PPCA2016 大作业：带 5 级流水的 MIPS 模拟器

柯嵩宇

2016 年 6 月 29 日

目录

1 基本要求

1.1 时间

见课程主页

1.2 实现语言

C++、Java 等，不推荐使用 Python，如果使用了其他的语言请提前告知助教以便助教配置评测环境

1.3 代码管理与提交

1.3.1 Git

要求全程使用 git 作为版本管理工具，学习并正确使用 git[1]。

1.3.2 Bitbucket

要求把代码推送到 bitbucket[2]上，在 bitbucket 上面的项目名字为 `mips-simulator`[3]，并且给用户 `BreakVoid` 一个 Read 权限[4]。

2 实现要求

要求实现的是一个能够正确解释 MIPS 汇编语言中与整数运算有关的一个子集的模拟器。关于程序的正确性方面，要求实现的模拟器有和 SPIM 相同的行为。

2.1 要求实现的汇编语言特性

一下提及的指令都以 SPIM Quick Reference 和 SPIM 的实际执行效果为准[5]。

2.1.1 汇编器指令

需要实现的 MIPS 汇编器指令如下:

.align n Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

.ascii str Store the string in memory, but do not null-terminate it.

[1]正确使用：助教可以通过你们的历史版本和提交记录了解你们的工作情况。如果只有一次或者非常少量的提交记录，那么将被判定为不正确的使用 Git，会在一定程度上影响成绩

[2]一个与 Github 类似的远程代码仓库，区别在于 Github 在免费使用时创建私有仓库的权限，而 bitbucket 的每个账户可以有 5 个免费的私有仓库

[3]项目名字很重要，这个关系到评测，直接影响到助教能否正确的通过批量测试脚本获取你们的代码

[4]禁止把仓库的权限设置成公开的 (public)

[5]SPIM Quick Reference: http://acm.sjtu.edu.cn/compiler/spim_ref.html

.asciiz str Store the string in memory and null-terminate it.

.byte b1, ..., bn Store the n values in successive bytes of memory.

.data The following data items should be stored in the data segment. If the optional argument addr is present, the items are stored beginning at address addr.

.half h1, ..., hn Store the n 16-bit quantities in successive memory halfwords.

.space n Allocate n bytes of space in the current segment (which must be the data segment in SPIM).

.text The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, the items are stored beginning at address addr.

.word w1, .. ., wn Store the n 32-bit quantities in successive memory words.

2.1.2 SPIM 指令集

算术和逻辑指令[6]:

- abs Rdest, Rsrc — Absolute Value
- add Rdest, Rsrc1, Src2 — Addition (with overflow)
- addi Rdest, Rsrc1, Imm — Addition Immediate (with overflow)
- addu Rdest, Rsrc1, Src2 — Addition (without overflow)
- addiu Rdest, Rsrc1, Imm — Addition Immediate (without overflow)
- and Rdest, Rsrc1, Src2 — AND
- andi Rdest, Rsrc1, Imm — AND Immediate
- div Rdest, Rsrc1, Src2 — Divide (with overflow)
- divu Rdest, Rsrc1, Src2 — Divide (without overflow)
- mul Rdest, Rsrc1, Src2 — Multiply (without overflow)
- mulo Rdest, Rsrc1, Src2 — Multiply (with overflow)
- mulou Rdest, Rsrc1, Src2 — Unsigned Multiply (with overflow)
- neg Rdest, Rsrc — Negate Value (with overflow)
- negu Rdest, Rsrc — Negate Value (without overflow)

[6]下面提及的 Rdest 为存储结果的寄存器，Rsrc, Rsrc1, Rsrc2 为操作数所在的寄存器，Src2 表示这个操作数既可以是立即数也可以是一个寄存器中的数据（即寄存器标号）

- nor Rdest, Rsrc1, Src2 NOR
- not Rdest, Rsrc NOT
- or Rdest, Rsrc1, Src2 OR
- ori Rdest, Rsrc1, Imm OR Immediate
- rem Rdest, Rsrc1, Src2 Remainder[7][8]
- remu Rdest, Rsrc1, Src2 Unsigned Remainder
- rol Rdest, Rsrc1, Src2 Rotate Left[9]
- ror Rdest, Rsrc1, Src2 Rotate Right
- sll Rdest, Rsrc1, Src2 Shift Left Logical
- sllv Rdest, Rsrc1, Rsrc2 Shift Left Logical Variable
- sra Rdest, Rsrc1, Src2 Shift Right Arithmetic
- srav Rdest, Rsrc1, Rsrc2 Shift Right Arithmetic Variable
- srl Rdest, Rsrc1, Src2 Shift Right Logical
- srlv Rdest, Rsrc1, Rsrc2 Shift Right Logical Variable
- sub Rdest, Rsrc1, Src2 Subtract (with overflow)
- subu Rdest, Rsrc1, Src2 Subtract (without overflow)
- xor Rdest, Rsrc1, Src2 XOR
- xori Rdest, Rsrc1, Imm XOR Immediate

常数操作指令：

- li Rdest, imm Load Immediate
- lui Rdest, imm Load Upper Immediate[10]

比较指令[11]：

[7]Put the remainder from dividing the integer in register Rsrc1 by the integer in Src2 into register Rdest. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

[8]On my computer, `rem -5 2` and `rem -5 -2` both return -1 which is same to C++'s remainder.

[9]Rotate the contents of register Rsrc1 left (right) by the distance indicated by Src2 and put the result in register Rdest.

[10]Load the lower halfword of the immediate imm into the upper halfword of register Rdest. The lower bits of the register are set to 0.

[11]对于所有的比较指令，结果为真返回 1，结果为假则返回-1

- seq Rdest, Rsrc1, Src2 Set Equal[12]
- sge Rdest, Rsrc1, Src2 Set Greater Than Equal
- sgeu Rdest, Rsrc1, Src2 Set Greater Than Equal Unsigned
- sgt Rdest, Rsrc1, Src2 Set Greater Than
- sgtu Rdest, Rsrc1, Src2 Set Greater Than Unsigned
- sle Rdest, Rsrc1, Src2 Set Less Than Equal
- sleu Rdest, Rsrc1, Src2 Set Less Than Equal Unsigned
- slt Rdest, Rsrc1, Src2 Set Less Than
- slti Rdest, Rsrc1, Imm Set Less Than Immediate
- sltu Rdest, Rsrc1, Src2 Set Less Than Unsigned
- sltiu Rdest, Rsrc1, Imm Set Less Than Unsigned Immediate
- sne Rdest, Rsrc1, Src2 Set Not Equal

分支与跳转[13]：

- b label Branch instruction.
  Unconditionally branch to the instruction at the label.
- beq Rsrc1, Src2, label Branch on Equal
- beqz Rsrc, label Branch on Equal Zero
- bge Rsrc1, Src2, label Branch on Greater Than Equal
- bgeu Rsrc1, Src2, label Branch on GTE Unsigned
- bgez Rsrc, label Branch on Greater Than Equal Zero
- bgezal Rsrc, label Branch on Greater Than Equal Zero And Link
- bgt Rsrc1, Src2, label Branch on Greater Than
- bgtu Rsrc1, Src2, label Branch on Greater Than Unsigned
- bgtz Rsrc, label Branch on Greater Than Zero
- ble Rsrc1, Src2, label Branch on Less Than Equal

[12]Set register Rdest to 1 if register Rsrc1 equals Src2 and to be 0 otherwise.

[13]每条 MIPS 的汇编指令都可以有一个或者多个 label，分支与跳转语句中的 label 就是汇编指令的 label，用于跳转到具有特定 label 的语句，注意 label 是全局唯一的

- bleu Rsrc1, Src2, label Branch on Less Than Equal for Unsigned
- blez Rsrc, label Branch on Less Than Equal Zero
- bgezal Rsrc, label Branch on Greater Than Equal Zero And Link
- bltzal Rsrc, label Branch on Less Than And Link
- blt Rsrc1, Src2, label Branch on Less Than
- bltu Rsrc1, Src2, label Branch on Less Than Unsigned
- bltz Rsrc, label Branch on Less Than Zero
- bne Rsrc1, Src2, label Branch on Not Equal
- bnez Rsrc, label Branch on Not Equal Zero
- j label Jump
- jal label Jump and Link
- jalr Rsrc Jump and Link Register
  Unconditionally jump to the instruction at the label or whose address is in register Rsrc. Save the address of the next instruction in register 31.
- jr Rsrc Jump Register
  Unconditionally jump to the instruction whose address is in register Rsrc.

Load 指令[14]:

- la Rdest, address Load Address[15]
- lb Rdest, address Load Byte
- lbu Rdest, address Load Unsigned Byte
  Load the byte at address into register Rdest. The byte is sign-extended by the lb, but not the lbu, instruction.
- ld Rdest, address Load Double-Word
  Load the 64-bit quantity at address into registers Rdest and Rdest + 1.
- lh Rdest, address Load Halfword
- lhu Rdest, address Load Unsigned Halfword
  Load the 16-bit quantity (halfword) at address into register Rdest. The halfword is sign-extended by the lh, but not the lhu, instruction

[14]从指定的内存地址读取数据并且保存到指定的寄存器，内存地址以 `offset(Rhead)` 或者是 `label` 的形式给出，即，寄存器中的首地址 + 偏移量，偏移量**可正可负**

[15]Load computed address, not the contents of the location, into register Rdest.

- lw Rdest, address — Load Word
  Load the 32-bit quantity (word) at address into register Rdest.

Store 指令[16]:

- sb Rsrc, address — Store Byte
  Store the low byte from register Rsrc at address.
- sd Rsrc, address — Store Double-Word
  Store the 64-bit quantity in registers Rsrc and Rsrc + 1 at address.
- sh Rsrc, address — Store Halfword
  Store the low halfword from register Rsrc at address.
- sw Rsrc, address — Store Word
  Store the word from register Rsrc at address.

数据移动指令:

- move Rdest, Rsrc — Move
  Move the contents of Rsrc to Rdest.

特殊指令:

- nop — No operation
  Do nothing.[17]
- syscall — System Call
  Register v0 contains the number of the system call (see System Services) provided by SPIM.[18]

系统调用:

[16]从指定的内存地址读取数据并且保存到指定的寄存器，内存地址以 `offset(Rhead)` 或者是 `label` 的形式给出

[17]注意一下，这里的 Do nothing 并不是什么都不做，而是执行一条指令，指令的目的是什么都不做，它会在占用流水线并且按照 5-stage 的划分在每个阶段都占用相应的部分，这条指令主要是调试代码的时候使用。可以在每条语句执行之后插入 nop 指令，这样程序员就可以清楚的知道每一条指令的执行结果。

[18]只需要模拟下面提到的几个系统调用即可

系统调用编号	说明	参数	结果
1	输出一个整数	$a0: 需要输出的整数	N/A
4	输出一个字符串，这个字符串要求以'\0' 结尾	$a0: 字符串的第一个字符的地址	N/A
5	读入一个整数	N/A	读入的整数保存在 $v0
8	读入一个字符串	$a0: 存放读入字符串的缓冲区 $a1: 读入最长的长度 +1 [19]	N/A
9	分配堆内存	$a0: 需要申请的连续的内存长度	$v0: 申请结果，即内存片段的首地址
10	结束运行	N/A	N/A
17	结束运行（有返回值）[20]	$a0: 程序运行结束的返回值	N/A

2.1.3 细节要求

2.1.4 5 级流水划分

Instruction Fetch 取指令阶段，按照 PC 寄存器的值去指定的内存位置读取指令并且加载到寄存器中。

Instruction Decode & Data Preparation 对 IF（取指令）阶段得到的指令进行解码，并且准备计算所需要的数据

Execution 执行完成这个条指令必要的运算，对于算术和逻辑运算，那么计算出运算结果，对于内存操作，那么计算出实际需要 load/store 的内存地址[21]，对于分支指令，可以不用执行运算，但是实际上的 MIPS 的 CPU 是通过当前的 PC 寄存器的值 + 一个跳转指令的偏移量计算出最终的跳转位置。

Memory Access 访问内存，完成 load/store 操作。

Write Back 回写，把算术，或者逻辑运算的结果写到指定寄存器，把从内存中 load 到的数据写入指定的寄存器。

2.1.5 寄存器读写顺序

由于流水线的存在，所以可能在同一个 CPU 时钟周期里面既要读寄存器又要写寄存器，先读还是先写会影响结果的正确性，所以我们规定，同一个 CPU 时钟周期里面先写寄存器，再读

[19]如果要读入'abc'，那么 $a1 应该设置成 4 或者更大

[20]C 语言中的 exit() 函数

[21]实际内存地址通过首地址 + 偏移量的形式给出，所以需要进行一次加法操作

寄存器[22]

2.1.6 对于 Hazard 的处理

流水线在处理的过程中由于各种原因会导致流水线不能像期望的那样正常工作，下面有三大类 Hazard[23]：

Structure Hazard 由于内存是总线上的设备，而总线的特性是同一时间[24]

[22]如果把一个周期分成两个半周期，那么可以简单的认为，前半周期写，后半周期读

[23]在 CAAQA 的中文版中，Hazard 被翻译成冒险，但是几乎所有的人都认为这个翻译并不是那么合理，所以我在这里就直接使用英文而不是用中文翻译

[24]1 个时钟周期