# 中国科学院大学计算机组成原理实验课

# 实 验 报 告

学号：_____2018K8009909006_____ 姓名：__唐宇菲_____ 专业：数学与应用数学_____

实验序号：__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。<br>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）。<br>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。 |

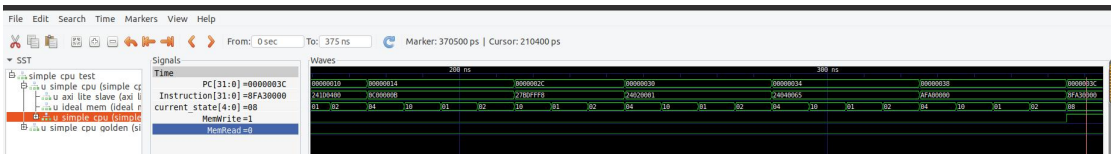
一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图、相应信号的仿真波形和信号变化的说明等）

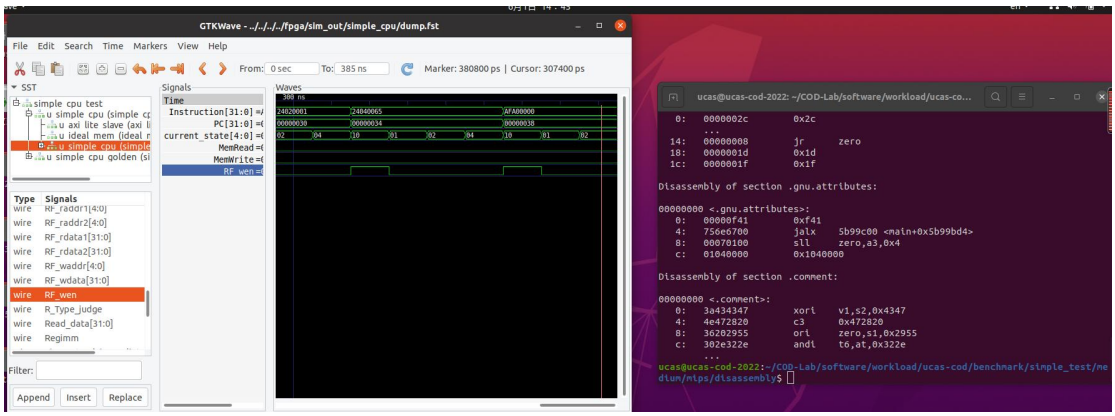
解：首先介绍整体实验的思路，一开始设计的状态机有两个特点

（1）由于 jr 是 R-Type 类型的指令，但其实际只需要跳转到寄存器 R[rs]中存储的 PC 信息即可，所以 jr 不需要经过 WB(写回阶段)，但是仿真之后总是报错

（2）对于 NOP 指令的处理，由于 NOP 指令实际上是一条 addi 类型的指令，属于 I-Type-Compute 类型，在仿真查看后发现，对比的金标准希望我们不要对寄存器 RF_wen 的写使能拉高（但实际上，由于寄存器堆中的 0 号寄存器是只读的，所以即使 RF_wen 拉高，对于结果是没有影响的。）

下图展示了 medium sum 组中最终的报错信息，关于 nop 指令需要在 ID 阶段就进行跳转。






一开始希望，对于所有的指令，PC 都在 EXE 阶段进行同一更新，NOP 指令作为一种 addi 指令，不作特殊处理。

但后续对比金标准，修改为在 IF 阶段统一更新 PC 为 PC + 4，之后的 EXE 阶段在处理 branch 和 jump 类型的指令时，不需要再进行 PC + 4 + offset,直接进行 PC + offset 的处理。

在对 jr 指令处理时，初始代码将其分类到状态机的 ID-> IW -> EXE -> IF,但对比金标准，发现没有作特殊分类，最后还是经过了一个 WB 阶段，尽管没有写回的动作。

对于分支延迟槽的处理，应该是为了使得 Instruction per cycle 变高，所以作的处理，我们现在假设 branch 指令之后跟随一条 nop 指令，而 nop 指令之后跳转到一条 addu（任意）指令。

Branch 对应的 PC 值就设为 PC， nop 指令的 PC 值 为 PC + 4

而我们所说的 offset 实际是 nop 和 addu 之间的 offset,

所以真实的 addu 相对于 branch 的地址是 PC + 4 + offset.

所以当我们设置在 IF 阶段就进行 PC + 4 的更新后，新的 branch 的 PC 值不需要再 + 4 ，换言之，这里并不是 PC + 8。

在调试完毕实验三之后，大概明白了和仿真的区别（出现了 FPGA 上板过但是仿真不过的情况），后来发现主要是 nop 指令、jr 指令、和 PC 在 IF 阶段更新的原因。

之后回过头来修改了实验二选做实验的多周期处理器。

可以说，这里的实验顺序是（完成实验三 -> 明确仿真不过的原因 -> 根据实验三修改实验二多周期处理器).

```
reg [31 : 0] ID_EXE_cpu_alu_A;
reg [31 : 0] ID_EXE_cpu_alu_B;
reg [2 : 0] ID_EXE_cpu_ALUop;
reg [31 : 0] ID_EXE_cpu_shifter_A;
reg [4 : 0] ID_EXE_cpu_shifter_B;
reg [1 : 0] ID_EXE_cpu_shifter_Shiftop;
reg [31 : 0] ID_EXE_cpu_PC_jump;
reg [31 : 0] ID_EXE_cpu_PC_R;
reg [31 : 0] ID_EXE_cpu_PC_branch;
reg ID_EXE_cpu_PC_jump_enable;
reg ID_EXE_cpu_PC_R_enable;
reg [5 : 0] ID_EXE_cpu_PC_branch_enable;

reg [31 : 0] ID_EXE_RF_rdata1;
reg [31 : 0] ID_EXE_RF_rdata2;

reg ID_EXE_Switch_WB;
reg ID_EXE_Switch_IF;


reg ID_EXE_RF_wen_1;
reg ID_EXE_MIPS_movn;
reg ID_EXE_MIPS_movz;
reg ID_EXE_MIPS_jal;
reg ID_EXE_MIPS_lui;
reg ID_EXE_R_Type_judge;
reg ID_EXE_I_Type_Compute_judge;
reg ID_EXE_store_Judge;
reg ID_EXE_load_judge;
```

```
reg EXE_WB_RF_wen_1;
reg EXE_WB_MIPS_movn;
reg EXE_WB_MIPS_movz;
reg EXE_WB_MIPS_jal;
reg EXE_WB_MIPS_lui;
reg EXE_WB_cpu_Zero;
reg EXE_WB_R_Type_judge;
reg EXE_WB_I_Type_Compute_judge;

reg [4 : 0] EXE_WB_rt;
reg [4 : 0] EXE_WB_rd;
reg [31 : 0] EXE_WB_PC_add;

reg EXE_WB_RF_wdata_1;
reg EXE_WB_RF_wdata_2;
reg EXE_WB_RF_wdata_3;

reg [31 : 0] EXE_WB_cpu_Result;
reg [31 : 0] EXE_WB_cpu_shifter_Result;
reg [31 : 0] EXE_WB_RF_rdata1;
reg [15 : 0] EXE_WB_Immediate;


reg [31 : 0] EXE_LD_RF_rdata1;
reg [31 : 0] EXE_LD_RF_rdata2;
reg [31 : 0] EXE_LD_Sign_extend_immediate;
reg [6 : 0]  EXE_LD_load;
reg EXE_LD_load_judge;
reg [4 : 0] EXE_LD_rt;
```

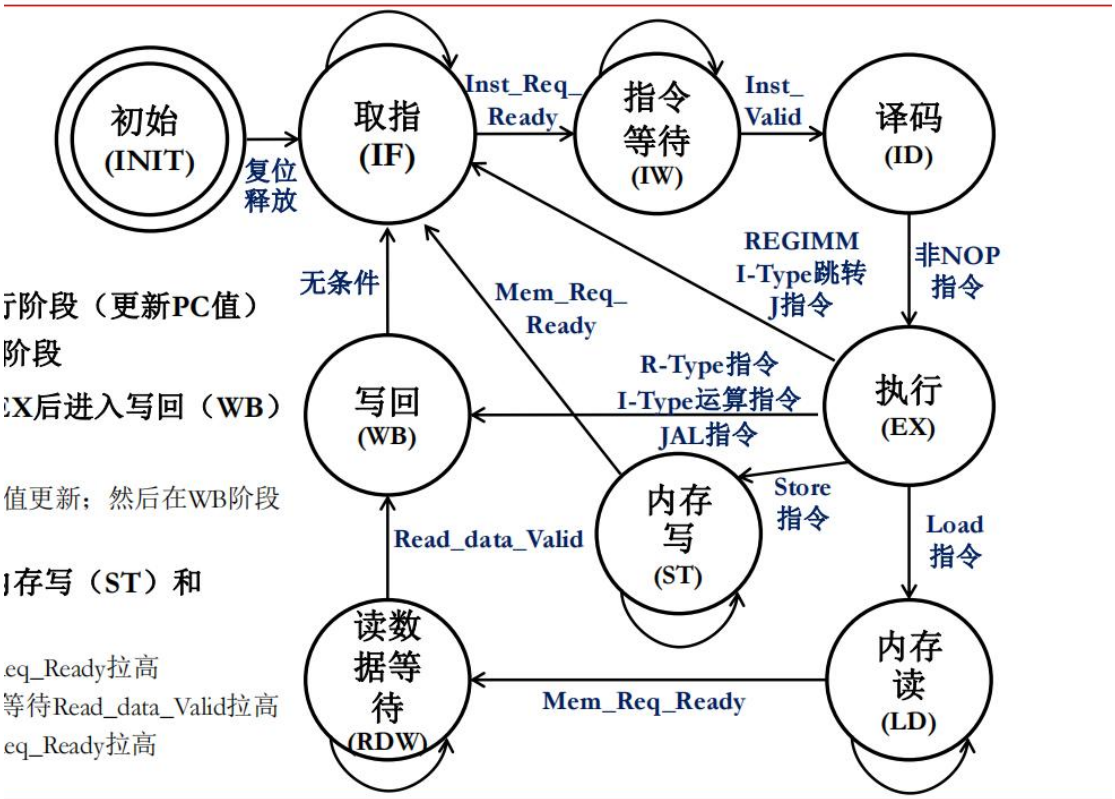
所有关于寄存器变量的定义都是以（当前态）_(下一状态)_实验二中的名字这样来定义。所有的数据线和实验二中类似，添加了状态机和寄存器的部分。

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）

解：状态机跳转如下:

(1) 遇到的问题一：从单周期到多周期之后，一个疑问就是，处理器怎样区分处于哪一个阶段？组合逻辑如何保证信号在对应的阶段有效？是对于组合逻辑逐条判断 current_state= IF/IW/ID/EXE/...吗?

实验三的状态转换设计与老师 ppt 上的状态转换图一致。




解决组合逻辑如何在各阶段有效的问题。
将一条指令的每一个阶段看成是彼此透明的不同模块。每个模块都“看”不到其它模块的信息（实际上电路当然是可见的，但设计的时候想象成是透明的）需要哪些信息，就必须通过不同阶段的寄存器存下数据，然后下一拍再传给我。
（2）问题二：但 PC 值可以看成是全局的，它可以由后面的阶段告诉 IF 立刻要跳转，则直接传递。这时通过对所处状态的限定来完成。
（1）和(2)之间的差异困扰了我很久。例如如果在 IF 统一更新 PC，那么 EXE 得到跳转命令后，是否需要在 EXE - IF 之间再设置一个逆向的寄存器。
理论上可以，但直接设计全局变量更加清楚。
（3） 当 LD 或 LDW 或 ST 在自己往自己跳转的时候，我们不希望 EXE-LD 的寄存器内容被刷新，因为此时这条指令还没有执行完。
问题（3）是在调代码的时候发现的，解决方案是，对于状态进行当前状态和下一状态的判断，如果当前状态是 LD，并且下一状态还是 LD，则我们不希望寄存器更新。
（3）寄存器赋初始值问题，在调代码的时候发现当 I-Type-Compute 下 RF_wen= 0,不能写，但由于是或的逻辑||，而恰好还没有执行任何 Load 指令，此时由于寄存器使能没有赋初始值，Load 对应的 RF_wen = X,
0 || X = X,导致寄存器的 RF_wen 对比出错。
（4）希望在上一条指令执行结束时，相应的使能信号都不要遗留在寄存器里，影响后续的执行。
结合(3)和(4),最终对于寄存器赋值的修改是，当前状态为 IF 时，所有寄存器数据统一清零。达到了赋初值和清除遗留指令的解决。
（5）各阶段跳转时，需要调用对应的寄存器。例如 EXE-WB 的寄存器和 EXE-LD

—WB 的寄存器不是同一时刻，不能混合使用。

三、 对讲义中思考题（如有）的理解和回答

解：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?

```
Launching hello benchmark...
tggetattr: Inappropriate ioctl for device
testing 1 2 0000fadeadf00d % DEADF000000001050 50 -50 429496time 60126.51ms
Hit good trap
```

首先，与 volatile 修饰符对应的修饰符是 register.

一个函数在执行时会使用多个局部变量，包括形参。这些变量使用频度差异较大，编译器会按照使用频度来划分，将常用变量放到闲置寄存器里，使得运算速度得到很大的提高。

不常用的放入内存，每次用到就去内存里拿。一个值存放在寄存器和内存里，访问速度的差别可达数百倍甚至上千倍。可见将变量放在寄存器里的意义还是很大的。

但对于可能被抢占的任务，一个变量放在寄存器里就成了问题，因为被强占后，同一个变量在内存里和在寄存器里的值可能会不一样。

这里的抢占，包括抢占式多任务操作系统里的多进程或多线程调度，也包括嵌入式开发里的中断，底层都是中断。对于非抢占式多任务则不存在这个问题，比如协程。

volatile 的含义就是明确告诉编译器，这个变量在每次访问时，都走内存，而不要用寄存器来缓存。这样在抢占式多任务里，就能确保每次拿到最新的值。

而 register 的意思则相反，告诉编译器这个变量尽可能放到寄存器里，以提高速度。如果寄存器不够用，则还是放回内存里。

实际使用中，如果一个变量仅在函数内使用，或作为值传递的形参，那么适合使用 register 修饰符。而如果是全局变量，且可能被多人任务读写，则应该明确地使用 volatile，比如中断中访问的全局变量，就应该用 volatile。

[寄存器和内存中值不同的原因]

中断发生时，cpu 的动作：

中断发生时，cpu 会立即把当前所有寄存器的值存入任务自己的内存区域里，空出所有寄存器，交给中断去做事。

中断处理函数返回后，cpu 再把需要唤醒的任务的内存区里寄存器部分内容一一载入到寄存器里，并跳转到上次中断的地址，传入 PC 指针。这样任务就可以继续执行了。

关键就是中断结束后恢复到寄存器的值是从任务私有内存区载入的，而不是从原始变量载入的，中断期间对变量的修改就无法立即反应到寄存器里。

四、 在课后，你花费了大约_____30_____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）

解：后续是根据实验三的内容最后完成了实验二的多周期。感觉实验三的限制比较多，例如 nop 指令是否可以使能 RF_wen,nop 指令是否可以在状态机中从 ID 流向 EXE 再流向 WB,
例如 jr 指令实际没有 WB 的操作，是否可以直接从 EXE 与 j 指令一同返回 IF。
又例如是否可以统一在 EXE 阶段更新 PC。
感觉最终的通过需要根据仿真来修改，限制比较多。
感谢张科老师理论课与实验课的仔细讲解。感觉对于时序逻辑部分的理解还需要再深入。