中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009907032 姓名：唐嘉良 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。

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）。

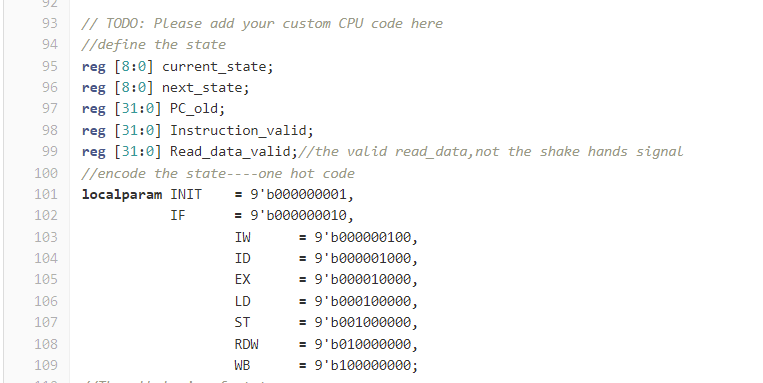
注3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

1. **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）**

本实验无需修改alu以及reg\_file。由于本实验没有什么至关重要的波形说明，debug的时候波形图也未能很好地帮助到我，本部分不放波形图。

本实验主体部分是custom\_cpu.v代码，我们要完成的任务就是将实验2的单周期处理器的主体进行修改，加入状态机将其转变为多周期处理器，并且进行一些多周期的适用性修改。

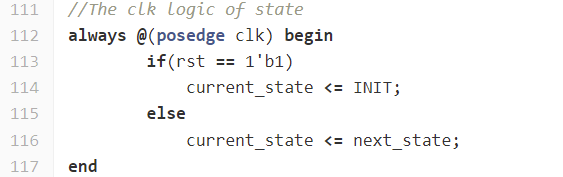
首先，为实现状态机，我们先用verilog语言中的参数类型定义所有状态的独热码（one-hot code），并且考虑到状态机的状态转移，定义current\_state和next\_state两个变量，分别代表当前状态和下一状态。如下所示：



可以看到我还另外定义了PC\_old、Instruction\_valid、Read\_data\_valid三个变量（它们分别与PC、Instruction以及Read\_data具有同样的位宽，其作用将在后面详细说明）。

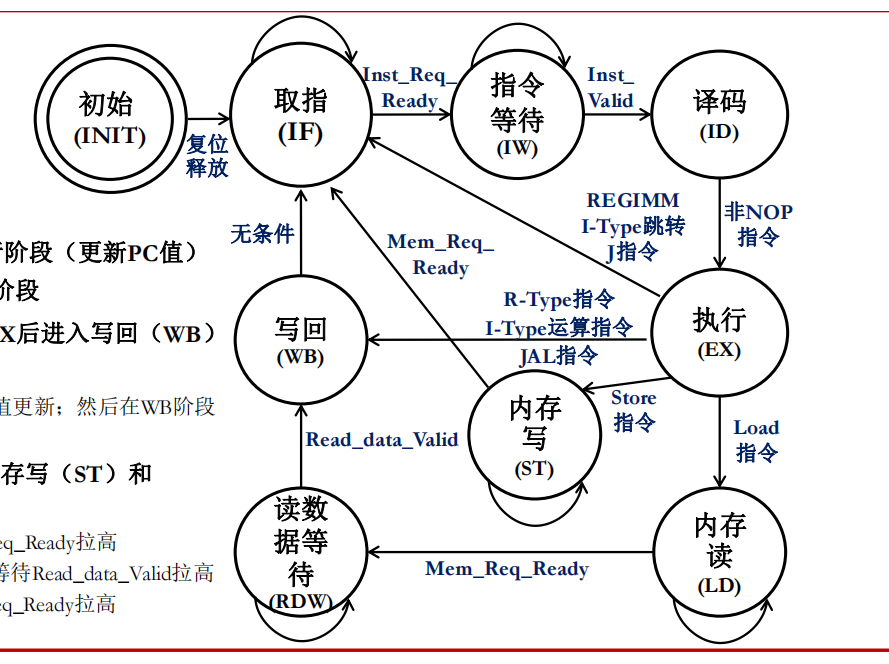
注意到，我没有采用宏定义而是参数类型来定义状态机状态。事实上对于目前的实验框架来说，二者都是可行的，但是参数定义对于含有状态机的项目框架来说更为严谨——因为`define宏定义在编译时自动替换整个设计中所定义的宏，而localparameter仅仅定义模块内部的参数，不会与模块外部的其它状态机混淆——如果两个模块的FSM均含有INIT这个名称的状态，那么宏定义会混淆而localpapameter并不会。

如下所示是FSM第一段的代码。由于我采取三段式写法，按照PPT所示，第一段应当为状态机的状态转移时序逻辑，即在每个时钟上升沿都进行状态转移，如有复位信号则无条件将状态机状态恢复至初始状态INIT。

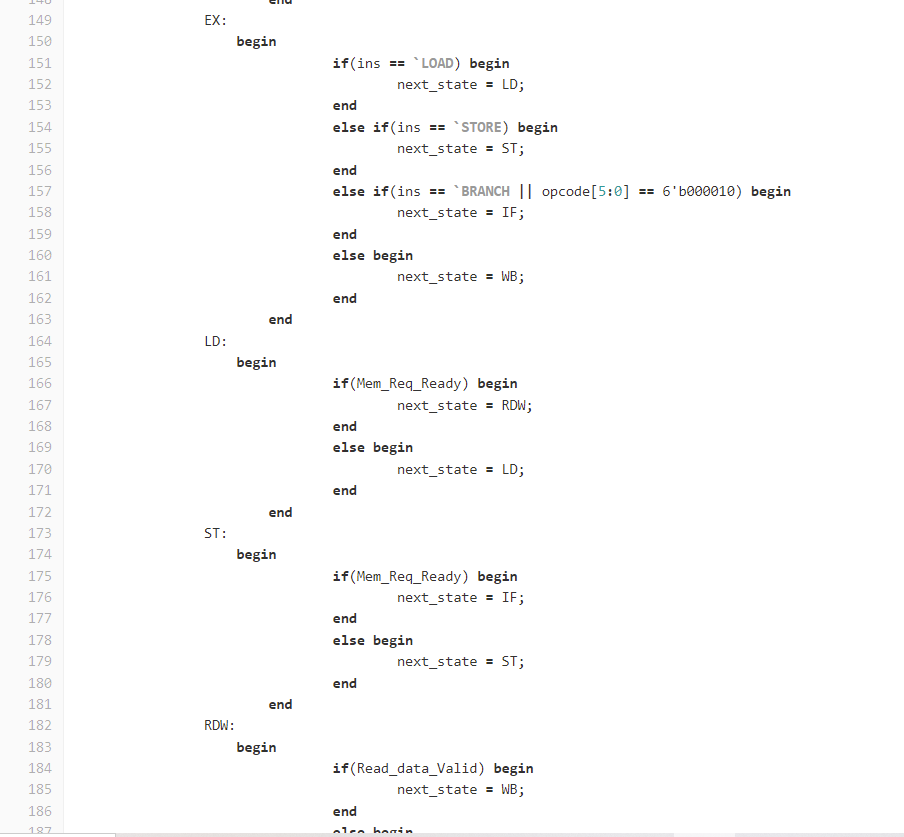


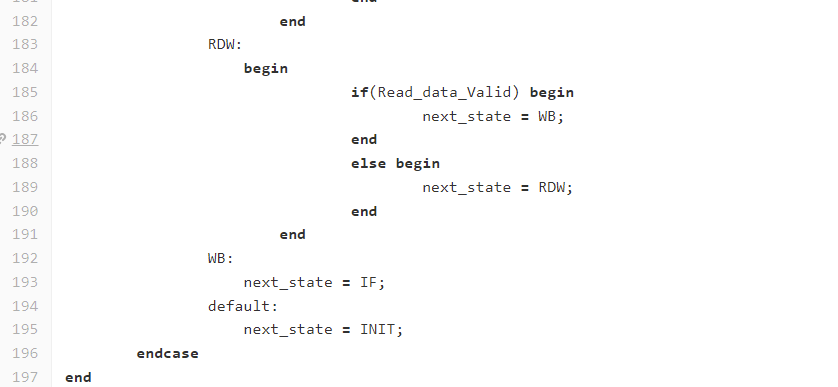
接下来是FSM第二段的代码。这一段的主要工作在于确定状态机的下一状态，采用连续赋值always @(\*)来实现。以IF为例，如果当前处于IF状态，那么要根据Inst\_Req\_Ready信号是否到来选择停留在IF还是进入IW等待接收状态，这时就采用if-else逻辑判断。如果Inst\_Req\_Ready信号到来，那么next\_state就是IW，如果尚未到来，那么仍然停留在IF。其它状态是同理的，根据PPT上的状态转移说明（如下图），可以轻易地写出第二段的代码。

这里我采取了一个小技巧：在EX状态下有四种转移，而我将最难判断的到达WB的转移放在最后一个互斥的else块中，剩余三个转移都属于较好写出条件的。这样减少了出错可能，同时简化了代码。



![U](03X$LH}J}Q00)REI)9CS](data:image/png;base64,)



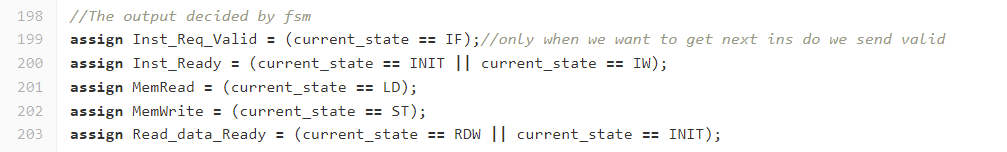


此外，我还注意到了一个细节：在INIT状态下，Inst\_Ready和Read\_data\_Read信号都要拉高，PPT中所说是“避免对复位释放后产生影响”（如下图），根据我的理解，这应当是由于rst==1时真实内存会释放一些无用数据，若CPU不接受，则无用数据将会一直占据内存端口，等到rst==0时CPU将会读出无用数据而非所需数据（指令）。



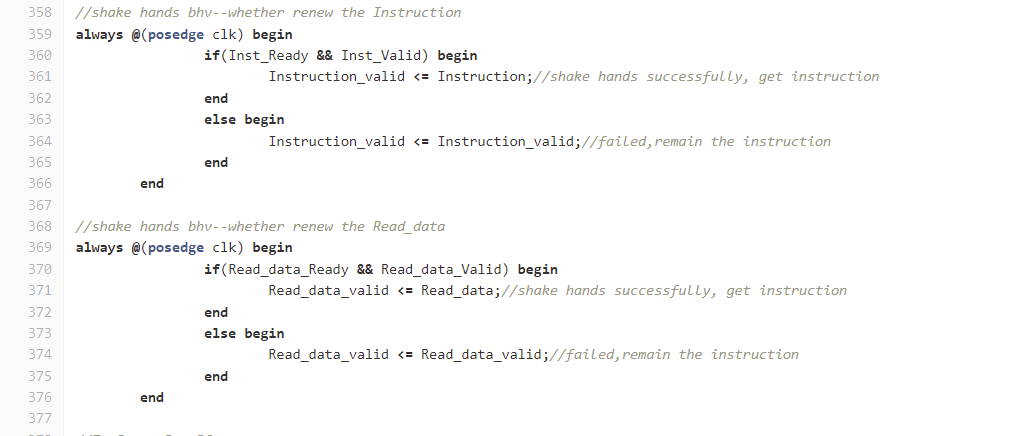
最后是FSM第三段的代码。第三段描述主要工作在于确定output信号，这里我选择采用组合逻辑，仍然是根据ppt上的output信号说明，在相应状态下拉高相应的output信号。

这里有一个比较需要考虑的点：在其它状态下，这些output信号是否要拉高？事实上，当且仅当我们真正需要实现该信号对应的功能时，才需要将其拉高——因为真实内存的具体实现方式我们是不了解的，这样考虑有助于减少错误的可能。



考虑握手机制，我们向内存发出请求后必须等到取指或者取数握手成功才能获取所需信号——这是与理想内存最大的不同点，而且注意到握手机制下内存的Instruction以及Read\_data端口输出的信号不再一直是我们所需信号，因此需要Instruction\_valid和Read\_data\_valid寄存器存放正在使用的指令或数据（这里解释了前文我声明这两个reg变量的用意），它们在时钟上升沿、握手成功的时候都会从真实内存的Instruction以及Read\_data端口获取信号传入CPU进行处理。其余情况，这两个有效指令与有效数据寄存器均不变，不论真实内存的指令和数据端口如何变化，于是CPU得以稳定处理所取指令以及数据。

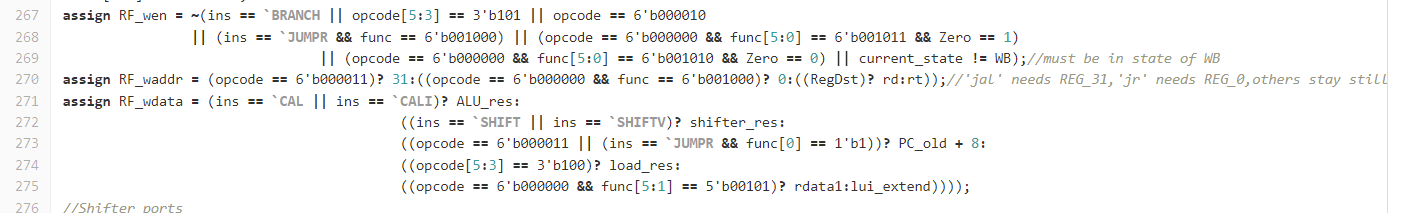
于是，所有单周期CPU中使用的Instruction以及Read\_data信号均需要修改为Instruction\_valid和Read\_data\_valid信号。



关于PC的变化，我们的处理如下图。当rst复位信号到来时复位PC，当取出的是NOP指令并且当前状态为ID时让PC=PC+4，因为我们即将进入下一个指令周期，而NOP指令的内容恰是什么也不做。当状态机处于EX执行阶段的时候，要根据不同指令对PC的改变来修改PC的值，这里如果是Jump控制信号拉高，说明接下来即将跳转执行，而PC也应相应地赋值成我们所计算出来的跳转地址Jaddr。如果不是跳转执行，则判断下一条指令是否是分支执行，如果是，那么PCSrc拉高，PC赋值为所计算出的分支地址PC\_BRANCH。如果都不是，那么应当正常地取出下一条顺序指令地址，即PC=PC+4。由于多周期时序，我们还必须增加一个PC不变的情况，用以在其它情况下保持PC的值。（尽管这不是必要的，还是可以让代码看得更清晰）



此外，我们还需要对其它控制信号进行一些适用性修改。

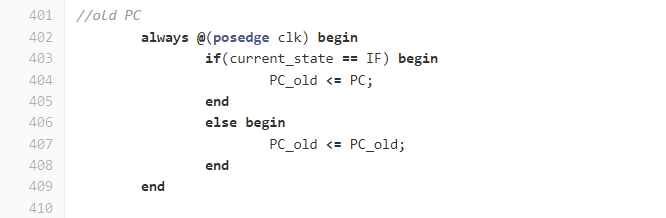


如上图，由于我们在执行阶段修改了PC的值，为了执行jal和jalr指令，我们又需要将原先的PC+8存储在相应寄存器中（reg 31 or rd），因此引入PC\_old进行旧PC的储存，并修改RF\_wdata中的PC+8为PC\_old+8，以便能够写入正确的地址。

另外，RF\_wen也需要进行修改。由于我们的状态机只可能在WB状态下产生写回操作，还需要增添一个current\_state是否为WB的判断。

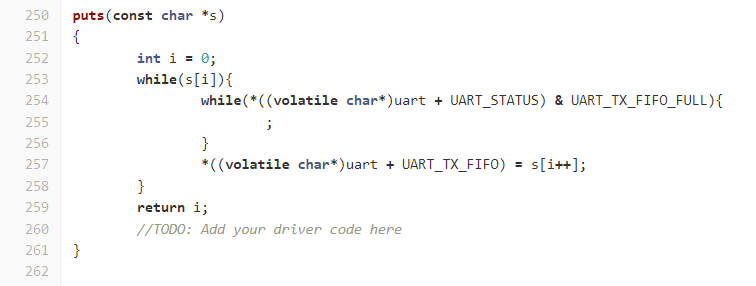
这里有一个非常细节的地方！jr指令默认写入0号寄存器（本实验是这么要求的，但是指令手册并没有说明）因此RF\_waddr需要加上jr指令的判断以确定写回寄存器是否是0号寄存器。一开始我并没有意料到jr指令需要写回0号寄存器，后来debug的时候无论如何都无法发现错误所在。好在后来有同学（匿名，不知道是谁）提醒了我。

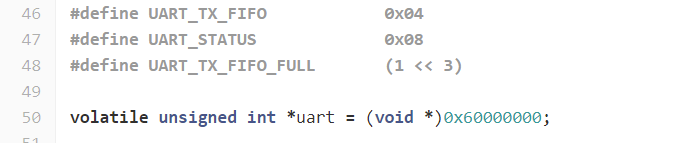
最后是考虑PC\_old的赋值逻辑。



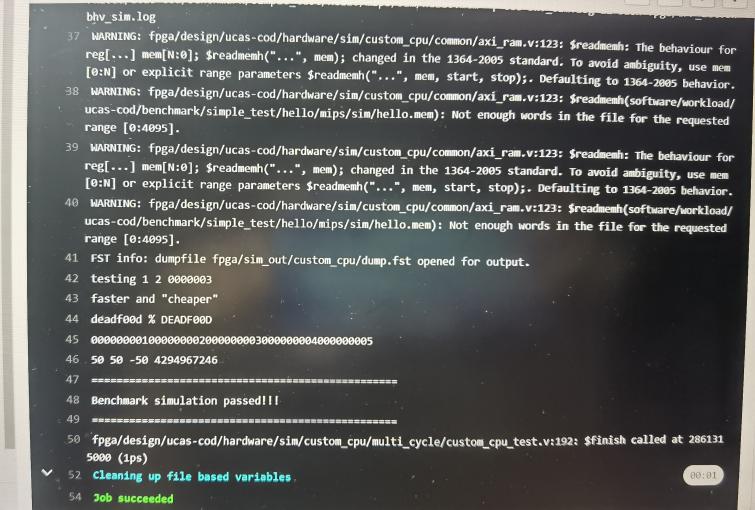
如上图，我们选择在进入IF周期的时候就将PC旧值储存下来（事实上只要在EX之前存储就可以，因为PC\_old的设计仅仅针对jal和jalr，它们均是在EX阶段修改了PC值），其他情况PC\_old均保持不变。这样，每进入一个指令周期，我们都会在IF阶段存储该指令的PC值。

到此，我们已经成功完成了一个多周期的CPU，接下来是软件部分的编写。

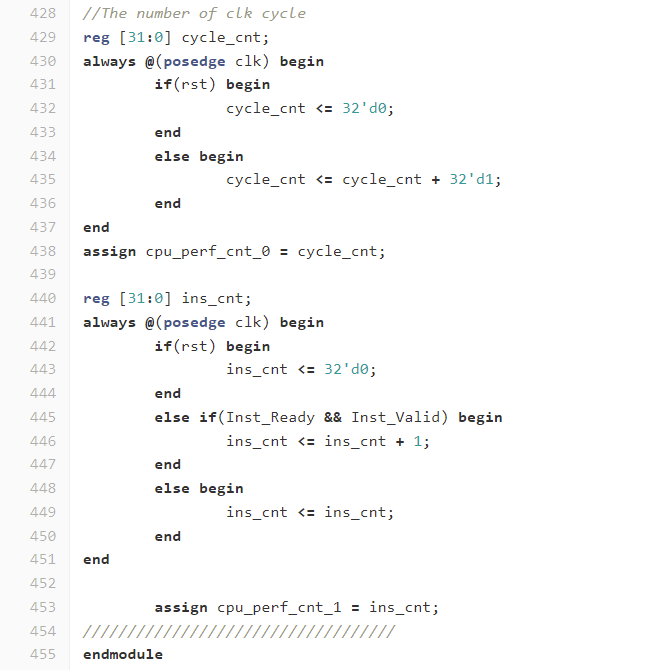


如上图，我们为实现打印字符串功能，先利用基址uart找到0x60000008地址的寄存器（队列状态寄存器），根据里面的内容判断队列是否为满。只有当队列不满的时候（内层while循环结束）才执行入队操作，将字符写入发送队列入口寄存器（地址为0x60000004）,并i++进入下一个外层while循环。当s[i]为NULL即字符串已经打印完毕的时候，我们退出循环，返回字符串长度i（此时i就是字符串长度）。

下面是打印成功的照片：

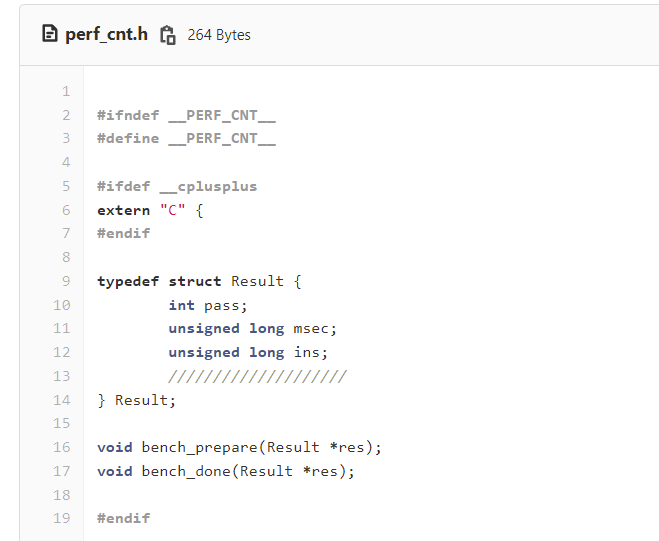


最后我们添加性能计数器。



如上图，我们写了两个性能计数器：一个统计时钟周期数，另一个统计指令数。为统计前者，我们在复位信号拉低的每个时钟上升沿给时钟周期数寄存器加1，并用assign语句将其连到0号性能计数器上；为统计后者，我们在每次取指令握手成功后都给指令数寄存器加1，并用assign语句将其连到1号性能计数器上。

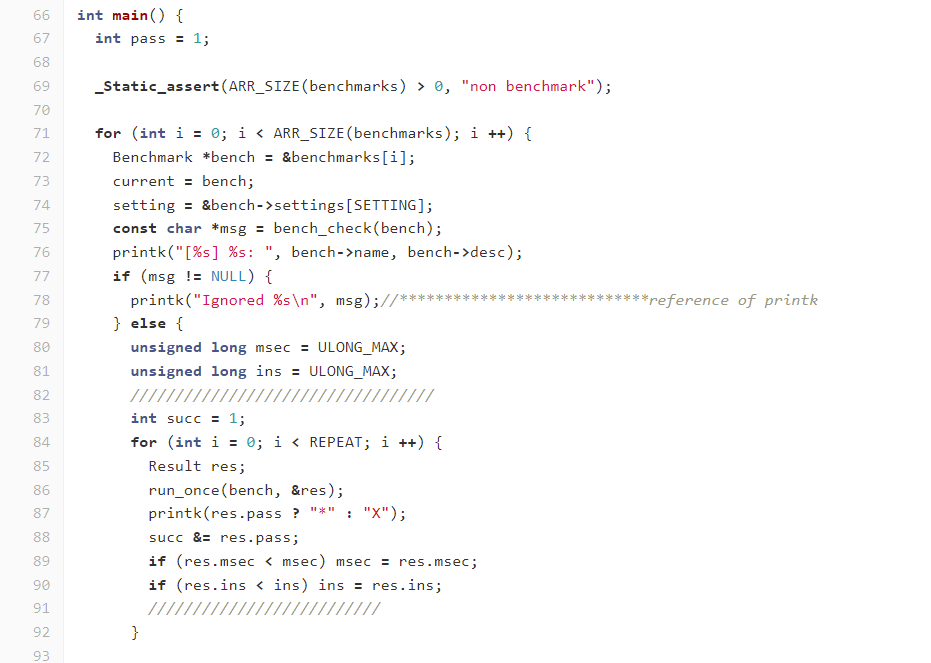
接下来是对软件的修改。

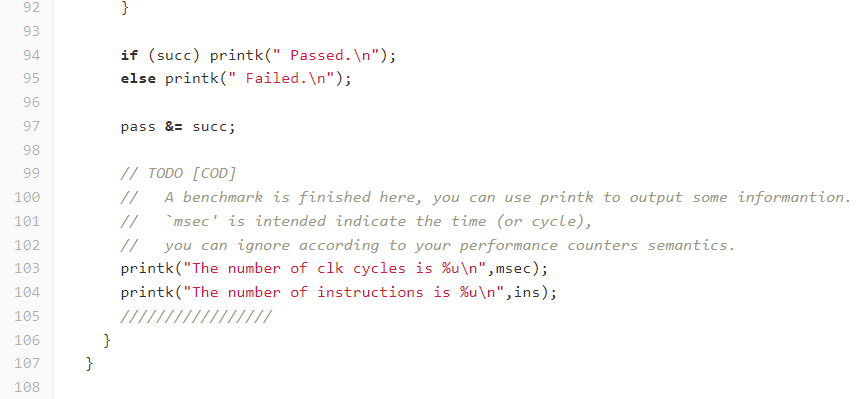


在perf\_cnt.h中，我们对Result结构体进行修改，增加ins变量（对应指令数计数器）。原先的mesc变量则对应时钟周期计数器。



在perf\_cnt.c中，我们利用up\_time和up\_insnum函数分别统计时钟周期数和指令数（根据计数器地址访问其中内容）。并且利用bench\_prepare函数来记录初始数值，以便在bench\_done函数中统计改变数目。





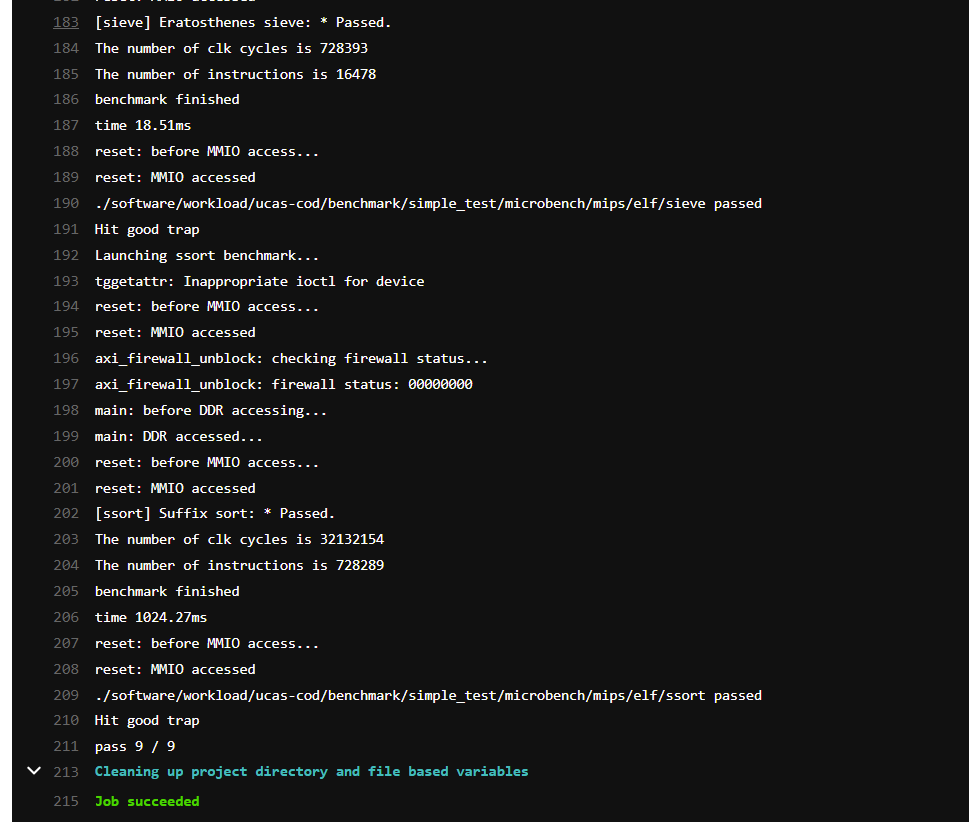
在bench.c中，我们首先在打////////的地方添加了对称的操作，即

IMG_256

和

IMG_257

在最后，我们利用printk函数打印性能计数器的值。printk的格式我是参考上面的printk的代码的。



如上图，这里展现了最终的性能计数器输出，可以看到，它们成功统计了时钟周期数以及指令数。发现指令数远小于时钟周期数，这与我们的预期是相符合的（因为真实内存访存速度慢，而且这是一个非流水线实现的多周期CPU）。

1. **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）**

硬件实现方面，遇到的问题主要在于控制信号的多周期适用性修改，尤其是三大RF信号（因为本代码是以单周期CPU代码为主体的，由于多周期和状态机的特性，我们必须对其进行适用性修改）！

由于我们在执行阶段修改了PC的值，为了执行jal和jalr指令，我们又需要将原先的PC+8存储在相应寄存器中（reg 31 or rd），因此引入PC\_old进行旧PC的储存，并修改RF\_wdata中的PC+8为PC\_old+8，以便能够写入正确的地址。

另外，由于我们的状态机只可能在WB状态下产生写回操作，还需要对RF\_wen增添一个current\_state是否为WB的判断。

这里有一个非常细节的地方：jr指令默认写入0号寄存器（本实验是这么要求的，但是指令手册并没有说明！）因此RF\_waddr需要加上jr指令的判断以确定写回寄存器是否是0号寄存器。一开始我并没有意料到jr指令需要写回0号寄存器，后来debug的时候无论如何都无法发现错误所在（对应波形图和错误信号五花八门、千奇百怪）。好在后来有同学（匿名，不知道是谁）提醒了我。

**三、对讲义中思考题（如有）的理解和回答**



volatile关键词的本意是“多变的”，作用在于避免编译器对其进行优化（由于访存操作的时间成本较大，编译器为避免重复访存，会将对同一地址的访存操作次数减少到1，尽管该地址所存的数值是变化的，这样就无法实现连续访存），以便我们能够时刻跟踪内存地址所存数值的变化情况。

如果去掉，那么访存读数将固定，无法读到最新的数值，输出也将出错。

**四、在课后，你花费了大约\_\_\_12\_\_\_小时完成此次实验。**

硬件耗时10h，软件耗时2h。（忽视serve云平台运行时间）

总共耗时24h。（算上serve云平台运行时间）

**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）**

希望下次能早一点将比较全的课件放到课程网站上，此次实验软件部分我对照v6版本课件竟无从下手，后来才发现v8课件加入了软件部分的具体操作指导，而v6里是缺失的。

整个实验是不算非常复杂的，因为硬件部分的代码都是基于单周期处理器的，而软件部分只要弄清楚原理则并不复杂。关键在于一些细节！尤其是控制信号的修改以及PC的跳转等，这些细节处的bug很难发现，即便利用波形图也难以看出来。我就因为jr不知道要写到0号寄存器而卡住很久（事实上指令手册和课件都没有说jr要写到0号寄存器，这属于缺失的信息）。

感谢19级的高学长，在多周期的PC\_old修改上给了我帮助。