 5 个。剩余未使用的 ALUop 编码为：011, 100, 101。新增的 3 个操作 XOR, NOR, SLTU 必须使用这 3 个剩余的编码。表格中的分配是：xor → 100, nor → 101, sltu → 011。

分析与结论

1. **编码位数：**共有 8 种不同的 ALU 操作需要区分。3 位 `ALUop` ($2^3 = 8$ 种状态) 是满足此需求的最小位数，因此在位数上没有优化空间。

2. **编码分配的合理性：**

   - 对于**逻辑运算组** (`and`，`or`，`xor`，`nor`)，其 `func` 字段分别为 `100100`, `100101`, `100110`, `100111`。当前的 `ALUop` 分配 (`000`，`001`，`100`，`101`) 完美支持了之前总结的解码规律：

     $$\begin{aligned} \text{ALUop}[2] &= \text{func}[2] \\ \text{ALUop}[1] &= 0 \\ \text{ALUop}[0] &= \text{func}[0] \end{aligned}$$

     这种分配使得从 `func` 到 `ALUop` 的控制逻辑最为简洁。如果将 `100` 或 `101` 分配给非逻辑操作，或者在 `xor` 和 `nor` 之间互换，这个简洁的解码逻辑就会被破坏。

   - 对于 `sltu`，其 `func` 字段为 `101011`。它被分配了唯一剩下的 `ALUop = 011`。这是在满足其他约束后的必然选择。

3. **优化空间结论：**在题目给定的强约束条件“不允许改变实验项目 1 中已实现的 5 种操作编码”下：

   - 新增的三个操作 `XOR`, `NOR`, `SLTU` 必须使用剩余的三个唯一 `ALUop` 编码。
   - 当前表格中的分配方案（`XOR` $\rightarrow$ `100`, `NOR` $\rightarrow$ `101`, `SLTU` $\rightarrow$ `011`）是唯一能够同时满足以下条件的方案：
     (a) 使用所有可用的 `ALUop` 编码。
     (b) 保持逻辑运算组从 `func` 到 `ALUop` 的解码逻辑最为规整和简单。

   因此，在现有约束下，表格中的 `ALUop` 编码方案**几乎没有进一步的优化空间**。任何对新操作编码的调整都会破坏已有的解码简洁性或无法满足编码的唯一性需求。如果允许修改原有的 5 个编码，则可能存在其他优化方案，但这超出了题目的限制范围。

五、 实验所耗时间

在课后，你花费了大约 20（代码完成 13 小时，完成实验报告 7 小时） 小时完成此次实验。