中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009925006　姓名：冯浩瀚　专业：计算机科学与技术

实验序号：03　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 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（比如关键 RTL 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，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

1. 处理器真实内存访问通路

该部分相比上个实验中单周期 mips 处理器的代码最大的变动是加入一个三段式状态机。

第一段用独热码将 9 个状态编码，并且用时序逻辑描述状态寄存器的同步状态改变。

```
// decode the state of the FSM using one-hot encoding
parameter INIT = 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;
reg [8:0] current_state, next_state;
reg [31:0] valid_Instruction, valid_Read_data;
// part 1
always @(posedge clk ) begin
        if(rst) current_state <= INIT;
        else current_state <= next_state;
end
```

第二段用 always 块的组合逻辑描述状态机的状态根据当前状态以及其他输入信号的改变。此处注意 case 语句要有 default 避免出现锁存器

```
// part 2
always @(*) begin
        case (current_state)
                INIT: next_state = IF;
                IF: if(Inst_Req_Ready) next_state = IW; else next_state = IF;
                IW: if(Inst_Valid) next_state = ID; else next_state = IW;
                ID: if(valid_Instruction[31:0] == 32'b0) next_state = IF; else next_state = EX;
                EX: if(type == `ILoad) next_state = LD;
                        else if(type == `IStore) next_state = ST;
                        else if (type == `REGIMM || type == `IBranch || opcode[5:0] == 6'b000010)
                                next_state = IF;
                        else next_state = WB;
                LD: if(Mem_Req_Ready) next_state = RDW; else next_state = LD;
                RDW: if(Read_data_Valid) next_state = WB; else next_state = RDW;
                WB: next_state = IF;
                ST: if(Mem_Req_Ready) next_state = IF; else next_state = ST;
                default: next_state = INIT;
        endcase
end
```

第三段用 always 时序逻辑以及 assign 组合逻辑，根据状态机当前状态，描述握手信号以及添加的 valid_Instruction 和 valid_Read_data 寄存器的同步变化。此二者保证了指令和读出的数据随着状态发生变化而变化或保持不变。

```
// part 3
assign Inst_Req_Valid = current_state == IF;
assign Inst_Ready      = (current_state == IW || current_state == INIT);
assign Read_data_Ready = (current_state == RDW || current_state == INIT);
assign MemRead         = current_state == LD;
assign MemWrite        = current_state == ST;
always @(posedge clk) begin
        valid_Instruction <= (Inst_Ready && Inst_Valid)? Instruction: valid_Instruction;
end
always @(posedge clk) begin
        valid_Read_data <= (Read_data_Ready && Read_data_Valid)? Read_data: valid_Read_data;
end
```

此外，还有一些细节也发生了变动。例如 RF_wen 等在单周期 cpu 中根据指令类型决定信号高低的使能信号在实验中应该改为由当前状态决定信号高低。

```
assign RF_wen = (type ==`IStore || (type ==`JType && opcode[0] == 0) ||
type == `IBranch || type == `REGIMM || type == `RJump && func[0] == 0 ||
type == `RMove && ~func[0] ^ Zero || current_state != WB)? 1'b0:1'b1;
```

2. 外设控制器访问方法与字符串打印功能实现

这里将*uart 强制类型转换成 char 类型指针，保证了指针加 1 同时地址偏移量也是 1。并且用 volatile 关键字使得编译器对访问该变量的代码不再进行优化，从而可以提供对地址的稳定访问。

```
int
puts(const char *s)
{
        //TODO: Add your driver code here
        unsigned int i = 0;
        while(s[i]){
                while (*((volatile char*)uart + UART_STATUS) & UART_TX_FIFO_FULL) ;
                *((volatile char*)uart + UART_TX_FIFO) = s[i];
                i++;
        }
        return i;
}
```

3. 处理器核性能计数器及性能评估方法

我选择加入的性能计数器是周期计数器，因此不需要在 perf_cnt.h 中新增结构体，只需修改 perf_cnt.c 中代码如图所示，修改_uptime()函数，在 bench_prepare 和 bench_done 中调用，计算 msec 值，最终在 bench.c 中使用 printk()函数输出。

```c
#include "perf_cnt.h"
#define cnt_addr_0 0x60010000
unsigned long _uptime() {
  // TODO [COD]
  //   You can use this function to access performance counter related with time or
  cycle.
  volatile unsigned long *Cycle_cnt = (volatile unsigned long*)cnt_addr_0;
  return *Cycle_cnt;
}

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();
}

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

```
// performance counter
// The number of cycle
reg [31:0] cycle_cnt;
always @(posedge clk) begin
        if(rst) cycle_cnt <= 32'd0;
        else cycle_cnt <= cycle_cnt + 32'd1;
end
assign cpu_perf_cnt_0 = cycle_cnt;
```

```
// 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 number of Cycle is %u\n", msec);
```

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）

1. 上次实验单周期 mips 处理器中存在的 bug 在测试样例中没有显现，在本次试验的测试样例中显现：

例如在描述 REGIMM 类型以及 I 类型分支指令中是否分支的 isbranch 时出现错误。

```
// branch control
wire isbranch = ((type == `REGIMM && REG[0] ^ ~Zero ) ||
                 (type == `IBranch && opcode[1] == 0 && opcode[0] ^ Zero) ||
                 (type == `IBranch && opcode[1:0] == 2'b10 && (~Zero || RF_rdata1 == 32'b0)) ||
                 (type == `IBranch && opcode[1:0] == 2'b11 && Zero && RF_rdata1 != 32'b0) )? 1'b1:1'b0;
```

2. 新增真实内存访问通路后出现的 bug：

例如需要新增 PC_pre 寄存器来存储未到达 IF 状态时的 PC 值，在 IF 状态再更新为 PC，在最初编写代码时未考虑到，使得 J 类型指令的 PC 值总出现错误，仔细思考后改正。

三、 对讲义中思考题（如有）的理解和回答

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

```
#define UART_TX_FIFO          0x04        → UART发送数据队列入口寄存器偏移地址
#define UART_STATUS           0x08        → UART队列状态寄存器偏移地址
#define UART_TX_FIFO_FULL     (1 << 3)    → UART发送数据队列状态标志位掩码

volatile unsigned int *uart = (void *)0x60000000;   → UART控制器基地址指针（地址不可修改）
```

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

答：volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改。遇到这个关键字声明的变量，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。当要求使用 volatile 声明的变量的值的时候，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。而且读取的数据立刻被保存。有些变量是用 volatile 关键字声明的。**当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 volatile 声明，该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。volatile 的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值。**

（C/C++ 中 volatile 关键字详解 | 菜鸟教程 (runoob.com)）

去掉 volatile 后，编译器将从 cpu 寄存器中访问 uart+UART_STATUS 地址，这样每个指令循环后取出的该地址的值可能不会正确改变。

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）

本次实验基于上个实验的代码，总体较为容易。但是 debug 过程较为繁琐：custom_sym 的 shuixianhua 测试样例需要花费 2h 的时间（到后期才意识到如果不需要该测试样例的运行结果可以直接将其 cancel，不影响后续 pipeline），fpga_emu 过程需要反复修改 emu_settings.yml 文件并推送到远程仓库…这些都给实验带来考验。但是这也训练了我通过波形定位 bug 的能力；另外，本次实验“软硬结合”，编写 VHDL 的同时也帮助我回忆巩固 C 语言的知识，还新学习了 volatile 关键字的用，这些都使我受益匪浅。

最终，感谢老师以及助教团队的辛勤付出。