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：3　实验名称：定制 MIPS 功能型处理器设计

（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。

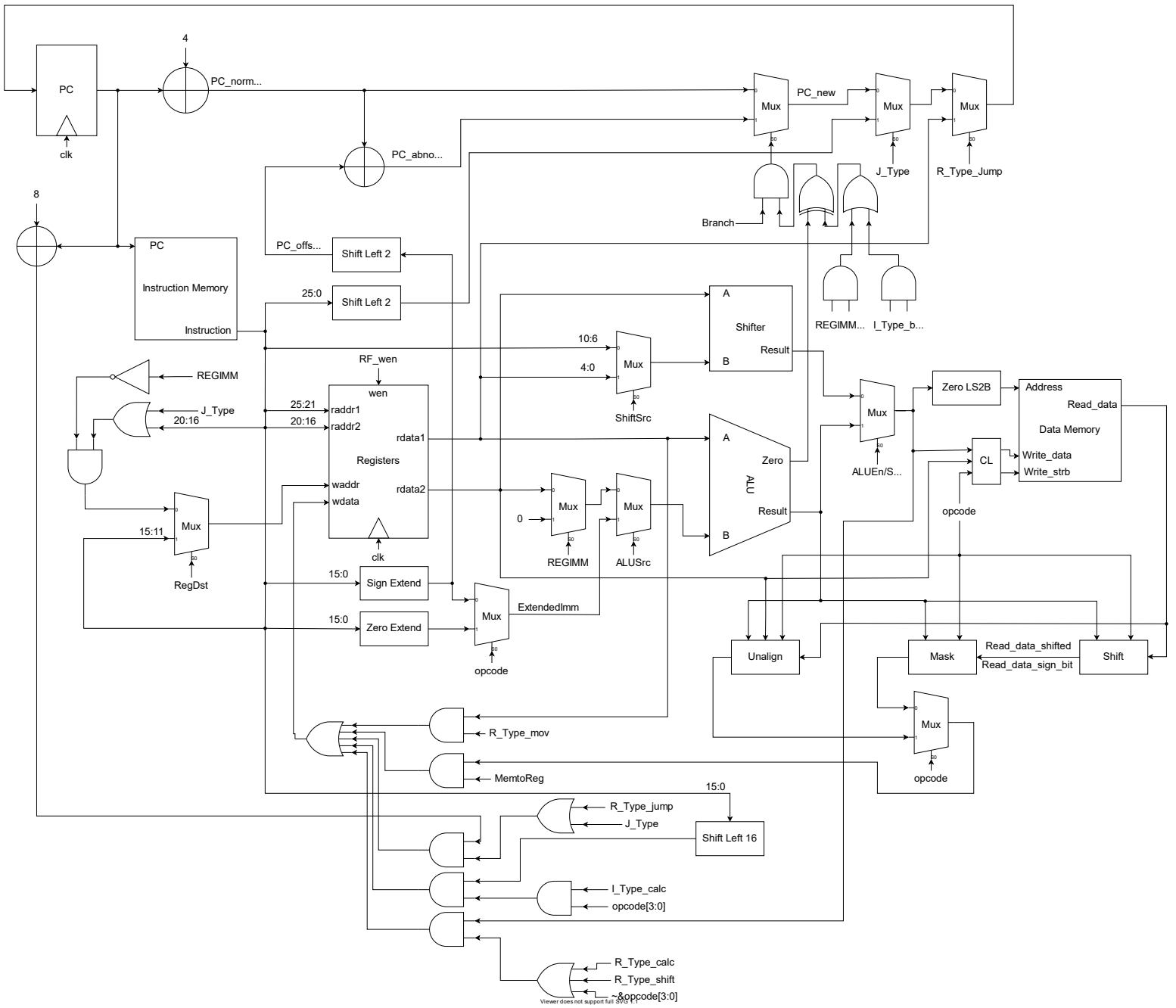
注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 GitLab 远程仓库 master 分支（具体命令详见实验报告）。

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)

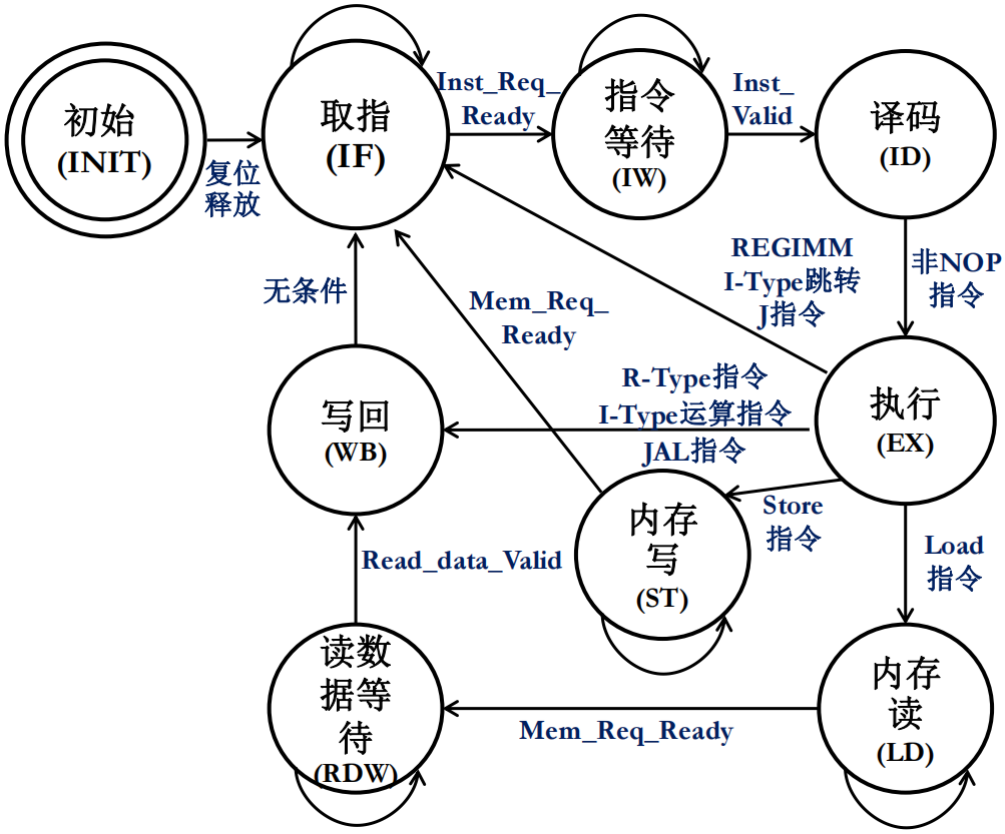
1．定制处理器电路结构图

多周期电路除了增加多周期状态机与寄存器外，其结构与 prj2 的单周期电路相似。电路结构图如下：



2. 多周期状态机

状态转移图如课程课件所示：



共有九个状态，采用独热码对这九个状态逐个进行编码：

```
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;
```

采用“三段式”状态机描述方法：

a. “第一段”用 always 时序逻辑，描述状态寄存器的同步状态跳转。

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

b. “第二段”用 always 组合逻辑，根据状态机当前状态和输入信号，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 (|IR) 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
```

c. “第三段”用 always 时序逻辑或 assign 组合逻辑，根据状态机当前状态，描述不同输出寄存器的同步变化。

3. 基于真实内存的多周期访存逻辑

```
    assign Inst_Req_Valid   = current_state[isIF];
    assign Inst_Ready       = current_state[isIW] ||
current_state[isINIT];
    assign MemWrite         = current_state[isST];
    assign MemRead          = current_state[isLD];
    assign Read_data_Ready  = current_state[isRDW] ||
current_state[isINIT];
```

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。

4. UART 控制器简单驱动程序

```
int
puts(const char *s)
{
    //TODO: Add your driver code here
    int i;
    for(i=0;s[i]!='\0';i++){
        while((*(volatile unsigned int *)
              ((void *)uart+UART_STATUS))&UART_TX_FIFO_FULL);
        *((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。

5. 性能计数器

```
    //cnt
    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;

    reg  [31:0] inst_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            inst_cnt <= 32'd0;
        end else if (current_state[isID]) begin
            inst_cnt <= inst_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_1 = inst_cnt;

    reg  [31:0] mr_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            mr_cnt <= 32'd0;
        end else if (current_state[isEX] && I_Type_mr) begin
            mr_cnt <= mr_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_2 = mr_cnt;

    reg  [31:0] mw_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            mw_cnt <= 32'd0;
        end else if (current_state[isEX] && I_Type_mw) begin
            mw_cnt <= mw_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_3 = mw_cnt;

    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[isIF] && next_state[isIF]) begin
            inst_req_delay_cnt <= inst_req_delay_cnt + 32'd1;
```

```
        end
    end
    assign cpu_perf_cnt_4 = inst_req_delay_cnt;

    reg  [31:0] inst_delay_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            inst_delay_cnt <= 32'd0;
        end else if (current_state[isIW] && next_state[isIW]) begin
            inst_delay_cnt <= inst_delay_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_5 = inst_delay_cnt;

    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[isLD] && next_state[isLD]) begin
            mr_req_delay_cnt <= mr_req_delay_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_6 = mr_req_delay_cnt;

    reg  [31:0] rd_delay_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            rd_delay_cnt <= 32'd0;
        end else if (current_state[isRDW] && next_state[isRDW])
begin
            rd_delay_cnt <= rd_delay_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_7 = rd_delay_cnt;

    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[isST] && next_state[isST]) begin
            mw_req_delay_cnt <= mw_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_8 = mw_req_delay_cnt;

    reg  [31:0] branch_inst_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            branch_inst_cnt <= 32'd0;
```

```
        end else if (current_state[isID] && (I_Type_b || REGIMM))
begin
            branch_inst_cnt <= branch_inst_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_9 = branch_inst_cnt;

    reg  [31:0] jump_inst_cnt;
    always @ (posedge clk) begin
        if (rst) begin
            jump_inst_cnt <= 32'd0;
        end else if (current_state[isID] && (R_Type_jump || J_Type))
begin
            jump_inst_cnt <= jump_inst_cnt + 32'd1;
        end
    end
    assign cpu_perf_cnt_10 = jump_inst_cnt;
```

共定义了 11 个性能计数器，其功能分别如下表所示：

名称	端口	描述
cycle_cnt	cpu_perf_cnt_0	Cycle Count
inst_cnt	cpu_perf_cnt_1	Instruction Count
mr_cnt	cpu_perf_cnt_2	Memory Read
mw_cnt	cpu_perf_cnt_3	Memory Write
inst_req_delay_cnt	cpu_perf_cnt_4	Instruction Request Delay
inst_delay_cnt	cpu_perf_cnt_5	Instruction Response Delay
mr_req_delay_cnt	cpu_perf_cnt_6	MemRead Request Delay
rd_delay_cnt	cpu_perf_cnt_7	Read Data Delay
mw_req_delay_cnt	cpu_perf_cnt_8	MemWrite Request Delay
branch_inst_cnt	cpu_perf_cnt_9	Branch Count
jump_inst_cnt	cpu_perf_cnt_10	Jump Count

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）

1. UART 控制器寄存器偏移地址计算错误

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

在驱动程序中，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。我最初企图通过以下代码来访问 STAT_REG 寄存器：

(volatile unsigned int *)(uart+UART_STATUS)

但事实上，这是存在问题的。经过反复调试后，我发现问题出在 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 0x08。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；但我的目标是让它每加一偏移 1 个字节，因此需要在加之前将 uart 强制类型转换为 void * 类型：

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

2. 独热码的使用

在最初的实现中，我虽然将状态按如下独热码编码：

```
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;
```

但并没有充分利用独热码的性质。例如，我使用了如下代码判断当前状态是否为 IF 状态：

```
if (current_state == IF)
```

在我看来，这样编码虽然带来了更多的硬件损耗，但比起以下编码更具可读性：

```
if (current_state[1])
```

在最终版本中，我定义了下标变量：

```
localparam isINIT   =0;
localparam isIF     =1;
localparam isIW     =2;
localparam isID     =3;
localparam isEX     =4;
localparam isST     =5;
localparam isLD     =6;
localparam isRDW    =7;
localparam isWB     =8;
```

并使用以下代码判断是否为 IF 状态：

```
If (current_state[isIF)
```

这样便兼具了代码高可读性与硬件低损耗。

三、 对讲义中思考题（如有）的理解和回答

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，在前文 for 循环的 while 循环中：

```
for(i=0;s[i]!='\0';i++){
    while((*(volatile unsigned int *)
          ((void *)uart+UART_STATUS))&UART_TX_FIFO_FULL);
    *((char *)uart+UART_TX_FIFO) = s[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。在每轮循环中，由于编译器的优化，while 判断条件将不会发生改变。因此，理论上来讲，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。

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。

实验结果证实了这个猜想。以下是正常的行为仿真与 fpga 运行结果：

```
53  testing 1 2 0000003
54  faster and "cheaper"
55  deadf00d % DEADF00D
56  00000000100000000200000003000000004000000005
57  50 50 -50 4294967246
58  ================================================
59  Benchmark simulation passed!!!
60  ================================================
```

```
50  testing 1 2 0000003
51  faster and "cheaper"
52  deadf00d % DEADF00D
53  00000000100000000200000003000000004000000005
54  50 50 -50 4294967246
```

以下是删去 volatile 后的行为仿真结果：

```
54  testing 1 2 0000003
55  faster and "cheaper"
56  deadf00d % DEADF00D
57  00000000100000000200000003000000004000000005
58  50 50 -50 4294967246==================================================
59  Benchmark simulation passed!!!
60  ================================================
```

可以看到，去掉 volatile 关键字后，在行为仿真这一步仍能正确打印出字符。不过，此时结尾处的换行符未正常显示。

在 fpga 上运行后，发现错误更为明显。大部分字符串均未正常显示：

```
50  testing 1 2 0000fatdeadf00d % DEADF0000000010000050 50 -50 429496time 10891.73ms
```

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 等待逻辑，不顾当前队列是否为满而向 TX FIFO 中写入。导致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。

四、 在课后，你花费了大约 10 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）

由于本次实验与上个多周期 CPU 的实验较为相似，加上资料完善，实现起来并不复杂。

遇到的主要问题在于云端平台测试较慢，往往改动一下代码需要等待较长时间才能获取实验结果。