中国科学院大学计算机组成原理实验课

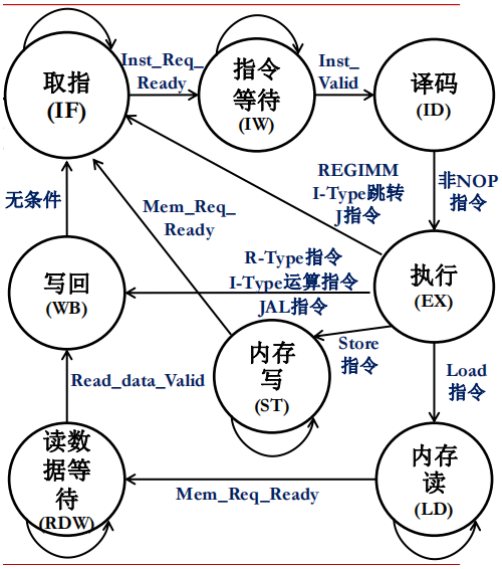
实 验 报 告

学号：2019K80009915022 姓名：栗祺菁 专业：计算机科学与技术

实验序号：prj3 实验名称：内存及外设通路设计及处理器性能评估

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.CPU：



主体状态图如上所示，省略复位状态。

思路：按照三段式状态机的步骤一步步思考。

第一段状态机：

用always时序逻辑，描述状态寄存器的同步状态跳转。

复位的时候当前状态置为RST状态，其它时间，每一拍都进入下一状态，注意这里的下一状态可能还是原状态，next_state的赋值在第二段状态机中实现。实现较简单，此处不放代码。

第二段状态机：

用always组合逻辑，根据状态机当前状态和输入信号，描述下一状态的计算逻辑。

对于取值状态、指令等待状态、内存写状态、内存读状态、读数据等待状态。只有input的Ready或Valid信号有效时，next_state才是真正的下一个状态，否则就停在原状态，等待握手成功。而译码、执行和写回都是一拍内完成的，下一状态不会是原状态。译码时若是NOP指令，就回到IF状态，否则就进入EX状态。执行时根据四类不同的指令，分别进到取值、写回、内存写、内存读状态。写回状态的下一状态一直是取值状态，相当于完成了一条指令，进入下一个指令周期。下面是第二段状态机的组合逻辑。

```
    //省略wire语句
    assign EX_IF=(opcode[5:0]==6'b000001 || opcode[5:2]==4'b0001 ||
opcode[5:0]==6'b000010);//从EX状态转到IF状态的情况，REGIMM，I-type跳转和J指令

    assign EX_WB=(opcode[5:0]==6'b000000 || opcode[5:3]==3'b001
||opcode[5:0]==6'b000011);//从EX转态转到WB状态的情况，R-type，I-type运算和Jal指令

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(Instruction_r==32'b0)begin
                    next_state=IF;
                end
                else begin
                    next_state=EX;
                end
            EX: if(EX_IF)begin
                    next_state=IF;
                end
                else if(EX_WB)begin
                    next_state=WB;
                end
                else if(opcode[5] && opcode[3])begin
                    next_state=ST;
                end
                else if(opcode[5] && ~opcode[3])begin
                    next_state=LD;
                end
            ST: if(Mem_Req_Ready)begin
                    next_state=IF;
                end
                else begin
                    next_state=ST;
                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;
            default:next_state=current_state;
        endcase
    end
```

第三段状态机：

用assign组合逻辑，根据状态机当前状态，描述不同输出寄存器的同步变化。

根据当前的状态决定输出信号的值。注意Inst_Ready除了在指令等待阶段拉高之外，在复位状态也应该拉高；Read_data_Ready除了在读数据等待阶段拉高之外，在复位状态也应该拉高。这样是为了避免访存错误（现有框架无法同时复位MIPS处理器和内存控制器）。另外的三个输出信号，在相应的状态拉高即可。下面是第三段状态机的组合逻辑。

```
    //省略wire语句
    //Inst_Req_Valid
    assign Inst_Req_Valid=(current_state[1])?1:0;//IF

    //Inst_Ready
    assign Inst_Ready=(current_state[0] || current_state[2])?1:0;//RST OR IW

    //MemWrite
    assign MemWrite=(current_state[5] && opcode[5] && opcode[3])?1:0;//ST

    //MemRead
    assign MemRead=(current_state[6] && opcode[5] && ~opcode[3])?1:0;//LD

    //Read_data_Ready
    assign Read_data_Ready=(current_state[0] || current_state[7])?1:0;//RST OR
RDW
```

Instruction寄存器和Read_data寄存器：

完成了三段式状态机的逻辑之后，还需要考虑因多周期带来的读入数据的多变性。比如对于指令Instruction，当指令等待阶段握手成功之后，输入Instruction就是这个指令周期的要用到的指令值，然后进入译码阶段。但是输入值Instruction是变化的，不能保证它在执行完这条指令之前不变化，所以应该将Inst_Ready和Inst_Valid握手成功时的指令值存下来，其它时候，这个存下来的值都保持不变。指令是由内存Memory输出的，那么同理Memory输出的数据Read_data也应该存下来。即在Read_data_Valid和Read_data_ready握手成功时，存下来读入的数据。下面是Instruction_r和Read_data_r的赋值

```
    //Instruction_r
    always @(posedge clk)begin
        if(Inst_Valid && Inst_Ready)begin
            Instruction_r<=Instruction;
        end
        else begin
            Instruction_r<=Instruction_r;
        end
    end

    //Read_data_r
    always @(posedge clk)begin
        if(Read_data_Valid && Read_data_Ready)begin
            Read_data_r<=Read_data;
        end
```

```
        else begin
            Read_data_r<=Read_data_r;
        end
    end
```

PC及PC_s:

接下来考虑PC的值。在现在的状态转移图中，为了取下一条指令，PC是在EX阶段更新的（除了NOP指令），但是在译码阶段及写回阶段需要用到此指令周期的PC值（即更新前的PC）进行计算和写Register操作。单周期中的PC值，在当前指令的所有操作完成前是不会更改的。为了同样实现这样的逻辑，在这里再加一个寄存器PC_s，存储当前指令周期有效的PC值。PC_s只在取值阶段更新，这样在后续的组合逻辑中都只用这个存下来的PC_s作为操作数，就可以实现和单周期一样的逻辑。下面是PC和PC_s的赋值逻辑。

```
    //省略wire或reg声明语句
    //PC
    always @(posedge clk)begin
        if(rst)begin
            PC<=32'b0;
        end
        else if(current_state[4]) begin//EX
            PC<=next_PC;//执行阶段更改PC值
        end
        else if(current_state[3]) begin//ID
            PC<=(Instruction_r==32'b0)?PC+4:PC;//NOP指令则在ID阶段就跳回取指阶段
        end
        else begin
            PC<=PC;
        end
    end

    //PC_s
    always @(posedge clk)begin
        if(current_state[1])begin//IF
             PC_s<=PC;
        end
        else begin
            PC_s<=PC_s;
        end
    end
```

最后将之前写的单周期的组合逻辑的部分，根据需要将Instruction改成Instruction_r,PC改成PC_s,Read_data改成Read_data_r。除此之外，输出的RF_wen信号应该加一个与操作，只有满足写的条件并且处于写回状态，它才可以拉高。

2.终端打印

处理器通过按地址读写端口寄存器，进行写数据操作，将要发送的8-bit字符写入TX FIFO寄存器的低8位。再通过FPGA云节点中的ARM处理器实现终端显示。

uart是UART控制器基地址指针，代表32bit的地址值。每个寄存器具有32-bit位宽，所以相邻寄存器的地址偏移值为4。UART发送数据队列入口寄存器偏移的地址数为4；UART队列状态寄存器偏移的地址数为8。uart+1指向的就是UART发送队列入口寄存器，uart+2指向的就是队列状态寄存器。要存储的是char类型的数据，将uart加1后，应再将uart强制类型转化为指向char类型的指针，相当于取该寄存器的低四个字节，然后将其值赋值为s[i]即可。

判断发送队列是否达到满状态：用发送队列状态寄存器的值与状态标志位掩码进行按位与操作即可。因为掩码只有第三比特为1，所以与的结果为1当且仅当状态寄存器的第三比特为1，即达到满状态。这个时候应该一直处于while循环内，进行等待。不满时，再进行赋值操作。

```
int puts(const char *s){
    unsigned int i=0;
    while(s[i]!='\0'){
        while((*(uart+2)) & UART_TX_FIFO_FULL);
        *(char*)(uart+1)=s[i];
        i++;
    }
    return i;
}
```

3.性能计数器

在CPU中增加4个性能计数器：处理器运行周期、完成执行的指令数、访存指令数、跳转指令数。

在bench.c中访问相应计数器的地址，进行计数，并输出值。

逻辑比较简单，不在此赘述。

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）

三段式状态机的逻辑本来应该不简单的，但是老师都已经给出了转态转移图，还把各种信号什么时候应该拉高等细节都已经告诉我们了，这个状态机就变得很简单了，只要根据内容填充就行。花费时间较多的是考虑存储从Mem读入的Instruction和Read_data的问题，开始没有想明白，实际上这种多周期的处理器的输入不是保持不变的，应该在数据或信号有效的阶段将有效值存下来，然后传递给后续的阶段。还有就是PC值的存储，这个虽然不是由外界输入，但是它为了适应输出PC的更改要求而在执行阶段就进行更改了，而译码时的PC+4的操作以及写回时写的PC+8，都应该是当前指令周期的PC值，即取值阶段的PC值，所以再用一个额外的寄存器存下来当前指令周期的PC值。

这些都想明白之后，调试过程也经历了很多波折。印象比较深刻的是，在后面组合逻辑的RF_wdata赋值处，我单周期中写的是MemRead拉高的时候，就让它赋值为modified_read_data(经过修改的mem输出数据)。但是现在的MemRead信号不再只取决于译码的opcode的值了，只有在LD阶段，它才会拉高。而RF_wdata的赋值实际上在取回指令后的译码阶段，它就完成了，那时MemRead信号还未拉高，所以就会导致load指令存不进去数的问题。最后，经过修改，使得若不满足别的赋值情况，RF_wdata就赋值为modified_read_data，这样就解决了这个问题。

还要一些别的bug，现在印象不是很深刻了，基本上就是组合逻辑中该改的信号没有改过来，通过查看波形，与金标准进行比对，找到出错的地方，然后查看逻辑，进行修改就可以了。

第二阶段的终端输出部分，因为对于偏移量的认知不够清晰，写错了几次。最终在同学的提醒下，写对了逻辑。

第三阶段较简单，主要是为了让我们了解性能计数器的原理以及访问方法。

三、 对讲义中思考题（如有）的理解和回答

在UART控制器基地址指针的定义处，用的是volatile unsigned int *uart。

这里的volatile修饰的是由uart指针指向的对象，即uart指针指向的寄存器的内容。

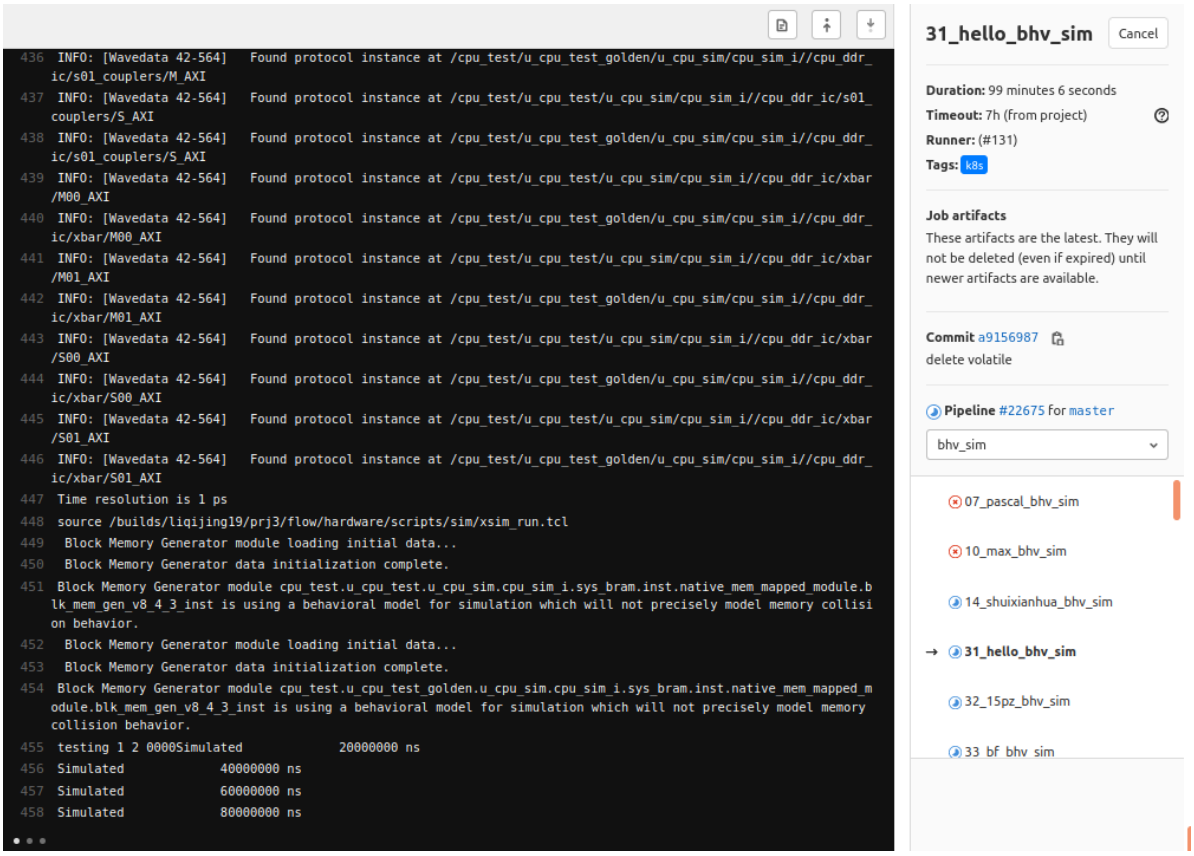
volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。当每次要求使用 volatile 声明的变量的值的时候，系统总是重新从该变量所在的内存读取数据，即编译器生成的汇编代码会重新从原地址读取数据放在相应的位

置，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。如果没有volatile，并且两次读取该变量值的代码之间没有对该变量进行任何操作，编译器就会对指令进行优化。优化做法是，在第二次使用该变量时它会直接使用上次读的数据，而不是重新从原地址读。这样以来，如果该变量是一个寄存器变量或者表示一个端口数据，它可能可以被一些编译器未知的因素进行修改。如果没有volatile，编译时编译器自动对其进行优化，第二次使用的数实际上可能并不是当时真正的变量值，这样就容易出错。而加了volatile之后，每次用到该变量时，它都会从原地址取数，所以说 volatile 可以保证对特殊地址的稳定访问。

除此之外，对于多线程的情况，有些变量也应该用volatile关键字进行声明。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，就应该用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。

删除volatile之后，原本10分钟跑完的hello测试点，现在跑两个小时都没有跑完，一直在simulation，猜测是因为队列一直处于满状态，然后就没有办法完成s的打印，一直在while循环中等待。而队列的满状态与否取决于队列状态寄存器中值的第三bit是否是1，即删除volatile之后*(uart+2)的第三位一直是1，这可能是因为它取的是上一次的值，但是在对编译器不可见的地方该寄存器的值进行了更改，进入了不满的状态，而它看不到更改，经过优化，又不再从原地址取数，就一直保留着满状态对应的值。

下面为删除volatile之后的第31个测试点hello的运行图



四、 在课后，你花费了大约_*12*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）

实验难度适中，但是我觉得我对于多周期的了解还是不够透彻。这个实验实现的多周期比理论课上简单，大多数组合逻辑还是在译码阶段就完成了，没有严格地按照阶段进行部件的使用。通过后面的流水部分实验、以及其它的选做实验我应该能对多周期有更加深刻的认识。

这次的实验感谢助教王嵩岳，以及刘泽昊同学对我的帮助，在我有疑惑和困难的时候认真耐心地帮助我。