中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929010 姓名：贾城昊 专业：计算机科学与技术

实验序号：3 实验名称：定制 MIPS 功能处理器设计（真实内存、外设与性能计数器访问）

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。

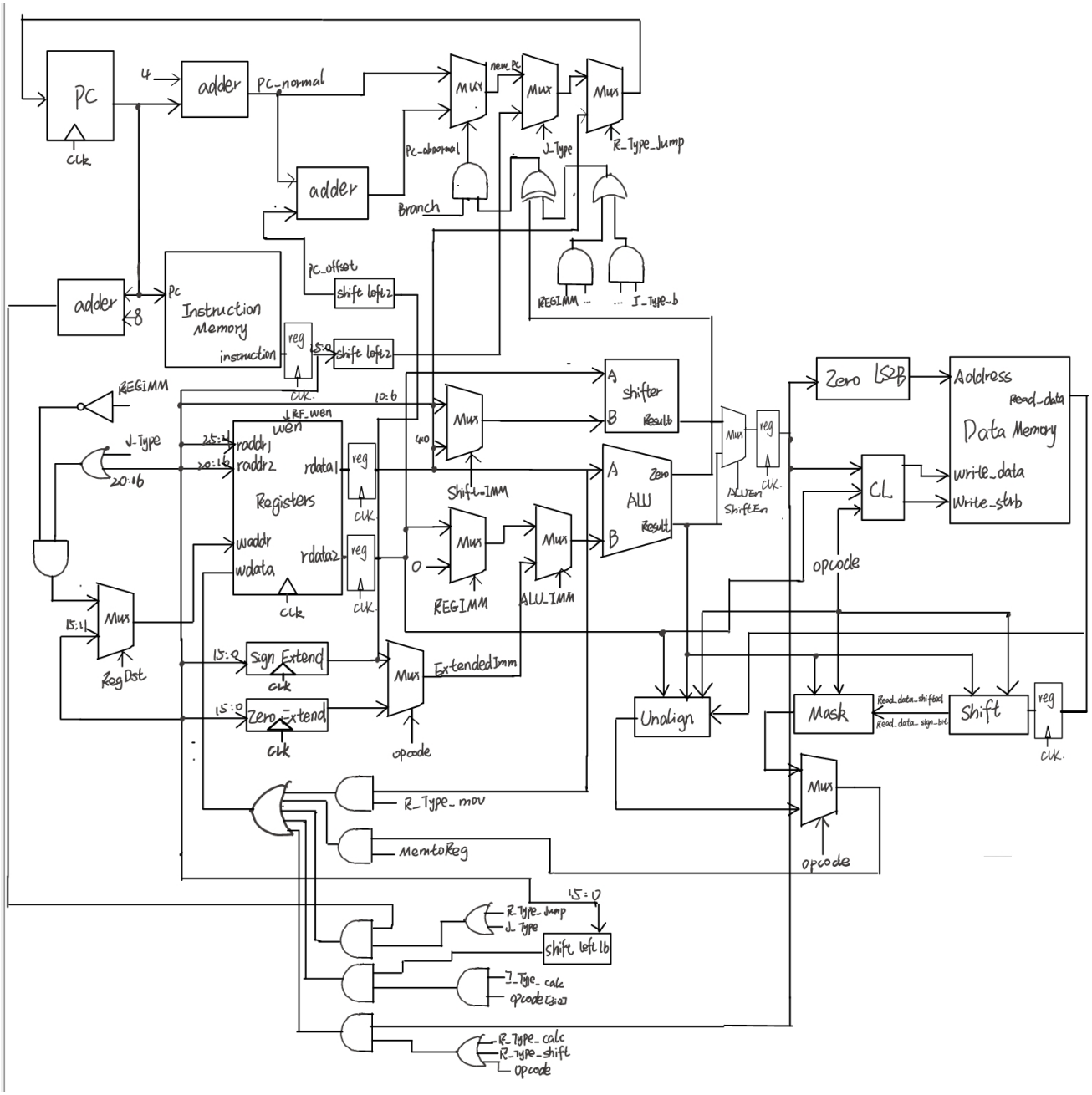
注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

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。

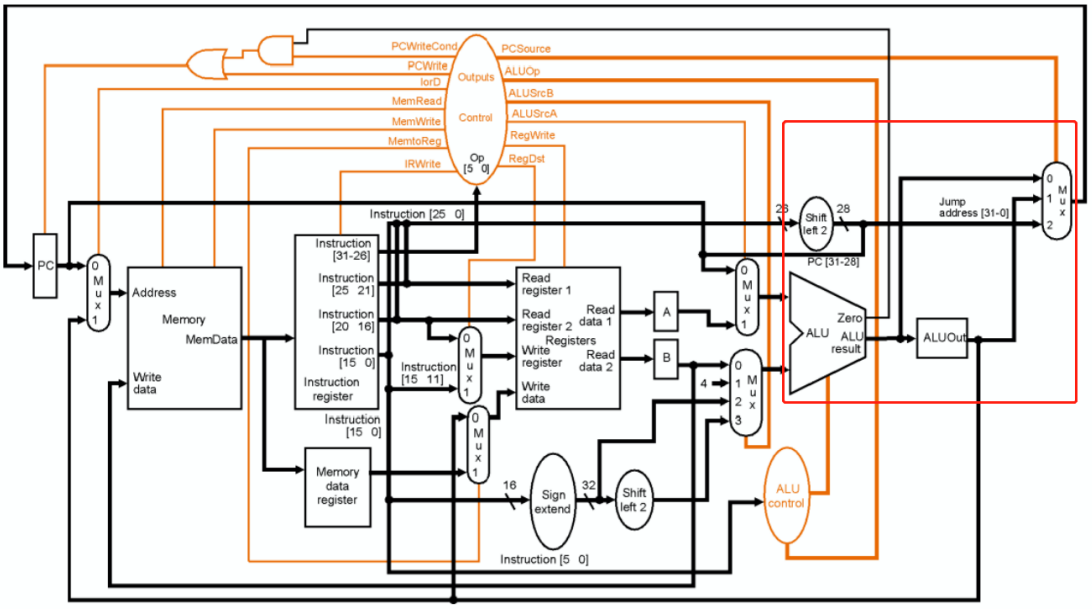
一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图{自行画图，推荐用 PPT 画逻辑结构框图，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

1） 逻辑电路图

该实验的电路与第二次实验的电路较为相似，只是增加了多周期状态机与几个寄存器（实际上与 pj2 的多周期处理器的结构基本相同），下面是在单周期处理器上增加了寄存器后的电路结构图，但是对于 ALU 的复用在下图没有展现。



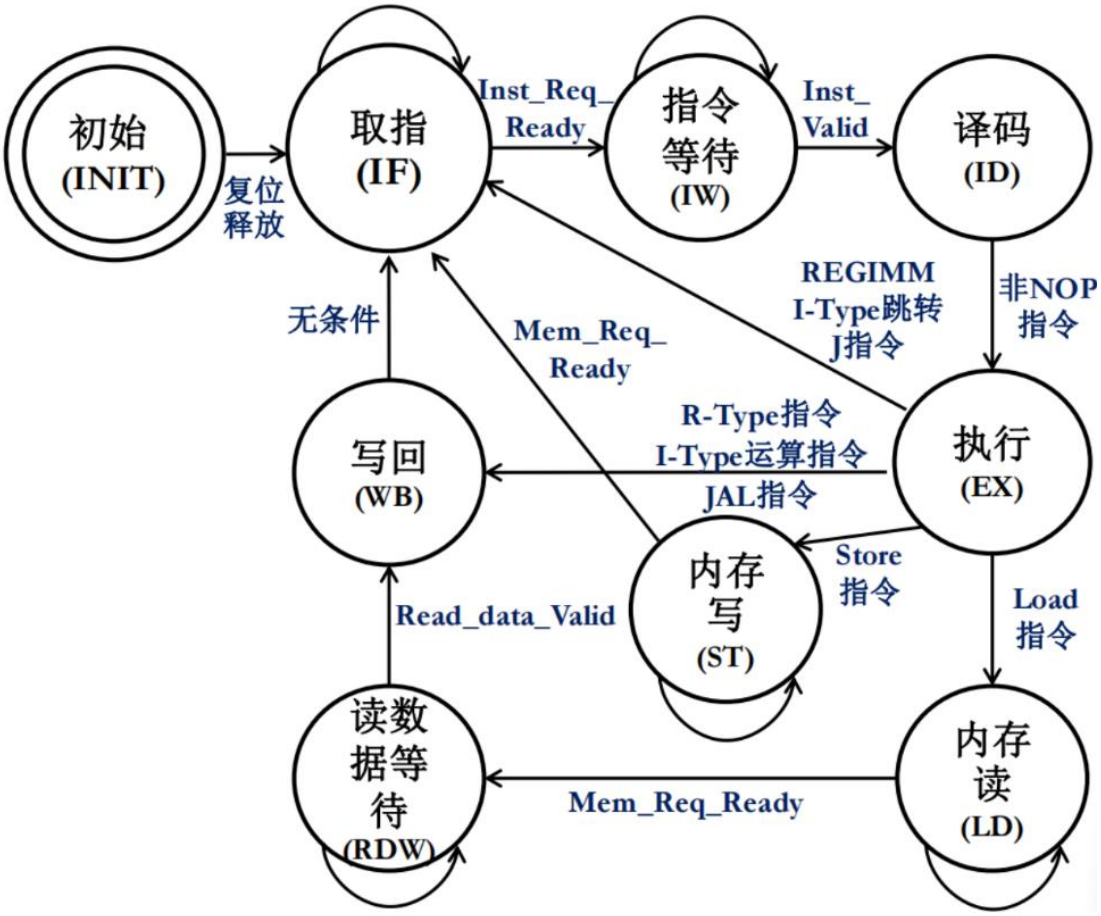
下面这张图是更新 PC 逻辑中对 ALU 的复用的示意图，虽然较为简略，但与本实验中的逻辑相似（见图中画红框的部分）



2） 关键 RTL 代码段：

a） 状态转移

状态转移图与课程 PPT 给出的一致，如下：



一共是 9 个状态，对这 9 个状态的编码如下图，采用独热码的方式。

```
localparam INIT =9'b000000001;
localparam IF   =9'b000000010;
localparam IW   =9'b000000100;
localparam ID   =9'b000001000;
localparam EX   =9'b000010000;
localparam ST   =9'b000100000;
localparam LD   =9'b001000000;
localparam RDW  =9'b010000000;
localparam WB   =9'b100000000;
```

状态机的描述采用的是三段式：

I. “第一段”采用 always 时序逻辑，描述状态寄存器的同步跳转。

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

II. “第二段”采用 always 组合逻辑，根据当前状态机状态和输入信号，描述下一状态的计算逻辑。

```
always @(*) begin
        case (current_state)
                INIT: next_state <= IF;
                IF: begin
                        if (Inst_Req_Ready) begin
                                next_state <= IW;
                        end
                        else begin
                                next_state <= IF;
                        end
                end
                IW: begin
                        if (Inst_Valid) begin
                                next_state <= ID;
                        end
                        else begin
                                next_state <= IW;
                        end
                end
                ID: begin
                        if (|reg_Instruction) begin
                                next_state <= EX;
                        end else begin
                                next_state <= IF;
                        end
                end
```

```
EX: begin
        if (R_Type | I_Type_calc | J_Type & opcode[0]) begin
                next_state <= WB;
        end
        else if (I_Type_mr) begin
                next_state <= LD;
        end
        else if (I_Type_mw) begin
                next_state <= ST;
        end
        else begin
                next_state <= IF;
        end
end
ST: begin
        if (Mem_Req_Ready) begin
                next_state <= IF;
        end
        else begin
                next_state <= ST;
        end
end
```

```
                LD: begin
                        if (Mem_Req_Ready) begin
                                next_state <= RDW;
                        end
                        else begin
                                next_state <= LD;
                        end
                end
                RDW: begin
                        if (Read_data_Valid) begin
                                next_state <= WB;
                        end
                        else begin
                                next_state <= RDW;
                        end
                end
                WB: next_state <= IF;
                default: next_state <= INIT;
        endcase
end
```

III. “第三段”采用 always 时序逻辑或 assign 组合逻辑，根据当前状态机的状态，描述不同输出信号的变化。

下面是第三段的 always 时序逻辑部分：

```
//reg_Instruction
always @(posedge clk) begin
        if (current_state == IW && Inst_Valid) begin
                reg_Instruction <= Instruction;
        end
end


//EXTEND_IMM
always @(posedge clk) begin
        if (current_state == ID) begin
                SignExtend <= {{(16){reg_Instruction[15]}}, reg_Instruction[15:0]};
                ZeroExtend <= {16'b0, reg_Instruction[15:0]};
        end
end
```

```
//PC
always @(posedge clk) begin
        if (current_state == ID) begin
                PC_normal <= PC;
        end
end

always @(posedge clk) begin
        if (rst) begin
                PC <= 32'd0;
        end
        else if (current_state == IW && Inst_Valid) begin
                PC <= ALUResult;
        end
        else if (current_state == EX) begin
                if (R_Type_jump) begin
                        PC <= RF_rdata1;
                end
                else if (J_Type) begin
                        PC <= {PC_normal[31:28], reg_Instruction[25:0], 2'b00};
                end
                else if ((Zero ^ (REGIMM & ~reg_Instruction[16] |
                                I_Type_b & (opcode[0] ^ (opcode[1] & |RF_rdata1)))) & Branch) begin //branch
                        PC = Result; // ID阶段算出的PC
                end
        end
end
```

```
//register
always @(posedge clk) begin
        if (current_state == ID) begin
                RF_rdata1 <= wire_RF_rdata1;
                RF_rdata2 <= wire_RF_rdata2;
        end
end


//Result
always @(posedge clk) begin
        if ((current_state == EX) || (current_state == ID)) begin
                Result <= {(32){ALUEn}} & ALUResult |
                        {(32){ShiftEn}} & ShifterResult; //Choose Result
        end
end


//WB
always @(posedge clk) begin
        if (current_state == RDW && Read_data_Valid) begin
                Read_data_reg <= Read_data;
        end
end
```

（注：其中重要部分在后面可能会再次提到）

b) 基于真实内存的多周期访问逻辑

```
//Handshake Signal
assign Inst_Req_Valid           = current_state == IF;
assign Inst_Ready               = current_state == IW || current_state == INIT;
assign MemWrite                 = current_state == ST;
assign MemRead                  = current_state == LD;
assign Read_data_Ready          = current_state == RDW || current_state == INIT;
```

如代码所示，在 IF 状态拉高 Inst_Req_Valid，在接收到 Inst_Req_Ready 有效时进入 IW 状态等待指令。在 IW 状态拉高 Inst_Ready,在接收到 Inst_Valid 有效时进入 ID 状态。对于 Store 指令，在 ST 状态拉高 MemWrite,在接收到 Mem_Req_Valid 有效时返回 IF 状态。对于 Load 指令，在 LD 状态拉高 MemRead, 在接收到 Mem_Req_Valid 有效时进入 RDW 等待读数,当 Read_data_Ready 有效

时返回 IF 状态。这些都是多周期内存访问的握手信号。

其它控制信号较单周期相比也有部分变化，明显改变控制信号有 RF_wen，ALUEn。这 2 个信号需要对当前状态进行考虑。RF_wen 只能在写回阶段拉高，而 ALUEn 需要在译码阶段无条件拉高（因为 PC 更新复用 ALU 的原因)。同时，其它的译码信号均是根据 reg_Instruction 生成

c) PC 更新的变化

```
//PC
always @(posedge clk) begin
        if (current_state == ID) begin
                PC_normal <= PC;
        end
end

always @(posedge clk) begin
        if (rst) begin
                PC <= 32'd0;
        end
        else if (current_state == IW && Inst_Valid) begin
                PC <= ALUResult;
        end
        else if (current_state == EX) begin
                if (R_Type_jump) begin
                        PC <= RF_rdata1;
                end
                else if (J_Type) begin
                        PC <= {PC_normal[31:28], reg_Instruction[25:0], 2'b00};
                end
                else if ((Zero ^ (REGIMM & ~reg_Instruction[16] |
                             I_Type_b & (opcode[0] ^ (opcode[1] & |RF_rdata1)))) & Branch) begin //branch
                        PC = Result; // ID阶段算出的PC
                end
        end
end
```

```
//PC_abnormal
wire [31:0] PC_offset;
assign PC_offset        = {SignExtend[29:0], 2'b00}; //offset*4
assign PC_abnormal      = PC_normal + PC_offset;
```

PC+4 的计算使用了 ALU,并在 IW 阶段且 Inst_Valid 有效时进行更新，而 branch 指令的目标地址在译码阶段就计算出来并存在寄存器中，在执行阶

段进行 branch 指令和 jump 指令的目标地址的更新。这样保证了除了 ALU 外，没有其它的加法器。

d) ALU 的变化

```
//ALUop && Shiftop
assign ALUop = (current_state == EX) ?
            ({(3){R_Type_calc}} & {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)
            : 3'b010;
```

```
//ALU
assign ALU_A = {(32){(current_state == IW && Inst_Valid) | (current_state == ID)}} & PC |
               {(32){current_state == EX}} & RF_rdata1;
assign ALU_B = {(32){(current_state == IW && Inst_Valid)}} & {29'b0, 3'b100} |
               {(32){current_state == ID}} & {{{(14){reg_Instruction[15]}}, reg_Instruction[15:0]}, 2'b00} |
               {(32){current_state == EX}} & (ALU_IM ? ExtendedImm : REGIMM ? 32'b0 :RF_rdata2);

alu ALU(
        .A              (ALU_A),
        .B              (ALU_B),
        .ALUop          (ALUop),
        .Overflow       (Overflow),
        .CarryOut       (CarryOut),
        .Zero           (Zero),
        .Result         (ALUResult)
);
```

对于 ALUop，执行阶段的编码逻辑与单周期相同，但取指和译码阶段，均是 010（加法），对于 ALU 的端口 A 和 B，执行阶段与单周期相同，但 IW 阶段且 Inst_Valid 有效时，A 为 PC，B 为 4（计算 PC+4），译码阶段，A 为 PC，B 为 offset，计算（PC+offset，即分支指令的目标地址）。

e) UART 控制器的简单驱动程序

```
int
puts(const char *s)
{
        //TODO: Add your driver code here
        int i;
        for(i=0;s[i]!='\0';i++){
                while((*(volatile unsigned int *)((char *)uart+UART_STATUS)) & UART_TX_FIFO_FULL);//判断队列是否已满
                *((char *)uart+UART_TX_FIFO) = s[i];
        }
        return i;
}
```

While 循环用于检查发送队列是否已满；

若未满,则向发送队列入口寄存器写入字符串的第 i 位字符。

f) 性能计数器

```
//Performance Counter
        //周期计数器
        reg  [31:0] cycle_cnt;
        always @ (posedge clk) begin
                if (rst) begin
                        cycle_cnt <= 32'd0;
                end else begin
                        cycle_cnt <= cycle_cnt + 32'd1;
                end
        end
        assign cpu_perf_cnt_0 = cycle_cnt;

        //指令计数器
        reg  [31:0] inst_cnt;
        always @ (posedge clk) begin
                if (rst) begin
                        inst_cnt <= 32'd0;
                end else if (current_state == ID) begin
                        inst_cnt <= inst_cnt + 32'd1;
                end
        end
        assign cpu_perf_cnt_1 = inst_cnt;
```

```
    //读内存计数器
    reg  [31:0] mr_cnt;
    always @ (posedge clk) begin
            if (rst) begin
                    mr_cnt <= 32'd0;
            end else if (current_state == ID && I_Type_mr) begin
                    mr_cnt <= mr_cnt + 32'd1;
            end
    end
    assign cpu_perf_cnt_2 = mr_cnt;

    //写内存计数器
    reg  [31:0] mw_cnt;
    always @ (posedge clk) begin
            if (rst) begin
                    mw_cnt <= 32'd0;
            end else if (current_state == ID && I_Type_mw) begin
                    mw_cnt <= mw_cnt + 32'd1;
            end
    end
    assign cpu_perf_cnt_3 = mw_cnt;
```

```
//取指请求延误周期计数器
reg  [31:0] inst_req_delay_cnt;
always @ (posedge clk) begin
        if (rst) begin
                inst_req_delay_cnt <= 32'd0;
        end else if (current_state == IF && next_state == IF) begin
                inst_req_delay_cnt <= inst_req_delay_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_4 = inst_req_delay_cnt;

//取值延误周期计数器
reg  [31:0] inst_delay_cnt;
always @ (posedge clk) begin
        if (rst) begin
                inst_delay_cnt <= 32'd0;
        end else if (current_state == IW && next_state == IW) begin
                inst_delay_cnt <= inst_delay_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_5 = inst_delay_cnt;
```

```
//读内存请求延误周期计数器
reg  [31:0] mr_req_delay_cnt;
always @ (posedge clk) begin
        if (rst) begin
                mr_req_delay_cnt <= 32'd0;
        end else if (current_state == LD && next_state == LD) begin
                mr_req_delay_cnt <= mr_req_delay_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_6 = mr_req_delay_cnt;

//从内存获取数据延误周期计数器
reg  [31:0] rdw_delay_cnt;
always @ (posedge clk) begin
        if (rst) begin
                rdw_delay_cnt <= 32'd0;
        end else if (current_state == RDW && next_state == RDW) begin
                rdw_delay_cnt <= rdw_delay_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_7 = rdw_delay_cnt;
```

```
//写内存请求延误周期寄存器
reg  [31:0] mw_req_delay_cnt;
always @ (posedge clk) begin
        if (rst) begin
                mw_req_delay_cnt <= 32'd0;
        end else if (current_state == ST && next_state == ST) begin
                mw_req_delay_cnt <= mw_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_8 = mw_req_delay_cnt;

//分支指令计数器
reg  [31:0] branch_inst_cnt;
always @ (posedge clk) begin
        if (rst) begin
                branch_inst_cnt <= 32'd0;
        end else if (current_state == ID && (I_Type_b || REGIMM)) begin
                branch_inst_cnt <= branch_inst_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_9 = branch_inst_cnt;
```

```
//跳转指令计数器
reg  [31:0] jump_inst_cnt;
always @ (posedge clk) begin
        if (rst) begin
                jump_inst_cnt <= 32'd0;
        end else if (current_state == ID && (R_Type_jump || J_Type)) begin
                jump_inst_cnt <= jump_inst_cnt + 32'd1;
        end
end
assign cpu_perf_cnt_10 = jump_inst_cnt;
```

如图，一共定义了 11 个性能计数器，其功能如下表所示：

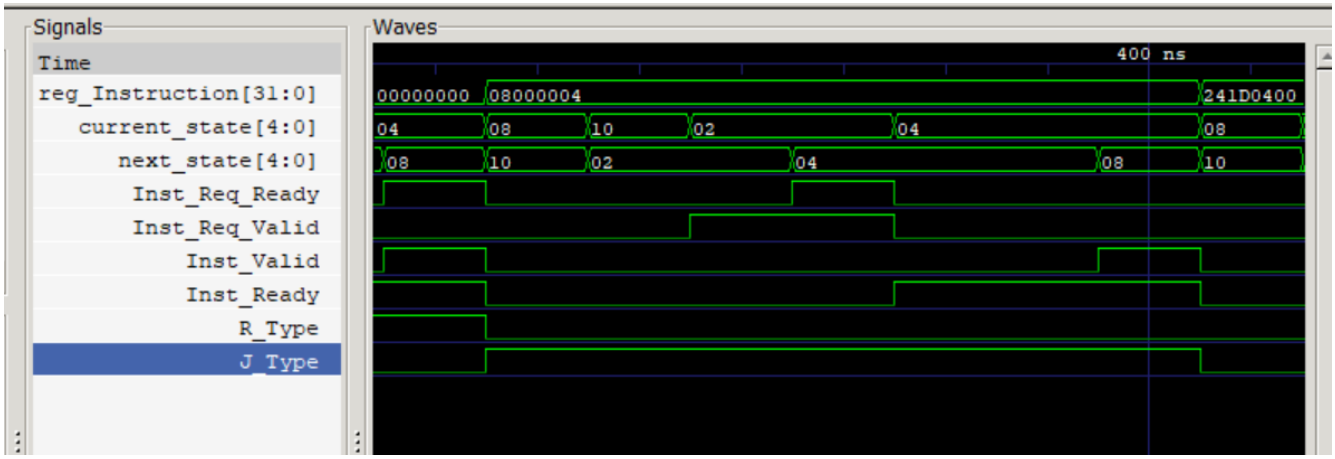
名称	端口	描述
cycle_cnt	cpu_perf_cnt_0	周期计数器
inst_cnt	cpu_perf_cnt_1	指令计数器
mr_cnt	cpu_perf_cnt_2	读内存计数器
mw_cnt	cpu_perf_cnt_3	写内存计数器
inst_req_delay_cnt	cpu_perf_cnt_4	取指请求延误周期计数器
inst_delay_cnt	cpu_perf_cnt_5	取值延误周期计数器
mr_req_delay_cnt	cpu_perf_cnt_6	读内存请求延误周期计数器
rdw_delay_cnt	cpu_perf_cnt_7	从内存获取数据延误周期计数器
mw_req_delay_cnt	cpu_perf_cnt_8	写内存请求延误周期寄存器
branch_inst_cnt	cpu_perf_cnt_9	写内存请求延误周期寄存器
jump_inst_cnt	cpu_perf_cnt_10	跳转指令计数器

最终，运行 micobench 的一个程序对性能计数器的打印结果如下：

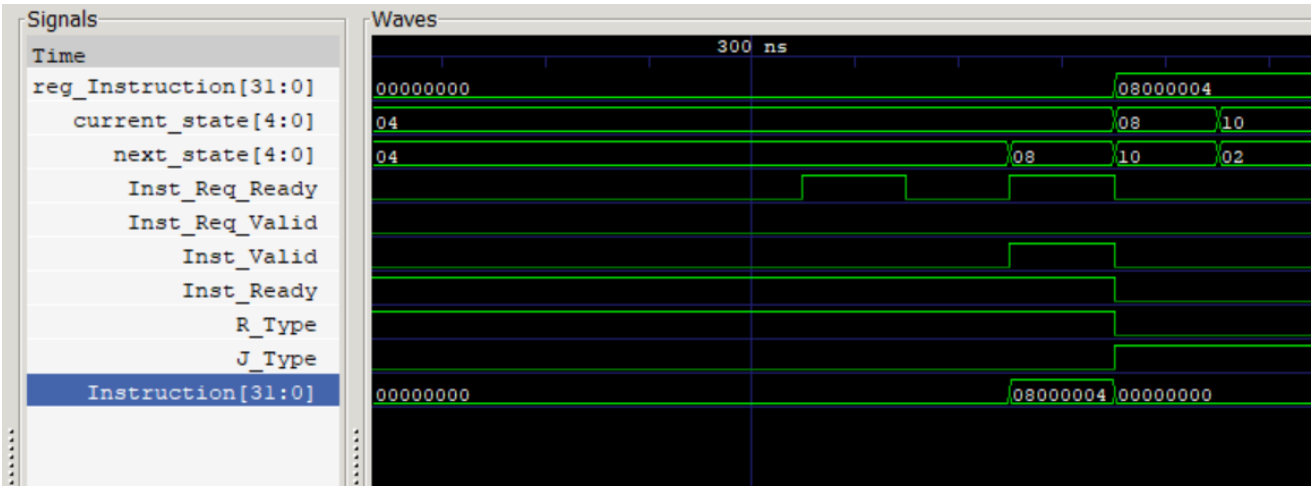
```
Result:
        Cycles: at least 52493029
        Instruction Count: at least 728307
        Memory Read Instruction Count: at least 11595
        Memory Write Instruction Count: at least 7711
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 48207856
        Memory Read Request Delay Cycles: at least 11599
        Read Data Delay Cycles: at least 782363
        Memory Write Request Delay Cycles: at least 7711
        Branch Instruction Count: at least 106130
        Jump Instruction Count: at least 693
```

3)　波形图

下面是一些波形图的展示：



例如在上图中，当 Inst_Req_Ready 与 Inst_Reg_Valid 都为 1 的时候，next_state 才会进行更新为 IW，同理当 Inst_Ready 与 Inst_Valid 都为 1 的时候，next_state 才会进行更新为 ID，满足真实内存的握手要求。



再比如上图中，当 Inst_Ready 与 Inst_Valid 都为 1 的时候，reg_Instruction 才会更新为 Instruction 的值。

（注：由于在本次实验中，如果仿真正确便无法下载波形图，所以上面波形图是在之前提交出错的版本出错前所跑出来的，它确实表现了各个信号的含义，以及状态机之间的转换。但由于是出错版本的波形，在这，本人就不进行进一步的展示了，也希望平台能对这个问题给出一点解决办法）

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）

本次实验的逻辑大部分地方跟单周期相同，主要的是添加几个寄存器，部分译码逻辑的改变和 PC 更新对 ALU 的复用。硬件部分的难点不多，如果需要挑选的话，本人认为具体如下：

难点：

1. 需要新添哪些寄存器

2．部分译码逻辑的改变

3．计算 PC 时候对 ALU 的复用（不再用额外的加法器）

对难点的思考：

1．需要新添哪些寄存器

哪些地方需要添加寄存器，总的规则就是如果数据需要保存，则需要添加寄存器，而需要新添的大部分寄存器在理论课也给了，而本人新添了对于立即数有符号和无符号扩展的结果添加了寄存器进行储存，以及对 PC+4 的值用一个寄存器进行保存，以免其发生变化。

综上，新添的寄存器主要包括指令的寄存器，寄存器堆数取数寄存器，执行结果寄存器，内存读数寄存器，立即数扩展的寄存器。

2．部分译码逻辑的改变

不是所有的译码逻辑均需要改变（不过单周期是通过输入的指令直接译码，多周期肯定是通过指令的寄存器的值进行译码）。而通过研究，发现主要有改变控制信号有 MemWrite，MemRead，RF_wen，ALUEn。这几个信号需要对当前状态进行考虑。

3．计算 PC 时候对 ALU 的复用（不再用额外的加法器）

这个理论课上对原理讲的很清楚，但相应的，这样的改变会导致很多地方的变化，最直接的，与 ALU 相关的输入都要发

生变化。如 ALUop 和 ALUEn 需要进行对应的调整，ALUEn 改变上一点已经讲了，ALUop 在取指和译码阶段均为 010（加法操作)。然后 ALU 的输入 A，B 也要有相应的变化，IW 阶段且 Inst_Valid 有效时是 PC 和 4,译码阶段是 PC 和对应的 offset，执行阶段与单周期相同。

然后 PC 的变化也导致一些其它地方逻辑的变化，例如对于 jal，jalr 的写回数据也要改变（因为 PC 会发生变化)，而本人的做法是用了一个 PC_normal 寄存器来存 PC+4 的值。再就是计算分支指令目标地址时候的立即数扩展的值也要有相应的改变。

遇到的重要 bug 以及解决过程:

本次实验遇到的 bug 大多都是在软件的代码上。硬件上的出错较少，可能是因为上次实验写过多周期处理器，像实现对 PC 的复用上，jal，jalr 的写回数据要改变，计算分支指令目标地址时候的立即数扩展的值不能直接用立即数扩展单元算的值，ALUop 和 ALUEn 需要进行对应的调整，ALU 的输入 A，B 也要有相应的变化,新添的寄存器在什么状态进行更新这些设计时容易出现的 bug 之前已经解决了，下面列举了本次实验在软件代码编写上遇到的一个主要的 bug:

1．puts 函数中指令使用不当，导致 UART 控制器寄存器偏移

地址计算错误。

在 puts 函数中，需要读取队列状态寄存器 STAT_REG 来判断当前队列是否已满,并将数据写入发送队列入口寄存器 TX FIFO。这里两个寄存器地址等于基地址 uart 加上相应的偏移量 UART_STATUS 或 UART_TX_FIFO。我最初的代码如下:

(volatile unsigned int *)(uart+UART_STATUS)

但事实上,这是存在问题的。经过调试后,发现问题出在 uart 和 UART_STATUS 相加的过程中。这里的 uart 被定义为 volatile unsinged int*类型, UART_STATUS 被宏定义为 Ox08,两者相加时按照 unsigned int 类型指针的加法,每加一相当于指针偏移 sizeof(unsinged int)个字节;但需要的是让它每加一偏移 1 个字节,因此需要在加之前将 uart 强制类型转换为 char*类型:

(volatile unsigned int *)((char *)uart+UART_STATUS)

这样，因为 char 数据长度为 1 字节，便解决了这个问题。

三、 对讲义中思考题（如有）的理解和回答

1. 上图中 volatile 关键字的作用是什么？如果去掉会出现什么后果?

请同学们在实验报告中给出思考及实验对比结果

volatile 关键字在 C 语言中被用来指示“被定义的变量可能随时会发生改变”。编译器面对带有 volatile 关键字的变量,将不会对其优化。

例如，在 puts 函数中的 while 循环中:

```
int
puts(const char *s)
{
        //TODO: Add your driver code here
        int i;
        for(i=0;s[i]!='\0';i++){
                while((*(volatile unsigned int *)((char *)uart+UART_STATUS)) & UART_TX_FIFO_FULL);//判断队列是否已满
                *((char *)uart+UART_TX_FIFO) = s[i];
        }
        return i;
}
```

这里如果不加 volatile，那么编译器有可能将初次进入 while 循环时该地址的取值作为每轮 for 循环中 while 循环的判断条件。这是因为编译器发现，在每轮循环中, while 判断条件将不会发生改变，从而编译时进行优化，将第一次读出的数据存在寄存器里。

因此,按照这个理论来讲,如果初次访问时当前队列状态寄存器显示已满,那么程序将死循环直至超时报错;如果初次访问时当前队列状态寄存器显示为空(未满),那么程序将无视 while 等待逻辑,不顾当前队列是否为满而向 TX FIFO 中写入。

但如果加上 volatile，编译器将不会优化这个变量。因此，在每轮 while 循环中,程序都会根据该地址对应的当前值来更新 while 判断条件。从而实现“一旦状态寄存器显示当前队列未满，便跳出 while 循环”的功能。

为此，我专门删去这个关键词进行了一次实验，实验结果证实了这个猜想。

以下是正常的 fpga 运行结果:

```
Result:
        Cycles: at least 1193080
        Instruction Count: at least 16496
        Memory Read Instruction Count: at least 344
        Memory Write Instruction Count: at least 129
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 1089420
        Memory Read Request Delay Cycles: at least 348
        Read Data Delay Cycles: at least 23396
        Memory Write Request Delay Cycles: at least 129
        Branch Instruction Count: at least 1847
        Jump Instruction Count: at least 4
```

```
Result:
        Cycles: at least 52493029
        Instruction Count: at least 728307
        Memory Read Instruction Count: at least 11595
        Memory Write Instruction Count: at least 7711
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 48207856
        Memory Read Request Delay Cycles: at least 11599
        Read Data Delay Cycles: at least 782363
        Memory Write Request Delay Cycles: at least 7711
        Branch Instruction Count: at least 106130
        Jump Instruction Count: at least 693
```

下面是删去 volatile 后的运行结果：

```
Result:
        Cycles: at leas9        Instruction Coul        Memory Read Ins         Memory Write Inn        Instructi l     Instruction
FetC3   Memory Read Reqa
        Read Data Delay:        Memory Write Rel
        Branch Instruct:        Jump Instruct benhtime 10969.69ms
```

```
[queen] Queen pl * Passed.
Result:
       Cycles: at leas0        Instruction Coul        Memory Read InsC        Memory Write Inn        Instructioc     Instruction
fety3   Memory Read Reqa
       Read Data Delay:        Memory Write Ree3       Branch Instructn        Jump Instructio:ben time 10969.41ms
reset: before MMIO access...
```

可见，大部分字符都没有正常打印。

猜测是因为初次访问时,队列状态寄存器显示空。在接下来的若干次 for 循环中，程序无视 while 循环的等待逻辑，导致不顾当前队列是否为满而向 TX FIFO 中写入。这样就导致了有些字符在还未输出时便已被后边的字符覆盖,造成最终的打印不全。

2. 处理器内部是否需要实现异步串行通信协议?

对于这个问题，需要指出异步串行通信协议不是必须的，但实现它有诸多好处，但同时也有很多困难。所以是否需要往往需要针对不同的计算环境，权衡性能和可靠性以及仔细考虑对设计和验证的复杂性。

在处理器内部，异步串行通信协议是一种常见的设计选择，尤其在多核处理器和并行计算环境中。异步串行通信协议允许处理器内的不同模块或功能单元以独立的方式进行通信，而无需依赖于全局时钟信号。

传统的同步通信协议使用全局时钟来同步各个模块的操作，这意味着所有操作都需要在时钟的边沿进行。这样的设计在小规模的处理器中可能是可行的，但在大规模和高性能的处理器中，全局时钟可能成为性能瓶颈。此外，同步通信协议还需要处理时钟抖动、时钟延迟等问题。

相比之下，异步串行通信协议通过使用独立的控制信号和数据传输手段，允许模块之间以自己的节奏进行通信。这样可以提高处理器的性能和可伸缩性，减少时钟相关的问题。异步串行通信协议可以提供更高

的吞吐量、更低的延迟和更好的功耗效率。

然而，实现异步串行通信协议也带来了一些挑战。异步设计需要更多的逻辑和状态管理，因此设计和验证复杂度更高。同时，异步通信也可能引入新的时序问题，如冲突、死锁和元数据错误。

因此，在处理器内部实现异步串行通信协议需要仔细考虑设计和验证的复杂性，并在性能和可靠性之间做出权衡。

四、 在课后，你花费了大约___10__小时完成此次实验。

本次实验硬件部分逻辑与 pj2 的多周期处理器相似,所以课下耗费的时间较少。

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）

这次实验并不复杂，难度适中，资料也特别完善。唯一的问题可能是测试样例跑的时间太长了，例如水仙花数。但本次实验给出了云端 FPGA 仿真加速工具的使用方法，算是较好的解决了这个问题。

通过这次实验,我受益匪浅：首先，对真实内存的访问逻辑以及针对真实内存的 cpu 的设计有了更深的了解。其次，通过这次实验，我进一步熟悉了状态机的三段式写法。然后，通过本次实验，我还了解到了 C 语言的部分语法，例如 volatile 关键字。除此之外，通过本次实验，我还了解

到了处理器对 I/O 端口的访问。

另外，gitlab 平台上如果仿真正确就无法下载波形图，这对实验报告的撰写造成了一定的困扰，希望能给出一定的解决办法。

最后，感谢助教在本次实验上的启发与耐心帮助。