中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2017K8009929044 姓名： 李昊宸 专业： 计算机科学与技术

实验序号：4 实验名称： 性能计数器设计

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 5 的阿拉伯数字。例如：2015K8009929000-prj1.pdf。PDF 文件大小应控制在 5MB 以内。

注 2：使用 git add 命令将 doc 目录下的实验报告 PDF 文件添加到本地仓库，然后 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）

三段式状态机：

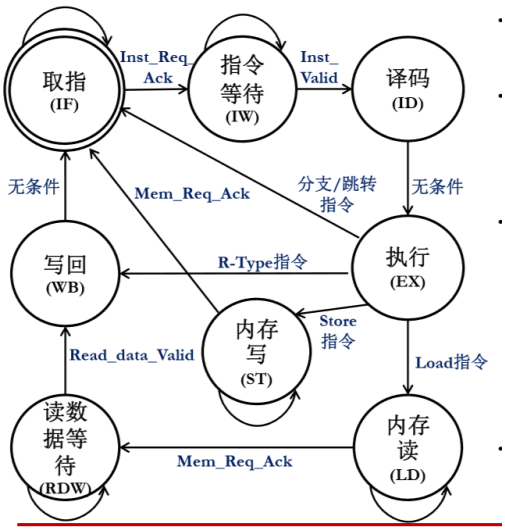
第一段：时序逻辑控制状态跳转

```
always@(posedge clk)                    //control unit
if(rst)                 state <= 5'd0;  //IF
else
                        state <= nextstate;
```

三段式的第一段叙述跳转条件为每一时钟上升沿触发。

第二段：组合逻辑控制 nextstage 跳转信号

```
always@(posedge clk)            //control unit
if(rst)                 state <= 5'd0;  //IF
else
                        state <= nextstate;
always@(*)
begin
case(state)
5'd0:                   if(Inst_Req_Ack) nextstate = 5'd1;   //IW
                        else             nextstate = 5'd0;
5'd1:                   if(Inst_Valid)   nextstate = 5'd7;   //ID
                        else             nextstate = 5'd1;
5'd7:                                    nextstate = 5'd2;   //EX
5'd2:                   if(op_BNE||op_BEQ||op_BGEZ||op_BGTZ||op_BLEZ||op_BLTZ||op_J||op_JAL)     nextstate = 5'd0; //Branch & jump
                        else if(op_LW||op_LB||op_LBU||op_LH||op_LHU||op_LWL||op_LWR)   nextstate = 5'd3; //load
                        else if(op_SW||op_SB||op_SH||op_SWL||op_SWR)                   nextstate = 5'd4; //store
                        else                                                           nextstate = 5'd5; //Write reg_file WB
5'd3:                   if(Mem_Req_Ack)  nextstate = 5'd6; //RDW
                        else             nextstate = 5'd3;
5'd4:                   if(Mem_Req_Ack)  nextstate = 5'd0;
                        else             nextstate = 5'd4;
5'd5:                   nextstate = 5'd0;
5'd6:                   if(Read_data_Valid)  nextstate = 5'd5;
                        else             nextstate = 5'd6;
default:                nextstate = 5'd0;
endcase
end
```



状态机的状态转移表写在状态机的第二段中。采用握手原理，对应路径的 Valid 和 Ack 信号同时拉高时进行数据传输完成的确认，跳转至下一状态。

此外，与实验 3 不同的是，完整多周期状态机需要将 ID 与 EX 阶段分开，并将 Jalr 指令与 jr 指令分开处理，一个作为 J 型，一个作为 R 型，故相应的 RegWrite 信号也将做出调整

第三段：按当前状态机所处状态对各信号赋值。

由于本实验没有 reg 型寄存器的使用，赋值部分全部采用 assign 阻塞赋值，故将 always 块省略。下面是与单周期处理器中有修改的控制信号的 RTL 代码。

```
assign MemRead  =   (state == 5'd3)?1:0;    //LOAD
```

MemRead 在 LD 状态被拉高

```
assign MemWrite =   (state == 5'd4)?1:0; //STORE
```

MemWrite 在 ST 状态被拉高

```
assign RegWrite =   (state == 5'd2&&op_JAL)?1:(state == 5'd5)&&(
```

RegWrite 可以在 EX 段被拉高（来自 jump and link 指令），也可以在 WB 段被拉高（R 型指令）

```
assign Inst_Req_Valid = (RST)?0:(state == 5'd0)?1:0;
assign Inst_Ack       = (RST)?1:(state == 5'd1)?1:0;
assign Read_data_Ack  = (state == 5'd6)?1:0;
```

三个控制信号，分别表示指令请求有效信号，指令接收信号，内存数据读取接收信号。其中为了防止出现“锁死”，在 rst 归零信号拉高时 Inst_Reg_Valid 信号

需要持续为低，Inst_Ack 需要持续拉高。另外，为了代码规范化，在赋值逻辑中没有直接出现 rst 信号，取而代之用另一信号 RST 代替之。

时钟时序逻辑：

```
always@(posedge clk)
begin
if(rst)
PC<=32'b0;
else if(state == 5'd2) PC<=PC2;
else PC <= PC;
end
```

rst 为高时持续为零，在状态机进行到 ID_EX 状态后跳转至下一指令地址。

```
always@(posedge clk)
if(rst)
instruction = 32'd0;
else if(Inst_Valid && Inst_Ack)
instruction = Instruction;
//else instruction <= instruction;

always@(posedge clk)
if(rst)
Read_data_continue = 32'd0;
else if(Read_data_Valid && Read_data_Ack )
Read_data_continue = Read_data;
//else Read_data_continue <= Read_data_continue;
```

两个用于存放当前指令信号和内存读取信号的寄存器。由于这两个信号只会存在一个时钟周期，但其数据需要在整个指令执行期间被使用，所以设置这两个寄存器是有必要的。

puts 函数：

采用指针宏定位地址，val1 为判满信号，temp1 暂存 FIFO 寄存器的高三个字节数据，将其与字符型指针数据相加后返还给 FIFO，实现低字节数据替换。

```
int
puts(const char *s)
{

        //TODO: Add your driver code here
        unsigned int i,a,b;
        unsigned int val;
        unsigned int val1;
        unsigned int temp;
        unsigned int temp1;
        i = 0;
        a = UART_STATUS/4;
        b = UART_TX_FIFO/4;
        for(i=0;*(s+i)!='\0';i++)
        {
            while(1)
            {
              val = *(uart + a);
              val1 = val&UART_TX_FIFO_FULL;
              if(!(val1))
              {
                  temp = *(uart + b);
                  temp1 = temp&0xffffff00;
                  *(uart + b) = temp1 + *(s+i);
                  break;
              }
            }

        }
        return i;
}
```

周期计数器：

```
//cycle_counter
always@(posedge clk)
begin
    if(rst)
        cycle_cnt <= 32'd0;
    else
        cycle_cnt <= cycle_cnt + 32'd1;
end
assign mips_perf_cnt_0 = cycle_cnt;

always@(posedge clk)
begin
    if(rst)
        memory_cnt <= 32'd0;
    else if(Read_data_Valid||Mem_Req_Ack)
        memory_cnt <= memory_cnt + 32'd1;
    else memory_cnt <= memory_cnt;
end
assign mips_perf_cnt_1 = memory_cnt;
```

mips_perf_cnt_0 用于计算时钟周期数

mips_perf_cnt_1 用于计算访存周期数

函数体：

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

unsigned long _uptime() {
  // TODO [COD]
  //   You can use this function to access performance counter related with t
  unsigned long *a =(void *)0x40020000;
  unsigned long b = 0;
  b = *a;
  return b;
}
unsigned long _read_memory_cycle() {
  unsigned long *c =(void *)0x40020008;
  unsigned long d = 0;
  d = *c;
  return d;
}

static 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->mem_cycle = _read_memory_cycle();
}

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

修改结构体 Result，让其含有两个对应计数器的组分，bench_prepare 与 bench_done 分别在 bench 执行开始前与执行结束后计算。其相减得到的值即为真实周期数。

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）

在该实验中遇到的主要问题是 RTL 代码的可综合性与硬件的支持性。比如在该实验中，时序电路的代码书写经常出现的问题是仿真波形正确，bit_stream 正确生成，查看 log 文件也没有错误，但是上板全部 bad trap。经分析后发现可能是 verilog 会将 always 过程块内阻塞赋值的自赋值行为看作可综合，但是在硬件逻辑优化中可能会将其优化删除，这导致了在硬件上该部分逻辑出现时数据信号没有自保存导致错误，这些部分在网上没有查阅到相关资料，为代码的书写和纠错造成了极大的麻烦，

三、 对讲义中思考题（如有）的理解和回答

volatile 关键字：

作用：指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。它表明该语句的值可能会改变，并不一定是一个确定的赋值常量。

下面是几个使用 volatile 的例子：

1）并行设备的硬件寄存器（如：状态寄存器）

2）一个中断服务子程序中会访问到的非自动变量（Non-automatic variables)

3）多线程应用中被几个任务共享的变量

printf 本质上为一次系统中断服务，如果在主程序中出现了 uart 指针关键字被宏定义的话可能导致该部分出现问题，因宏是控制全局的语句，此处使用 volatile 语句可保证在 puts 子函数调用中保证调用值是在程序中声明的值，而非外部调用值。

四、 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）

实验本身充分理解之后不难，但是缺少必要的参考材料，如哪些语句是不可综合的，哪些语句是不符合逻辑规范的等等。