中国科学院大学计算机组成原理（研讨课）

实 验 报 告

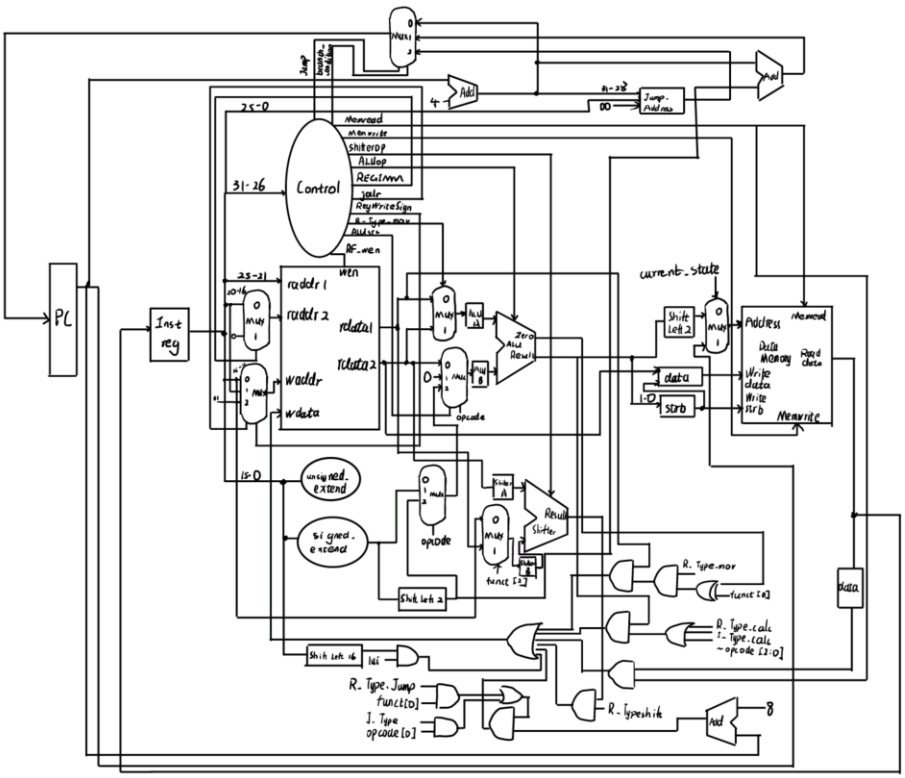
学号：2021K8009929016　　姓名：李金明　专业：计算机科学与技术

实验序号：3　　实验名称：定制 MIPS 功能型处理器设计

一、 逻辑电路结构及说明

1. 逻辑电路结构图

相比单周期处理器电路图主要的改变有：



增加了若干寄存器，用于存放 ALU 操作数，Shifter 操作数和指令等。

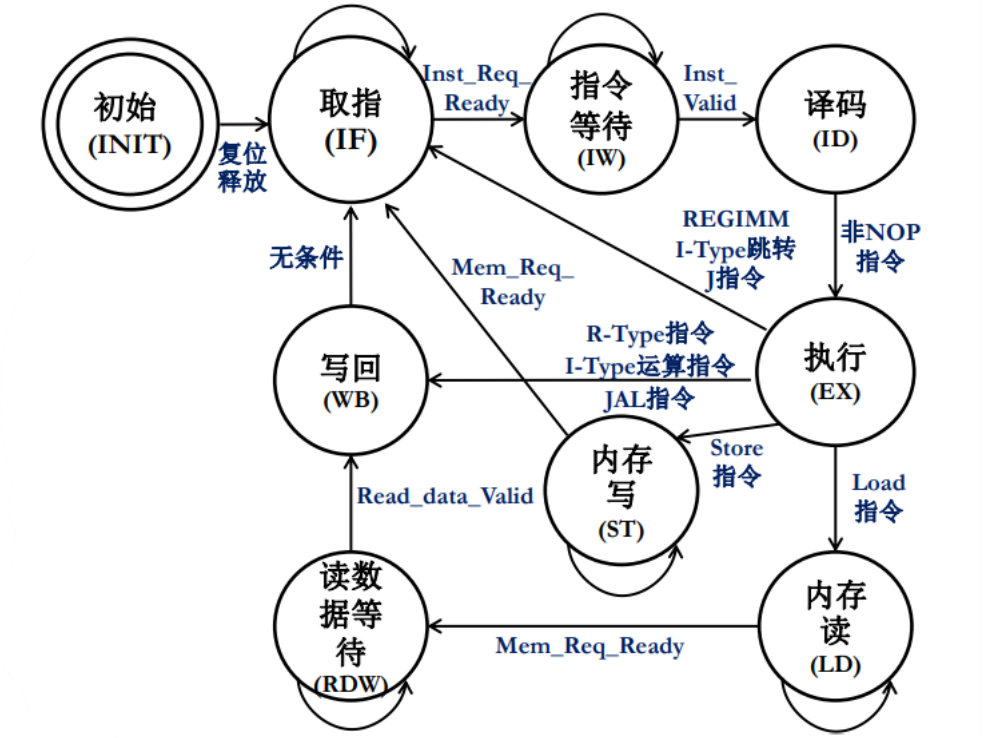
不需要单独设置存放指令的 memory。

其他改变如 ALU 的复用由于绘制较为麻烦没有体现在电路图上。

2. 代码实现部分

a. 状态转移部分

该部分电路设计使用三段式状态机，代码依照以下的状态转移图进行设计：



状态机前两段设计由以下代码给出：

```
`define S_IF 9'b000000001
`define S_ID 9'b000000010
`define S_EX 9'b000000100
`define S_ST 9'b000001000
`define S_WB 9'b000010000
`define S_LD 9'b000100000
`define S_RDW 9'b001000000
`define S_IW 9'b010000000
`define S_INIT 9'b100000000
    assign state_IF = current_state == `S_IF;
    assign state_ID = current_state == `S_ID;
    assign state_EX = current_state == `S_EX;
    assign state_ST = current_state == `S_ST;
    assign state_WB = current_state == `S_WB;
    assign state_LD = current_state == `S_LD;
    assign state_RDW = current_state == `S_RDW;
    assign state_IW = current_state == `S_IW;
    assign state_INIT = current_state == `S_INIT;
```

```
    always @ (posedge clk) begin
            if (rst) current_state <= `S_INIT;
            else current_state <= next_state;
    end
    always @ (*) begin
        case (current_state)
            `S_INIT: next_state = `S_IF;
            `S_IF: begin
                if (Inst_Req_Ready) next_state = `S_IW;
                else next_state = `S_IF;
            end
            `S_IW: begin
                if (Inst_Valid) next_state = `S_ID;
                else next_state = `S_IW;
            end
            `S_ID: begin
                if (Instruction_reg == 32'b0) next_state = `S_IF;
                else next_state = `S_EX;
            end
            `S_EX: begin
                if (I_Type_memw) next_state = `S_ST;
                else if (I_Type_memr) next_state = `S_LD;
                else if (R_Type || I_Type_calc || (J_Type &&
opcode[0])) next_state = `S_WB;
                else next_state = `S_IF;
            end
            `S_ST: begin
                if (Mem_Req_Ready) next_state = `S_IF;
                else next_state = `S_ST;
            end
            `S_WB: next_state = `S_IF;
            `S_LD: begin
                if (Mem_Req_Ready) next_state = `S_RDW;
                else next_state = `S_LD;
            end
            `S_RDW: begin
                if (Read_data_Valid) next_state = `S_WB;
                else next_state = `S_RDW;
            end
            default: next_state = `S_INIT;
        endcase
    end
```

总共有 9 个状态，这 9 个状态使用独热码的方式进行编码。

第一段使用 always 时序逻辑描述状态机的跳转。

第二段使用 always 组合逻辑，根据当前状态和输入信号决定下一状态。

b. 基于真实内存的多周期访问逻辑

```
assign MemRead = state_LD;
assign Read_data_Ready = state_RDW | state_INIT;
assign MemWrite = state_ST;
assign Inst_Req_Valid = state_IF;
assign Inst_Ready = state_IW | state_INIT;
```

在 IF 阶段需要拉高 Inst_Req_Valid 信号，并在 Inst_Req_Ready 信号拉高时进入 IW 状态。

在 IW 状态需要拉高 Inst_Ready 信号，并在 Inst_Valid 信号拉高时获取指令并转入 ID 状态。

在 ST 和 LD 状态分别需要拉高 MemWrite 信号和 MemRead 信号，并在 Mem_Req_Ready 信号拉高时向内存写入数据或从内存读取数据，并转移至下一个状态。其中 ST 状态转入 ID 状态，LD 状态转入 RDW 状态准备写回数据。

在 RDW 状态需要拉高 Read_data_Ready 信号，并在 Read_data_Valid 信号拉高后转移至 WB 状态。

c. 指令的获取

```
always @ (posedge clk) begin
    if (state_IW && Inst_Valid) Instruction_reg <= Instruction;
end
```

在指令等待阶段且 Inst_Valid 信号拉高时，将指令存入寄存器，这是这一

阶段的指令。

d. ALU 及 shifter 相关代码

```
    assign ALUop =
    (state_IF | state_ID)? `ALUOP_ADD :
    ((state_EX)? ((R_Type_calc)? ((funct[3:2] == 2'b00)? {funct[1],
2'b10} :
    ((funct[3:2] == 2'b01)? {funct[1], 1'b0, funct[0]} :
    {~funct[0], 2'b11})) :
    ((I_Type_calc && opcode[2:0] != 3'b111)? ((opcode[2:1] == 2'b00)?
{opcode[1], 2'b10} :
    ((opcode[2])? {opcode[1], 1'b0, opcode[0]} :
    {~opcode[0], 2'b11})) :
    ((R_Type_mov || (I_Type_branch && opcode[1] == 1'b0))? `ALU_SUB :
    ((REGIMM || (I_Type_branch && opcode[1] == 1'b1))? `ALU_SLT :
    `ALUOP_ADD)))) : `ALUOP_ADD);

    always @(posedge clk) begin
        if (state_ID) begin
            ALUOut <= ALU_Result;
            ALU_A_reg <= (({32{R_Type_mov | (I_Type_branch &
~opcode[1])}} & RF_rdata2) |
                        ({32{R_Type_jump | J_Type}} & PC) |
                        ({32{I_Type_branch & opcode[1]}} & {32{1'b0}}) |
                        ({32{R_Type_calc | REGIMM | I_Type_calc |
I_Type_memw | I_Type_memr}} & RF_rdata1));
            ALU_B_reg <= (({32{R_Type_calc}} & RF_rdata2) |
                          ({32{I_Type_branch & opcode[1] |
(I_Type_branch & ~opcode[1])}} & RF_rdata1) |
                ({32{REGIMM | R_Type_mov}}) & {32{1'b0}} |
                ({32{ALUsrc}} & extend) |
                ({32{R_Type_jump | J_Type}} & {{29{1'b0}}, 3'b100}));
            Shifter_A_reg <= RF_rdata2;
            Shifter_B_reg <= {5{funct[2]}} & RF_rdata1 |
                                {5{~funct[2]}} & shamt;
        end
        if (state_EX) begin
            if (R_Type_shift) ALUOut <= Shifter_Result;
            else ALUOut <= ALU_Result;
        end
    end
```

```
    assign ALU_A = ({32{state_IF | state_ID}} & PC) |
                   ({32{state_EX}} & ALU_A_reg);
    assign ALU_B = ({32{state_IF}} & {{29{1'b0}}, 3'b100}) |
                   ({32{state_ID}} & {sign_left_extend}) |
                   ({32{state_EX}} & ALU_B_reg);
    assign Shifter_A = {32{state_EX}} & Shifter_A_reg;
    assign Shifter_B = {32{state_EX}} & Shifter_B_reg;
    assign Shiftop = ({2{R_Type_shift && state_EX}}) & funct[1:0];
```

这里的代码描述了 ALU 及 Shifter 两个操作数的赋值和 Shifterop 及 ALUop 的处理，与多周期处理器实验中的代码相同，实现了 ALU 的复用，没有改动。

e. PC 的赋值

```
    always @(posedge clk) begin
        if (rst) PC <= 32'b0;
        else if (state_IF) PC_4 <= ALU_Result;
        else if (state_ID && Instruction_reg == 32'b0) PC <= PC_4;
        else if (state_EX) begin
            if (R_Type_jump) PC <= RF_rdata1;
            else if (J_Type) PC <= {PC_4[31:28], instr_index[25:0],
2'b00};
            else if (I_Type_branch && (~opcode[1] && (opcode[0] ^
Zero) ||
                                       (opcode[1] && (opcode[0] ^
(ALU_Result == 32'b0)))) ||(REGIMM && (rt[0] ^ (ALU_Result ==
32'b1))))
                    PC <= ALUOut;
            else PC <= PC_4;
        end
    end
```

PC 的赋值仅进行了小部分改动：将 PC 更新为 PC+4 的状态有 IF 更改为 ID 状态（NOP 指令）或 EX 阶段（非 NOP 指令）。

f. 性能计数器的设计

```
    reg [31:0] cycle_cnt;
    reg [31:0] memory_cnt;
    reg [31:0] instruction_cnt;
    reg [31:0] wait_cnt;
    reg [31:0] load_cnt;
    reg [31:0] instruction_fetch_cnt;
    reg [31:0] instruction_wait_cnt;
    reg [31:0] mem_wait_cnt;
    reg [31:0] rdw_cnt;

    always @(posedge clk) begin
        if (rst) cycle_cnt <= 32'b0;
        else cycle_cnt <= cycle_cnt + 32'b1;
    end
    assign cpu_perf_cnt_0 = cycle_cnt;

    always @(posedge clk) begin
        if (rst) memory_cnt <= 32'b0;
        else if (state_LD || state_ST) memory_cnt = memory_cnt
+32'b1;
    end
    assign cpu_perf_cnt_1 = memory_cnt;

    always @(posedge clk) begin
        if (rst) instruction_cnt <= 32'b0;
        else if (state_IF) instruction_cnt = instruction_cnt +32'b1;
    end
    assign cpu_perf_cnt_2 = instruction_cnt;

    always @(posedge clk) begin
        if (rst) wait_cnt <= 32'b0;
        else if (state_IF && ~Inst_Req_Ready || state_IW &&
~Inst_Valid || (state_LD || state_ST) && ~Mem_Req_Ready || state_RDW
&& ~Read_data_Valid) wait_cnt = wait_cnt +32'b1;
    end
    assign cpu_perf_cnt_3 = wait_cnt;
```

```
always @(posedge clk) begin
        if (rst) load_cnt <= 32'b0;
        else if (state_EX && I_Type_memr) load_cnt <= load_cnt +
32'b1;
    end
    assign cpu_perf_cnt_4 = load_cnt;

    always @(posedge clk) begin
        if (rst) instruction_fetch_cnt <= 32'b0;
        else if (state_IF && ~Inst_Req_Ready) instruction_fetch_cnt =
instruction_fetch_cnt +32'b1;
end
    assign cpu_perf_cnt_5 = instruction_fetch_cnt;

    always @(posedge clk) begin
        if (rst) instruction_wait_cnt <= 32'b0;
        else if (state_IW && ~Inst_Valid) instruction_wait_cnt =
instruction_wait_cnt +32'b1;
    end
    assign cpu_perf_cnt_6 = instruction_wait_cnt;

always @(posedge clk) begin
        if (rst) mem_wait_cnt <= 32'b0;
        else if ((state_LD || state_ST) && ~Mem_Req_Ready)
mem_wait_cnt = mem_wait_cnt +32'b1;
    end
    assign cpu_perf_cnt_7 = mem_wait_cnt;

    always @(posedge clk) begin
        if (rst) rdw_cnt <= 32'b0;
       else if (state_RDW && ~Read_data_Valid) rdw_cnt = rdw_cnt
    +32'b1;
    end
    assign cpu_perf_cnt_8 = rdw_cnt;
```

共设计了 9 个性能计数器，计数内容如下：

cpu_perf_cnt_0：周期计数器；

cpu_perf_cnt_1：访问 memory 次数计数器；

cpu_perf_cnt_2：指令计数器；

cpu_perf_cnt_3：等待时长计数器；

cpu_perf_cnt_4：load 类指令计数器；

cpu_perf_cnt_5：IF 状态等待周期数；

cpu_perf_cnt_6：IW 状态等待周期数；

cpu_perf_cnt_7：LD 及 ST 状态等待周期数；

cpu_perf_cnt_8：RDW 状态等待周期数。

g. Puts 函数设计

```
int
puts(const char *s)
{
    volatile char *uart_tem = (void *)uart;
    int i;

    for (i = 0; s[i] != '\0'; i++)
    {
        while ((*(uart_tem + UART_STATUS) & UART_TX_FIFO_FULL));
        *(uart_tem + 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相应字符，最后返回字符串长度

h. Bench_prepare 函数及 bench_done 函数设计

```
volatile unsigned int* const cycle_cnt = (void *)0x60010000;
volatile unsigned int* const memory_cnt = (void *)0x60010008;
volatile unsigned int* const instruction_cnt = (void *)0x60011000;
volatile unsigned int* const wait_cnt = (void *)0x60011008;
volatile unsigned int* const load_cnt = (void *)0x60012000;
volatile unsigned int* const if_cnt = (void *)0x60012008;
volatile unsigned int* const iw_cnt = (void *)0x60013000;
volatile unsigned int* const memw_cnt = (void *)0x60013008;
volatile unsigned int* const rdw_cnt = (void *)0x60014000;

void bench_prepare(Result *res) {
  res->msec = _uptime();
  res->mem_c = *memory_cnt;
  res->inst_c = *instruction_cnt;
  res->wait_c = *wait_cnt;
  res->ld_c = *load_cnt;
  res->if_c = *if_cnt;
  res->iw_c = *iw_cnt;
  res->memw_c = *memw_cnt;
  res->rdw_c = *rdw_cnt;
}

void bench_done(Result *res) {
  res->msec = _uptime() - res->msec;
  res->mem_c = *memory_cnt - res->mem_c;
  res->inst_c = *instruction_cnt - res->inst_c;
  res->wait_c = *wait_cnt - res->wait_c;
  res->ld_c = *load_cnt - res->ld_c;
  res->if_c = *if_cnt - res->if_c;
  res->iw_c = *iw_cnt - res->iw_c;
  res->memw_c = *memw_cnt - res->memw_c;
  res->rdw_c = *rdw_cnt - res->rdw_c;
}
```

算法开始运行前，调用 bench_prepare 函数，获取计数器初值，并保存在 Result 结构体中。

算法运行结束后，调用 bench_done 函数，获取计数器当前值，与计数器初值相减，即可得到执行周期数。

i. 性能计数器打印结果

```
Cycles: 52557723
Memory Load/Store: 54020
Instructions: 728305
Cycles Waiting totally: 49089087
Load Instructions: 11595
Cycles Waiting for Fetching Instructions: 0
Cycles Waiting for Instructions: 48248267
Cycles Waiting for Requesting Data: 34725
Cycles Waiting for Reading Data: 806890
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

由于在阶段二已经进行写过多周期处理器，故此阶段实验并没有花费很大的力气。主要更改有 PC 值更新的阶段、状态转移部分的变化、控制信号的变化。

主要难点我认为有对实验的相关理解。以前使用理想内存，数据相当于想要就有，这次使用现实内存加入了等待，加入了新的信号，需要学会如何使用这些信号。

通过对状态转移图的观察和对真实内存的理解，我逐渐理解了各个信号的含义及使用方式。

三、 对讲义中思考题（如有）的理解和回答

1．Volatile 关键字的作用是什么？如果去掉会有什么后果?

Volatile 关键字提醒编译器它后面定义的变量随时有可能发生变化，因此编译后的程序每次需要存储或读取这个变量的时候，会告诉编译器不对该变量做优化，而从变量内存地址中读取数据。

若将上文中 puts 函数中的 volatile 删去，那么由于编译器发现 uart_tem 值不会改变，会将第一次进入 while 循环条件判断的 uart_tem 值作为接下来每次 for 循环进入 while 循环条件判断的

uart_tem 值。这会导致若初次判断时队列状态寄存器未满，程序接下来即使队列已满，也会写入值。

2. 处理器内部是否需要实现异步串行通信协议

不一定。

同步通信协议使用时钟信号来进行操作，所有操作在时钟边沿进行。同步通信协议通信效率较高，但要求接收端和发送端时钟频率一致，二者间允许的误差较小，对于大规模处理器而言处理起来较为复杂。

而异步通信使用独立的传输手段，允许模块间高效率的通信从而提高处理器的性能，减少时钟相关问题，但异步通信协议通信效率可能降低。

总之，二者各有利弊，需要结合具体情况做出选择。

四、 在课后，你花费了大约 3 小时完成此次实验?

五、 对于此次实验的心得、感受和建议

由于前一阶段写了多周期处理器，我认为本次实验较为简单，改动较少。

本次实验行为仿真中矩阵乘及水仙花测试时间较长，且由于需要观察 fpga 中的结果，对实验产生了一定的不便。

开始是 fpga 阶段存在 bug，有概率不通过，我按照课上所讲 fpga 加速

进行了尝试，没有找到问题所在，最后重新跑了一次源代码的错误阶段，在未加修改的情况下通过了，带来了一些时间上的浪费。

非常感谢所有老师的认真讲解及解答问题。