浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：张天逸

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3220106424

指导教师：姜晓红

2024 年　10 月　12　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：Pipelined CPU with Cache

学生姓名：张天逸 专业：计算机科学与技术 学号：3220106424

同组学生姓名： 指导老师：姜晓红

实验地点： 实验日期：2024 年 10 月 12 日

一、 实验目的和要求

理解 Cache 中 Tag, Valid, Dirty 的作用和 Least Recently Used 的概念

理解 Cache Management Unit 的作用及与 CPU 结合的方法

理解 CMU 的各种 State，以及处理 Hit，Miss 等状况的方法

掌握验证 Cache 和 CMU 设计正确性的方法

二、 实验内容和原理

CMU 状态转换过程

cmu 模块具有以下状态：S_IDLE ：空闲状态;S_BACK ：将脏缓存行写回内存;S_FILL ：从内存加载数据到缓存;S_WAIT ：等待状态，表示内存事务结束。

S_IDLE 状态将 word_count 复位为 0,条件为：如果 en_r（读使能）或 en_w（写使能）被置为高电平，缓存控制器会检查请求的数据是否在缓存中（cache_hit）。如果数据在缓存中（cache_hit = 1），next_state 保持为 S_IDLE。如果缓存未命中（cache_hit = 0），控制器会检查当前缓存行是否有效（cache_valid = 1）且是脏的（cache_dirty = 1）。如果是，则进入 S_BACK 状态，将脏缓存行写回内存。如果缓存行无效或者不是脏的，则进入 S_FILL 状态，从内存中加载数据。

S_BACK 状态将脏缓存行写回内存。该状态会保持在 S_BACK，直到收到内存的确认信号（mem_ack_i），表示当前字已经成功写入内存。当接收到 mem_ack_i 时，增加 word_count。当所有缓存行中的字都写回后（word_count 达到最大值），

转到 S_FILL 状态。

S_FILL 状态将内存中的数据填充到缓存中。该状态会保持在 S_FILL，直到所有字从内存加载到缓存。当 mem_ack_i 被置为高电平时，增加 word_count。当 word_count 达到最大值时，转到 S_WAIT 状态。

S_WAIT 状态用于确保所有操作完成后再返回 S_IDLE。将 word_count 复位为 0 然后转到 S_IDLE 状态。

CMU.V

```
module cmu (
        // CPU side
        input clk,
        input rst,
        input [31:0] addr_rw,
        input en_r,
        input en_w,
        input [2:0] u_b_h_w,
        input [31:0] data_w,
        output [31:0] data_r,
        output stall,

        // mem side
        output reg mem_cs_o = 0,
        output reg mem_we_o = 0,
        output reg [31:0] mem_addr_o = 0,
        input [31:0] mem_data_i,
        output [31:0] mem_data_o,
        input mem_ack_i,
```

```
        // debug info
        output [2:0] cmu_state
    );
```

```
    `include "addr_define.vh"
    reg [ADDR_BITS-1:0] cache_addr = 0;
    reg cache_load = 0;
    reg cache_replace = 0;
    reg cache_store = 0;
```

```
reg [2:0] cache_u_b_h_w = 0;
reg [WORD_BITS-1:0] cache_din = 0;
wire cache_hit;
wire [WORD_BITS-1:0] cache_dout;
wire cache_valid;
wire cache_dirty;
wire [TAG_BITS-1:0] cache_tag;
```

```
cache CACHE (
    .clk(clk),
    .rst(rst),
    .addr(cache_addr),
    .load(cache_load),
    .replace(cache_replace),
    .store(cache_store),
    .invalid(1'b0),
    .u_b_h_w(cache_u_b_h_w),
    .din(cache_din),
    .hit(cache_hit),
    .dout(cache_dout),
    .valid(cache_valid),
    .dirty(cache_dirty),
    .tag(cache_tag)
);
```

```
localparam
    S_IDLE = 0,
    //S_PRE_BACK = 1, //This state is redundant.
    S_BACK = 2,
    S_FILL = 3,
    S_WAIT = 4;
```

```
reg [2:0]state = 0;
reg [2:0]next_state = 0;
reg [ELEMENT_WORDS_WIDTH-1:0]word_count = 0;
reg [ELEMENT_WORDS_WIDTH-1:0]next_word_count = 0;
assign cmu_state = state;
reg sstall;
always @ (posedge clk) begin
    if (rst) begin
        state <= S_IDLE;
        word_count <= 2'b00;
    end
    else begin
```

```
            state <= next_state;
            word_count <= next_word_count;
        end
    end
```

```
    // state ctrl
    always @ (*) begin
```

```
            case (state)
                S_IDLE: begin
                    if (en_r || en_w) begin
                        if (cache_hit)
                            next_state = S_IDLE;//
                        else if (cache_valid && cache_dirty)
                            next_state = S_BACK;//
                        else
                            next_state = S_FILL;//
                    end
                    else begin
                        next_state = S_IDLE;//
                    end
                    next_word_count = 2'b00;
                end
```

```
                S_BACK: begin
                    if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})    //
2'b11 in default case
                        next_state = S_FILL;
                    else
                        next_state = S_BACK;
```

```
                    if (mem_ack_i)
                        next_word_count = word_count+1;
                    else
                        next_word_count = word_count;
                end
```

```
                S_FILL: begin
                    if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
                        next_state = S_WAIT;
                    else
                        next_state = S_FILL;
```

```
                    if (mem_ack_i)
```

```
                    next_word_count = word_count+1;
                else
                    next_word_count = word_count;
            end
```

```
            S_WAIT: begin
                next_state = S_IDLE;
                next_word_count = 2'b00;
            end
```

```
            default: begin
                next_state = 3'hx;
                next_word_count = 2'bxx;
            end
        endcase
```

```
    end

    // cache ctrl
    always @ (*) begin
        case(state)
            S_IDLE, S_WAIT: begin
                cache_addr = addr_rw;
                cache_load = en_r;
                cache_store = en_w;
                cache_replace = 1'b0;
                cache_u_b_h_w = u_b_h_w;
                cache_din = data_w;
            end
            S_BACK: begin
                cache_addr = {cache_tag, addr_rw[ADDR_BITS-TAG_BITS-1:BLOCK_WIDTH],
word_count, {ELEMENT_WORDS_WIDTH{1'b0}}}; //
                cache_load = 1'b0;
                cache_store = 1'b0;
                cache_replace = 1'b0;
                cache_u_b_h_w = 3'b010;
                cache_din = 32'b0;
            end
            S_FILL: begin
                cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH], word_count,
{ELEMENT_WORDS_WIDTH{1'b0}}};
                cache_load = 1'b0;
                cache_store = 1'b0;
                cache_replace = mem_ack_i;
```

```
                cache_u_b_h_w = 3'b010;
                cache_din = mem_data_i;
            end
```

```
            default: begin
                cache_addr = 32'hxxxxxxxx;
                cache_load = 1'hx;
                cache_store = 1'hx;
                cache_replace = 1'bx;
                cache_u_b_h_w = 3'hx;
                cache_din = 32'hxxxxxxxx;
            end
        endcase
    end
    assign data_r = cache_dout;
```

```
    // mem ctrl
    always @ (*) begin
        case (state)
            S_IDLE, S_WAIT: begin
                mem_cs_o = 1'b0;
                mem_we_o = 1'b0;
                mem_addr_o = 32'b0;
            end
```

```
            S_BACK: begin
                mem_addr_o = {cache_tag, addr_rw[ADDR_BITS-TAG_BITS-1:BLOCK_WIDTH],
word_count, {ELEMENT_WORDS_WIDTH{1'b0}}};
                mem_cs_o = 1'b1;
                mem_we_o = 1'b1;
            end
```

```
            S_FILL: begin
                mem_addr_o = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH], word_count,
{ELEMENT_WORDS_WIDTH{1'b0}}};
                mem_cs_o = 1'b1;
                mem_we_o = 1'b0;
            end
```

```
            default: begin
                mem_addr_o = 32'hxxxxxxxx;
                mem_cs_o = 1'bx;
                mem_we_o = 1'bx;
            end
```

```
        endcase
    end

    assign mem_data_o = cache_dout;
    assign stall = (next_state != S_IDLE);

endmodule
```

CMU 与 MEM，CPU 之间的数据存取过程，以及 CMU stall 的触发过程。

当 CPU 请求读/写数据时，会先请求 CMU 模块。如果该地址的数据命中缓存，CMU 可以快速提供数据，不需要访问内存。如果缓存命中，则可以直接完成数据的读/写操作，不会影响 CPU 的继续执行。
当 CPU 请求的数据不在缓存中时，CMU 需要从内存中加载该数据到缓存，然后才能继续执行。如果目标缓存行无效或者是干净的，CMU 可以直接将新数据从内存加载到缓存。如果目标缓存行是脏数据，在新的数据加载到缓存之前，需要先将脏数据写回内存。

在需要访问内存的情况下，CMU 模块会通过内存接口向内存请求读/写操作。
在 S_BACK 中，CMU 将脏数据从缓存写回到内存。
在 S_FILL 时，CMU 会将内存中的数据读到缓存中，以后续满足 CPU 对数据的请求。
CMU stall 的作用是在数据尚未准备好时阻止 CPU 继续执行，防止读取到错误的数据。cmu_stall 信号是在以下情况下被触发的：当 CPU 发出读/写请求但缓存中没有所需的数据，CMU 需要从内存中加载数据。在加载数据过程中，CPU 无法继续进行操作，需要暂停，直到数据加载完成。这时会触发 cmu_stall。在数据加载到缓存之前，如果目标缓存行是脏数据，CMU 首先需要将脏数据写回到内存，这同样需要时间。在此期间，CPU 也必须暂停，等待写回操作的完成，这时也会触发 cmu_stall。在与内存交互的过程中，内存的响应时间较长，通常需要多个时钟周期。在此期间，CPU 的指令流水线需要暂停，直到数据准备好。

代码中 CMU Stall 与控制信号的关联

cmu_stall 影响指令的执行，例如 PC_EN_IF，以及流水线寄存器的使能信号。当 cmu_stall 为高时，这些使能信号为低，表示暂停相应的模块，避免错误的数据流通

HAZARDDETECTIONUNIT.V

```
`timescale 1ps/1ps

module HazardDetectionUnit(
    input clk,
    input Branch_ID, rs1use_ID, rs2use_ID,
    input[1:0] hazard_optype_ID,
    input[4:0] rd_EXE, rd_MEM, rs1_ID, rs2_ID, rs2_EXE,
    input cmu_stall,
    output PC_EN_IF, reg_FD_EN, reg_FD_stall, reg_FD_flush,
        reg_DE_EN, reg_DE_flush, reg_EM_EN, reg_EM_flush, reg_MW_EN, reg_MW_flush,
    output forward_ctrl_ls,
    output[1:0] forward_ctrl_A, forward_ctrl_B
);
    assign reg_FD_EN = ~cmu_stall & ~load_stall;
    assign reg_DE_EN = ~cmu_stall & ~load_stall;
    assign reg_EM_EN = ~cmu_stall & ~load_stall;
    assign reg_MW_EN = ~cmu_stall & ~load_stall;
    assign reg_EM_flush = 0;
    assign reg_MW_flush = 0;
```

```
    reg[1:0] hazard_optype_EXE, hazard_optype_MEM;
    always@(posedge clk) begin
        hazard_optype_MEM <= hazard_optype_EXE & {2{~reg_EM_flush}};
        hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
    end
```

```
    parameter hazard_optype_ALU = 2'd1;
    parameter hazard_optype_LOAD = 2'd2;
    parameter hazard_optype_STORE = 2'd3;
```

```
    wire rs1_forward_1     = rs1use_ID && rs1_ID == rd_EXE && rd_EXE && hazard_optype_EXE
== hazard_optype_ALU;
    wire rs1_forward_stall = rs1use_ID && rs1_ID == rd_EXE && rd_EXE && hazard_optype_EXE
== hazard_optype_LOAD
                           && hazard_optype_ID != hazard_optype_STORE;
```

```
    wire rs1_forward_2     = rs1use_ID && rs1_ID == rd_MEM && rd_MEM && hazard_optype_MEM
== hazard_optype_ALU && !rs1_forward_1;
    wire rs1_forward_3     = rs1use_ID && rs1_ID == rd_MEM && rd_MEM && hazard_optype_MEM
== hazard_optype_LOAD;
```

```
    wire rs2_forward_1     = rs2use_ID && rs2_ID == rd_EXE && rd_EXE && hazard_optype_EXE
== hazard_optype_ALU;
    wire rs2_forward_stall = rs2use_ID && rs2_ID == rd_EXE && rd_EXE && hazard_optype_EXE
== hazard_optype_LOAD
                             && hazard_optype_ID != hazard_optype_STORE;
    wire rs2_forward_2     = rs2use_ID && rs2_ID == rd_MEM && rd_MEM && hazard_optype_MEM
== hazard_optype_ALU && !rs2_forward_1;
    wire rs2_forward_3     = rs2use_ID && rs2_ID == rd_MEM && rd_MEM && hazard_optype_MEM
== hazard_optype_LOAD;
```

```
    wire load_stall = rs1_forward_stall | rs2_forward_stall;
```

```
    assign PC_EN_IF =  ~load_stall & ~cmu_stall ;
    assign reg_FD_stall = load_stall;
    assign reg_FD_flush = Branch_ID;
    assign reg_DE_flush = load_stall;
```

```
    assign forward_ctrl_A = {2{rs1_forward_1}} & 2'd1 |
                            {2{rs1_forward_2}} & 2'd2 |
                            {2{rs1_forward_3}} & 2'd3 ;
```

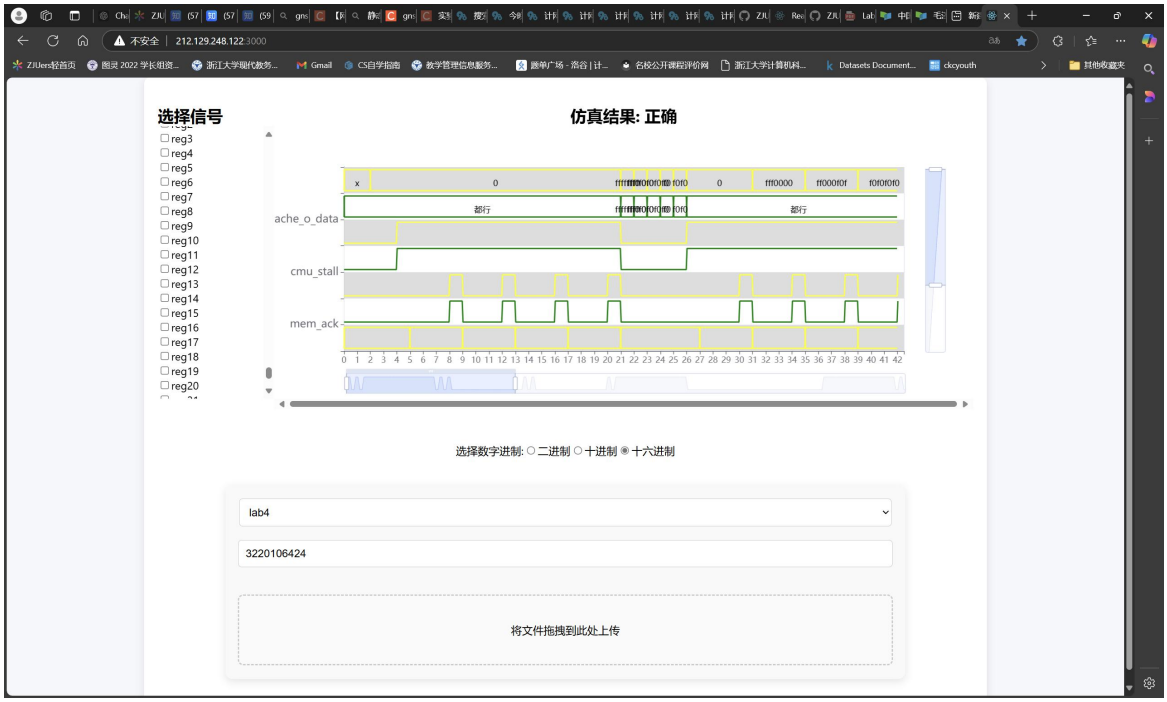
```
    assign forward_ctrl_B = {2{rs2_forward_1}} & 2'd1 |
                            {2{rs2_forward_2}} & 2'd2 |
                            {2{rs2_forward_3}} & 2'd3 ;
```

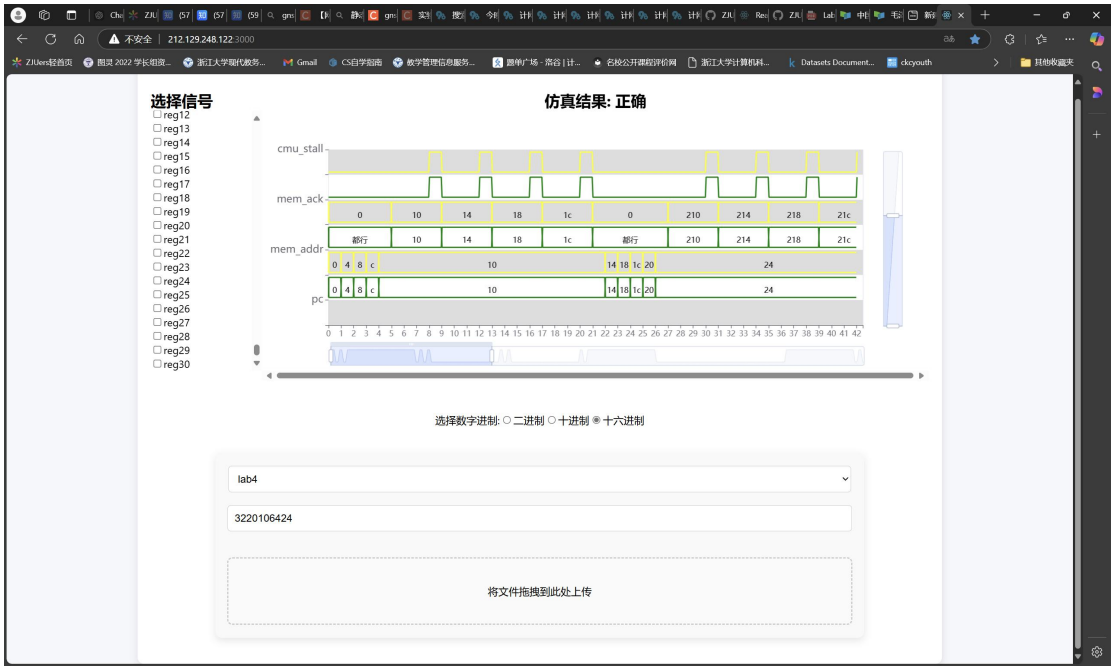
```
    assign forward_ctrl_ls = rs2_EXE == rd_MEM && hazard_optype_EXE ==
hazard_optype_STORE
                            && hazard_optype_MEM == hazard_optype_LOAD;
```

```
endmodule
```

三、　实验过程和数据记录及结果分析

开始 lb x1, 0x01C(x0)此时 cache 为空,发生了 miss，把对应数据读入缓存





之后 4 条指令都是 hit

接下来 lw x0，0x210(x0)是 miss 读入 0x210-0x21c 数据

之后 3 条指令都无异常

接下来 sb x1，0x0(x0)发生 miss 读入 0x000-0x00c 数据

之后两条指令都命中

接下来 lw x6 0x200(x0)发生 miss 读入 0x200-0x20c 数据

接下来 lw x7 0x400(x0)发生 miss 把 0x000-0x00c 放回 ram，之后读入 0x400-0x40c

接下来 lw x8 0x410(x0)发生 miss 但是因为数据不是 dirty 所以不用写回，读入 0x410-0x41c

四、 讨论与心得

本次实验依托于实验三，本身不算难，但是生成 bit 流时候长期停留在 running route_design 让人心烦意乱