中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929016　　姓名：李金明　专业：计算机科学与技术

实验序号：5.3　　实验名称：处理器性能增强设计

一、 关键代码及说明

本部分实验实现了五级流水 CPU，将指令执行拆分为 IF、ID、EX、MEM 和 WB 阶段，分别使用五个模块编写，最后在 custom_cpu.v 文件中进行连接。

1. State_IF.v 文件

a. 接口说明

```
9     input               clk,
10    input               rst,
11
12    //Instruction request channel
13    output reg [31:0]   PC,                 //同时传向ID阶段
14    output              Inst_Req_Valid,
15    input               Inst_Req_Ready,
16
17    //Instruction response channel
18    input      [31:0]   Instruction,
19    input               Inst_Valid,
20    output              Inst_Ready,
21    output reg [31:0]   Instruction_reg,    //存有Instruction值，传给ID以分析
22
23    input      [31:0]   branch_PC,          //跳转/分支PC目标值
24
25    output              complete_this,      //本阶段完成标志
26
27    input               fb_ex_branch,       //需要跳转/分支反馈信号
28    input               fb_mem,             //访存反馈信号
29
30    output     [31:0]   cpu_perf_cnt_2
```

b. 工作原理

在 IF 模块中，我使用了状态机进行工作，状态机第二段由以下代码给出：

```
49      always @ (*) begin
50          case (current_state)
51              `S_INIT: next_state = `S_IF;
52              `S_IF: begin
53                  if (Inst_Req_Ready) next_state = `S_IW;
54                  else next_state = `S_IF;
55              end
56              `S_IW: begin
57                  if (Inst_Valid) begin
58                      if (fb_ex_branch | clear_for_branch) next_state = `S_IF;
59                      else next_state = `S_COM;
60                  end
61                  else next_state = `S_IW;
62              end
63              `S_COM: begin
64                  if (fb_mem) next_state = `S_COM;
65                  else next_state = `S_INIT;
66              end
67              default: next_state = `S_INIT;
68          endcase
69      end
```

其相当于将原先多周期处理器中的 IF 阶段和 IW 阶段整合了进来。当收到运行阶段有跳转的反馈时，若模块正在等待指令，则不需要继续等待，直接跳回 IF 准备去取目标指令。若访存反馈信号拉高，流水线工作也需要暂停，体现在价将状态停留在 COM 状态

c. PC 值的更新

```
77      always @ (posedge clk) begin
78          if (rst) PC <= 32'b0;
79          else if (state_IW & (fb_ex_branch | clear_for_branch)) PC <= branch_PC;
80          else if (state_COM)
81              if (fb_ex_branch | clear_for_branch) PC <= branch_PC;
82              else if (~fb_mem) PC <= PC + 4;
83      end
```

由于 ID 和 EX 阶段都只需要 1 个周期，故在 IF 中的 IW 和 COM 阶段判断上一条指令是否有跳转即可。若无跳转，则 PC 值加四，否则更新为需要跳转的地址。

d. 工作结束标志

```
89      assign complete_this = state_COM;
```

当状态处于 COM 时，即说明此次取指完成，该模块工作完成。

2. State_ID.v 文件

a. 接口说明

```
21  module state_ID(
22      input               clk,
23      input               rst,
24
25      input               complete_pre,       //前序阶段完成信号
26      output reg          complete_this,      //本阶段完成信号
27
28      input      [31:0]   PC_input,           //前阶段传入PC值
29      output reg [31:0]   PC_output,          //向后阶段传出PC值
30
31      output reg [31:0]   branch_PC_reg,      //跳转/分支PC目标地址(若有)
32      input      [31:0]   Instruction_reg,    //传入待分析指令
33
34      //寄存器相关信号
35      input      [31:0]   RF_rdata1,
36      input      [31:0]   RF_rdata2,
37      output     [ 4:0]   RF_raddr1,
38      output     [ 4:0]   RF_raddr2,
39      output reg [ 4:0]   RF_waddr,           //传出寄存器写地址，待写回阶段使用
40
41      //向下一阶段传出寄存器读到的数据
42      output reg [31:0]   RF_rdata1_out,
43      output reg [31:0]   RF_rdata2_out,
44
45      output reg [19:0]   Inst_Decode,        //指令分析结果汇总(见138行)
46      output reg [31:0]   imm_r,              //立即数
47
48      input               fb_ex_branch,       //需要跳转/分支反馈信号
49      input               fb_mem,             //访存反馈信号
50
51      input      [31:0]   wb_from_ex,         //旁路输入(非load指令从ex阶段获取)
52      input      [31:0]   wb_from_load,       //旁路输入(load指令从mem阶段获取)
53
54      output     [31:0]   cpu_perf_cnt_1,
55      output     [31:0]   cpu_perf_cnt_4
56  );
```

b. 工作原理说明

该部分大部分都与多周期处理器中的译码过程相同。为了减少输入输出信号，我使用了 Inst_Decode 这样一个信号来存储译码结果，它的各位含义如下：

```
139     always @ (posedge clk) begin
140         if (complete_pre & ~fb_ex_branch & ~fb_mem)
141             Inst_Decode <= {
142                 R_Type, I_Type, calc, shift, I_Type_load, I_Type_jump, mul, //19-13
143                 S_Type, B_Type, U_Type, J_Type, AUIPC,  //12-8
144                 funct3, //7-5
145                 ALUop,  //4-2
146                 Shiftop //1-0
147             };
148     end
```

这样设计可以使得只用一个信号而传输大量的信息。

c. 工作结束标志

```
177     always @ (posedge clk) begin
178         if (rst) complete_this <= 0;
179         else if (~fb_mem) begin
180             if (complete_pre & ~fb_ex_branch)
181                 complete_this <= 1;
182             else complete_this <= 0;
183         end
184     end
```

由于 ID 模块最多只需运行 1 个周期，所以只要在前序模块（IF）完成工作且上一条指令不是跳转指令时，即可说明本模块完成工作，开启下一模块的工作。

3. State_EX.v 文件

a. 接口说明

```
3   module state_EX(
4       input               clk,
5       input               rst,
6
7       input               complete_pre,       //前一阶段完成信号
8       output reg          complete_this,      //本阶段完成信号
9
10      //寄存器相关信号
11      input       [31:0]  RF_rdata1,
12      input       [31:0]  RF_rdata2,
13      input       [ 4:0]  RF_waddr_in,
14      output reg  [ 4:0]  RF_waddr_out,
15
16      input       [19:0]  Inst_Decode,        //指令译码结果
17      input       [31:0]  imm,                //立即数
18      input       [31:0]  PC_input,           //输入PC
19      output reg  [31:0]  PC_output,          //输出PC
20
21      output              fb_ex_branch,       //需要跳转/分支反馈信号
22      input               fb_mem,             //访存反馈信号
23
24      output reg  [ 8:0]  mem_info_out,       //包含funct3[2], 确定load/store信号, strb值
25      output reg  [31:0]  Write_data_out,     //store写入内存数据/写回寄存器数据
26      output reg  [31:0]  mem_address_out     //store/load地址
27  );
```

b. 工作原理说明

EX 模块同样改动不大，与原先多周期处理器的 EX 阶段大同小异。需要说明的一个是该阶段产生跳转指令反馈信号，当检测到该条指令需要进行跳转操作时，拉高 fb_ex_branch，由此告诉 IF 和 ID 模块停止并重新工作。由以下代码

给出：

```
101     assign fb_ex_branch = (complete_pre) & (J_Type | I_Type_jump | (B_Type & (funct3[2] ^ funct3[0] ^ Zero)));
```

另一个需要说明的是 mem_info_out 信号，与 ID 模块中的 Inst_Decode 信号类似，它汇总了多个信号，具体各位含义如下：

```
125     always @ (posedge clk) begin
126         if (rst) mem_info_out <= 32'b0;
127         else if (complete_pre && ~fb_mem)
128             mem_info_out <= {funct3, S_Type, I_Type_load, Write_strb};
129     end
```

另外，为了节省输入输出信号，我还将 store 指令需要写入的数据以及写回寄存器的数据（除 load 指令外，此时还没有获取到内存数据），使用一个寄存器表示，节约了资源。

c. 工作结束标志

```
155     always @ (posedge clk) begin
156         if (rst) complete_this <= 0;
157         else if (!fb_mem) complete_this <= complete_pre;
158     end
```

与 ID 模块相同，该模块最多也只需要工作一个周期，若 ID 模块工作完成，且不存在访存需要暂停流水线的情况，便可说明本模块工作完成，从而进入下一个模块的工作。

4. State_MEM.v 文件

a. 接口说明

```
module state_MEM (
    input               clk,
    input               rst,

    input               complete_pre,       //前一阶段完成信号
    output reg          complete_this,      //本阶段完成信号

    input       [31:0]  PC_input,           //输入PC值
    output reg  [31:0]  PC_output,          //输出PC值

    input       [ 4:0]  RF_waddr_in,        //寄存器写地址(输入)
    input       [ 8:0]  mem_info_in,        //包含funct3[2], 确定load/store信号, strb值
    input       [31:0]  Write_data_in,      //store写入内存数据/写回寄存器数据(输入)
    output reg  [31:0]  Write_back_reg,     //写回寄存器数据(输出)
    input       [31:0]  mem_address_in,     //store/load地址

    output reg  [4:0]   RF_waddr_out,       //寄存器写地址(输出)

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

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

    output              fb_mem              //访存反馈信号
);
```

b. 工作原理说明

在 MEM 模块中，我整合了多周期 CPU 中的 ST、LD 和 RDW 状态，也使用了状态机，状态机第二段描述对应代码如下：

```
73      always @ (*) begin
74          case (current_state)
75              `S_INIT: begin
76                  if (~complete_pre) next_state = `S_INIT;
77                  else if (S_Type) next_state = `S_ST;
78                  else if (I_Type_load) next_state = `S_LD;
79                  else next_state = `S_INIT;
80              end
81              `S_ST: begin
82                  if (Mem_Req_Ready) next_state = `S_COM;
83                  else next_state = `S_ST;
84              end
85              `S_LD: begin
86                  if (Mem_Req_Ready) next_state = `S_RDW;
87                  else next_state = `S_LD;
88              end
89              `S_RDW: begin
90                  if (Read_data_Valid) next_state = `S_COM;
91                  else next_state = `S_RDW;
92              end
93              `S_COM: next_state = `S_INIT;
94              default next_state = `S_INIT;
95          endcase
96      end
```

c. 反馈信号的生成

在 MEM 模块中，没了防止不同指令引发的访存冲突，在访存等待内存返回请求应答以及发送数据时，要暂停流水线的工作，反馈信号的生成如下：

```
144     assign fb_mem = ~rst & (state_LD | state_RDW | state_ST);
```

d. 寄存器写相关数据

在 MEM 模块中，所有需要写回的指均已获取到需要写回的数据。根据 IO 信号说明，若指令不是 load 型，则直接为 Write_data_in，否则依照指令选择从内存中读到数据的片段作为写回数据。

```
114     always @ (posedge clk) begin
115         if (complete_pre & state_INIT & S_Type)
116             memory_data <= Write_data_in;
117         else if (state_RDW & Read_data_Valid)
118             Write_back_reg <= ({32{memory_byte & ~funct3[2]}} & {{24{lb_data[7]}}, lb_data}) |
119                               ({32{memory_half & ~funct3[2]}} & {{16{lh_data[15]}}, lh_data}) |
120                               ({32{memory_word & ~funct3[2]}} & Read_data) |
121                               ({32{memory_byte & funct3[2]}} & {{24{1'b0}}, lb_data}) |
122                               ({32{memory_half & funct3[2]}} & {{16{1'b0}}, lh_data});
123         else if (complete_pre & state_INIT & ~(S_Type | I_Type_load))
124             Write_back_reg <= Write_data_in;
125     end
```

其他部分与多周期 CPU 中的设计相差不大。

e. 工作结束标志

本阶段工作完成的标志根据不同类型的指令分类处理。对于访存指令，需要在结束本阶段访存工作后置 1；对于非访存指令，只需要在前一阶段完成工作就可置 1。

```
146     always @ (posedge clk) begin
147         if (rst) complete_this <= 0;
148         else if (state_INIT & complete_pre & (state_COM | (state_INIT & (~complete_pre | (~S_Type & ~I_Type_load)))) | state_COM) complete_this <= 1;
149         else complete_this <= 0;
150     end
```

5. State_WB.v 文件

a. 接口说明

```
3   module state_WB(
4       input                clk,
5       input                rst,
6
7       input        [31:0]  PC_input,        //输入PC值(待退役)
8       input                complete_pre,    //前序阶段完成信号
9       input        [ 4:0]  RF_waddr,        //寄存器写地址
10      input        [31:0]  RF_wdata,        //寄存器写数据
11
12      output               RF_wen,          //寄存器写使能
13      output  [69:0]  inst_retire      //退役相关信号
14  );
```

b. 工作原理说明

WB 模块工作非常简单，只需要注意提交指令即可。由于行为仿真只检查 RF_wen 值为 1 时的指令提交正确情况，且只比对一个周期，如果持续多个周期会产生错误，故指令提交代码如下：

```
17      assign RF_wen = (complete_pre & (|RF_waddr));
18      assign inst_retire = (complete_pre)? {RF_wen, RF_waddr, RF_wdata, PC_input} : 70'b0;
```

6. 各个模块的连接

```
112     reg_file RegFile(
113         .clk(clk),
114         .waddr(RF_waddr),
115         .raddr1(RF_raddr1),
116         .raddr2(RF_raddr2),
117         .wen(RF_wen),
118         .wdata(RF_wdata),
119         .rdata1(RF_rdata1),
120         .rdata2(RF_rdata2)
121     );

123     state_IF IF(
124         .clk(clk),
125         .rst(rst),
126
127         .PC(PC_12),
128         .Inst_Req_Valid(Inst_Req_Valid),
129         .Inst_Req_Ready(Inst_Req_Ready),
130
131         .Instruction(Instruction),
132         .Inst_Valid(Inst_Valid),
133         .Inst_Ready(Inst_Ready),
134         .Instruction_reg(Instruction_now),
135
136         .branch_PC(branch_PC),
137         .complete_this(complete_12),
138
139         .fb_ex_branch(fb_ex_branch),
140         .fb_mem(fb_mem),
141         .cpu_perf_cnt_2(cpu_perf_cnt_2)
142     );
143
144     assign PC = PC_12;
```

```
    state_ID ID(
        .clk(clk),
        .rst(rst),

        .complete_pre(complete_12),
        .complete_this(complete_23),

        .PC_input(PC_12),
        .PC_output(PC_23),

        .branch_PC_reg(branch_PC),
        .Instruction_reg(Instruction_now),

        .RF_rdata1(RF_rdata1),
        .RF_rdata2(RF_rdata2),
        .RF_raddr1(RF_raddr1),
        .RF_raddr2(RF_raddr2),
        .RF_waddr(RF_waddr_23),
        .RF_rdata1_out(RF_rdata1_23),
        .RF_rdata2_out(RF_rdata2_23),

        .Inst_Decode(Inst_Decode),
        .imm_r(imm),

        .fb_ex_branch(fb_ex_branch),
        .fb_mem(fb_mem),

        .wb_from_ex(wb_from_ex),
        .wb_from_load(wb_from_load),

        .cpu_perf_cnt_1(cpu_perf_cnt_1),
        .cpu_perf_cnt_4(cpu_perf_cnt_4)
    );

    state_EX EX(
        .clk(clk),
        .rst(rst),

        .complete_pre(complete_23),
        .complete_this(complete_34),

        .RF_rdata1(RF_rdata1_23),
        .RF_rdata2(RF_rdata2_23),
        .RF_waddr_in(RF_waddr_23),
        .RF_waddr_out(RF_waddr_34),

        .Inst_Decode(Inst_Decode),
        .imm(imm),

        .PC_input(PC_23),
        .PC_output(PC_34),

        .fb_ex_branch(fb_ex_branch),
        .fb_mem(fb_mem),

        .mem_info_out(mem_info),
        .Write_data_out(Write_data_34),
        .mem_address_out(mem_address)
    );
```

```
206     assign wb_from_ex = Write_data_34;
207
208     state_MEM MEM(
209         .clk(clk),
210         .rst(rst),
211
212         .complete_pre(complete_34),
213         .complete_this(complete_45),
214
215         .PC_input(PC_34),
216         .PC_output(PC_45),
217
218         .RF_waddr_in(RF_waddr_34),
219         .mem_info_in(mem_info),
220         .Write_data_in(Write_data_34),
221         .Write_back_reg(RF_wdata),
222         .mem_address_in(mem_address),
223
224         .RF_waddr_out(RF_waddr),
225
226         .Address(Address),
227         .MemWrite(MemWrite),
228         .Write_data(Write_data),
229         .Write_strb(Write_strb),
230         .MemRead(MemRead),
231         .Mem_Req_Ready(Mem_Req_Ready),
232
233         .Read_data(Read_data),
234         .Read_data_Valid(Read_data_Valid),
235         .Read_data_Ready(Read_data_Ready),
236
237         .fb_mem(fb_mem)
238     );
```

```
240     assign wb_from_load = RF_wdata;
241
242
243     state_WB WB(
244         .clk(clk),
245         .rst(rst),
246
247         .complete_pre(complete_45),
248
249         .PC_input(PC_45),
250         .RF_waddr(RF_waddr),
251         .RF_wdata(RF_wdata),
252
253         .RF_wen(RF_wen),
254         .inst_retire(inst_retire)
255     );
```

需要说明的首先是 PC 值，PC 值与 IF 模块传给 ID 模块的 PC 值相同，同步更新。另外要说明的是与寄存器堆相连的信号，读相关信号与 ID 模块相连，写相关信号中，写地址和写数据由 MEM 模块与寄存器堆相连，写使能信号则由 WB 模块与寄存器堆相连。其他的 IO 信号则没什么改变，在对应的模块中进行

连接。

7. 流水线 CPU 性能比较

对比设计的多周期 CPU，我设计的流水线 CPU 性能提升有限。对比相同的程序，在一些程序能运行的比多周期 CPU 快，以 15pz 为例：

```
49  [15pz] A* 15-puzzle search: * Passed.
50  Cycles: 452180695
```

图 1 程序 15pz 在流水线 CPU 运行时间

```
51  [15pz] A* 15-puzzle search: * Passed.
52  Cycles: 523945040
```

图 2 程序 15pz 在多周期 CPU 运行时间

可以看出在该程序中运行效率提上了 10%以上。

而在一些程序中，流水线 CPU 反而运行速度更慢，以 ssort 为例

```
249  [ssort] Suffix sort: * Passed.
250  Cycles: 49438955
```

图 3 程序 ssort 在流水线 CPU 运行时间

```
267  [ssort] Suffix sort: * Passed.
268  Cycles: 44719087
```

图 4 程序 ssort 在多周期 CPU 运行时间

我认为主要问题在处理跳转指令上，我完全按照提取地址上相邻的下一条指令进行处理，面对跳转或者分支指令时会浪费大量的时间资源。

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. 在需要暂停流水线的情况中不知道怎么设计。

在流水线 CPU 设计中，遇到分支或跳转指令时，需要清空已经进行的工作；当正在访存时，为了避免冲突，也需要暂停流水线。这里我分类讨论，在 IF 模块中引入了状态机，这是因为 IF 模块也需要访存，不是一个周期可以完成，思路即在需要进行跳转时，重新进行取值操作。而 ID 和 EX 阶段正常工作均只需

要一个周期，故它们面对暂停或重新取值（仅 ID 模块会遇到），只需要保持完成信号为 0，将流水线“卡住”，不向下一步进行即可

三、 在课后，你花费了大约 40 小时完成此次实验?

四、 对于此次实验的心得、感受和建议

这部分实验难度较大。主要是设计上较为复杂，有一种刚接触设计 CPU 的感觉。这里几乎需要从头开始，将各个模块的工作划分出去，然后需要有效连接各个模块，同时还要保证各个模块间的正常工作并尽量追求高效率。虽然大部分代码可以仅在原先的多周期 CPU 稍加修改完成，但由于工作不在一个模块中完成，以往仅用 wire 传输的数据现在需要通过模块间的连接传输，需要较为精巧的设计各个模块的 IO 接口，减少无用信号，尽量合并有用信号。因此这个实验难度较大主要在于工作量较大，且设计很自由，不同人可以采用不同的想法，需要自己琢磨出更为高效的工作方式。

感谢主讲老师及各位助教老师的讲解，让我深入了解到了流水线 CPU 的工作方式，在理论课之外更深地了解了流水线 CPU，意识到课上给出的流程图仅是一种理想化模型，实际上设计还有更多的问题需要考虑。