32位MIPS处理器实验测试文档

Cache小组

2015年1月19日

目录

1 引言

1.1 编写目的

测试是程序开发的一个重要组成部分。需要通过测试证明程序的正确性，并且保证程序能够在各种边界条件、极端情况下都正常工作。

测试对于硬件的开发尤为重要，硬件开发中的问题往往比较隐蔽，难以准确定位，调试比较困难。据以往学长参与此实验的经验，在调试方面花费的时间要多于编写程序所花的时间，最终实验成功与否，很大程度上也取决于测试工作进行的是否充分。

因此，本小组撰写此测试文档，为该实验提出一份完整可行的测试方案，同时也记录我们调试过程中遇到的问题与解决方案，作为之后完成此实验的参考。

1.2 测试方案

本项目的测试工作从硬件模块的单元测试开始，逐渐将各个模块组合联调，使其最终能够正常运行一个操作系统。总体测试框架如下：

1. 单元测试
2. 单指令测试
3. 指令片段测试
4. 系统测试

2 单元测试

2.1 ALU模块

2.1.1 测试方法

通过拨码开关输入操作数、操作符检查运算结果正确性。使用状态机，在高频下自动产生操作数、操作符，并于VHDL内置运算结果比较，以检验在高频下连续运算结果的正确性。除此之外，还需要检查高频下乘法速度是否满足需要。

2.1.2 测试用例

除了简单测例、边界测例外，还需要测试加、减、移位运算的溢出。 测例诸如0xffffffff+1、0xffffffff+2、0x00000001-2等。

2.1.3 测试结果

所有运算操作结果正确，乘法运算可以在25MHz频率下一个周期内得出正确结果。

2.2 指令解码模块

2.2.1 功能概述

在指令解码周期的时钟上升沿，产生该指令所需的全部控制信号，并对当前指令、通用寄存器编号进行锁存。

2.2.2 测试方法

通过拨码开关输入指令，状态机始终控制在解码阶段，通过手动时钟提供时钟上升沿，将产生的控制信号输出到LED灯查看。

2.2.3 测试用例

使用全部MIPS32指令进行测试。但是因为实验板上LED灯只有16个，一次只能查看部分信号，且输入指令占用全部的32个拨码开关，导致测试效果不佳。

因此在该阶段只进行了少量测试，指令解码模块的测试工作后延至单指令测试阶段。

2.2.4 测试结果

指令解码模块在测试中发现了如下问题：

1. JAL和J指令的定义在原有指导文档上有误，地址的高四位并不发生改变。上网查找了标准的MIPS32指令进行修正，之后JAL和J指令能够正常工作。
2. ALU控制信号ALUOp错误。编码过程中曾对ALUOp进行过一次修改，但并没有体现在数据通路表中。修改了数据通路表和相关代码后工作正常。

2.3 访存模块

2.3.1 测试方法

1. 通过拨码开关输入访问存储（包括Ram、Flash、串口）的物理地址、数据、使能，将读取到的数据通过LED显示出来，检查数据的正确性。
2. 使用状态机，在高频下连续地向存储中读写数据，检查数据的正确性。

访问串口时需要在PC端同步接受、发送。

2.3.2 测试结果

Ram、Flash可在50MHz时钟下正常工作，串口在25MHz以上时钟工作时，连续两个时钟周期写入有一定几率出现错误，原因不详。考虑到操作系统使用串口时不可能出现连续两个时钟访问串口，此问题无需修复。

2.4 MMU模块

2.4.1 功能概述

该模块实现虚拟地址到物理地址的转换，并将物理地址转换为RAM、Flash、ROM的真实地址，实现TLB表的查找、重填以及相关异常的触发。

2.4.2 测试方法

主要测试地址转换过程与TLB表的查询过程。由于拨码开关和LED等数量有限，因此固定访存地址的低16位，拨码开关只改变可能影响内存映射的高16位地址。其余拨码开关通过组合，查看不同的输出信号。在VHDL代码中固化一个TLB表项，通过拨码开关控制写入TLB。

2.4.3 测试用例

测试用例参见表1

2.4.4 测试结果

该模块测试中发现了TLB命中策略的问题，两个EntryLo页的命中与标准实现相反。

后在向勇老师的课件中，找到一份TLB的详细实现方案,在此基础上稍作修改即可实现TLB。

2.5 通用寄存器模块

2.5.1 功能概述

这一模块实现数据通路上的控制信号与通用寄存器的交互，实际上是一个经过编码的片内RAM。

2.5.2 测试方法

对片内RAM的测试，包括读和写的操作，即对各个寄存器写入数据，然后读出到另一个信号处，并对其进行检查。

按照数据通路中定义的控制信号，通过拨码开关对这一模块进行输入，执行对各个寄存器的写操作，然后输入操作进行读操作，将结果通过LED灯进行显示。

2.5.3 测试结果

从输出端读到了写入的数据。

表 1: MMU模块测试用例

类型	说明
非映射地址转换	地址高16位选择非映射地址区间，观察输出的to_physical_addr信号是否正确。在非映射状态对RAM、Flash与串口均进行测试。
映射地址转换	地址高16位选择映射地址区间，观察观察输出的to_physical_addr信号是否正确。对RAM、Flash与串口均进行测试。
异常触发	不写入TLB表项进行映射地址的访存，触发TLBMiss异常。修改TLB表项，dirty位置1，触发TLBModify异常。修改输入的align_type对齐方式，触发地址不对齐异常。
串口状态位	串口状态位封装在MMU内部，在MMU模块测试串口是否可读可写。

2.6 CP0模块

2.6.1 功能概述

这一模块功能上类似一个片内RAM，但由于异常操作的存在，需要对CP0寄存器组同时进行多地址的读、写操作，因此实际实现上采用数组的方式实现。

2.6.2 测试方法

对CP0寄存器的测试与通用寄存器组类似，首先进行读写功能的测试。

除此之外，异常的操作要求一次性写入大量数据，因此需要单独进行处理。此处需要对输入端给入对应的控制信号、状态信号、数据信号。

2.6.3 测试结果

由于CP0组的所有数据都随时可以读到，可以观察到写入数据时，对应的LED灯正确变化。

2.7 异常处理模块

2.7.1 功能概述

异常处理模块主要实现异常的检查、数据转发。

2.7.2 测试方法

模仿异常发生时的情况，针对中断、TLB miss、ADE、SYSCALL、RI发生时应有的输入信号，通过拨码开关给入异常数据，然后检查给到CP0模块的输出异常数据。

2.7.3 测试结果

输出结果满足数据通路定义的信号。

3 单指令测试

3.1 算数指令

3.1.1 测试方法

算数指令是最基本的指令，因此放在最前面测试。

算数指令大多会写会通用寄存器组，由于之前已经确定通用寄存器组可以正确工作，此处只需检查给到通用寄存器组的地址、数值、写信号是否正确即可。

由于此时整个CPU的整合尚未完全完成，不能确定访存路径是否正确，通过拨码开关代替访存部分，即将所有需要从访存取得的数据改由拨码开关输入，同时手动按时钟进行输入。通过LED灯检查给通用寄存器组的写信号是否符合预期。

另外，随着测试的进行，调试工具也在同步开发。在指令测试的后半部分时，加入调试工具，在电脑端查看输出信号。

3.1.2 测试样例

由于开始时所有通用寄存器的数值均为0，需要给入恰当的数据才能有效检查，因此按照由易入难的顺序进行测试。

1. ADDIU
   ADDIU 1 0 4
   ADDIU 2 1 -1

2. ADDU
   ADDIU 1 0 4
   ADDIU 2 0 -2
   ADDU 3 1 2

3. SLT
   ADDIU 1 0 4
   ADDIU 3 0 5
   SLT 2 1 3
   SLT 2 3 1
   ADDIU 3 0 -2
   SLT 2 1 3
   SLT 2 3 1

4. SLTI
   ADDIU 1 0 4
   SLTI 3 1 3
   SLTI 3 1 5
   SLTI 3 1 -1

5. SLTIU
   ADDIU 1 0 4
   SLTIU 3 1 3
   SLTIU 3 1 5
   SLTIU 3 1 -1

6. SLTU
   ADDIU 1 0 4
   ADDIU 3 0 5
   SLTU 2 1 3
   SLTU 2 3 1
   ADDIU 3 0 -2
   SLTU 2 1 3
   SLTU 2 3 1

7. SUBU
   ADDIU 1 0 4
   ADDIU 2 0 -2
   SUBU 3 1 2

8. MULT,MFLO,MFHI
   ADDIU 1 0 4
   ADDIU 3 0 5
   MULT 1 3
   MFLO 2
   MFHI 2

9. MTLO,MTHI
   ADDIU 1 0 4
   ADDIU 3 0 5
   MTLO 1
   MTHI 3
   MFLO 2
   MFHI 2

10. AND,ANDI,LUI,NOR,OR,ORI,XOR,XORI,SLL,SLLV,SRA,SRAV,SRL,SRLV,LHU
    采用类似的方法进行测试

3.1.3 问题与解决

测试的过程中发现了一系列错误，包括写会通用寄存器的操作控制信号设计有误，写地址来源在一种控制信号组合下有误等。

通过检查给到通用寄存器的信号，可以判断是否发生错误。发生错误后，要逆着数据通路检查沿途的控制信号、数据信号。这个操作一开始通过LED灯进行检查，之后加入调试工具，可以直接在每次时钟之后看到所有沿途信号，进行检查。 最终，以上指令均检查完成

3.2 跳转指令

3.2.1 测试方法

跳转指令除了通用寄存器，还需要检查PC的变化，以及RPC的保存。

3.2.2 测试说明

跳转指令包括BEQ,BGEZ,BGTZ,BLEZ,BLTZ,BNE,J,JAL,JALR,JR,ERET
检查时，需要对正、负值都进行检查。

3.2.3 测试结果

确认以上指令可以按照实验指导书的定义运行。
另外，之后运行ucore时发现实验指导书的定义是错误的，尤其是J类指令的左移两位问题，在确定新的要求后进行了修改。

3.3 访存指令

3.3.1 测试方法

预先通过controller.py在内存与Flash中写入一段数据，之后通过load与store指令进行数据的拷贝、覆盖等操作，最后通过controller.py将结果读出与输入做比较。

3.3.2 测试说明

物理访存模块结合MMU模块共同测试，需要测试的访存指令包括LW、SW、LB、LBU、SB、LHU。按word进行操作的指令，在RAM、Flash、串口均进行测试，按half-word与byte进行操作的指令，仅对RAM做测试。

3.3.3 测试结果

测试中发现访存指令工作不稳定，表现为访存指令有一定概率错误地触发异常，或者读取出错误的数据。可能的原因是该阶段采用的手动时钟，在按下开关的瞬间可能产生毛刺，导致访存指令出现多次访存、锁存地址错误等情况。后将该测试移至指令片段测试阶段，在高速时钟下，该问题得到解决。

3.4 特殊指令

此部分包括SYSCALL,CACHE,NOP,TLBWI,采用与其他指令相同的方法进行输入，然后通过对控制信号的监控进行检查。

4 指令片段测试

项目测试进入第三个阶段。在前两阶段完成对CPU单指令运行的测试后，此阶段完成对多条、不同类型指令组合的测试。此阶段的测试目标为，将不同指令组合所产生的问题解决。同时，前两阶段的测试均使用手动时钟，在此阶段换为实验板上提供的时钟，对CPU在高频下的表现做测试。

在此阶段，我们也完成了我们调试工具的最初版本，将可能需要查看的信号输出到debug模块，通过拨码开关控制，可以查看不同的信号，信号的值通过LED灯进行显示。

4.1 连续算数指令测试

4.1.1 测试方法

对连续算数指令进行测试。

利用VHDL中的std_logic_vector生成一个32位数组，以此数组作为ROM，保存测试过程中所需的所有指令。使用汇编语言写好测试指令，在尾部加while死循环，保证运算的结果可以保存在寄存器中。用mips-linux-gnu-gcc进行编译，将其中的二进制指令部分通过我们开发的romgen工具转换为VHDL语言可以接受的形式。至此测试指令准备完毕。

按下rst开关后，将PC指向ROM起始地址，即可开始运行指令片段。

运行结束后，通过拨码开关，将结果寄存器中的数值显示在LED灯上。通过调整拨码开关查看不同寄存器的值，检查计算是否正确。

4.1.2 测试样例

测试样例参见表3

表 3: 算数指令测试样例

类型	说明
普通算数指令	包括add、sub、slt、shift等多类指令的组合，主要是对前一阶段单指令测试的补充，保证算数指令可以在组合的情况下工作。
乘法指令	主要对乘法指令的速度是否满足需求进行测试，即在mult指令后紧跟mfhi、mflo指令，测试乘法模块是否正常
综合算数测试	乘法指令与普通算数指令混合使用。

4.1.3 问题与解决

该测试中未发现问题。算数指令能够在高频下以各种组合正常执行。

4.2 连续访存指令测试

4.2.1 测试方法

对连续访存指令进行测试。

测试指令仍保存在ROM中，ROM的生成方式与算数指令相同。

利用贾开的controller.py工具，预先向Flash与RAM中写入一些数据。按下rst运行，之后仍通过controller.py工具，将RAM中的内容读出，与写入的内容做比较。

4.2.2 测试样例

测试样例参见表5。

表 5: 访存指令测试样例

类型	说明
读Flash写RAM	预先向Flash指定位置写入一段数据，测试指令中，通过while循环将该段数据读出，写入RAM的指定位置。之后通过controller.py读出RAM中的数据，与写入的数据进行对比。
读RAM写RAM	预先向RAM指定位置写入一段数据，测试指令中，通过while循环将该段数据读出，写入另一个RAM位置。之后通过controller.py读出RAM写入的数据，与写入的数据进行对比。
LB与SB指令	预先向RAM指定位置写入一段数据，测试指令中，对于每一个4byte数据，用LB指令读取第一个byte的内容，之后连续三次SB指令写入后三个byte。通过controller.py读出该段RAM的数据，每连续四个byte都应该有相同的内容。
串口连续输出	预先向RAM指定位置写入一段数据，测试指令中，通过一个while循环读出该段数据，分四次将每一个byte输出到串口地址。在电脑端将接收程序terminal的输出重定向到文件，与写入的文件进行对比。

4.2.3 问题与解决

该测试中发现串口接收端存在问题。第一次测试中发现电脑端乱码，经过检查是波特率的设置出现了问题，如果CPU端将时钟二分频，则在电脑端需要将波特率变为二分之一。第二次测试中发现电脑端接受的数据顺序相反，经过检查是大小端的问题，改变接受程序即可解决。

通过该测试，证明了串口的输出在高频下是能够正常工作的。为之后升级测试工具，将信号通过串口输出做好了准备。

4.3 模拟操作系统测试

4.3.1 测试方法

模拟操作系统的工作过程进行测试。

将一段测试代码用mips-linux-gnu-gcc进行编译，之后修改编译出的二进制文件（linux下可用bless工具），将二进制文件前四个byte改为该段指令的条数。之后将该二进制文件烧入Flash起始处。

ROM中实现一段访操作系统bootloader的指令。首先从Flash起始位置读入指令条数，之后循环将指令读入RAM起始地址，之后jump到RAM起始地址执行。

由于前一测试已经将串口输出调通，因此可以将代码的运行结果通过串口输出到电脑。

4.3.2 测试样例

Flash中的测试代码可以是任意mips32指令的组合。

4.3.3 问题与解决

该测试中未发现问题。

5 系统测试

项目测试进入第四个阶段。在该阶段，直接采用ucore作为我们的测试程序。从lab1到lab8逐渐增加功能，最终运行一个完整的操作系统。

该阶段在软件硬件接口方面可能出现问题，原有操作系统与硬件之间的接口可能需要修改。此外，异常处理、系统调用、中断等情况，在前期指令片段测试中比较难以模拟，有可能在此阶段出现问题。

5.1 测试内容

采用ucore的lab1到lab8作为测试内容。 各个lab的说明详见表7

表 7: lab1至lab8简要说明

编号	说明
lab1	通过bootloader启动操作系统， 初始化异常处理向量，初始化时钟中断。 运行正常则可以向串口周期性的输出数字。
lab2	在lab1的基础上，完成物理内存的初始化。 操作系统内部通过assert进行检查， 如果出现问题会报assert failed错误 并进入debug monitor
lab3	在lab2的基础上，完成虚拟内存的管理， 包括对TLB表项的初始化、重填等操作。
lab4	创建内核级线程。 该lab正常运行结束后也会进入debug monitor模式， 只要根据终端的输出，观察到通过了一系列check即可。
lab5	创建用户级线程。 正常运行结束的表现同lab4。
lab6	实现线程间的调度。
lab7	实现线程的同步互斥。
lab8	实现文件系统，能够进入用户态，通过串口输入指令并执行。

5.2 测试方法

在系统测试阶段，使用本组开发的调试工具进行测试。通过debug工具设置断点，运行至断点后通过串口输出所有可能需要查看的信号。在PC端查看信号进行调试。

另一种测试方法为修改操作系统。在操作系统中插入输出语句，重新编译操作系统后烧入Flash执行。通过查看启动过程中的输出信息，定位出现问题的位置。

该阶段测试需要用到两个串口。

5.3 问题与解决

该阶段发现了比较多的问题，在此逐个罗列问题的现象、原因与解决方案。

5.3.1 操作系统启动

问题现象：原先设定为按下rst键后执行硬件初始化，再按下clk之后开始运行。但是上述过程总是需要反复多次，才能启动操作系统。

问题原因：硬件初始化问题。VHDL语言中在信号定义时给定的初始值是不稳定的。需要根据rst信号进行一次赋值，才能完成正常初始化。

解决方案：在每一个独立模块中，都添加rst信号，与实验板的rst键相连。对于每一个用到的信号，都在rst为0时赋值为应有的初始值。

5.3.2 代码段覆盖

问题现象：在lab1测试过程中，随着程序的不断执行，一部分指令被数据覆盖，导致操作系统崩溃。

问题原因：编译器版本不同，导致代码段组装顺序不同。初始化内核栈指针时，将其设置为了代码中间的一个函数，导致运行过程中代码段被覆盖。

解决方案：根据不同的编译器版本，可能需要修改kern_boot函数，设置栈指针为代码段顶端的kern_init函数。

此次实验中采用的编译工具为mips-linux-gnu工具链，版本为(Sourcery CodeBench Lite 2014.05-27)2.24.51.20140217。在该版本下只有lab1出现了这个问题。

5.3.3 时钟中断

问题现象：在lab1测试过程中，最后无法触发时钟中断。不能周期性地输出数字。

问题原因：在os_lab中，启动过程中是通过MASK位进行中断屏蔽，保证不会被时钟中断所打断。此次试验在硬件上没有提供对MASK位的支持，而是初始化compare寄存器全为1，compare过大导致不能触发时钟中断。

解决方案：在操作系统上进行修改。CP0的Compare寄存器初始化为0xFFFFFFFF，保证初始化结束之前一定不会触发时钟中断。初始化完成后增加一次对clock_intr函数的调用，将Compare寄存器重填为设定的数字，之后即可触发时钟中断。

5.3.4 CP0寄存器编号

问题现象：lab3不能正常触发TLB异常。

问题原因：CP0寄存器编号错误。在原实验指导文档中的CP0寄存器编号有误，导致异常相关的信息被写入了错误的寄存器，异常触发失败。

解决方案：参考《See MIPS Run》，其中有标准MIPS32的CP0寄存器编号，按照书中的编号实现。

5.3.5 串口地址

问题现象：在lab3中，串口不能够正常输出启动过程的信息。但在lab1和lab2中能够正常工作。

问题原因：从lab3开始，代码支持了在实验板与qemu两个平台上的运行，两平台的串口地址不同。

解决方案：在编译操作系统前需要先调用to_thin脚本，将操作系统转换为实验板上的版本。

5.3.6 系统调用返回

问题现象：在lab5中，syscall系统调用创建用户级线程，但是syscall异常返回后程序陷入死循环。

问题原因：lab5中的syscall异常返回后，下一条指令就是用户态程序的第一条指令，在取指阶段会立刻触发tlbmiss异常。原有实现中，eret_enable信号在下一条指令的解码阶段消除，但是这种情况下，根本没有进入解码阶段，反复从eret的返回地址处取指令，导致程序陷入死循环。

解决方案：将eret_enable信号在取指阶段消除，用户态程序即可正常调用。

5.3.7 Flash访问

问题现象：lab8加载文件系统是SFS_MAGIC与预设值不等。

问题原因：Flash数据线只有16条，所以实际上Flash连续的4个byte中只有2个为有效数据，因此产生了两种Flash访问方式如下：

一种方式为每次访问Flash的4个byte，得到32位数据中只有低16位有效。之后在软件层进行控制，连续访问两次，将结果移位拼接得到32位有效数据。另一种方式为每次访问Flash在硬件上访问8个byte，硬件上将两次得到的数据进行拼接，对操作系统层提供与RAM访问相同的接口。

此次实验中在硬件上实现方式为第一种方式，但是在操作系统上要求为第二种方式。

解决方案：lab1到lab7不需要进行任何修改，因为所有Flash访问操作均是在bootloader部分完成。在lab8中由于涉及到文件系统sfs.img的加载，需要修改lab8/kern/fs/devs/dev_disk0.c中的disk0_read_blks_nolock函数，将Flash起始地址与DISK0_BLKSIZE均乘2，之后通特别外实现的memcpy_flash将sfs.img加载到内存中。

5.3.8 地址信号

问题现象：lab8多个线程切换时内存管理错误。

问题原因：由于实现方式的原因，取指令和访存均需要两个上升沿才能够完成。在取指令模块和MMU模块，第二个访存上升沿时又对地址信号进行了一次赋值，而这时的地址信号可能已经发生了改变，导致内存管理出现异常。

解决方案：在取指令模块和MMU模块增加对地址的锁存，且只有在访存或取指令的第一个上升沿才允许对地址进行重新赋值。