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：prj3　　实验名称：内存及外设通路设计与处理器性能评估

一、 逻辑电路结构与仿真波形的截图及说明

本次实验中 alu.v，reg_file.v，shifter.v 文件直接复用 prj2（单周期处理器设计）中的对应文件；custom_cpu.v 文件中的除与内存访问相关数据通路、多周期设计相关的状态机之外的 RTL 代码均直接复用 prj2 中的对应文件（mips_cpu.v）中的对应 RTL 代码。

（一）真实内存访问的多周期处理器设计（custom_cpu.v）

1．真实内存访问的 MIPS 多周期处理器状态转移图：

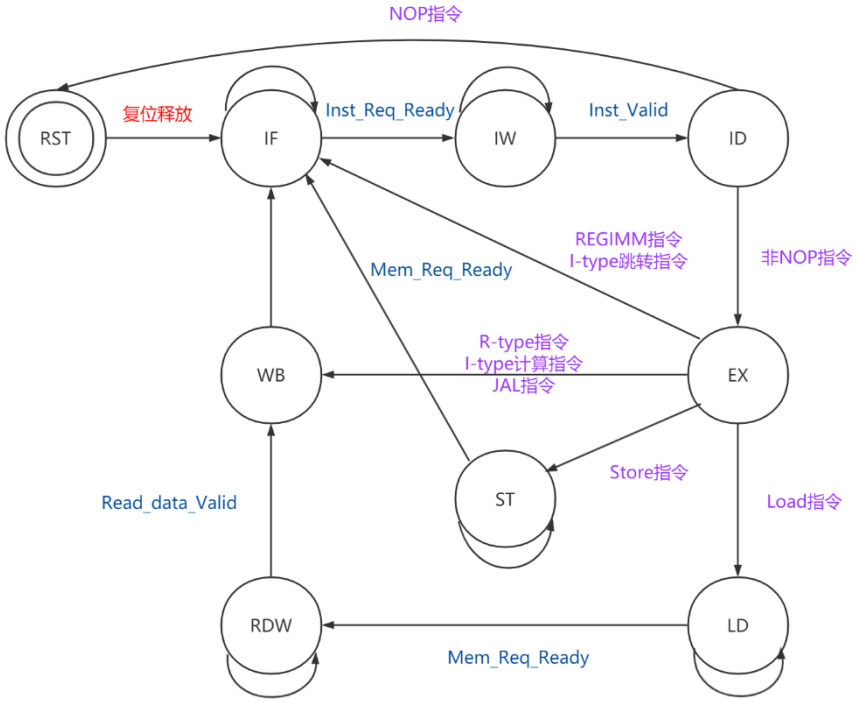

图 1. 真实内存访问的 MIPS 多周期处理器状态转移图

2. 根据状态转移图设计的三段式状态机 RTL 代码

（1）状态机的时序逻辑

```
111     always @ (posedge clk) begin
112        if(rst) begin
113            current_state <= RST;
114        end
115        else begin
116            current_state <= next_state;
117        end
118     end
```

（2）次态的计算组合逻辑

```
120      always @ (*) begin
121         case(current_state)
122             RST: begin
123                 if(rst) begin
124                     next_state = RST;
125                 end
126                 else begin
127                     next_state = IF;
128                 end
129             end
130
131             IF: begin
132                 if(Inst_Req_Ready) begin
133                     next_state = IW;
134                 end
135                 else begin
136                     next_state = IF;
137                 end
138             end
139
140             IW: begin
141                 if(Inst_Valid) begin
142                     next_state = ID;
143                 end
144                 else begin
145                     next_state = IW;
146                 end
147             end
148
149             ID: begin
150                 if(~|Instruction_reg) begin
151                     next_state = IF;
152                 end
153                 else begin
154                     next_state = EX;
155                 end
156             end
```

```
            //TO IF: REGIMM, I-Type branch, j
            //REGIMM: opcode = b000001
            //I-Type: opcode[5:2] = 0001
            //j: opcode = 000010
            //TO WB: R_type, I-type calculate, jal
            //R_type: opcode = 000000
            //I-type calculate: opcode[5:3] = 001
            //jal: opcode = 000011
            //TO ST: store : opcode[5:3] = 101
            //TO LD: load : opcode[5:3] = 100
            EX: begin
                if(((~|opcode[5:1]) & opcode[0]) | (opcode[5:2] == 4'b0001) | (opcode == 6'b000010)) begin
                    next_state = IF;
                end
                else if((~|opcode) | (opcode[5:3] == 3'b001) | (opcode == 6'b000011)) begin
                    next_state = WB;
                end
                else if(opcode[5:3] == 3'b101) begin
                    next_state = ST;
                end
                else if(opcode[5:3] == 3'b100) begin
                    next_state = LD;
                end
            end

            WB: begin
                next_state = IF;
            end

            ST: begin
                if(Mem_Req_Ready) begin
                    next_state = IF;
                end
                else begin
                    next_state = ST;
                end
            end
            LD: begin
                if(Mem_Req_Ready) begin
                    next_state = RDW;
                end
                else begin
                    next_state = LD;
                end
            end

            RDW: begin
                if(Read_data_Valid) begin
                    next_state = WB;
                end
                else begin
                    next_state = RDW;
                end
            end

            default: next_state = current_state;
        endcase
    end
```

（3）输出逻辑：包含指令寄存器、PC 等

```
218     //Instruction_reg
219     always  @ (posedge  clk) begin
220        if(Inst_Valid & Inst_Ready) begin
221            Instruction_reg <= Instruction;
222        end
223        else begin
224            Instruction_reg <= Instruction_reg;
225        end
226     end
```

```
228     //Read_data_reg
229     always  @ (posedge clk) begin
230        if(Read_data_Valid & Read_data_Ready) begin
231            Read_data_reg <= Read_data;
232        end
233        else begin
234            Read_data_reg <= Read_data_reg;
235        end
236     end
```

```
238     //PC_reg
239     always @ (posedge clk) begin
240        if(current_state[1]) begin
241            PC_reg <= PC;
242        end
243        else begin
244            PC_reg <= PC_reg;
245        end
246     end
247
248     always @ (posedge clk) begin
249         if(rst) begin
250             PC <= 32'b0;
251         end
252         else if(current_state[3]) begin //ID
253             PC <= branch ? branch_result : jump ? jump_result : PC4;
254         end
255         else begin
256             PC <= PC;
257         end
258     end
```

（4）控制信号

```
260     assign Inst_Req_Valid = current_state[1];
261     assign Inst_Ready = current_state[0] | current_state[2];
262     assign Read_data_Ready = current_state[0] | current_state[8];
```

```
430     assign MemWrite = current_state[6]; //ST
431     assign MemRead  = current_state[7]; //LD
```

特别的，RF_wen 信号只在 WB 阶段拉高

```
assign RF_wen =  (current_state == WB) &
```

同时，将所有原先由 wire 型 Instruction，Read_data，PC 等驱动的信号分别转换为由 reg 型 Instruction_reg，Read_data_reg，PC_reg 等驱动，保证运行时信号正确。

3. 逻辑电路结构

与真实内存访问相关逻辑电路结构图如下：

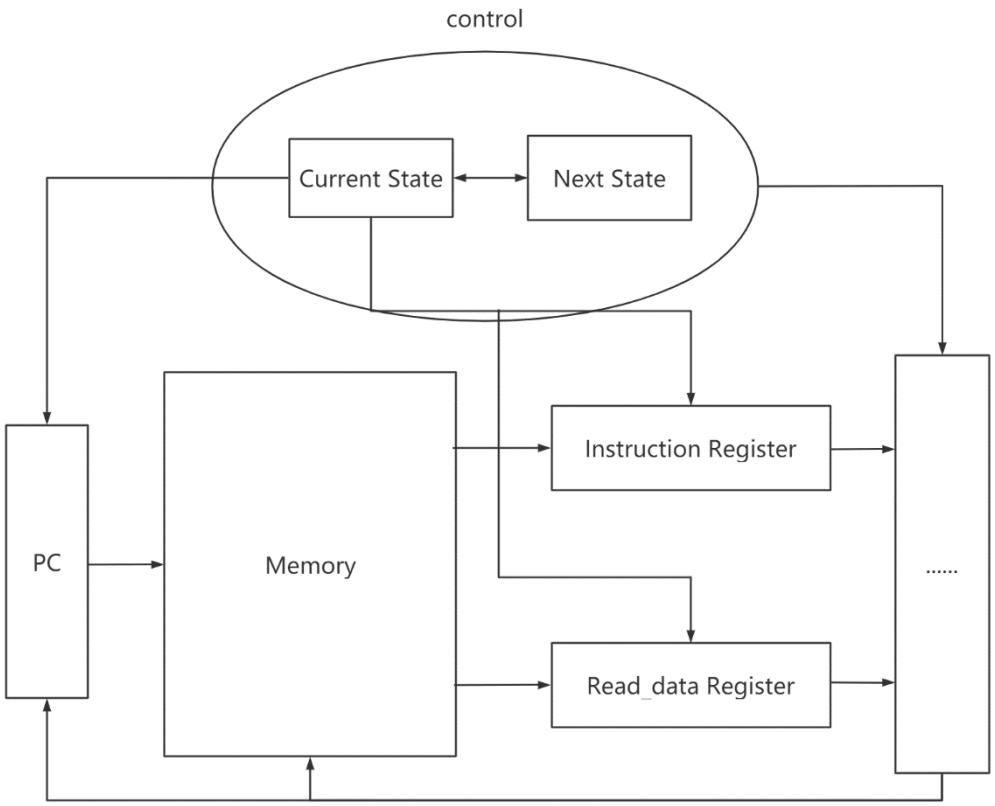

图 2 逻辑电路结构图

上图中省略部分与 prj2 中单周期 MIPS 处理器中的数据通路与控制信号相同，可详见 prj2。

通过时序逻辑确定当前处理器所处的状态，进而确定控制信号、选通相应的数据通路，执行对应状态下所应执行的操作。

（二）外设控制器访问（printf.c）

1. puts 函数代码：

```
249 int
250 puts(const char *s)
251 {
252         int i = 0;
253         while(s[i]){
254                 while ((*(volatile char *)(uart + 2)) & UART_TX_FIFO_FULL) ;
255                 *(volatile char *)(uart + 1) = s[i++];
256         }
257         return i;
258
259 }
```

其中 volatile 关键字保证编译器编译时不对其后变量相关代码进行优化，保证每次循环都直接从其对应内存地址中取值。

2. 上版运行结果

```
162 testing 1 2 0000003
163 faster and "cheaper"
164 deadf00d % DEADF00D
165 00000000100000000200000003000000004000000005
166 50 50 -50 4294967246
```

结果与标准一致

（三）性能计数器设计

1. 周期计数器添加

（1）custom_cpu.v 中 RTL 代码：

```
480  //cycle count
481  reg [31:0] cycle_cnt;
482  always @(posedge clk) begin
483      if(rst) begin
484          cycle_cnt <= 32'b0;
485      end
486      else begin
487          cycle_cnt <= cycle_cnt + 32'b1;
488      end
489  end
490  assign cpu_perf_cnt_0 = cycle_cnt;
491
492  endmodule
```

（2） bench.c

①_uptime 函数代码：

```
11  unsigned long _uptime() {
12    // TODO [COD]
13    //   You can use this function to access performance counter related with time or cycle.
14    unsigned long * cycle_count_addr;
15    cycle_count_addr = (unsigned long *)0x40020000;
16    return *cycle_count_addr;
17  }
```

②main 函数代码：

```
108         printk("The number of cycle counts: %u\n", msec);
```

（3）上版运行结果（以“15pz”为例）

```
Launching 15pz benchmark...
tggetattr: Inappropriate ioctl for device
argv[1]: ../benchmark/microbench/bin/15pz
after init mapping
after setting up CPU reset
after write PL DDR
after read PL DDR
after releasing CPU reset
[15pz] A* 15-puzzle search: * Passed.
The number of cycle counts: 536868716
benchmark finished
Hit 0 trap
Hit good trap
```

可正确计算运行周期数并输出。

2. 添加除周期计数器之外计数器：

本次实验中，添加 6 个性能计数器：内存访问计数器，执行指令总数计数器，条件跳转类指令总数计数器，条件跳转类指令条件满足总数计数器，无条件跳转类指令总数计数器。

一下为添加性能计数器所需代码

（1）custom_cpu.v 中 RTL 代码（以条件跳转类指令总数计数器为例）：

```
484     reg [31:0] branch_cnt;

521    //beq, bne, blez, bgtz: opcode[5:2] = 0001 , opcode[1:0] = 00, 01, 10, 11
522    //bltz, bgez:         : opcode[5:0] = 000001,  rt[0] = 0, 1
523    always @(posedge clk) begin
524        if(rst) begin
525            branch_cnt <= 32'b0;
526        end
527        else if(current_state[4] & (((~|opcode[5:1]) & opcode[0]) | ((~|opcode[5:3]) & opcode[2]))) begin
528            branch_cnt <= branch_cnt + 32'b1;
529        end
530        else begin
531            branch_cnt <= branch_cnt;
532        end
533    end

512     assign cpu_perf_cnt_3 = branch_cnt;
```

（2）bench.c

①Result 结构体修改：

```
typedef struct Result {
  int pass;
  unsigned long msec;
  unsigned long mem_cycle;
  unsigned long ins_count;
  unsigned long branch_count;
  unsigned long branch_suc_count;
  unsigned long jump_count;
  unsigned long jump_suc_count;
} Result;
```

②读取串口函数代码：

```
unsigned long _read_mem_cycle(){
  unsigned long * mem_cycle_addr;
  mem_cycle_addr = (unsigned long *)0x40020008;
  return *mem_cycle_addr;
}

unsigned long _ins_cycle() {
  unsigned long * ins_count_addr;
  ins_count_addr = (unsigned long *)0x40021000;
  return *ins_count_addr;
}

unsigned long _branch_cycle(){
  unsigned long * branch_count_addr;
  branch_count_addr = (unsigned long *)0x40021008;
  return *branch_count_addr;
}

unsigned long _branch_suc_cycle() {
  unsigned long * branch_suc_count_addr;
  branch_suc_count_addr = (unsigned long *)0x40022000;
  return *branch_suc_count_addr;
}

unsigned long _jump_cycle(){
  unsigned long * jump_count_addr;
  jump_count_addr = (unsigned long *)0x40022008;
  return *jump_count_addr;
}
```

各串口的地址与 custom_cpu.v 文件中对应功能所使用串口的地址保持一致。

③main 函数：

```
140     unsigned long mem_cycle = ULONG_MAX;
141     unsigned long ins_count = ULONG_MAX;
142     unsigned long branch_count = ULONG_MAX;
143     unsigned long branch_suc_count = ULONG_MAX;
144     unsigned long jump_count = ULONG_MAX;

153       if (res.mem_cycle < mem_cycle) mem_cycle = res.mem_cycle;
154       if (res.ins_count < ins_count) ins_count = res.ins_count;
155       if (res.branch_count < branch_count) branch_count = res.branch_count;
156       if (res.branch_suc_count < branch_suc_count) branch_suc_count = res.branch_suc_count;
157       if (res.jump_count < jump_count) jump_count = res.jump_count;

167     printk("The number of reading memory counts: %u\n", mem_cycle);
168     printk("The number of instruction counts: %u\n", ins_count);
169     printk("The number of branch operation counts: %u\n", branch_count);
170     printk("The number of successful branch operation counts: %u\n", branch_suc_count);
171     printk("The number of jump operation counts: %u\n", jump_count);
```

（3）上版运行结果（以“ssort”为例）

```
313 [ssort] Suffix sort: * Passed.
314 The number of cycle counts: 52755151
315 The number of reading memory counts: 54122
316 The number of instruction counts: 728408
317 The number of branch operation counts: 106130
318 The number of successful branch operation counts: 61552
319 The number of jump operation counts: 707
```

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、云平台调试过程中的难点等）

（一）代码逻辑错误

1. custom_cpu.v

（1）信号设置错误

问题（原因）：编写代码时大小写区分不当，导致信号值为 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）状态机相关

问题（原因）：状态机转移过程设计出错。

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，比对状态机状态转移图，加以修正。

2. printf.c

问题：结果打印不全。

原因：当 UART 控制器满时误加 break。

解决方案：比对流程图，定位错误点，加以修正：删去 break 语句。

（二）调试过程难点

1. 波形查看

问题(原因)：DUMP_TIME 区间设置不当(最大值超出已经仿真最大时间)，导致波形文件无法打开。

解决方案：根据仿真报错中已仿真的最大时间设置 DUMP_TIME 上界。

2. 性能计数器结果

问题：仿真阶段性能计数器输出结果始终为 0，上版结果正常。

该问题未能解决，疑似平台问题。

三、 对讲义中思考题的理解和回答

volatile 关键字作用：提示编译器其后的变量随时可能变化。保证编译器在编译时不对与该关键字相关的代码进行优化(默认循环中变量内存地址中数据不变（暂时使用寄存器中的值))，从而直接从变量内存地址中读取数据，提供对特殊地址的稳定访问

删去 volatile 关键字后果：编译器在编译时对相关代码进行优化，编译后的程序每次需要存储或读取其后的变量的时候，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。

四、 在课后，你花费了大约 15（30+）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．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 prj2 中 mips_cpu.v 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。

2．代码优化

puts 函数优化：

原 puts 函数代码：

```
249 int
250 puts(const char *s)
251 {
252         int i = 0;
253         while(s[i])
254                 if(!(*((char *)(uart + 2)) & UART_TX_FIFO_FULL))
255                         *(char *)(uart + 1) = s[i++];
256         return i;
257 }
```

优化后 puts 函数代码：

```
249 int
250 puts(const char *s)
251 {
252         int i = 0;
253         while(s[i]){
254                 while ((*(volatile char *)(uart + 2)) & UART_TX_FIFO_FULL) ;
255                 *(volatile char *)(uart + 1) = s[i++];
256         }
257         return i;
258
259 }
```

将 if 语句转化为 while 语句，将 URAT 控制器 TX FIFO 满时的判断次数减半。

感谢冯浩原同学（充当小黄鸭角色）和王嵩岳助教帮助我 debug 和解决云平台相关问题时提供的帮助；感谢黄天羿同学为代码优化提供思路。

（二）实验建议

1. 仿真与上版

本次实验在平台上仿真时间远远大于上版运行时间，当进行实验第二、第三部分（外设控制器访问、性能计数器访问）的代码编写和修改时，依旧需要跑完仿真，时间成本高。同时由于上版时对仿真结果存在依赖，无法使上版与仿真同

时运行。

希望可以在代码已经保证可以正常仿真时，允许仿真与上版同步运行。

2. 仿真波形获得

本次实验中仿真波形主要使用于 debug 阶段。

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，超出仿真运行时间波形文件无法生成），在一定情况下增加了仿真次数，使得 debug 周期变长。

希望可以通过脚本文件，自动根据仿真时间确定合适 DUMP_TIME 并生成对应波形文件。