

中国科学院大学

《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 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。

1 定制 MIPS 功能形处理器电路设计及说明

1.1 逻辑电路结构

本次 CPU 设计可延用上次实验中设计的单周期 (以及简单多周期) 处理器的电路结构。因此, 两次实验的电路结构图是十分相似的。不过, 在多周期 CPU 的设计过程中, 我们新引入了状态转移自动机模块。此外, 原先在单周期 CPU 中的部分组合逻辑模块需要变成时序的。例如寄存器写回数据源选择器, 由于写回操作是在下一个指令周期完成的, 因此, 这个模块非常有必要设计成一个时序逻辑模块。

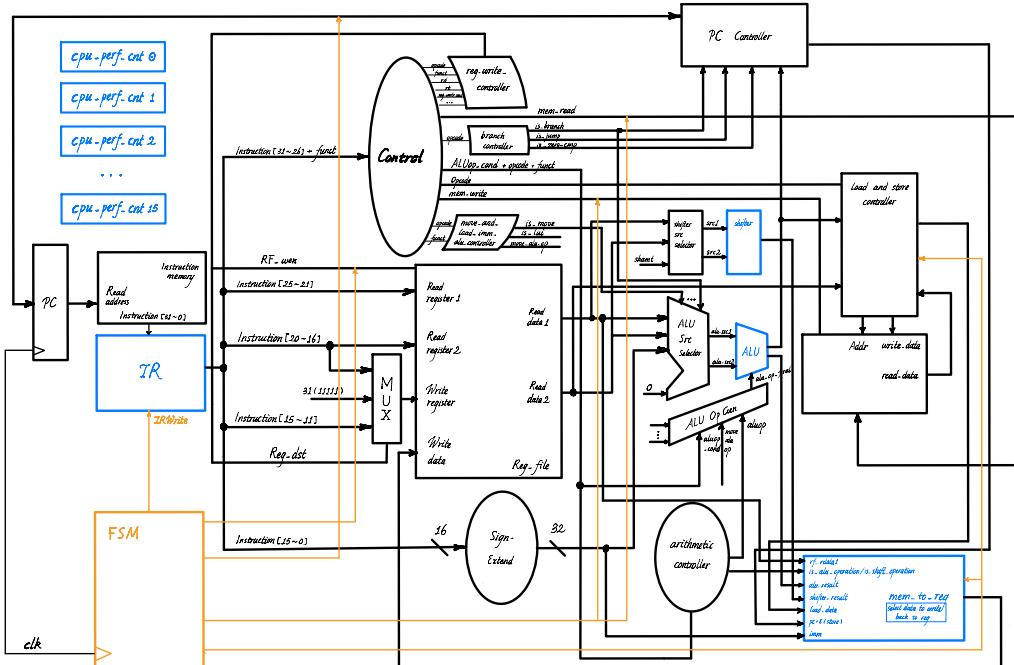


图 1: 定制 MIPS 功能形处理器电路结构图

图中,黄色的模块便是 FSM 状态控制模块,蓝色模块则为将原来的组合逻辑模块改为时序模块的部分。

1.2 RTL 设计及说明

1.2.1 状态机的定义

在设计多周期 CPU 的过程中,我主要参照了讲义中所给出的 FSM 的状态图。本次实验的状态转移图相比于上次实验的简单状态转移更加复杂,因此,在设计状态机时也需要更加复杂的代码。

- INIT 状态的添加:** 本次实验的状态转移图中新添加了一个初始状态 (INIT), 每次接收到重置信号后, 需要先回到这个状态, 然后在下一周期再转移到取指阶段。
- 访存阶段的添加:** 本次实验的状态转移图中, 访存阶段分为两个状态: 存储器读 (LD) 和存储器写 (ST)。在这两个状态中, CPU 需要等待存储器的响应信号, 为此, 在 LD 状态和 WB 状态之间还添加了 RDW 状态。

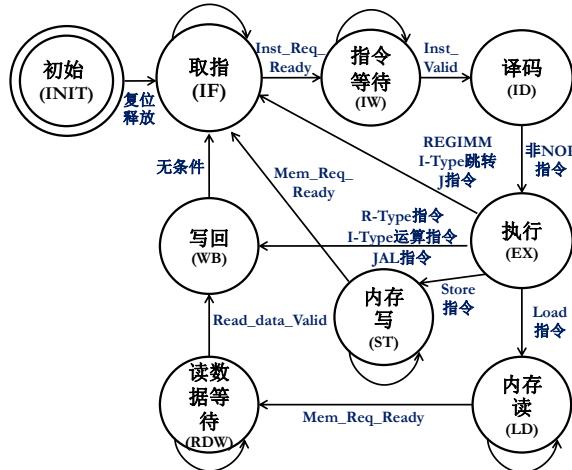


图 2: FSM 状态转移图

图 2 为有限状态自动机的状态转移图。据此,我设计了下面的状态机模块代码:

首先,我们需要定义 FSM 状态数,FSM 共有九个状态,其定义如下:

```
// -- FSM State Definitions --
localparam INIT = 9'b0000000001, // Initial State
      IF  = 9'b0000000010, // Instruction Fetch
      IW  = 9'b0000000100, // Instruction Wait
      ID  = 9'b0000001000, // Instruction Decode
      EX  = 9'b0000010000, // Execute
      ST  = 9'b0000100000, // Store - Memory Write
      LD  = 9'b0001000000, // Load - Memory Read
      RDW = 9'b0100000000, // Read Data Wait
      WB  = 9'b1000000000; // Write Back
```

状态机的第一段描述状态机的状态更新过程,其逻辑如下:

```
// -- FSM State Register --
always @(posedge clk) begin
  if (rst) begin
```

```

    current_state <= INIT;      // Reset state
end else begin
    current_state <= next_state; // Update state based on FSM logic
end
end

```

状态机的第二段根据译码信息决定当前状态对应的下一个状态是什么。完全对应于图 2 中的状态转移图。其逻辑如下：

```

// -- FSM Next State Logic --
always @(*) begin
    case (current_state)
        INIT : begin
            if (rst) begin
                next_state = INIT;      // Reset state
            end
            else begin
                next_state = IF;       // Move to Instruction Fetch
            end
        end
        IF   : begin
            if (Inst_Req_Ready) begin
                next_state = IW;      // Instruction requested, wait for it
            end
            else begin
                next_state = IF;       // Keep fetching
            end
        end
        IW   : begin
            if (Inst_Valid) begin
                next_state = ID;      // Instruction received, move to Decode
            end
            else begin
                next_state = IW;       // Wait for valid instruction
            end
        end
        ID   : begin
            if (NOP) begin
                next_state = IF;       // NOP, go back to Fetch
            end
            else begin
                next_state = EX;       // Proceed to Execute
            end
        end
        EX   : begin
            if (is_j_instr || is_REGIMM || is_branch) begin

```

```

        next_state = IF;           // Control transfer instructions, back to Fetch
    end
else if (is_load_store) begin
    if (is_load) begin
        next_state = LD;      // Load instruction, go to Load state
    end
    else if (is_store) begin
        next_state = ST;      // Store instruction, go to Store state
    end
    else begin
        next_state = IF;      // Should not happen, default to Fetch
    end
end
else begin
    next_state = WB;          // ALU/other ops, go to Write Back
end
end

LD  : begin
if (Mem_Req_Ready) begin
    next_state = RDW;        // Memory ready for load, wait for read data
end
else begin
    next_state = LD;         // Wait for memory to be ready
end
end

RDW : begin
if (Read_data_Valid) begin
    next_state = WB;          // Data read from memory, go to Write Back
end
else begin
    next_state = RDW;        // Wait for valid read data
end
end

ST  : begin
if (Mem_Req_Ready) begin
    next_state = IF;          // Memory ready for store, data written, back to Fetch
end
else begin
    next_state = ST;          // Wait for memory to be ready
end
end

WB  : begin
next_state = IF;           // Write Back complete, back to Fetch
end

default: begin

```

```

        next_state = IF;           // Default to Instruction Fetch
    end
endcase
end

```

状态机的第三段便是将当前状态同各种控制信号相联系。首先便是握手信号。这些信号直接由当前状态和下一个状态决定：

```

assign Inst_Req_Valid = (current_state == IF); // Request instruction fetch
assign Inst_Ready = (current_state == INIT) || (current_state == IW);
assign Read_data_Ready = (current_state == INIT) || (current_state == RDW);

```

1.2.2 重要寄存器的添加

我们需要添加指令寄存器 (IR) 用于在一个指令周期内存储当前指令。因此，它在一个指令周期内仅更新一次。其定义与更新逻辑如下：

```

assign IRWrite = (current_state == IW) && Inst_Valid;

// -- Instruction Register Logic --
always @(posedge clk) begin
    if (IRWrite) begin
        IR <= Instruction; // Reset instruction register
    end else begin
        IR <= IR;          // Fetch instruction from memory
    end
end

```

由于 ALU 和移位器的计算结果不会在当前周期立刻被用到，因此我们需要将其计算结果暂存起来。为此，我添加了 ALU 和移位器的结果寄存器。其定义与更新逻辑如下：

```

//-- ALU Execution --
wire [31:0] alu_out;           // ALU computation result

alu instance_alu (
    .A      (alu_src1),      // Input A from Src Sel
    .B      (alu_src2),      // Input B from Src Sel
    .ALUop  (alu_op_final), // Input Opcode from Op Gen
    .Overflow (),
    .CarryOut (),
    .Zero   (alu_zero),     // Output: Zero flag -> To EX (PC Ctrl), WB
    .Result  (alu_out)       // Output: ALU computation result -> To EX (PC Ctrl), MEM, WB
);

always @(posedge clk) begin
    alu_result <= alu_out; // Store ALU result for MEM stage
end

//-- Shifter Execution --
wire [31:0] shifter_out;       // Shifter computation result

```

```

shifter instance_shifter (
    .A      (shifter_src1),      // Input Data from Src Sel
    .B      (shifter_src2[4:0]), // Input Shift Amount from Src Sel
    .ShiftOp (shifter_op),     // Input Opcode from ID
    .Result   (shifter_out)     // Output: Shifter result -> To WB
);

always @(posedge clk) begin
    shift_result <= shifter_out; // Store Shifter result for WB stage
end

```

这里有一个小问题,那就是我注意到 ALU 和移位器的计算结果都会在下一个时钟周期被使用,因此我为了简单起见,让结果寄存器在每个时钟周期都更新。事实上更好的设计是,仅让 ALU 和移位器的结果在某几个状态更新。

由于写回操作是在下一个指令周期开始的上升沿完成的,因此我们需要将写回数据源选择器设计成一个时序逻辑模块。其定义与更新逻辑如下:

```

//-- Write Back Data Selection (Uses ID signals, EX results, MEM results) --
wire [31:0] _RF_wdata;                                // Write data to RegFile
(processed from ID, EX, MEM)
assign _RF_wdata = (is_lui) ? lui_result :           // P1: LUI (from ID)
                    (current_state == RDW && next_state == WB) ? load_data : // P2: Load
                    (is_move) ? RF_rdata1 :           // P3: MOVZ/MOVN result
                    (is_alu_operation) ? alu_result : // P4: Other ALU result from EX
                    (is_shift_operation) ? shift_result : // P5: Shifter result
                    (is_link_jump) ? pc_store :       // P6: Link address (PC+8)
                    32'b0;                           // Default: 0 (No writeback)
                                                // -> To RegFile instance

always @(posedge clk) begin
    RF_wdata <= _RF_wdata;                         // Store write data for RegFile
end

```

1.2.3 PC 更新逻辑

我定义了控制 PC 在何时更新的组合逻辑信号,使用这个组合逻辑信号控制 PC 时序模块的更新逻辑。其定义如下:

```

//=====
// PIPELINE STAGE 1: Instruction Fetch (IF)
//=====

//-- PC Register --
wire pc_write_enable = (current_state == IW && next_state == ID) ||
                      ((current_state == EX) && is_jump) ||
                      ((current_state == EX) && is_branch && branch_condition_satisfied);

pc instance_pc (
    .clk      (clk),
    .rst      (rst),

```

```

.pc_write_enable (pc_write_enable), // Input: PC write enable signal (from control logic)
.next_pc  (next_pc),           // Input: Next PC value (Calculated in EX stage)
.pc        (current_pc)        // Output: Current PC value (Used across stages)
);

//=====
// Module: pc (Remains unchanged)
// Description: Program Counter register. Stores the address of the next
//               instruction to be fetched.
//=====

module pc (
    input      clk,
    input      rst,
    input      pc_write_enable,
    input [31:0] next_pc,
    output reg [31:0] pc
);
    always @ (posedge clk) begin
        if (rst) begin
            pc <= 32'h00000000;
        end else if (pc_write_enable) begin
            pc <= next_pc;
        end
    end
endmodule

```

1.2.4 外设控制器

我们需要在 printf.c 的 puts 函数中实现 UART 控制器的驱动程序。

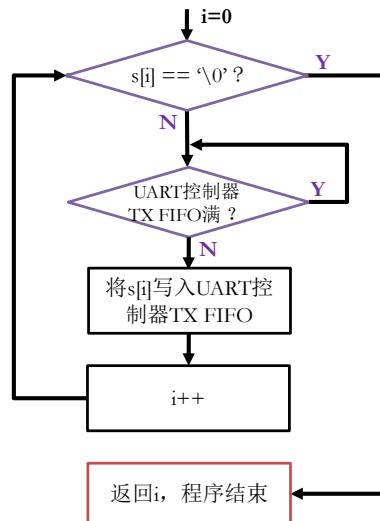


图 3: UART 控制器驱动程序工作流程图

在 TX FIFO 已满时，循环等待；当 FIFO 未满时，向其中写入下一个数据。在实现的时候有两个细节需要

注意：首先，我们知道 UART 的起始地址为 0x60000000，同时，我们知道 UART_TX_FIFO 和 UART_STATUS 的偏移量为 0x04 和 0x08。我们希望通过 C 语言指针来访问到这两个地址。但是，根据 C 语言指针的加减运算规则，指针加一在内存中对应的偏移量和指针的数据类型密切相关。因此，在这里我们应当在加之前把偏移量除以 sizeof(unsigned int)。另一个细节是 volatile 的使用。这是为了告诉编译器，这个变量的值随时可能在程序代码的控制范围之外被改变（在这里，是我们设计的 CPU 硬件），因此，对这个变量的读写操作具有副作用，因此不能被优化掉，并且每次访问都必须严格执行。这个驱动程序的具体实现如下：

```
=====
 * puts: send characters in input string to UART TX FIFO in order
 *
 * @s: input string
 *
 * Return: return the actual string length that has been sent out
=====
*/
int puts(const char *s) {

    // TODO: Add your driver code here

    int i;
    volatile unsigned int *uart_tx_fifo = uart + UART_TX_FIFO/sizeof(unsigned int);
    volatile unsigned int *uart_status = uart + UART_STATUS/sizeof(unsigned int);

    for (i = 0; s[i] != '\0'; i++) {
        /* if tx_fifo is full, loop */
        while (*uart_status & UART_TX_FIFO_FULL)
            ;
        *uart_tx_fifo = (unsigned int)s[i];
    }

    return i;
}
```

1.2.5 性能计数器

在本次实验中，我用满了十五个性能计数器。统计了一些不同的性能计数，并发现了一些有意思的现象。我的性能计数器定义如下：

```
=====
// Performance Counters
=====

reg [31:0] perf_cycle_count;      // cnt_0
reg [31:0] perf_retired_inst_count; // cnt_1
reg [31:0] perf_retired_load_count; // cnt_2
reg [31:0] perf_retired_store_count; // cnt_3
reg [31:0] perf_branch_executed_count; // cnt_4
reg [31:0] perf_branch_taken_count; // cnt_5
reg [31:0] perf_if_stall_count;     // cnt_6
```

```

reg [31:0] perf_mem_access_stall_count; // cnt_7 (LD/ST stalls on Mem_Req_Ready)
reg [31:0] perf_iw_stall_count;      // cnt_8
reg [31:0] perf_rdw_stall_count;    // cnt_9
reg [31:0] perf_jump_executed_count; // cnt_10
reg [31:0] perf_alu_op_executed_count; // cnt_11
reg [31:0] perf_shift_op_executed_count; // cnt_12
reg [31:0] perf_nop_in_id_count; // cnt_13
reg [31:0] perf_total_mem_ops_count; // cnt_14
reg [31:0] perf_reg_writes_count; // cnt_15

```

我为这些指令设定了一些更新条件。对于每个性能计数器，只有在时钟上升沿时更新条件为真，性能计数器的值才更新。这些更新条件如下：

```

// Determine when an instruction is considered retired (Same as before)
assign increment_retired_inst =
  (current_state == WB && Regwrite_fsm && !NOP && (is_alu_operation || is_shift_operation ||
    is_lui || is_link_jump)) ||
  (current_state == WB && Regwrite_fsm && !NOP && is_load) ||
  (current_state == ST && Mem_Req_Ready && !NOP && is_store) ||
  (current_state == EX && !NOP &&
    (is_j_instr || is_jr_instr || is_REGIMM || (is_branch && !is_link_jump)) &&
    (next_state == IF))
  );

assign increment_retired_load = (current_state == WB && Regwrite_fsm && !NOP && is_load);
assign increment_retired_store = (current_state == ST && Mem_Req_Ready && !NOP && is_store);
assign increment_branch_executed = (current_state == EX && !NOP && is_branch);
assign increment_branch_taken = (current_state == EX && !NOP && is_branch &&
  branch_condition_satisfied);
assign increment_if_stall = (current_state == IF && !Inst_Req_Ready && !rst);
assign increment_mem_access_stall = ((current_state == LD || current_state == ST) &&
  !Mem_Req_Ready && !rst);
assign increment_iw_stall = (current_state == IW && !Inst_Valid && !rst);
assign increment_rdw_stall = (current_state == RDW && !Read_data_Valid && !rst);
assign increment_jump_executed = (current_state == EX && !NOP && is_jump);
// For ALU/Shift, ensure it's not a mem op address calc or branch compare
assign increment_alu_op_executed = (current_state == EX && !NOP && is_alu_operation &&
  !is_load_store && !is_branch && !is_jump);
assign increment_shift_op_executed = (current_state == EX && !NOP && is_shift_operation &&
  !is_load_store && !is_branch && !is_jump);
assign increment_nop_in_id = (current_state == ID && NOP && !rst); // NOP is decoded in ID
assign increment_total_mem_ops = (current_state == EX && !NOP && is_load_store); // Count when it
  enters EX, destined for MEM
assign increment_reg_writes = (current_state == WB && RF_wen && !rst);

```

这些信号的更新逻辑是相似的，在下面仅列举几个：

```

// cnt_0: Cycle Count
always @(posedge clk) begin
  if (rst) begin

```

```

perf_cycle_count <= 32'd0;
end else begin
    perf_cycle_count <= perf_cycle_count + 1;
end
end
assign cpu_perf_cnt_0 = perf_cycle_count;

// cnt_1: Retired Instruction Count
always @(posedge clk) begin
    if (rst) begin
        perf_retired_inst_count <= 32'd0;
    end else if (increment_retired_inst) begin
        perf_retired_inst_count <= perf_retired_inst_count + 1;
    end
end
assign cpu_perf_cnt_1 = perf_retired_inst_count;

...
// cnt_15: Register File Writes
always @(posedge clk) begin
    if (rst) begin
        perf_reg_writes_count <= 32'd0;
    end else if (increment_reg_writes) begin
        perf_reg_writes_count <= perf_reg_writes_count + 1;
    end
end
assign cpu_perf_cnt_15 = perf_reg_writes_count;

endmodule

```

为了让程序能够访问性能计数器记录的结果,我们需要对 `perf_cnt.h`, `perf_cnt.c`, 以及 `bench.c` 中的相关函数作出修改,使得它们兼容我们的新增功能。

首先,我丰富了 `perf_cnt.h` 中的结果结构体定义:

```

typedef struct Result {
    int pass;           // Whether the benchmark passed or failed
    unsigned long msec; // Typically used for cycle count of the benchmark run (cpu_perf_cnt_0)

    unsigned long perf_retired_inst_count; // From cpu_perf_cnt_1
    unsigned long perf_retired_load_count; // From cpu_perf_cnt_2
    unsigned long perf_retired_store_count; // From cpu_perf_cnt_3
    unsigned long perf_branch_executed_count; // From cpu_perf_cnt_4
    unsigned long perf_branch_taken_count; // From cpu_perf_cnt_5
    unsigned long perf_if_stall_count; // From cpu_perf_cnt_6
    unsigned long perf_mem_access_stall_count; // From cpu_perf_cnt_7
    unsigned long perf_iw_stall_count; // From cpu_perf_cnt_8
    unsigned long perf_rdw_stall_count; // From cpu_perf_cnt_9
    unsigned long perf_jump_executed_count; // From cpu_perf_cnt_10
    unsigned long perf_alu_op_executed_count; // From cpu_perf_cnt_11
}

```

```

    unsigned long perf_shift_op_executed_count; // From cpu_perf_cnt_12
    unsigned long perf_nop_in_id_count; // From cpu_perf_cnt_13
    unsigned long perf_total_mem_ops_count; // From cpu_perf_cnt_14
    unsigned long perf_reg_writes_count; // From cpu_perf_cnt_15

} Result;

```

随后,我实现了一次性读取所有性能计数器结果的函数,以及初始化和在测试程序结束后统计各性能计数器结果的 C 函数。这些函数的定义如下:

```

// Define base address for performance counters for clarity
#define PERF_CNT_BASE_ADDR 0x60010000UL

// Define offsets for each counter based on the PPT address map
#define PERF_CNT_0_OFFSET 0x0000
#define PERF_CNT_1_OFFSET 0x0008
#define PERF_CNT_2_OFFSET 0x1000
#define PERF_CNT_3_OFFSET 0x1008
#define PERF_CNT_4_OFFSET 0x2000
#define PERF_CNT_5_OFFSET 0x2008
#define PERF_CNT_6_OFFSET 0x3000
#define PERF_CNT_7_OFFSET 0x3008
#define PERF_CNT_8_OFFSET 0x4000
#define PERF_CNT_9_OFFSET 0x4008
#define PERF_CNT_10_OFFSET 0x5000
#define PERF_CNT_11_OFFSET 0x5008
#define PERF_CNT_12_OFFSET 0x6000
#define PERF_CNT_13_OFFSET 0x6008
#define PERF_CNT_14_OFFSET 0x7000
#define PERF_CNT_15_OFFSET 0x7008

// Helper macro to get the full address of a counter
#define PERF_COUNTER_ADDR(offset) (volatile unsigned long *) (PERF_CNT_BASE_ADDR + (offset))

static inline void read_all_perf_counters(Result *res) {

    res->perf_retired_inst_count = *PERF_COUNTER_ADDR(PERF_CNT_1_OFFSET);
    res->perf_retired_load_count = *PERF_COUNTER_ADDR(PERF_CNT_2_OFFSET);
    res->perf_retired_store_count = *PERF_COUNTER_ADDR(PERF_CNT_3_OFFSET);
    res->perf_branch_executed_count = *PERF_COUNTER_ADDR(PERF_CNT_4_OFFSET);
    res->perf_branch_taken_count = *PERF_COUNTER_ADDR(PERF_CNT_5_OFFSET);
    res->perf_if_stall_count = *PERF_COUNTER_ADDR(PERF_CNT_6_OFFSET);
    res->perf_mem_access_stall_count = *PERF_COUNTER_ADDR(PERF_CNT_7_OFFSET);
    res->perf_iw_stall_count = *PERF_COUNTER_ADDR(PERF_CNT_8_OFFSET);
    res->perf_rdw_stall_count = *PERF_COUNTER_ADDR(PERF_CNT_9_OFFSET);
    res->perf_jump_executed_count = *PERF_COUNTER_ADDR(PERF_CNT_10_OFFSET);
    res->perf_alu_op_executed_count = *PERF_COUNTER_ADDR(PERF_CNT_11_OFFSET);
    res->perf_shift_op_executed_count = *PERF_COUNTER_ADDR(PERF_CNT_12_OFFSET);
    res->perf_nop_in_id_count = *PERF_COUNTER_ADDR(PERF_CNT_13_OFFSET);
    res->perf_total_mem_ops_count = *PERF_COUNTER_ADDR(PERF_CNT_14_OFFSET);
}

```

```

    res->perf_reg_writes_count = *PERF_COUNTER_ADDR(PERF_CNT_15_OFFSET);

}

```

在程序开始和程序结束时工作的两个性能统计函数：

```

void bench_prepare(Result *res) {
    // TODO [COD]
    // Add preprocess code, record performance counters' initial states.
    // You can communicate between bench_prepare() and bench_done() through
    // static variables or add additional fields in `struct Result`
    res->msec = _uptime();
    read_all_perf_counters(res);
}

void bench_done(Result *res) {
    // TODO [COD]
    // Add postprocess code, record performance counters' current states.
    res->msec = _uptime() - res->msec;
    Result temp;
    read_all_perf_counters(&temp);
    res->perf_retired_inst_count = temp.perf_retired_inst_count - res->perf_retired_inst_count;
    res->perf_retired_load_count = temp.perf_retired_load_count - res->perf_retired_load_count;
    res->perf_retired_store_count = temp.perf_retired_store_count - res->perf_retired_store_count;
    res->perf_branch_executed_count = temp.perf_branch_executed_count -
        res->perf_branch_executed_count;
    res->perf_branch_taken_count = temp.perf_branch_taken_count - res->perf_branch_taken_count;
    res->perf_if_stall_count = temp.perf_if_stall_count - res->perf_if_stall_count;
    res->perf_mem_access_stall_count = temp.perf_mem_access_stall_count -
        res->perf_mem_access_stall_count;
    res->perf_iw_stall_count = temp.perf_iw_stall_count - res->perf_iw_stall_count;
    res->perf_rdw_stall_count = temp.perf_rdw_stall_count - res->perf_rdw_stall_count;
    res->perf_jump_executed_count = temp.perf_jump_executed_count - res->perf_jump_executed_count;
    res->perf_alu_op_executed_count = temp.perf_alu_op_executed_count -
        res->perf_alu_op_executed_count;
    res->perf_shift_op_executed_count = temp.perf_shift_op_executed_count -
        res->perf_shift_op_executed_count;
    res->perf_nop_in_id_count = temp.perf_nop_in_id_count - res->perf_nop_in_id_count;
    res->perf_total_mem_ops_count = temp.perf_total_mem_ops_count - res->perf_total_mem_ops_count;
    res->perf_reg_writes_count = temp.perf_reg_writes_count - res->perf_reg_writes_count;
}

```

将统计结果输出到终端的 main 函数：

```

int main() {
    int pass = 1;

    _Static_assert(ARR_SIZE(benchmarks) > 0, "non benchmark");

    for (int i = 0; i < ARR_SIZE(benchmarks); i++) {

```

```

Benchmark *bench = &benchmarks[i];
current      = bench;
setting      = &bench->settings[SETTING];
const char *msg = bench_check(bench);
printf("[%s] %s: ", bench->name, bench->desc);

if (msg != NULL) {
    printk("Ignored %s\n", msg);
} else {
    unsigned long best_msec           = ULONG_MAX;
    int succ                         = 1;

    unsigned long best_perf_retired_inst_count = 0;
    unsigned long best_perf_retired_load_count = 0;
    unsigned long best_perf_retired_store_count = 0;
    unsigned long best_perf_branch_executed_count = 0;
    unsigned long best_perf_branch_taken_count = 0;
    unsigned long best_perf_if_stall_count = 0;
    unsigned long best_perf_mem_access_stall_count= 0;
    unsigned long best_perf_iw_stall_count = 0;
    unsigned long best_perf_rdw_stall_count = 0;
    unsigned long best_perf_jump_executed_count = 0;
    unsigned long best_perf_alu_op_executed_count = 0;
    unsigned long best_perf_shift_op_executed_count = 0;
    unsigned long best_perf_nop_in_id_count = 0;
    unsigned long best_perf_total_mem_ops_count = 0;
    unsigned long best_perf_reg_writes_count = 0;

    for (int r = 0; r < REPEAT; r++) {
        Result current_run_res;
        run_once(bench, &current_run_res);

        printk(current_run_res.pass ? "*" : "X");
        succ &= current_run_res.pass;

        if (current_run_res.msec < best_msec) {
            best_msec           = current_run_res.msec;
            best_perf_retired_inst_count = current_run_res.perf_retired_inst_count;
            best_perf_retired_load_count = current_run_res.perf_retired_load_count;
            best_perf_retired_store_count = current_run_res.perf_retired_store_count;
            best_perf_branch_executed_count = current_run_res.perf_branch_executed_count;
            best_perf_branch_taken_count = current_run_res.perf_branch_taken_count;
            best_perf_if_stall_count   = current_run_res.perf_if_stall_count;
            best_perf_mem_access_stall_count= current_run_res.perf_mem_access_stall_count;
            best_perf_iw_stall_count   = current_run_res.perf_iw_stall_count;
            best_perf_rdw_stall_count  = current_run_res.perf_rdw_stall_count;
            best_perf_jump_executed_count = current_run_res.perf_jump_executed_count;
            best_perf_alu_op_executed_count = current_run_res.perf_alu_op_executed_count;
            best_perf_shift_op_executed_count = current_run_res.perf_shift_op_executed_count;
            best_perf_nop_in_id_count   = current_run_res.perf_nop_in_id_count;
        }
    }
}

```

```

        best_perf_total_mem_ops_count = current_run_res.perf_total_mem_ops_count;
        best_perf_reg_writes_count = current_run_res.perf_reg_writes_count;
    }
}

if (succ) {
    printk(" Passed.\n");
} else {
    printk(" Failed.\n");
}
pass &= succ;

if (succ && best_msec != ULONG_MAX) {
    printk(" --- Performance Counters for [%s] (Best Cycle Run Deltas) ---\n", bench->name);
    printk("   Cycles:           %u\n", (unsigned int)best_msec);
    printk("   Retired Instructions: %u\n", (unsigned int)best_perf_retired_inst_count);
    printk("   Retired Loads:      %u\n", (unsigned int)best_perf_retired_load_count);
    printk("   Retired Stores:     %u\n", (unsigned int)best_perf_retired_store_count);
    printk("   Branches Executed:  %u\n", (unsigned int)best_perf_branch_executed_count);
    printk("   Branches Taken:     %u\n", (unsigned int)best_perf_branch_taken_count);
    printk("   IF Stalls:          %u\n", (unsigned int)best_perf_if_stall_count);
    printk("   MEM Access Stalls:  %u\n", (unsigned int)best_perf_mem_access_stall_count);
    printk("   IW Stalls:          %u\n", (unsigned int)best_perf_iw_stall_count);
    printk("   RDW Stalls:         %u\n", (unsigned int)best_perf_rdw_stall_count);
    printk("   Jumps Executed:     %u\n", (unsigned int)best_perf_jump_executed_count);
    printk("   ALU Ops Executed:   %u\n", (unsigned int)best_perf_alu_op_executed_count);
    printk("   Shift Ops Executed: %u\n", (unsigned int)best_perf_shift_op_executed_count);
    printk("   NOPs in ID:         %u\n", (unsigned int)best_perf_nop_in_id_count);
    printk("   Total MEM Ops Issued: %u\n", (unsigned int)best_perf_total_mem_ops_count);
    printk("   Register Writes:    %u\n", (unsigned int)best_perf_reg_writes_count);
    printk(" --- End of Counters for [%s] ---\n", bench->name);
} else if (succ) {
    printk(" --- No performance data to display for [%s] (no runs completed with data) ---\n",
           bench->name);
}
}

printf("benchmark finished\n");

if(pass)
    hit_good_trap();
else
    nemu_assert(0);

return 0;
}

```

2 实验中遇到的问题,对问题的思考以及解决办法

2.1 debug 记录

由于我已经在项目二中解决了不少问题,因此,在本次实验中,我的 debug 过程比较轻松。我遇到的第一个问题是 PC 的更新逻辑问题。之前,在理想内存情况下,PC 只需要在 IF 阶段更新即可。我在本次实验的第一版设计中沿用了这个设计。但是,我发现在非理想内存中,这容易导致一个非常严重的问题。那就是,从内存中获取指令往往需要很多个时钟周期,假如在这些时钟周期中处理器都处于 IF 状态,那么处理器的 PC 就会在这一个状态下反复更新,这样,处理器的行为就会出现异常。于是,我改变了 PC 的更新逻辑,变成了下面代码中所示的形式,解决了问题:

```
//-- PC Register --
wire pc_write_enable = (current_state == IW && next_state == ID) ||
                      ((current_state == EX) && is_jump) ||
                      ((current_state == EX) && is_branch && branch_condition_satisfied);
```

我遇到的第二个问题是在执行 `memcpy` 测试程序的时候,写回寄存器堆的数据和金标准不一致。我发现,这仍然是理想内存处理器改造成真实内存处理器的遗留问题导致的。在理想内存情况下, `mem_read` 即可作为判断写回数据是否为 `load_data` 的依据。但是,在当前情况下, `mem_read` 信号有效的时间段并非应当写回的时间段,因此,我们需要另一个标志来判断是否选择写回 `load_data`。最终写回数据源选择器的实现逻辑如下:

```
//-- Write Back Data Selection (Uses ID signals, EX results, MEM results) --
wire [31:0] _RF_wdata;                                     // Write data to RegFile
(processed from ID, EX, MEM)
assign    _RF_wdata      = (is_lui)           ? lui_result :      // P1: LUI (from ID)
                           (current_state == RDW && next_state == WB) ? load_data : // P2: Load
                           (is_move)          ? RF_rdata1 :        // P3: MOVZ/MOVN result
                           (is_alu_operation) ? alu_result :       // P4: Other ALU result from EX
                           (is_shift_operation) ? shift_result : // P5: Shifter result
                           (is_link_jump)     ? pc_store :        // P6: Link address (PC+8)
                           32'b0;                           // Default: 0 (No writeback)
                                                 // -> To RegFile instance

always @ (posedge clk) begin
    RF_wdata <= _RF_wdata;                                // Store write data for RegFile
end
```

我遇到的第三个问题是在测试 `hello` 测试程序的时候,在本地仿真时仅仅会打印测试通过信息,而不会打印出任何的字符串。但是在上传到线上时,程序能够正常测试并输出正确的字符串。这个问题至今未解决,也没有看到其他同学遇到。可能是因为我在本地配置某些环境的时候的操作偶然间导致了这个错误。

2.2 关于源代码实现的一些小思考

在完成性能计数器的设计时,我由于害怕时钟周期数太大,在打印中使用了 `%lu` 作为打印格式。我发现,在实际执行结果中, `printf` 并没有正确识别这个占位符,而是直接将 `lu` 作为文字打印出来。这与我的预期严重不符。于是我便阅读源代码打算一探究竟。我注意到源代码自己实现了 `stdio.h`,并且其中有这样的表述:`#define printf printf`。这个宏定义说明,老师希望我们注意,这个打印函数并不是原本库函数的 `printf`。随后,我又查看了 `printf.c` 中 `printf` 函数的实现,我发现,该函数的功能依赖于 `mini_vsnprintf` 函数和我们实现的 `puts` 函数。`mini_vsnprintf` 中有这样的表述:

```
switch (ch) {
```

```

case 0:
    goto end;

case 'u':
case 'd':
    len = mini_itoa(va_arg(va, unsigned int), 10, 0, (ch == 'u'), bf,
                    zero_pad);
    _puts(bf, len, &b);
    break;

case 'x':
case 'X':
    len = mini_itoa(va_arg(va, unsigned int), 16, (ch == 'X'), 1, bf,
                    zero_pad);
    _puts(bf, len, &b);
    break;

case 'c':
    _putc((char)(va_arg(va, int)), &b);
    break;

case 's':
    ptr = va_arg(va, char *);
    _puts(ptr, mini_strlen(ptr), &b);
    break;

default:
    _putc(ch, &b);
    break;
}

```

这说明,%lu 并不在我们的打印函数的支持范围内。因此,我们只好改用 %u。

2.3 一个奇怪的 bug

如果已经实现了 MIPS CPU 和 RISCV32 CPU, 并且测试完 MIPS CPU 之后希望立即测试 RISCV32 CPU, 那么测试程序运行一定会出错(即便已经确保代码是正确的)。一种解决办法是将 RISCV32 CPU 的代码原封不动的复制并粘贴一次, 这可以解决问题。反之, 先测试 RISCV32 后测试 MIPS 也会出错, 解决方案是一样的。目前尚不清楚这一问题造成的机理。推测 make 命令的行为和代码的最新改动时间存在一定的联系。

3 仿真结果与分析

3.1 仿真结果分析

下面是我设计的处理器在某个样例上的仿真波形图。可以看出, 仿真结果与预期一致。

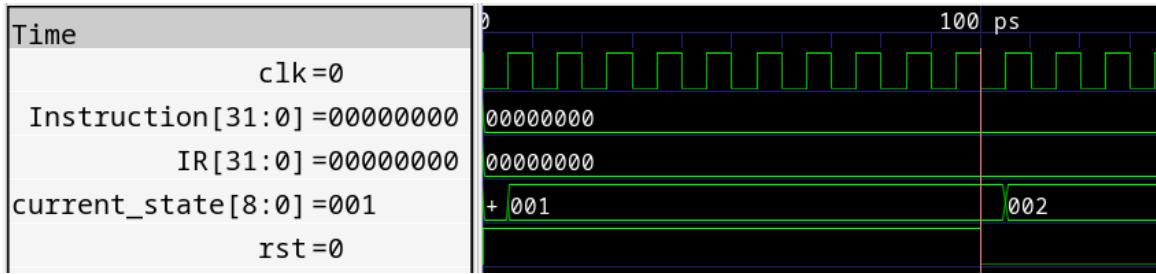


图 4: 状态机初始状态转换测试

图 4 展示了 CPU 从初始化状态向取值阶段的状态转移。可以看出,当 `rst` 信号不再有效时,状态机在下一个时钟上升沿正确的转向了 IF 状态。

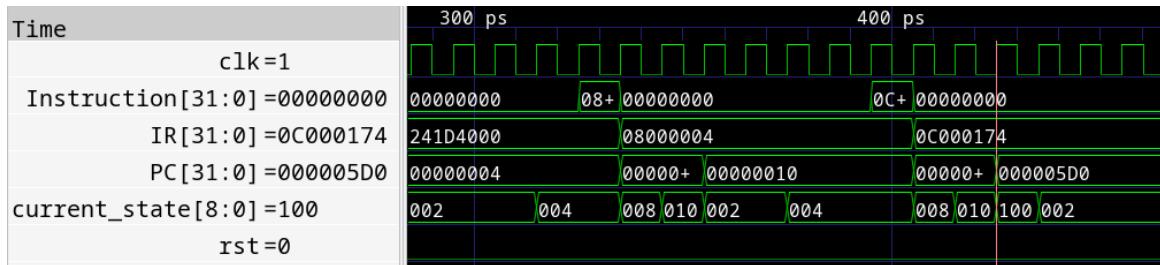


图 5: 状态机跳转行为

图 5 展示了 CPU 在跳转指令的时候的行为。可以看出,在 EX 阶段结束的时钟上升沿,PC 正确完成了更新,并且正确的进入了 WB 阶段,将 $PC + 8$ 存入三十一号寄存器。

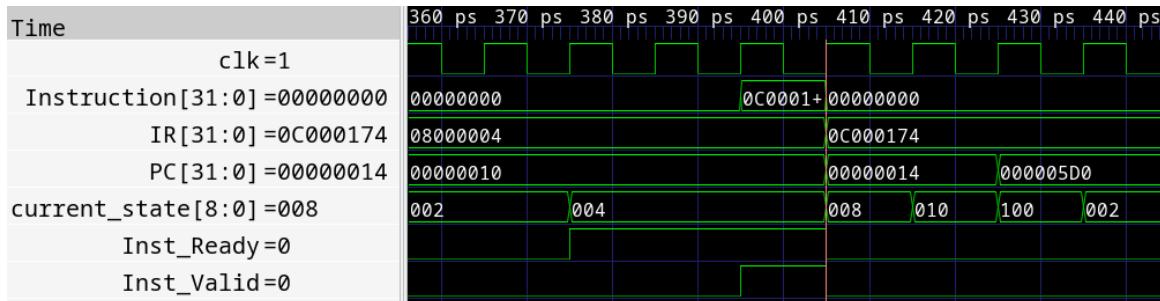


图 6: 握手信号工作过程

图 6 展示了 CPU 在指令等待阶段的握手信号的工作过程。可以看到,当 `Ready` 和 `Valid` 信号同时有效时,握手成功,指令内容写入指令寄存器。

3.2 外设控制器统计数据分析

由于我的本地测试环境的原因,我无法在本地测试 `hello` 和 `microbench` 测试程序。于是,我只能在网上测试这个程序。下面是我在网上测试的结果:

```
$ bash ./fpga/design/ucas-cod/run/simple_cpu/fpga_run.sh $BENCH_SUITE $TARGET_DESIGN $CPU_ISA
RUNNER_CNT = 3
Completed FPGA configuration
Launching hello benchmark...
tggetattr: Inappropriate ioctl for device
```

```

reset: before MMIO access...
reset: MMIO accessed
axi_firewall_unblock: firewall error status: 00000000
main: before DDR accessing...
main: DDR accessed...
reset: before MMIO access...
reset: MMIO accessed
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADFOOD
00000000100000000200000000300000000400000005
50 50 -50 4294967246
time 10189.50ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/hello/riscv32/elf/hello passed
Hit good trap

```

可以看出，测试结果输出的字符串与预期一致，并且测试通过。下面是我在网上测试的 microbench 测试程序的结果（部分）：

```

$ bash ./fpga/design/ucas-cod/run/simple_cpu/fpga_run.sh $BENCH_SUITE $TARGET_DESIGN $CPU_ISA
RUNNER_CNT = 1
Completed FPGA configuration
Launching 15pz benchmark...
tggetattr: Inappropriate ioctl for device
reset: before MMIO access...
reset: MMIO accessed
axi_firewall_unblock: firewall error status: 00000000
main: before DDR accessing...
main: DDR accessed...
reset: before MMIO access...
reset: MMIO accessed
[15pz] A* 15-puzzle search: * Passed.
--- Performance Counters for [15pz] (Best Cycle Run Deltas) ---
Cycles: 522472293
Retired Instructions: 5223425
Retired Loads: 2109998
Retired Stores: 1118774
Branches Executed: 662796
Branches Taken: 624442
IF Stalls: 0
MEM Access Stalls: 5466314
IW Stalls: 346871556
RDW Stalls: 140455787
Jumps Executed: 1271
ALU Ops Executed: 728807
Shift Ops Executed: 77933
NOPs in ID: 0
Total MEM Ops Issued: 3228768

```

```

Register Writes:      3441867
--- End of Counters for [15pz] ---
benchmark finished
time 5296.06ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/microbench/riscv32/elf/15pz passed
Hit good trap

...
Launching ssort benchmark...
tggetattr: Inappropriate ioctl for device
reset: before MMIO access...
reset: MMIO accessed
axi_firewall_unblock: firewall error status: 00000000
main: before DDR accessing...
main: DDR accessed...
reset: before MMIO access...
reset: MMIO accessed
[ssort] Suffix sort: * Passed.
--- Performance Counters for [ssort] (Best Cycle Run Deltas) ---
Cycles:              44730754
Retired Instructions: 618538
Retired Loads:        11291
Retired Stores:       7433
Branches Executed:   132631
Branches Taken:      66423
IF Stalls:            0
MEM Access Stalls:   33584
IW Stalls:            40950220
RDW Stalls:           762717
Jumps Executed:      869
ALU Ops Executed:    283903
Shift Ops Executed:  181542
NOPs in ID:          0
Total MEM Ops Issued: 18720
Register Writes:     478486
--- End of Counters for [ssort] ---
benchmark finished
time 1405.11ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/microbench/riscv32/elf/ssort passed
Hit good trap
pass 9 / 9
Job succeeded

```

以上数据反映了几个很有意思的现象：

- 时钟周期与指令周期: 我们发现时钟周期数远大于指令周期数。这虽然是多周期处理器最基本的设计要求,但是依然印证了计组理论课中所讲授的知识。
- NOP 指令计数: 通过统计数据我们发现, NOP 指令的计数为零, 这说明测试样例中没有被注入 NOP 指令。但是, 通过查看本地仿真波形, 我发现, 这个数据并不为零, 甚至 NOP 指令计数多达几千条。这说明, 本地仿真测试用例和线上仿真测试可能存在不一致性。
- IF 阶段停顿: 通过统计数据我们发现, IF 阶段的停顿计数为零, 这说明测试样例中没有发生 IF 阶段的停顿。但是, 通过查看本地仿真波形, 我发现这个数据并不为零, 这说明了本地仿真和线上仿真测试用例可能不一致。

4 对讲义中思考题的理解和回答

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

C 标准手册中说, 一个具有 `volatile` 限定类型的对象, 可能会以实现(通常指编译器)未知的方式被修改, 或者可能具有其他未知的副作用。因此, 任何引用此类对象的表达式, 都必须严格按照抽象机器的规则进行求值。此外, 在每个序列点, 最后存储在该对象中的值都必须与抽象机器所规定的值一致, 除非该值已被前面提到的未知因素所修改。对于具有 `volatile` 限定类型的对象, 何为对它的‘访问’, 则是由实现定义的。

因此, 在我们使用 `volatile` 关键字修饰的对象时, 编译器不会对其进行优化。这样, 我们就可以确保每次访问该对象时, 都会从内存中读取最新的值, 而不是使用寄存器中的缓存值。如果不加上, 则会导致仿真结果乱码和本地仿真时无法输出字符串等问题。

那么, 更具体地, 是否使用 `volatile` 关键字对程序的编译结果会产生什么影响呢? 我们来看如下的 C 程序:

```
int puts(const char *s) {
    // TODO: Add your driver code here

    int i;
    volatile unsigned int *uart_tx_fifo = uart + UART_TX_FIFO/sizeof(unsigned int);
    volatile unsigned int *uart_status = uart + UART_STATUS/sizeof(unsigned int);

    for (i = 0; s[i] != '\0'; i++) {
        /* if tx_fifo is full, loop */
        while (*uart_status & UART_TX_FIFO_FULL)
            ;
        *uart_tx_fifo = (unsigned int)s[i];
    }

    return i;
}
```

如果使用了 `volatile` 关键字, `while` 循环在每次进入循环时会重新从内存中读取数据至寄存器中, 并判断是否满足队列满的条件。但是, 如果不使用该关键字, 编译后的程序只会访存一次, 将访存时队列的状态存于寄存器中, 在执行 `while` 循环时, 会反复比较该寄存器的值是否满足队列满的条件。这样, 如果第一次访存时队列为空, 那么后面就会一直认为队列为空, 不断地向 FIFO 中写入数据。如果第一次访存时队列为满, 那么程序就会一直卡在 `while` 循环中, 不终止。这就是为什么本地仿真会不终止, 线上仿真会出现乱码。

5 收获与感想

本次“定制 MIPS 功能型处理器设计”实验让我受益匪浅,是一次理论与实践深度结合的宝贵经历。

首先,通过本次实验,我对多周期 MIPS 处理器的设计原理与实现细节有了更为深刻和具象的理解。从单周期处理器的基础出发,过渡到多周期设计,我亲身体会到了为了优化性能、平衡各阶段延迟所做的必要改进。特别是在状态机(FSM)的设计上,本次实验的状态转移图远比之前复杂,引入了如 INIT(初始状态)、IW(指令等待)、LD(访存读)、ST(访存写)、RDW(读数据等待)等多个精细控制状态。这让我深刻理解了 FSM 在复杂时序控制中的核心作用,以及如何通过状态转移来协调处理器内部各个部件的有序工作,尤其是在与非理想内存进行交互时,如何通过握手信号(如 Inst_Req_Ready, Mem_Req_Ready, Inst_Valid, Read_data_Valid)来管理数据流和处理潜在的停顿。

其次,在具体的 RTL 设计层面,我学会了如何在多周期设计中合理地引入和管理关键寄存器。例如,指令寄存器(IR)用于在一个指令周期内锁存当前指令;ALU 和移位器的结果寄存器用于暂存计算结果,以供后续周期使用;写回数据源选择器也从组合逻辑改为了时序逻辑,以适应写回操作在下一指令周期完成的特性。这些设计让我明白了时序逻辑在多周期处理器中的重要性,以及如何通过寄存器来切分数据通路,解决跨周期数据依赖的问题。PC 更新逻辑的调整也是一个重要的学习点,理解了在非理想内存条件下,PC 不能简单地在每个 IF 状态都更新,而是需要更精确的条件控制,例如在指令有效获取后(IW -> ID)或跳转/分支指令执行时更新。

第三,外设控制器(UART)驱动程序的编写,尤其是对 C 语言中 `volatile` 关键字作用的深入探究,是一大实践收获。通过在 `puts` 函数中与 `UART_TX_FIFO` 和 `UART_STATUS` 寄存器交互,我理解了内存映射 I/O 的原理,以及 `volatile` 如何防止编译器对硬件寄存器的访问进行不当优化,确保每次读写操作都能真实反映硬件状态。这个问题也让我反思了软件与硬件协同设计时需要注意的细节。

第四,性能计数器的设计与应用是本次实验的另一大亮点。我设计并实现了多达十五个性能计数器,用于统计周期数、各类指令(加载、存储、分支、跳转、ALU、移位)的执行次数、不同类型的停顿(IF 阶段停顿、访存停顿、IW 停顿、RDW 停顿)等。这不仅锻炼了我设计复杂计数逻辑的能力,更重要的是,通过分析这些计数器在运行不同测试程序时的输出数据,我对处理器的动态行为、性能瓶颈(例如大量的 IW 和 RDW 停顿反映了存储器访问的延迟影响)有了直观的认识。这让我体会到性能分析工具对于处理器设计优化的重要性。

最后,实验过程中遇到的问题及其解决过程,如 PC 更新逻辑错误、`memcpy` 测试中写回数据源选择错误等,都极大地锻炼了我的调试能力和问题定位能力。从现象出发,分析波形,回溯 Verilog 代码逻辑,再结合 MIPS 指令集行为进行排查,这个过程虽然曲折,但每一次成功解决问题都带来了成就感和更深层次的理解。

总而言之,本次实验不仅巩固了《计算机组成原理》课程所学的理论知识,如数据通路、控制器设计、存储器层次结构、I/O 接口等,更让我掌握了将理论应用于实践,设计、实现、调试一个功能相对完整的多周期处理器的工程能力。这些宝贵的经验和技能,对我未来进一步学习更复杂的计算机体系结构以及从事相关领域的研究或工作都将打下坚实的基础。

6 实验所耗时间

在课后,你花费了大约 50 小时完成此次实验。

7 致谢

感谢宁彦祯同学和宋俊仪同学,他们的真知灼见给了我很大的启发;感谢朱徐塬学长及各位助教,您的帮助对我有非常大的指导意义。