中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 宋俊仪 学号 2023K8009929018 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、 多周期处理器状态机设计

- **状态转移分析**

  讲义中的逻辑电路设计如下:(见图 1)

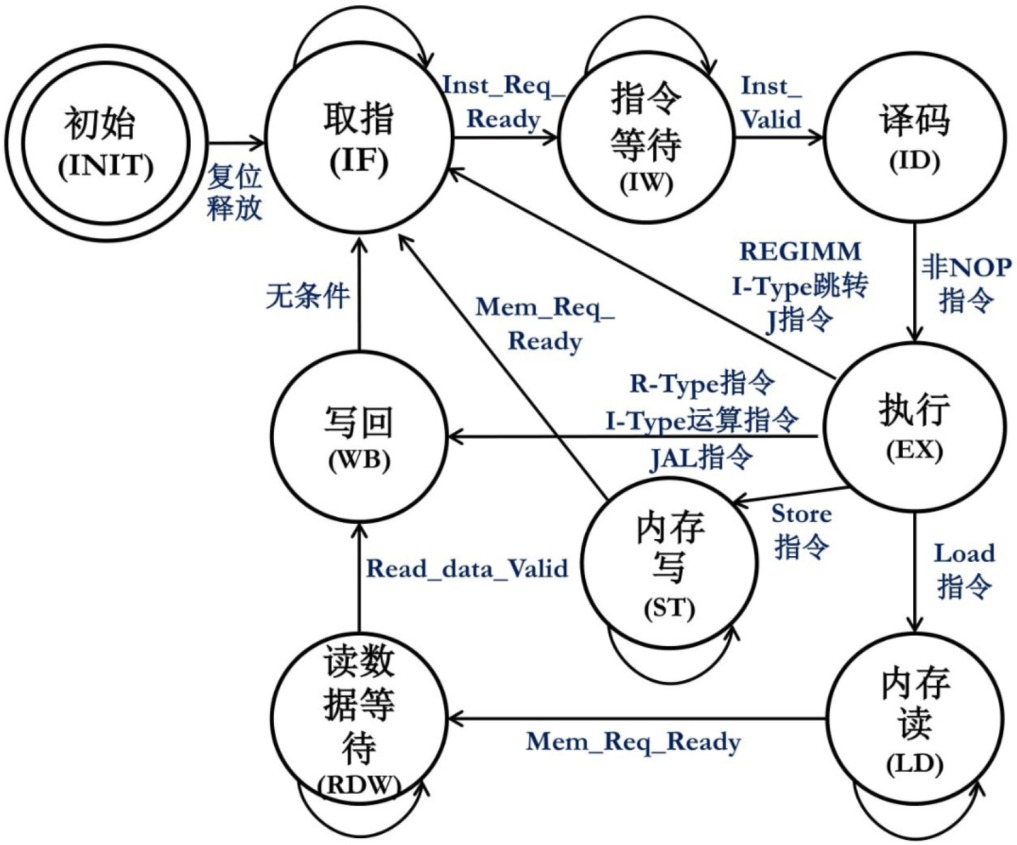

图 1: 定制 MIPS 处理器状态转移图

通过图 1，对多周期 CPU 各状态分析如下：

1 设计思路概述

此多周期 CPU 设计将指令执行划分为多个状态（周期）。与单周期 CPU 不同，不同指令可能需要不同数量的时钟周期来完成。这种设计的优点在于可以平衡各个操作所需的时间，避免时钟周期受限于最慢指令，同时硬件资源可以在不同周期内复用。

- **状态驱动执行：** CPU 的核心是一个状态机，根据当前状态和指令类型以及外部信号（如内存准备好）转换到下一个状态。
- **指令周期可变：**简单指令（如某些跳转）可能较快完成，而复杂指令（如访存）则需要更多周期。
- **PC 更新策略：**
  - 对于 REGIMM、I-Type 跳转和 J 指令，根据图示，在译码（ID）阶段计算目标地址并更新 PC 值，然后直接进入取指（IF）阶段获取下一条指令。文本补充说明这些指令在“执行阶段（更新 PC 值）后”进入 IF，这可能指 PC 更新的逻辑在 ID 阶段启动并在 EX 阶段前完成，或者该类指令的 EX 阶段非常短促并直接导向 IF。
  - 对于 JALR、JAL、JR 指令，根据文本，在执行（EX）阶段完成 PC 值更新；然后在写回（WB）阶段将返回地址写入 rd 寄存器。
- **访存操作细化：**访存操作被分解为独立的内存读（LD）、内存写（ST）和读数据等待（RDW）三个状态，以处理内存访问的延迟。
  - LD 状态：拉高 MemRead 信号，等待 Mem_Req_Ready 信号变高。
  - RDW 状态：等待 Read_data_Valid 信号变高。（文本中“RDW 状态要拉高 Read_data_Ready”可能指 CPU 发出准备好接收数据的信号，但图中未明确显示此信号作为状态转换条件）。
  - ST 状态：拉高 MemWrite 信号，等待 Mem_Req_Ready 信号变高。

2 各状态（周期）详解

INIT (初始状态)

- **目的：** CPU 复位后的初始状态。
- **操作：**等待“复位释放”信号。
- **下一状态：**当复位信号解除后，无条件转换到 IF 状态。

IF (取指阶段)

- **目的：**从内存中获取指令。
- **操作：**
  - 将程序计数器（PC）的值发送到内存地址总线。
  - 发出指令读取请求。
- **状态转换（根据图示）：**
  - 如果内存未准备好响应指令请求（‘Inst_Req_Ready‘为低，图中表现为 IF 的自循环，并有一条到 IW 的路径），则转换到 IW 状态（Instruction Wait）。

  - 如果指令已成功从内存取回(‘Inst_Valid‘为高),则转换到 ID 状态。
  - (图中 IF 自身有循环箭头,表示若 ‘Inst_Req_Ready‘为高但 ‘Inst_Valid‘未高,则可能在此等待。或者,若 ‘Inst_Req_Ready‘一直为低,则通过转换到 IW 状态等待。)

IW (指令等待)

- **目的:**当 IF 阶段发起取指请求但内存未准备好时,进入此状态等待。
- **操作:**等待。
- **状态转换:**当内存准备好接收取指请求(‘Inst_Req_Ready‘为高)时,转换回 IF 状态。

ID (译码阶段)

- **目的:**对获取的指令进行译码,并读取寄存器操作数。
- **操作:**
  - 解析指令的操作码、功能码、寄存器地址等。
  - 从寄存器堆读取源寄存器的值。
  - 对于 REGIMM、I-Type 跳转、J 指令:计算跳转目标地址,并准备更新 PC。
- **状态转换:**
  - 如果是 REGIMM、I-Type 跳转或 J 指令:更新 PC 后,转换到 IF 状态取下一条指令。
  - 如果是非 NOP 的其他指令(如 R-Type 运算、I-Type 运算、Load/Store、JAL):转换到 EX 状态。

EX (执行阶段)

- **目的:**执行指令的主要操作,如算术逻辑运算、地址计算等。
- **操作:**
  - R-Type 指令、I-Type 运算类指令:ALU 执行相应的算术或逻辑运算。
  - Load/Store 指令:ALU 计算访存的有效地址。
  - JAL 指令:ALU 进行相关计算(如目标地址),更新 PC,准备 link address (返回地址)。
  - JALR、JR 指令(根据文本):完成 PC 值更新,准备 link address。
- **状态转换:**
  - R-Type 指令、I-Type 运算类指令、JAL 指令:转换到 WB 状态。
  - Load 指令:转换到 LD 状态。
  - Store 指令:转换到 ST 状态。

LD (内存读)

- **目的:**启动对数据存储器的读操作。
- **操作:**
  - 将 EX 阶段计算出的有效地址发送到内存。

  - 拉高 ‘MemRead‘信号，向内存发出读请求。

- **状态转换（根据图示和文本）：**
  - 若内存未准备好接收请求（‘Mem_Req_Ready‘未高），则停留在 LD 状态等待（通过自循环）。
  - 当内存已接收读请求并准备处理（‘Mem_Req_Ready‘为高）时，转换到 RDW 状态。

RDW (读数据等待)

- **目的：**等待数据从内存中返回。
- **操作：**
  - 等待内存发出 ‘Read_data_Valid‘信号，表示数据已准备好。
- **状态转换（根据图示）：**
  - 若数据未有效（‘Read_data_Valid‘未高），则停留在 RDW 状态等待（通过自循环）。
  - 当数据已有效（‘Read_data_Valid‘为高）时，转换到 WB 状态。

ST (内存写)

- **目的：**将数据写入数据存储器。
- **操作：**
  - 将 EX 阶段计算出的有效地址和要写入的数据（来自寄存器）发送到内存。
  - 拉高 ‘MemWrite‘信号，向内存发出写请求。
- **状态转换（根据图示和文本）：**
  - 若内存未准备好接收请求或完成写入（‘Mem_Req_Ready‘未高），则停留在 ST 状态等待（通过自循环）。
  - 当内存已接收写请求或完成写入（‘Mem_Req_Ready‘为高）时，转换到 IF 状态，取下一条指令。

WB (写回阶段)

- **目的：**将指令执行的结果或从内存读取的数据写回寄存器堆。
- **操作：**
  - R-Type 指令、I-Type 运算类指令：将 ALU 的运算结果写入目标寄存器。
  - Load 指令：将从 RDW 状态获得的数据（来自内存）写入目标寄存器。
  - JAL 指令：将返回地址（PC+offset）写入目标寄存器（通常是 ra）。
  - JALR、JR 指令（根据文本）：将返回地址写入 rd 寄存器。
- **状态转换：**无条件转换到 IF 状态，开始下一条指令的取指。

- 以单周期 CPU（prj2）为基础，设计本次实验 CPU

1 逻辑电路的设计及分析

按照讲义要求，我经过简单分析和思考，在单周期 CPU 电路图的基础上，加入了状态转移模块，并让它输出当前状态（Current_State），并且在时钟上升沿时，Current_State 的值会更新为下一个状态（Next_State）。原本的单周期 CPU 中的 PC 模块、寄存器写回以及取指模块等都受到当前状态的控制，只有在当前状态为 IW 时，才会更新 PC 的值；只有在当前状态为 WB 时，才会将结果写入寄存器中。具体的电路设计如下：（见图 2）

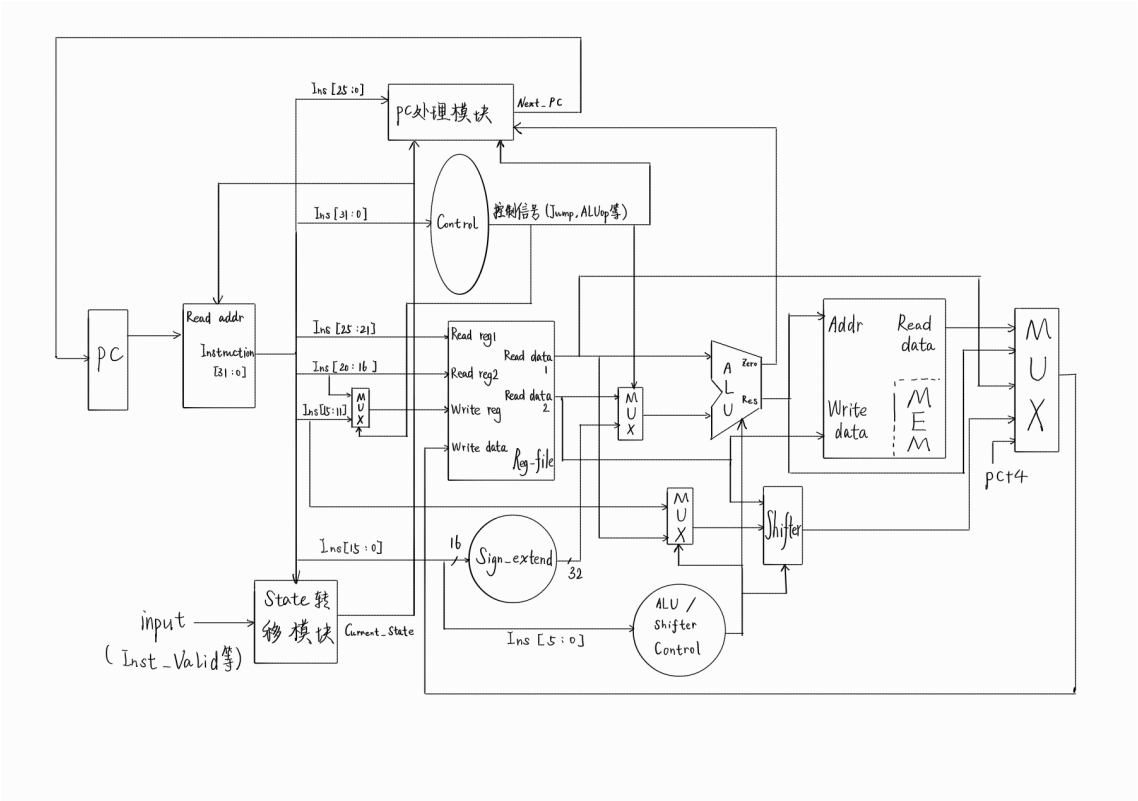

图 2: Custom_CPU 电路设计简图

上图中除了 State 转移模块外，其他模块都是单周期 CPU 中的模块，只是它们的输入和输出几乎都没有太大变化。但是 PC 模块、寄存器写回模块和取指模块的输入和输出都受到了 Current_State 的控制，而且原本在单周期 CPU 中，J 型指令往寄存器中写入 PC+8 的值，而在本次设计的 CPU 中，由于 PC 在 IW 阶段就更新了，所以在 WB 阶段写入寄存器的值就是 PC+4+ 立即数了。

2 状态转移模块的实现（三段式）

按照要求，我汇总了状态转移过程中最复杂的部分，即从 EX 向 LD、ST、WB 状态的转移。通过查阅上一次实验总结的 MIPS 指令译码表，我得出了下面的 EX 状态转移分析（见图 3）；并按照讲义提示和要求，以三段式的方式实现了状态转移模块的设计。具体的实现如下：

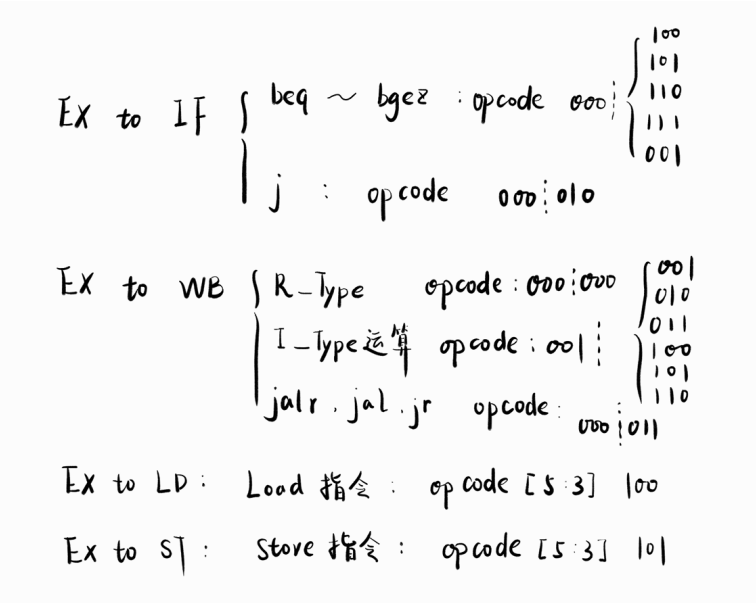

图 3: EX 状态转移分析

根据上面的分析结果，因为 EX 向下一个状态的转移是最为复杂的，所以我单独为 EX 设计了控制信号，并完成了 EX 向其他状态转移的控制信号的具体代码：（见图 4）

```
wire          EX_IF;          //EX to IF or not
wire          EX_WB;          //EX to WB or not
wire          EX_LD;          //EX to LD or not
wire          EX_ST;          //EX to ST or not
wire [8:0]    current_stata;
assign EX_IF = (Opcode[5:3] != 3'b000)                          ? 0 :
               (Opcode[2:0] != 3'b000 && Opcode[2:0] != 3'b011) ? 1 :
                                                                  0 ;
assign EX_WB = (Opcode == 6'b000000 || Opcode == 6'b000011) ? 1 :
               (Opcode[5:3] != 3'b001)                      ? 0 :
               (Opcode[2:0] != 3'b000)                      ? 1 :
                                                              0 ;
assign EX_LD = Opcode[5:3] == 3'b100;
assign EX_ST = Opcode[5:3] == 3'b101;
```

图 4: EX to next_state 模块代码实现

按照三段式的要求：以时序逻辑的方式，在 always 模块中实现了状态的置位和更新操作（见图 5）并且在 always 的组合逻辑模块中实现了状态转移的逻辑运算，得到 next_state 的值（见图 6）。最后利用不同的状态来产生控制信号，并从我的 CU 模块中输出到 CPU 中去（见图 7）。

```
always @(posedge clk) begin
    if(rst == 1'b1) begin
        current_stata_reg <= INIT;
    end
    else begin
        current_stata_reg <= next_stata_reg;
    end
end
```

图 5: 三段式——时序逻辑更新状态

```
EX:begin
    if(EX_IF) begin
        next_stata_reg = IF;
    end
    else if(EX_WB) begin
        next_stata_reg = WB;
    end
    else if(EX_ST) begin
        next_stata_reg = ST;
    end
    else if(EX_LD) begin
        next_stata_reg = LD;
    end
    else begin
        next_stata_reg = current_stata_reg;
    end
end
```

图 6: 以 always 的组合逻辑模块计算 next_state（以 EX 为例）

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

图 7: 三段式——组合逻辑计算控制信号

二、 各模块实现及完整代码分析

- 整体保持单周期的逻辑电路设计

在设计多周期 CPU 时，我的设计思路是保持单周期 CPU 的逻辑电路设计不变，只是在单周期 CPU 的基础上，增加了状态转移模块和控制信号模块。在单周期 CPU 的基础上，增加了状态转移模块和控制信号模块。这样可以避免对单周期 CPU 的逻辑电路设计进行大规模的修改，同时也可以保持单周期 CPU 的逻辑电路设计的完整性。具体的状态转移模块在上部分中已经进行了说明，这里不再赘述。下面主要介绍在本次定制型处理器设计中，增加的模块和实现的代码。与 Prj2 单周期 CPU 相比，有所增添改变的部分。

- **取指逻辑的设计**

在本次实验的单周期 CPU 中，取指只可以在 IW 结束时进行。根据我在 debug 过程中查看的 golden 模块中的 Instruction 的变化，我发现标准的取指应该是时在 IW 的最后一个时钟周期，也就是满足 current_ state=IW 且 next_state=ID 时，才会更新指令。但是在我的代码实现中，我简化了（其实也是一开始没有考虑到这么复杂）这个逻辑，我就直接在 current_state=ID 阶段更新 Instruction 的值了。（见图 8）

```
//instruction只在IF阶段更新
reg  [31:0] instruction;
always @(*) begin
    if(current_stata == IW) begin
        instruction = Instruction;
    end
end
```

图 8: 取指逻辑模块（只有在 IW 阶段更新指令）

- **RF_wdata 的改动**

相对于单周期 CPU，RF_wdata 也要有所改变。这一点我一开始没有发现，经过长达半天的 debug，我才发现 RF_wdata 在 Load 指令下很特殊，不可以单纯照搬单周期 CPU 的组合逻辑。因为 Load 指令中，输入的 Read_data 信号，是在 RDW 阶段给出的，在 WB 阶段，需要往寄存器中写的时候，又需要用到这个 Read_data 信号。所以需要给 RF_wdata 增加一个时序逻辑的 always 模块来让它可以延迟一个时钟周期，这样在我 wen=1 的时候也可以正确的写入。（见图 9）

```
//RF_wdata的选择
    assign RF_wdata = Opcode == 6'b0 ? (
                      func   == 6'b001000 || func == 6'b001001 ? PC+4    :
                      func   == 6'b001011 || func == 6'b001010 ? rsdata  :
                      func[5]== 1'b0                           ? rddata  :
                                                                 Result) :
                      Opcode      == 6'b001111 ? imm_32     :
                      Opcode[5:3] == 3'b001    ? Result     :
                      Opcode[5:3] == 3'b100    ? mem_data   :  PC+4      ;
    reg [31:0] _RF_wdata;
    always @(posedge clk) begin
        _RF_wdata <= RF_wdata;
    end
```

图 9: 增加了时序逻辑模块的 RF_wdata

- **PC 模块设计**

在单周期 CPU 中，PC 模块的设计是比较简单的，只需要在时钟上升沿时更新 PC 的值即可。但是在多周期

CPU 中，PC 模块的设计就变得复杂了，因为在多周期 CPU 中，PC 的值需要在不同的状态下进行更新。单周期中非跳转指令就是简单的 PC+4，跳转指令就是一下跳转到目标地址。而在多周期中，非跳转指令变化不大，但是跳转指令不再是一次就跳转到目标地址，而是和非跳转指令一样，在 ID 阶段就要进行一次 PC+4，而在 EX 阶段还需要再次跳转。因此相较于单周期的译码表中的逻辑，跳转的的 next_PC 的计算逻辑也要进行修改。比如之前的 PC+4[31:28],tar,2'b0 就需要改成 PC[31:28],tar,2'b0，因为 PC 在 ID 就已经加 4 了。具体的实现如下：（见图 10）

```
assign next_pc = branch    ? PC+imm_32_SE00       :
                 Opcode[1] ? {PC[31:28],tar,2'b0} :
                             rsdata  ;
reg [31:0] reg_pc;
always @(posedge clk) begin
    if(rst) begin
       reg_pc <= 32'b0;
    end
    else if(current_stata == ID) begin
       reg_pc <= PC+4;
    end
    else if(current_stata == EX) begin
        if(is_jorb) begin
            reg_pc <= next_pc;
        end
    end
end
assign PC = reg_pc;
```

图 10: PC 模块代码实现

• **RF_wen 的修改**

这里最开始我也没有发现 RF_wen 不可以简单的使用单周期 CPU 的组合逻辑电路来实现。之后经过 debug，我发现 RF_wen 的值置 1 的先决条件是 current_state=WB。这个错误一开始还并没有发现，测试了多个程序后才出现。之后我认真分析了这里出错的原因。我发现通常情况下，没有加上 current_state=WB 的条件，让 wen 在多个状态都置 1 时，会在多个时钟周期上升沿进行写回，但是也不一定会出错。而当指令比较“特殊”时，才会出现错误。比如 ADDI 指令，当操作数寄存器和写回寄存器是同一个寄存器时，才会出现错误，因为每一个时钟周期写回后，又会重新取到这个寄存器的值，就会反复计算叠加上去，导致了错误的结果。具体的 RF_wen 的实现如下：（见图 11）

```
//写使能RF_wen判断
    assign RF_wen = current_stata != WB                                  ? 1'b0 :
                    Opcode[5:3]   == 3'b101                              ? 1'b0 :
                    Opcode        == 6'b000010                           ? 1'b0 :
                    branch                                               ? 1'b0 :
                    (Opcode       == 6'b000000 && func == 6'b001000)     ? 1'b0 :
                    (Opcode       == 6'b000000 && func == 6'b001011 && zero)  ? 1'b0 :
                    (Opcode       == 6'b000000 && func == 6'b001010 && ~zero) ? 1'b0 :
                                                                           1'b1;
```

图 11: RF_wen 的实现

• **外设控制器的设计**

完成外设控制器需要做的事情事情是将需要打印的字符存入 UART 的发送队列。此前, 需要进行一个对发送队列是否为空的判断, 如果为空, 直接发送, 否则等待。判断发送队列是否为空, 需要访问发送队列状态寄存器 STAT_REG, 写入的位置则为发送队列入口寄存器 TX_FIFO（代码实现见图 12）。这些寄存器都已经过物理地址编址, 使得我们在软件层面可以通过访问的方式进行操作。

```
puts(const char *s)
{
    int i;
    //定义指向UART发送FIFO和状态寄存器的指针
    volatile unsigned int *uart_tx_fifo = uart + UART_TX_FIFO/sizeof(unsigned int);
    volatile unsigned int *uart_status = uart + UART_STATUS/sizeof(unsigned int);

    //循环遍历字符串中的每个字符，直到遇到空字符'\0'
    for (i = 0; s[i] != '\0'; i++) {
        /* if tx_fifo is full, loop (如果发送FIFO已满，则循环等待) */
        while (*uart_status & UART_TX_FIFO_FULL)
            ; //(空循环体，持续检查)
        *uart_tx_fifo = (unsigned int)s[i]; //(将字符写入发送FIFO)
    }

    return i; //(返回发送的字符数)
}
```

图 12: 外设控制器的代码实现

总结代码工作流程：

1. 初始化指向 UART 发送数据寄存器和状态寄存器的指针。
2. 逐个字符地遍历输入的字符串。
3. 对于每个字符：
   a. 检查 UART 发送 FIFO 是否已满：通过读取状态寄存器并检查特定标志位。
   b. 如果 FIFO 已满，则等待（在一个空 while 循环中不断检查），直到 FIFO 有空间。
   c. 如果 FIFO 未满，则将当前字符写入发送 FIFO 数据寄存器。
4. 当所有字符（直到结束符）都已处理完毕，返回发送的字符总数。

这种通过不断查询硬件状态来决定下一步操作的方式称为轮询 (Polling)。volatile 关键字确保了对硬件寄存器的每次读写都是真实的，不会被编译器优化掉。

• **功能计数器的实现**

本次实验先只按照讲义要求仅实现了 CLK 的计数功能，在后续 prj 中会用到其他功能计数器，之后会添加。CLK 计数器具体代码如下：（见图 13）

```
//功能计数器
    reg [31:0] cycle_cnt;
    always @(posedge clk) begin
        if(rst == 1'b1) begin
            cycle_cnt <= 32'b0;
        end
        else begin
            cycle_cnt <= cycle_cnt + 32'b1;
        end
    end
    assign cpu_perf_cnt_0 = cycle_cnt;

endmodule
```

图 13: CLK_counter 的代码实现

三、 实验中的难点及 debug 过程

1 组合逻辑和时序逻辑的使用

在实验中，如果不使用单周期 CPU 的已经设计过的代码，直接重新设计逻辑电路，完成代码的话，思路会更清晰，但是工作量会太大。因此我主体上还是使用了单周期 CPU 的代码，只是在此基础上进行修改和添加。而这时的难点就是要考虑清楚：什么时候要将原本的组合逻辑修改为时序逻辑；什么地方原本计算逻辑只与 Instruction 码有关，而不需要考虑 current_state 的值；什么地方需要考虑 current_state 的值。

其中第二点是最难的，因为在单周期 CPU 中，所有的组合逻辑都是和 Instruction 码有关的，而在多周期 CPU 中，组合逻辑不仅和 Instruction 码有关，还和 current_state 的值有关。但是因为我想要尽可能简化工作量，尽可能保留更多的原本的单周期 CPU 的逻辑设计和原代码，所以有些地方，按讲义的多周期的要求，应该需要受到 current_state 的控制，但是我没有进行修改。例如：译码阶段计算各个控制信号（见图 14 图 15）

```
//branch计算
    reg reg_brance;
    wire brance;
    assign new_branch = Opcode[5:2] == 1'h1 | Opcode == 6'b000001;
    always @(posedge clk) begin
        if(current_stata == ID) begin
            reg_brance <= new_branch;
        end
        else begin
            reg_brance <= brance;
        end
    end
    assign brance = reg_brance;
```

图 14: 两种代码对比–时序实现

```
//branch计算
    assign branch = Opcode[5:2] == 1'h1 | Opcode == 6'b000001;

//move计算
    assign move = (Opcode == 6'b000000 && (func == 6'b001011 || func == 6'b001010));

//jump计算
    assign jump = Opcode == 6'b000010 || Opcode == 6'b000011 || (Opcode == 6'b000000 && (func == 6'b001000 || func == 6'b001001));
```

图 15: 两种代码对比–组合实现

在多周期 CPU 中要求每一个控制信号应该在译码阶段，也就是图 14 中的 ID 阶段就应该计算出来。但是我发现其实按照组合逻辑产生效果也是一样的。因为 Instruction 的更新只会在 IW 阶段进行，因此我不修改单周期 CPU 中的组合逻辑也是可以正确输出控制信号结果的，即图 15 中的 branch 和 jump 信号的计算。但是又并不是所有的信号都可以简单的利用组合逻辑来替代，因此产生了很多错误，给我 debug 过程增加了难度。

2 Debug 过程

本次实验因为我尽可能保留了原本的单周期 CPU 代码，因此一开始就出现了很多错误。比如前面提到的 RF_wen 的错误，RF_wdata 的错误，取指逻辑的错误等。还有一些是因为我在设计时没有考虑到 current_state 的值会影响组合逻辑的输出，

除此以外，我在本次实验 Debug 时间最长的一个错误是本地通过了几乎所有程序后，在最后一个测试集的最后一个测试程序中出现了错误。因为我的单周期 CPU 是“正确的”，因为通过了平台的测试。因此我一直认为是我的多周期中，更改了原本代码的地方出现错误。而事实上，是我的多周期 CPU 是错误的，我的 ALU 中 NOR 计算逻辑是错误的，但是神奇的是，这个错误并没有在单周期测试中发现，因为似乎那些测试集中根本没有 NOR 指令。下面我以这个的例子来具体说明一下 Debug 过程。

举例说明：使用本地测试后的波形图进行 debug

1. 先根据测试用例，运行代码，查看报错信息，下载波形图，并根据报错信息定位波形图时刻（见图 16，图 17）

```
[queen] Queen placement: ================================================
ERROR: at                    170ns, PC = 0x00001490.
Yours:     Write_data & 0xffffffff = 0x000000f0
Reference: Write_data & 0xffffffff = 0x000000e0
================================================
```

图 16: 报错信息

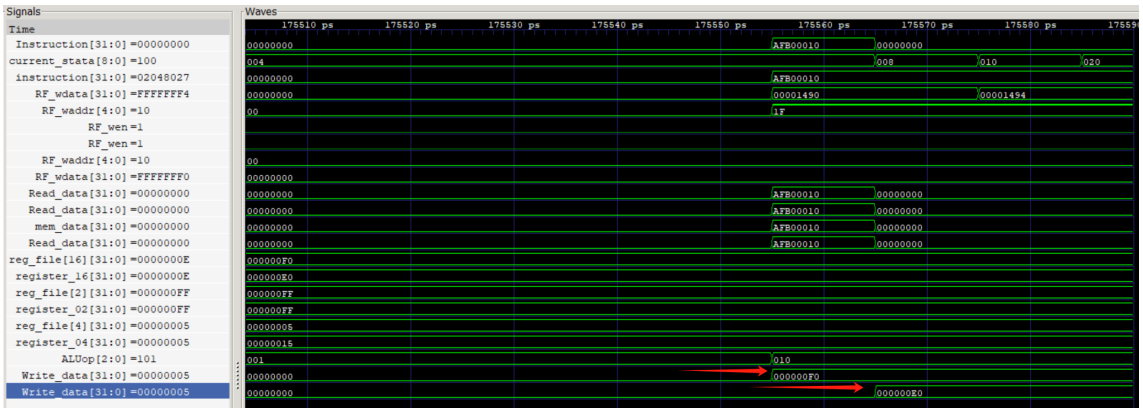

图 17: 波形图

2. 上图中箭头位置即为出错位置，可以定位到此时的指令为：AFB00010，输入 MIPS Converter 工具进行转换，得到出错指令为：（见图 18）

```
SW $s0 0x0010 $sp
Binary: 10101111101100000000000000010000
Hex: 0xafb00010
```

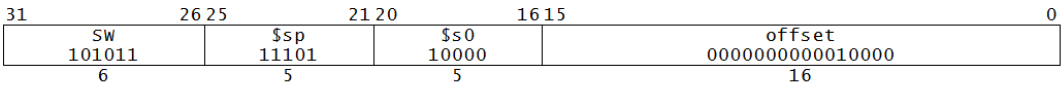

图 18: 指令分析

3. 经过指令分析，发现这里的 Load 指令，是将 16 号寄存器的值存入到内存中，因此 Write_data 错误的本质原因是我的 16 号寄存器的值有问题。这里就非常麻烦了。

   在本次实验中，老师为我们提供的 testbench 中，没有对 RF 的三个值进行检查，也就是说当我在某个时刻，我的 RF_wdata 可能就已经错误了，但是并没有被检查出来。于是程序依旧没有报错，继续向前执行，直到在之后可能很远的某个时刻，又有某个指令使用到了刚刚出错的寄存器，这时候才会被检查出来，并在波形图中表现出来。于是我调出了我的 CPU 和 golden 模块中的 RF_wen,RF_wdata 和 RF_waddr 的波形图进行对比（见图 19）

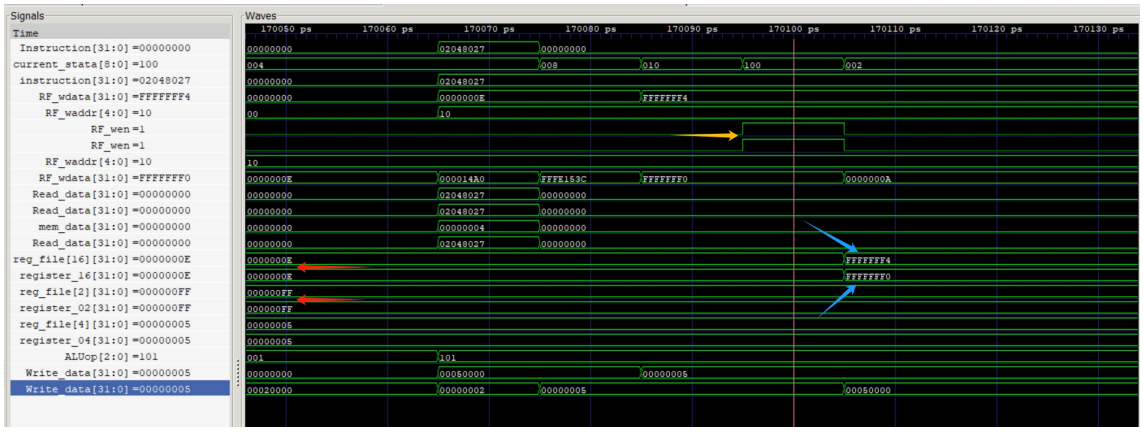

图 19: 波形图定位错误

经过了 300 多个时钟周期的回溯，检查了 30 余条指令，定位到了首次出现错误的时刻。上图中红色箭头指向说明什么是“首次出错”，因为此时的指令的两个寄存器操作数是正确的，是和 golden 模块一致的，但是它的写回寄存器的值是错误的，即蓝色箭头所指向的 RF_wdata 的值是错误的。图中黄色箭头指向说明了此时的 wen=1，需要写入。

4. 经过分析，发现此处是一个 NOR 型指令，两个操作数是正确的，而结果错误，说明了我的 ALU 出现了问题，返回 alu.v 代码界面，检查后发现，计算逻辑确实出现了错误（见图 20）

```
assign xor_result  = A^B;
assign nor_result  = A~^B;
```

图 20: alu——NOR 错误代码

5. 修改 alu.v 代码后，重新编译，运行测试集，所有测试集都成功通过。

四、 思考题

- **volitile 关键字的作用和思考**

图 21: 思考题

1 volatile 关键字的作用是什么

`volatile` 是一个 C/C++ 语言的类型限定符 (type qualifier)，它告诉编译器，被 `volatile` 修饰的变量（或指针指向的内存区域）可能会在程序控制流之外被意外地改变。因此，编译器不应该对这个变量的访问进行某些类型的优化。

具体来说，对于如下代码：

```
volatile unsigned int *uart = (void *)0x60000000;
```

`volatile` 关键字主要有以下作用：

防止编译器优化掉对该内存地址的读写操作

- **对于读操作：**当程序读取 `*uart`（或通过偏移量如 `*(uart + UART_STATUS/sizeof(unsigned int))` 读取状态寄存器）时，`volatile` 确保每次读取都会**真正地从内存地址 `0x60000000`（或其偏移地址）去获取数据**。编译器不会因为觉得“这个值刚刚读过，而且程序没有修改它，所以可以直接使用上次存放在 CPU 寄存器里的值”而优化掉实际的内存读取。这是因为硬件状态（如 UART 的 FIFO 是否已满）可能由硬件自身改变，而不是由当前程序的代码流改变。
- **对于写操作：**当程序写入 `*uart`（或通过偏移量如 `*(uart + UART_TX_FIFO/sizeof(unsigned int)) = data;` 写入数据寄存器）时，`volatile` 确保每次写入都会**真正地将数据写入到内存地址 `0x60000000`（或其偏移地址）**。编译器不会因为觉得“这个写入的值后续没有被程序读取”或者“连续两次写入同样的值”而优化掉实际的内存写入。对于硬件寄存器，写入操作本身可能就是一种触发信号或命令。

防止编译器重排（reorder）对该内存地址的访问顺序（相对于其他 `volatile` 访问）

虽然 `volatile` 对指令重排的保证在 C/C++ 标准中不如专门的内存屏障（memory barriers/fences）那么强，但它确实会限制编译器在优化时对 `volatile` 变量访问的自由重排。程序中访问这些硬件寄存器的顺序往往非常重要。例如，你可能需要先检查状态寄存器，然后再写入数据寄存器。`volatile` 有助于维持这种预期的访问顺序。

简而言之

`volatile` 告诉编译器:“这个变量很特殊,不要对它的访问做任何自作聪明的假设和优化,老老实实地按照代码写的顺序去读写内存。”

在 UART 的上下文中

- **`UART_STATUS` 寄存器:**它的值(比如 FIFO 是否满、是否收到数据等)是由 UART 硬件实时更新的。程序需要读取这个寄存器的最新状态才能做出正确的判断。如果没有 `volatile`,编译器可能在循环中只读取一次状态寄存器,然后一直使用这个旧值,导致程序逻辑错误(例如,FIFO 已空但程序以为还是满的,从而卡死在等待循环中)。
- **`UART_TX_FIFO` 寄存器:**向这个寄存器写入数据,会触发 UART 硬件将数据发送出去。这个写入操作具有“副作用”(side effect)。如果编译器优化掉了这个写入,数据就不会被发送。

2 如果去掉 volatile 会出现什么后果

- **读操作:**如果去掉 `volatile`,编译器可能会优化掉对状态寄存器的重复读取,导致程序在判断 FIFO 是否满时使用了过时的值,从而可能导致死循环或错误的行为。假设有一个代码中,它循环等待 UART 发送 FIFO 变为空闲:(见图 22)

```
// 假设 uart_status 指向 UART_STATUS 寄存器
// 错误示例: 如果 uart_status 不是 volatile
while (*uart_status & UART_TX_FIFO_FULL) {
    // 等待FIFO不满
}
// FIFO 不满了, 可以发送数据
*uart_tx_fifo = next_char;
```

图 22: 试例代码 1

如果 uart_status 不是 volatile,编译器在第一次进入 while 循环时读取 *uart_status 的值。如果此时 UART_TX_FIFO_FULL 位是 1(FIFO 已满),编译器可能会将 *uart_status 的值加载到一个 CPU 寄存器中。在后续的循环迭代中,编译器可能会认为既然程序代码没有修改 *uart_status,那么它的值就不会变,于是直接使用 CPU 寄存器中缓存的旧值进行判断。即使 UART 硬件已经发送完数据,FIFO 变空,UART_STATUS 寄存器的实际值已经改变,但程序仍然使用旧的缓存值,导致 while 循环变成死循环,程序卡死。

- **写操作:**如果去掉 `volatile`,编译器可能会优化掉对数据寄存器的写入操作,导致数据无法发送出去。
- **重排问题:**去掉 `volatile` 可能导致编译器重排对状态寄存器和数据寄存器的访问顺序,从而破坏程序逻辑。编译器为了优化,可能会将 *uart_tx_fifo = char_to_send; 这个写操作重排到 if 判断之后。但硬件操作的逻辑可能是先写入数据,然后状态寄存器才会反映出相应的错误状态。如果顺序颠倒,错误检测逻辑就会失效:(见图 23)

```c
// 错误示例：如果 uart_status 和 uart_tx_fifo 都不是 volatile
*uart_tx_fifo = char_to_send;
if (*uart_status & SOME_ERROR_FLAG) { // 假设这里检查发送错误
    // 处理错误
}
```

图 23: 试例代码 2

3 总结思考

volatile 关键字在与硬件交互时的极端重要性。去掉 volatile 几乎肯定会导致程序在与硬件交互时出现严重错误，尤其是在开启编译器优化的情况下。这是嵌入式系统编程中一个非常基础且关键的概念。

通过思考题得到的几点认识：

- volatile 的核心是告诉编译器“不要优化对这个内存的访问”。
- 硬件寄存器的值会独立于程序流程而改变（由硬件驱动）。
- 对硬件寄存器的写操作本身就是一种行为（触发硬件）。

五、 实验所耗时间

在课后，你花费了大约 30（代码完成 5 小时，Debug 完成 15 小时，完成实验报告 10 小时） 小时完成此次实验。

致谢：感谢韩初晓同学和朱徐塬助教在实验 debug 过程中给予的帮助和指导