[在此处键入文档摘要。摘要通常为文档内容的简短概括。在此处键入文档摘要。摘要通常为文档内容的简短概括。]

32位MIPS处理器实验报告

[键入文档副标题]

汤宸

成大立 2016011029

李云飞 2016011017

目录

[实验目的（包含需求） 2](#_Toc519845639)

[设计方案 2](#_Toc519845640)

[成员分工 2](#_Toc519845641)

[整体架构 2](#_Toc519845642)

[ALU 2](#_Toc519845643)

[单周期 2](#_Toc519845644)

[流水线 3](#_Toc519845645)

[模块设计 3](#_Toc519845646)

[ALU 3](#_Toc519845647)

[汇编器 3](#_Toc519845648)

[串口 3](#_Toc519845649)

[定时器外设（这里面把中断处理写了） 3](#_Toc519845650)

[单周期 3](#_Toc519845651)

[流水线 7](#_Toc519845652)

[关键代码 7](#_Toc519845653)

[仿真结果 7](#_Toc519845654)

[调试情况 7](#_Toc519845655)

[单周期处理器 7](#_Toc519845656)

[综合情况 7](#_Toc519845657)

[感想体会 7](#_Toc519845658)

# 实验目的（包含需求）

# 设计方案

## 成员分工

## 整体架构

### ALU

### 单周期

单周期数据通路为指令与数据存储器分开的哈佛体系架构。数据通路上的硬件有程序计数器处理单元PC、控制单元Control、指令存储器InstructinMemory、寄存器堆RegisterFile、立即数扩展单元、算数逻辑单元ALU、数据存储器DataMemory以及若干多路选择器、加法器等。此外，为了支持中断处理，在CPU之外添加了定时器外设Peripheral和串口IO设备UART。处理器时钟通过分频单元cpu\_clk生成。



图 单周期CPU整体框图

### 流水线

## 模块设计

### ALU

### 汇编器

汇编器的作用是将MIPS汇编代码翻译成机器码，并且写入CPU的InstructionMemory里面。本实验中的汇编器支持翻译的汇编代码文件可以包含：？？指令，标签，注释，空行等，并且可以指定写入指令存储器时的首地址。为了方便调试，我们的汇编器将翻译好的机器码直接写成verilog语句，并且在每条语句之后用注释打出可读性更好的汇编指令。

### 串口

UART协议串口主要包括串口时钟生成、接收机和发射机三个部分。我们的串口工作在9600波特率，而FPGA的系统时钟为100MHz，因此需要一个时钟生成模块来产生合适的分频驱动串口。串口时钟频率为9600\*16 = 153.6kHz，需要对系统时钟作651分频。分频通过计数器实现，每数到651个上升沿，输出一个高电平，否则输出低电平。

我们对串口状态uart\_con的设置方式与指导书略有不同。

为了与中断处理逻辑相配合，我们对接收机和发射机的输出状态作了以下约定：

接收机的输出状态表示有没有收到新数据。当接收机收到一个数据的结束位后，输出一个高电平，然后在下一个串口时钟置0，也就是每次收完新数据，输出一个高电平脉冲。

发射机的输出表示当前发射机是否处于发送状态。在发射机发送起始位到结束位之间，发送状态置高电平，否则置低电平。

### 定时器外设（这里面把中断处理写了）

### 单周期

#### CPU时钟生成

ego1开发板的系统时钟为100MHz，单周期处理器的数据通路比较长，可能不能在一个系统时钟周期内跑完。因此需要从系统时钟生成一个更慢的时钟驱动cpu。这里我们通过简单的计数方法对系统时钟作分频，可以生成占空比50%，频率为系统时钟的时钟，n取决于计数器的位宽。用系统时钟上升沿驱动计数器计数，每当计数到溢出，就使输出的clk取反，并且把计数器清零。

#### 控制单元

控制单元根据从指令存储器取出的instruction以及CPU当前所处的状态（用户态，内核态，有无中断）生成数据通路中其他模块需要的控制信号。控制信号包括PC更新的源PCSrc，写回寄存器的选择RegDst，寄存器堆写使能RegWr，ALU两个操作数的源ALUSrc1、ALUSrc2，ALU执行的运算类型选择ALUFun，ALU运算是否有符号选择Sign，数据存储器（包含DataMemory，Peripheral，UART）读使能MemRd，数据存储器写使能MemWr，写回寄存器数据源MemtoReg，立即数扩展使能ExtOp，高位立即数使能LUOp。

最高优先级的判断是当前CPU处于用户态还是内核态。用户态指的是CPU执行用户代码，允许中断；而内核态是不接收中断的，CPU在执行中断处理程序或者异常处理程序。//还没有写用户态和内核态在控制单元的输出上有什么区别。

其次，在用户态的前提下判断是否发生中断。如果中断，那么控制单元应该产生能够保存现场的信号。写入寄存器目标设为26号（$k0）寄存器，即将写入的值选择当前PC值，PCSrc选择异常代码入口0x80000004。

如果在用户态且没有发生中断，就应该按照指令的opcode和funct识别出指令类型，再相应地给出数据通路各信号的选通情况。我们设计的真值表如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Opcode | Funct | PCSrc | RegDst | RegWr | ALUSrc1 | ALUSrc2 | ALUFun | Sign | MemRd | MemWr | MemtoReg | ExtOp | LUOp |
| add | 000000 | 100000 | 000 | 00 | 1 | 0 | 0 | 000000 | 1 | 0 | 0 | 00 | x | x |
| addu | 000000 | 100001 | 000 | 00 | 1 | 0 | 0 | 000000 | 0 | 0 | 0 | 00 | x | x |
| sub | 000000 | 100010 | 000 | 00 | 1 | 0 | 0 | 000001 | 1 | 0 | 0 | 00 | x | x |
| subu | 000000 | 100011 | 000 | 00 | 1 | 0 | 0 | 000001 | 0 | 0 | 0 | 00 | x | x |
| and | 000000 | 100100 | 000 | 00 | 1 | 0 | 0 | 011000 | 0 | 0 | 0 | 00 | x | x |
| or | 000000 | 100101 | 000 | 00 | 1 | 0 | 0 | 011110 | 0 | 0 | 0 | 00 | x | x |
| xor | 000000 | 100110 | 000 | 00 | 1 | 0 | 0 | 010110 | 0 | 0 | 0 | 00 | x | x |
| nor | 000000 | 100111 | 000 | 00 | 1 | 0 | 0 | 010001 | 0 | 0 | 0 | 00 | x | x |
| sll | 000000 | 000000 | 000 | 00 | 1 | 1 | 0 | 100000 | x | 0 | 0 | 00 | x | x |
| srl | 000000 | 000010 | 000 | 00 | 1 | 1 | 0 | 100001 | x | 0 | 0 | 00 | x | x |
| sra | 000000 | 000011 | 000 | 00 | 1 | 1 | 0 | 100011 | x | 0 | 0 | 00 | x | x |
| slt | 000000 | 101010 | 000 | 00 | 1 | 0 | 0 | 110101 | 1 | 0 | 0 | 00 | x | x |
| jr | 000000 | 001000 | 011 | x | 0 | x | x | x | x | 0 | 0 | x | x | x |
| jalr | 000000 | 001001 | 011 | 00 | 1 | x | x | x | x | 0 | 0 | 10 | x | x |
| lw | 100011 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 0 | 1 | 0 | 01 | 1 | 0 |
| sw | 101011 |  | 000 | x | 0 | 0 | 1 | 000000 | 0 | 0 | 1 | x | 1 | 0 |
| lui | 001111 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 0 | 0 | 0 | 00 | x | 1 |
| addi | 001000 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 1 | 0 | 0 | 00 | 1 | 0 |
| addiu | 001001 |  | 000 | 01 | 1 | 0 | 1 | 000000 | 0 | 0 | 1 | 00 | 1 | 0 |
| andi | 001100 |  | 000 | 01 | 1 | 0 | 1 | 011000 | 0 | 0 | 0 | 00 | 0 | 0 |
| slti | 001010 |  | 000 | 01 | 1 | 0 | 1 | 110101 | 1 | 0 | 0 | 00 | 1 | 0 |
| sltiu | 001011 |  | 000 | 01 | 0 | 0 | 1 | 110101 | 0 | 0 | 0 | 00 | 1 | 0 |
| beq | 000100 |  | 001 | x | 0 | 0 | 0 | 110011 | x | 0 | 0 | x | 1 | x |
| bne | 000101 |  | 001 | x | 0 | 0 | 0 | 110001 | x | 0 | 0 | x | 1 | x |
| blez | 000110 |  | 001 | x | 0 | 0 | 0 | 111101 | x | 0 | 0 | x | 1 | x |
| bgtz | 000111 |  | 001 | x | 0 | 0 | 0 | 111111 | x | 0 | 0 | x | 1 | x |
| bltz | 000001 |  | 001 | x | 0 | 0 | 0 | 111011 | x | 0 | 0 | x | 1 | x |
| j | 000010 |  | 010 | x | 0 | x | x | x | x | 0 | 0 | x | x | x |
| jal | 000011 |  | 010 | 10 | 1 | x | x | x | x | 0 | 0 | 10 | x | x |
| 异常 |  |  | 101 | 11 | 1 | x | x | x | x | 0 | 0 | 10 | x | x |
| 中断 |  |  | 100 | 11 | 1 | x | x | x | x | 0 | 0 | 11 | x | x |

表 控制单元真值表

#### PC

PC单元的作用是计算下一条指令的地址。其中PC的最高位还用来作为CPU用户套/内核态的标志位，最高位为0表示用户态，最高位为1表示内核态。用户态和内核态应该由中断决定，而不是PC。因此，需要给PC的跳转做一些规定，以满足标志位的设定。首先，只有中断和异常才能给PC最高位置1。只有jr和jalr指令才能使PC最高位置0。其他情况下PC的跳转都应该保持PC最高位不变。

PC单元的更新结果将被用来送入指令存储器取址，而且PC的最高位不参与取址。

为了满足我们的CPU功能需求，PC更新的取值有6种可能：PC\_plus\_4，分支目标Branch\_target，跳转目标Jump\_target，寄存器中的值Databus1，中断入口0x80000004，异常入口0x80000008。

PC\_plus\_4的计算需要保证PC最高位不变，因此取为原PC最高位和通常的(PC+4)结果低31位拼接。在没有发生分支、跳转的情况下，PC应该更新为PC\_plus\_4。Branch\_target计算方法如图所示，其中Z是ALU的输出。如果Z的最低位是1，说明有可能要发生条件分支，选通PC\_plus\_4+偏移量；如果Z的低位是0，说明不可能发生条件分支，应该选通正常的PC\_plus\_4。Jump\_target的值直接取为PC+4的高四位和指令中的立即数偏移拼接的结果。Databus1是为jr和jalr指令准备的，它们会将寄存器中的值赋给PC的下一次取值。



图 Branch\_target计算方法

#### 寄存器堆

寄存器堆支持两个读取端和一个写入端。读取通过组合逻辑实现，可以随时读出数据；写入通过时序逻辑实现，一条指令如果要写回寄存器，会在下一个周期的时钟上升沿实现寄存器中数据的更新。写回寄存器的目标通过多路选择器选择，除了rt、rd，还有31号和26号寄存器。写回31号寄存器用来支持jal指令，写回26号寄存器用来支持异常和中断保存现场的需要。



图 寄存器堆设计

#### 数据存储器与写回部分

我们的单周期设计中，数据存储器包含三个部分：DataMemory也即通常意义的内存，Peripheral专为外设开辟的存储段，UART专为串口开辟的存储段。这三个部分的地址都分布在4G存储空间中，其中DataMemory地址从0x00000000~0x000003ff，实际使用的地址为0x00000000~0x0000007f。Peripheral地址从0x40000000~0x40000014，UART地址从0x40000018~0x40000023。这三部分的写操作都是通过指令译码得到的控制信号使能的，由于一条指令只能指定向一个地址里写，因此三部分存储器不会产生冲突，本质上和只有DataMemory的时候并没有区别。在读存储器时，三个存储器同时被读使能，但是对于超过地址范围的存储器，rdata取0，因此只需要将三个存储器的rdata端输出或起来，就能合并出一个读到的数。



图 写回寄存器多路选择

在写回寄存器的多路选择器中，我们设置了四种选择情况：把ALU计算结果写回；将存储器中读取的结果写回；将最高位置0的PC\_plus\_4写回；将当前PC写回。

在原先的设计中，后两路选择合并为一个PC\_plus\_4，但是该设计不能实现jr和jalr指令把PC最高位清零。而在我们的新设计中，最高位置0的PC\_plus\_4对应的是jalr,jal指令，使得以后执行jr和jalr指令的时候取出寄存器中的值最高位是0，从而使PC[31]清零。

第四路选择PC是为中断准备的。由于中断到来的时候，控制单元立刻将控制信号置为保存中断现场需要的信号，在中断到来时正在执行的指令并没有执行完。因此，在中断现场地址$k0寄存器中要存的应该是这条没有正确执行完的指令的地址，以便中断处理程序结束后跳回正常处理程序时能够把这条指令重新执行一次。

### 流水线

#### 数据通路

#### 转发单元

流水线CPU中共设计了7个转发通路，以应对不同的转发需求。

ForwardA和ForwardB分别控制ALU的两个操作数（$rs端和$rt端）的输入，这里解决的冲突的典型例子是R型-R型冲突。如果上一条指令要写寄存器，并且与本条指令的源寄存器相同，则将上条指令的运算结果直接从EX/MEM寄存器返回到ALU输入；如果上上条指令要写寄存器，并且与本条指令的源寄存器相同，则将上上条指令的运算结果从MEM/WB寄存器返回到ALU输入；如果上条指令和上上条指令都产生数据冲突，则以上条指令的计算结果为准。

ForwardC和ForwardD用于在ID阶段进行beq指令的跳转判断。如果上一条指令要写寄存器，本条指令是beq指令，且上条指令的目标寄存器地址与本条指令的源寄存器地址相同，则将运算结果从EX/MEM寄存器返回到RF的数据读出端进行相等判断。注意不必从ID/EX寄存器转发，因为beq指令在ID阶段会引起一个周期的阻塞，因此beq进行跳转时，上一条指令正在进行MEM访问。也不必从MEM/WB寄存器转发，因为RF可以同时读写（这个问题后续会有说明）。

ForwardPC解决的是jal-jr冲突。如果jal指令向$ra存入的地址恰好是jr指令的跳转地址，就要进行转发。在ID阶段，检测本条指令是不是jr指令，并且检查EX/MEM寄存器中jal控制信号是否为1。如果转发条件成立，PC的新值就不应该从RF读出，而应该从EX/MEM寄存器的PC中读出。这里要从EX/MEM寄存器转发，而不从ID/EX寄存器转发，原因也是jr指令会在ID阶段阻塞一个周期。

Forwardsw解决的是use-and-store冲突，或者在理论课的ppt上被称为存储器-存储器复制冲突，即上一条指令的计算结果，在本条指令中要存入存储器。检测上一条指令是否要写寄存器、本条指令是否要写存储器、本条指令的$rt与上条指令的目标存储器是否相同，若转发条件成立，从MEM/WB寄存器中将计算结果转发到存储器的数据写入端。

最后一个转发解决的是RF在同一个周期内读写的问题。我们发现，假如RF只有在时钟上升沿才能写入，那么做不到在同一个周期内既写入又读出。我们将RF改为组合逻辑，在任何时刻都能写入，但这会引起RF中数据的不稳定，因为组合逻辑会互相影响，造成混乱。因此必须加入转发功能，如果上上上条指令（间隔两条指令）要写寄存器，且目标寄存器地址与本条指令的源寄存器地址相同，则RF的数据输出端的数据应来自MEM/WB寄存器，而不是RF。

#### 冒险检测

冒险检测单元的主要功能是，根据需要将流水线阻塞一个周期，并清空某些段间寄存器的取值。它需要处理以下四种情况。

load-and-use冒险。在ID阶段，如果检测到上条指令是lw指令，且上条指令的$rt与本条指令的源寄存器地址相同，那么要求PC和IF/ID寄存器保持不变（阻塞一个周期），并且将ID/EX寄存器清空（相当于向lw指令和use指令中间插入了一条nop）。之后的工作由转发单元完成。

分支指令冒险。在ID阶段，如果检测到本条指令是beq指令，并且发现确实要发生跳转，则产生控制信号，通过PCSrc的选择向PC写入新值，并且清空IF/ID寄存器。这是因为PC+4这条指令并不是我们希望执行的，将它清空相当于在beq指令后面插入了一条nop。

跳转指令冒险。跳转指令不仅包括j和jal，还包括jr和jalr。这些操作和beq是类似的，在ID阶段发现要跳转后，一方面向PC写入新值，另一方面清空IF/ID寄存器，相当于阻塞了一个周期。

**（这里还有问题：为什么不从ID/EX寄存器转发？）**

中断信号处理。中断信号与jal指令相似，都是将某个地址存入某个特殊的寄存器，并且向PC写入某个新的取值。因此大部分硬件实现可以和jal指令复用（例如阻塞和清空寄存器），只需要做一下或操作即可，但它们也存在一定的区别。首先，jal指令向$ra写入的是PC+4，但中断信号要求向$k0写入PC。这是因为中断信号的优先级更高，当ID阶段的Control译码模块检测到IRQ中断信号时，当前正在ID阶段的指令是没有被正确执行的，当中断返回后仍要执行它，因此在WB阶段，应该将MEM/WB寄存器中的PC减4之后写入$k0。另外，中断IRQ信号会保持很多个周期的高电平，直到中断处理代码关闭中断为止，不像jal指令的控制信号只有一个周期的高电平。这就导致Control模块一直在接收中断信号，来不及把它关闭。我们借用超声测距实验中产生窄脉冲的思想，将IRQ信号延时一个周期后取非，并和原来的IRQ信号做与运算，就得到了宽度只有一个周期的IRQ上升沿指示信号，用它作为Control模块的输入。

# 关键代码

# 仿真结果

# 调试情况

## 单周期处理器

单周期处理器的调试主要涉及汇编代码、汇编器、单周期本身数据通路以及和外部设备联合的调试。调试的难度不大，只是过程比较繁琐。大部分外部单元在调通以后直接复用到流水线中，减少了后面的工作量。调试主要在软件上仿真进行，仿真结果正确以后，在硬件上通过加debug core观察中间信号以及将关键绑定在led上观察的方法很快跑通。

### 汇编器

我们的汇编器是用python写的一段小程序，本质上就是字符串处理。

中文注释的存在使得汇编器无法工作，经检查是因为文件编码格式被默认设为gbk，改为utf8问题解决。

写汇编器的时候没有和写汇编代码的同学约定好数制，调试时决定同时支持10进制和16进制，在后来改汇编代码的时候，这个设定使得代码编写更加自由。

处理跳转指令的时候，我一开始把偏移量写成了从PC到分支目标的差，事实上应该是从PC+4到分支目标的差。这个bug是在看数据通路仿真的时候发现的，再回过来改汇编器和指令存储器，过程比较愚蠢。这件事情告诉我做单元测试的必要性。比如应该在写好汇编器以后翻译一段涵盖所有支持指令的代码，然后与标准结果（可以从mars模拟器中获得）比对，确认没问题后再和后面的模块连接起来调。

### PC最高位清零问题

在原始的指令存储器设定中，PC复位时取值为0x80000000，第0条指令会使PC跳转到求最大公约数程序入口，接着PC会开始按照正常程序的要求跳转。然而根据PC最高位的约定，在此过程中它一直会保持1，也就是在执行最大公约数代码时CPU一直处于内核态，这是不可接受的。为了使最高位置0，一定需要用寄存器中的值来给PC赋值，问题就变成了怎样比较好地给寄存器里存上最大公约数入口，而这个问题可以用jal指令解决。

最终我们的指令存储器写成伪代码是

*pseudo code*

0: jump to 16

...

...

15: jr $ra

16: jump and link 15

17: start gcd

...

### 数据存储器读出冲突

在烧录debug core之后发现，串口发送的操作数成功进入串口数据地址，但是写入寄存器的数竟然和串口里的不一样，总是多出几位1。一开始从软件层面检查，发现这是一个简单的lw操作，于是去查数据通路，发现lw涉及到把数据存储器里的数存到寄存器里，而我们的设计里有三个数据存储器，原则上它们的输出只能有一个是非0的。检查仿真发现会出现超过1个输出非0的情况。问题出在DataMemory只用了地址的低几位寻址，导致实际想访问外设地址的时候，也能从DataMemory里读出数来。解决方法很简单，寻址前加入高几位的判断即可。

### 串口时序

在第一版设计中，串口接收端接到一个新数据后rx\_status置1，只有reset才会清零；uart\_con记录串口收发状态，直连到串口收发端的使能或者输出信号。这种只能支持收一个数，不满足接收两个操作数的要求。

第二版修改了串口接收端，使得rx\_status在收到新数据后保持一个串口时钟的高电平，然后置0。rx\_status依然直连到uart\_con。这样虽然有了高低电平的变化，但是不符合uart\_con数位的规定了：约定是遇到“读”串口存储器时把uart\_con[3]清零，但这个版本所做的事情是uart\_con[3]的清零不依赖“读”信号。

最终我们的方案是手动检测rx\_status的上升沿来判断收到新数据，即在cpu\_clk的每个上升沿检查rx\_status，如果前一个周期为低电平，后一个周期为高电平，说明收到一个新数据，此时给uart\_con[3]置1，然后在遇到“读”信号的时候（汇编代码控制的把串口收到的数据读到寄存器里去）给uart\_con[3]清零。（添加仿真图形）

有了设置接收状态的经验，我们对发送状态也做了类似的处理。在串口发射器里面，每一个数据发射从起始位到结束位tx\_status都置高电平，其他情况下置低电平。而在uart\_con[2]和uart\_con[4]的设置上，通过在cpu\_clk的每个上升沿检查tx\_status，如果前一个周期是高电平，后一个周期是低电平，表示发射完一个数据，置低电平，然后在遇到“写”信号的时候（把最大公约数计算结果写入串口数据地址中）把uart\_con[2]和uart\_con[4]置高电平。

### MemtoReg数据通路的修改

我们原来按照理论课上的讲解，给MemtoReg设置了三种选择，分别是从ALU计算结果，存储器读取结果和PC+4选择写入寄存器。但是在调试的时候，有一次中断恰好发生在了j loop指令，检查$k0寄存器中的值，发现存入的是循环体后面的一条指令地址，这样中断结束以后就不会再执行循环体了，而为了程序正常运行应该要重新执行j loop运行一次循环。在这个bug的启发下，我们想到不仅是j指令，每个指令在遇到中断后都不能正常执行，应该保存的中断返回地址是当前指令而不是下一条指令。所以我们给MemtoReg加了一种选择，可以将PC写入寄存器。

### 数码管按字寻址

仿真发现在可以把两个操作数正确地读进内存的情况下，扫描显示译码结果也不对。检查发现是汇编代码和数据存储器之间的接口没有对好。传给数据存储器应该是最后两位是0的地址，而汇编代码里面写的是去掉两个0的地址。修改汇编之后问题解决。

### 支持接收多组操作数

实验指导书中没有明确说需要接收多组操作数，我们为了一次性给多个测试数据，决定创造这个需求。

我们一开始的汇编程序（包含中断）是只为实现一组操作数设计的，为了支持接收多组操作数，需要在接收完一组数据之后把准备接收操作数的地址清空。但是我们不能简单地用两个寄存器实现接收操作数和清零，因为这两个寄存器还会被绑定在数码管上实现操作数显示，到时候就会出现闪烁。所以我们最后用了四个寄存器$s0,$s1,$s5,$s6实现。$s5,$s6负责从串口读操作数，每当完成一组操作数求最大公约数，就把$s5,$s6清空，准备接收新的操作数。$s0,$s1是$s5,$s6的缓存，没有清空过程，负责绑定到数码管上显示。

调试主要修改了中断部分的汇编代码以及连带的指令存储器，调试结果四个寄存器的变化情况和预想的一样。

### 定时器周期

上板子调的时候发现，数码管每段线都有亮度，应该亮的几根线和其他的线亮度区分不明显。与其他同学讨论之后发现是数码管扫描频率过高的问题。数码管扫描只在中断时进行，只要把定时器中断频率降下来，数码管的显示就清晰了。

# 综合情况

# 感想体会