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 3 实验名称 定制 MIPS 功能型处理器设计

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。

一、 Verilog 代码的编写

本次实验的难度不高, 因为这次实验基本上是在上一次单周期 cpu 设计的基础上经过多周期改良之后设计的, 因此整体的代码量并不高, 但是这次实验中仍然有一些不太平凡的工作, 例如一些控制信号的多周期适应性处理, 状态机编写, 时序的设计等等.

由于本次实验是在上一次单周期 cpu 设计的基础上发展而来, 于是本次报告并不太涉及原本指令实现部分的代码的讲解, 主要聚焦于这次实验的独特性工作, 例如状态机的设计以及一些寄存器变量的改进与设计

1.1 状态机 FSM 设计

本次实验的状态机严格按照讲义中所描述的状态机进行三段式设计, 状态机转移图如下

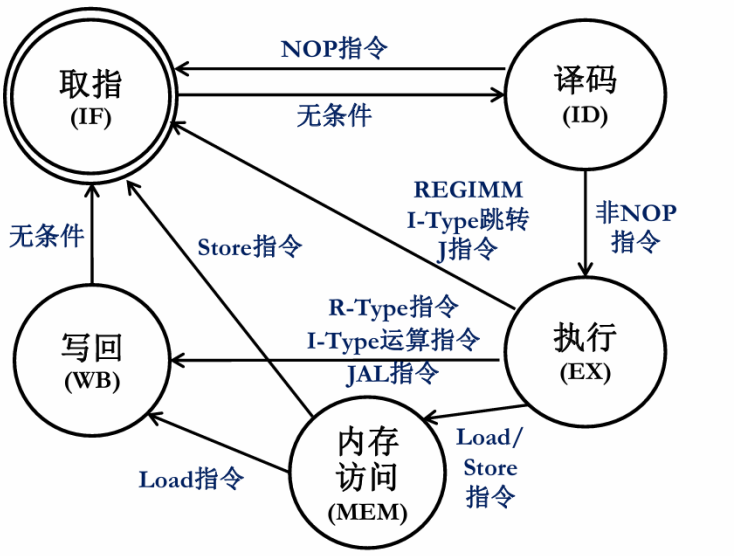

图 1: 状态机转移图

另外值得一提的是, 对于 JR 指令, 在讲义中要求在 EX 状态后进入 WB 状态, 但是在 MIPS 指令集中的 JR 指令貌似并不需要经过写回这个阶段, 在程序 debug 的时候因为对于 JR 状态的处理导致了与参考的 cpu 信号领先了一个周期导致过不了测试, 但是个人认为这里的 JR 指令无论经过 WB 与否是不影响这个程序运行的

1.1.1 状态机的状态定义

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

在这里利用 localparam 来定义参数而不是用 define 来定义宏, 对于目前的实验而言二者差别不大, 但是 localparam 仅仅只用于当前模块, 而 define 则是公用的全局定义, 而这些状态直在当前的模块中使用, 因此这里利用 localparam 来定义状态机的状态, 这样可以避免定义混淆而出现意料之外的错误

1.1.2 状态机第一段

利用同步跳转, 描述了状态的转移, 值得注意的是在本次实验中, rst 信号对于各个寄存器的复位是同步的

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

在每个时钟信号上升沿到来的时候, 将 current_state 的值赋值为 next_state, 这样就实现了状态机的转移.

1.1.3 状态机第二段

状态机的第二段利用 always@(*) 的连续赋值实现了对于 next_state 的赋值, 描述了状态转移

```
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(Instruction_current[31:0] == 32'b0) begin
                                next_state      =       IF;
                        end
                        else begin
                                next_state      =       EX;
                        end
                end
                EX: begin
                        if(opcode==`REGIMM || opcode[5:2]==4'b0001 ||
    opcode==`J) begin
    //表示直接返回取指阶段的三种指令类型，REGIMM, I跳转和J
                                next_state      =       IF;
                        end
                        else if (opcode[5:3]==3'b101) begin
                                next_state      =       ST;
                        end
                        else if (opcode[5:3]==3'b100) begin
                                next_state      =       LD;
                        end
                        else begin
                                next_state      =       WB;
                        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: begin
                        next_state      =       IF;
                end
                default : begin
                        next_state      =       INIT;
                end
        endcase
end
```

这里有几个要点, 首先是 EX 阶段的状态转移明显比其他的状态转移要复杂, 因为不同的指令有不同的状态转移, 于是我们可以充分利用 if-else 语句, 将描述比较复杂的直接跳转回 WB 状态的指令写入最后的 else 块中, 来简化程序的复杂度, NOP 在译码之后直接进入取指其次是注意不要忘记了 default 以及复位信号, 还有这种连续赋值要用阻塞赋值, 以避免 fpga 的模拟不通过!!

1.1.4 状态机第三段

根据当前状态来对这些控制信号进行赋值, 这里利用组合逻辑来避免时序出现错误, 要注意 INIT 状态的时候拉高 Inst_Ready 和 Read_data_Ready, 来避免复位释放后造成的影响, 这里的理解是在非取指或者访存阶段, CPU 仍然会从内存中读出一些数据, 如果不拉高这两个信号来将这些不需要数据读出, 会导致最终这里读出来的数据并不是我们想要的, 其他的信号赋值严格按照讲义上操作, 在此不多赘述

```
assign Inst_Req_Valid   = (current_state == IF);
assign Inst_Ready       = (current_state == INIT || current_state == IW);
assign MemRead          = (current_state == LD);
assign MemWrite         = (current_state == ST);
assign Read_data_Ready  = (current_state == RDW || current_state == INIT);
```

1.2 指令寄存器的设计

我们只有在 IW 阶段接收到有效的 Inst_Valid 信号时, 才会将当前的指令 Instruction_current 赋值为内存读出的数据 Instruction, 其他时候保持不变, 来避免之后的 ID 以及 EX 等部分的组合逻辑信号出现错误, 另外要记得在复位信号有效的时候复位为 NOP.

```
reg  [31:0] Instruction_current;
always @(posedge clk) begin
```

```
        if(rst) begin
                Instruction_current   <=        32'b0;
        end
        else if(Inst_Ready && Inst_Valid) begin
                Instruction_current     <=      Instruction;
        end
        else begin
                Instruction_current     <=      Instruction_current;
        end
end
```

1.3 PC 的修改

在本次实验中, PC 也要进行一些适应性的修改, 由于担心模块给定的短沟 PC 不能改成寄存器类型, 于是自己定义了一个 PC_reg, 我采用的是在 EX 阶段进行跳转, 这里还有一个情况狂就是 NOP 在 ID 阶段直接改写 PC, 之后重新进入译码阶段

```
reg [31:0] PC_reg;
always @(posedge clk) begin
        if (rst) begin
                PC_reg  <=      32'b0;
        end
        else if(Instruction_current==32'b0 && current_state==ID) begin
  //表示NOP指令，在译码后直接回到取指阶段
                PC_reg  <=      PC_reg + 4;
        end
        else if(current_state==EX) begin
  //对于其他类型的指令，根据指令类型来判断跳转
                PC_reg  <=      (Jump?      Jump_addr:
                                 Branch_f? PC_reg + Branch_addr + 4:
                                           PC_reg + 4);
        end
        else begin
                PC_reg  <=      PC_reg;
    //对于其他状态，PC不改变
        end
end

assign PC = PC_reg;
```

另外, 由于 JAL 和 JALR 指令需要写回 PC+8 的值, 而这个 PC 所指的更新前的 PC, 所以需要在更新 PC 的时候, 用一个寄存器来保存修改前的 PC

```
reg [31:0] old_PC;

always @(posedge clk) begin
        if(rst) begin
```

```
                old_PC    <=     32'b0;
        end
        else if (current_state==IF) begin
                old_PC  <=  PC_reg;
        end
        else begin
                old_PC  <=  old_PC;
        end
end
```

1.4 译码阶段和执行阶段的时序修改

在译码阶段和执行阶段, 我将一些输出信号改成了寄存器变量, 以避免出现时序错误, 如果全部复制粘贴 simple_cpu 的组合逻辑代码, 有可能在一个周期内完成全部的 ID 和 EX 操作, 与我们的预期不符

这一部分的修改比较简单, 只需要改成时序逻辑和加上复位, 其他的和单周期 cpu 完全一样

1.4.1 ALU 操作数的适应性修改

```
always @(posedge clk) begin
        if(rst) begin
                ALUop_A    <=     32'b0;
        end
        else if(current_state==ID) begin
                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;
        end
        else begin
                ALUop_A <= ALUop_A;
        end
end

//运算器输入选择控制信号, rdata or imm 如果是Rtype或者beq和bne则输入rdata2
assign ALUSrc = (opcode==`R_TYPE || opcode[5:1]==5'b00010)? 0:1;

//运算器操作数B选择信号
always @(posedge clk) begin
        if(rst) begin
                ALUop_B    <=     32'b0;
        end
        else if(current_state==ID) begin
                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;
        end
        else begin
                ALUop_B <= ALUop_B;
        end
end
```

1.4.2 Shifter 操作数的适应性修改

```
//移位器操作数A选择信号
always @(posedge clk) begin
        if(rst) begin
                Shiftop_A   <=  32'b0;
        end
        else if(current_state==ID) begin
                Shiftop_A  <=   (opcode==`LUI)?          op_imm:
    //LUI指令对立即数做移位
                                                         rdata2;          //rt移动
        end
        else begin
                Shiftop_A <= Shiftop_A;
        end
end

//移位器操作数B选择信号
always @(posedge clk) begin
        if(rst) begin
                Shiftop_B   <=  5'b0;
        end
        else if(current_state==ID) begin
                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]
        end
        else begin
                Shiftop_B <= Shiftop_B;
```

```
        end
end
```

1.4.3 EX 阶段结果的时序修改

利用一个寄存器来存储 EX 阶段的输出结果 Result

```
always @(posedge clk) begin
        if(rst) begin
                Result    <=     32'b0;
        end
        else if(current_state==EX) begin
                Result <= (AluShi_sel)?   Shifter_result: ALU_result;
        end
        else begin
                Result <= Result;
        end
end
```

1.5 写回阶段的适应性修改

1.5.1 写使能信号的适应性修改

这是一个关键的修改, 因为写回寄存器只可能在 WB 阶段执行, 因此需要修改信号 RF_wen, 它是原本的使能信号和状态为 WB 的与, 这样才能正确在 WB 阶段才执行写回操作

```
assign RF_wen    =  RF_wen_i && (current_state==WB);
```

1.5.2 Load 指令的适应性修改

这一部分的修改和 Instruction 修改的思路一样, 主要的考虑是 Read_data 只在某个特殊的状态才是我们想要的

```
always @(posedge clk) begin
        if(rst) begin
                Read_data_current  <=   32'b0;
        end
        else if(Read_data_Ready && Read_data_Valid) begin
                Read_data_current <= Read_data;
        end
        else begin
                Read_data_current <= Read_data_current;
        end
end
```

1.6 指令周期计数器的编写

由于时间原因, 我再本次实验中只写了一个用于计数时钟周期的计数器, 没每个时钟上升沿到来时加一, 复位信号有效时置零

```
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;
```

二、 C 代码的编写

2.1 puts 函数的编写

puts 函数用于建立外部 IO 设备和 CPU

```
int puts(const char *s)
{
        int len = 0;
        while(s[len]){
                while(*((volatile char*)uart + UART_STATUS) & UART_TX_FIFO_FULL);
                *((volatile char*)uart + UART_TX_FIFO) = s[len];
                len++;
        }
        return len;
}
```

在这段代码中, 外部循环用于遍历这个字符串, 内部的循环用于从基址 uart 找到 UART_STATUS 的寄存器, 根据其中的内容判断当前的队列是否满, 这个内层的循环结束代表了队列现在不满, 然后在执行入队操作

2.2 perf_cnt 的编写

这部分的代码量非常小, 就只有几行, 按照讲义对着写即可

```
unsigned long _uptime() {
  volatile unsigned long* cycle_cnt = (volatile unsigned long*) 0x60010000;
  unsigned long cnt = *cycle_cnt;
  return cnt;
}

void bench_prepare(Result *res) {
  res->msec = _uptime();
}

void bench_done(Result *res) {
```

```
12    res->msec = _uptime() - res->msec;
13  }
```

三、 实验过程中遇到的问题、对问题的思考过程及解决方法

1. 本次实验虽然代码量总体而言比 simple_cpu 要小得多, 但是 debug 的过程却要更复杂一些, 因为有可能当前的错误是好几个周期之前所产生的, 导致产生一些意想不到的错误
2. 另外在本次实验中的时序也是很容易出现问题的, 因此要严格按照课上讲义的状态机来编写时序, 我在本次实验中添加了译码阶段和执行阶段的寄存器, 用于尽可能避免时序提前
3. 在本次实验中出现的另一大问题是 fpga 的编码, 我一开始对 next_state 赋值是用的非阻塞赋值, 这样是过不了 fpga 的测试的, 因为实际上 always@(*) 所描述的是组合逻辑, 所以不能用非阻塞赋值, 应该用阻塞赋值来实现连续赋值
4. 在本次实验中出现的 always 块中运用了很多 case 和 if-else 语句, 我们在编写的时候要注意 if-else 枚举全部情况, case 要写 default
5. 另外我在写 puts 函数的使用一开始第一条语句用的是逻辑与, 导致了 fpga 不过, 应该用按位与才能达成正确的结果 (地址有很多位)
6. 在本次实验中我还发现了一个现象, 就是我一开始的 fpga 测试集之中只有 hello 和 microbench 这两个没过, basic 三个都过了, 但是 hello 和 microbench 测试集的 emu 都能过, 所以这次实验中 emu 的评估也不是涉及真正的 IO 操作的, emu 只是生成指定区间的波形，和程序正确性没有关系

四、 思考题

4.1 volatile 关键字的作用是什么？如果去掉会出现什么后果？

volatile 这个单词的本义是指不稳定的, 多变的, 这个在 C 程序的关键字我认为其作用是控制编译器, 让编译器在编译的时候不要对这个语句在编译中进行优化, 让 C 程序编译成机器语言的时候保持其 C 程序本身的语句 (比如不将两个语句合成一个, 不将多条访存语句合并成一条), 这样可以帮助我们可以从波形中准确的知道内存的操作, 更好观察.

五、 实验所耗时间

在课后，你花费了大约____10____ 小时完成此次实验。