中国科学院大学计算机组成原理实验课

实 验 报 告

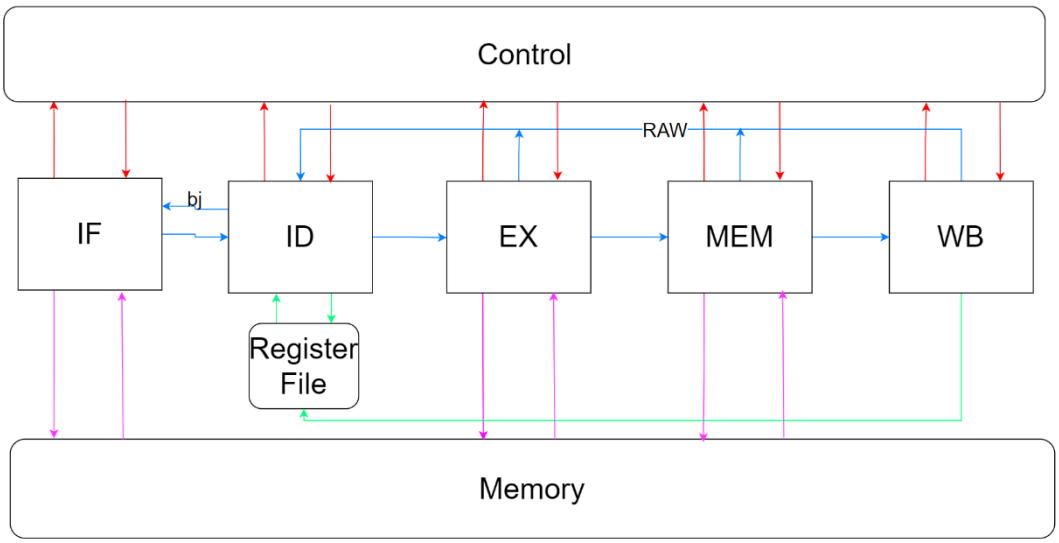
学号：2019K8009915025　　姓名：马月骁　　专业：计算机科学与技术

实验序号：prj5-turbo-cpu　　实验名称：复杂处理器设计

一、 逻辑电路结构与 FPGA 运行结果的截图及说明

复杂处理器设计实验使用 RSICV 指令集，使用流水线架构为五级流水线。复杂处理器设计实验共使用 9 个硬件设计文件以及 1 个头文件：custom_cpu.v，IF_stage.v，ID_stage.v，EX_stage.v，MEM_stage.v，WB_stage.v，reg_file.v，alu.v，shifter.v；my_cpu.h。其中 reg_file.v，alu.v，shifter.v 直接复用 prj4 中的对应文件；其余文件为本次实验中编写。

以下为复杂处理器设计实验中设计的 RSICV 五级流水线处理器整体逻辑电路结构图：



RSICV 五级流水线处理器整体逻辑电路结构图

（一）RSICV 五级流水线处理器设计与相关代码

1．流水线控制与数据传递

（1）各流水级间

后一级流水向前一级发送 yy_allowin 信号：高电平表示后一级流水可以正常接收数据；低电平表示后一级流水处于工作状态或阻塞状态。

前一级流水向后一级发送 xx_to_yy_valid 信号：高电平表示前一级流水正常工作，输出数据正确；低电平表示前一级流水处于工作状态或阻塞状态。

前一级流水向后一级发送 xx_to_yy_bus：当 xx_to_yy_valid 为高电平时 bus 中正确传递数据。

（2）各流水级内

各流水级内部由 xx_allowin，xx_ready_go 信号以及 xx_valid 寄存器控制：

xx_allowin 表示该级流水能否正常读入数据，高电平时修改 xx_valid 寄存器。xx_allowin 信号由后一级流水的 yy_allowin 信号以及本级流水的 xx_ready_go 控制。

xx_ready_go 表示该级流水工作是否完成完成。xx_ready_go 信号由该级流水所需动作具体决定。通过 xx_ready_go 信号可实现阻塞控制。

xx_valid 表示该级流水线工作是否有效。xx_valid 寄存器由 xx_allowin 信号以及上一级流水传入的 zz_to_xx_valid 信号控制与决定。当 xx_valid 为高电平时，更新流水线中寄存器。

xx_valid 寄存器与 xx_ready_go 信号共同控制 xx_to_yy_valid 信号。

以 MEM_stage 相关代码为例：

```
63    assign mem_ready_go    = ~mem_load_wen | Read_data_Valid;
64    assign mem_allowin     = !mem_valid || mem_ready_go && wb_allowin;
65    assign mem_to_wb_valid = mem_valid & mem_ready_go;
66    always @(posedge clk) begin
67        if(rst) begin
68            mem_valid <= 1'b0;
69        end
70        else if(mem_allowin) begin
71            mem_valid <= ex_to_mem_valid;
72        end
73
74        if(ex_to_mem_valid & mem_allowin) begin
75            ex_to_mem_bus_r <= ex_to_mem_bus;
76        end
77    end
```

2. RAW 处理（旁路设计）：

由于在 ID 阶段访问寄存器时，对应寄存器中内容可能已经被 EX，MEM，WB 三个阶段中改变，故需要三条从 EX，MEM，WB 到 ID 的旁路来保证 ID 阶段读取寄存器中数据的正确性。

特别的，对于 load 指令，由于其在 MEM 阶段才能得到正确数据，并且内存访问存在延时，故需要添加 rdw_xx_addr_valid 以及 rdw_xx_data_valid 信号来确定旁路传回的寄存器地址以及数据的有效性。

以下为 EX_stage 相关代码：

```
99     assign rdw_ex_bus = {~ex_load_wen, //38:38
100                         ex_wb_wen & ex_valid, //37:37
101                         ex_dest, //36:32
102                         result //31:0
103                        };
```

在 ID 阶段，寄存器中正确内容由旁路和当时寄存器内容确定。其间优先级关系：EX（by_path）>MEM（by_path）>WB（by_path）>ID（reg_file）

特别的，对于 load 指令，由于其在 MEM 阶段才能得到正确数据，并且内存访问存在延时，故需要在其未得到正确数据时将 ID 以及 IF 阻塞，直到 load 取到正确数据。

以下为 ID_stage 相关代码：

数据选择：

```
assign rs1_from_ex  = (|rs1) & rdw_ex_data_valid & (rdw_ex_addr == rs1);
assign rs1_from_mem = (|rs1) & ~rs1_from_ex & rdw_mem_data_valid & (rdw_mem_addr == rs1);
assign rs1_from_wb  = (|rs1) & ~rs1_from_ex & ~rs1_from_mem & rdw_wb_data_valid & (rdw_wb_addr == rs1);
assign rs1_from_rf  = ~(rs1_from_ex |rs1_from_mem | rs1_from_wb);
assign rs2_from_ex  = (|rs2) & rdw_ex_data_valid & (rdw_ex_addr == rs2);
assign rs2_from_mem = (|rs2) & ~rs2_from_ex & rdw_mem_data_valid & (rdw_mem_addr == rs2);
assign rs2_from_wb  = (|rs2) & ~rs2_from_ex & ~rs2_from_mem & rdw_wb_data_valid & (rdw_wb_addr == rs2);
assign rs2_from_rf  = ~(rs2_from_ex |rs2_from_mem | rs2_from_wb);
assign rs1_value  = ({32{rs1_from_ex}} & rdw_ex_data) |
                    ({32{rs1_from_mem}} & rdw_mem_data) |
                    ({32{rs1_from_wb}} & rdw_wb_data) |
                    ({32{rs1_from_rf}} & RF_rdata1);
assign rs2_value  = ({32{rs2_from_ex}} & rdw_ex_data) |
                    ({32{rs2_from_mem}} & rdw_mem_data) |
                    ({32{rs2_from_wb}} & rdw_wb_data) |
                    ({32{rs2_from_rf}} & RF_rdata2);
```

ID 阻塞（通过控制 ID_ready_go 实现）

```
assign id_ready_go   = ~((B_type & (~rs1_from_rf | ~rs2_from_rf)) |
                         (~rdw_ex_addr_valid & (rs1_from_ex | rs2_from_ex)) |
                         (~rdw_mem_addr_valid & (rs1_from_mem | rs2_from_mem)) |
                         (~rdw_wb_addr_valid & (rs1_from_wb | rs2_from_wb))
                        );
assign id_allowin     = !id_valid || id_ready_go && ex_allowin;
assign id_to_ex_valid = id_valid & id_ready_go;
always @(posedge  clk) begin
    if(rst) begin
        id_valid <= 1'b0;
    end
    else if(id_allowin) begin
        id_valid <= if_to_id_valid;
    end

    if(if_to_id_valid & id_allowin) begin
        if_to_id_bus_r <= if_to_id_bus;
    end
end
```

3. branch 与 jump 处理

branch 与 jump 相关处理在 ID 流水级完成，并将结果直接传回 IF 流水级。

branch：为在 ID 得到 branch 跳转结果，例化 alu.v 中的 CLA 模块，用于计算 branch 比较结果。

jump：jalr 需要通过计算来求得跳转的 PC 值，可复用 CLA 实现。

CLA 相关代码与设计思路可参考 prj2。

以下为 ID 中相关代码：

```
245     assign CLA_A = rs1_value;
246     assign CLA_B = ({32{B_type}} & ~rs2_value) |
247                    ({32{I_type_jalr}} & imm_extension);
248     CLA CLA(
249         .A(CLA_A),
250         .B(CLA_B),
251         .CIN(B_type),
252         .SF(CLA_SF),
253         .ZF(CLA_ZF),
254         .OF(CLA_OF),
255         .CF(CLA_CF),
256         .S(CLA_result)
257     );
258     assign branch        = B_type & (((~|funct3) & CLA_ZF) |
259                                      ((funct3 == 3'b001) & ~CLA_ZF) |
260                                      ((funct3 == 3'b100) & (CLA_OF ^ CLA_SF)) |
261                                      ((funct3 == 3'b101) & ~(CLA_OF ^ CLA_SF)) |
262                                      ((funct3 == 3'b110) & CLA_CF) |
263                                      ((&funct3) & ~CLA_CF)
264                                     );
265     assign branch_result = id_pc + Btype_extension;
266     assign jump          = J_type | I_type_jalr;
267     assign jump_result   = ({32{J_type}} & (id_pc + Jtype_extension)) |
268                            ({32{I_type_jalr}} & {CLA_result[31:1], 1'b0});
269     assign bj_wen        = (branch | jump) & id_valid;
270     assign bj_pc         = ({32{branch}} & branch_result) |
271                            ({32{jump}} & jump_result);
272     assign bj_bus        = {bj_wen, bj_pc};
273     assign to_if_valid   = id_valid;
```

由于当 ID 阶段处理 branch 以及 jump 类型指令时，IF 阶段已经向内存发送了访问请求，并且随后会取得对应指令。但依据 branch 以及 jump 此时取得的指令可能是错误的，故需要添加寄存器来协同控制取得指令的有效性，并当 ID 阶段处理 branch 以及 jump 类型指令时，将 IF 阻塞。

特别的，当 branch 前存在 load 指令时，亦可通过上述方式控制，

以下为 IF_stage 相关代码：

```
42      assign {bj_wen, bj_pc}          = bj_bus;
43      assign to_if_valid              = ~rst;
44      assign seq_pc                   = if_pc + 4;
45      assign next_pc                  = bj_wen ? bj_pc : seq_pc;
46
47      reg bj_wen_r;
48      reg bj_wen_t;
49      always @(posedge clk) begin
50          if(rst) begin
51              bj_wen_r <= 1'b0;
52          end
53          else if(id_valid | if_allowin) begin
54              bj_wen_r <= bj_wen;
55          end
56
57          bj_wen_t <= bj_wen_r;
58      end
```

```
60      always @(posedge clk) begin
61          if(rst) begin
62              next_pc_r <= seq_pc;
63          end
64          else if (id_valid | ~bj_wen_r & bj_wen_t | (~|if_pc)) begin
65              next_pc_r <= next_pc;
66          end
67      end
```

4．与寄存器堆与内存交互

（1）寄存器堆

寄存器堆在 ID 流水级中例化，故需要数据通路从 WB 流水级将写寄存器的相关数据传入 ID 流水级。

（2）内存

取指：取值在 IF 流水级完成，通过添加 ins_req_succ 寄存器记录是否成功发送访问请求。

以下为 IF_stage 相关代码：

```
88      always @(posedge clk) begin
89          if(if_allowin) begin
90              ins_req_succ <= Inst_Req_Valid & Inst_Req_Ready;
91          end
92
93      end
94      assign pc             = next_pc_r;
95      assign Inst_Req_Valid = to_if_valid & if_allowin;
96      assign Inst_Ready     = ins_req_succ & id_allowin | rst ;
```

存入/取出数据：在 EX 流水级发送访问请求，在 MEM 流水级实现 load 取数。

以下为 EX_stage 相关代码：

```
133     always @(posedge clk) begin
134         if(~mem_req_succ | ex_allowin) begin
135             mem_req_succ <= Mem_Req_Ready & (MemRead | MemWrite);
136         end
137     end
138
139     assign Address    = {ALU_result[31:2], 2'b0};
140     assign MemRead    = ex_load_wen & ex_valid & ~mem_req_succ;
141     assign MemWrite   = ex_store_wen & ex_valid & ~mem_req_succ;
```

以下为 MEM_stage 相关代码：

```
79    assign Read_data_Ready = mem_load_wen & mem_valid & wb_allowin | rst;
80    assign mem_read_data   = Read_data;
```

（二）FPGA 运行结果分析

basic，medium，advanced 三组测试正常通过

hello 打印结果如下：

```
42  testing 1 2 0000003
43  faster and "cheaper"
44  deadf00d % DEADF00D
45  00000000100000000200000000300000004000000005
46  50 50 -50 4294967246
```

以下进行 RISCV 五级流水线处理器与 RISCV 多周期处理器性能比较，由于 testbench 基本一致，故在指令一致的基础上进行两处理器的 CPI 比较。

testbench	指令总数	周期数（多周期）	周期数（五级流水）	CPI（多周期）	CPI（五级流水）
32_15pz	5224573	570658298	383339212	109.23	73.37
33_bf	452946	42701054	x	94.27	x
34_dinic	16783	1526821	1189346	90.97	70.87
35_fib	2549617	198544257	199454174	77.87	78.23
36_md5	5007	403005	350087	80.49	69.92
37_qsort	9572	813526	719930	84.99	75.21
38_queen	81582	7077208	5736816	86.75	70.32
39_sieve	10287	829272	765305	80.61	74.40
40_ssort	619137	49172528	45267933	79.42	73.11

RISC-V 多周期与五级流水线处理器性能计数器数据处理表

注：五级流水处理器中 testbench 33_bf 运行出现问题，具体问题在第二部分讨论，相关数据不纳入此处结果比较。

由上表可得，除 testbench 35_fib 之外，五级流水线处理器相较多周期处理器的 CPI 都有减少，并且对于越复杂的程序处理时间缩短的越多。

五级流水线处理器的平均 CPI 为 73.18，多周期处理器的平均 CPI 为 87.18。可以认为五级流水线处理器的性能相较多周期处理器有一定的提升。

但是由于访问真实内存时，延时较高。当频繁进行真实内存访问时，五级流

水线处于频繁处于部分阻塞的状态，导致并行的指令数减少，以至于同多周期处理器性能差异不大。同时，由于运行状态稳定性的波动，可能导致类似于 testbench 35_fib 中五级流水线处理器性能劣于多周期处理器的情况。

因此，五级流水线处理器的平均 CPI 相较多周期处理器有所减少，但减少幅度不大，需要 cache 的辅助来实现性能的进一步提升。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

（一）流水线控制信号与数据通路设计不当

问题：RAW 问题中 load 处理不当；branch 跳转处理不当。具体体现为未进行恰当的阻塞。

解决方案：根据波形图，定位出错信号位置，确定相关出错指令，针对相应情况添加/修改阻塞条件，保证数据的正确性。

（二）代码逻辑错误

1. 缺少信号声明（信号名错误）

问题（原因）：编写代码时遗漏信号申明，导致运行时信号值为 z，无法正常运行。

解决方案：根据波形图，依据逻辑回溯错误信号，定位出错时间点和出错信号，再加以修改。

2. 信号赋值出错

问题：编写代码时由于笔误设置信号宽度出错、选取信号位数出错、按位赋值时选取错误信号位数。

解决方案：根据波形图，定位出错信号位置，逐行排查逻辑，加以修改；编写代码时小心谨慎，防止书写错误。

3. 译码出错

问题：本次实验在 prj4 的基础上修改实现，由于部分指令的处理流程有所修改，导致潜在需要修改部分疏忽未修改。

解决方案：根据波形图，定位出错信号位置，确定出错指令，比对译码表，以及 prj4 中相关代码。

（三）FPGA 上版运行错误

问题：testbench 33_bf 在未添加后续 IO 访问相关软件设计代码时能够正常通过仿真，上版运行结果也正确。但完善后续 IO 访问相关软件设计代码后无法通过上版测试。

可能的解决方案：

1. 抓取 FPGA 版上信号，精准定位出错位置

2. 在软件设计代码中添加打印语句，定位大致出错范围

（四）调试过程难点

1. 性能计数器结果

问题：仿真阶段相关 testbench 无法正常运行。

该问题由于比对机制造成，暂无解决方案。

2. 调试时间成本时间过高

问题 1：在全自动开发流程中，当仿真出错时，需要修改.gitlab-ci.yml 脚本（增加 DUMP_TIME)，重新 push，重新仿真才能看到波形。同时由于 DUMP_TIME 要求于仿真运行时间保持一致(少于仿真运行时间看不到错误点)，在一定情况下增加了仿真次数，使得 debug 周期变长。

解决方案：提交前检查代码，确认代码正确性，减少潜在 bug。(或者直接写一个大区间，增加平台负担)

问题 2：修改软件代码时，需要重新运行硬件相关流程，造成时间与平台资源浪费。

解决方案：下载 bit_gen 的 Job artifacts，并修改脚本，跳过硬件相关流程。

三、 对讲义中思考题（如有）的理解和回答

本次实验无思考题

四、 在课后，你花费了大约 40 小时完成此次实验。

五、　对于此次实验的心得、感受和建议）

（一）实验心得、感受

1．数据通路与控制信号

在编写代码之前，先厘清所需通路与控制信号可使后续代码编写事半功倍。

2．代码编写

在已有代码的基础上进行修改可使代码编写速度大大加快。如本次实验中 custom_cpu.v 即是在 prj4 的基础上修改所得。

在修改时需要检查所有需要修改的地方，防止遗漏。如变量名修改，若存在遗漏。会导致相关信号出错。

参考以往编写代码经验，可以缩短代码编写周期。

3．debug

开始实验项目前应保证头脑清醒，前一天休息不足容易导致编写代码时低级错误频发（如笔误写错信号位宽等）。清醒的头脑可以减少无谓的错误，进而可使代码编写事半功倍。

在 push 代码前，trigger pipeline 前应当再检查一遍代码，减少潜在错误。看代码时间远远小于仿真时间，有利于减少 debug 时间。

（二）实验建议

希望早点发布。

（三）致谢

感谢楼持恒同学和我讨论流水线相关设计问题，并帮助进行部分bug解决。

感谢王嵩岳助教及时反馈我关于云平台的问题以及讨论上版运行出错的解决方案