中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K8009907015　姓名：高鸣驹　专业：计算机科学与技术

实验序号：03 实验名称：内存与外设通路设计与处理器性能评估

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在本地仓库的主目录下。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 4 的阿拉伯数字。例如：2019K8009929000-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：学号-prj5-projectname.pdf，例如：2019K8009929000-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、相应信号的仿真波形和信号变化的说明等）

1. 访问真实内存控制器

首先，使用 One-Hot 编码，代码如下图所示：

```
//one hot code to describe state
parameter RST  = 9'b000000001;
parameter IF   = 9'b000000010;
parameter IW   = 9'b000000100;
parameter ID   = 9'b000001000;
parameter EX   = 9'b000010000;
parameter LD   = 9'b000100000;
parameter ST   = 9'b001000000;
parameter RDW  = 9'b010000000;
parameter WB   = 9'b100000000;
```

状态机采用三段式：

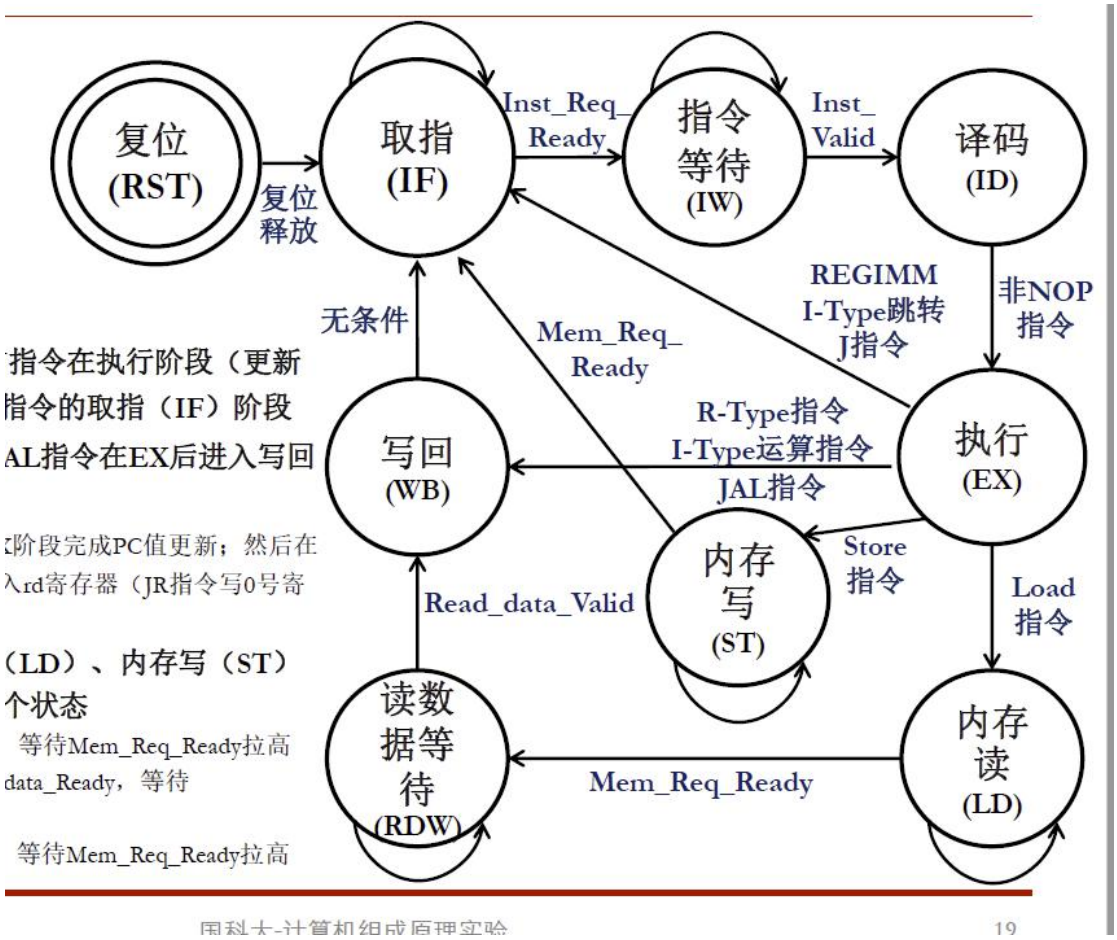
第一段：

```
//The first stanza
always @(posedge clk) begin
    if(rst) begin
        current_state <= RST;
    end else begin
        current_state <= next_state;
    end
end
```

第二段（核心部分）：

```
//The second stanza
always @(*) begin
    case(current_state)
        RST:
            next_state = IF;
        IF:
            if(Inst_Req_Ready) begin
                next_state = IW;
            end else begin
                next_state = IF;
            end
        IW:
            if(Inst_Valid) begin
                next_state = ID;
            end else begin
                next_state = IW;
            end
        ID:
            if(Instruction1[31:0] == 32'b0) begin
                next_state = IF;
            end else begin
                next_state = EX;
            end
        EX:
            if(Type == `TYPELOAD) begin
                next_state = LD;
            end else if(Type == `TYPESTORE) begin
                next_state = ST;
            end else if(opcode == `REGIMM || opcode[5:2] == `IBRANCH || opcode == 6'b000010) begin
                next_state = IF;
            end else begin
                next_state = WB;
            end
        LD:
            if(Mem_Req_Ready) begin
                next_state = RDW;
            end else begin
                next_state = LD;
            end
        RDW:
            if(Read_data_Valid) begin
                next_state = WB;
            end else begin
                next_state = RDW;
            end
        WB:
            next_state = IF;
        ST:
            if(Mem_Req_Ready) begin
                next_state = IF;
            end else begin
                next_state = ST;
            end
        default:
            next_state = RST;
    endcase
end
```

第二段是状态机比较重要的一部分，具体写的思路如下图所示：



这个图来自于 PPT，充分体现了访问内存时的握手机制，比如在取指令的时候只有当 Inst_Valid 和 Inst_ready 同时拉高的时候，next_state 才是 ID，照着上面这个状态转移图写，深刻理解一下握手机制，状态机的第二段很好写。

第三段（核心部分）：

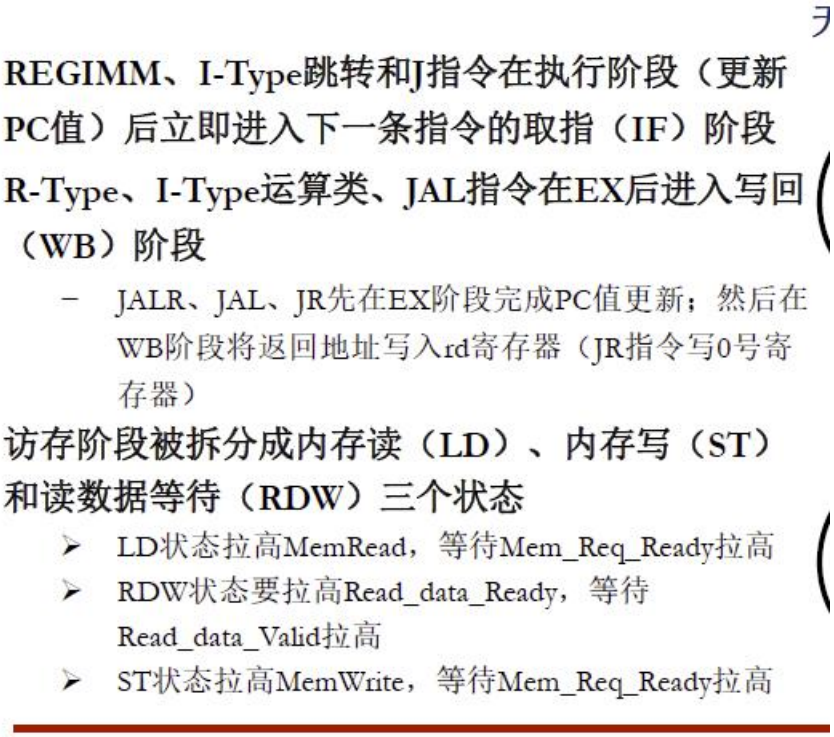
第三段也是状态机比较重要的一部分，这一部分主要是体现访问真实内存的握手机制的，具体有以下几点：

(1)

```
//The Third Stanza
assign Inst_Req_Valid  = current_state == IF;//Only if instruction
assign Inst_Ready      = (current_state == IW || current_state == RST);
assign Read_data_Ready = (current_state == RDW || current_state == RST);
assign MemRead         = current_state == LD;//Load Instructions
assign MemWrite        = current_state == ST;//Store Instructions
```

第一点是一些控制（握手）信号的定义与连接，比如当处于取指

阶段的时候 Inst_Req_Valid 信号要拉高，当处于 LD 阶段的时候，MemRead 信号要拉高。这些信号什么时候拉高什么时候拉低 PPT 中有明显的提示与解释，深刻理解这些信号的含义后根据 PPT 内容进行信号赋值即可：



（PPT 中关于握手信号的赋值的说明）

(2) 握手的实现：

```
//shake hands
always @(posedge clk) begin
    /*If shake hands successfully, then change Instructions, else do not change*/
    Instruction1 <= (Inst_Ready && Inst_Valid)? Instruction:Instruction1;
end

always @(posedge clk) begin
    /*If shake hands successfully, then change Read_data, else do not change*/
    Read_data1 <= (Read_data_Ready && Read_data_Valid)? Read_data: Read_data1;
end

//PC_value change
always @(posedge clk) begin
    if(rst) begin
        PC <= 32'b0;
    end else if(current_state == EX) begin
        PC <= Jump? Jumpaddr:((Branch & branchen)? branch_PC: PC_4);
    end else if(Instruction1 == 32'b0 && current_state == ID) begin
        PC <= PC_4;
    end else
        PC <= PC;
end
```

（握手实现的 RTL 代码）

握手的实现起来比较简单，具体就是相应的 valid 和 ready 信号同时拉高之后，从内存取出来的数据才有效。这体现在取指和执行阶段(Load 指令)。取指阶段，只有当 Inst_valid 和 Inst_Ready 同时有效的时候，指令才能被更新取出，在设计中，我不再用之前的 Instruction 信号来表示真正的指令，取而代之的是 Instruction1 信号，这个信号记录的是真实的指令（仅在握手成功时更新)。同理，Read_data 也做相应的处理。

接下来是处理 PC，PC 的跳转和加 4 只可在 EX 阶段完成，然而有一种特殊情况，就是为 NOP 指令的情况，这时没有 EX 阶段，最后只能到 ID 阶段，但是 PC 依然要完成 PC+4 的工作，否则会陷入死循环，因此这条指令特殊处理，如上图的 Verilog 所示。

(3) 一些细节：

还有一些细节需要注意，比如说：

```
assign RF_wen = (Type == `TYPEBRANCH || Type == `TYPESTORE || opcode == 6'b000010
                || (Type == `TYPEMOVE && funccode[0] == 1'b1 && Zero == 1)
                || (Type == `TYPEMOVE && funccode[0] == 1'b0 && Zero == 0) || current_state != WB)? 1'b0: 1'b1;
```

（RF_wen 信号的描述）

RF_wen 信号什么时候拉高需要更改，当不为 WB 状态的时候这个信号需要拉低。

```
//To Store Pre_PC
always @(posedge clk) begin
    if(current_state == IF) begin
        pre_PC <= PC;
    end else begin
        pre_PC <=  pre_PC;
    end
end
```

```
assign RF_wdata = (Type[3:1] == 3'b000)? ALU_result:
                  ((Type[3:1] == 3'b001)? shifter_result:
                  ((opcode == 6'b000011 || (Type == `TYPEJS && funccode == 6'b001001))? pre_PC + 8:
                  ((Type == `TYPELOAD)? load_result:
                  ((Type == `TYPEMOVE)? rdata1:lui_extend))));
```

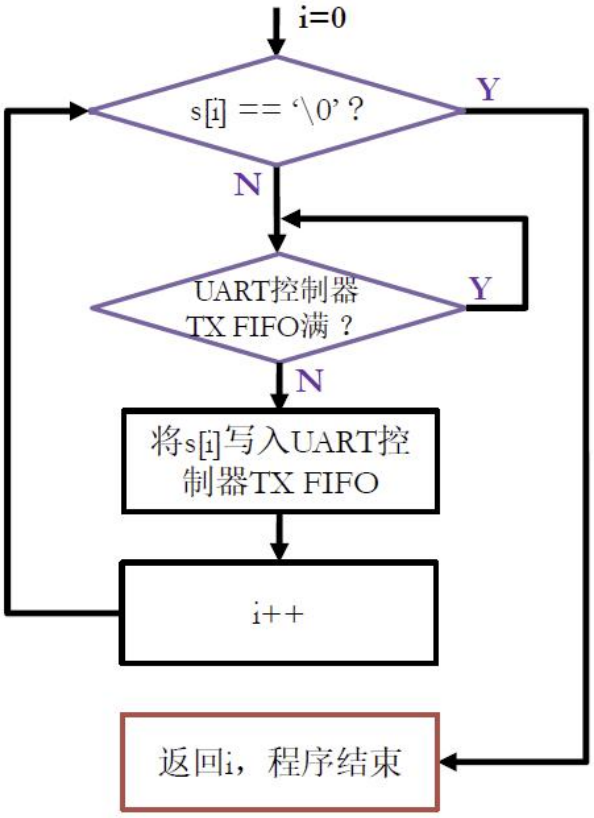
遇到 J-Type 指令并需要想寄存器中写数据时，写的时之前的 PC 加 8，因此需要把之前的 PC 存一下，否则存入的是将要跳转的指令的地址+8.

2. 访问简单 I/O 外设

这个函数相对好写，但是需要注意的点也不少，具体函数如下：

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

这个函数主要根据的是 PPT 中的流程图来设计的：



（puts 函数的流程图）

（1） 判断队满的方式，是用 STAT_REG 里的内容与 0001 进行与操作，如果结果不为 0，说明 STAT_REG 里的内容的第三位不为 0，说明队列已满，这时候要一直持续循环，等到队不满为止

（2） 关于地址的偏移，我这里运用了一个比较巧妙的办法，由于 int 型指针加 1 对应的地址偏移量是 4，而 char 型指针加一对应的地址偏移量是 1，因此我先把 int 类型的指针 uart 强制类型转化成了 char 类型的指针，然后再加上了地址偏移量，这样可以得到正确的地址。同时我前面还加了 volatile 关键字防止编译器的优化。

（3） 由于这是一个字符一个字符读入的，因此队列没满的时候，将 s 数组的字符一个一个读入直到 s[i]为‘\0’，然后返回的 i 即为读

入字符的长度，这样就实现了这个函数的功能。

这个函数写完后在 fpga 测试中的打印如下图：

```
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
00000000100000000200000000300000004000000005
50 50 -50 4294967246
Hit 0 trap
Hit good trap
pass 1 / 1
```

3. 添加性能计数器并对复杂 benchmark 进行性能评测

这一部分我利用了四个接口添加了四个性能计数器，分别计算周期数，指令数，访存数，成功的分支跳转数，具体的 RTL 代码如下：

```
/*The number of cycle*/
reg [31:0] cycle_cnt;

always @(posedge clk) begin
    if(rst) begin
        cycle_cnt <= 32'd0;
    end else begin
        cycle_cnt <= cycle_cnt + 32'd1;
    end
end
assign cpu_perf_cnt_1 = cycle_cnt;
```

（周期数计数器）

```
/*The number of visiting mem*/
reg [31:0] Mem_cnt;

always @(posedge clk) begin
    if(rst) begin
        Mem_cnt <= 32'd0;
    end else if((MemRead || MemWrite) && Mem_Req_Ready) begin
        Mem_cnt <= Mem_cnt + 32'd1;
    end else begin
        Mem_cnt <= Mem_cnt;
    end
end
```

（访存数计数器）

```
/*The number of branch*/
reg [31:0] Branch_cnt;

always @(posedge clk) begin
    if(rst) begin
        Branch_cnt <= 32'd0;
    end else if(current_state == EX && branchen && Branch) begin
        Branch_cnt <= Branch_cnt + 32'd1;
    end else begin
        Branch_cnt <= Branch_cnt;
    end
end
```

（成功的分支跳转数计数器）

```
/*The number of Instrutions*/
reg [31:0] Ins_cnt;

always @(posedge clk) begin
    if(rst) begin
        Ins_cnt <= 32'd0;
    end else if(Inst_Ready && Inst_Valid) begin
        Ins_cnt <= Ins_cnt + 1;
    end else begin
        Ins_cnt <= Ins_cnt;
    end
end

assign cpu_perf_cnt_3 = Ins_cnt;
```

（指令数计数器）

具体代码的书写，周期数计数器的代码书写在 PPT 中已有且比较简单，访存次数计数器的代码大体结构和周期计数器的一样，有区别的只是在访存握手成功的时候计数器的值才+1，同理成功的分支跳转数计数器也只有在执行阶段且为 branch 指令且 branchen 拉高的时候才加一，指令计数器也只在取指阶段访存握手成功的时候才加 1。

在软件中对这几个计数器的调用，具体的代码如下图所示：

（1） 扩展结构体的定义：

```
typedef struct Result {
  int pass;
  unsigned long msec;
  unsigned long memnum;
  unsigned long branchnum;
  unsigned long instnum;
} Result;
```

（2） 添加 _upxx()函数

```c
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*)count1addr;
  return *Cycle_cnt;
}

unsigned long _upmem() {
  volatile unsigned long *Mem_cnt = (volatile unsigned long*)count0addr;
  return *Mem_cnt;
}

unsigned long _upbranch() {
  volatile unsigned long *Branch_cnt = (volatile unsigned long*)count2addr;
  return *Branch_cnt;
}

unsigned long _upInst() {
  volatile unsigned long *Inst_cnt = (volatile unsigned long*)count3addr;
  return *Inst_cnt;
}
```

这些函数的书写形式类似，在 PPT 中已经给出了 cpu_perf_cnt 的地址，将这些地址的具体值写在宏定义中，然后直接访问这些地址中的数据即可，这里加入了 volatile 关键字防止编译器的优化。

（3） 在 prepare 和 done 中调用这些函数

```c
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 -> memnum    = _upmem();
  res -> branchnum = _upbranch();
  res -> instnum   = _upInst();
}

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

在运行前调用（2）中所说的那些函数，记录下 cpu_perf_cnt 寄存器中的初值，然后在结束后再次调用，用末值减去初值，便是在运行过程中的计数。

（4） 打印功能

```
    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.memnum < memc) memc = res.memnum;
      if (res.branchnum < brcc) brcc = res.branchnum;
      if (res.instnum < insc) insc = res.instnum;
    }

    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 informantion.
    //   `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);
    printk("The number of Mem visiting is %u\n",memc);
    printk("The number of branch is %u\n",brcc);
    printk("The number of Instructions is %u\n",insc);
  }
}
```

如图所示，最后如果计数器内的数字没有超过 unsigned long 数据类型能表示的最大整数，那么就把它们打印下来。

在 fpga 的运行结果如下图所示：

```
The number of Cycle is 1233041
The number of Mem visiting is 483
The number of branch is 1667
The number of Instructions is 16552
benchmark finished
Hit 0 trap
Hit good trap
```

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）

（1） 实验中最主要的问题是在第一部分的调试中，第一部分的大体框架比较好写，但是写完框架之后基于真实内存和理想内存的区别，有很多细节需要调整，比如说在一些 J-Type 指令中写入寄存器中的地址不是 PC+8 而是 pre_PC+8。这些调试工作主要通过看波形解决，但这

个实验的波形明显比 prj2 实验的波形要难看，需要锁定 curren_state 然后分析指令，有时不好看出来哪里错了可以调出金标准的波形进行比较从而确定 bug 在哪里。

（2） 实验的第二部分在关于偏移地址的使用第一次也出现了问题，由于没有考虑 int 型指针加 1 其实地址是加 4，导致刚开始对自己的错误百思不得其解，后来调出波形看与金标准基本一样，于是考虑是软件的问题，经过逐句检查最终才发现 bug 的所在。

三、 对讲义中思考题（如有）的理解和回答

思考题：volatile 关键字的作用是什么，如果去掉会出现什么样的后果?

回答：volatile 的中文意思是“易变的”。我们可以看出，在下面代码中：

```
while(*((volatile char*)uart + UART_STATUS) & UART_TX_FIFO_FULL){
```

Uart+UART_STATUS 计算出来的地址实际上没有变的，如果不加 volatile 关键字，经过编译优化后，编译器为了减少内存的访问，实际上看到内存没有变化，就只会进行一次访存。加了 volatile 关键字后，就能消除编译器的编译优化，从而提供了对这个地址的稳定的访问，这样才是我们想要达到的目的。

四、 在课后，你花费了大约 10 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）

这个实验虽然分了三部分，但其实写的代码并不多，主要是对 prj2 中单周

期处理器的一个完善，增加了真实内存的访问，打印和性能计数器等功能。虽然代码量较少，但是细节很多，尤其是第一部分，因此 debug 的时间几乎占了绝大部分时间，同时加上增加了状态机，波形变得更加复杂，需要更多的信号才能锁定错误的位置。第二部分和第三部分的硬件代码写起来较容易，第三部分的软件代码也较容易，第二部分的软件代码需要对软硬件的交互有一些深刻的理解才能防止 bug 的出现(比如说 volatile 关键字的使用)。总体来说，这个实验完善了我们的 cpu，加深了我们对软硬件协同工作的理解，也训练了状态机“三段式”的写法，虽然 debug 的过程很艰难，云平台 bhv 的 31-40 有些测试程序跑的很慢而且经常排队，有时候需要等很久，但是收获也是很大的。