中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 薛翼舟 学号 2023K8009929044 专业 计算机科学与技术

实验项目编号 1 实验名称 单周期非流水线 simple_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。

1 逻辑电路结构与仿真波形的截图及说明

在本实验中, 基本思路根据理论课以及研讨课所展示的那张简单 cpu 的图来进行设计, 我在本次实验中采用模块化设计的思路, 并非是在 simple_cpu 中塞下全部代码, 而是将译码 id, 执行 ex, 指令计数器 pc 在不同的.v 文件里编辑, 并最终在 simple_cpu.v 文件之中进行实例化, 以提高代码的可读性和可拓展性, 下面根据模块来分别汇报

1.1 译码模块

1.2 12234

在本次实验之中, 译码模块是较为关键的模块, 因为在这个阶段产生了几乎所有的控制信号, 以及后续译码阶段所用到的 alu 和 shifter 的两个操作数, 下面对每个控制信号以及一些操作数的选择进行汇报讲解

```
`timescale 10ns / 1ns
`include "define.v"

module id(
    input           clk,

    input [31:0]   PC,
    input [31:0]    Instruction,


    output [4:0]   waddr,
    output [4:0]   raddr1,
    output [4:0]   raddr2,

    //定义一些控制信号
    output     Regdst,
    output     Branch,
```

```
    output     MemRead,
    output          MemWrite,
    output     MemtoReg,
    output     ALUSrc,
    output [2:0]  ALUop,
    output [1:0] Shiftop,
    output     RegWrite,
    output     Jump,

    //ALU和Shifter二者二选一的选择信号
    output     AluShi_sel,

    //ALU的操作数
    output [31:0] ALUop_A,
    output [31:0] ALUop_B,

    //Shifter的操作数
    output [31:0] Shiftop_A,
    output [4:0] Shiftop_B,

    //与内存有关操作的控制信号，符号扩展和字长
    output     Ext,
    output [1:0] memLen, //2'b00字节，2'b01半字，2'b10字


    //有关分支的offset输出
    output [15:0] offset,

    //长跳转的位置指令
    output [25:0] instr_index,

    //分支和跳转的目标地址
    output [31:0] Jump_addr,
    output [31:0] Branch_addr,

    //从寄存器中读出来的两个数据,用于ALUop和shiftop的选择
    input [31:0] rdata1,
    input [31:0] rdata2

);
```

在程序头部定义接口，各个接口的含义已经在注释中写出

```
//进行R-type指令分块赋值
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 [4:0] shamt = Instruction [10:6];
  wire [5:0] func = Instruction [5:0];
```

```
//add I-type branch wire
assign offset = Instruction [15:0];
//add I-type calculate wire
wire [15:0] imm     = Instruction [15:0];
//add I-type memory wire
wire [4:0] base     = Instruction [25:21]; //和rs在位置上等价


//add REGIMM wire
wire [4:0] REG = Instruction [20:16];

//add J-type wire
assign instr_index = Instruction [25:0];
```

进行指令分段, 用于之后控制信号产生中的布尔表达式判断, 提高可读性, 接下来将利用上面所定义的一些接口, 来产生各个控制信号吗, 下面对各个控制信号的产生逻辑进行讲解

1.2.1 与内存读写有关的控制信号

第一部分包括内存读信号, 内存写信号, 内存向寄存器写信号, 内存读出数据以后的符号位扩展选择信号 (零扩展和符号位扩展), 以及读出的内存长度 (包括字节, 半字, 字)

```
//内存控制信号和内存->寄存器
assign MemRead = opcode[5] & ~opcode[3];
assign MemWrite = opcode[5] & opcode[3];
assign MemtoReg = MemRead;

assign Ext = (opcode[5:4]==2'b10 && (opcode[2]==0))?  1 :  //符号扩展
                    0;   //零扩展

assign memLen = ({opcode[5:4],opcode[1:0]}==4'b1000)?  2'b00: //字节
                ({opcode[5:4],opcode[1:0]}==4'b1001)?  2'b01: //半字
                ({opcode[5:4],opcode[1:0]}==4'b1010)?  2'b10:
                //低两位为10的时候是非对齐, 用10来标记非对齐
                ({opcode[5:4],opcode[1:0]}==4'b1011)?  2'b11: //字
                                                       2'b00; //默认
```

其中内存读写信号由 opcode 直接产生, 发现读操作的各个指令 (包括各种 l 型指令) 中, 内存读一定伴随着写入寄存器, 于是直接将 MemtoReg 赋值为 MemRead, Ext 的零扩展主要面对的是读出的数据进行无符号写入的情况 (LBU, LHU), Memlen 的产生主要是面对根据 opcode 的高两位 (用于确定是内存读写命令) 和低两位 (用于确定是字节, 半字, 字以及非对齐, 我们这里利用 memlen=10 来标记非对齐的情况) 来进行判断

第二部分是本实验中的重难点, 也就是读写部分的地址选择, 以及非对齐读写的内容和写入位置的选择, 另外要注意本次实验采用小尾端设计

1. Address, 用于确定读写命令的地址

   ```
   assign Address =  ex_Result & 32'hfffffffc; //对内存访存的地址
   wire [1:0] effaddr_ctrl;
   assign effaddr_ctrl = ex_Result[1:0];
   ```

由于有非对齐存储的情况, 于是将 Address 的低两位清零, 来将这两位作为控制信号, 通过改变写入的内容, 来实现不同位置的写读以及非对齐写读

2. 有关内存读写的各个控制信号, 包括左右非对齐写读控制信号, 另外在本实验中我们将读和写的内容分为 4 个字节来进行操作, 极大程度上提高了代码可读性和简洁度

```
wire [1:0] effaddr_ctrl;
assign effaddr_ctrl = ex_Result[1:0];
wire rl_sel;         //用于选择左非对齐读入和右非对齐写入,0为L,1为R
assign rl_sel = opcode[2];

//将从内存读出来的4个字节数据分成4个字节,用于小尾端和非对齐写入
wire [7:0] read_byte_3;
wire [7:0] read_byte_2;
wire [7:0] read_byte_1;
wire [7:0] read_byte_0;

assign read_byte_3 = Read_data [31:24];
assign read_byte_2 = Read_data [23:16];
assign read_byte_1 = Read_data [15: 8];
assign read_byte_0 = Read_data [ 7: 0];

//用于记录写回寄存器原本的数据, 用于生成RF_wdata
wire [31:0] wdata_i;
assign wdata_i  =  rdata2;

//由于非对齐和对齐中写入的字节不同,于是更改Write_data;
wire [5:0] Write_data_sel;
assign Write_data_sel = {rl_sel,memLen,effaddr_ctrl};

//将从寄存器读出来的4个字节数据分成4个字节,用于小尾端和非对齐写入
wire [7:0] wb3;
wire [7:0] wb2;
wire [7:0] wb1;
wire [7:0] wb0;

assign wb3 = rdata2 [31:24];
assign wb2 = rdata2 [23:16];
assign wb1 = rdata2 [15: 8];
assign wb0 = rdata2 [ 7: 0];
```

3. Write_strb 代表写入的位置, 对于不同的读写写命令有着不同的赋值, 利用左右非对齐写入和地两位来产生非对齐的控制信号 (由于以下几部分的代码比较长, 于是直接插入截图来保证报告的整齐)

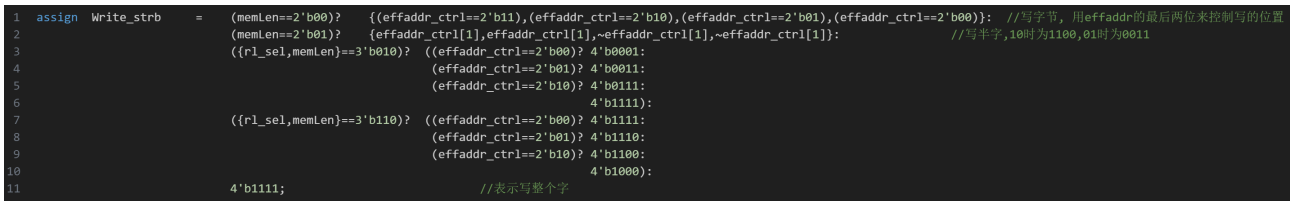

图 1: Write_strb 的产生

4. RF_wdata 代表写入寄存器的数据, 由于在本实验中我们将读和写的内容分为 4 个字节来进行操作, 对齐处理的情况比较简单, 但是在非对齐的情况下, 我们需要利用地址的低两位作为控制信靠来确定写入的内容 (根据 MIPS) 指令集手册, 值得一提的是利用 00 和 10 来分别表示半字, Write_data 同理

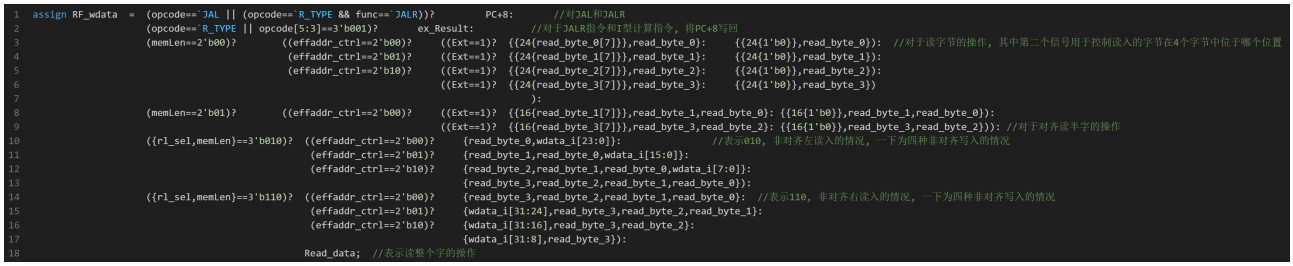

图 2: RF_wdata 的产生

5. Write_data 表示写入内存的数据, 要注意符号拓展和零扩展, 以及左右非对齐情况下的写入

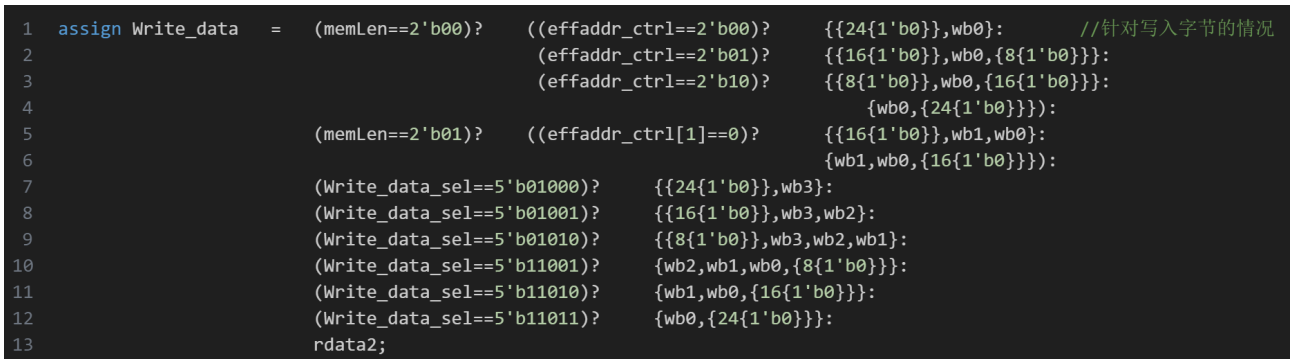

图 3: Write_data 的产生

1.2.2 寄存器输入的控制信号

主要是控制读或者写的内容是 rt 寄存器还是 rd 寄存器

```
assign RegDst = (opcode==`R_TYPE)? 1:0;
```

1.2.3 分支和跳转信号

分支和跳转信号以及 R 型的跳转指令, 利用 opcode 和 func 来判断他们, 利用布尔表达式产生控制信号

```
//分支控制信号
assign Branch = (opcode[5:2]==4'b0001 || opcode==`REGIMM);
```

```
//跳转控制信号
assign Jump = (opcode[5:1]==5'b00001 || (opcode==`R_TYPE && func[3:1]==3'b100) || opcode==`JAL);
```

1.2.4 读写寄存器的赋值

raddr1 和 raddr2 就是 rs 和 rt, 值得注意的是 waddr 的赋值, 如果是 R 型指令需要赋值为 rd, 另外对于 JAL 指令, 需要赋值为 31, 因为要将 PC+8 赋值给该特殊寄存器, waddr 在其他情况下为 rt

```
//读寄存器地址
assign raddr1 = rs;
assign raddr2 = rt;

//写寄存器地址
assign waddr = (RegDst==1)?   rd:
              (opcode==`JAL)?  5'b11111:
               rt;
```

1.2.5 运算器和移位器的功能选择信号

这两部分同样是本次实验的重点, 运算器的部分我列出了各个指令以及他们所对应的 ALUop, 并且利用卡诺图化简, 来最终实现这个 ALUop 的赋值, 另外我将不需要 ALU 的指令 (包括跳转) 的 ALUop 赋值为 OR, 对应的 ALU 操作数和是 0, 以避免意料之外的错误.

```
assign ALUop =
((opcode[5]==1) || (opcode==`ADDIU) || (opcode==`R_TYPE && (func==`ADDU || func[5:3]==3'b001)))?
   `ADD :  //用到add的情况
((opcode==`R_TYPE && func==`SUBU) || opcode[5:1]==5'b00010)?
`SUB : //用到sub的情况
((opcode==`R_TYPE && func==`AND_) || opcode==`ANDI)?
`AND : //用到and的情况
((opcode==`R_TYPE && func==`OR_) || opcode==`ORI)?
OR  : //用到or的情况
((opcode==`R_TYPE && func==`XOR_) || opcode==`XORI)?
`XOR : //用到xor的情况
(opcode==`R_TYPE && func==`NOR_)?
`NOR : //用到NOR的情况
((opcode==`R_TYPE && func==`SLT_) || opcode==`SLTI || opcode==`REGIMM || opcode[5:1]==5'b00011)?
`SLT :  //用到SLT的情况
((opcode==`R_TYPE && func==`SLTU_) || opcode==`SLTIU)?
`SLTU:  //用到SLTU的情况
`OR; //有的情况不需要ALU,默认赋值为or
```

对于移位操作器 Shifter, 住的一体的就是 LUI 指令, 在这个指令我们赋值为左移, 另外有的情况下不移位, 我们设置为 01

```
assign Shiftop = ((opcode==`R_TYPE && func[5:3]==3'b000))?  func[1:0] :   //Rtype算数移位指令
              ((opcode==`LUI))?                          2'b00    :        //加载到高位
                                                      2'b01;           //其他情况下不移位
```

另外考虑到无论如何 ALU 和 Shifter 总会计算出一个结果, 这意味着每一条指令总会产生两个运算的结果, 于是我们引入一个新的控制信号, 用于在 ALU 和 Shifter 的结果之间进行选择

```
assign AluShi_sel = ((opcode==`R_TYPE && func[5:3]==3'b000) || opcode==`LUI);
    //结果为1,选择shifter,否则选择alu
```

1.2.6 ALU 和 Shifter 操作数的选择

首先我们要对立即数进行拓展, 分为符号位拓展和零扩展

```
wire [31:0] op_imm;
assign op_imm = (opcode[5:2]==4'b0011)?  {{16{1'b0}},imm}: //零扩展
            {{16{imm[15]}},imm}; //符号扩展
```

其次是 ALU 的两个操作数, 另外对于 bgez 等四个指令, 我通过对操作数的选择来简化之后分支指令的产生, 例如 bgtz 是数据大于 0, 于是在这种情况下, 我将操作数 a 设为 0, 操作数 b 设为 rdata1, 这样就可以直接利用 SLT 的结果来判断, 省略了之后对 SLT 结果的一步操作, 具体的 ALUop_A 和 ALUop_B 如下'

```
assign ALUop_A = (opcode==`BGTZ)?    32'b0:     //0<rdata1时为1,表示rdata大于0时分支
                (opcode==`REGIMM && REG[0]==1)?  {32{1'b1}}:
                    //-1<rdata1时为1,表示rdata大于等于0时分支
                rdata1;

assign ALUop_B = (opcode==`R_TYPE && (func==`MOVZ || func==`MOVN))?  32'b0:
    //对于R中的两个移动指令,ALUop为add,因此rdata1+0
                (opcode==`REGIMM && REG[0]==0)?         32'b0:     //rdata1小于0时分支
                (opcode==`BLEZ)?              32'b1:     //rdata1小于1时分支,也就是小于等于0时分支
                (opcode==`BGTZ || (opcode==`REGIMM && REG[0]==1))?  rdata1:
                    //对于上面的两种情况,B应该是rdata1
                (ALUSrc)?                op_imm:
                rdata2;
```

之后是对 Shifter 操作数的赋值, 这个要简单很多, 在 R 型指令中利用 shamt 来进行赋值, 或者利用 rdata1 中的低 5 位数据来进行赋值 (可变移位), 同样还是将 LUI 单独拿出来判断

```
assign Shiftop_A = (opcode==`LUI)?       op_imm:    //LUI指令对立即数做移位
                                    rdata2;     //rt移动
assign Shiftop_B =   (opcode==`R_TYPE && rs==5'b0)?  shamt:
                     (opcode==`LUI)?      5'b10000: //LUI指令将立即数左移16位
                     rdata1[4:0]; //可变目标左右移量为rs[4:0]
```

1.2.7 寄存器写使能信号

在这一部分中定义一个中间控制信号 mov_con, 利用控制 movn 和 movz, 来简化判断

```
//定义一个中间控制信号,用于控制movn和movz
wire mov_con;
assign mov_con = (rdata2 == 32'b0);

//定义寄存器写使能信号
```

```
assign RegWrite =   (opcode==`R_TYPE && func==`JR)?                  0:   //JR时不写入
     ((opcode == `R_TYPE && {func[5:3],func[1]}!=4'b0011) || opcode[5:3]==3'b001 || MemRead)?  1:
     (opcode==`R_TYPE && func==`MOVZ)?               mov_con: //rt==0则写入
     (opcode==`R_TYPE && func==`MOVN)?               ~mov_con: //rt!=0则写入
     (opcode==`JAL)?                          1:   //JAL的情况，拉高wen
                          0;   //其他情况下不写入
```

1.2.8 分支和跳转的地址

这是本模块的最后一部分，用于计算分支和跳转的地址，主要是针对 Jump 指令和分支指令，具体操作严格按照 MIPS 指令集手册的定义，其实这一部分放在 ex.v 中也可以

```
//计算出分支目标地址
wire [15:0] offset_temp;
assign offset_temp = offset << 2;
assign Branch_addr = {{16{offset_temp[15]}},offset_temp};

//计算出跳转目标地址
assign Jump_addr =  (opcode[5:1]==5'b00001)? {PC[31:28],instr_index,2'b00} : //J型指令
            rdata1;          //R型跳转指令
```

1.3 执行模块

由于主要的工作都在译码模块中，所以执行模块的设计相对简单，主要是实例化 ALU 和 Shifter，另外还有一部分是分支控制信号的产生，主要就是对比大小的跳转指令利用 ALU_result 作为分支控制信号，以及 BEQ 和 BNE 指令分别利用 Zero 和 Zero 作为分支控制信号

```
module ex(
    input         clk,

    input [31:0]  PC,
    input [31:0]  Instruction,

    //ALU需要用到的数据以及控制信号
    input [31:0]  ALUop_A,
    input [31:0]  ALUop_B,
    input [2:0]   ALUop,

    //Shifter需要用到的数据以及控制信号
    input [31:0]  Shiftop_A,
    input [ 4:0]  Shiftop_B,
    input [1:0]   Shiftop,

    //id阶段读入的offset偏移
    input [15:0]  offset,

    //ALU和Shifter结果二选一信号
    input         AluShi_sel,
```

```
    //定义ALU和Shifter输出信号
    output [31:0] Result,

    //有关ALU结果的寄存器信号控制
    output        ALU_Branch
);
    //实例化ALU
    wire Zero;
    wire [31:0] ALU_result;
    wire [5:0] opcode;
    assign opcode = Instruction[31:26];

    alu alu_ex(
           .A            (ALUop_A),
           .B            (ALUop_B),
           .ALUop        (ALUop),
           .Overflow     (),
           .CarryOut     (),
           .Zero         (Zero),
           .Result       (ALU_result)
    );

    //实例化Shifter
    wire [31:0] Shifter_result;

    shifter shifter_ex(
           .A            (Shiftop_A),
           .B            (Shiftop_B),
           .Shiftop      (Shiftop),
           .Result       (Shifter_result)
    );

    //在Shifter和ALU的结果中二选一,输出一个结果
    assign Result = (AluShi_sel)? Shifter_result:
                                  ALU_result;

    //对于分支指令,将其分支控制信号赋值
    wire  ALU_Branch_temp;
    assign ALU_Branch_temp = (opcode[5:1]==5'b00011 || opcode==`REGIMM)? ALU_result :
                          //对于比大小的分支指令
                          (opcode==`BEQ)?                          Zero   :
                          //对于beq, 两个相等, 跳转
                          (opcode==`BNE)?                          ~Zero  :
                          //对于bne, 两个不相等, 跳转
                                                                   0;
                          //其他一般情况下, 不跳转
    assign ALU_Branch   = ALU_Branch_temp;

endmodule
```

1.4 PC 模块

PC 模块主要是对 PC 进行加 4, 以及对分支和跳转指令的处理, PC 的设计同样简单, 但值得注意的是要 PC <= PC + Branch_addr + 4 来流出分支延迟槽, 虽然在目前的单周期以及多周期中没有作用, 但是在流水线中分支延迟槽是不可或缺的, 另外, PC 也是本次实验中除了之前写的 reg_file 以外唯一的时序逻辑

```
 module pc(
   input         clk,
   input         rst,

   //输入的有关分支和跳转的控制信号
   input         Branch,
   input         ALU_Branch,
   input         Jump,

   //输入的有关分支和跳转的目标地址
   input [31:0]  Jump_addr,
   input [31:0]  Branch_addr,

   output reg [31:0] PC
);
   wire Branch_f;
   assign Branch_f = Branch && ALU_Branch;

   always @(posedge clk) begin
          if(rst) begin
                 PC <= 32'd0;
          end
          else if(Branch_f)begin      //对于分支
                 PC <= PC + Branch_addr + 4;
          end
          else if(Jump)begin          //对于跳转
                 PC <= Jump_addr;
          end
          else begin                  //其他一般情况
                 PC <= PC + 4;
          end
   end

endmodule
```

1.5 运算模块

本模块是上个项目 prj1 所实现的, 在此不多赘述

```
 `timescale 10 ns / 1 ns

 //define the operation
 `define DATA_WIDTH 32
 `define AND 3'b000
```

```
`define OR 3'b001
`define XOR 3'b100
`define NOR 3'b101
`define ADD 3'b010
`define SUB 3'b110
`define SLT 3'b111
`define SLTU 3'b011

module alu(
  input [`DATA_WIDTH - 1:0] A,
  input [`DATA_WIDTH - 1:0] B,
  input [            2:0] ALUop,
  output                  Overflow,
  output                  CarryOut,
  output                  Zero,
  output [`DATA_WIDTH - 1:0] Result
);

  wire [31:0] temp_sub;
  assign temp_sub = A + ~B + 1;
  wire [32:0] unsigned_A;
  wire [32:0] unsigned_B;
  wire [32:0] unsigend_sub;

  //对于无符号数的比较，将输入拓展一位，转化成有符号数来进行比较
  assign unsigned_A =  {1'b0,A};
  assign unsigned_B =  {1'b0,B};
  assign unsigend_sub = unsigned_A + ~unsigned_B + 1;

  assign {CarryOut,Result} = (ALUop == `AND) ? A & B:
          (ALUop == `OR ) ? A | B:
          (ALUop == `XOR) ? A ^ B:
          (ALUop == `NOR) ? {1'b0,~(A | B)}:
          (ALUop == `ADD) ? A + B:
          (ALUop == `SUB) ? A + ~B + 1:
          (ALUop == `SLT) ? (({A[31],B[31]} == 2'b10) ? 1 :
                  ({A[31],B[31]} == 2'b01) ? 0 :
                (temp_sub[31] == 1) ? 1: 0) :
          (ALUop == `SLTU) ? (unsigend_sub[32] == 1) :
          0;

  assign Overflow =
  (ALUop == `ADD) ? ((A[31] && B[31] && ~Result[31]) || (~A[31] && ~B[31] && Result[31])) :
  (ALUop == `SUB) ? ((A[31] && ~B[31] && ~Result[31]) || (~A[31] && B[31] && Result[31])) :
       0;
     //overflow in sub : +- -> -; -+ -> +
     //overflow in add : ++ -> -; -- -> +

  assign Zero = (Result == 0);
endmodule
```

1.6 寄存器堆模块

reg_file 同样在 prj1 中实现, 在此不多赘述

```
`define DATA_WIDTH 32
`define ADDR_WIDTH 5

module reg_file(
  input                   clk,
  input [`ADDR_WIDTH - 1:0] waddr,
  input [`ADDR_WIDTH - 1:0] raddr1,
  input [`ADDR_WIDTH - 1:0] raddr2,
  input                   wen,
  input [`DATA_WIDTH - 1:0] wdata,
  output [`DATA_WIDTH - 1:0] rdata1,
  output [`DATA_WIDTH - 1:0] rdata2
);

  // TODO: Please add your logic design here
  // define a register 32*32
  reg [`DATA_WIDTH - 1:0] myreg [`DATA_WIDTH - 1:0];

  //write
  always @(posedge clk) begin
    //check wen and waddr
    if(wen && waddr)begin
      myreg[waddr] <= wdata;
    end
  end

  //read1, set 0 if address = 0
  assign rdata1 = (raddr1 == 0)? 32'h00000000 : myreg[raddr1];
  //read2, set 0 if address = 0
  assign rdata2 = (raddr2 == 0)? 32'h00000000 : myreg[raddr2];

endmodule
```

1.7 移位器模块

shifer 模块中, 利用 shifterop 来对 op_A 进行不同的操作, 分为左移, 逻辑左移以及逻辑右移, 在设计 shifter 的时候, 利用了 $ signed 来进行无符号数和有符号数的转换, 另外我们需要引入三个中间结果变量来承接三个结果, 否则算术右移会出现意料之外的问题, 会在下一部分中进行具体的阐述

```
`define DATA_WIDTH 32

module shifter (
    input [`DATA_WIDTH - 1:0] A,
```

```
    input [           4:0] B,
    input [           1:0] Shiftop,
    output [`DATA_WIDTH - 1:0] Result
);
     //转换为有符号数
  wire signed [`DATA_WIDTH-1:0] A_signed;
     assign A_signed = $signed(A);

  //值得注意的是，在这里需要用这三个wire来承接三个左右移的结果，经过测试，在三目运算符中的运算如果有
  //无符号数的出现，会将运算公式中的所有有符号数转换为无符号数，来产生结果
     wire [`DATA_WIDTH-1:0] result_sll;
     wire [`DATA_WIDTH-1:0] result_srl;
     wire [`DATA_WIDTH-1:0] result_sra;


     assign result_sll = A << B;
     assign result_srl = A >> B;
     assign result_sra = A_signed >>> B;

     assign Result = (Shiftop == 2'b00) ? result_sll : // 逻辑左移
                 (Shiftop == 2'b10) ? result_srl : // 逻辑右移
                 (Shiftop == 2'b11) ? result_sra : // 算术右移
                 A;                              // 默认直接输出A
endmodule
```

1.8 顶层 simple_cpu

这一部分主要是各个线的连接以及实例化, 没有什么值得讨论的计数细节

```
`timescale 10ns / 1ns
`include "define.v"

module simple_cpu(
  input           clk,
  input           rst,

  output [31:0]   PC,
  input [31:0]    Instruction,

  output [31:0]   Address,    //对内存访存的地址
  output          MemWrite,
  output [31:0]   Write_data,
  output [ 3:0]   Write_strb,

  input [31:0]    Read_data,
  output          MemRead
);

  // THESE THREE SIGNALS ARE USED IN OUR TESTBENCH
```

```
// PLEASE DO NOT MODIFY SIGNAL NAMES
// AND PLEASE USE THEM TO CONNECT PORTS
// OF YOUR INSTANTIATION OF THE REGISTER FILE MODULE
wire      RF_wen;
wire [4:0]  RF_waddr;
wire [31:0]    RF_wdata;


//将指令分层
wire [5:0] opcode =  Instruction[31:26];
wire [5:0] func = Instruction[ 5: 0];

//用于存储译码阶段中读出来的两个数据
wire [31:0]    rdata1;
wire [31:0]    rdata2;

//进行控制信号的声明
wire   Regdst;
wire      Branch;
wire      MemtoReg;
wire   ALUSrc;
wire  [2:0]   ALUop;
wire  [1:0] Shiftop;
wire   RegWrite;
wire   Jump;
wire   AluShi_sel;

wire      Ext;
wire  [1:0] memLen;

wire  [15:0]  offset;
wire  [25:0] instr_index;
wire  [31:0] Jump_addr;
wire  [31:0] Branch_addr;

//定义两个器件的操作数
wire [31:0] ALUop_A;
wire [31:0] ALUop_B;
wire [31:0] Shiftop_A;
wire [4:0] Shiftop_B;

wire [4:0] raddr1;
wire [4:0] raddr2;


//首先进行id译码
id  id_ex(
  .clk    (clk),
  .PC     (PC),
  .Instruction (Instruction),
```

```
  .waddr    (RF_waddr),
  .raddr1   (raddr1),
  .raddr2   (raddr2),

  .Branch   (Branch),
  .MemRead  (MemRead),
  .MemWrite (MemWrite),
  .MemtoReg (MemtoReg),
  .ALUop    (ALUop),
  .Shiftop  (Shiftop),
  .RegWrite (RF_wen),
  .Jump   (Jump),

  .AluShi_sel (AluShi_sel),

  .ALUop_A (ALUop_A),
  .ALUop_B (ALUop_B),

  .Shiftop_A (Shiftop_A),
  .Shiftop_B (Shiftop_B),

  .Ext    (Ext),
  .memLen   (memLen),

  .offset   (offset),
  .instr_index (instr_index),

  .Jump_addr (Jump_addr),
  .Branch_addr (Branch_addr),

  .rdata1   (rdata1),
  .rdata2   (rdata2)
);

wire [31:0] ex_Result;
wire      ALU_Branch;

//进行寄存器堆的实例化，要注意整个程序中只能有一个寄存器堆，否则会产生多个寄存器
//导致产生并不希望的结果
reg_file reg_file_ex(
  .clk    (clk),
  .waddr    (RF_waddr),
  .raddr1   (raddr1),
  .raddr2   (raddr2),
  .wen    (RF_wen),
  .wdata    (RF_wdata),
  .rdata1   (rdata1),
  .rdata2   (rdata2)
);
```

```
  //之后进行ex执行指令阶段
  ex  ex_ex(
    .clk    (clk),
    .PC     (PC),
    .Instruction (Instruction),

    .ALUop_A (ALUop_A),
    .ALUop_B (ALUop_B),
    .ALUop    (ALUop),

    .Shiftop_A (Shiftop_A),
    .Shiftop_B (Shiftop_B),
    .Shiftop (Shiftop),

    .offset   (offset),

    .AluShi_sel (AluShi_sel),

    .Result   (ex_Result),

    .ALU_Branch (ALU_Branch)
  );

  pc  pc_ex(
    .clk    (clk),
    .rst    (rst),

    .Branch   (Branch),
    .ALU_Branch (ALU_Branch),
    .Jump   (Jump),
    .Branch_addr (Branch_addr),
    .Jump_addr (Jump_addr),

    .PC     (PC)
  );
endmodule
```

2 实验过程中遇到的问题、对问题的思考过程及解决方法

在本实验中, 主要遇到了两个问题, 下面分别进行讲解

2.1 多次实例化

在本次实验中, 由于对 verilogHDL 语言的组合逻辑性质的了解不清晰, 所以为了方便多次实例化了 reg_file, 但是这样实际上是在对应的电路中产生了多个寄存器, 这样会导致在读寄存器的时候, 读出来的值并不是我们想要的值, 所以在一个 simple_cpu 中, 有且只能由一个对于 reg_file 的实例化, 并且在一次实例化中, 将所有的读和写都进行完成

2.2 Shifter 模块算数右移

在进行行为逻辑仿真测试的时候, 发现逻辑右移无论如何不能产生一个正确的结果, 于是进行了分别的测试, 原始未修改的代码如下

```
assign Result = (Shiftop == 2'b00) ? A << B :  // 逻辑左移
                (Shiftop == 2'b10) ? A >> B :      // 逻辑右移
                (Shiftop == 2'b11) ? A >>> B : // 算术右移
                A;
```

这种是不对的, 因为 verilog 语言中, 会将未定义类型的端口自动定义为无符号数, 所以这里的 A 是无符号数

```
wire signed [31:0] A_signed = $signed(A);
assign Result = (Shiftop == 2'b00) ? A << B :  // 逻辑左移
                (Shiftop == 2'b10) ? A >> B :      // 逻辑右移
                (Shiftop == 2'b11) ? A_signed >>> B : // 算术右移
                A;
```

以据我们的预期, 这样的应该能够产生正确的结果, 但是实际上这样的程序所得到的结果和一开始原始的程序时相同的, 于是我在本地的 vivado 中进行了对于三目运算符中有关无符号数和有符号数的测试

```
assign Result = A_signed >>> B : // 算术右移
                A;
```

这样得出的同样是无符号的结果

```
assign Result = A_signed >>> B : // 算术右移
                A_signed;
```

但是这样就能够得出有符号的正确结果

```
wire [31:0] sra;
assign sra = A_signed >>> B;
assign Result = sra : // 算术右移
                A;
```

这样能够得出正确的结果, 如果在端口定义环节将 A 定义为有符号数也可以得到正确的结果, 于是我得出的结论是, 在三目运算符中如果涉及运算操作 (包括左右移等), 三目运算符中出现的任何一个无符号数会将所有的有符号数转化成无符号数再进行运算 (可能是为了避免溢出), 但是如果是赋值操作, 则不会进行这样的转换, 于是在本实验中, 需要引入中间变量来承接结果

3 对讲义中思考题(如有)的理解和回答

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

<table>
<tr><th>R-Type 指令</th><th>func （6-bit）</th><th>ALUop （3-bit）</th><th>ALUop编码</th><th>opcode （6-bit）</th><th>I-Type 指令</th><th>ALUop编码</th></tr>
<tr><td>add</td><td>10 00 00</td><td rowspan="2">010</td><td rowspan="4">ADD/SUB：func[3:2] == 2’b00
ALUop = {func[1], 2’b10}</td><td>001 0 00</td><td>addi</td><td rowspan="2">ADD：opcode[2:1] == 2’b00
ALUop = {opcode[1], 2’b10}</td></tr>
<tr><td>addu</td><td>10 00 01</td><td>001 0 01</td><td>addiu</td></tr>
<tr><td>sub</td><td>10 00 10</td><td rowspan="2">110</td><td></td><td></td><td></td></tr>
<tr><td>subu</td><td>10 00 11</td><td></td><td></td><td></td></tr>
<tr><td>and</td><td>10 01 00</td><td>000</td><td rowspan="4">逻辑运算：func[3:2] == 2’b01
ALUop = {func[1], 1’b0, func[0]}</td><td>001 1 00</td><td>andi</td><td rowspan="3">逻辑运算：opcode[2] == 1’b1
ALUop = {opcode[1], 1’b0, opcode[0]}</td></tr>
<tr><td>or</td><td>10 01 01</td><td>001</td><td>001 1 01</td><td>ori</td></tr>
<tr><td>xor</td><td>10 01 10</td><td>100</td><td>001 1 10</td><td>xori</td></tr>
<tr><td>nor</td><td>10 01 11</td><td>101</td><td>001 1 11</td><td>lui</td><td>非运算类指令，需单独处理</td></tr>
<tr><td>slt</td><td>10 10 10</td><td>111</td><td rowspan="2">比较运算：func[3:2] == 2’b10
ALUop = {~func[0], 2’b11}</td><td>001 0 10</td><td>slti</td><td rowspan="2">比较运算：opcode[2:1] == 2’b01
ALUop = {~opcode[0], 2’b11}</td></tr>
<tr><td>sltu</td><td>10 10 11</td><td>011</td><td>001 0 11</td><td>sltiu</td></tr>
</table>

图 4: ALUop 的编码

对于 R-type 的指令, 对于同一类运算 (加减法, 逻辑运算, 比大小), 他们的 ALUop 和自己的 func 是有相同位的, 于是可以采用判断 func 再利用拼接来产生 ALUop , 另外对于 I-type 指令也是类似的规律, 此外对于 R 型和 I 型指令, 他们的拼接过滤是完全相同的, 差别是一个用 opcode, 一个用 func.

我们观察到 R 和 I 的信号产生差别只有 opcode 和 func, 是否存在将通以运算的 R 和 I 型指令合并的可能? 另外目前 ADD/SUB 和逻辑运算的 ALUop 编码中，部分位的使用可能存在冗余。例如 ADD/SUB 的 {func[1], 2’b10} 中可以检查是否有进一步压缩的可能性。

4 实验所耗时间

在课后，你花费了大约 18 小时完成此次实验。