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 5.1 实验名称 增强功能型处理器设计

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 电路设计及说明

本次实验的目标是设计并实现一个支持部分 RV32I 指令集的五级流水线处理器。鉴于流水线设计的复杂性, 我将处理器划分为取指 (Instruction Fetch, IF)、译码与寄存器读取 (Instruction Decode/Register Fetch, ID)、执行 (Execute, EX)、访存 (Memory Access, MEM) 和写回 (Write Back, WB) 五个主要阶段, 并设计了相应的模块来完成各自的功能。与理论课学习的经典五级流水线结构基本一致, 但在此基础上, 我特别关注了数据冒险和控制冒险的处理。为此, 设计中实现了一个前递单元 (Forwarding Unit, FWDU) 来解决大部分数据冒险, 并通过一套全局暂停机制 (`pipeline_advance_enable`) 及特定的 Load-Use 冒险暂停信号 (`ld_use_hazard_stall`) 来控制流水线的流动和气泡插入。特别地,本设计尝试在 ID 阶段即进行分支条件的判断和跳转目标地址的计算, 以期尽早解决控制冒险。

1.1 数据通路总体结构

本 CPU 采用了经典的五级流水线结构,分别为取指 (Instruction Fetch, IF)、译码与寄存器读取 (Instruction Decode/Register Fetch, ID)、执行 (Execute, EX)、访存 (Memory Access, MEM) 和写回 (Write Back, WB)。流水线的核心思想是通过并行处理不同指令的不同阶段来提高处理器的吞吐率。

为了管理数据和控制信号在各级之间的传递,设计中使用了四组主要的流水线寄存器:IF/ID 寄存器 (`fd_..._reg`)、ID/EX 寄存器 (`de_..._reg`)、EX/MEM 寄存器 (`em_..._reg`) 和 MEM/WB 寄存器 (`mw_..._reg`)。这些寄存器的更新受到全局流水线推进使能信号 `pipeline_advance_enable` 的控制。

本设计的关键特点之一是在 ID 阶段进行分支指令的条件判断和跳转目标地址的计算。IDU 模块会输出计算得到的下一 PC 地址 (`next_pc_from_idu_internal`) 给 IFU 模块。这种早期解析控制流的策略旨在减少控制冒险带来的性能损失, 但事实上, 这种策略会导致 IDU 模块的关键路径过长, 使得处理器的时序不容易收敛。在后期 cache 的设计过程中,这个问题尤为明显。

为处理数据冒险, 设计中实现了一个前递单元 (FWDU), 它能够将 EX、MEM、WB 阶段的结果数据前递给 ID 阶段需要这些数据的指令。对于无法通过前递完全解决的 Load-Use 冒险, 则通过 `ld_use_hazard_stall` 信号触发流水线暂停。该暂停机制会: 1) 优先在 ID/EX 流水线寄存器中插入气泡, 以清除导致后续冒险检测持续有效的条件; 2) 通过影响全局的 `pipeline_advance_enable` 信号, 暂停 IFU 的取指和 IF/ID 流水线寄存器的更新, 直到冒险解除。

IFU 和 MEMU 模块内部均包含状态机 (FSM) 来管理与外部存储器(指令存储器和数据存储器)的握手协议和潜在的多周期操作,它们输出的就绪信号 (`if_stage_ready`, `mem_stage_ready`) 是构成 `pipeline_advance_enable` 的重要组成部分,从而实现了基于存储器响应的流水线反压。

1.2 关键模块结构设计说明

下面将对流水线中各个关键模块的功能和设计进行说明。

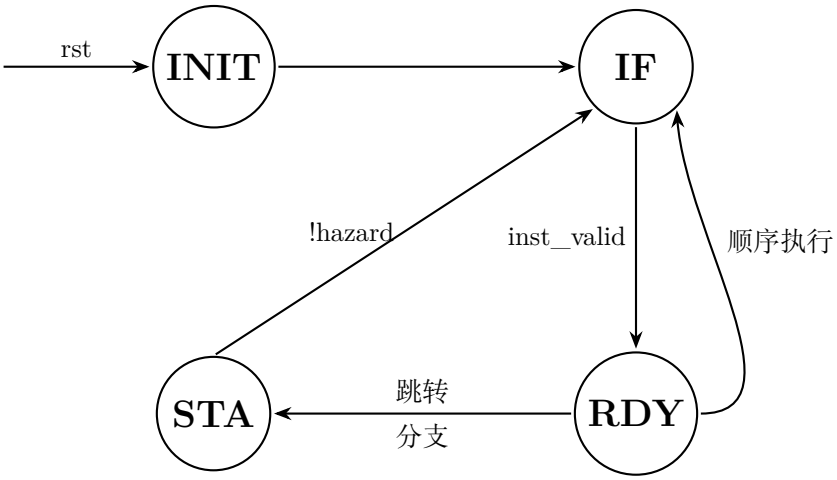
1.2.1 取指单元 (IFU - Instruction Fetch Unit)

IFU 模块负责根据程序计数器 (PC) 的值从指令存储器获取指令。其主要输入包括时钟、复位、来自 IDU 计算的下一 PC 地址 (`next_pc`)、Load-Use 冒险暂停信号 (`ld_use_harzard`) 以及全局流水线推进使能 (`pipeline_advance_enable`)。主要输出包括当前获取的指令 (`IR`)、指令有效信号 (`fetched_inst_valid`)、当前 PC 值 (也作为顶层 CPU 的 PC 输出 `PC`)、以及与指令存储器的握手信号 (`Inst_Req_Valid`, `Inst_Ready`) 和自身的就绪信号 (`IF_Ready`)。

IFU 内部包含一个状态机来管理取指流程, 其状态包括初始化 (INIT)、发送取指请求 (IF)、等待指令返回 (IW)、就绪 (RDY) 以及因冒险暂停的等待 PC 更新状态 (STA)。PC 的更新逻辑也封装在 IFU 内部,它在 STA 状态且无冒险时,根据输入的 `next_pc` 更新 PC 值。当 `ld_use_harzard` 有效时,IFU 会暂停 PC 更新并停留在 STA 状态,直到冒险解除。IFU 取到的指令和对应 PC 会锁存到 IF/ID 流水线寄存器。

在后续实现指令 cache 后,我发现 CPU 与 cache 之间繁杂的握手信号影响了从 cache 取指令的效率,因此我简化了状态机,将 IF 与 IW 合并为一个状态。同时,我发现,顺序执行指令时无需等待 ID 阶段计算出的目标地址,直接取 `PC + 4` 即可。因此,我优化了这段逻辑,在 `is_sequential_fetch` 的情况下进一步提高了取指效率。最终,IFU 的工作效率可以达到两拍一取指。

IF 阶段的状态机示意图如下,不过需要注意的是,该状态机必须与 icache 配合使用才能生效。因为当前状态机不包含和内存的完整握手信号,完整的内存交互信号由 icache 来负责。




```
module ifu (
  input clk,
  input rst,

  output reg [31:0] IR,
  output reg      fetched_inst_valid,

  output reg [31:0] PC,
```

```
  output          Inst_Req_Valid,
  input           Inst_Req_Ready,

  input     [31:0] instruction,
  input           Inst_Valid,
  output          Inst_Ready,

  input     [31:0] next_pc,

  input           ld_use_harzard,
  input           pipeline_advance_enable,

  output          IF_Ready,

  input           branch_condition_satisfied, // For perf counters
  input           Read_data_Ready, // For perf counters
  input           Read_data_Valid,

  output          IRWrite, // For perf counters
  output          IF_stall,
  output          IW_stall
);

  localparam INIT = 5'b00001;
  localparam IF = 5'b00010;
  localparam IW = 5'b00100;
  localparam RDY = 5'b01000;
  localparam STA = 5'b10000;

  reg [4:0] next_state;
  reg [4:0] current_state;

  wire _ld_use_branch_handler;
  assign _ld_use_branch_handler = branch_condition_satisfied && Read_data_Ready && Read_data_Valid;
  reg ld_use_branch_handler;

  always @(posedge clk) begin
    ld_use_branch_handler <= _ld_use_branch_handler;
  end

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

  always @(*) begin
    case (current_state)
      INIT : begin
```

```
      if (rst) begin
        next_state = INIT;
      end
      else begin
        next_state = IF;
      end
    end
//    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 = RDY;
//      end
//      else begin
//        next_state = IW;
//      end
//    end
    IF : begin // to boost frequency, this state handles both instruction fetch and stall
      if (Inst_Valid) begin
        next_state = RDY;
      end else begin
        next_state = IF;
      end
    end
    RDY : begin
      if (pipeline_advance_enable) begin
        if (is_sequential_fetch) begin
          next_state = IF;
        end else begin
          next_state = STA;
        end
      end
      else begin
        next_state = RDY;
      end
    end
    STA : begin
      if (!ld_use_harzard) begin
        next_state = IF;
      end
      else begin
        next_state = RDY;
      end
    end
```

```
    default :
      next_state = INIT;
  endcase
end

// -- FSM Output Logic --
wire pc_write_enable;
assign IRWrite = (current_state == IF) && Inst_Valid;
assign Inst_Req_Valid = (current_state == IF); // Request instruction fetch
assign Inst_Ready = (current_state == INIT) || (current_state == IF);
assign IF_Ready = (current_state == RDY);
assign pc_write_enable = (current_state == STA) && !ld_use_harzard;
assign pc_write_sequential = (current_state == RDY) && is_sequential_fetch && !ld_use_harzard &&
    pipeline_advance_enable;
assign IF_stall = (current_state == IF && !Inst_Req_Ready && !rst);
assign IW_stall = (current_state == IW && !Inst_Valid && !rst);

wire [ 6:0] opcode;          // Opcode field
assign opcode = IR[6:0];
wire is_sequential_fetch;
assign is_sequential_fetch = !(opcode == 7'b1101111) && // JAL
                         !(opcode == 7'b1100111) && // JALR
                         !(opcode == 7'b1100011); // Store
wire [31:0] sequential_next_pc;
assign sequential_next_pc = PC + 4; // Sequential fetch is just PC + 4

always @(posedge clk) begin
  if (rst) begin
    PC <= 32'b0;
  end
  else if (pc_write_sequential) begin
    PC <= sequential_next_pc;
  end
  else if (pc_write_enable || ld_use_branch_handler) begin
    PC <= next_pc;
  end
end

always @(posedge clk) begin
  if (rst) begin
    IR <= 32'b0;
    fetched_inst_valid <= 0;
  end
  else if (current_state == IF && Inst_Valid) begin
    IR <= instruction;
    fetched_inst_valid <= 1'b1;
  end
end
endmodule
```

1.2.2 译码单元 (IDU - Instruction Decode Unit)

IDU 模块是流水线的核心控制信号产生单元。它接收来自 IF/ID 寄存器的指令 (IR) 和 PC (pc)，以及经过前递单元处理后的源操作数值 (RF_rdata1, RF_rdata2)。

IDU 的主要功能包括：

- **指令字段解码：**提取 opcode, funct3, funct7, rs1, rs2, rd 等字段。

```
// -- Instruction Field Decoding --
assign    opcode  = IR[ 6: 0]; // Opcode field
assign    funct3  = IR[14:12]; // Funct3 field
assign    funct7  = IR[31:25]; // Funct7 field
assign    rs1     = IR[19:15]; // Source register 1
assign    rs2     = IR[24:20]; // Source register 2
assign    rd      = IR[11: 7]; // Destination register
assign    RF_waddr = rd;     // Register file write address
```

- **立即数生成：**根据指令类型生成不同格式的立即数 (imm_I, imm_S, imm_B, imm_U, imm_J) 并选择当前指令所需的立即数 (imm)。

```
// -- Immediate type decoding --
assign    imm_I   = {{20{IR[31]}}, IR[31:20]};                  // Immediate value (I-type)
assign    imm_S   = {{20{IR[31]}}, IR[31:25], IR[11:7]};        // Immediate value (S-type)
assign    imm_B   = {{20{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0}; // Immediate value
assign    imm_U   = {IR[31:12], 12'b0};                         // Immediate value (U-type)
assign    imm_J   = {{12{IR[31]}}, IR[19:12], IR[20], IR[30:21], 1'b0}; // Immediate value
assign    imm     = (is_I) ? imm_I :
                    (is_S) ? imm_S :
                    (is_B) ? imm_B :
                    (is_U) ? imm_U :
                    (is_J) ? imm_J : 32'b0; // Default immediate value
```

- **指令类型判断：**产生 is_R, is_I, is_S, is_B, is_U, is_J, is_load, is_store 等控制信号。

```
// -- Control Signals for Instruction Types --
assign    is_I    = is_OPIMM || is_load || is_jalr; // I-type instruction
assign    is_S    = (opcode == 7'b0100011);    // S-type instruction
assign    is_B    = (opcode == 7'b1100011);    // B-type instruction
assign    is_U    = is_lui || is_auipc;        // U-type instruction
assign    is_J    = (opcode == 7'b1101111);    // J-type instruction
assign    is_R    = (opcode == 7'b0110011);    // R-type instruction
```

- **ALU 与移位器操作数及操作码选择：**根据指令类型选择送往 EX 阶段的 ALU 操作数 (alu_src1, alu_src2)、移位器操作数 (shifter_src1, shifter_src2) 以及对应的操作码 (alu_op, shifter_op)。同时产生 is_alu_operation 和 is_shifter_operation 信号。

```
// -- Decode ALU and shifter operations based on instruction type --
assign alu_src1 = (is_jalr)              ? current_pc : // For AUIPC, use current PC
                  (is_I || is_R || is_S || is_B) ? RF_rdata1 : // rs1 for most operations
                  (is_J)                 ? current_pc : // PC for JAL offset calculation
```

```
                (is_lui)                  ? 32'b0     : // Zero for LUI (0 + imm)
                (is_auipc)                ? current_pc : // PC for AUIPC (PC + imm)
                                            32'b0;     // Default to zero

assign alu_src2 = (is_jalr)    ? 4          : // JALR uses 4 for address calculation
                (is_I)         ? imm        : // Immediate for I-types
                (is_J)         ? 4          : // JALR stores pc + 4 in rd
                (is_R)         ? RF_rdata2  : // rs2 for R-type operations
                (is_S)         ? imm        : // Immediate for S-type
                (is_B)         ? RF_rdata2  : // rs2 for B-type (branch comparison)
                (is_U)         ? imm        : // Immediate for U-type (LUI/AUIPC)
                                  32'b0;       // Default to zero

assign shifter_src1 = RF_rdata1;               // Source for shift operations is always rs1
assign shifter_src2 = (is_R) ? RF_rdata2 :     // R-type shifts use lower 5 bits of rs2
                      (is_I) ? imm       :     // I-type shifts use lower 5 bits of imm
                               32'b0;           // Default
```

- **访存控制信号生成:**产生 mem_read_internal, mem_write_internal, mem_width, mem_signed 以及用于 Store 指令的写数据 write_data (其值来源于输入的 RF_rdata2)。

```
// mem_read and mem_write logic
assign mem_read_internal = is_load; // Load instructions
assign mem_write_internal = is_S; // Store instructions
```

- **写回控制信号生成:**产生寄存器写使能 RF_wen 和目标寄存器地址 RF_waddr (即 rd 字段)。

```
assign    RF_wen          = (is_I || is_R || is_J || is_U) && (rd != 0);
```

- **下一 PC 地址计算:**根据当前指令类型(分支、JAL、JALR)和分支条件判断结果 (branch_condition_satisfied) 计算下一条指令的 PC 地址 (next_pc)。分支条件 (equal, ltu, lts) 在 IDU 内部根据输入的 RF_rdata1 和 RF_rdata2 直接进行比较。

```
// -- PC Update Logic --
assign    next_pc    = (is_J || is_jalr) ? jump_target :   // Use jump target for JAL/JALR
                       (is_B && branch_condition_satisfied) ? branch_target : // Use branch
                           target if condition is satisfied
                       current_pc + 4;                     // Default: PC + 4 for normal
                           instruction flow
```

- **寄存器堆读地址输出:**输出解码得到的 rs1 和 rs2 字段作为寄存器堆的读地址 (raddr1, raddr2)。

```
// assign raddr signals for register file access
assign raddr1 = rs1; // Register file read address 1
assign raddr2 = rs2; // Register file read address 2
```

- **冒险检测信号输出:**输出 ID_rs1 和 ID_rs2 (即解码的 rs1, rs2 地址) 给前递单元和控制单元用于数据冒险检测。

```
// assign ID_rs signals
assign ID_rs1 = rs1; // Source register 1
assign ID_rs2 = rs2; // Source register 2
```

IDU 的所有输出信号(除直接送往寄存器堆和 IFU 的信号外)都会被锁存到 ID/EX 流水线寄存器。

1.2.3 执行单元 (EXU - Execute Unit)

EXU 模块负责执行算术逻辑运算和移位运算。它接收来自 ID/EX 流水线寄存器的操作数 (alu_src1, alu_src2, shifter_src1, shifter_src2) 和操作码 (alu_op, shifter_op)。EXU 内部实例化了 alu 和 shifter 两个子模块。EXU 会驱动相应的运算单元并输出其计算结果 (alu_result, shift_result), 这些结果将被送往 EX/MEM 流水线寄存器。对于 Load/Store 指令, ALU 在此阶段计算有效内存地址, 该地址也作为 alu_result 输出。

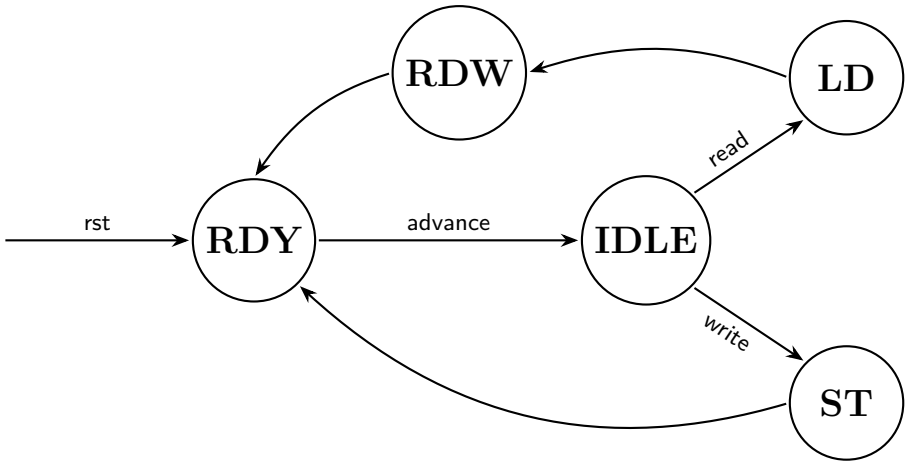
```
alu instance_alu (
    .A        (alu_src1),      // Input A from Src Sel
    .B        (alu_src2),      // Input B from Src Sel
    .ALUop    (alu_op),        // Input Opcode from Op Gen
    .Overflow (),
    .CarryOut (),
    .Zero     (),              // Output: Zero flag -> To EX (PC Ctrl), WB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

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   (shift_result)       // Output: Shifter result -> To WB
);
```

1.2.4 访存单元 (MEMU - Memory Access Unit)

MEMU 模块负责处理 Load 和 Store 指令的内存访问操作。它接收来自 EX/MEM 流水线寄存器的内存地址 (alu_result)、待存储数据 (RF_rdata2, 在顶层对应 em_store_data_value_reg) 以及访存控制信号 (MemRead_bus_in, MemWrite_bus_in, mem_width, mem_signed)。

MEMU 内部包含一个 FSM 来管理与数据存储器的握手(Mem_Req_Ready, Read_data_Valid, Read_data_Ready)和多周期访存。

FSM 的状态转移简图如下:



- 对于 Load 指令 (MemRead_bus_in=1),MEMU 向内存发出读请求,并在接收到有效数据 (Read_data) 后,进行必要的字节选择和符号扩展(根据 mem_width 和 mem_signed),然后将最终结果 (RF_wdata 输出) 送往 MEM/WB 流水线寄存器。

```
wire [31:0] _load_data; // Data loaded from memory, to be passed to WB stage
reg [31:0] load_data; // Register to hold the load data
assign    _load_data       = (current_state == RDW && Read_data_Valid) ? // Process
    Read_data -> To WB Stage
                           ( (width_LB || width_LBU) ? // LB/LBU
                             (addr_0 ? {{24{(mem_signed & Read_data [ 7])}}, Read_data [
                                 7: 0]} :
                              addr_1 ? {{24{(mem_signed & Read_data [15])}}, Read_data
                                  [15: 8]} :
                              addr_2 ? {{24{(mem_signed & Read_data [23])}}, Read_data
                                  [23:16]} :
                                  {{24{(mem_signed & Read_data [31])}}, Read_data
                                      [31:24]} )

                           : (width_LH || width_LHU) ? // LH/LHU
                             (addr_0 ? {{16{(mem_signed & Read_data [15])}}, Read_data
                                 [15: 0]} :
                              addr_2 ? {{16{(mem_signed & Read_data [31])}}, Read_data
                                  [31:16]} :
                                  32'b0 )

                           : (width_LW) ? Read_data // LW

                           : 32'b0 ) // Default unknown load
                           : 32'b0; // Default if not a load

always @(posedge clk) begin
  if (Read_data_Valid) begin
    load_data <= _load_data; // only update load_data when Read_data is valid
  end else begin
    load_data <= load_data;
  end
```

```
end
```

- 对于 Store 指令 (MemWrite_bus_in=1)，MEMU 将地址、待写数据 (Write_data 输出，源自输入的 RF_rdata2 所对应的流水线传递值) 和字节选通 (Write_strb) 发送给内存。

```
assign    Write_data = {32{MemWrite}} & (
                        (({32{width_SB}} & {4{RF_rdata2[7:0]}}) | // SB
                        ({32{width_SH}} & {2{RF_rdata2[15:0]}}) | // SH
                        ({32{width_SW}} & RF_rdata2)        // SW
                       ));
```

- 对于非访存指令，从 EX 阶段传来的 ALU 结果 (alu_result) 会直接作为 RF_wdata 透传到 MEM/WB 流水线寄存器。

```
assign RF_wdata = MemRead_bus_in ? load_data : alu_result;
```

MEMU 输出 mem_ready 信号，当其内部 FSM 处于空闲 (RDY) 状态时为高，表示 MEM 阶段已准备好处理下一条指令。

1.2.5 写回单元 (WBU - Write Back Unit) 与寄存器堆

WB 阶段是流水线的最后阶段。在本设计中，实际的寄存器写回操作由顶层逻辑根据 MEM/WB 流水线寄存器 (mw_..._reg) 中的信号直接驱动独立的 reg_file 模块的写端口。这些信号包括写使能 (wb_rf_wen_internal)、写地址 (mw_rf_waddr_reg) 和写数据 (mw_data_to_wb_reg)。

WBU 模块 (wbu_inst) 的主要职责是接收来自 MEM/WB 寄存器的最终指令信息（包括 PC、是否写回、写回地址和数据），并生成用于仿真验证和调试的 inst_retire 信号。

```
   assign wb_rf_wen_internal = mw_rf_wen_reg && mw_inst_valid_reg && pipeline_advance_enable;

   reg_file reg_file_inst (
       .clk   (clk),
       .wen   (wb_rf_wen_internal),
       .waddr (wb_rf_waddr_internal),
       .wdata (wb_rf_wdata_internal),
       .raddr1 (idu_regfile_raddr1_out), // From IDU
       .raddr2 (idu_regfile_raddr2_out), // From IDU
       .rdata1 (rf_read_data1_raw_internal), // To FWDU
       .rdata2 (rf_read_data2_raw_internal) // To FWDU
   );

module wbu (
  input       inst_valid,
  input [31:0] pc,
  input       RF_wen,
  input [ 4:0] RF_waddr,
  input [31:0] RF_wdata,

  output [69:0] inst_retire
);
```

```
  assign inst_retire = {70{inst_valid}} & {RF_wen, RF_waddr, RF_wdata, pc};

endmodule
```

1.2.6 前递单元 (FWDU - Forwarding Unit)

FWDU 负责解决数据冒险 (RAW Hazard)。它监测 ID 阶段指令的源寄存器地址 (`ID_rs1`, `ID_rs2`) 是否与 EX、MEM、WB 阶段指令的目标寄存器地址 (`EX_rd`, `MEM_rd`, `WB_rd`) 发生冲突。如果检测到冲突，并且后续阶段的指令会写回该寄存器，FWDU 会选择相应阶段的计算结果 (`EX_alu_result`, `MEM_alu_result` 或 `MEM_RF_wdata`, `WB_RF_wdata`) 而不是从寄存器堆读取的旧值，作为 IDU 的输入操作数 (`fwdu_src1`, `fwdu_src2`)。前递的优先级是 EX → ID 高于 MEM → ID 高于 WB → ID，以确保总是获取最新的数据。FWDU 的判断不考虑目标寄存器为 x0 的情况。

```
  /* conditions of different bypass */
  wire ex_bp1, ex_bp2, mem_bp1, mem_bp2, wb_bp1, wb_bp2;

  /* the "0" register doesn't matter */
  assign ex_bp1 = (ID_rs1 == EX_rd) && ( |EX_rd[4:0]);
  assign ex_bp2 = (ID_rs2 == EX_rd) && ( |EX_rd[4:0]);
  assign mem_bp1 = (ID_rs1 == MEM_rd) && (|MEM_rd[4:0]);
  assign mem_bp2 = (ID_rs2 == MEM_rd) && (|MEM_rd[4:0]);
  assign wb_bp1 = (ID_rs1 == WB_rd) && ( |WB_rd[4:0]);
  assign wb_bp2 = (ID_rs2 == WB_rd) && ( |WB_rd[4:0]);


  /* EX bypass is prior to MEM bypass, so as MEM to WB */

  assign fwdu_src1 = ex_bp1 ? EX_alu_result :
              mem_bp1 ? MEM_alu_result :
              wb_bp1 ? WB_RF_wdata  :
              RF_rdata1;

  assign fwdu_src2 = ex_bp2 ? EX_alu_result :
              mem_bp2 ? MEM_alu_result :
              wb_bp2 ? WB_RF_wdata  :
              RF_rdata2;
```

1.2.7 控制逻辑与冒险处理

流水线的控制逻辑主要由顶层模块的 `pipeline_advance_enable` 和 `ld_use_hazard_stall` 信号以及各阶段模块内部的 FSM 协同完成。

- **`ld_use_hazard_stall` (Load-Use 冒险暂停):** 该信号在顶层模块中通过组合逻辑产生。它检测当前 ID 阶段的指令的源操作数是否依赖于当前 EX 阶段或 MEM 阶段的 Load 指令的结果。当 `ld_use_hazard_stall` 有效时：

  1. ID/EX 流水线寄存器会优先被强制插入一个“气泡”(无效指令，控制信号置为 NOP 状态)。这个气泡在下一周期进入 EX 阶段，其如 `de_mem_read_reg` 等关键控制位为 0，从而使得 `ld_use_hazard_stall`

信号的计算条件在下一周期不再满足,冒险得以解除。

2. IFU 模块接收此信号,其内部 FSM 会暂停 PC 更新并停留在等待状态(如 STA 态或使其无法进入 RDY 后的 STA 态),直到冒险解除。
3. 全局的 `pipeline_advance_enable` 信号也会因为 `ld_use_hazard_stall`(通过影响 IFU 的 `if_stage_ready`)而可能变为低,从而暂停流水线上游(IF/ID)的正常推进。

- **`pipeline_advance_enable`(全局流水线推进使能):** 该信号由 `if_stage_ready`(来自 IFU)和 `mem_stage_ready`(来自 MEMU)相与产生。当它为高时,表示主要的、可能产生结构性暂停的 IF 和 MEM 阶段都准备好了,流水线寄存器可以从上一级加载新的数据。当它为低时,相应的流水线寄存器会保持其当前值,实现流水线的暂停。Load-Use 冒险会优先通过 ID/EX 气泡插入来处理,并间接通过影响 IFU 状态来影响 `if_stage_ready`,进而影响 `pipeline_advance_enable`。

2 实验过程中遇到的问题、对问题的思考过程及解决方法

本次流水线处理器的设计与调试过程充满了挑战,但也收获颇丰。从最初的基础功能验证到通过所有测试样例,期间遇到了诸多与流水线控制、数据冒险、控制冒险以及模块接口交互相关的复杂问题。以下将对主要遇到的问题及其分析和解决过程进行阐述:

1. **问题:早期接口与逻辑错误**

   - **现象:** IFU 模块的 `RF_waddr` 输出信号位宽与预期不符,导致功能异常。同时,在初步集成时,`bne` 指令无法正常工作,表现为 PC 跳转错误,如错误日志所示:

   Listing 1: bne 指令执行错误日志 - 0ns

   ```
   ================================================================
   ERROR: at                00ns.
   Yours:    PC = 0x00000024, rf_waddr = 0x0a, rf_wdata = 0x00000000
   Reference: PC = 0x0000001c, rf_waddr = 0x0a, rf_wdata = 0x00000008
   ================================================================
   ```

   - **思考与解决:** 仔细检查后发现,`idu` 模块中,将解码出的 `rs1` 和 `rs2` 字段赋值给输出端口 `ID_rs1` 和 `ID_rs2`(用于前递单元和控制单元)的逻辑缺失。同时,将 `rs1` 和 `rs2` 字段连接到寄存器堆读地址端口 `raddr1` 和 `raddr2` 的 `assign` 语句也遗漏了。在旁路单元 `fwdu` 中,也存在将内部计算的 `data1/data2` 错误地连接到同名内部信号而非模块输出端口 `fwdu_src1/fwdu_src2` 的问题。补全这些缺失的连接和修正端口名后,`bne` 指令行为恢复正常。

2. **问题:Load 指令 (`lw`) 实现错误,读取地址而非数据**

   - **现象:** CPU 在执行 `lw` 指令时,错误地将计算出的内存地址(例如 `0xc8`)作为数据写回目标寄存器,而不是从该地址读取的内存内容。错误日志如下:

   Listing 2: lw 指令错误日志 - 10ns (错误 1)

   ```
   ================================================================
   ERROR: at                10ns.
   Yours:    PC = 0x00000028, rf_waddr = 0x0c, rf_wdata = 0x000000c8
   Reference: PC = 0x00000028, rf_waddr = 0x0c, rf_wdata = 0x00000000
   ================================================================
   ```

- **思考与解决：** 问题在于 idu 模块未能正确产生 mem_read_internal 控制信号。该信号本应在解码到 Load 指令时置高，以启动后续的内存读取操作。由于遗漏了 assign mem_read_internal = is_load; 这条语句，导致 Load 指令在 MEMU 阶段没有被识别为读操作，最终使得 MEMU 将传入的地址(alu_result)作为了写回数据。补全该 assign 语句后解决。

3. **问题：Load-Use Hazard 导致的流水线死锁**

- **现象：**当 Load 指令后紧跟一条依赖其结果的指令时，CPU 发生死锁。具体表现为：ld_use_hazard_stall 信号有效，IFU 的 FSM 卡在 STA 状态导致 IF_Ready 为低，进而 pipeline_advance_enable 为低，使得 ID/EX 寄存器无法更新并插入用以解除冒险的气泡。
- **思考：**死锁的关键在于气泡插入逻辑依赖于 pipeline_advance_enable，而 pipeline_advance_enable 又因为冒险导致的 IFU 暂停而无法变高。必须解耦气泡插入与全局推进使能的直接依赖。
- **解决：**修改了 IFU 的状态机，当在 STA 状态检测到 ld_use_harzard 时，使其跳转回 RDY 状态。更重要的是，调整了 ID/EX 流水线寄存器的更新逻辑，使其在检测到 ld_use_hazard_stall 时，无论 pipeline_advance_enable 状态如何，都优先强制载入气泡(特别是将 de_mem_read_reg 等关键控制位置为 0)。同时，ld_use_hazard_stall 通过影响 IFU 的行为间接影响 if_stage_ready，从而控制 pipeline_advance_enable 来暂停上游。

4. **问题：Store 指令 (sw) 后紧跟 Load 指令 (lw) 时，MEMU FSM 状态转换错误**

- **现象：** 当 sw 指令的内存写操作完成，MEMU 的 FSM 从 ST 状态转换到 RDY 状态后，紧接着 EX/MEM 寄存器中是 lw 指令的信息(MemRead_bus_in=1)。但 FSM 在 RDY 状态判断下一状态时，错误地再次识别为 sw 指令(可能因为 MemWrite_bus_in 信号相对于 FSM 状态更新存在延迟或采样问题)，导致状态机进入错误的 ST 状态，而不是预期的 LD 状态。
- **思考：** FSM 在 RDY 状态决定下一跳时，必须基于当前（即新指令对应的）MemRead_bus_in 和 MemWrite_bus_in。同时，pipeline_advance_enable 信号需要被正确地纳入 FSM 从 RDY 状态转换的条件中。
- **解决：** 新增了一个 IDLE 状态来处理这种情况。修改后逻辑为：当处于 RDY 状态时，首先根据 pipeline_advance_enable 信号到达 IDLE 状态，在这个新状态下，直接根据当前输入的 MemRead_bus_in 和 MemWrite_bus_in 来决定下一个状态是 LD、ST 还是跳转到 RDY。全局的 pipeline_advance_enable 仍然控制流水线寄存器是否更新。

5. **问题：Store 指令 (sw) 写入内存的数据错误**

- **现象：** sw 指令执行后，后续的 lw 指令从相应地址读出的数据与预期不符。例如，在 load-store 测试的早期，lw 期望读出 4 但实际读出 0。

Listing 3: sw 后 lw 数据错误日志 - 10ns (load-store)

```
================================================================
ERROR: at                10ns.
Yours:    PC = 0x00000028, rf_waddr = 0x0c, rf_wdata = 0x00000000
Reference: PC = 0x00000028, rf_waddr = 0x0c, rf_wdata = 0x00000004
================================================================
```

- **思考与解决：** 一方面，发现 idu 模块中用于 Store 指令的源数据信号 Write_data（即 RF_rdata2 的值）在某些情况下未能正确赋值或传递。另一方面，在 memu 模块中，用于字节选择和符号扩展的 load_data 信号的位宽未显式定义。通过确保 idu 正确输出 Store 数据，并在 memu 中为 load_data 显式定义位宽 wire [31:0] _load_data; 及后续的寄存器 load_data 后，问题得到改善。根本原

因在于追踪 sw 指令的数据源，确保其在 IDU 阶段准备的 RF_rdata2 值是正确的，并且 MEMU 在写内存时使用了这个正确的值。

6. **问题：IFU 的取指请求长时间得不到内存响应，导致流水线暂停**

   - **现象：** IFU 发出 Inst_Req_Valid 后，外部存储器接口的 Inst_Req_Ready 持续为低，导致 IFU 卡在 IF 状态。
   - **思考：**通过分析仲裁器 cpu_to_mem_axi_2x1_arb.v 的行为，发现其赋予了数据内存请求 (cpu_mem_arvalid) 比指令获取请求 (cpu_inst_arvalid) 更高的优先级。
   - **解决：**进一步追查发现，确实是内存响应规则造成的问题。助教老师为我更改了 Testbench 的仲裁器响应逻辑，为指令获取的 cpu_inst_arready 增加了条件 && !cpu_mem_arvalid，确保在数据通路请求也被仲裁器锁存但尚未被 AXI RAM 响应时，指令通路仍能得到服务，从而解决了指令获取被饿死的问题。

7. **问题：Store 指令被错误地“吃掉”或流水线寄存器内容未及时更新**

   - **现象：**在 load-store 测试中，某条 Store 指令似乎没有按预期执行，导致后续 Load 指令读取到旧数据。如日志：

Listing 4: Store 指令执行问题日志 - 40ns (load-store)

```
================================================================
ERROR: at                40ns.
Yours:    PC = 0x0000003c, rf_waddr = 0x0c, rf_wdata = 0x00000000
Reference: PC = 0x0000003c, rf_waddr = 0x0c, rf_wdata = 0x00000004
================================================================
```

   - **思考：** 检查 IF/ID 流水线寄存器 (fd_..._reg) 的更新逻辑。发现在 ld_use_hazard_stall 为高时，我使用了用 NOP 覆盖 fd_instruction_reg 的语句。这可能导致一个有效的指令(如 Store)在 IF/ID 阶段因为前序 Load-Use 冒险而被覆盖为 NOP，从而被“吃掉”。
   - **解决：**注释掉了在 ld_use_hazard_stall 时，用 NOP 覆盖掉 fd_inst_valid_reg 的逻辑，确保流水线各级寄存器在 pipeline_advance_enable 控制下能正确传递或保持指令信息。

8. **问题：Load-Use 冒险暂停时间不足**

   - **现象：** 即使有 ld_use_hazard_stall，lw 指令后的分支指令仍然在内存读取完成前使用了旧数据，导致分支错误。
   - **思考：**原先的 ld_use_hazard_stall 仅考虑了 ID 阶段对 EX 阶段 Load 指令的依赖。这不足以覆盖 Load 指令在 MEM 阶段的实际内存访问延迟。
   - **解决：**扩展了 ld_use_hazard_stall 的判断逻辑，使其不仅检查 ID 对 EX 的 Load 依赖，还检查 ID 对 MEM 阶段(且该 Load 指令确实在进行内存读操作)的 Load 指令的依赖，如下所示。这确保了只要 ID 阶段依赖的 Load 指令的数据尚未从内存中准备好，流水线就会保持暂停。

Listing 5: 修正后的 ld_use_hazard_stall 逻辑 (扩展到 MEM 阶段)

```
assign ld_use_hazard_stall =
   (de_inst_valid_reg && de_mem_read_reg && de_rf_wen_reg) &&
   ( (de_rf_waddr_reg == idu_rs1_for_hazard_out && idu_rs1_for_hazard_out != 0) ||
     (de_rf_waddr_reg == idu_rs2_for_hazard_out && idu_rs2_for_hazard_out != 0) )
 ||
```

```
    (em_inst_valid_reg && em_mem_read_reg && em_rf_wen_reg) &&
    ( (em_rf_waddr_reg == idu_rs1_for_hazard_out && idu_rs1_for_hazard_out != 0) ||
      (em_rf_waddr_reg == idu_rs2_for_hazard_out && idu_rs2_for_hazard_out != 0) );
```

9. **问题:JAL/JALR 指令写回 ra 寄存器的值为跳转目标地址而非 PC+4**

- **现象:** 在 recursion 测试中, jalr 和 jal 指令执行后, ra 被写入了跳转目标地址而不是链接地址 PC+4。

Listing 6: JAL/JALR 写回错误日志 (recursion)

```
================================================================
ERROR: at                 00ns.
Yours:    PC = 0x00000230, rf_waddr = 0x01, rf_wdata = 0x00000018
Reference: PC = 0x00000230, rf_waddr = 0x01, rf_wdata = 0x00000234
================================================================
```

- **思考:** JAL/JALR 指令需要将 PC+4 作为链接地址存入目标寄存器。问题在于 ALU 被错误地配置为计算跳转目标,并且这个跳转目标被用作了写回数据。
- **解决:** 修改了 IDU 中为 JAL/JALR 指令选择 ALU 操作数 alu_src2 的逻辑。确保对于 JAL 和 JALR, alu_src2 被设置为 4, 使得 ALU 计算 current_pc + 4 作为链接地址。而实际的跳转目标地址则由 IDU 的 next_pc 逻辑独立计算并送往 IFU。

Listing 7: IDU 中修正 JAL/JALR 的 alu_src2 逻辑

```
assign alu_src2 = (is_I)       ? imm        :
                  (is_J || is_jalr) ? 32'd4    : // For JAL/JALR, ALU calculates PC+4
                  (is_R)       ? RF_rdata2  :
                  // ... other types ...
                                  32'b0;
```

同时, 确保 JALR 指令的 alu_src1 优先使用 current_pc 进行链接地址计算, 而不是用于计算 rs1+imm 。跳转目标的计算则依赖于 RF_rdata1 和 imm_I 在 next_pc 逻辑中组合。

10. **问题:Load Half-word (lh) 指令符号扩展错误**

- **现象:** 在 load-store 测试中, lh 指令从内存读取一个负半字 (如 0x8000) 时, CPU 将其零扩展为 0x00008000 而非符号扩展为 0xFFFF8000。

Listing 8: lh 符号扩展错误日志 (load-store - 2)

```
================================================================
ERROR: at                 10ns.
Yours:    PC = 0x00000078, rf_waddr = 0x0a, rf_wdata = 0x00008000
Reference: PC = 0x00000078, rf_waddr = 0x0a, rf_wdata = 0xffff8000
================================================================
```

- **思考:** 问题在于 IDU 是否正确生成了 mem_signed 控制信号, 以及 MEMU 是否正确使用了该信号进行符号扩展。
- **解决:** 确认了 IDU 中根据 funct3 产生 mem_signed 的逻辑本身是正确的。最终发现是在 IDU 的输出端口列表中遗漏了对 mem_signed 信号的实际赋值连接, 导致该控制信号未能正确传递到后续流水线阶段。补全该赋值后问题解决。

整个调试过程充满了反复的分析、定位、修改和验证，深刻体会到流水线设计的复杂性和细节的重要性。通过结合错误日志、反汇编代码和波形图，逐层追溯信号流和控制逻辑，最终才得以解决这些盘根错节的问题。

3 实验结果

3.1 仿真结果与波形图分析

图 1: 寄存器堆写使能信号的传递

图 1 展现了在五级流水线设计中，寄存器堆写使能信号在流水线寄存器之间的传递过程。从图中可以清晰的看出流水线“流动”的过程。

3.2 UART 控制器打印结果

以下为 Hello 程序的执行结果，说明 UART 控制器功能正常。

```
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
00000000100000000200000000300000000400000005
50 50 -50 4294967246
time 10200.11ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/hello/riscv32/elf/hello passed
Hit good trap
```

3.3 性能计数器统计结果分析

Listing 9: RISC-V 统计结果

```
--- Performance Counters for [bf] ---
  Cycles:                38814005
  Retired Instructions:  404967
  Retired Loads:         94513
  Retired Stores:        5981
  Branches Executed:     91039
  Branches Taken:        36085
  IF Stalls:             0
  MEM Access Stalls:     112450
  IW Stalls:             29897078
  RDW Stalls:            6443174
  Jumps Executed:        47906
  ALU Ops Executed:      135350
  Shift Ops Executed:    78063
  NOPs in ID:            0
  Total MEM Ops Issued:  100490
  Register Writes:       307959
--- End of Counters for [bf] ---
benchmark finished
time 741.36ms
```

Listing 10: RISC-V 流水线处理器统计结果

```
--- Performance Counters for [bf] ---
  Cycles:                38802291
  Retired Instructions:  452870
  Retired Loads:         94514
  Retired Stores:        5981
  Branches Executed:     91039
  Branches Taken:        36085
  IF Stalls:             0
  MEM Access Stalls:     112450
  IW Stalls:             29927844
  RDW Stalls:            6447060
  Jumps Executed:        47906
  ALU Ops Executed:      374812
  Shift Ops Executed:    78063
  NOPs in ID:            0
  Total MEM Ops Issued:  100491
  Register Writes:       307958
--- End of Counters for [bf] ---
benchmark finished
time 739.97ms
```

由于我在 RISC-V 多周期处理器和 RISC-V 流水线处理器中统计这些量的逻辑不甚相同，因此二者的数据存在细微的差异。忽略掉这些细微的差异，我们可以看出，仅仅在流水线存在的情况下，对 CPU 性能的提升十分有限。事实上，我们可以看出，流水线暂停 (无论是取指令过程中的暂停还是访存过程中的暂停) 占了绝大部分的时钟周期，并且，在上述测试样例中，流水线处理器的由指令等待造成的暂停时间明显比多周期处理器要长。因此，在这个测试样例中，流水线对性能提升有限是合理的。由此，我们发现，`IW_stalls` 的轻微波动就会把我们流水线处理器带来的性能优势一把抹去。事实上，在后续实现了 cache 之后，我发现，icache 对性能的提升是最大的。

在下面这个测试样例中，可以看出，流水线在某些测试程序上面，性能提升能达到将近 20%:

Listing 11: RISC-V 统计结果

```
--- Performance Counters for [queen] ---
  Cycles:                6951385
  Retired Instructions:  79447
  Retired Loads:         14420
  Retired Stores:        12362
  Branches Executed:     6078
  Branches Taken:        1471
  IF Stalls:             0
  MEM Access Stalls:     51500
  IW Stalls:             5487631
  RDW Stalls:            982974
  Jumps Executed:        4118
  ALU Ops Executed:      38354
  Shift Ops Executed:    4112
  NOPs in ID:            0
  Total MEM Ops Issued:  26778
  Register Writes:       61019
 --- End of Counters for [queen] ---
benchmark finished
time 137.76ms
```

Listing 12: RISC-V 流水线处理器统计结果

```
--- Performance Counters for [queen] ---
  Cycles:                5798766
  Retired Instructions:  81506
  Retired Loads:         14421
  Retired Stores:        12362
  Branches Executed:     6078
  Branches Taken:        1471
  IF Stalls:             0
  MEM Access Stalls:     51500
  IW Stalls:             5409133
  RDW Stalls:            1000347
  Jumps Executed:        4118
  ALU Ops Executed:      77399
  Shift Ops Executed:    4112
  NOPs in ID:            0
  Total MEM Ops Issued:  26779
  Register Writes:       61018
 --- End of Counters for [queen] ---
benchmark finished
time 122.66ms
```

4 对讲义中思考题(如有)的理解和回答

在本实验对应的讲义中未明确指定思考题。但通过本次流水线处理器的设计与实现,引发了我对以下几方面的深入思考:

1. **流水线暂停的代价与收益:**本设计采用在 ID 阶段解析分支并在检测到 Load-Use 冒险时暂停流水线的策略。这种方式避免了复杂的分支预测器和预测失败时的冲刷开销,简化了控制逻辑。但代价是,当 Load-Use 冒险发生或内存访问存在延迟时,流水线会产生实际的暂停周期。对于访存密集型或数据依赖紧密的程序,这种暂停可能比偶发的分支预测失败惩罚更为显著。高性能处理器通常会采用更激进的动态分支预测和乱序执行等技术来掩盖这些延迟,但其设计复杂度也大大增加。

2. **控制信号的产生与传递时机:**在流水线设计中,控制信号(如写使能、操作码、暂停信号)必须在正确的时间点产生,并准确地随数据在流水线寄存器中传递。例如,本次调试中遇到的 `RF_wen` 信号因流水线暂停而持续有效导致 testbench 匹配错误(只要写使能有效一周期,Testbench 就会往下匹配一行,导致一条指令的写回匹配了两行 golden trace)的问题,以及 `next_pc` 的计算与 PC 更新时机不匹配的问题,都凸显了对流水线各阶段信号同步和时序关系的精确把握至关重要。

3. **模块化设计与接口定义:**将 CPU 划分为 IFU, IDU, EXU, MEMU 等模块有助于降低设计复杂度。但模块间接口的清晰定义和正确连接是成功的关键。如此次实验中,`mem_width` 信号的编码约定、`idu` 模块操作数输入的来源(寄存器堆还是旁路单元的输出)、以及 Load 指令结果的符号扩展责任划分等,都需要在接口层面明确。

4. **验证的挑战:**即使是相对简单的五级流水线,其可能的状态和交互也异常复杂。基于 trace 文件的比对是一种有效的验证方法,但它依赖于"黄金模型"trace 的正确性。例如,如果希望在本地测试 microbench 测试程序集,由于 trace 文件过大,一开始并没有提供,就需要自己使用一个正确的(比如说多周期)CPU 来生

成金标准 trace，才能进行 difftest。设计可调试性（如提供足够的内部状态观测点、性能计数器）也至关重要。事实上，在调试流水线 RISC-V 处理器的过程中，遇到的许多问题都是“可调试性”非常差的。例如，在首次运行程序时基本都会都会遇到的程序卡死的问题，以及 AXI 内存行为不符合预期的现象等。此时就需要我们使用一些另外的手段来调试程序。例如，通过 Verilator 将 Verilog 代码编译成 C 程序这一点特性，我们可以从 C 程序的角度来控制处理器运行的暂停；同时，我们也可以通过阅读 AXI 内存的 Verilog 源代码来了解其内部各个信号的含义，进而从波形图中抓取内存中的波形图来辅助调试。

5 实验所耗时间

在课后，我花费了大约 80 小时完成此次流水线处理器的设计、实现、调试及报告撰写工作。其中，大部分时间消耗在流水线控制逻辑的调试，特别是数据冒险、控制冒险以及与外部存储器接口交互时产生的暂停和时序问题。

6 实验总结与心得体会

在本次《计算机组成原理》的实验五中，我成功地设计并实现了一个五级流水线的 RISC-V 功能型处理器。这个过程不仅是对我 Verilog HDL 编程和硬件设计能力的综合检验，更是一次对流水线核心概念、数据冒险与控制冒险处理机制、以及模块化设计思想的深度实践。

与之前设计的多周期处理器相比，流水线处理器的设计引入了显著的复杂性，主要体现在需要在各级流水线之间精确传递数据和控制信号，并有效处理因此产生的各种冒险。本次设计中，我将 CPU 划分为取指 (IFU)、译码 (IDU)、执行 (EXU)、访存 (MEMU) 和写回 (WBU) 五个主要模块，并通过 IF/ID、ID/EX、EX/MEM、MEM/WB 四级流水线寄存器进行连接。

在数据冒险处理方面，我实现了一个前递单元 (FWDU)，能够将 EX、MEM、WB 阶段的结果提前反馈给 ID 阶段需要这些数据的指令，从而避免了大多数情况下的流水线暂停。针对无法通过前递完全解决的 Load-Use 冒险，我设计了 `ld_use_hazard_stall` 检测逻辑。该逻辑在检测到 ID 阶段指令依赖于 EX 或 MEM 阶段的 Load 指令结果时被触发。其核心作用是：优先使 ID/EX 流水线寄存器强制插入一个“气泡”（无效指令，并将关键控制信号如 `de_mem_read_reg` 置零），这个动作独立于全局流水线是否因其他原因暂停。这个气泡在下一周期进入 EX 阶段，从而使得 `ld_use_hazard_stall` 的计算条件不再满足，冒险得以解除。同时，`ld_use_hazard_stall` 信号也会通知 IFU 暂停取指和 PC 更新，并通过影响最终的 `pipeline_advance_enable` 信号来暂停流水线上游的正常推进。

在控制冒险处理方面，本设计在 ID 阶段即完成分支条件的判断和跳转目标地址的计算，并将下一 PC 地址直接反馈给 IFU。这种早期解析的策略，配合由 Load-Use 冒险或存储器延迟（通过 MEMU 的 `mem_stage_ready` 信号反映）引起的全局流水线暂停机制 (`pipeline_advance_enable` 由 `if_stage_ready` 和 `mem_stage_ready` 共同决定)，旨在简化分支预测和冲刷逻辑。IFU 和 MEMU 内部也设计了状态机来处理与外部存储器的异步握手和潜在的多周期操作。

调试过程是本次实验最具挑战性的部分。从最初的 PC 更新错误（如 JAL/JALR 链接地址写回错误）、Load 指令符号扩展问题，到由于流水线暂停控制不当导致的死锁（例如，`ld_use_hazard_stall` 有效时，气泡无法插入 ID/EX 寄存器）、错误的内存读写（例如，SW 指令后紧跟 LW，MEMU 状态转换错误或使用了旧的控制信号）、以及由于仲裁器行为导致指令获取请求长时间得不到服务等。每一个问题的解决都依赖于对波形图的细致分析、对流水线时序、数据流和控制流交互的反复推敲。例如，确保在 Load-Use 冒险发生时，气泡能够被可靠地注入 ID/EX 级以打破死锁循环，以及解决由于 MEMU 的 `load_data` 信号在错误时机被清零导致 Load 结果错误的问题，都是调试过程中的关键突破。针对 testbench 报告的同一条指令写回两次的问题，通过引入 `wb_protector` 机制，确保了写回操作的单周期有效性。

此外，本次实验还实现了 16 个性能计数器。这些计数器的设计与集成过程，让我对衡量和评估处理器性能的指标有了初步认识。

总而言之，通过本次流水线处理器的设计与实现，我不仅巩固了计算机组成原理的理论知识，更在实践中深刻体会到了流水线设计的精髓与挑战。从数据通路的构建、控制信号的生成与传递，到冒险的检测与处理，每一个环节都需要严谨的逻辑思维和细致的时序考量。这段经历极大地提升了我的硬件设计和调试能力，也为我未来学习更高级的处理器架构和并行计算技术打下了坚实的基础。

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，您们的帮助对我有非常大的指导意义。