中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 专业： 计算机科学与技术

实验序号： 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 格式保存在~/COD-Lab/reports 目录下。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 GitLab 远程仓库 master 分支（具体命令详见实验报告）。

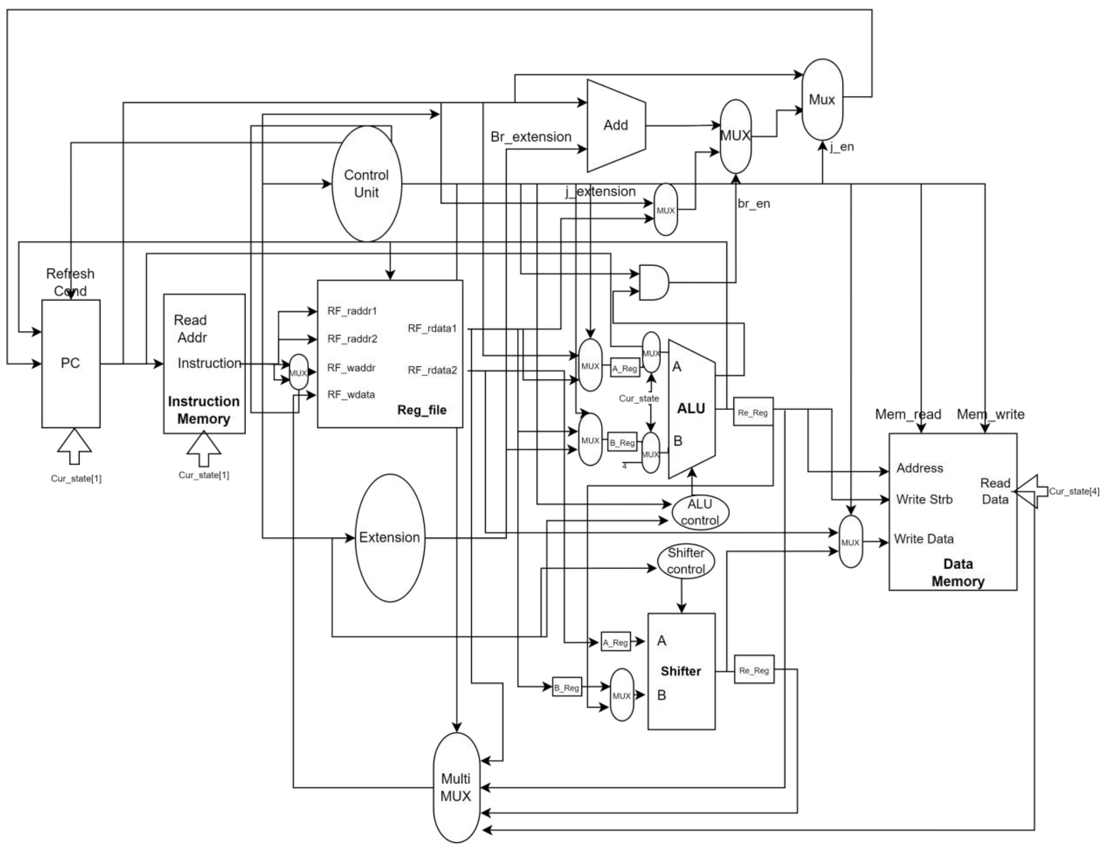
注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

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

* 说明：MIPS 定制处理器基于 prj2 实现的多周期处理器进行修改，其中 reg_file.v alu.v shifter.v 可直接复用，cpu 部分除状态机及相关部分代码外可直接复用，以下仅对修改部分和新增功能进行说明。

1. Custom_cpu.v 代码说明

逻辑结构图：和 prj2 实现的多周期处理器结构基本一致。



（1）状态机部分

```
//define the state of machine by one-hot
        localparam      RST     =9'b000000001,
                        IF      =9'b000000010,
                        IW      =9'b000000100,
                        ID      =9'b000001000,
                        EX      =9'b000010000,
                        LD      =9'b000100000,
                        ST      =9'b001000000,
                        RDW     =9'b010000000,
                        WB      =9'b100000000;
//define signal for State Machine
        //set 1 to Inst_Ready and Read_data_Ready to prevent Error
        assign Inst_Req_Valid   = current_state[1] ;    //IF
        assign Inst_Ready       = current_state[0] | current_state[2] ;//RST IW
        assign Read_data_Ready  = current_state[0] | current_state[7] ;//RST RDW
//State Machine
        //Part I:
    always @ (posedge clk) begin
       if(rst) begin
           current_state <= RST;
       end
```

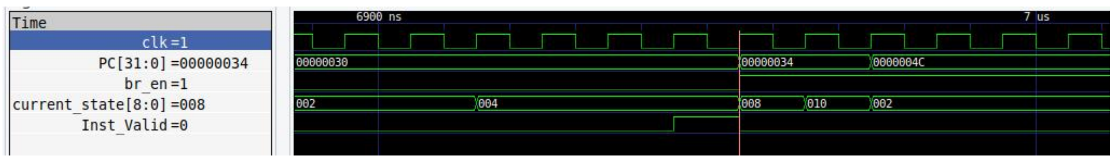
```
        else begin
            current_state <= next_state;
        end
    end

        //Part II: Combinatorial logic
        always @ (*)
        begin
                case(current_state)
                        RST: begin
                                next_state = IF ;
                        end
                        IF:     begin
                                if (Inst_Req_Ready)
                                        next_state = IW;
                                else
                                        next_state = IF;
                        end
                        IW: begin
                                if (Inst_Valid)
                                        next_state = ID;
                                else
                                        next_state = IW;
                        end
                        //NOP: Instr = 32'b0
                        ID: begin
                                if(~|Instruction_tmp)
                                        next_state = IF ;
                                else
                                        next_state = EX ;
                        end
                        //EX->IF:
                                //REGIMM        2: opcode[5:0]=000001
                                //I_branch      4: opcode[5:2]=0001
                                //J             1: opcode[5:0]=000010
                        //EX->WB
                                //R_Type        18: opcode[5:0]=6'b0 include JALR,JR //JR:
rs->GPR[0]
                                //I_calc        7: opcode[5:3]=001 include LUI
                                //JAL           1: opcode[5:0]=000011
                        //EX->LD
                                //load          7: opcode[5:3]=100
                        //EX->ST
                                //store         5: opcode[5:3]=101
                        EX: begin
                                if((opcode[5:0]==6'b000001) | (opcode[5:2]==4'b0001) |
(opcode[5:0]==6'b000010) )
                                        next_state = IF;
                                else if((~|opcode)| (opcode[5:3]==3'b001) |
(opcode[5:0]==6'b000011))
                                        next_state = WB;
                                else if(opcode[5:3]==3'b100)
                                        next_state = LD;
                                else if(opcode[5:3]==3'b101)
                                        next_state = ST;
                                else
                                        next_state = RST;
                        end
                        LD: begin
                                if(Mem_Req_Ready)
                                        next_state = RDW;
                                else
                                        next_state = LD;
                        end
                        ST: begin
                                if(Mem_Req_Ready)
                                        next_state = IF;
                                else
                                        next_state = ST;
                        end
                        RDW: begin
                                if(Read_data_Valid)
                                        next_state = WB;
                                else
                                        next_state = RDW;
                        end
                        WB: begin
                                next_state = IF;
                        end
                        default: begin
                                next_state = RST;
                        end
                endcase
        end

        //Part III: deal with output
        //one always module to deal one reg
        always @ (posedge clk) begin
                if(Inst_Ready & Inst_Valid) //after response
                        Instruction_tmp <= Instruction ;
        end

        always @ (posedge clk) begin
                if(Read_data_Ready & Read_data_Valid)
                        Read_data_tmp <=Read_data ;
        end

        always @ (posedge clk) begin
                if(current_state[1])//IF
                        PC_tmp <= PC;
        end

        always @ (posedge clk) begin
                if (rst) begin
                        PC <= 32'b0;
                end
                else if(current_state[2] & Inst_Valid & ~rst) begin
                        PC <= ALU_Result; //PC+4
                        //IW, consider the cycle before ID
                end
                else if(current_state[4] & ~rst)begin
                        if(br_en)
                                PC   <= ALU_Result_tmp;
                                //PC <= br_tar;
                        else if(j_en)
                                PC <= j_tar;
                        else
                                PC <= PC;
                        //PC <= br_en ? br_tar : j_en ? j_tar : PC ;
                        //j or br OP refresh in next IF, judge by EX
                        //note that the default result is PC4
                end
        end
```

状态机第一部分改变了状态复位值，设置从 RST 状态进行释放。

状态机第二部分相对 prj2 多周期处理器新增了取指和访存部分针对真实内存的等待阶段，并添加了相应握手信号。同时考虑到写内存只要求内存闲时即可写；但是读内存在内存闲时完成握手后，还需要等待内存返回数据，故将读写内存状态进行分离，并添加 RDW 读等待状态。

状态机第三部分针对握手信号和状态的改变进行了调整。当相应握手信号均为高时，将 Instruction 和 Read_data 进行寄存，PC 在 IF 阶段进行寄存。同时在一次指令周期中，PC 在 IW 阶段进行默认更新(+4)，且考虑到 IW 可能有多拍，因此添加 Inst_Valid 使得 PC 只在 IW 阶段的最后一拍更新；对于成功的跳转指令，PC 将在 EX 阶段进行额外的一次更新。

相应波形说明：



光标所指位置为 IW（004）阶段的最后一拍，PC 完成顺序更新；由于该条指令为条件跳转指令，在 EX（010）阶段 PC 再次更新为跳转值。

（2）访存相应控制信号修改

```
//mem control
        assign MemRead  = current_state[5];  //LD //100
        assign MemWrite = current_state[6];  //ST //101
```

由于只有 LOAD 类型指令可进入 LD 阶段，只有 STORE 类型指令可进入 ST 阶段，将控制信号赋值为对应阶段即可。

（3）性能统计相关信号

```
//counter
        //set specific name to distinguish: (v means victory, f means failure)
                //cycle, ins, mem(num of visit mem), delay,
                //branch_v, branch_f,jump_v,jump_f

        reg [31:0]      cycle_cnt;              //count the num of clock
        reg [31:0]      inst_cnt;               //count the num of obtained instruction
        reg [31:0]      ex_cnt;                 //count the num of execution
        reg [31:0]      mem_cnt;                //count the num of mem visit
        reg [31:0]      mem_delay_cnt;          //count the num of delay of mem,including read and
write
        reg [31:0]      br_j_v_cnt;             //count the num of PC not refresh for PC4
        reg [31:0]      br_j_f_cnt;             //count the num of PC refresh for PC4, including
```

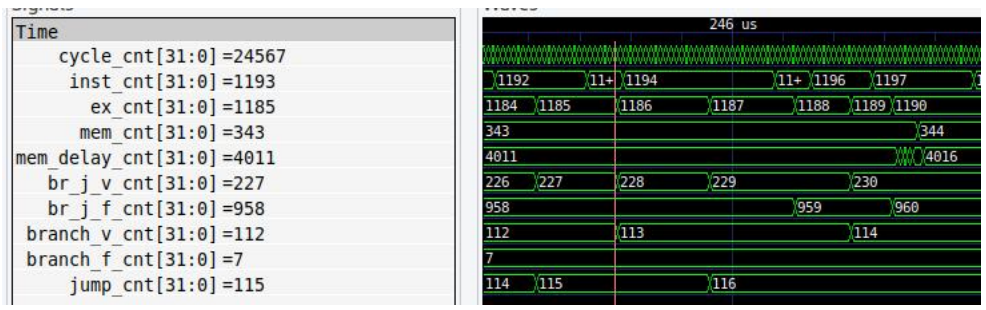
```
instr not br or j
        reg [31:0]      branch_v_cnt;          //count the num of successful branch
        reg [31:0]      branch_f_cnt;          //count the num of failed branch
        reg [31:0]      jump_cnt;              //count the num of jump (always success)

        assign cpu_perf_cnt_0   = cycle_cnt;
        assign cpu_perf_cnt_1   = inst_cnt;
        assign cpu_perf_cnt_2   = ex_cnt;
        assign cpu_perf_cnt_3   = mem_cnt;
        assign cpu_perf_cnt_4   = mem_delay_cnt;
        assign cpu_perf_cnt_5   = br_j_v_cnt;
        assign cpu_perf_cnt_6   = br_j_f_cnt;
        assign cpu_perf_cnt_7   = branch_v_cnt;
        assign cpu_perf_cnt_8   = branch_f_cnt;
        assign cpu_perf_cnt_9   = jump_cnt;

        always @ (posedge clk) begin
                if(rst)
                        cycle_cnt <= 32'b0;
                else
                        cycle_cnt <= cycle_cnt + 32'b1;
        end

        always @ (posedge clk) begin
                if(rst)
                        inst_cnt <= 32'b0;
                else if(current_state[1]) //IF
                        inst_cnt <= inst_cnt + 32'b1;
        end

        always @ (posedge clk) begin
                if(rst)
                        ex_cnt <= 32'b0;
                else if(current_state[4])//EX
                        ex_cnt <= ex_cnt + 32'b1;
        end

        always @ (posedge clk) begin
                if(rst)
                        mem_cnt <= 32'b0;
                else if(current_state[5] | current_state[6]) //ST LD
                        mem_cnt <= mem_cnt + 32'b1;
        end
        always @ (posedge clk) begin
                if(rst)
                        mem_delay_cnt <= 32'b0;
                else if(( (current_state[5]|current_state[6]) &
~Mem_Req_Ready )|( current_state[7] & ~Read_data_Valid )) //LD ST ~Mem_Req_Ready -- RDW
~Read_data_Valid
                        mem_delay_cnt <= mem_delay_cnt + 32'b1;
        end
        //state = EX 4
        //Branch 4 :                    opcode[5:2]=0001
        //REGIMM 2 :                    opcode[5:0]=000001
        //jump   4 : J JAL              opcode[5:1] =00001
                  //JR JALR             opcode[5:0] = 000000 & func[5:1]=00100
        always @ (posedge clk) begin
                if(rst)
                        br_j_v_cnt <= 32'b0;
                else if(current_state[4] & ( ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) &
br_en) | (opcode[5:1]==5'b00001 | (opcode[5:0]==6'b000000 & func[5:1]==5'b00100)) ))
                        br_j_v_cnt <= br_j_v_cnt + 32'b1;  //branch_suc or jump
        end

        always @ (posedge clk) begin
                if(rst)
                        br_j_f_cnt <= 32'b0;
                else if(current_state[4] & ~( ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) &
br_en) | (opcode[5:1]==5'b00001 | (opcode[5:0]==6'b000000 & func[5:1]==5'b00100)) ))
                        br_j_f_cnt <= br_j_f_cnt + 32'b1;  //branch_fail
        end

        always @ (posedge clk) begin
                if(rst)
                        branch_v_cnt <= 32'b0;
                else if(current_state[4] & ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) &
br_en))
                        branch_v_cnt <= branch_v_cnt + 32'b1;
        end

        always @ (posedge clk) begin
                if(rst)
                        branch_f_cnt <= 32'b0;
                else if(current_state[4] & ((opcode[5:2]==4'b0001 | opcode[5:0]==6'b000001) &
~br_en))
                        branch_f_cnt <= branch_f_cnt + 32'b1;
        end
```

```
	always @ (posedge clk) begin
		if(rst)
			jump_cnt <= 32'b0;
		else if(current_state[4] & (opcode[5:1]==5'b00001 | (opcode[5:0]==6'b000000 & 
func[5:1]==5'b00100)))
			jump_cnt <= jump_cnt + 32'b1;
	end
```

本次实验设置了十个性能统计计数器，分别为 cycle_cnt（时钟周期数统计）、inst_cnt（取得指令数统计）、ex_cnt（执行指令数统计）、 mem_cnt（访存次数统计）、mem_delay_cnt（访存延迟统计） br_j_v_cnt（PC 跳转次数统计）、 br_j_f_cnt（PC 顺序更新次数统计）、branch_v_cnt（条件跳转成功次数统计）、 branch_f_cnt（条件跳转失败次数统计）、 jump_cnt（无条件跳转次数统计）。

以上计数器在复位阶段置零，并在满足对应条件的时钟上升沿更新。连接计数器和对应输出端口即可。

相应波形说明：



```
Time
     cycle_cnt[31:0] =24567
      inst_cnt[31:0] =1193
        ex_cnt[31:0] =1185
       mem_cnt[31:0] =343
 mem_delay_cnt[31:0] =4011
     br_j_v_cnt[31:0] =227
     br_j_f_cnt[31:0] =958
   branch_v_cnt[31:0] =112
   branch_f_cnt[31:0] =7
      jump_cnt[31:0] =115
```

某一时刻的性能计数器统计值，其中 ex_cnt = br_j_v_cnt + br_j_f_cnt 、br_j_v_cnt = branch_v_cnt + jump_cnt，均符合预期。

2. 外设控制器访问 puts.c 代码说明

```
/*==============================================================
 * puts: send characters in input string to UART TX FIFO in order
 * @s: input string
 *
 * Return: return the actual string length that has been sent out
 *=================================================================
 */
int
puts(const char *s)
{
	//TODO: Add your driver code here
	//UART control unit:
		//UART_TX_FIFO : offset from TX to uart
		//UART_STATUS  : offset from STAT to uart
	//wait until STAT is not full, then send a characters
	//use volatile to provide stable access to a particular address
	/*
	* OPTIMIZE:
	* reverse cond of while to cond of if, save while inside
	* (char *)(base+1) is equal to (char *)base + 4, save shifter
	*/
	int i=0;
	while (s[i]){
		if (!( *((volatile char *)uart + UART_STATUS) & UART_TX_FIFO_FULL))
			*((volatile char *)uart + UART_TX_FIFO)=s[i++];
	}
	return i;
}
```

对于输入的字符串 s，需将其逐位打印，并且返回成功打印的字符数，因此设置 i 作为字符数组下标进行循环枚举。

由于 TX FIFO 寄存器只可写不可读，且需要更改 32 位中的最低 8 位，因此考虑将 uart 从 int 型指针强制类型转化为 char 型指针，使其指向的数据为 1 字节。同时添加 volatile 关键字阻止编译器对访问变量的代码优化，保证系统总是从其所在内存读取数据，从而稳定访问该地址。

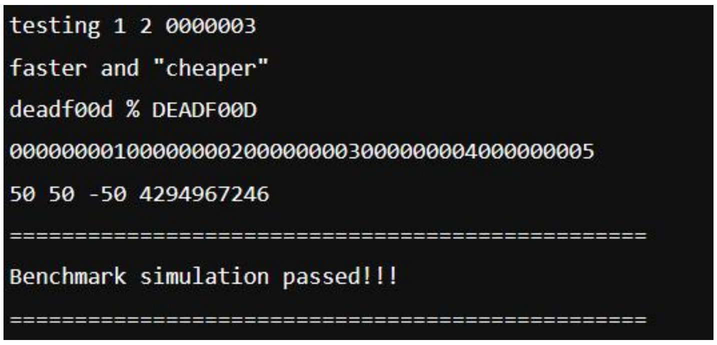
将类型转换后的 uart 加上寄存器偏移地址即得 TX FIFO 和 STAT_REG 的绝对地址。取对应地址的值就可得到 TX FIFO 和 STAT_REG 的最低字节。

将 STAT_REG 的最低字节和 UART_TX_FIFO_TULL（1000）进行按位与

即可得到最低第 3 位为对应值，其余位为 0 的数。将其取反后作为条件，即可保证只有 STAT_REG 最低第 3bit 为 0，才可打印字符 s[i].

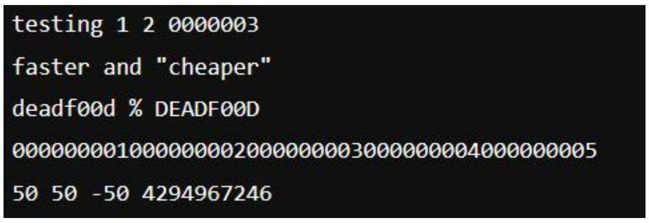
将 TX FIFO 的最低字节赋值为 s[i]，即实现了将待发送的 8bit 字符写入 TX FIFO 寄存器。

仿真运行结果：



```
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
0000000010000000020000000030000000040000000005
50 50 -50 4294967246
==================================================
Benchmark simulation passed!!!
==================================================
```

上板运行结果：



```
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
0000000010000000020000000030000000040000000005
50 50 -50 4294967246
```

3. 性能统计相关软件代码说明

（1） Perf_cnt.h 说明

```
typedef struct Result {
        int pass;
        unsigned long msec;             //cycle_cnt
        unsigned long inst_cnt;
        unsigned long ex_cnt;
        unsigned long mem_cnt;
        unsigned long mem_delay_cnt;
        unsigned long br_j_v_cnt;
        unsigned long br_j_f_cnt;
        unsigned long branch_v_cnt;
        unsigned long branch_f_cnt;
        unsigned long jump_cnt;
} Result;
```

在结构体中添加性能计数器相关统计变量。

（2） Perf_cnt.c 说明

```
#include "perf_cnt.h"

//The addr of specific perf_cnt:
        //perf_cnt_0  ->  0x60010000
        //perf_cnt_1  ->  0x60010008
        //perf_cnt_2  ->  0x60011000
        //...


//The following 10 func is side-by-side
unsigned long _uptime() {
  // TODO [COD]
  //   You can use this function to access performance counter related with time or cycle.
        unsigned long *msec_addr;
        msec_addr = (unsigned long *)0x60010000;
        return *msec_addr;
}

unsigned long _inst_freq() {
        unsigned long *inst_cnt_addr;
        inst_cnt_addr = (unsigned long *)0x60010008;
        return *inst_cnt_addr;
}

unsigned long _ex_freq() {
        unsigned long *ex_cnt_addr;
        ex_cnt_addr = (unsigned long *)0x60011000;
        return *ex_cnt_addr;
}

unsigned long _mem_freq() {
        unsigned long *mem_cnt_addr;
        mem_cnt_addr = (unsigned long *)0x60011008;
        return *mem_cnt_addr;
}
```

```c
unsigned long _mem_delay_freq() {
        unsigned long *mem_delay_cnt_addr;
        mem_delay_cnt_addr = (unsigned long *)0x60012000;
        return *mem_delay_cnt_addr;
}

unsigned long _br_j_v_freq() {
        unsigned long *br_j_v_cnt_addr;
        br_j_v_cnt_addr = (unsigned long *)0x60012008;
        return *br_j_v_cnt_addr;
}

unsigned long _br_j_f_freq() {
        unsigned long *br_j_f_cnt_addr;
        br_j_f_cnt_addr = (unsigned long *)0x60013000;
        return *br_j_f_cnt_addr;
}

unsigned long _branch_v_freq() {
        unsigned long *branch_v_cnt_addr;
        branch_v_cnt_addr = (unsigned long *)0x60013008;
        return *branch_v_cnt_addr;
}

unsigned long _branch_f_freq() {
        unsigned long *branch_f_cnt_addr;
        branch_f_cnt_addr = (unsigned long *)0x60014000;
        return *branch_f_cnt_addr;
}

unsigned long _jump_freq() {
        unsigned long *jump_cnt_addr;
        jump_cnt_addr = (unsigned long *)0x60014008;
        return *jump_cnt_addr;
}
void bench_prepare(Result *res) {
  // TODO [COD]
  //   Add preprocess code, record performance counters' initial states.
  //   You can communicate between bench_prepare() and bench_done() through
  //   static variables or add additional fields in `struct Result`
  res->msec            = _uptime();
  res->inst_cnt        = _inst_freq();
  res->ex_cnt          = _ex_freq();
  res->mem_cnt         = _mem_freq();
  res->mem_delay_cnt    = _mem_delay_freq();
  res->br_j_v_cnt       = _br_j_v_freq();
  res->br_j_f_cnt       = _br_j_f_freq();
  res->branch_v_cnt     = _branch_v_freq();
  res->branch_f_cnt     = _branch_f_freq();
  res->jump_cnt         = _jump_freq();
}

void bench_done(Result *res) {
  // TODO [COD]
  //  Add postprocess code, record performance counters' current states.
  res->msec             = _uptime()             - res->msec;
  res->inst_cnt         = _inst_freq()          - res->inst_cnt;
  res->ex_cnt           = _ex_freq()            - res->ex_cnt;
  res->mem_cnt          = _mem_freq()           - res->mem_cnt;
  res->mem_delay_cnt    = _mem_delay_freq()     - res->mem_delay_cnt;
  res->br_j_v_cnt       = _br_j_v_freq()        - res->br_j_v_cnt;
  res->br_j_f_cnt       = _br_j_f_freq()        - res->br_j_f_cnt;
  res->branch_v_cnt     = _branch_v_freq()      - res->branch_v_cnt;
  res->branch_f_cnt     = _branch_f_freq()      - res->branch_f_cnt;
  res->jump_cnt         = _jump_freq()          - res->jump_cnt;
}
```

添加性能计数器相关函数，每个函数访问该性能计数器对应位置，返回其中的值。Bench_prepare 函数在运行 benchmark 前调用，将每个性能计数器的初始值存储于结构体变量中，Bench_done 函数在 benchmark 后调用，获取当前每个性能计数器的值，并减去存储于结构体中的初始值，即可得到该 benchmark 运行过程中的性能统计结果。

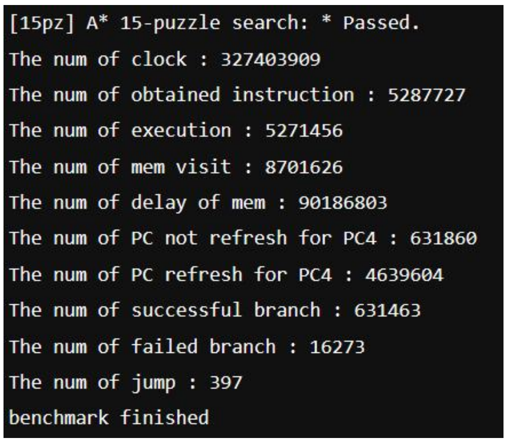
（3） Bench.c 说明

```
for (int i = 0; i < ARR_SIZE(benchmarks); i ++) {
    Benchmark *bench = &benchmarks[i];
    current = bench;
    setting = &bench->settings[SETTING];
    const char *msg = bench_check(bench);
    printk("[%s] %s: ", bench->name, bench->desc);
    if (msg != NULL) {
      printk("Ignored %s\n", msg);
    } else {
  unsigned long msec      = ULONG_MAX;
  unsigned long inst_cnt    = ULONG_MAX;
  unsigned long ex_cnt      = ULONG_MAX;
  unsigned long mem_cnt     = ULONG_MAX;
  unsigned long mem_delay_cnt = ULONG_MAX;
  unsigned long br_j_v_cnt    = ULONG_MAX;
  unsigned long br_j_f_cnt    = ULONG_MAX;
  unsigned long branch_v_cnt  = ULONG_MAX;
  unsigned long branch_f_cnt  = ULONG_MAX;
  unsigned long jump_cnt    = ULONG_MAX;
  int succ = 1;
      for (int i = 0; i < REPEAT; i ++) {
        Result res;
        run_once(bench, &res);
        printk(res.pass ? "*" : "X");
        succ &= res.pass;
        if (res.msec    < msec)   msec    = res.msec;
        if (res.inst_cnt  < inst_cnt)   inst_cnt  = res.inst_cnt;
        if (res.ex_cnt    < ex_cnt)   ex_cnt    = res.ex_cnt;
        if (res.mem_cnt   < mem_cnt)  mem_cnt   = res.mem_cnt;
        if (res.mem_delay_cnt   < mem_delay_cnt)mem_delay_cnt = res.mem_delay_cnt;
        if (res.br_j_v_cnt  < br_j_v_cnt)   br_j_v_cnt  = res.br_j_v_cnt;
        if (res.br_j_f_cnt  < br_j_f_cnt)   br_j_f_cnt  = res.br_j_f_cnt;
        if (res.branch_v_cnt  < branch_v_cnt) branch_v_cnt  = res.branch_v_cnt;
        if (res.branch_f_cnt  < branch_f_cnt) branch_f_cnt  = res.branch_f_cnt;
        if (res.jump_cnt  < jump_cnt)   jump_cnt  = res.jump_cnt;
      }

      if (succ) printk(" Passed.\n");
      else printk(" Failed.\n");

      pass &= succ;

      // TODO [COD]
      //   A benchmark is finished here, you can use printk to output some information.
      //   `msec' is intended indicate the time (or cycle),
      //   you can ignore according to your performance counters semantics.
    printk ("The num of clock : %u\n" , msec);
    printk ("The num of obtained instruction : %u\n" , inst_cnt);
    printk ("The num of execution : %u\n" , ex_cnt);
    printk ("The num of mem visit : %u\n" , mem_cnt);
    printk ("The num of delay of mem : %u\n" , mem_delay_cnt);
    printk ("The num of PC not refresh for PC4 : %u\n" , br_j_v_cnt);
    printk ("The num of PC refresh for PC4 : %u\n" , br_j_f_cnt);
    printk ("The num of successful branch : %u\n" , branch_v_cnt);
```

```
    printk ("The num of failed branch : %u\n" , branch_f_cnt);
    printk ("The num of jump : %u\n" , jump_cnt);

    }
  }
```

对每个 benchmark，运行 run_once 函数，将对应性能统计结果存储于结构体变量 Res 中，当其值小于默认值（最大值）时，将性能统计值赋值为结构体中所存储的值。然后调用 printk 函数将所有性能统计结果打印。

性能统计结果：（以 15pz 为例）



```
[15pz] A* 15-puzzle search: * Passed.
The num of clock : 327403909
The num of obtained instruction : 5287727
The num of execution : 5271456
The num of mem visit : 8701626
The num of delay of mem : 90186803
The num of PC not refresh for PC4 : 631860
The num of PC refresh for PC4 : 4639604
The num of successful branch : 631463
The num of failed branch : 16273
The num of jump : 397
benchmark finished
```

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，逻辑仿真和 FPGA 调试过程中的难点等）

问题 1：状态机中赋值条件未覆盖所有情况

在本次实验测试中五个阶段测试均可通过，并未发现错误。在后续实验 prj4 中通过硬件仿真加速工具发现状态机第二部分的一个 case 中，if 未写 else，形成了 latch。

问题 2：延迟降低

验收过程中，常老师提到可以用与或电路代替三目运算，也即将电路从级联改为树形连接，从而降低延迟。将代码中三目运算层级大于 2 的均替换为与或

门，降低了延迟。

问题 3：命名歧义

在 CPU 设计中，对 PC、Instruction 及 ALU 和 shifter 的操作数和结果进行寄存。原文件将某 wire 型信号延后一拍的寄存器命名为 xxx_reg，和 vivado 编译中对于 reg 型变量添加的后缀重复，容易产生歧义，故将后缀改为 xxx_tmp。

三、 对讲义中思考题（如有）的理解和回答

volatile 可保证系统总是从其所在内存读取数据，阻止编译器对访问变量的代码优化，从而提供对特殊地址的稳定访问，即被 volatile 修饰的变量修改后会直接写回内存，其他操作访问该变量将直接从内存中读取数值。由此保证了某变量修改后全局可见，且保证了多线程下相关代码的执行顺序。

第二部分提供的 puts 代码由于对于寄存器的读写均针对绝对地址，且只在一次操作完成，稳定性较好。其他线程对寄存器的改变对不会影响一次读或写寄存器得到的结果，因此即使删除了 volatile 关键字，仍在仿真和上板阶段都得到了正确的结果。但在与其他同学的讨论中发现，对于某些 puts 函数实现方法，删除 volatile 会导致打印信息不全的问题。

四、 在课后，你花费了大约_____5______小时完成此次实验。

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）

心得感受：

本次实验基于 prj2 的多周期处理器，针对真实内存访问进行了一定优化。硬件部分实现难度较低，主要需要加深对于握手信号的理解。在外设可控制器访问阶段，需要了解不同寄存器的读写权限，在特定条件下进行指定位数字符的传送。性能计数器的逻辑思路较为简单，且重复性较强，只需针对欲统计数值添加即可。

建议：

在 prj2 多周期处理器的基础上，本次实验难度较小，虽然实现了软硬件的协同，但从需要完成的代码部分来看，其联系仍较为割裂。建议适当加深对外设控制器、打印和性能计数函数等的讲解和要求，如当 puts 函数将字符写入 TX FIFO 后，该字符将被如何处理以使对应位置可接收下一字符。如 bench.c 如何对调用各函数最终实现将性能统计结果打印在日志中。

另外，在本次实验中，发现多次上板运行报错，需要重复 retry 才可通过，在消耗大量时间的同时也导致了更多无用的板卡占用。希望能进一步加强 FPGA 稳定性，减少不同并行任务的相互影响。

致谢：

感谢常老师在代码优化上的建议（详见问题 2、3）。