数字逻辑与处理器基础实验

流水线MIPS处理器实验报告

学号：2020010768

班级：无05

姓名：付宇辉

时间：2022年7月16日

**目录**

[一、实验名称与内容 1](#_Toc108879972)

[二、设计方案 1](#_Toc108879973)

[（一）总体设计 1](#_Toc108879974)

[（二）IF阶段 2](#_Toc108879975)

[（三）ID阶段 2](#_Toc108879976)

[（四）EX阶段 3](#_Toc108879977)

[（五）MEM阶段 3](#_Toc108879978)

[（六）WB阶段 3](#_Toc108879979)

[（七）设计细节 4](#_Toc108879980)

[I. 分支与跳转 4](#_Toc108879981)

[II. 避免冒险 4](#_Toc108879982)

[III. 外设处理 5](#_Toc108879983)

[三、文件清单与关键代码 5](#_Toc108879984)

[（一）文件清单 5](#_Toc108879985)

[（二）关键代码 7](#_Toc108879986)

[四、综合与实现情况 8](#_Toc108879987)

[（一）时序性能 8](#_Toc108879988)

[（二）逻辑资源占用情况 9](#_Toc108879989)

[五、仿真验证 9](#_Toc108879990)

[六、硬件调试情况 10](#_Toc108879991)

[七、心得体会 11](#_Toc108879992)

[附录 支持的指令格式 12](#_Toc108879993)

# 一、实验名称与内容

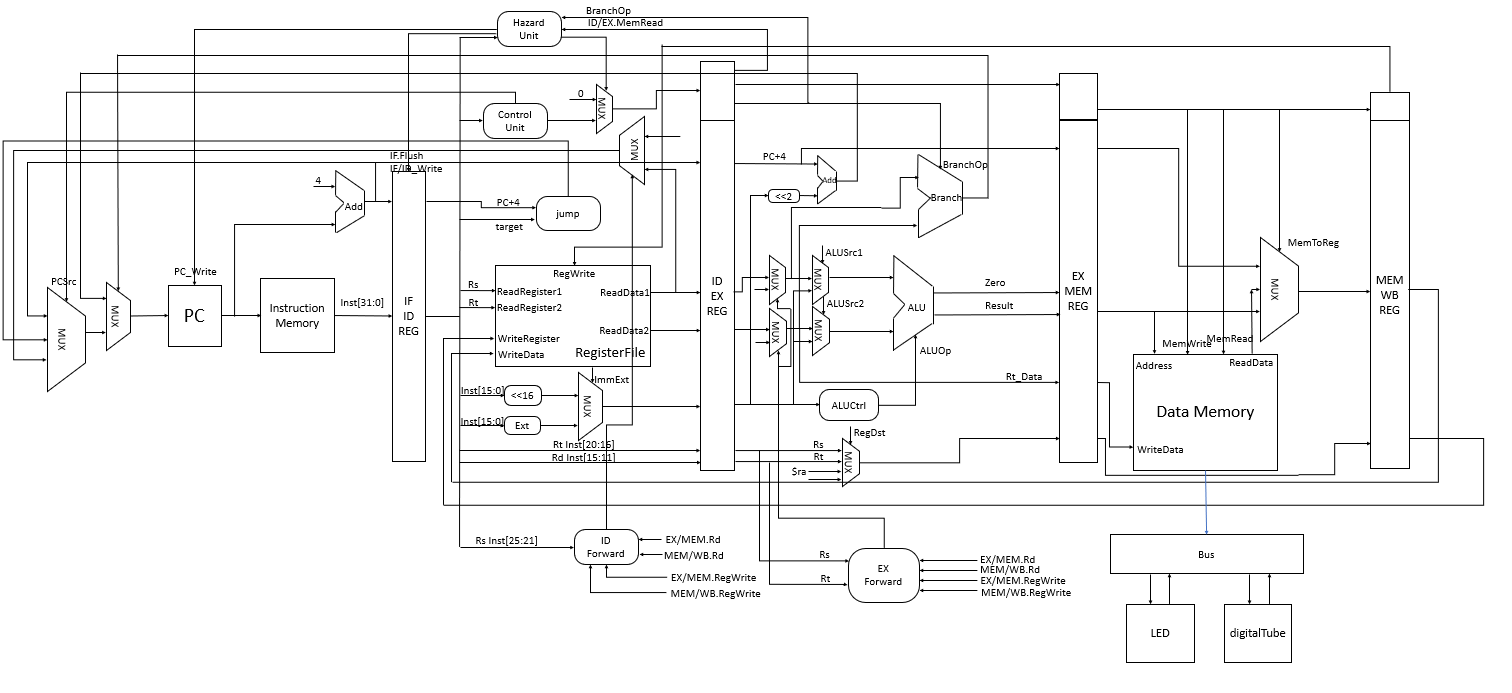
实验名称：流水线MIPS处理器的设计

实验内容：我选择的是实验内容1，即，将春季学期实验四设计的单周期（多周期）MIPS 处理器改进为流水线结构，并利用此处理器完成字符串搜索算法。

# 二、设计方案

## （一）总体设计

根据理论课的学习，使用Verilog，设计了支持MIPS32核心指令的五级流水线。总体设计的数据通路如下，主体的设计结构与下图一致。



设计的CPU支持的指令如下：

1. R型指令：add、addu、sub、subu、and、or、xor、nor、slt、sltu、sll、、srl、sra、jr、jalr；
2. I型指令：lui、addi、addiu、andi、sltiu、lw、sw、beq、bne、blez、bgtz、bltz；
3. J型指令：j、jal。

上述指令的具体格式参见附录。

CPU为五级流水线设计，分为取指令（IF）、译指令（ID）、执行（EX）、访存（MEM）、写回（WB）五个阶段。采用了完全的Forwarding解决数据依赖冒险，模拟寄存器的先写后读；利用stall一个周期的方式解决load-use冒险；分支指令提前到EX阶段进行判断，J型指令与jr和jalr在ID阶段进行跳转。

下面对各个方面介绍CPU的设计细节。

## （二）IF阶段

IF阶段进行的是PC的更新和指令存储器的访问。

当时钟上升沿来的时候，PC会根据控制信号进行更新。不进行跳转，更改为PC+4；分支跳转，更新为branch\_target；直接跳转更新为jump\_target。PC的写入信号收到PCWrite控制，通过控制写入信号可以进行stall周期，来起到解决竞争和冒险的问题。

## （三）ID阶段

ID阶段进行的是对IF阶段取出指令的译码并进行访问寄存器堆取值，同时进行立即数的扩展，通过Control模块进行控制信号的生成，通过Hazard模块进行冒险的判断，进行跳转目标的生成，ID阶段的转发。

ID阶段利用IF和ID之间的寄存器，进行操作。对寄存器堆的读写，是通过指令译码选择，并且进行数据转发存入下一个级间寄存器。立即数扩展有三种方式，分别是左移16位、符号扩展、零扩展。最重要的部分是将指令的OpCode和Funct部分输入Control模块，生成控制信号，并且让控制信号在各个流水线级间进行传递，保证流水线各个部分工作能够正常进行。

## （四）EX阶段

EX阶段进行的是ALU运算，同时生成分支的目标和branch\_hazard信号。

我在这里直接秉持着“不重复造轮子的原则”，直接将数字逻辑与处理器基础理论课大作业部分的ALU和ALUControl模块直接复用。ALU接收两个32位整型数据，支持位运算、加法、减法、小于比较功能。控制信号由ALUControl接收ID\_EX\_Reg存储的控制信号后产生。

Branch接收两个32位整型数据，同样根据控制信号进行等于、不等于、小于等于零、小于零、大于零的判断，输出信号branch\_hazard，当其为高电平时进行分支跳转。同时将立即数左移两位，与PC+4求和，得到分支跳转的目标。

ALU的输入时读取上一个级间寄存器与EX阶段的转发的结果选择得到的。

## （五）MEM阶段

MEM阶段进行对数据存储器的访存操作，同时与外设LED和数码管进行交互。数据存储功能；0x40000000～0x7FFFFFFF（字节地址）为外设地址空间，对其地址的读写对应到相应的外设资源。

MEM阶段根据EX\_MEM\_Reg中存储的MemToReg信号，从PC、PC+4，Rt内容、访存结果中选择写入寄存器堆的数据。

## （六）WB阶段

WB阶段是对寄存器堆的写入操作。

通过写入信号的控制，我们将上个级间寄存器需要写入的数据，写入寄存器堆，完成一个指令整个执行的过程。

写入的数据也会进行数据的转发操作，为了避免数据冒险。

## （七）设计细节

### I. 分支与跳转

CPU 支持 j、jal、jr、jalr 以及多条分支指令，分别在 ID 阶段和 EX 阶段进行判断和执行。对于 j、jal、jr、jalr 四条指令，在 ID 阶段控制信号产生模块生成 jump\_hazard 信号，传入冒险处理模块。冒险处理模块产生 IF\_Flush 信号，将 IF\_ID\_Reg 的指令置为 32 位0，从而清理掉下一个周期 ID 阶段将译码的信号。

对于分支指令，处理器默认不跳转，即 IF 和 ID 不暂停工作。在分支指令的 ID 阶段，控制信号产生模块根据分支指令的类型，产生 BranchOp 并存储到流水线寄存器中。在 EX阶段，Branch 模块接收转发后的 rs\_data 和 rt\_data 作为输入，并进行相关判断。当判断为真时，branch\_hazard 信号拉高，冒险模块随即产生 IF\_Flush 和 ID\_Flush 指令，将对应的 IF\_ID\_Reg 和 ID\_EX\_Reg 中的关键信号改为 0，即清理掉错误的指令。而当判断为假时，处理器继续运行，可以减少一定的 CPI。简言之，此 CPU 对分支的预测总为假。

### II. 避免冒险

上面介绍了分支跳转如何解决，下面着重介绍如何避免数据冒险。

整体来说，避免数据冒险是通过转发解决的。根据理论课的学习，设计了四条转发路径，来解决数据冒险。

EX\_MEM转发到ID：对于前前条指令执行写入寄存器的操作，不可能从寄存器中读取数据，而jr和jalr需要读取Rs寄存器的数据。所以将EX的结果从EX\_MEM\_Reg转发到ID，从而不用添加nop或者stall便可以在ID阶段跳转。

MEM\_WB转发到ID：实际写寄存器在WB阶段后的一个周期开始，我们将MEM\_WB\_Reg中的write\_data转发到ID，模拟了先写后读，保证WB阶段的同时，ID可以读取到正确的数据。

EX\_MEM转发到EX：对于前一条指令或者需要写入寄存器的指令，可以采取从EX\_MEM\_Reg转发到EX阶段的策略，将同一时刻正在WB写入数据用于EX的计算中。

MEM\_WB转发到EX：对于前前条、执行访存或者需要写入寄存器的指令，可以从MEM\_WB\_Reg转发到EX阶段，将WB阶段写入的数据用于EX的计算中。

对于Load-Use冒险，采取阻塞一个周期的方式，然后可以通过上述转发操作，解决。

### III. 外设处理

直接通过写入特定地址的数据存储器当中的数据来控制外设的工作状态。

我通过直接编写汇编代码，将外设的控制信号写入特定的存取器地址，来控制外设。

但是由于编写汇编代码的复杂性，目前只能将一位十六进制数显示在数码管上，这是需要改进的地方，希望后续能够找到更好的解决方案。

# 三、文件清单与关键代码

## （一）文件清单

本次报告的目录结构如下：

├─assets 报告中使用的图片

│ critical\_path.jpg

│ critical\_path\_graph.jpg

│ instruction\_count.jpg

│ running\_periods.jpg

│ schematic.jpg

│ timing\_summary.jpg

│ utilization\_hierarchy.jpg

│ utilization\_summary.jpg

├─report 实验报告

│ report.docx

│ report.pdf

├─src 代码文件

│ ├─constraints

│ │ CPU.xdc 约束文件

│ ├─simulation

│ │ tb.v CPU testbench文件

│ └─source

│ ALU.v ALU计算模块

│ ALUControl.v ALU控制信号生成单元

│ Branch.v 分支判断

│ Control.v 控制信号生成单元

│ CPU.v 顶层文件

│ DataMem.v 数据存储器

│ EX\_Forward.v EX阶段转发单元

│ EX\_MEM\_Reg.v EX\_MEM级间寄存器

│ Hazard.v 冒险控制单元

│ ID\_EX\_Reg.v ID\_EX级间寄存器

│ ID\_Forward.v ID阶段转发单元

│ IF\_ID\_Reg.v IF\_ID级间寄存器

│ InstMem.v 指令存储器

│ MEM\_WB\_Reg.v MEM\_WB级间寄存器

│ PC.v PC寄存器

│ RegisterFile.v 寄存器堆

└─src

│ test.dat 测试用字符串和模式串

│ generate\_InstMem.py 通过MARS十六进制指令文件生成InstMem文件

├─asm 编写的汇编文件

│ bf.asm

│ horspool.asm

│ kmp.asm

├─inst\_hex MARS生成的十六进制指令

│ bf.hex

│ horspool.hex

│ kmp.hex

└─inst\_mem 生成的指令存储器文件

bf.txt

horspool.txt

kmp.txt

## （二）关键代码

ID阶段转发代码

assign rs\_data\_foward\_id = (id\_forward\_1 == 2'b00)? rs\_data:

                           (id\_forward\_1 == 2'b01)? ex\_mem.ALUOut:

                           mem\_wb.Write\_data;

assign rt\_data\_foward\_id = id\_forward\_2? mem\_wb.Write\_data: rt\_data;

EX阶段转发代码

wire [31:0] rs\_data\_foward\_ex, rt\_data\_foward\_ex;

assign rs\_data\_foward\_ex = (ex\_forward\_1 == 2'b01)? ex\_mem.ALUOut:

                           (ex\_forward\_1 == 2'b10)? mem\_wb.Write\_data:

                           id\_ex.Rs\_Data;

assign rt\_data\_foward\_ex = (ex\_forward\_2 == 2'b01)? ex\_mem.ALUOut:

                           (ex\_forward\_2 == 2'b10)? mem\_wb.Write\_data:

                           id\_ex.Rt\_Data;

wire [31:0] alu\_src1, alu\_src2;

assign alu\_src1 = (id\_ex.ALUSrc[1:0] == 2'b01)? id\_ex.Imm:

                  (id\_ex.ALUSrc[1:0] == 2'b10)? 32'h0:

                  rs\_data\_foward\_ex;

assign alu\_src2 = id\_ex.ALUSrc[2]? id\_ex.Imm: rt\_data\_foward\_ex;

Hazard控制代码

wire load\_use\_hazard;

assign load\_use\_hazard = reset? 1'b0: (ID\_EX\_MemRead && (ID\_EX\_Rt != 0) && ((ID\_EX\_Rt == IF\_ID\_Rs) || (ID\_EX\_Rt == IF\_ID\_Rt)));

assign PC\_Wen = ~load\_use\_hazard;

assign IF\_Wen = ~load\_use\_hazard;

assign IF\_Flush = reset? 1'b0: (jump\_hazard || branch\_hazard);

assign ID\_Flush = reset? 1'b0: (branch\_hazard || load\_use\_hazard);

PC更新代码

assign PC\_plus4 = {PC\_o[31], PC\_o[30:0] + 31'd4};

assign PC\_next = branch\_hazard? branch\_target:

                 (PCSrc == 2'b00)? PC\_plus4:

                 (PCSrc == 2'b01)? jump\_target:

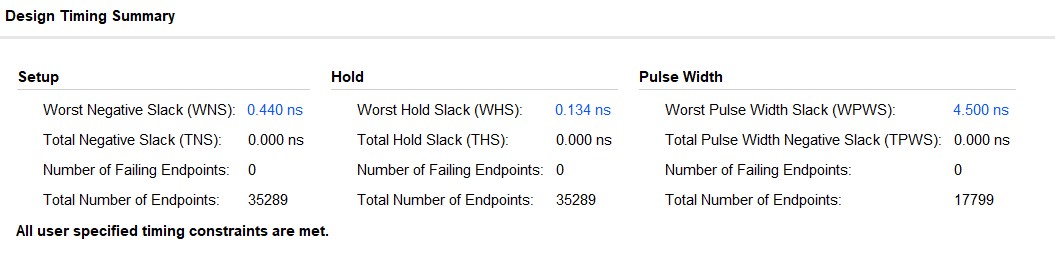
                 (PCSrc == 2'b10)? jr\_target:

                 32'h00400000;

# 四、综合与实现情况

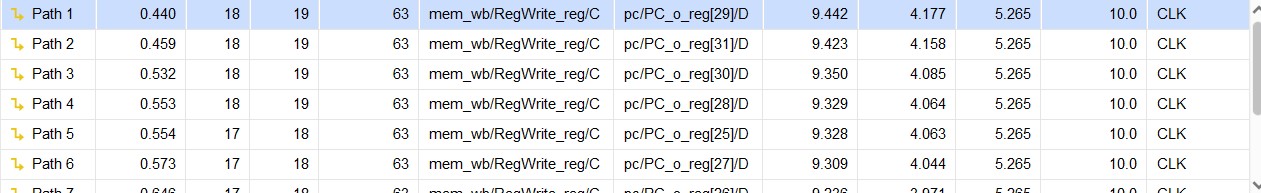
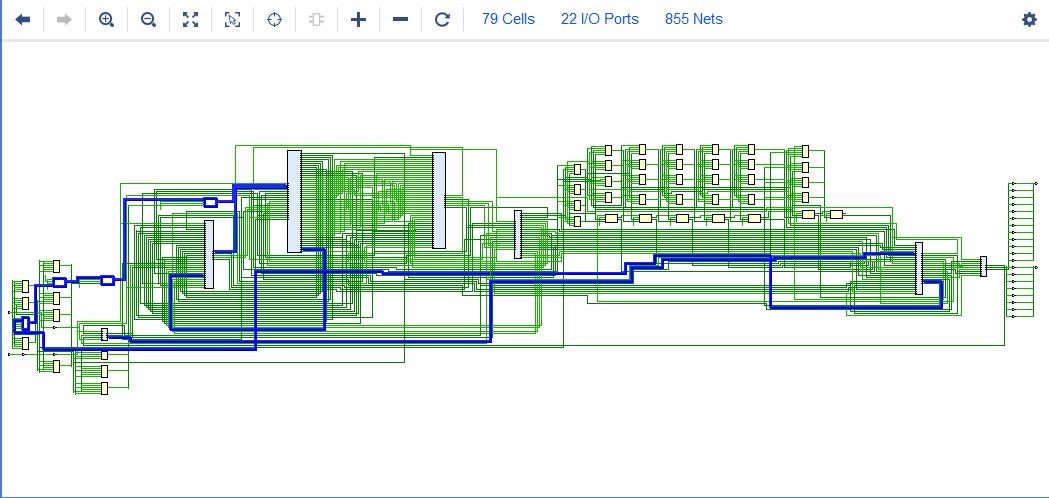
## （一）时序性能

本次的管脚约束中，将时钟周期设置为10.00ns，观察综合后的时序裕量为0.440ns



预计时钟周期最短应为9.560ns，因此频率最高大致为

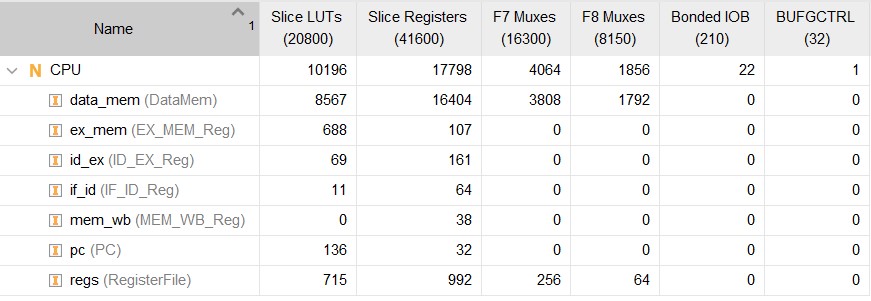
观察关键路径：

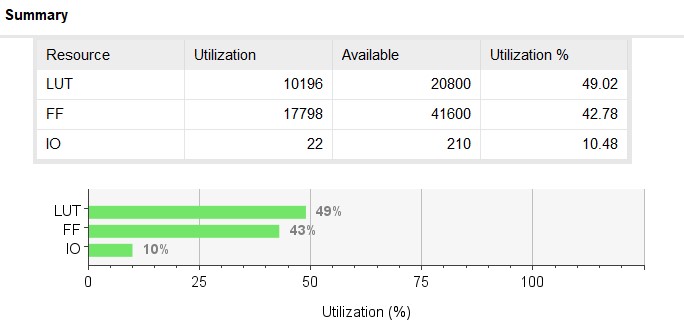


可以看到关键路径是包含数据寄存器DataMem和冒险检测单元的路径。因此处理器的频率主要被数据存储器和冒险检测单元所限制。

## （二）逻辑资源占用情况

处理器的逻辑资源占用情况如下：



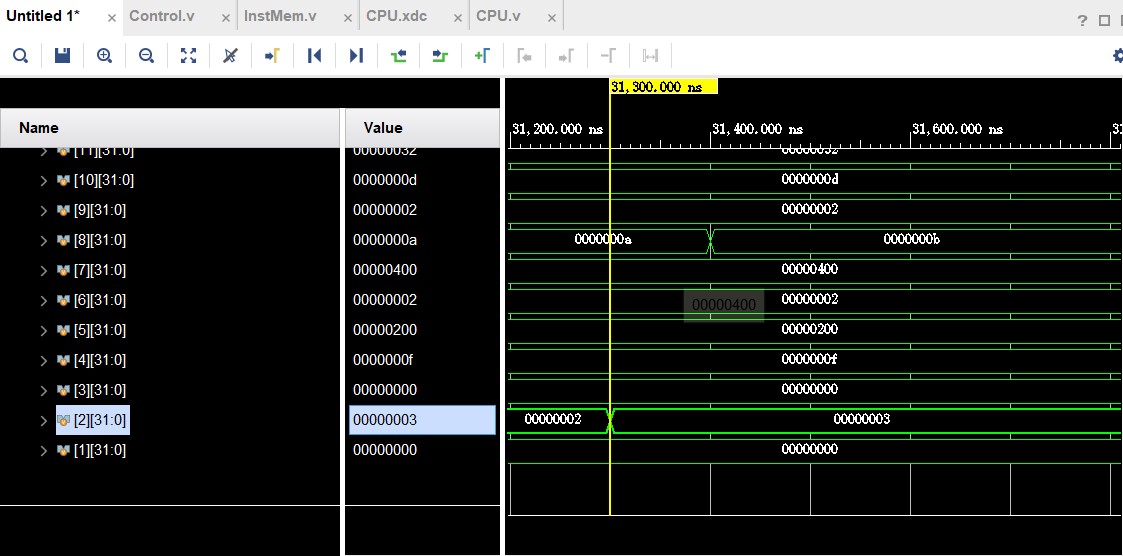


# 五、仿真验证

本次实验经过了三种方法匹配字符串，汇编程序见的test文件夹。

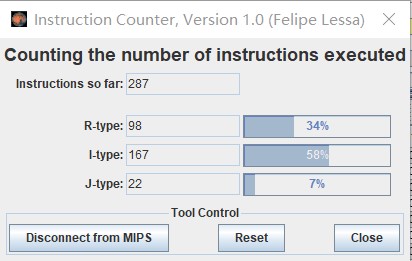
进行多次测试，答案均是正确的。

进行仿真结果如下：



可以看到，最终寄存器$v0的值为3。

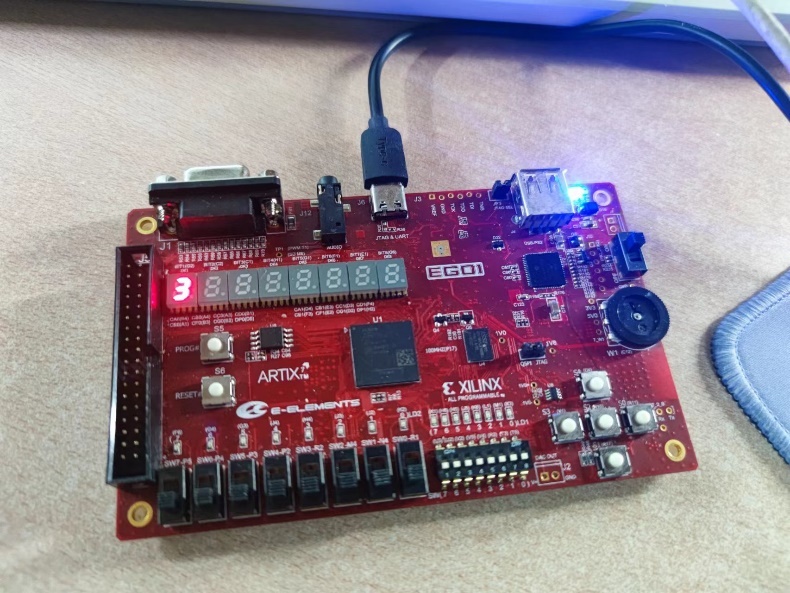
同时我们用MARS模拟器统计一个汇编文件当中的指令数，得到总共有287条指令。



同时我们通过vivado仿真的结果看出，经过了312个周期，CPU计算得到了结果。所以此时用的CPI为312/287=1.09，但是主频相对于单周期处理器有巨大的提升。

# 六、硬件调试情况

将设计代码使用Vivado 2021.2烧录到FPGA上之后运行，七段数码管上很快便出现结果“3”。通过按下复位键，也发现结果正确。



# 七、心得体会

这次实验是我到目前为止花时间最多的一次，同时也遇到了很多bug需要解决。但同时也有很多地方做了妥协，比如没有很好进行外设的实现，而是折中的利用汇编代码来有瑕疵的实现了数码管显示结果的要求（具体见上文描述），同时由于外设实现的不好，也没有完成UART的选做要求，这是我需要努力的地方。

同时，我基本上完全独立完成了本次实验。在编写代码的过程中，有一些在理论课上不懂的地方，通过实验我也有了更深的了解，我非常喜欢这种在实践中学习以及巩固知识的过程，也让我对课上所讲的流水线有了更深的理解。

在遇到的具体的bug中，我觉得最让我烦心的是对于冒险的处理，stall和转发的控制花费了我大量的时间，最后将自己写的汇编程序正确地跑在CPU上的时候也有很大的成就感。

这次实验也有很多不足之处。首先，处理器的频率没有很高，没有进行更深入的优化；第二，存储器没有使用更好的IP Core，因此在切换汇编程序等操作上耗费了一些时间；第三，一些细节上也编写的稍显繁琐，没有进行优化，等等。

总而言之，本次实验让我有很多收获。最后感谢老师在这一学期的倾情讲授，感谢助教的悉心指导，没有老师和助教的帮助，我也难以得到这么多的收获。

# 附录 支持的指令格式

