计算机组成原理课程设计
实验报告

多周期/流水线处理器设计与实现

小组成员：

CMQ，WDT，HSF

WZW，LW，DWY

班级：07111503/07111506

二零一七年九月十八号

目录

多周期/流水线处理器设计与实现

一、 实验目的

1. 深入了解现代计算机硬件设计的基本流程和方法。
2. 了解典型的 RISC 处理器 MIPS 的体系结构与指令设计
3. 了解汇编语言到机器语言到计算机执行软硬的逻辑关系与设计
4. 掌握多周期处理器的设计方法与流程
5. 培养硬件级操作的调试与排错能力。

二、 实验内容

设计多周期或者流水线控制的处理器，支持 MIPS 指令子集:Lui，Addiu，Add， Lw，Sw，Beq，j，Ori，Addi，Bgtz， Jal，Jr， Sub， And， XOR，BGEZ，OR， Slt 共十八条指令以及自己设计的指令：Andi，Ori，Slti。

三、 实验器材与环境

操作系统：Windows 10 Enterprise

仿真软件：ModelSim 10.4

系统硬件：Intel® Core ™ i7-8550U CPU @ 1.80GHz 1.99GHz，RAM-16.0GB

四、 成员与分工

成员	分工	计分
陈牧乔（组长）	结合多周期数据通路特点，设计写使能的控制代码编写，输入输出代码编写，指令代码实现，撰写报告	10
王铎暾	多周期模块化设计，结合多周期处理器结构图画出状态	9.5

	转换图，构造 FSM 及 FSM 化简，指令代码实现，流水线初步实现，视频制作，撰写报告	
侯思凡	单周期代码框架到多周期代码框架转移，FSM 设计及代码实现，撰写报告。	9.5
王紫薇	写使能代码编写，FSM 合并设计，仿真测试，撰写报告	9
罗薇	仿真测试，状态转换合并及代码优化，撰写报告	9
杜文仪	仿真测试，数据通路及寄存器代码编写，撰写报告	9

五、 实验原理

在多周期的实践中，区别于单周期处理器设计，指令的执行完成不是耗费整个周期，而是被分成多个阶段，每个阶段在一个时钟周期内完成，使得整个时钟周期被降到最复杂的阶段所花实践(单周期时钟周期的 1/4, 1/3，取决于所分阶段的数量)。且利用使能信号对每个阶段的访存和读写进行一个限制，每一步的结果都能在下一个时钟周期保存到相应的单元（主要是寄存器相应位置中），对阶段的切分主要利用加入的寄存器来实现，但同时也需要考虑整个状态的转换。

在设计中，参考了课件中数据通路与写使能控制的设计（如图 1，图 2），在此基础上设计了共八个状态的状态转换图（八个转换状态包括了 R 型指令，I 型指令，J 型指令的执行过程，以及整个的数据存取过程）。在实践中，在单周期的基本设计上进行扩展，主要是寄存器的重新设计和指令读入和结果存取的重新设计。

数据通路：

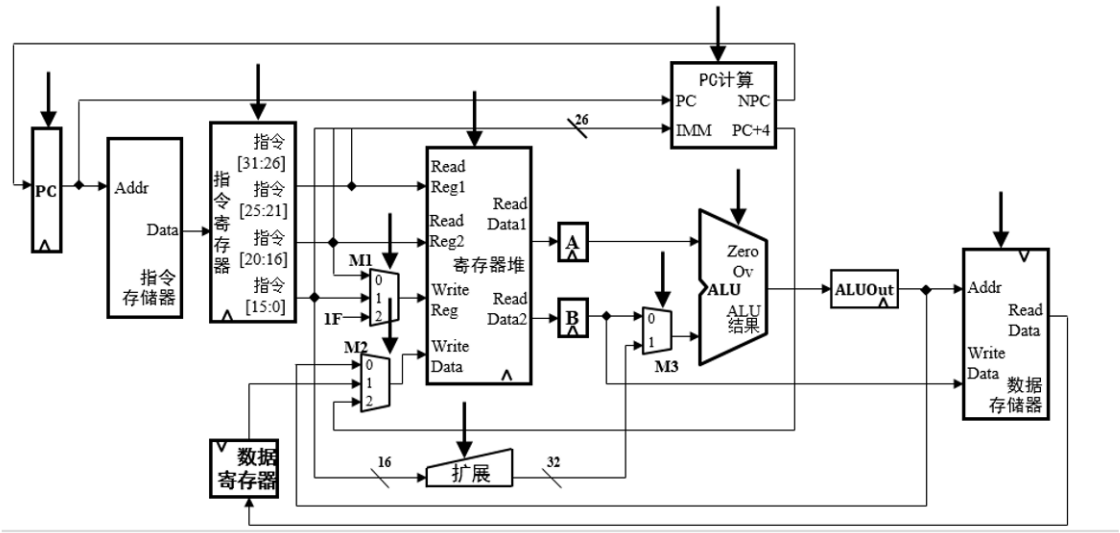

图 1 多周期数据通路设计

写使能：

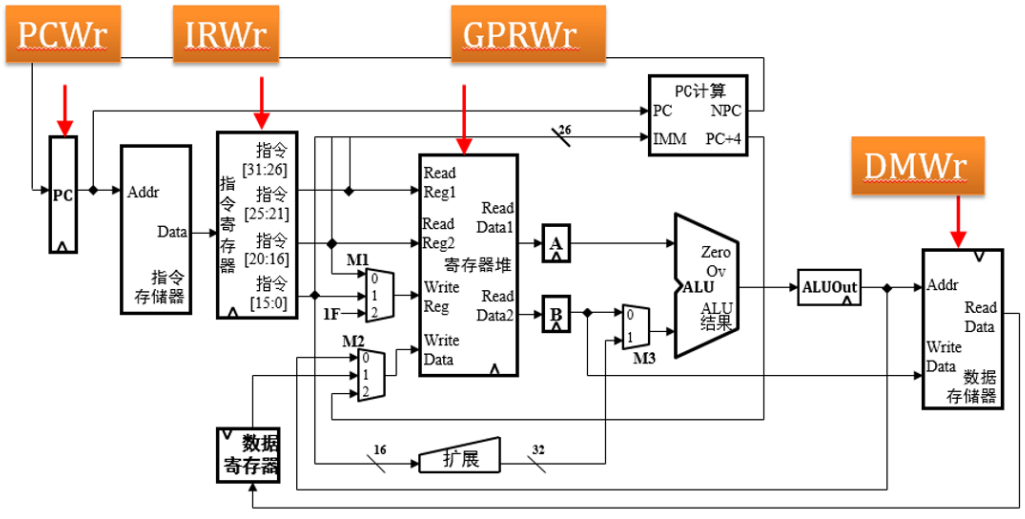

图 2 写使能参考设计

六、 实验步骤

6.1 状态转换图设计

多周期 CPU 设计的精髓在于将指令分为多个阶段进行执行，每一个阶段都在一个时钟周期内实现，所以在进行实现之前，我们需要确定多周期的执行状态，并根据指令类型的不同(R-TYPE,I-TYPE,J-TYPE)来确定每条指令的状态转换图。我们参考了课件中的多周期参考设计和图 1 图 2 的数据通路及写使能设计，共设计了 S0 到 S7 共八个状态,具体如下：

S0: 初始状态 (Initial)

S1: 比较状态-区分指令，方便后续操作 (Compare)

S2: 运算指令状态 (add/sub/lw 等)

S3: 读取内存状态

S4: 写进寄存器状态

S5: 写入内存状态

S6: 比较指令状态 (beq/bgtz/bltz 等)

S7: 跳转指令状态 (J/Jal/Jr)

整个状态的转换设计的概念，可以从图 3 看出，在下节我们将对三种指令的执行和对应控制信号的设计进行详细阐述。通过图 3 的状态转换设计，我们共实现了 21 条指令（18 条要求的指令加上 andi,oxri,slti 三条指令）

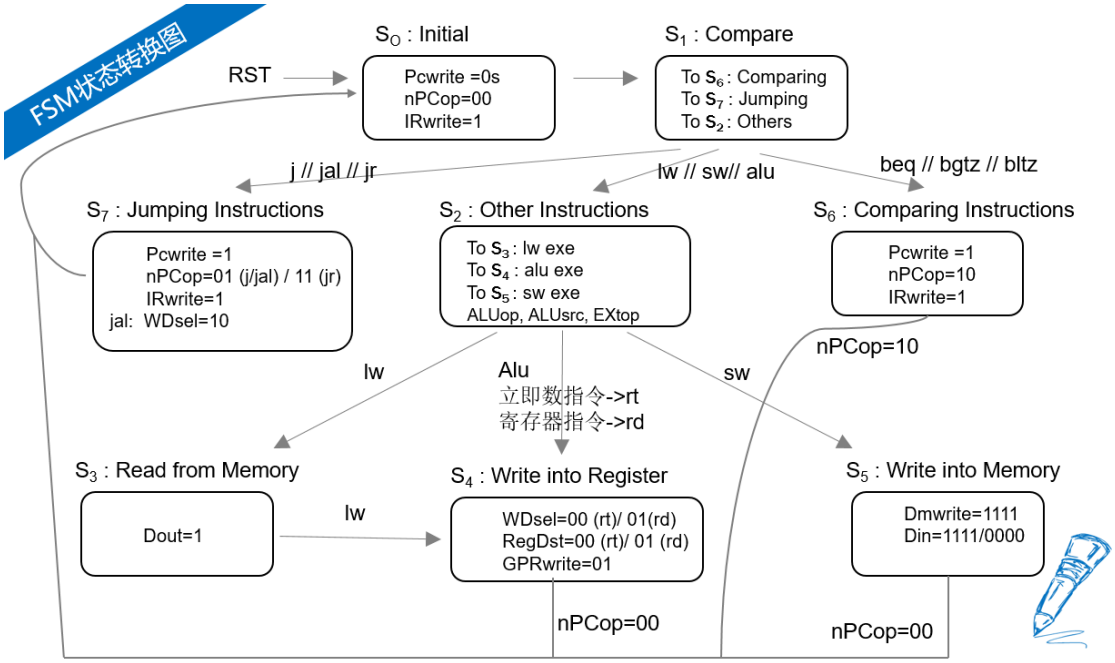

图 3 状态转换图

6.2 控制信号及指令的状态转换

根据图 3 的状态转换图，就可以根据每条指令需要执行的内容进行指令的 FSM 设计，因此需要一系列控制信号来控制状态的不同转换，以及指令对内存和寄存器的存器。根据多周期处理器的设计以及图 3 的状态转换图，我们一共设计了共 12 个控制信号，具体为：

Pcwrite：控制是否写 PC

nPCop：控制是否读取下一条指令

IRwrite：控制指令的读写

GPRwrite：控制寄存器的读写

ALUop：控制运算指令的执行

ALUsrc：控制运算操作数的类型（取寄存器/PC 值/内存）

EXtop：符号扩展控制

Dmwrite：内存的写使能控制

RegDst：选择目标寄存器（rd/rt）

WDsel：寄存器的写使能

Din: 写入流的控制 (进内存)

Dout:读入流的控制 (从内存)

对于不同类型的指令，通过不同的控制信号，进行不同的状态转换，完成相应的指令功能，图 4 给出了三种类型指令执行的状态转换，具体的转换与控制信号的值如下：

初始状态 (Initial):

PCwrite 的值归为 0 (表示不用写 PC)，nPCop 的值为 0 (表示以后正常的执行 pc=pc+4,顺序读取下一条指令)，IRwrite=1（表示读取指令），然后进入状态 S1。

(1) R-TYPE

对于所有的 R 型指令都要进入 S2-运算指令状态。但如果是 LW 指令则需要先读取内存，再将在内存中读取的数据写入寄存器，若如果是 sw 则进入 S5 状态将数据写入内存即可。其他的运算型指令包括(and,add,sub 等)则需要进行对应的运算操作，因为这里都是 R 型指令，利用的是 RD 寄存器，将运算的结果保存在该寄存器中然后进行寄存器的更新。

S2 状态: WDsel=01 (RD)，RegDst=01 (RD)，GPRwrite=01,(写入寄存器)

S3 状态(LW): Dout=1 (从内存中读取内容)

S5 状态(SW): Din=1 (将内容写入内存)

在执行完指令过后，nPCop=00 (按地址顺序进入下一个指令: pc=pc+4)，回到 S0 状态。

(2) I-TYPE

对于 I 型指令，若是普通的立即数操作指令(addi，andi，ori 等)则进入 S2 状态，然后利用 RT 寄存器进行运算结果的暂存，以及后续的寄存器的更新。若是比较型的指令(beq，bgtz，bltz 等)则进入 S6-比较指令状态进行另外的运算，主要是跳转不一样，因为有可能将不是按地址顺序进入下一个指令。

S2 状态: WDsel=00 (RT)，RegDst=01 (RD)，GPRwrite=01,(写入寄存器)

S6 状态: Pcwrite=1(写入 pc)，nPCop=10，IRwrite=1，nPCop=10 (可能执行指令的跳转)

在执行完指令过后，可能进行指令的跳转，然后都会回到 S0 状态。

(3) J-TYPE

对于 J 型指令，会进入 S7-跳转指令状态，然后进行 j，jal，jr 三种指令的不同处理，主要是 jal 指令还需要在同时把返回调用点的地址存储在$ra 中，所以需要一个 WDsel 信号，然后根据指令的不同设置不同的 nPCop 信号进行不同的跳转设置。

S7 状态：Pcwrite=1 (写入 pc)， nPCp=01 (j/jal)/11(jr),IRwrite=1,

若是 jal: WDsel=10

在执行完毕过后，同样的回到 S0 初始状态。

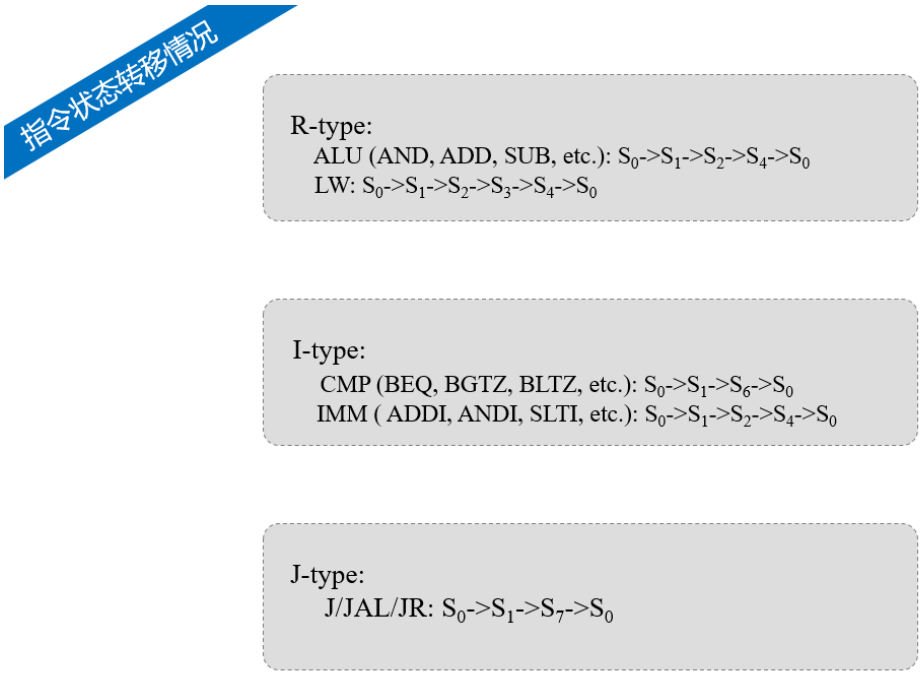

图 4 指令状态转移

6.3 代码实现及流水线

在设计完所有指令的执行流程以及控制信号后，需要进行对状态转换的代码实现的具体进行架构，我们利用了 case 结构对 FSM 的转换进行了具体实现：

```
always @ (posedge clk or posedge rst)    //clk following state changing!
   if(rst)
      fsm_state <= S0;
   else
      case(fsm_state)
          S0:fsm_state <= S1;   //read rs, rt
          S1:begin
             if(beq||bgtz||bgez)    //branch
                 fsm_state <= S6;
              else if(j||jal||jr)              //jump
                fsm_state <= S7;
             else                                  //exe
                fsm_state <= S2;
             end
          S2:begin
              if(lw)             //load
                 fsm_state <= S3;
              else if(sw)                //store
                 fsm_state <= S5;
              else                                   //exe
                 fsm_state <= S4;
             end
          S3:fsm_state <= S4;
          S4:fsm_state <= S0;
          S5:fsm_state <= S0;
          S6:fsm_state <= S0;
          S7:fsm_state <= S0;
          default: fsm_state <= S0;
     endcase
```

图 5 FSM 代码实现

能够从图 5 中清楚的看到一共八个状态的转换，以及在执行完相应指令后都会回到 S0 初始状态。

关于流水线的设计，我们小组尝试了初步的运算层面上的流水线设计，参考了顶层设计中将 IR 指令分割的方法，通过将 ALUout，即运算结果分别放在各级的寄存器中，实现了运算层级的流水线，使得可以运算和写入内存/寄存器的部分可以重合进行。具体到代码实现层次，在 S2-运算指令状态中，我们利用 Verilog 语言设置了中间的寄存器，将运算结果拆分为前后两部分 16 位，从而实现流水线的初步设计。

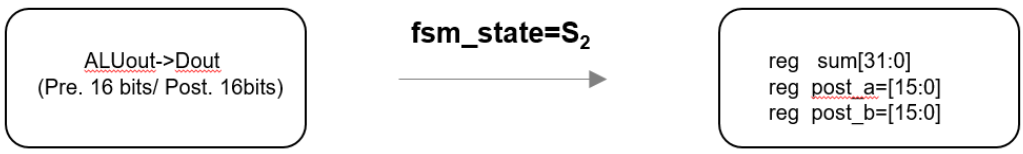

图 6 流水线设计

6.4 实验结果

(1) 斐波拉契数列的测试

我们将老师给的 testbench.v 文件放入 modelsim 里面进行结果测试，能

够清晰的看到仿真成功，所有的结果都已经顺利的写入到了 Data Memory 中：

```
VSIM 2> run -all
# dm[          0]=          1
# dm[          1]=          1
# dm[          2]=          2
# dm[          3]=          3
# dm[          4]=          5
# dm[          5]=          8
# dm[          6]=         13
# dm[          7]=         21
# dm[          8]=         34
# dm[          9]=         55
# dm[         10]=         89
# dm[         11]=        144
# dm[         12]=        233
# dm[         13]=        377
# dm[         14]=        610
# dm[         15]=        987
# dm[         16]=       1597
# dm[         17]=       2584
# dm[         18]=       4181
# dm[         19]=       6765
# Simulation is successfull!
# ** Note: $stop    : E:/modelsim_own_two/testbench.v(35)
#    Time: 32540 ns  Iteration: 2  Instance: /testbench
```

图 7 Results of Fibonacci

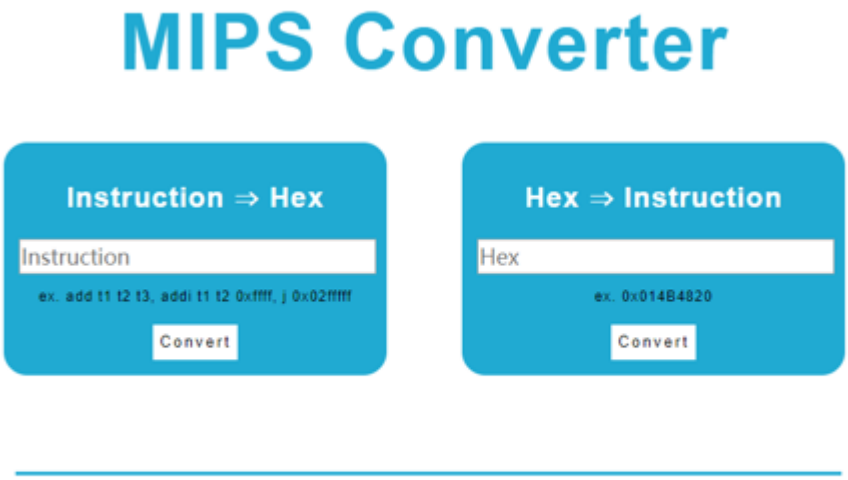

图 8 MIPS<->机器码

(2) 指令仿真与测试

我们利用了 MIPS Converter(如图 8)将需要测试的指令转换为可以进行测试的机器码，进行测试的机器码见文件 testcode.txt，**指令文件** test.asm，上

述机器指令基本包括了我们设计的所有指令,然后我们利用查看波形的方式验证了指令设计的正确性，能够看到如下波形：

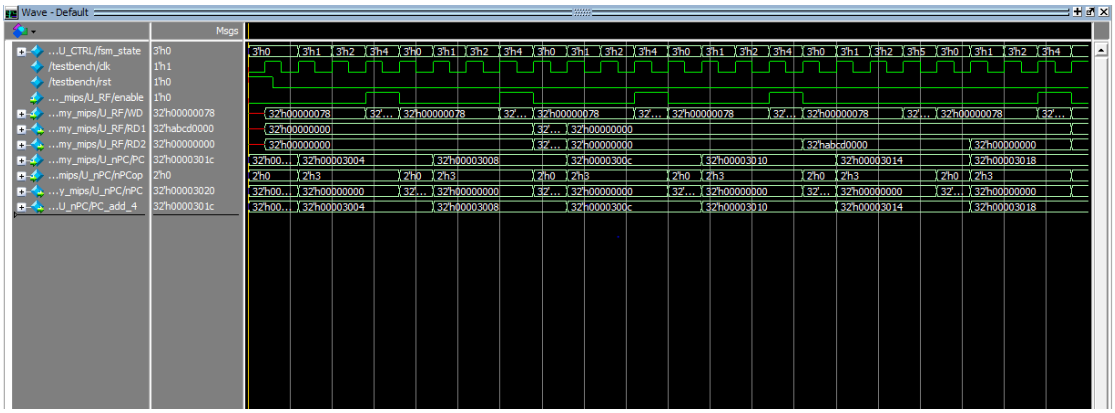

图 9 多周期测试波形

因为图片在报告中可能较小，可以点开多周期测试波形.PNG 文件进行清晰查看，我们能够清晰看到 npc 数值的向上递增，使能信号的有效，以及最重要的 fsm_state 的转换，比如图中 S0->S1->S2->S4->S0 表示 R 型指令的成功执行，S0->S2->S5->S0 表示 SW 指令的成功执行等。

我们同时也测试了自己选择实现的指令的正确性，以自己设计的 slti 指令为例的测试结果如下：

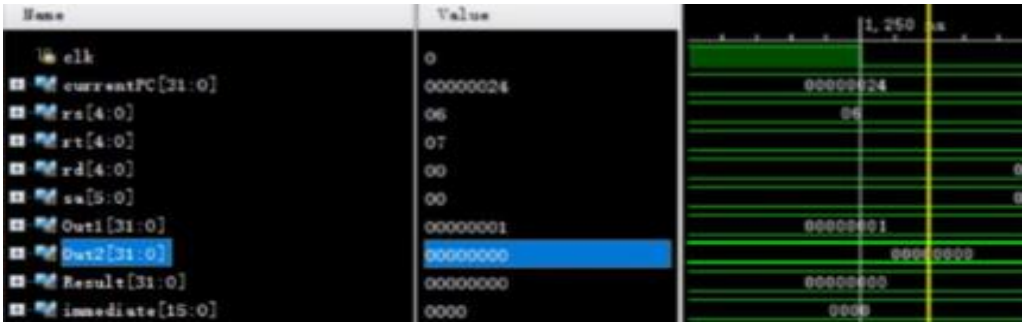

图 10 slti 指令测试波形

能够清晰的看到 RS 寄存器存储了 out1 的值位 1，RT 存储了 out2 的值（也是立即数 immediate 的值），由于 out1>out2 表示 slt a,b,0 指令中 a>b 所以 result 值为 0，充分展现了 slti 指令的正确执行。

七、 实验总结

在本次的CPU多周期设计与实践中，充分展现了需要对MIPS指令结构的掌握能力，以及需要的团队协作精神。在一开始如何将单周期的基础设计进行扩展到多周期是我们遇到的第一个瓶颈，通过询问老师和小组讨论，我们进行合理分工，将之前的单周期的结果先进行一个合并，然后再进行状态转换图的设计，从而顺利的进行了多周期的设计。

然后我们也尝试了流水线的设计，最初想要完成比较完整的流水线设计，但发现需要重新设计各种寄存器结构，所以我们主要完成了在ALU，计算阶段的流水线设计，使得写入寄存器操作和计算过程可以并行，然后经过小组讨论和交流，也对最后设计的新的指令和整个设计项目进行了测试和仿真波形的验证。最后在进行总结和汇报的时候，我们也将这个多周期斐波拉契的结果进行了后续的数码管设计，并录制了完整的视频，使得整个多周期的设计有了比较完整的结果和下板验证。

八、 实验心得

8.1 CMQ

虽然之前在计算机组成原理课程中学习过CPU的相关知识，但是拿到实验任务后还是感觉到陌生，不禁在心里打起了退堂鼓。在认真完成单周期CPU实验后再看多周期CPU实验，心中渐渐有了思路。多周期CPU的数据通路和单周期CPU的数据通路结构大体相同，不同之处主要在于控制模块的编写以及状态机的引入。状态机指示了不同指令划分的微操作执行的不同阶段，是多周期CPU的核心部分。

实验遇到的最大的问题还是对于CPU概念本身的理解，这是个最根本的方向性问题。在之前学习计算机组成原理时对这些概念都有过接触，但是当时对它们的理解大多还是停留在记忆层面，时间一长就记不起什么了。这次实验从

CPU 部件内部构造到不同部件构成的数据通路，从单周期、多周期到流水线，理论与实践结合，让我对无论是 CPU 的逻辑结构还是指令的动态执行过程都有了更加深入的理解，将原本散落在脑海中的一些生硬的概念有机地串联起来。我想这就是学习的过程，由浅入深，从理论到实际，是一次又一次的反复，却总有新的认知。

当实验的方向性问题得以解决，也就是知道了该做什么，接下来的问题就是具体怎么做了。这涉及到小组成员的分工与合作问题，处于认知之后的决策和行动层面。这次实验我被委以组长的重任，对于之前没有太多组长经验的我是一次不小的锻炼。实验过程遇到了许多实际操作方面的问题，比如实验进度安排不合理（由于还有汇编实验，想弄完汇编再集中弄计组，后来发现留给计组从零开始的时间不太充裕了）、成员分工问题（是横向分工还是纵向分工）以及在代码编写和调试过程中遇到的各种 bug。在小组成员的团结协作下，攻坚克难，最终成功完成了实验内容。

8.2 DWY

这次实验对我总的来说比较困难，一是因为原理上掌握的不熟悉，代码也很难看懂，二是软件也很生疏，会遇到各式各样的问题，实验环境比较差，但是在我们小组的共同努力下来说取得了一定的成功，对此我也感觉很有成就感，也收获了很多以前没有接触过的知识。

8.3 HSF

组原模块二是整个小学期最费时费力的实验了。单周期慢慢地学习了解，也能摸索出其中一二；实验三不过也是在实验二的基础上做些工作。多周期设计完全是一头雾水地猜想，实现的时候也会卡壳，甚至怀疑之前的工作是不是设计得不好，需要回溯修正。

好的地方在于，可以以单周期设计为基础，像 im、dm 模块就是利用了单周期的设计。集思广益能够迸发更多意想不到的办法，但是缺少项目经验的我们对于分工与集成又很难拿捏。经常在写一个模块的时候发现事先商量好的接口需要一些新的改动，写着写着觉得这几个模块重复了、冗余了、或者残缺了。

大三以来经历了很多集体合作的项目，这次实验对计算机组成原理又有了

新的认识，也对硬件设计的掌握更上了一层。困难越大收获越多，总体来讲，对最后我们实现的功能还是比较满意的。

8.4 LW

在多周期 CPU 的设计之前，我们已完成了单周期 CPU 设计。相比之下，单周期 CPU 在一个时钟周期内完成所有的工作，包括从指令取出，和得到结果。而多周期 CPU 的设计将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成。这样做的优点是不仅能提高 CPU 的工作频率，还为组成指令流水线提供了基础。

单周期所有的指令都要按顺序执行 IF->ID->EXE->MEM->WB，多周期可以不用，比如 beq 指令就只需要 IF->ID->EXE，jal 指令只需要 IF->ID，就可以省很多不必要的时间。

总的来说，本次实验难点在于控制单元、寄存器模块与存储器模块的设计，以及通过组合逻辑处理不同周期的指令。

8.5 WZW

模块二的完成建立在完成单周期设计抽选指令，理解老师提供的框架和多周期处理器的工作原理的基础上。小组成员一起讨论研究了多周期 CPU 的设计，单周期的简单之处在于每条指令的执行需要一个时钟周期，一条指令执行完再执行下一条指令。多周期要把指令的执行分成多个阶段，设计状态转化图，通过完成 FSM、DM 的设计来完成基本架构；还要通过写使能的控制，防止竞争的影响。

我们最终设计了八个状态，支持包括 andi, oxri, slti 在内的一共 21 条指令。我的工作一项主要工作就是 Modelsim 的调试。多周期的的复杂程度和代码量都不小，对于一条具体的指令，在不同的状态每个控制信号都有不同的值，每个模块的输入输出我们都要进行仿真来检查，这个过程枯燥但极其重要，期间遇到错误也能很直观的看到问题所在。在多周期的基础上我们也尝试进行了流水线处理，将 32 位数分成两级的 16 位流水处理，结果正确。

整个框架在与小组成员的不断交流中逐渐完善，在一次次仿真测试中减少错误。一开始面对多周期我感到无从下手，在和其他同学的讨论之后逐渐理

解。由于时间有限，我们对流水线的处理还比较基础，之后可以完善这一部分，并比较流水线处理对运行速度的影响。通过这门实验，我更加了解了处理器的设计原理和方法，也感受到了团队协作对实验的积极作用。

8.6 WDT

在本次多周期CPU设计实验过程中，我主要负责了FSM状态转换图的各种设计和实践，我充分的学习到了关于Verilog语言的相关知识，也对CPU从设计到实践有了充分认识，特别是在状态转换的设计中，如果转换的状态设计有问题将会直接导致后续实践的问题，所以我也充分利用了小组讨论和老师交流将状态转换的设计进行了优化。然后是在实践中，因为模块化的设计使得整个小组可以分别同时开工，且能够互相交流，所以整个多周期设计的团队交互将能够比较高效的完成。且在流水线的设计中，我也充分学习到了Verilog语言中寄存器数据类型的比较方便的用法，在今后我也会更加注意在代码前的整体设计工作，使得代码实现的时候更快速，更准确。

在全部整个实践过程中，我都有一定的参与，我觉得最困难的部分就是在代码实现过后的调试与仿真过程，因为所有的代码优化和结果输出都需要通过调试才能有结果的展现，才能发现问题。但我可能对于看波形这方面不是特别擅长，但在团队的帮助下，一个一个时钟进行细致的分析，才最终读懂了波形然后一步一步的分析输出，又通过输出的反馈来优化代码，从而形成了一个开发的优化的闭环。

九、 参考文献

[1] 《MIPS指令令集规范》

[2] CPU设计：https://blog.csdn.net/leishangwen/article/details/39277137

[3] 《57条MIPS指令说明》

[4] 流水线设计：https://blog.csdn.net/times_poem/article/details/52033535

[5] 《数字设计和计算机体系结构》