中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929034 姓名：侯汝垚 专业：计算机科学与技术

实验序号：5.5 实验名称：完整系统集成

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 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、 逻辑电路结构与仿真波形的截图及说明

1. 流水线中断处理

```
wire IF_intr_block = (IF_IW_intr || IW_ID_intr) || (ID_EX_intr || EX_MEM_intr) || (MEM_RDW_intr || RDW_WB_intr);
```

```
reg IF_IW_intr;
always @(posedge clk) begin
        if (rst)
                IF_IW_intr <= 1'b0;
        else if (Go && (load_hazard_control))
                IF_IW_intr <= 1'b0;
        else if (Go)
                IF_IW_intr <= intr && ~no_intr;
end
```

在 IF 级判断是否有中断，中断信息进入流水先一路传递，在流水线中存在中断信息时阻塞 IF 级，即不再发送指令请求，PC 不变，在中断信息进入 IW 级后，IR 中填入 NOP 指令，当中断信息到写回阶段后，将 PC 改为中断服务程序地址并保存 PC_plus，即进入中断时的下一条指令的地址。

```
always @(posedge clk) begin
        if (rst)
                PC_reg <= 32'b0;
        else if (Go && load_hazard_control)
                PC_reg <= PC_reg;
        else if (Go && (branchValid || jump))
                PC_reg <= PC_branch;
        else if (Go && RDW_WB_MRET)
                PC_reg <= MEPC;
        else if (Go && RDW_WB_intr)
                PC_reg <= 32'b100;
        else if (Go)
                PC_reg <= PC_plus;
end
```

```
wire MRET = ID_EX_MRET || EX_MEM_MRET || MEM_RDW_MRET || RDW_WB_MRET;
// store instruction
reg [31:0] IW_ID_IR;
always @(posedge clk) begin
        if (rst)
                IW_ID_IR <= 32'b0;
        else if ((load_hazard_control || IW_intr_block) && Go)
                IW_ID_IR <= IW_ID_IR;
        else if ((branchValid || jump || pre_failed || MRET) && Go)
                IW_ID_IR <= 32'b0;
        else if (Go)
                IW_ID_IR <= IR;
end
```

```
reg [31:0] MEPC;
always @(posedge clk) begin
        if (rst)
                MEPC <= 32'b0;
        else if (RDW_WB_intr && Go)
                MEPC <= PC_plus;
end
```

当遇到 MRET 指令时，在 ID 阶段即可译码得到是否为 MRET，若为 MRET，则类似分支预测失败时候的处理，将 IF 级和 IW 级对应的指令无效化，变为 NOP，然后将 MEPC 寄存器的值写入 PC。

2. 软件支持的缓存一致性

增加五组信号 cpu 和 cache 间的信号来支持三个 cache 操作，clean、invalid 和 flush。其中 cache_op 为对应的操作，00 未定义，01 为 clean，10 为 invalid，11 为 flush，然后两组握手信号和内存读类似，地址采用访存的 Address 通道。

```
// Cache operation channel
output [ 1:0] Cache_op,
output        Cache_op_Req_Valid,
input         Cache_op_Req_Ready,

output        Cache_op_Ready,
input         Cache_op_Valid,
```

修改 dcache 对应的状态机，增加 cache 操作对应的状态。Cache_op 状态根据 cache_op 进行相应的操作，若操作为 invalid 只需将对应 cache 块 valid 置零，若操作为 clean 且对应 cache 块为 dirty 则进入 cache 块写回的状态然后，写回后 valid 置零，若操

作为 clean 则通过排队器选出所有 dirty 的 cache 块，重复进行 clean 操作。当 cache 操作完成后进入 op_valid 状态拉高 Cache_op_Valid，等待握手成功后回到初始 WAIT 状态。

```
CACHE_OP      = 20'b01000000000000000000,
OP_VALID      = 20'b10000000000000000000;
```

```
wire [7:0] valid_suite = (valid_array[0] & dirty_array[0] | valid_array[1] & dirty_array[1]) | (valid_array[2] & dirty_array[2] | valid_array[3] & dirty_array[3]);
// a queue to find the valid and dirty suite
wire [7:0] flush_suite =  {
        ((~valid_suite[0] && ~valid_suite[1]) && (~valid_suite[2] && ~valid_suite[3])) && ((~valid_suite[4] && ~valid_suite[5]) && (~valid_suite[6] && valid_suite[7])),
        ((~valid_suite[0] && ~valid_suite[1]) && (~valid_suite[2] && ~valid_suite[3])) && ((~valid_suite[4] && ~valid_suite[5]) && valid_suite[6]),
        (~valid_suite[0] && ~valid_suite[1]) && (~valid_suite[2] && ~valid_suite[3]) && (~valid_suite[4] && valid_suite[5]),
        (~valid_suite[0] && ~valid_suite[1]) && (~valid_suite[2] && ~valid_suite[3]) && valid_suite[4],
        (~valid_suite[0] && ~valid_suite[1]) && (~valid_suite[2] && valid_suite[3]),
        ~valid_suite[0] && ~valid_suite[1] && valid_suite[2],
        ~valid_suite[0] && valid_suite[1],
        valid_suite[0]
};
```

```
// select a dirty block in a suite to write back
wire [3:0] flush_valid_block =  {
        ~valid_array[0][flush_suite_addr] && ~valid_array[1][flush_suite_addr] && ~valid_array[2][flush_suite_addr] && valid_array[3][flush_suite_addr],
        ~valid_array[0][flush_suite_addr] && ~valid_array[1][flush_suite_addr] && valid_array[2][flush_suite_addr],
        ~valid_array[0][flush_suite_addr] && valid_array[1][flush_suite_addr],
        valid_array[0][flush_suite_addr]
};

wire [3:0] flush_block = flush_valid_block & {dirty_array[3][flush_suite_addr], dirty_array[2][flush_suite_addr],
                                              dirty_array[1][flush_suite_addr], dirty_array[0][flush_suite_addr]};

wire [31:0] flush_addr = {32{flush_block[0]}} & {rd_tag[0], flush_suite_addr, 5'b0}
                       | {32{flush_block[1]}} & {rd_tag[1], flush_suite_addr, 5'b0}
                       | {32{flush_block[2]}} & {rd_tag[2], flush_suite_addr, 5'b0}
                       | {32{flush_block[3]}} & {rd_tag[3], flush_suite_addr, 5'b0};
```

3. 定制指令

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

使用 custom-0 作为 cache 指令，定义为 C-type，func3 字段表示进行的 cache 操作，001 表示 clean，010 表示 invalid，100 表示 flush。因 rdata2 在存数指令中也要传到 MEM 级，而 rdata1 只传到 EX 级，所以用 rs2 寄存器的值 rdata2 表示需要刷新的 cache 块地址，传入 Address。

```
wire C_type = (opcode[6:2] == 5'b00010); // custom-0 cache instruction
```

```
assign Cache_op = {2{EX_MEM_func3[0]}} & 2'b01
                | {2{EX_MEM_func3[1]}} & 2'b10
                | {2{EX_MEM_func3[2]}} & 2'b11;
```

```
assign Address  = {32{~EX_MEM_Cache}} & {EX_MEM_ALUOut[31:2], 2'b0}
                | {32{ EX_MEM_Cache}} & EX_MEM_rdata2;
```

在 dma 每次填充完一片缓存区后执行 flush 操作，将 cache 块写回内存，使其可以通过 dma 传输。同理 dnn 实验中在启动硬件加速器后刷新 cache，保证读取到的是正确的输出。

```c
void cache_flush()
{
        asm volatile(
            ".insn r 0x0b, 4, 0, zero, zero, zero"
            :
            :
            : "memory");
}
```

```c
for (int i = 0; i < sub_buf_num; i++) {
        generate_data((unsigned int *)buf);

        // move buffer pointer to next sub region
        buf += DMA_SIZE;
        dma_buf_stat++;

USE_DMA
        // refresh head ptr in DMA engine
        cache_flush();
        reg_val = *(dma_mmio + (DMA_HEAD_PTR >> 2));
        reg_val += DMA_SIZE;
        *(dma_mmio + (DMA_HEAD_PTR >> 2)) = reg_val;
        printf("%x\n", *(dma_mmio + (DMA_CTRL_STAT >> 2)))

}
```

4. 实验框架修改

因为这个实验没有为学生准备太多框架，一些内容需要自行修改，很多东西 ppt 中没有介绍，我自行探索了一番。

首先是 risc-v 的中断服务程序编写，先前的 dma 实验中只有 mips 的汇编，因此我仿照 mips 的内容创建 risc-v/common/start.S 和 intr_handler.S，其中 start.S 复制了 mips 的 start.S，只是将寄存器名前面的$删去。Intr_handler.S 修改较多，保存能以后可以使用多个寄存器，更加方便。

software > workload > ucas-cod > benchmark > simple_test > dma_test > riscv32 > common > start.S

```
1   .globl global_result
2   .globl dma_buf_stat
3   .globl start
4   .globl _SP_START_
5   .type start, @function
6   .section .start
7
8   start:
9           li sp, 0x4000
10          j continued
11          # nop here
12
13  global_result:
14          .word 0xffffffff
15
16  dma_buf_stat:
17          .word 0x0
18
19  continued:
20          la  sp, _SP_START_
21          jal main
22          jal hit_good_trap
23
```

```
.data
last_tail_ptr:
        .word 0

.globl intr_handler
        .align  2
        .type intr_handler, @function
        .section .exception

intr_handler:
        # TODO: Please add your own interrupt handler for DMA engine

        # set INTR to 0
        sw      t0, -4(sp)
        sw      t1, -8(sp)
        sw      t2, -12(sp)
        sw      t3, -16(sp)

        li      t2, 0x7fffffff
        lui     t0, 0x60020
        lw      t1, 0x14(t0)
        and     t1, t1,t2
        sw      t1, 0x14(t0)

        # store new tail_ptr, calculate new_tail_ptr - last_tail_ptr in $k0
        lw      t1, 0x08(t0)
        lw      t2, last_tail_ptr
        sub     t2, t1, t2
        la      t3, last_tail_ptr
        sw      t1, 0(t3)

        # (dma_buf_stat - $k0) / dma_buf_size

        # dma_buf_stat
        lw      t1, 0x10(zero)
        # dma_buf_size
        lw      t3, 0x10(t0)
L1:
        addi    t1, t1, -1
        sub     t2, t2, t3
        bne     t2, zero, L1

        sw      t1, 10(zero)
        lw      t0, -4(sp)
        lw      t1, -8(sp)
        lw      t2, -12(sp)
        lw      t3, -16(sp)
        mret
```

然后修改仿真顶层模块 cpu_test_top.v 的连接，增加五个信号。

```
wire [ 1:0] Cache_op;
wire        Cache_op_Req_Valid;
wire        Cache_op_Req_Ready;
wire        Cache_op_Valid;
wire        Cache_op_Ready;
```

```
`ifdef USE_DCACHE
  dcache_wrapper  u_dcache_wrapper (
`else
  mem_if_wrapper  u_mem_if_wrapper (
`endif
        .cpu_clk        (cpu_clk),
        .cpu_reset      (~cpu_reset_n),

        .Address        (Address),
        .MemWrite       (MemWrite),
        .Write_data     (Write_data),
        .Write_strb     (Write_strb),
        .MemRead        (MemRead),
        .Mem_Req_Ready  (Mem_Req_Ready),

        .Read_data       (Read_data),
        .Read_data_Valid (Read_data_Valid),
        .Read_data_Ready (Read_data_Ready),

        .cpu_cache_op       (Cache_op),
        .Cache_op_Req_Valid (Cache_op_Req_Valid),
        .Cache_op_Req_Ready (Cache_op_Req_Ready),
        .Cache_op_Valid     (Cache_op_Valid),
        .Cache_op_Ready     (Cache_op_Ready),

        .cpu_mem_araddr    (cpu_mem_araddr ),
```

```
  //custom CPU core
  custom_cpu    u_cpu (
        .clk            (cpu_clk),
        .rst            (~cpu_reset_n),

        .PC             (PC),
        .Inst_Req_Valid (Inst_Req_Valid),
        .Inst_Req_Ready (Inst_Req_Ready),

        .Instruction    (Instruction),
        .Inst_Valid     (Inst_Valid),
        .Inst_Ready     (Inst_Ready),

        .Address        (Address),
        .MemWrite       (MemWrite),
        .Write_data     (Write_data),
        .Write_strb     (Write_strb),
        .MemRead        (MemRead),
        .Mem_Req_Ready  (Mem_Req_Ready),

        .Read_data       (Read_data     ),
        .Read_data_Valid (Read_data_Valid),
        .Read_data_Ready (Read_data_Ready),

        .Cache_op       (Cache_op),
        .Cache_op_Req_Valid (Cache_op_Req_Valid),
        .Cache_op_Req_Ready (Cache_op_Req_Ready),
        .Cache_op_Valid (Cache_op_Valid),
        .Cache_op_Ready (Cache_op_Ready)
  );
```

然后修改 fpga 连接，仿照前面的语法连接几个 pin。

```
connect_bd_net [get_bd_pins u_custom_cpu/Cache_op] \
    [get_bd_pins ${mem_if_entity}/cpu_cache_op]
connect_bd_net [get_bd_pins u_custom_cpu/Cache_op_Req_Valid] \
    [get_bd_pins ${mem_if_entity}/Cache_op_Req_Valid]
connect_bd_net [get_bd_pins u_custom_cpu/Cache_op_Req_Ready] \
    [get_bd_pins ${mem_if_entity}/Cache_op_Req_Ready]
connect_bd_net [get_bd_pins u_custom_cpu/Cache_op_Ready] \
    [get_bd_pins ${mem_if_entity}/Cache_op_Ready]
connect_bd_net [get_bd_pins u_custom_cpu/Cache_op_Valid] \
    [get_bd_pins ${mem_if_entity}/Cache_op_Valid]
```

在 dcache 的封装模块 dcache_wrapper.v 中也连接这五个信号。

```
module dcache_wrapper (
        input           cpu_clk,
        input           cpu_reset,

        //Memory request channel
        input  [31:0]   Address,
        input           MemWrite,
        input  [31:0]   Write_data,
        input  [ 3:0]   Write_strb,
        input           MemRead,
        output          Mem_Req_Ready,

        //Memory data response channel
        output [31:0]   Read_data,
        output          Read_data_Valid,
        input           Read_data_Ready,

        // Cache operation channel
        input [ 1:0]    cpu_cache_op,
        input           Cache_op_Req_Valid,
        output          Cache_op_Req_Ready,
        output          Cache_op_Valid,
        input           Cache_op_Ready,

        //AXI AR Channel for data
```

```
dcache_top      u_dcache (
        .clk            (cpu_clk),
        .rst            (cpu_reset),

        //CPU interface
        .from_cpu_mem_req_valid    (MemRead | MemWrite),
        .from_cpu_mem_req          ( (~MemRead) | MemWrite ),
        .from_cpu_mem_req_addr     (Address),
        .from_cpu_mem_req_wdata    (Write_data),
        .from_cpu_mem_req_wstrb    (Write_strb),
        .to_cpu_mem_req_ready      (Mem_Req_Ready),

        .to_cpu_cache_rsp_valid    (Read_data_Valid),
        .to_cpu_cache_rsp_data     (Read_data),
        .from_cpu_cache_rsp_ready  (Read_data_Ready),

        //Memory interface
        .to_mem_rd_req_valid       (cpu_mem_arvalid),
        .to_mem_rd_req_addr        (to_mem_rd_req_addr),
        .to_mem_rd_req_len         (cpu_mem_arlen),
        .from_mem_rd_req_ready     (cpu_mem_arready),

        .from_mem_rd_rsp_valid     (cpu_mem_rvalid),
        .from_mem_rd_rsp_data      (cpu_mem_rdata),
        .from_mem_rd_rsp_last      (cpu_mem_rlast),
        .to_mem_rd_rsp_ready       (cpu_mem_rready),

        .to_mem_wr_req_valid       (cpu_mem_awvalid),
        .to_mem_wr_req_addr        (to_mem_wr_req_addr),
        .to_mem_wr_req_len         (cpu_mem_awlen),
        .from_mem_wr_req_ready     (cpu_mem_awready),

        .to_mem_wr_data_valid      (cpu_mem_wvalid),
        .to_mem_wr_data            (cpu_mem_wdata),
        .to_mem_wr_data_strb       (cpu_mem_wstrb),
        .to_mem_wr_data_last       (cpu_mem_wlast),
        .from_mem_wr_data_ready    (cpu_mem_wready),

        .from_cpu_cache_op           (cpu_cache_op),
        .from_cpu_cache_op_req_valid (cpu_cache_op_req_valid),
        .to_cpu_cache_op_req_ready   (cache_op_req_ready),
        .to_cpu_cache_op_valid       (cpu_cache_op_valid),
        .from_cpu_cache_op_ready     (cpu_cache_op_ready)
);
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

1. Risc-v 汇编出错

```
riscv32/common/intr_handler.S:19: Error: illegal operands `addi t2,0xffff'
```

Risc-v 中 I-type 的立即数只支持十二位，而 mips 的立即数为 16 位，因此为了加载一个 32 位数需要先通过 lui 加载高的 20 位，然后用 I-type 指令加载低 12 位。

2. 仿真加速得到的波形没有增加的信号

本来想用仿真加速进行调试，但是发现波形中没有增加的五个信号，根据对实验框架的观察，fpga_emu 和行为仿真用的不是同一个顶层模块，如果想看需要修改 emu 文件夹中的模块。但是修改之后用 GTKWave 看时这几个信号为高阻态，说明还是有些实验框架上的细节没有更改，不能正确运用这套调试工具。

3. dma 错误通过测试

在验收时惊喜地发现 dma 居然通过了测试，然而这套没有经过 debug 的代码通过测试是不可能的。仔细观察发现程序运行时间较长并且没有打印“Prepare DMA engine”等信息，说明这次通过只是巧合。

```
RUNNER_CNT = 4
Completed FPGA configuration
Launching data_mover_dma 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
time 1000743.57ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/dma_test/riscv32/elf/data_mover_dma passed
Hit good trap
```

三、 在课后，你花费了大约 10 小时完成此次实验。

四、 对于此次实验的心得、感受和建议

本次实验的实验框架不太完善，虽然实验思路比较清晰，但是没有配套的调试或测试环境，自主探索空间比较大，最终还是没有完成。Cache 指令和中断的总体思路已经写完，不过没有经过调试，距离成功还比较遥远，算是一个半成品。