中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 薛翼舟 学号 2023K8009929044 专业 计算机科学与技术

实验项目编号 3 实验名称 定制 MIPS 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。

一、 逻辑电路结构与仿真波形的截图及说明(比如 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

基本上就是对基于 MIPS 指令集的多周期 CPU 进行 risc-v 指令集的修改, 而且由于 risc-v 指令集的指令相比 MIPS 更加规整, 加上之前写 MIPS-cpu 的经验, 本实验的难度不大. 下面就按照代码的顺序进行讲解

1.1 状态机部分

risc-v 多周期处理器的装体积的状态和 mips 相同, 区别在于执行阶段

```
EX: begin
                if(opcode==`B_type) begin
                        next_state      =       IF;
                end
                else if (opcode==`S_type) begin
                        next_state      =       ST;
                end
                else if (opcode==`I_type_l) begin
                        next_state      =       LD;
                end
                else begin
                        next_state      =       WB;
                end
        end
```

可以直接根据指令的类型来跳转, 相比 mips 更加简单

1.2 指令分段 (立即数还原)

risc-v 指令集的一大特点就是器指令的规整型, 这给 cpu 的编写增加了极大程度上的便利, 就比如说 func3 段, 对于不同的指令, 比如 R-type 指令的 ADD 和 I-type 指令的 ADDI, 它们的 func3 段都是 000, 这就使得我们可以直接使用 func3 段来判断指令的类型, 而不需要再去判断 opcode 段, 这就使得代码更加简洁.

不过, 这也就导致 risc-v 指令中的立即数会拆开分散在指令中的各段, 我们在译码阶段需要根据不同的指令将这些立即数进行还原, 之后的 EX 阶段我们根据各个指令的需求来分别选取要用的立即数即可

```
wire [31:0]U_imm = {Instruction_current[31:12],{12{1'b0}}};
wire [31:0]I_imm = {{20{Instruction_current[31]}},Instruction_current[31:20]};
wire [31:0]S_imm = {{20{Instruction_current[31]}},Instruction_current[31:25],Instruction_current[11:7]};
wire [31:0]B_imm = {{20{Instruction_current[31]}},Instruction_current[7],Instruction_current[30:25],Instruction_current[11:8],1'b0};
wire [31:0]J_imm = {{12{Instruction_current[31]}},Instruction_current[19:12],Instruction_current[20],Instruction_current[30:25],Instruction_current[24:21],1'b0};
```

1.3 译码部分的其他适应性修改

译码部分其他的适应性修改并不多, 首先就是两个计算机选择信号 ALUop 和 Shifterop, 由于我们前面提到过的 risc-v 指令集的规整性, 我们发现这两部分要比 mips 规整很多

```
assign ALUop =  (opcode==`I_type_l || opcode==`S_type)?          `ADD:
                (opcode==`B_type && func[2:1]==2'b00)?           `SUB:
                (opcode==`B_type && func[2:1]==2'b10)?           `SLT:
                (opcode==`B_type && func[2:1]==2'b11)?           `SLTU:
                (opcode==`R_type && func_r==`SUBr)?              `SUB:
                (func==3'b000)?                                  `ADD:
                (func==3'b010)?                                  `SLT:
                (func==3'b011)?                                  `SLTU:
                (func==3'b100)?                                  `XOR:
                (func==3'b110)?                                  `OR:
                (func==3'b111)?                                  `AND:
                                                                 `OR;
```

我们只需要将几中特殊的指令, 比如分支指令和访存指令单独判断, 剩下的我们可以直接用 func3 段来直接判断 ALU 操作类型, 同样的, 对于 Shifterop

```
assign Shiftop = (func==3'b001)?        2'b00:  //SLL
                 (func_r==4'b0101)?     2'b10:  //SRL
                 (func_r==4'b1101)?     2'b11:  //SRA
                                        2'b01;
```

在 Shifterop, 这种对比就更加明显了, 之后对于各种控制信号, 由于 risc-v 指令对于指令的分类更加细化, 于是我们可以直接根据 opcode 来简单地判定,

```
//寄存器堆写使能信号
assign  RF_wen_i  =
                (opcode==`AUIPC || opcode==`JAL || opcode==`JALR)?      1:
                (opcode==`I_type_l || opcode==`LUI)?                    1:
                (opcode==`I_type_c || opcode==`R_type)?                 1:
                                                                        0;
//分支控制信号
assign Branch = (opcode==`B_type);

```

```
//跳转控制信号
assign Jump    = (opcode==`JAL || opcode==`JALR);
```

另外, 比较特别的是 risc-v 集中的分支和跳转指令和 mips 中略有不同, 根据指令手册来对着编写即可, 代码如下

```
wire [31:0]Jump_r      = (rdata1 + I_imm) & 32'hfffffffe;
assign PC_Jump   =        (opcode==`JAL)? PC_reg + J_imm:
                                          Jump_r;
```

1.4 访存部分

在 EX 阶段基本上没有什么修改, 我们直接来看访存部分, risc-v 指令集并没有 mips 指令集中令人恶心的非对齐加载与存储, 极大程度上化简了代码, 其他部分没什么区别, 值得注意的是 AUIPC 指令是我们 mips 指令中所没有的, 它的作用是将旧的 PC 加上立即数然后存入目标寄存器当中, 但值得注意的是还是要根据 Address 的低两位来决定要写回的数据

```
assign RF_wdata  =      (opcode==`JAL ||opcode==`JALR)?         old_PC + 4:
                        (opcode==`AUIPC)?                       old_PC + U_imm:
                        (opcode==`LUI)?                         U_imm:
                        (opcode==`I_type_1)?
                        ((func[1:0]==2'b00)?    ((eff==2'b00)?  ((func[2]==0)?  {{24{read_byte_0[7]}},read_byte_0} :    {{24{1'b0}},read_byte_0}):
                                                 (eff==2'b01)?  ((func[2]==0)?  {{24{read_byte_1[7]}},read_byte_1} :    {{24{1'b0}},read_byte_1}):
                                                 (eff==2'b10)?  ((func[2]==0)?  {{24{read_byte_2[7]}},read_byte_2} :    {{24{1'b0}},read_byte_2}):
                                                                ((func[2]==0)?  {{24{read_byte_3[7]}},read_byte_3} :    {{24{1'b0}},read_byte_3}) ):
                         (func[1:0]==2'b01)?    ((eff==2'b00)?  ((func[2]==0)?  {{24{read_byte_1[7]}},Read_data_current [15:0]} : {{24{1'b0}},Read_data_current [15:0]}):
                                                                ((func[2]==0)?  {{24{read_byte_3[7]}},Read_data_current [31:16]} : {{24{1'b0}},Read_data_current [31:16]})):
                         Read_data_current
                        ):
                        Result;
```

```
assign  Write_strb      =       (func==3'b000)? ((eff==2'b00)?  4'b0001:
                                                 (eff==2'b01)?  4'b0010:
                                                 (eff==2'b10)?  4'b0100:
                                                                4'b1000):
                                (func==3'b001)? ((eff==2'b00)?  4'b0011:
                                                                4'b1100):
                                                                4'b1111;
```

```
assign Write_data    =          (func==3'b000)? ((eff==2'b00)?  {{24{1'b0}},reg_byte_0}:
                                                 (eff==2'b01)?  {{16{1'b0}},reg_byte_0,{8{1'b0}}}:
                                                 (eff==2'b10)?  {{8{1'b0}},reg_byte_0,{16{1'b0}}}:
                                                                {reg_byte_0,{24{1'b0}}}):
                                (func==3'b001)? ((eff==2'b00)?  {{16{1'b0}},reg_byte_1,reg_byte_0}:
                                                                {reg_byte_1,reg_byte_0,{16{1'b0}}}):
                                                                rdata2;
```

二、 性能计数器部分

为了比对 mips 和 risc-v 这两个指令集下 cpu 的性能表现, 我添加了两个性能计数器, 分别是指令计数器以及访存计数器, 分别用于计算指令数和访存次数, 用于比较性能表现, 代码与之前的时钟周期计数器基本相同, 在此不多赘述, 详见代码

2.1 RTL 代码

```
reg [31:0] ins_cnt;

always @(posedge clk) begin
        if(rst) begin
                ins_cnt <= 32'd0;
        end
        else if (Inst_Ready && Inst_Valid )begin
                ins_cnt <= ins_cnt + 32'd1;
        end
        else begin
                ins_cnt <= ins_cnt;
        end
end

assign cpu_perf_cnt_1 = ins_cnt;

reg [31:0] mem_cnt;

always @(posedge clk) begin
        if(rst) begin
                mem_cnt <= 32'd0;
        end
        else if (Read_data_Ready && Read_data_Valid)begin
                mem_cnt <= mem_cnt + 32'd1;
        end
        else begin
                mem_cnt <= mem_cnt;
        end
end

assign cpu_perf_cnt_2 = mem_cnt;
```

2.2 C 语言部分

我一开始所写的 C 语言部分时在复制粘贴的过程中出现了忘了修改函数名的低级错误, 导致三个结果都出现了意料之外的错误

```
unsigned long _upins() {
  volatile unsigned long* ins_cnt = (volatile unsigned long*) 0x60010008;
  unsigned long ins = *ins_cnt;
  return ins;
}

unsigned long _upmem() {
```

```
  volatile unsigned long* mem_cnt = (volatile unsigned long*) 0x60011000;
  unsigned long mem = *mem_cnt;
  return mem;
}

void bench_done(Result *res) {
  // TODO [COD]
  //  Add postprocess code, record performance counters' current states.
  res->msec = _uptime() - res->msec;
  res->ins = _upins() - res->ins;
  res->mem = _upmem() - res->mem;
}
```

三、 实验过程中遇到的问题、对问题的思考过程及解决方法

在本实验中, 我一开始在编写 RF_wdata 时, 忘记了考虑 lb 指令中, 到底是写回第几字节的问题, 因而出现了意料之外的错误, 于是我溯源了很久, 终于在这个报错位置很之前的位置发现了问题所在, 于是我在 RF_wdata 中添加了对 Address 的低两位的判断. 不过从这次调试中我发现金标准的 Read_data 信号是自动低两位对齐的

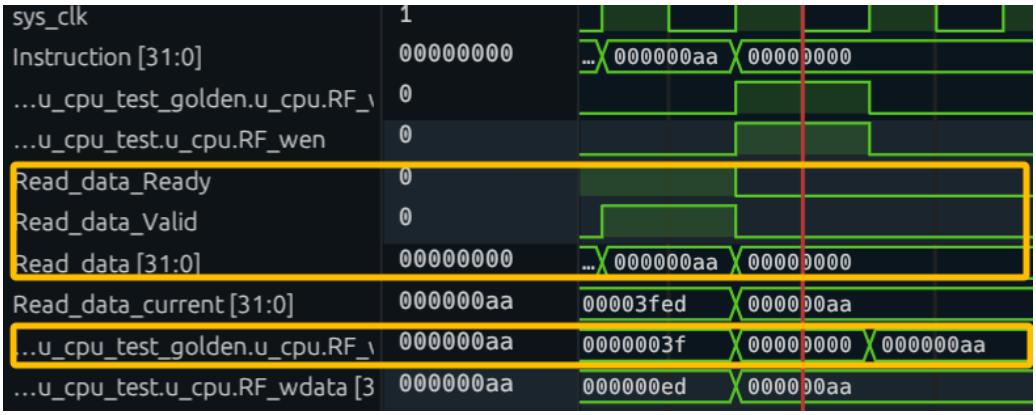

图 1: 错误出现的位置

如图所示, 我上午 RF_wdata 是 aa, 而金标准是 00, 但是两个 Read_data 是相同的, 所以问题出现的原因是我所选取的 Read_data 的字节是不对的

四、 思考题

我们根据我所写的三个性能计数器来进行 risc-v 和 mips 处理器的性能比较, 如下是 microbench 测试集的统计结果

15pz	周期数	指令数	访存次数	时间(ms)
risc-v	523381381	5224461	2109987	5248.78
mips	527927996	5287713	2112785	5295.22

bf	周期数	指令数	访存次数	时间(ms)
risc-v	38811933	452834	94502	688.07
mips	46343304	559051	94500	767.63

dinic	周期数	指令数	访存次数	时间(ms)
risc-v	1475814	16671	4031	6616.06
mips	1703609	19328	4627	6732.11

fib	周期数	指令数	访存次数	时间(ms)
risc-v	181088216	2549505	4315	1826.90
mips	179408048	2525724	4403	1810.67

md5	周期数	指令数	访存次数	时间(ms)
risc-v	384681	4895	534	201.73
mips	404156	5229	471	253.89

qsort	周期数	指令数	访存次数	时间(ms)
risc-v	782725	9460	1513	956.13
mips	704017	8341	1513	1211.79

queen	周期数	指令数	访存次数	时间(ms)
risc-v	6894575	81470	14409	81.93
mips	6849403	80858	14409	82.07

sieve	周期数	指令数	访存次数	时间(ms)
risc-v	744622	10175	334	21.63
mips	1191835	16480	336	26.87

ssort	周期数	指令数	访存次数	时间(ms)
risc-v	44731952	619025	11280	1346.68
mips	52508142	728291	11587	1654.13

图 2: 各个统计结果

我们初步看出, 基本上所有的 9 个程序都是 risc-v 比 mips 要快, 并且指令数和, 访存次数要少, 以下是平均的结果. 同时我们可以看出在 sieve 程序中的优化效果最好, 达到了将近 20%

平均值				
	平均值	指令数	访存次数	时间(ms)
risc-v	87,820,364	1,013,377	251,656	1,774
mips	90,590,612	1,046,226	251,736	1,803

图 3: 9 个测试程序统计结果

从平均的结果我们可以得出结论, risc-v 指令集完成一个程序所用的指令数和访存次数都少于 mips 指令集, 除了个别例外 (其实例外中, 例如 fib, risc-v 和 mip 的差据也很小), risc-v 指令集所用的时间都小于 mips 指令集

五、 实验所耗时间

在课后，你花费了大约____6____ 小时完成此次实验。