

中国科学院大学计算机组成原理实验课

实 验 报 告

学号: 2018K8009929046 姓名: 何咏哲 专业: 计算机科学与技术

实验序号: 3 实验名称: 内存及外设通路设计与处理器性能评估

注 1: 请在实验项目个人本地仓库中创建顶层目录 doc。撰写此 Word 格式实验报告后以 PDF 格式保存在 doc 目录下。文件命名规则: 学号-prjN.pdf, 其中学号中的字母“K”为大写,“-”为英文连字符,“prj”和后缀名“pdf”为小写,“N”为 1 至 4 的阿拉伯数字。例如: 2018K8009929000-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: 学号-prj5-projectname.pdf, 例如: 2018K8009929000-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 doc 目录下的实验报告 PDF 文件添加到本地仓库,并通过 git push 推送提交。

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(比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等)

图 1: PC 值在 EX 阶段进行更新

```
129 always @(posedge clk) begin
130     if(rst) begin
131         PC <= 32'b0;
132     end else if(state==EX) begin
133         PC <= Jump ? JumpAddress:((Branch&Branch_eff)?branch_PC:PC_4);
134     end else begin
135         PC <= PC;
136     end
137 end
```

图 2: 使用 originPC 来记录 EX 阶段更新之前的 PC 值。

```
139 always @(posedge clk) begin
140     if(state==IF) begin
141         originPC <= PC;
142     end else begin
143         originPC <= originPC;
144     end
145 end
```

图 3: 三段式中的第一段,描述当前状态的变化。引入了复位态,目的是较简便地实现握手信号的控制,避免死锁。

```

121 always @(posedge clk) begin
122     if(rst) begin
123         state <= RSTSTATE;
124     end else begin
125         state <= next_state;
126     end
127 end

```

图 4：三段式中的第二段，描述下一状态转变。其中若处于复位态，则下一状态无条件为 IF。

```

148 always @(*) begin
149     case(state)
150         RSTSTATE: next_state = IF;
151         IF: if(Inst_Req_Ack) begin
152             next_state = IW;
153         end else begin
154             next_state = IF;
155         end
156         IW: if(Inst_Valid) begin
157             next_state = ID;
158         end else begin
159             next_state = IW;
160         end
161         ID: next_state = EX;
162         EX: case(opcode)
163             `REGIMM: if(func[4:1]==`BRANCH_Z) begin
164                 next_state = IF;
165             end else begin
166                 next_state = WB;
167             end
168             default: if(opcode[5:2]==`BRANCH || opcode[5:1]==`JUMP) begin
169                 next_state = IF;
170             end else if(opcode[5:3]==`LOAD) begin
171                 next_state = LD;
172             end else if(opcode[5:3]==`STORE) begin
173                 next_state = ST;
174             end else begin
175                 next_state = WB;
176             end
177         endcase
178         LD: if(Mem_Req_Ack) begin
179             next_state = RDW;

```

```

178         LD: if(Mem_Req_Ack) begin
179             next_state = RDW;
180         end else begin
181             next_state = LD;
182         end
183         ST: if(Mem_Req_Ack) begin
184             next_state = IF;
185         end else begin
186             next_state = ST;
187         end
188         WB: next_state = IF;
189         RDW: if(Read_data_Valid) begin
190             next_state = WB;
191         end else begin
192             next_state = RDW;
193         end
194         default: next_state = state;
195     endcase
196 end

```

图 5: 状态机的描述使用 parameter, 并且采用了独热码(one-hot)。

```
103 //hand_shaking
104 reg [31:0] valid_Instruction;
105 reg [31:0] valid_Read_data;
106 // To describe the state of FSM
107 // use one-hot code
108 parameter IF = 9'b0_0000_0001;
109 parameter IW = 9'b0_0000_0010;
110 parameter ID = 9'b0_0000_0100;
111 parameter EX = 9'b0_0000_1000;
112 parameter LD = 9'b0_0001_0000;
113 parameter ST = 9'b0_0010_0000;
114 parameter WB = 9'b0_0100_0000;
115 parameter RDW = 9'b0_1000_0000;
116 parameter RSTSTATE = 9'b1_0000_0000;
117 reg [8:0] state;
118 reg [8:0] next_state;
119 reg [31:0] originPC;
```

图 6: 用于记录真正有效的 instruction 和 read_data。这两个变量使用的是寄存器类型, 因为需要让其在不更新时保持原值, 使用 reg 类型可以避免组合环。

```
274 always @(posedge clk) begin
275     valid_Instruction <= (Inst_Ack && Inst_Valid)? Instruction : valid_Instruction;
276 end
277
278 always @(posedge clk) begin
279     valid_Read_data <= (Read_data_Valid && Read_data_Ack)? Read_data : valid_Read_data;
280 end
```

图 7: 握手信号的实现。其中 MemRead 和 MemWrite 在 prj2 中也出现过, 此处为根据 prj3 实验要求重新改造的结果。

```
282 assign Inst_Ack = (state==RSTSTATE || state==IW)?1:0;
283 assign Inst_Req_Valid = (state == IF)?1:0;
284 assign MemRead = (state == LD)?1:0;
285 assign MemWrite = (state == ST)?1:0;
286 assign Read_data_Ack = (state == RDW)?1:0;
```

图 8: 外设控制器访问部分的 puts() 函数的实现。由于 UART 是一个指向 unsigned int 类型的指针, 而 TX_FIFO 和 STAT_FIFO 在 UART 的基础上分别偏离了 4 个和 8 个字节。因此有两种实现访问偏移地址的方法, 一种是先将 UART 强制类型转换为 char* 类型, 然后直接加上宏定义的偏移量; 另一种是通过 UART 的地址加上 (宏定义的偏移量/4) 来访问。我采取的是前者。并且使用了 volatile 关键字, 避免编译器的优化使得内存没有被正常访问, 否则会导致程序异常运行。

```

249 int
250 puts(const char *s)
251 {
252     //TODO: Add your driver code here
253     int i = 0;
254     while(s[i]!='\0'){
255         while((*volatile unsigned int*)((char*)uart + UART_STATUS)) & UART_TX_FIFO_FULL)
256             ;
257         volatile unsigned int* uart_tx_fifo_write = (volatile unsigned int*)((char*)uart + UART_TX_FIFO);
258         *uart_tx_fifo_write = s[i];
259         i++;
260     }
261     return i;
262 }

```

图 9：性能计数器在 cpu 中的实现。一共实现了 6 个寄存器，分别记录了处理器运行周期、完成执行的指令数、访存指令数、访存延时、跳转发生/不发生指令数等性能指标。其中访存延时计数器根据助教老师的建议进行了修改，具体在下部分进行介绍。

```

198 // The number of clock cycle
199 reg [31:0] Cycle_cnt;
200 always @(posedge clk) begin
201     if(rst) begin
202         Cycle_cnt <= 32'b0;
203     end else begin
204         Cycle_cnt <= Cycle_cnt + 31'b1;
205     end
206 end
207 assign mips_perf_cnt_0 = Cycle_cnt;

```

```

209 // The number of obtained instruction
210 reg [31:0] Inst_cnt;
211 always @(posedge clk) begin
212     if(rst) begin
213         Inst_cnt <= 32'b0;
214     end else if(state == IF) begin
215         Inst_cnt <= Inst_cnt + 31'b1;
216     end else begin
217         Inst_cnt <= Inst_cnt;
218     end
219 end
220 assign mips_perf_cnt_1 = Inst_cnt;

```

```

222 // The number of Visiting Memory
223 reg [31:0] Memory_cnt;
224 always @(posedge clk) begin
225     if(rst) begin
226         Memory_cnt <= 32'b0;
227     end else if((state == LD || state == ST) && Mem_Req_Ack) begin
228         Memory_cnt <= Memory_cnt + 31'b1;
229     end else begin
230         Memory_cnt <= Memory_cnt;
231     end
232 end
233 assign mips_perf_cnt_2 = Memory_cnt;

235 // The delay time of Visiting Memory Stage
236 reg [31:0] Delay_cnt;
237 always @(posedge clk) begin
238     if(rst) begin
239         Delay_cnt <= 32'b0;
240     end else if((state==LD || state==ST) && !Mem_Req_Ack) || (state==RDW && !Read_data_Valid)) begin
241         Delay_cnt <= Delay_cnt + 31'b1;
242     end else begin
243         Delay_cnt <= Delay_cnt;
244     end
245 end
246 assign mips_perf_cnt_3 = Delay_cnt;

248 // The number of really branch while the branch instruction
249 reg [31:0] Branch_cnt;
250 always @(posedge clk) begin
251     if(rst) begin
252         Branch_cnt <= 32'b0;
253     end else if(state==EX && Branch && Branch_eff) begin
254         Branch_cnt <= Branch_cnt + 31'b1;
255     end else begin
256         Branch_cnt <= Branch_cnt;
257     end
258 end
259 assign mips_perf_cnt_4 = Branch_cnt;

261 // The number of NOT really branch while the branch instruction
262 reg [31:0] nBranch_cnt;
263 always @(posedge clk) begin
264     if(rst) begin
265         nBranch_cnt <= 32'b0;
266     end else if(state==EX && Branch && !Branch_eff) begin
267         nBranch_cnt <= nBranch_cnt + 31'b1;
268     end else begin
269         nBranch_cnt <= nBranch_cnt;
270     end
271 end
272 assign mips_perf_cnt_5 = nBranch_cnt;

```

图 10：性能处理器物理地址的宏定义，以及对结构体的域进行了扩展，以记录全部六个性能指标的数值。各个域的含义与上图中性能计数器按顺序一一对应。


```

6 #define mips_perf_cnt_0 0x40020000
7 #define mips_perf_cnt_1 0x40020008
8 #define mips_perf_cnt_2 0x40021000
9 #define mips_perf_cnt_3 0x40021008
10 #define mips_perf_cnt_4 0x40022000
11 #define mips_perf_cnt_5 0x40022008
12
13 typedef struct Result {
14     int pass;
15     unsigned long msec;
16     unsigned long inst;
17     unsigned long memory;
18     unsigned long delay;
19     unsigned long branch;
20     unsigned long nbranch;
21 } Result;

```

图 11：访问各个性能计数器的函数，实现顺序与图 9 一致。

```

24 unsigned long _uptime() {
25     volatile unsigned long* Cycle_cnt = (unsigned long*)mips_perf_cnt_0;
26     return *Cycle_cnt;
27 }
28
29 unsigned long _upInst() {
30     volatile unsigned long* Inst_cnt = (unsigned long*)mips_perf_cnt_1;
31     return *Inst_cnt;
32 }
33
34 unsigned long _upMem() {
35     volatile unsigned long* Memory_cnt = (unsigned long*)mips_perf_cnt_2;
36     return *Memory_cnt;
37 }
38
39 unsigned long _upDelay() {
40     volatile unsigned long* Delay_cnt = (unsigned long*)mips_perf_cnt_3;
41     return *Delay_cnt;
42 }
43
44 unsigned long _upBranch() {
45     volatile unsigned long* Branch_cnt = (unsigned long*)mips_perf_cnt_4;
46     return *Branch_cnt;
47 }
48
49 unsigned long _upnBranch() {
50     volatile unsigned long* nBranch_cnt = (unsigned long*)mips_perf_cnt_5;
51     return *nBranch_cnt;
52 }

```

图 12：记录开始的数据以及结果数据的函数。

```

54 static void bench_prepare(Result *res) {
55     res->msec      = _uptime();
56     res->inst      = _upInst();
57     res->memory    = _upMem();
58     res->delay     = _upDelay();
59     res->branch    = _upBranch();
60     res->nbranch   = _upnBranch();
61 }
62
63 static void bench_done(Result *res) {
64     res->msec      = _uptime()      - res->msec;
65     res->inst      = _upInst()      - res->inst;
66     res->memory    = _upMem()       - res->memory;
67     res->delay     = _upDelay()     - res->delay;
68     res->branch    = _upBranch()    - res->branch;
69     res->nbranch   = _upnBranch()   - res->nbranch;
70 }

```

图 13：对记录结果进行输出。顺序与上文一致。

```

139     for (int i = 0; i < REPEAT; i++) {
140         Result res;
141         run_once(bench, &res);
142         printk(res.pass ? "*" : "X");
143         succ &= res.pass;
144         if (res.msec < msec) msec = res.msec;
145         if (res.inst < inst) inst = res.inst;
146         if (res.memory < memory) memory = res.memory;
147         if (res.delay < delay) delay = res.delay;
148         if (res.branch < branch) branch = res.branch;
149         if (res.nbranch < nbranch) nbranch = res.nbranch;
150     }
151
152     if (succ) printk(" Passed.\n");
153     else printk(" Failed.\n");
154
155     pass &= succ;
156
157     printk("Time Cycle: %u \n", msec);
158     printk("Instruction Number: %u \n", inst);
159     printk("Memory Visit: %u \n", memory);
160     printk("Delay Time: %u \n", delay);
161     printk("Branch Number: %u \n", branch);
162     printk("Not Branch Number: %u \n", nbranch);
163

```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，仿真、本地上板及云平台调试过程中的难点等）

问题 1：不太清楚三段式的写法

解决：通过查阅资料弄清楚了。

问题 2：状态机的描述过程中，对于“复位信号未释放时，需保证 MIPS 处

理器不会发出取指访存请求”这一要求的实现感到困惑。

解决：一开始思考能否通过增加一个复位状态来实现这一功能。首先对复位状态与其他状态的关系不是很清楚，为了方便就通过 rst 信号来控制组合逻辑里的状态转移关系，经常老师提醒后知道这样是不符合规范的，于是在整理清楚各个状态之间的关系后发现，当前状态为复位态时，可以让下一状态无条件赋值为 IF，这样即使 rst 一直拉高，在对 state 的赋值中 state 也总是赋值为复位态，符合设计要求。

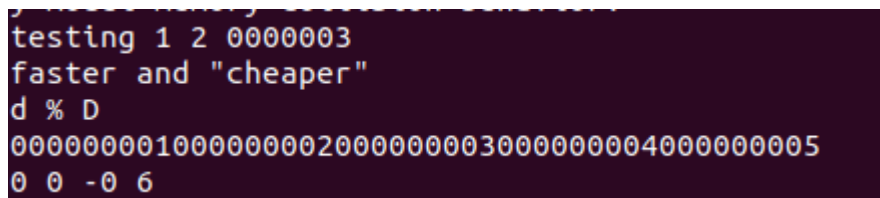
问题 3：PC 值的更新问题。PC 应在 EX 阶段进行更新，然而 JALR 等指令在写入寄存器时需要用到原来的 PC 值，导致错误。

解决：通过新增一个 originPC 寄存器来保存跳转之前的 PC 值，具体做法是让 originPC 在 IF 阶段更新，这样就避免了 PC 更新的问题。

问题 4：在仔细检查完 mips_cpu 没有问题之后，上板依旧不通过。

解决：把所有需要修改的地方仔细排查后，发现问题可能出现在之前写的 alu 和 reg_file 上面。发现 alu 中 sra 的操作不可综合，之前使用的是 \$signed() 转换以及 >>> 算术右移操作，修改之后变为 {{32{A[31]}}}, A >> B。之前 regfile 中 if-else 条件语句不完善，以及 0 号寄存器和其他寄存器放在同一个 always 块里面赋值。修改之后完善了分支，不同寄存器在不同的 always 块里面赋值。于是上板得以通过。另外在帮助其他同学 debug 的过程中，也发现了同样的问题，即 mips_cpu 没有问题，但是结果就是不正确，在修改完上述两个功能部件之后就可以通过了。

问题 5：uart 打印不全，如下图所示。



解决：经同学提醒可能是 BGTZ 出了问题，后来发现果然是在增加 BGTZ 判断时括号匹配的问题，修改完之后就能正常打印了。据我了解，其他遇到相同情况的同学也是因为 BGTZ 有点问题。

问题 6：一开始不理解 STAT_FIFO 寄存器的作用，认为如果 FIFO 队列已满的话，按照 PPT 上面的流程图，程序会进入死循环，并通过这种方式来报错。

解决：通过同学的帮助，知道了在程序工作过程中，FIFO 队列的状态可能由满变为空，因此我们需要实现的不是进入死循环，而是等待，等待可以写入的时候应该继续写入。而在判断 FIFO 是否满的实现时也遇到了一点问题，具体与思考题中 volatile 关键字的作用有关。如果不加 volatile 关键字，编译器在优化时可能不会每次都向真实内存读取数据，导致程序不按照预期的功能进行。

问题 7：性能计数器的实现有细微的问题。

解决：在 prj3 的验收过程中，陈欲晓老师指出我的访存等待时间计数器实现不太对，因为一个完整的内存读过程包括 LD 和 RDW，不仅需要记录 LD 状态等

待 Mem_Req_Ack 信号的时钟,也需要记录 RDW 阶段等待 Read_data_Valid 信号的时钟。经过简单的修改得以解决。

三、 对讲义中思考题（如有）的理解和回答

Volatile 是 C 语言中的一个关键字,其字面意思是不稳定的,在 C 语言中的作用是告诉编译器,这个变量的值随时可能改变,不应该对其值的读取进行优化,而应该每次都真正地去访问内存读取数据。若去掉 volatile 关键字,可能会导致编译器对 uart 所指向的物理地址进行优化,不会每次都读取数据,而 STAT_FIFO 寄存器是根据其与 uart 之间的地址偏移量来确定的,因此会顺便导致 FIFO 判满的错误。通过将.c 文件转化为汇编文件可以清楚地看到这种优化带来的差别。当然这种优化取决于编译器,因此上述推导仅仅是“可能”,若编译器没有进行任何的优化,则将 volatile 关键字去掉也不会有影响。

四、 在课后,你花费了大约 10 小时完成此次实验。

阶段提交: 6h

访存通路: 3h

性能计数器: 1h

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂,是否缺少了某些你认为重要的信息或参考资料,对实验项目的建议,对提供帮助的同学的感谢,以及其他想与任课老师交流的内容等）

实验难度不高,但是时间略微紧张,可能是因为最近到了期末季,deadline 急剧增加。在自己 debug 以及帮助其他同学 debug 的过程中,发现 alu 和 reg 的遗留问题可能带来错误结果,但是一开始并没有意识到这一点,从而在 cpu 的代码中辛苦找错误很久而未果,希望明年的课程中老师能够对这一现象进行一点点提示,减少同学们 debug 时浪费的时间。另外在云平台的上板测试中,我的 microbench 组的 queen 测试用例一直无法通过,找了很多同学帮忙找错也没有找到,经过一个多星期依旧是原样,希望老师能够帮我再检查一下,或者考虑一下是否会是云平台出现了异常。

本次实验需要感谢刘听雨和王嵩岳两位同学对我的帮助。