# 中国科学院大学
# 《计算机组成原理(研讨课)》实验报告

姓名 韦欣池 学号 2022K8009907004 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——真实内存、外设与性能计数器访问

| | |
|---|---|
| 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)

**• 处理器真实内存访问通路设计。**

在实验二中, 我们使用的是理想内存, 而在实验三我们需要使用真实内存, 关于二者的区别, 在讲义中以及交代: 对于理想内存, 内存访问与寄存器访问具有相同的延时, 并且读写操作全部在一个时钟周期内完成; 对于真实内存, 一次读/写操作需要多个时钟周期, 且需要遵循一定的约束才能够进行内存的读写。

因此, 我们首先要修改访存接口, 实验二中有四个通道与内存有关(见下图 1), 我们需要为每个通道添加握手信号。Valid 信号高电平表示发送方发出的请求或应答内容有效, Ready 信号高电平表示接收方可以接收发送方的请求或应答(见下图 2)。

- 四个访存通道
  - 指令请求发送通道（程序计数器PC作为内存读地址）
  - 指令应答接收通道（指令Instruction）
  - 数据请求发送通道（数据访问地址 + 读/写控制信号 + 写数据）
  - 数据应答接收通道（读数据Read_data）

图 1: Project2 中的四个访存通道

```
//Instruction request channel
output reg [31:0] PC,        //程序计数器
output        Inst_Req_Valid,
input         Inst_Req_Ready,

//Instruction response channel
input  [31:0] Instruction,  //从内存（Memory）中读取至处理器的指令
input         Inst_Valid,
output        Inst_Ready,

//Memory request channel
output [31:0] Address,      //数据访存指令使用的内存地址
output        MemWrite,     //内存访问的写使能信号（高电平有效）
output [31:0] Write_data,   //内存写操作数据
output [ 3:0] Write_strb,   //内存写操作字节有效信号（支持32/16/8-bit内存写）
                            //Write_strb[i] == 1表示
                            //Write_data[8×(i+1)-1 : 8×i]位会被写入内存的对应地址
output        MemRead,      //内存访问的读使能信号（高电平有效）
input         Mem_Req_Ready,

//Memory data response channel
input  [31:0] Read_data,    //从内存中读取的数据
input         Read_data_Valid,
output        Read_data_Ready,
```

图 2: 四个通道添加的握手信号

为了实现以上功能，我们需要将实验二的单周期处理器进行修改，将其改为多周期处理器，而要用时序逻辑电路完成多周期处理器的内容，在数字电路课程中我们学习了通过三段式状态机的方法来完成，我们在多周期状态机中设置了如下图的 9 种状态：

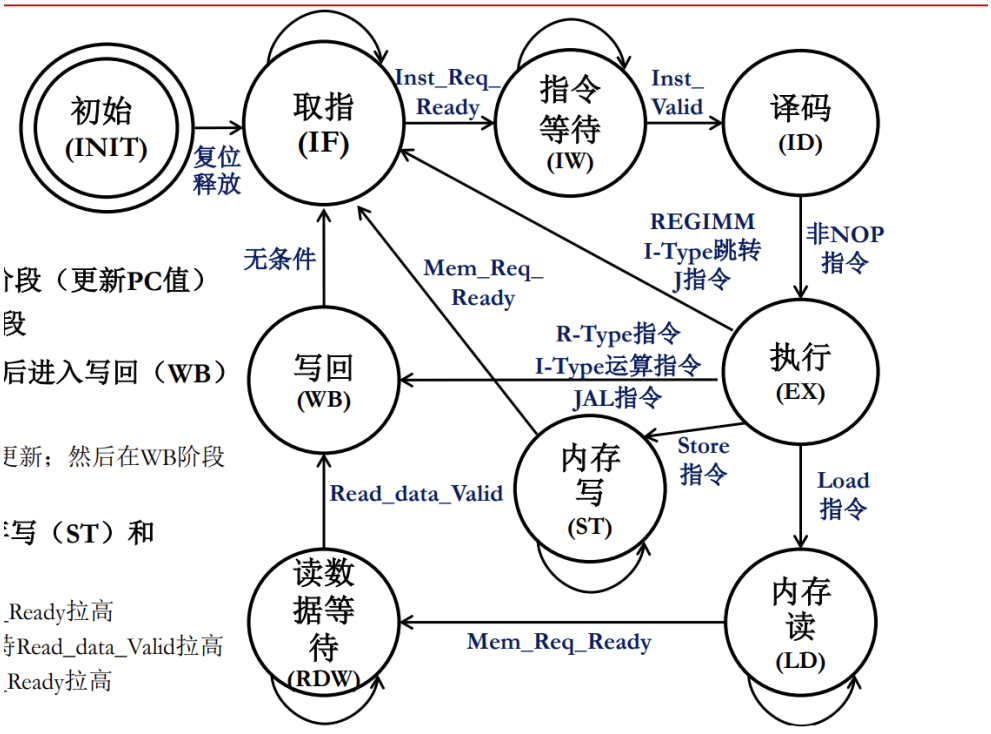



图 3: MIPS 处理器状态转移图

然后我们通过设置独热码将这几种状态分别进行编码：

```
localparam  INIT = 9'b000000001,
            IF = 9'b000000010,
            IW = 9'b000000100,
            ID = 9'b000001000,
            EX = 9'b000010000,
            ST = 9'b000100000,
            WB = 9'b001000000,
            LD = 9'b010000000,
            RDW = 9'b100000000;
```

图 4: 9 种状态的编码

三段式的过程见下图：

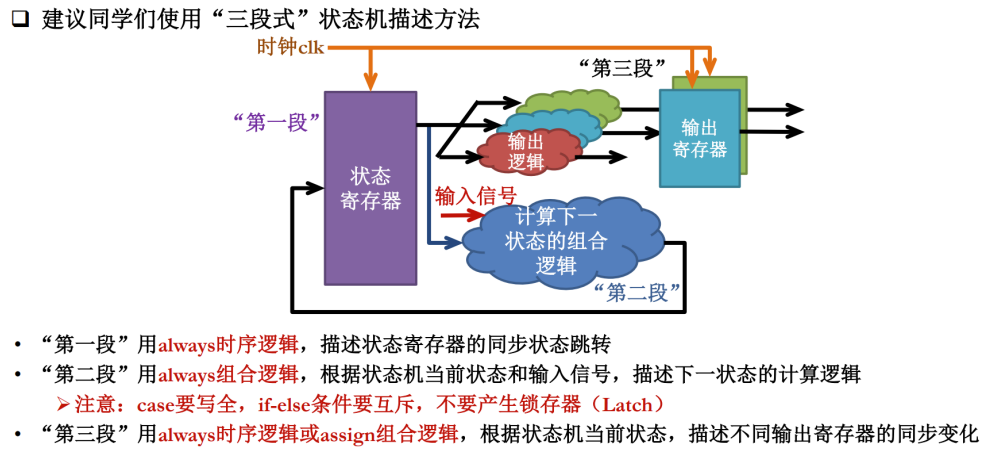



图 5: 三段式状态机步骤

对于第一段，我们需要描述状态寄存器的同步状态跳转，该部分内容在讲义中已经介绍的很清楚了，直接仿照 PPT 完成即可。

```
1  //第一段：描述状态寄存器的同步跳转状态
2  always@(posedge clk)
3  begin
4      if(rst)
5          current_state <= INIT;
6      else
7          current_state <= next_state;
8  end
```

图 6: 状态机第一段:寄存器的同步状态跳转

对于第二段,我们需要根据状态机当前状态和输入信号,描述下一状态的计算逻辑。意思是说,当状态机处于某种状态时,我们需要根据某些信号的取值,来决定下一状态。

```verilog
//第二段：根据状态机当前状态和输入信号，描述下一状态的计算逻辑
always@(*)
begin
    case(current_state)
        INIT:   next_state = IF;
        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
        ID: begin
            if(Instruction_current != 32'b0)    //非NOP指令
                next_state = EX;
            else
                next_state = IF;
        end
        EX: begin
            if(opcode == 6'b000001 || opcode[5:2] == 4'b0001 || opcode == 6'b000010)    //REGIMM,I-Type跳转指令,J指令
                next_state = IF;
            else if(opcode == 6'b0 || opcode[5:3] == 3'b001 || opcode == 6'b000011)     //R-Type指令,I-Type运算指令,JAL指令
                next_state = WB;
            else if(opcode[5:3] == 3'b101)  //I-Type内存写指令
                next_state = ST;
            else if(opcode[5:3] == 3'b100)  //I-Type内存读指令
                next_state = LD;
            else
                next_state = EX;
        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:     next_state = IF;
        default:
            next_state = current_state;
    endcase
end
```

图 7: 状态机第二段:描述次态的计算组合逻辑

对于第三段,我们需要根据状态机当前状态,描述不同输出寄存器的同步变化。我们需要修改与内存有关的变量的值的变化,即在单周期处理器中,每次时钟上升沿这些变量都会随之而改变,但是在多周期处理器中,我们需要根据当前的状态以及相关的握手指令来进行判断这些与内存读写相关的变量的变化。

```
//第三段：根据状态机当前状态，描述不同输出寄存器的同步变化
//用current_state处理输出变量的组合逻辑
assign PC_next  = PC + 4;
assign Inst_Req_Valid = (current_state == IF) ? 1 : 0;
assign Inst_Ready = (current_state == IW || current_state == INIT) ? 1 : 0;
assign Read_data_Ready = (current_state == INIT || current_state == RDW) ? 1 : 0;

//PC寄存器的时序逻辑电路
always@(posedge clk)
begin
    if (rst)            //rst同步的高电平复位信号
    begin
        PC <= 32'b0;
    end
    else if(current_state == EX)    //根据操作不同，选择相应的下一条指令的PC地址
    begin
        PC <= ({32{Jump}} & Jump_addr) | ({32{Branch}} & Branch_addr) | ({32{~Jump}} & {32{~Branch}} & PC_next);
    end
    else if (Instruction == 32'b0 && current_state == IW && Inst_Ready && Inst_Valid)   //当指令为NOP
    begin
        PC <= PC_next;
    end
end
//判断是否更新PC值的时序逻辑电路
always@(posedge clk)
begin
    if (rst)
    begin
        PC_current <= 32'b0;
    end
    else if(current_state == IF)
        PC_current <= PC;
end
//判断是否更新Instruction_current值的时序逻辑电路
always@(posedge clk)
begin
    if(rst)
    begin
        Instruction_current <= 32'b0;
    end
    else if(Inst_Ready && Inst_Valid)
        Instruction_current <= Instruction;
end
//判断是否更新读数据值的时序逻辑电路
always@(posedge clk)
begin
    if(rst)
    begin
        Read_data_current <= 32'b0;
    end
    else if(Read_data_Ready && Read_data_Valid)
        Read_data_current <= Read_data;
end
```

图 8: 状态机第三段:输出逻辑的处理

从上图中我们可以看到我对 PC 进行了相关修改,因为只有在 EX 阶段,读到相应的指令之后我们才会修改 PC 值,而在实验二中是时钟上升沿就会进行 PC 值的修改,所以我设置了 PC_current 变量,用于保持当前 PC 值不变,从而避免了 PC 值在 EX 阶段变化影响计算结果。

同时,Instruction 变量我们也另外设置了一个 Instruction_current 变量,因为只有在对应的握手信号都为 1 的时候我们才需要去更新当前的 Instruction。Read_data 变量同理。

需要注意,在我们添加这些变量,以及完成相应的时序控制之后,要在实验二的代码部分进行修改,以保证前面的组合逻辑的变量的正确连接。

除此之外的其余部分,除了需要添加一些性能计数器的代码之外,都是实验二照搬过来。

- **外设控制器访问方法与字符串打印功能实现**。

该部分主要是对软件进行编写,要实现“终端打印”功能,可理解为处理器通过串口发送(TX)字符串。该部分

实现的是通用异步串行收发器(UART)控制器。该控制器帮助处理器实现通用异步串行通信协议,并对处理器隐藏实现细节,其仅仅将一系列 32-bit 数据/控制/状态寄存器呈现给处理器,供处理器上运行的软件读写访问。

在处理器访问内存的时候,我们需要向其提供地址,由于我们使用的是真实内存,因此我们需要对外设控制器的 I/O 端口进行物理地址编址,本实验使用的是内存-I/O 端口统一编址。

❑ **UART控制器寄存器接口定义（ benchmark/common/printf.c ）**

```
#define UART_TX_FIFO            0x04
#define UART_STATUS             0x08
#define UART_TX_FIFO_FULL       (1 << 3)

volatile unsigned int *uart = (void *)0x60000000;
```

UART发送数据队列入口寄存器偏移地址
UART队列状态寄存器偏移地址
UART发送数据队列状态标志位掩码
UART控制器基地址指针（地址不可修改）

图 9: UART 控制器寄存器接口定义

从上图定义的接口,我们首先要找到寄存器的首地址为 UART 基地址 + 入口寄存器偏移地址,然后判断状态寄存器是否满,如果满的话必须等待,不满的话则可以将数据写入其中,然后进行下一个循环。

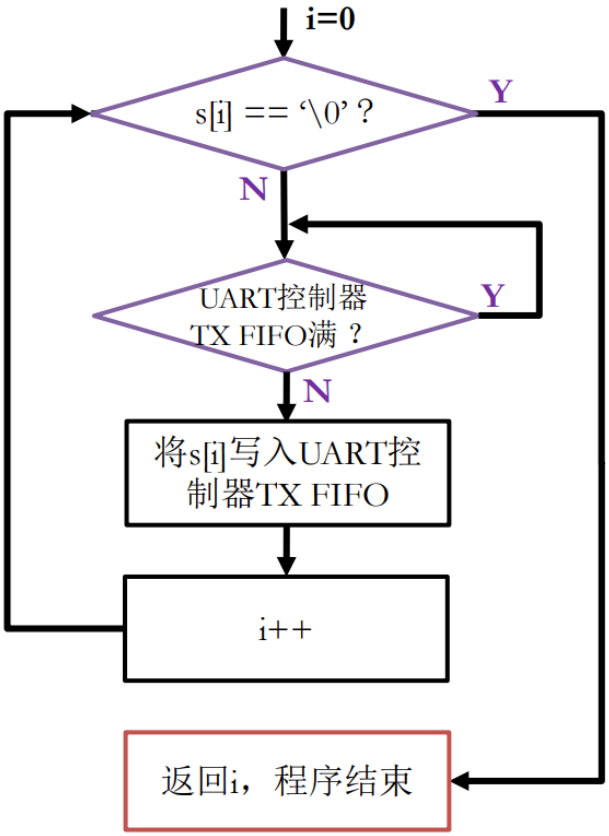
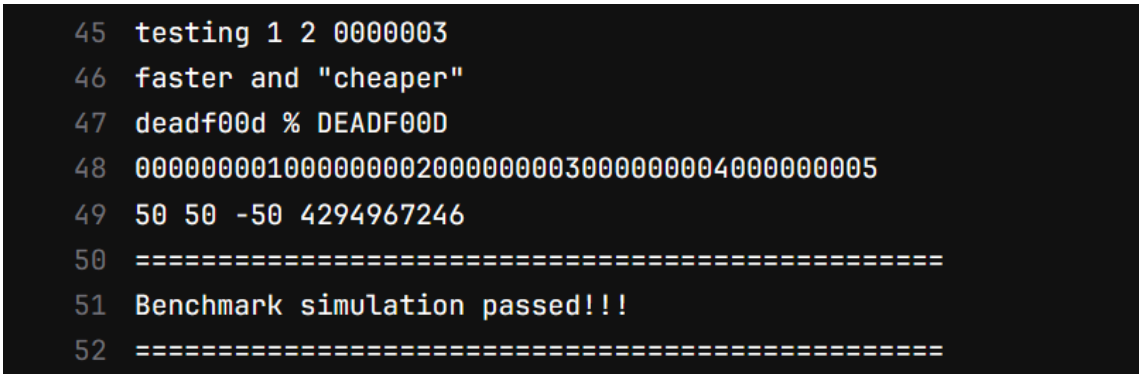



图 10: 写数据的流程图

```c
int
puts(const char *s)
{
    //TODO: Add your driver code here
    int i = 0;
    while(s[i]!='\0'){
        //将uart转化为char*类型，因为要以字节为单位进行偏移。然后将UART_STATUS加到该地址上，得到指向UART状态寄存器的地址。
        while((*(volatile unsigned int *)((char *)uart+UART_STATUS)) & UART_TX_FIFO_FULL)   //确保读到最新的UART状态，并判断UART控制器TX FIFO是否已满
            ;                                   //若已满，则一直等待直到不满
        *((char *)uart + UART_TX_FIFO) = s[i];      //将s[i]写到UART控制器TX FIFO中
        i++;
    }
    return i;                                   //返回打印字符串的字符个数
}
```

图 11: puts() 函数



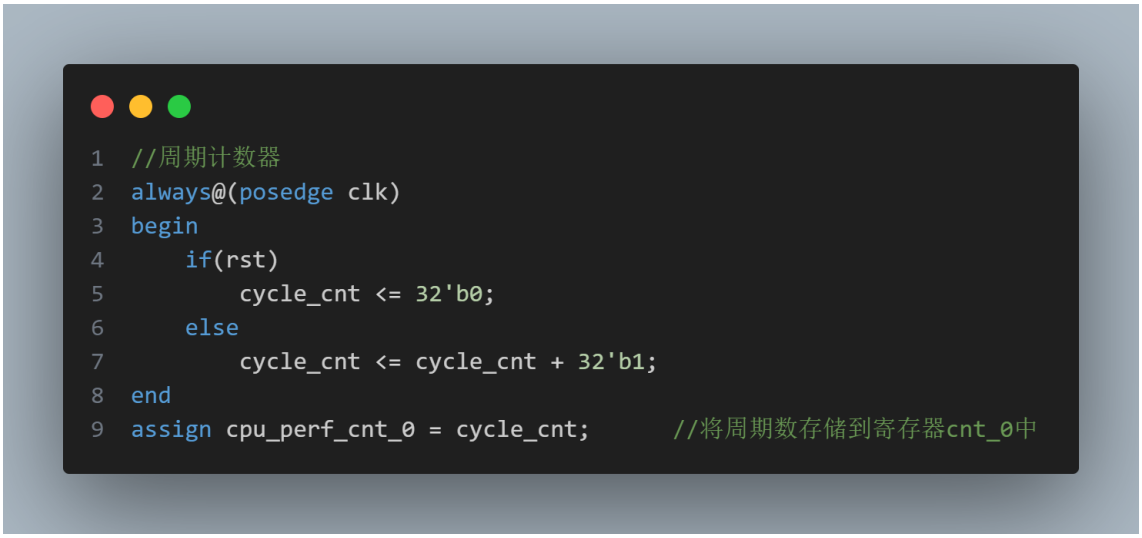
```
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
00000000100000000200000000300000000400000000 5
50 50 -50 4294967246
====================================================
Benchmark simulation passed!!!
====================================================
```

图 12: [hello,hello] 打印成功截图

• **性能计数器实现及访问方法**。

在 CPU 或计算机系统的底层性能在线统计、分析和调优的过程中，我们需要性能计数器来统计指令执行的硬件行为。

在所给的部分代码中，已在 custom_cpu.v 模块顶层预留了性能计数器接口，本部分中我实现了两个性能计数器——时钟周期数计数器，指令数计数器。



```c
//周期计数器
always@(posedge clk)
begin
    if(rst)
        cycle_cnt <= 32'b0;
    else
        cycle_cnt <= cycle_cnt + 32'b1;
end
assign cpu_perf_cnt_0 = cycle_cnt;      //将周期数存储到寄存器cnt_0中
```

图 13: 周期计数器代码

```
//指令数计数器
always@(posedge clk)
begin
    if(rst)
        inst_cnt <= 32'b0;
    else if(Inst_Ready && Inst_Valid)
        inst_cnt <= inst_cnt + 32'b1;
end
assign cpu_perf_cnt_1 = inst_cnt;       //将指令个数存储到寄存器cnt_1中
```

图 14: 指令数计数器代码

除了在硬件部分实现以外，还需要在软件部分进行实现，下图为实现计数器的一般步骤。

❑ **步骤1：修改Result结构体定义，添加统计变量**

- 例如：添加unsigned long mem_cycle，统计内存访问周期

❑ **步骤2：参考_uptime()，添加其他性能计数器的访问函数**

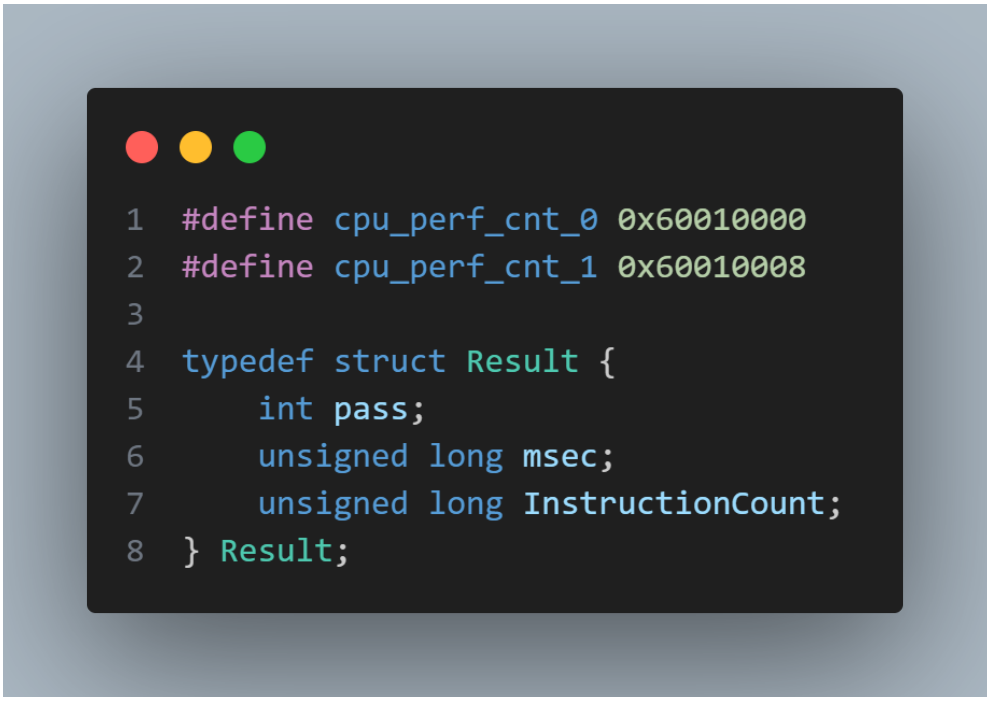
- 例如：添加_read_mem_cycle()函数

❑ **步骤3：在bench_prepare()和bench_done()两个函数中，添加对计数器访问函数（如_read_mem_cycle()函数）的调用**

❑ **步骤4：在main()函数结尾，添加相应的结果显示打印功能**

图 15: 访问计数器的一般步骤

具体来说，首先修改 perf_cnt.h 部分，添加所要用到的性能计数器接口，然后修改 Result 结构体，添加相对应的变量用来与对应的接口连接(见下图 16)。



```
#define cpu_perf_cnt_0 0x60010000
#define cpu_perf_cnt_1 0x60010008

typedef struct Result {
    int pass;
    unsigned long msec;
    unsigned long InstructionCount;
} Result;
```

图 16: 步骤 1 部分代码

然后,在 perf_cnt.c 中,我们首先修改 _uptime() 函数,该函数的作用是返回 cpu_perf_cnt_0 的值,同样我们仿照该函数,又添加了一个、_upinst() 函数,来返回 cpu_perf_cnt_1 的值。

```
unsigned long _uptime() {
  // TODO [COD]
  //   You can use this function to access performance counter related with time or cycle.
  volatile unsigned long *cycle_cnt = (unsigned long*)cpu_perf_cnt_0;   //将第0个性能计数器转换为cycle_cnt的类型并赋值给cycle_cnt
  return *cycle_cnt;
}

unsigned long _upinst() {
  volatile unsigned long *inst_cnt = (unsigned long*)cpu_perf_cnt_1;    //将第1个性能计数器同样转换类型并赋值给inst_cnt
  return *inst_cnt;
}
```

图 17: 步骤 2 部分代码

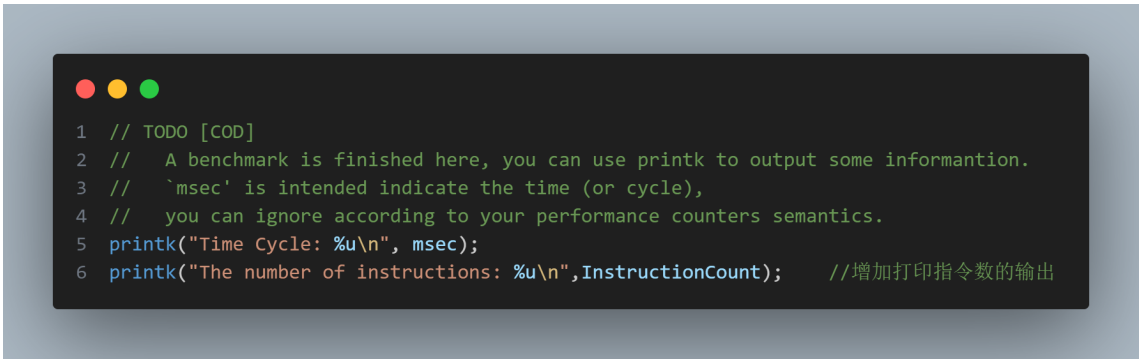
再修改 bench_prepare() 和 bench_done() 函数,添加对计数器访问函数的调用。

```
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`
  //调用函数，将性能计数器与对应的接口连接
  res->msec = _uptime();
  res->InstructionCount = _upinst();
}

void bench_done(Result *res) {
  // TODO [COD]
  //  Add postprocess code, record performance counters' current states.
  //在统计时，要获取当前计数器的值，则应当用总值减去初值
  res->msec = _uptime() - res->msec;
  res->InstructionCount = _upinst() - res->InstructionCount;
}
```

图 18: 步骤 3 部分代码

最后在 bench.c 代码中，修改 main() 函数部分内容，使其完成相应结果的打印。



```
// TODO [COD]
//   A benchmark is finished here, you can use printk to output some informantion.
//   `msec' is intended indicate the time (or cycle),
//   you can ignore according to your performance counters semantics.
printk("Time Cycle: %u\n", msec);
printk("The number of instructions: %u\n",InstructionCount);    //增加打印指令数的输出
```

图 19: 步骤 4 部分代码

同时，在 main() 函数的其他部分，仿照周期计数器来添加相对应的内容。

```c
if (msg != NULL) {
  printk("Ignored %s\n", msg);
} else {
  unsigned long msec = ULONG_MAX;
  unsigned long InstructionCount = ULONG_MAX;   //增加InstructionCount来统计指令数
  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.InstructionCount < InstructionCount) InstructionCount = res.InstructionCount;  //增加更新指令数的条件判断
  }
```

图 20: 步骤 4 部分代码

```
190  Time Cycle: 53820176
191  The number of instructions: 728289
192  benchmark finished
193  time 1681.37ms
```

图 21: 两个计数器统计结果

上图为两个计数器的统计结果，可以看出指令数远小于周期数，满足多周期 cpu 的特点。

## 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）

- **处理器真实内存访问通路设计**。

本部分的主要问题就是状态转移图的实现，虽然从图 3 中我们已经可以获得绝大部分信息，但是在细节处理上还是会遇到一些困难。在我遇到的困难中，最大在某一时刻取指令错误，从而导致 PC 错误的跳转，然后导致 PC 值和金标准的 PC 值不一致。

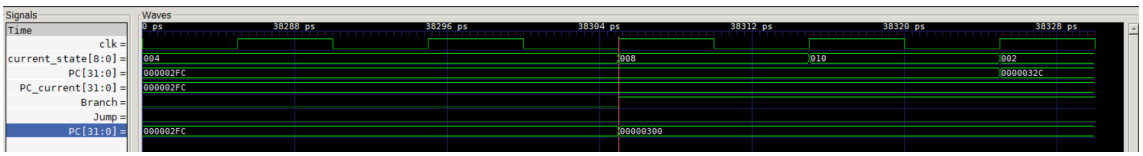



图 22: PC 值错误的波形图

从波形图中我们看到，我的 PC 值在 38ns 左右时，当状态在 ID 阶段时，Branch 信号突然升高，表面译码认为当前读到的指令为分支指令，但是从金标准中我们可以看出实际指令应该是非跳转类指令，即在 EX 阶段时 PC 正常加 4，但是由于我的代码判断出来的指令是 Branch 类，导致在 EX 阶段按照相应的跳转地址进行错误跳转。

找到了错误的地方之后，我仔细检查了自己的状态转移部分的代码，并进行了多次尝试，始终没有找到答案，这时我想到了是不是可能是前面实验二部分的代码存在问题，但是当时的测试部分没有检验出来。然后我最终找

到了在下图 23 中的代码部分存在了问题。

```
1  //寄存器读写操作
2  assign RF_wen = (opcode == 6'b000001 || opcode[5:2] == 4'b0001 || (opcode[5] && opcode[3])
3          || (opcode == 6'b0 && func == 6'b001000)) || (opcode == 6'b000010) ? 0     //REGIMM指令/I-Type分支指令/I-Type内存写指令/R-Type JR指令/J-Type J指令
4           : (opcode == 6'b0 && func[5:1] == 5'b00101) ? func[0] ^ (RF_rdata2 == 32'b0)          //R-Type MOVZ/MOVN指令
5           : ((opcode == 6'b000011 && current_state == EX) || (current_state == WB)
6          || (opcode == 6'b0 && func == 6'b001001 && current_state == EX)) ? 1                  //J-Type JAL指令/R-Type JALR指令
7           : 0;
```

图 23: 实验二部分中存在问题的代码

代码中第四行的部分一开始是在倒数第二行的位置，由于该部分内容使用的是三目运算符来进行选择的，该算符表达式中当找到一个条件满足之后，就不会判断后续的条件，因此当有编码重合或者冲突的时候，条件判断的先后顺序会影响程序的正确性。

- **外设控制器访问方法与字符串打印功能实现部分和性能计数器实现部分**。

后两部分主要是软件的编写，和一小部分硬件的编写，在实现第二个性能计数器部分，虽然绝大部分都是仿照时钟周期计数器的内容来写我的第二个指令数计数器的内容。

在硬件部分，我一开始写的时候没有仔细考虑什么时候需要增加指令数，而是直接照搬时钟数的内容，导致了错误(见下图 24)。

```
190  Time Cycle: 56970168
191  The number of instructions: 56970546
192  benchmark finished
193  time 1745.91ms
```

图 24: 性能计数器部分中存在的问题

可以看出指令数和周期数存在明显的错误。当我修改了代码(见图 14)之后，则得到了合理的结果(见图 21)。

## 三、 对讲义中思考题的理解和回答

- **思考题 1：下图中 volatile 关键字的作用是什么？如果去掉会出现什么后果？**。

❑ **UART控制器寄存器接口定义（ benchmark/common/printf.c ）**

```
#define UART_TX_FIFO          0x04
#define UART_STATUS           0x08
#define UART_TX_FIFO_FULL     (1 << 3)

volatile unsigned int *uart = (void *)0x60000000;
```

UART发送数据队列入口寄存器偏移地址
UART队列状态寄存器偏移地址
UART发送数据队列状态标志位掩码
UART控制器基地址指针（地址不可修改）

图 25: 思考题部分

volatile 的本意是“易变的”，因为访问寄存器要比访问内存单元快的多，所以编译器一般都会作减少存取内存的优化。当要求使用 volatile 声明变量值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据，所以当遇到这个关键字声明的变量时，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

因为图中括号部分要求地址不可修改，所以我们可以使用 volatile 来声明该指针，这样该内容就不会因为系统优化而被改变。

如果去掉该关键字，则该指针所指向的地址可能会出错。

## 四、 实验所耗时间

在课后，你花费了大约____8____ 小时完成此次实验。

## 五、 实验心得

该实验的主要部分是设计多周期处理器的状态转移图，我们通过数字电路课程中所学的三段式来完成状态转移图的设计，然后通过对实验二代码的复用完整了大部分的内容。但是在实验二中有一个没有发现的错误在实验三中被发现了，这表明测试并不一定能够完全正确的判断代码的正确性，这就要求我们在 debug 的时候不能想着自己复用以前的代码一定是完全正确的，当发现错误并且在检查完自己现阶段写的内容并没有错误的时候，可以去向前检查一下，有可能发现没有发现的错误。

对于该实验，整体难度并不大，因为绝大部分都是复用的代码，因此我们应当在写代码的时候考虑到其可复用性，添加详细的注释有助于自己以后或者别人的阅读。我们还应当规范的写自己的代码，对于缩进、变量的命名等都应该养成良好的习惯。