# 中国科学院大学计算机组成原理实验课

# 实 验 报 告

学号：2020K8009970001　姓名：金扬　专业：计算机科学与技术

实验序号：2　实验名称：简单功能性处理器设计（基于 MIPS 32 位指令集）

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在~/COD-Lab/reports 目录下。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 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图、相应信号的仿真波形和信号变化的说明等）

## 1．单周期 CPU

### a) 电路整体结构图

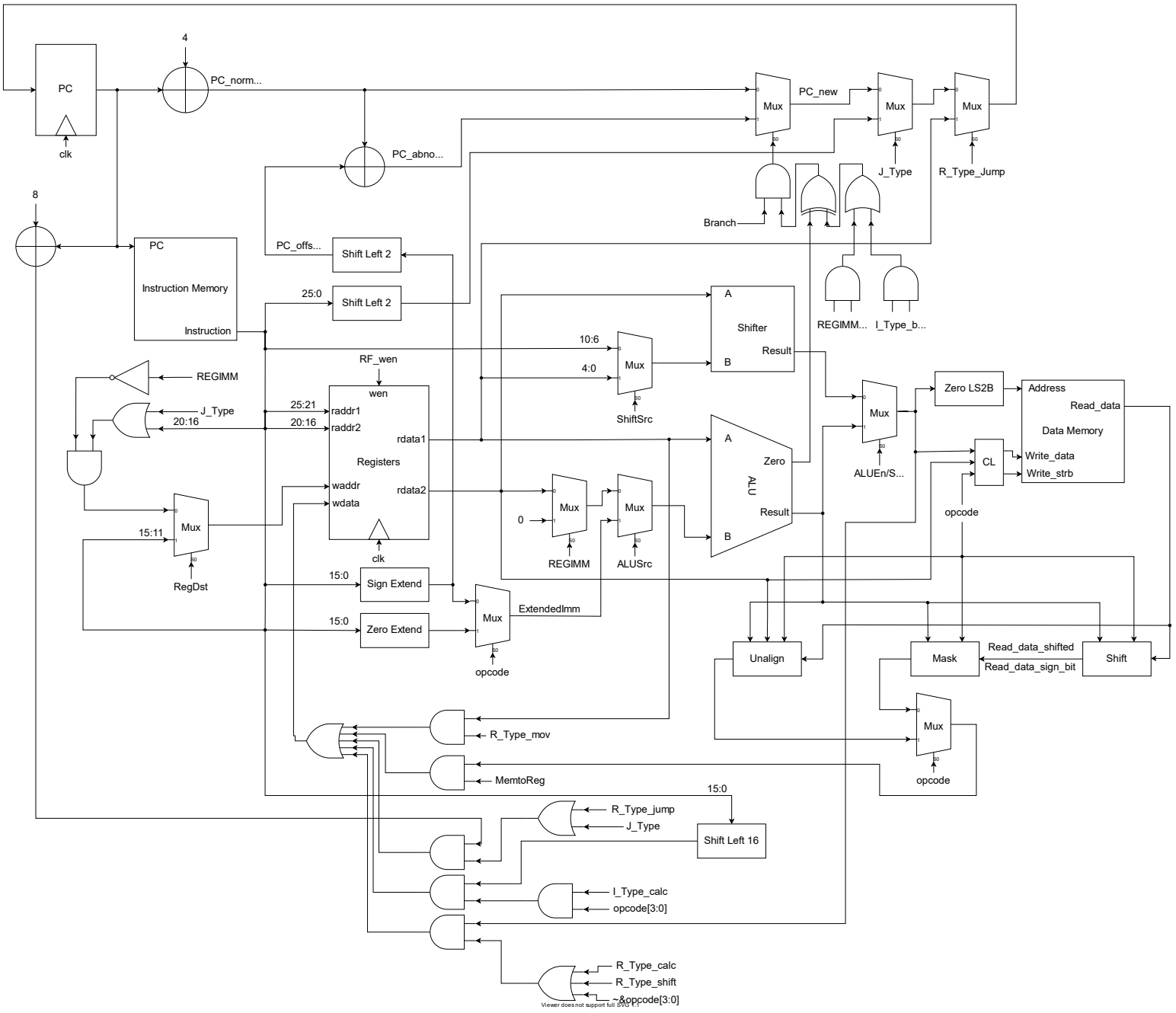


图 1 电路整体结构图

b) PC 更新

```
    //PC
    assign PC_normal = PC + 4;
    assign PC_new = ((Zero ^ (
            REGIMM & ~Instruction[16]
            | I_Type_b & (opcode[0] ^ (opcode[1] & |RF_rdata1))
        )) & Branch) ? PC_abnormal : PC_normal;
    always @(posedge clk) begin
        if(rst) begin
            PC <= 32'd0;
        end else begin
            PC <= R_Type_jump ? RF_rdata1 : J_Type ?
{Instruction[25:0], 2'b00} : PC_new;
        end
    end

    //PC Adder
    assign PC_offset    = {SignExtend[29:0], 2'b00};
    assign PC_abnormal  = PC_normal + PC_offset;
```

如图，

对于跳转指令，PC 值直接由寄存器或指令指定；

对于分支指令，PC_abnormal 值为 PC+4 基础上增加偏移值 PC_offset；

对于其他指令，PC_normal 值为 PC+4。

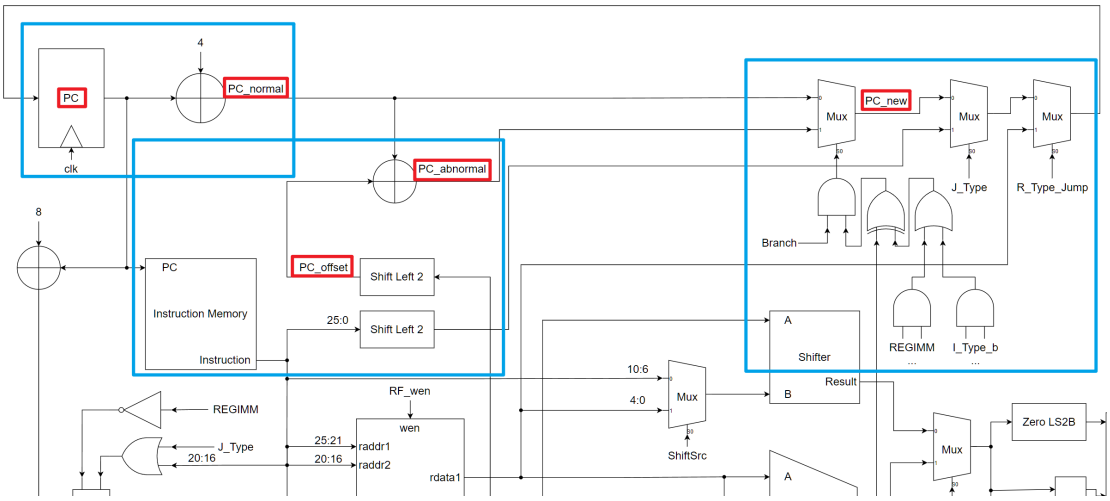


图 2 PC

c) 寄存器

```
reg_file Registers(
    .clk    (clk),
    .waddr  (RF_waddr),
    .raddr1 (Instruction[25:21]),
    .raddr2 (Instruction[20:16]),
    .wen    (RF_wen),
    .wdata  (RF_wdata),
    .rdata1 (RF_rdata1),
    .rdata2 (RF_rdata2)
);
assign RF_waddr = RegDst ? Instruction[15:11] :
    {(5){~REGIMM}} & ({(5){J_Type}} | Instruction[20:16]);
```

如图，寄存器写入地址根据指令类型相应生成。

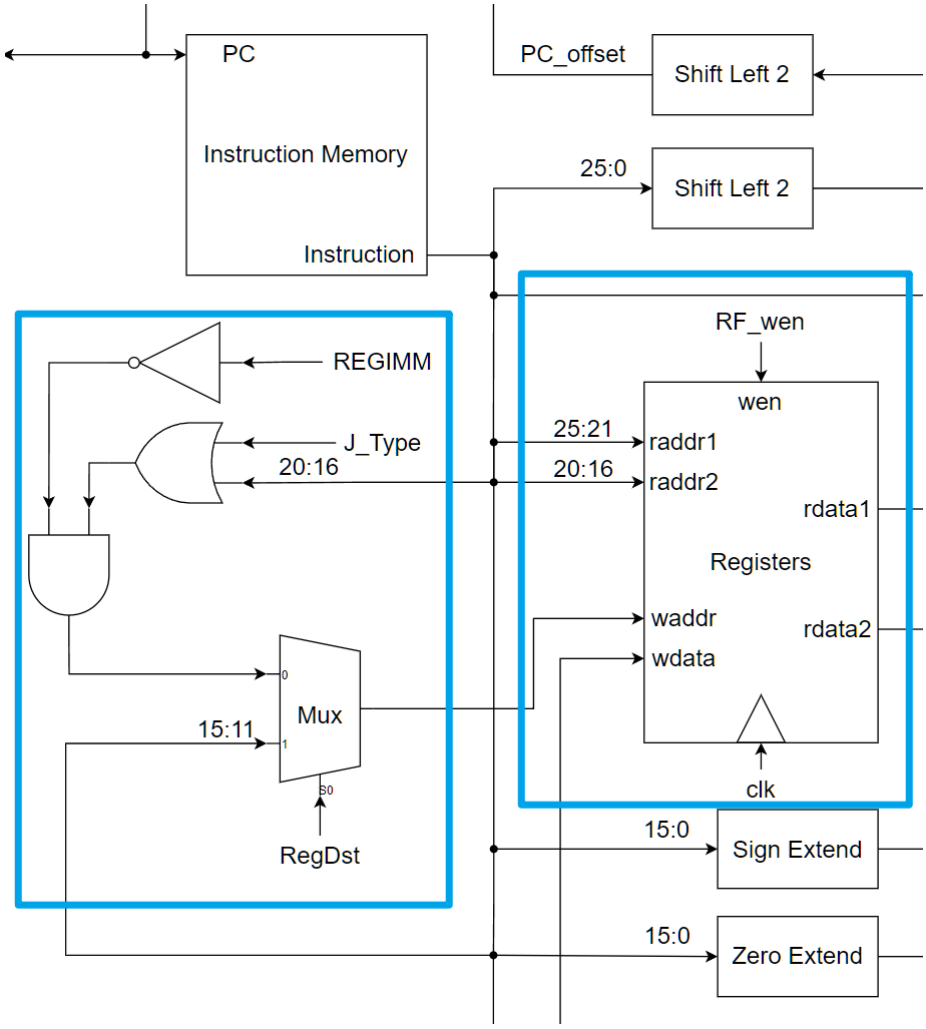


图 3 寄存器

d) 立即数扩展

```
    //Sign extend
    assign SignExtend = {{(16){Instruction[15]}},
Instruction[15:0]};
    //Zero extend for I-Type andi, ori, xori
    assign ZeroExtend = {16'b0, Instruction[15:0]};
    //Select
    assign ExtendedImm = opcode[5:2] == 4'b0011 ? ZeroExtend :
SignExtend;
```

如图，对立即数分别进行符号扩展与补零扩展，并根据指令类型进行选择。

其中对于分支指令，符号扩展值还用于生成 PC 偏移量 PC_offset。

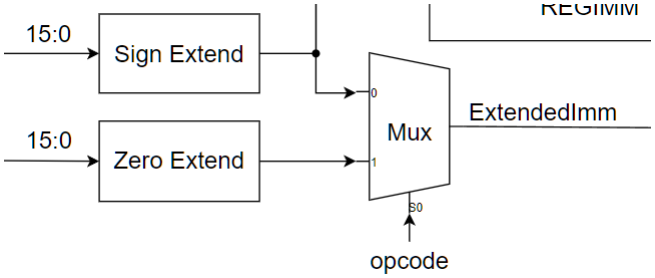


图 4 立即数扩展

e) 控制信号生成

```
    assign R_Type       = opcode[5:0] == 6'b000000;
    assign R_Type_calc  = R_Type & func[5];
    assign R_Type_shift = R_Type & (func[5:3]==3'b000);
    assign R_Type_jump  = R_Type & ({func[5:3],func[1]}==4'b0010);
    assign R_Type_mov   = R_Type & ({func[5:3],func[1]}==4'b0011);
    assign REGIMM       = opcode[5:0] == 6'b000001;
    assign J_Type       = opcode[5:1] == 5'b00001;
    assign I_Type_b     = opcode[5:2] == 4'b0001;
    assign I_Type_calc  = opcode[5:3] == 3'b001;
    assign I_Type_mr    = opcode[5:3] == 3'b100;
    assign I_Type_mw    = opcode[5:3] == 3'b101;

    assign RegDst   = R_Type;
    assign Branch   = REGIMM | I_Type_b;
    assign MemRead  = I_Type_mr;
    assign MemtoReg = I_Type_mr;
    assign ALUEn    = R_Type_calc | REGIMM | I_Type_b | I_Type_calc
| I_Type_mr | I_Type_mw;
    assign ShiftEn  = R_Type_shift;
    assign MemWrite = I_Type_mw;
    assign ALUSrc   = I_Type_calc | I_Type_mr | I_Type_mw;
    assign ShiftSrc = func[2];
    assign RF_wen   = R_Type & (~R_Type_mov | (func[0]^~|RF_rdata2))
                          & (~R_Type_jump | func[0])
                         // jalr allowed, jr not
   /* must add the line above
    * since the benchmark judges RF_wen==1 (when jr) to be wrong
    * even if RF_waddr==0
    * when the RF naturally prevent data from writing in
    */
                  | J_Type & opcode[0]
                  | I_Type_calc
                  | I_Type_mr;
```

先根据指令内容生成指令类型，再根据指令类型生成各类控制信号。

f) ALU

```
    alu ALU(
        .A          (RF_rdata1),
        .B          (ALUSrc ? ExtendedImm : REGIMM ?
32'b0 :RF_rdata2),
        .ALUop      (ALUop),
        .Overflow   (Overflow),
        .CarryOut   (CarryOut),
        .Zero       (Zero),
        .Result     (ALUResult)
    );
    assign ALUop = {(3){R_Type_calc | R_Type_jump}} & {
        func[1] & ~(func[3] & func[0]),
        ~func[2],
        func[3] & ~func[2] & func[1] | func[2] & func[0]
    } | {(3){I_Type_calc}} & {
        opcode[1] & ~(opcode[3] & opcode[0]),
        ~opcode[2],
        opcode[3] & ~opcode[2] & opcode[1] | opcode[2] & opcode[0]
    } | {(3){REGIMM}}
      | {(3){I_Type_b}}  & {2'b11, opcode[1]}   // slt 111 sub 110
      | {(3){I_Type_mr | I_Type_mw}} & 3'b010;
```

如图，输入 A 为寄存器第一读端口值，输入 B 需要根据指令类型选择后生成。

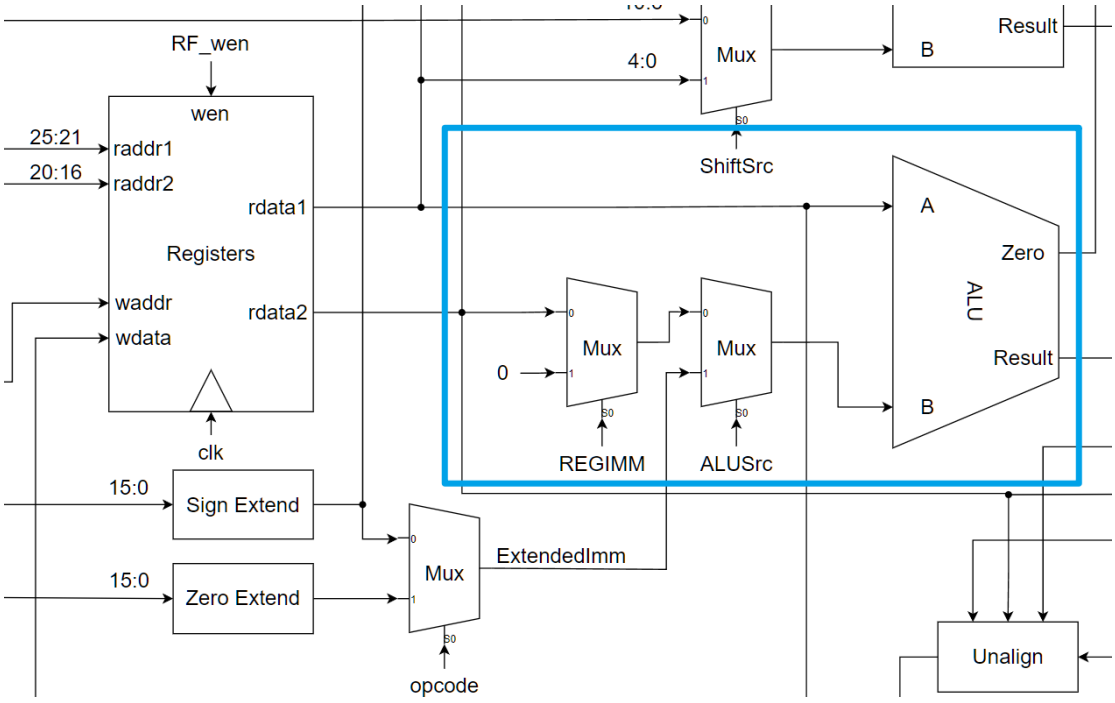


图 5 ALU

g) Shifter

```
shifter Shifter(
    .A          (RF_rdata2),
    .B          (ShiftSrc ? RF_rdata1[4:0] : Instruction[10:6]),
    .Shiftop    (Shiftop),
    .Result     (ShifterResult)
);
assign Shiftop = func[1:0];
```

如图，输入 A 为寄存器第二读端口值，输入 B 需要根据指令类型选择后生成。

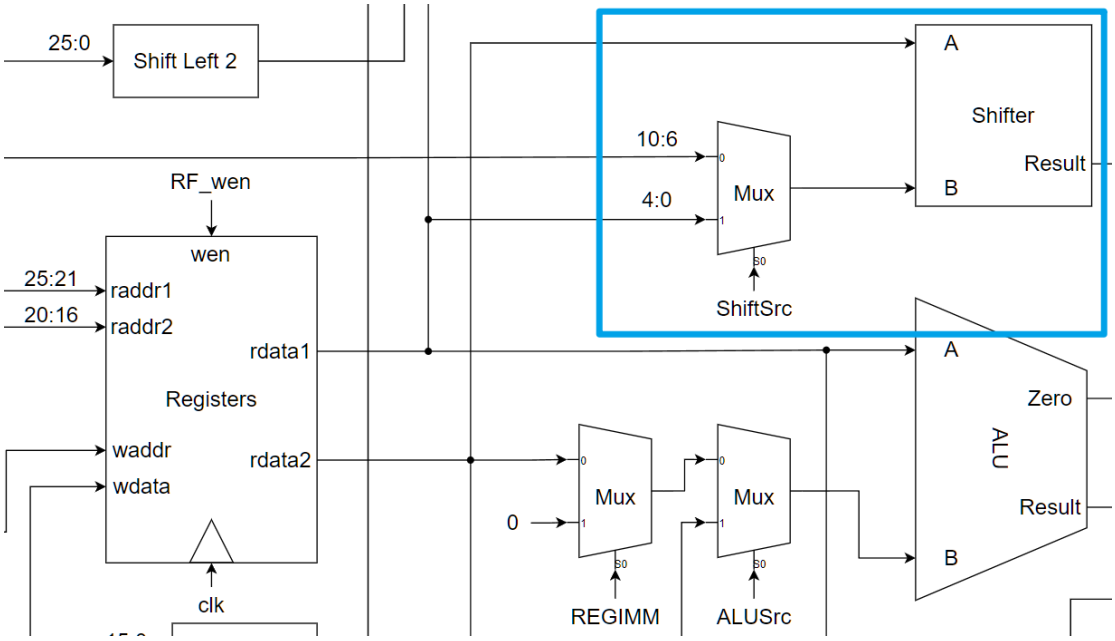


图 6 Shifter

h) 执行结果选择

```
assign Result = {(32){ALUEn}} & ALUResult
              | {(32){ShiftEn}} & ShifterResult;
```

如图，根据控制信号从 Shifter 和 ALU 中选择结果。

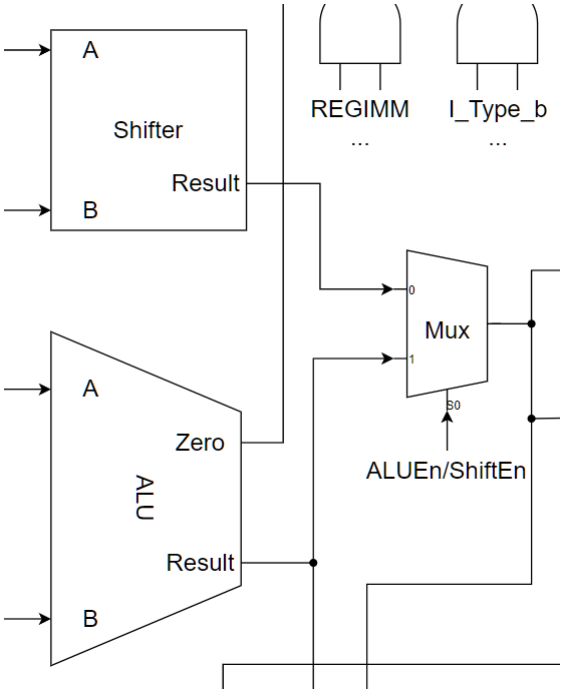


图 7 执行结果选择

i) 内存读取地址与写入数据生成

```
    //Data Memory
    assign Address      = Result & ~32'b11;
    assign Write_data   = opcode[2:0] == 3'b010 ? RF_rdata2 >> {~Result[1:0], 3'b0} : RF_rdata2 << {Result[1:0],
3'b0};
    assign Write_strb   =   {(4){~opcode[2] & opcode[1] & ~opcode[0]}} & {  // swl
                              Result[1] & Result[0], Result[1], Result[1] | Result[0], 1'b1
                            }
                        |   {(4){ opcode[2] & opcode[1] & ~opcode[0]}} & {  // swr
                              1'b1, ~(Result[1]&Result[0]), ~Result[1], ~(Result[1]|Result[0])
                            }
                        |   {(4){~opcode[1] | opcode[0]}} & {
                              ( Result[1] | opcode[1]) & ( Result[0] | opcode[0]),
                              ( Result[1] | opcode[1]) & (~Result[0] | opcode[0]),
                              (~Result[1] | opcode[1]) & ( Result[0] | opcode[0]),
                              (~Result[1] | opcode[1]) & (~Result[0] | opcode[0])
                            };
```

如图，根据指令类型选择输入方式与内容。

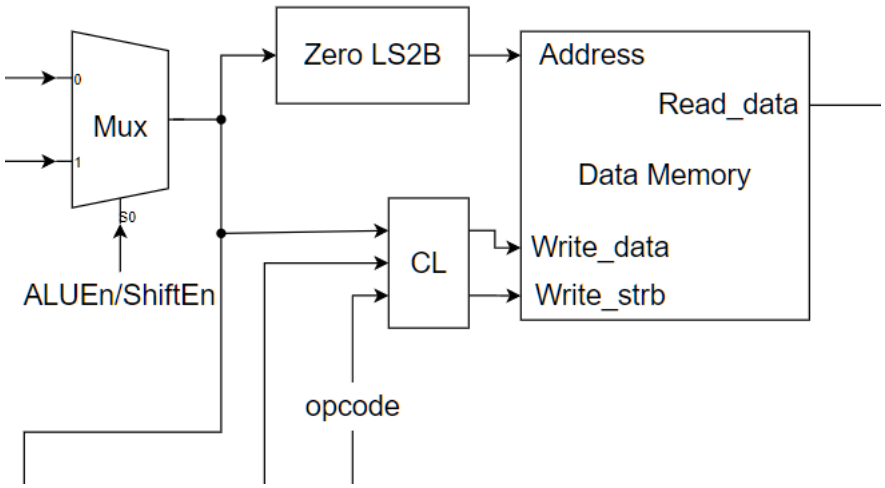


图 8 访存

j) 内存读取数据处理与写回

```
assign Read_data_shifted    = Read_data >> {Result[1:0], 3'b0};
assign Read_data_masked     = Read_data_shifted & {{(16){opcode[1]}}, {(8){opcode[0]}}, {(8){1'b1}}}
                            | {(32){~opcode[2] & Read_data_sign_bit}} & ~{{(16){opcode[1]}}, {(8){opcode[0]}}, {(8){1'b1}}};
assign Read_data_sign_bit   = Read_data_shifted[opcode[1:0]==2'b01 ? 15 : 7];
assign Read_data_unaligned  = {(32){~opcode[2]}} & (
                               (Read_data << {~Result[1:0], 3'b0}) | RF_rdata2 & ({(32){1'b1}} >> { Result[1:0], 3'b0})
                            )|{(32){opcode[2]}} & (
                               (Read_data >> { Result[1:0], 3'b0}) | RF_rdata2 & ({(32){1'b1}} << {~Result[1:0], 3'b0})
                            );
assign RF_wdata = {(32){MemtoReg & (~opcode[1] |  opcode[0])}} & Read_data_masked
                | {(32){MemtoReg & ( opcode[1] & ~opcode[0])}} & Read_data_unaligned
                | {(32){R_Type_mov}} & RF_rdata1
                | {(32){R_Type_jump | J_Type}} & PC + 8
                | {(32){I_Type_calc & (&opcode[3:0])}} & {Instruction[15:0], 16'd0}
                | {(32){R_Type_calc | R_Type_shift | I_Type_calc & ~&opcode[3:0]}} & Result;
```

如图，根据指令类型，依次对读取的数据进行移位、掩码、拼接操作；并在经过数据选择后写回寄存器。

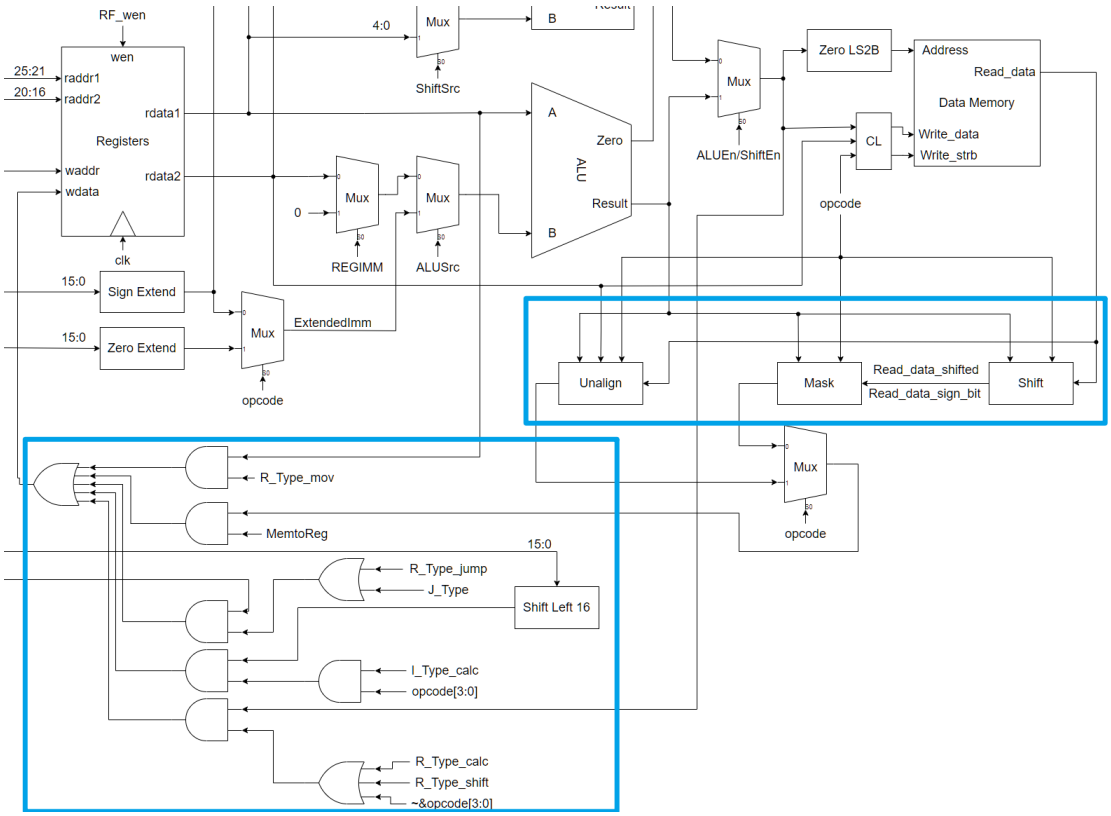


图 9 写回

2. 多周期 CPU

a) 下一状态生成

```
always @(*) begin
    case (current_state)
        IFSTATE: next_state <= IDSTATE;
        IDSTATE: begin
            if(|Instruction_cache) begin
                next_state <= EXSTATE;
            end else begin
                next_state <= IFSTATE;
            end
        end
        EXSTATE: begin
            if(R_Type | I_Type_calc | J_Type & opcode[0]) begin
                next_state <= WBSTATE;
            end else if(I_Type_mr | I_Type_mw) begin
                next_state <= MEMSTATE;
            end else begin
                next_state <= IFSTATE;
            end
        end
        MEMSTATE: begin
            if(I_Type_mr) begin
                next_state <= WBSTATE;
            end else begin
                next_state <= IFSTATE;
            end
        end
        WBSTATE: next_state <= IFSTATE;
        default: next_state <= IDSTATE;
    endcase
end
```

依据指令类型生成下一状态。状态转移图如课件中所示。

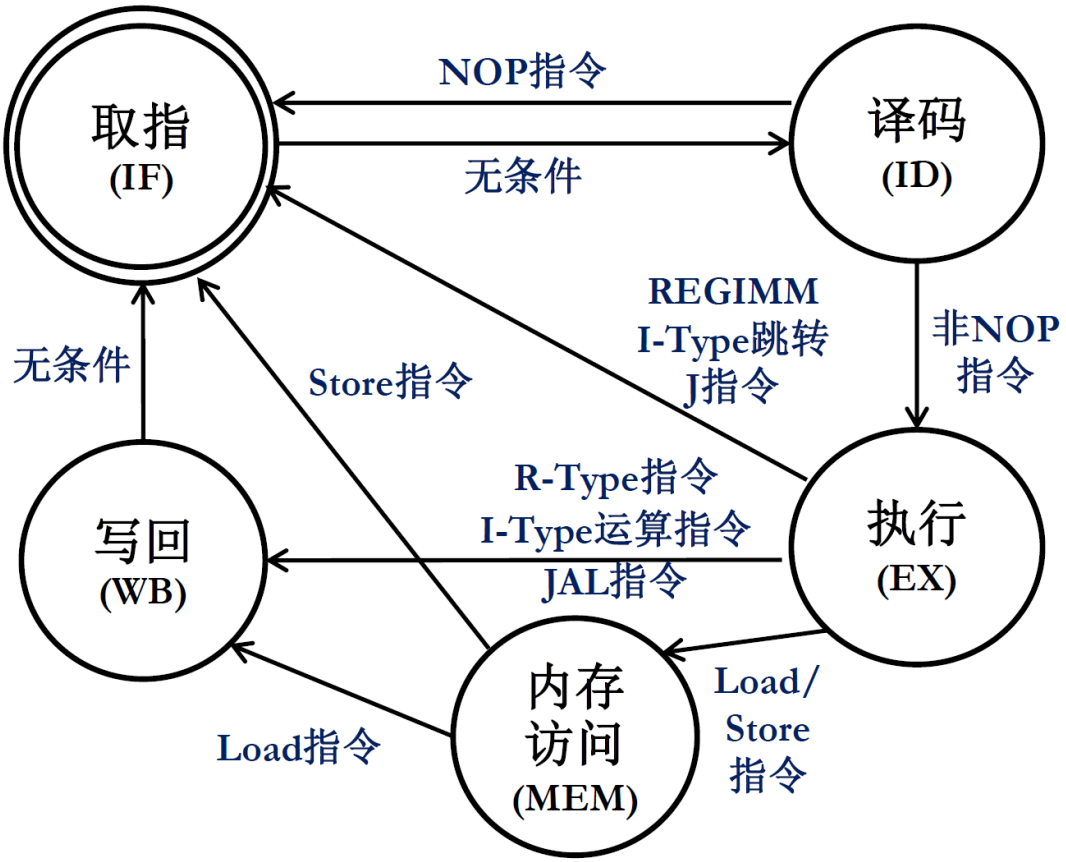


图 10 状态转移图

b) 状态转移

```
always @ (posedge clk) begin
    if(rst) begin
        current_state <= IFSTATE;
    end else begin
        current_state <= next_state;
    end
end
```

时钟上升沿触发。当复位信号有效时，将当前状态置位为初始取指状态；

否则，将当前状态更新为下一状态。

c) PC 更新

```
always @(posedge clk) begin
    if (rst) begin
        PC <= 32'd0;
    end else if (current_state == IFSTATE) begin
        PC <= PC + 4;
    end else if (current_state == EXSTATE) begin
        if (R_Type_jump) begin
            PC <= RF_rdata1;
        end else if (J_Type) begin
            PC <= {Instruction_cache[25:0], 2'b00};
        end else if ((Zero ^ (
            REGIMM & ~Instruction_cache[16]
            | I_Type_b & (opcode[0] ^ (opcode[1] & |RF_rdata1))
        )) & Branch) begin
            PC <= PC + {SignExtend[29:0], 2'b00};
        end
    end
end
```

时钟上升沿触发。依据复位信号、当前状态与当前指令类型，对 PC 进行更新。

d) 增加寄存器

主要包括 PC 寄存器、指令寄存器、寄存器堆取数寄存器、执行结果寄存器、内存读数寄存器等。

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，逻辑仿真和 FPGA 调试过程中的难点等）

本次实验中遇到的大部分问题均来源于代码编写过程中的缺漏与失误。解决方法是对比波形，主要可以归结为下述若干步骤：

1. 查看报错原因，利用错误信息定位出错时刻。错误信息包含了出错的数据线路与出错时刻，方便进一步找到错误根源。
2. 借助波形图获取出错时刻对应的 PC 值，利用汇编代码文件确定该时刻执行的指令。
3. 根据指令的功能需求与出错线路信息自下而上地寻找错误根源，对比设计文件判断出错原因，并加以改进。

经过总结反思，出错的主要原因在于对 Verilog 语法特性了解不足。

例如在 shifter 的算术右移的实现中，我最初采用了 A>>>B 的写法。但是实践证明，这种写法在无符号数类型的线路上实现的功能与 A>>B 相似，无法实现算术右移的预期功能。

经过查阅相关资料后，我把移位改成了($signed(A))>>>B，希望通过将 A 转为有符号数来实现算术右移。但是实践证明，这种写法也无法实现预期功能。猜测主要原因在于接受线路并非有符号数类型。

最终我采取了(A>>B) | ~((~32'b0)>>B) & {(32){A[31]}}的写法，利用逻辑右移与符号掩码实现了算术右移。

又如在 load，store 类型指令的访存、写回操作中，需要根据执行结果（即

地址）的末二位确定读取或写入数据的字段。其中需要完成对数据的移位操作，移位量取决于地址末二位乘 8.

为了避免使用乘法，我曾尝试将地址末二位左移三位来计算移位量：

```
Read_data >> (Result[1:0] << 3);
```

这种写法实际上是无法达到预期功能的，原因在于 Result[1:0]为一个 2bit 的线路，经过移位操作后仍然只能表示 2bit 的信息，但 8,16,24 等数的二进制表示均超过了 2bit。

我也曾尝试定义一个常量数组来解决问题。

```
wire [4:0] Location [1:0];
assign Location[2'b00] = 5'd0;
assign Location[2'b01] = 5'd8;
assign Location[2'b10] = 5'd16;
assign Location[2'b11] = 5'd24;
Read_data >> Location[Result[1:0]];
```

但这种写法不够优美，性能较差。在助教的启发下，我发现我陷入了 C 等编程语言的惯性思维，忽视了 Verilog 这种电路描述语言独有的语法特性。

最终解决方案如下：

```
Read_data >> {Result[1:0], 3'b0};
```

三、 对讲义中思考题（如有）的理解和回答

**ALUop 的编码有什么规律?**

1. 对于 R-Type 指令，利用 func[3:2]可以确定 ALU 运算类型（加减运算/逻辑运算/比较运算)。
2. 对于 I-Type 指令，利用 opcode[2:1]可以确定 ALU 运算类型。
3. 对于每个特定的运算类型，R-Type 指令的 func 与 I-Type 指令的 opcode 使用同一套逻辑即可得出 ALUop 编码，便于简化电路结构。

**表格中的 ALUop 编码是否还有优化空间?**

1. 在不修改指令与 ALU 实现的情况下，ALUop 编码已经达到最优化。可以采用卡诺图依据 func 或 opcode 的末四位对 ALUop 进行统一编码，但并不能有效减少电路延迟或减少逻辑元器件的使用量，反而降低了代码可读性。
2. 如果允许修改指令编码，可以令每条指令的[3:1]位表示对应的 ALUop。最低位用于表示同 ALUop 的不同指令（例如 add 与 addu)，以区分无符号数操作与有符号数操作。这样可以省去生成 ALUop 过程中的解码过程，将 func 或 opcode 的[3:1]作为 ALUop 的值直接输入。
3. 如果不允许修改指令编码，但允许修改 ALU 实现及 ALUop 标准，那么还有优化空间。例如在 ALU 中对于 sub,subu,slt,sltu 指令需要取补，可以令 ALUop 增加一位用于区分。

四、 在课后，你花费了大约 20 小时完成此次实验。

五、 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）

本次实验难度适中、资料完善，主要问题在于有些测试点测试时间略长（例如水仙花数）。

非常感谢助教老师的帮助，让我对 Verilog 的用法有了更深刻的了解，例如基于 always 语句的时序逻辑与组合逻辑中 if 语句使用上的差异等细节。