

# lab1-2113087 李秉睿

在本次实验中，我们完成了 32 位 MIPS 常见指令集的模拟执行过程。

## 相关链接

本次实验源码详见链接[lab1](#)

## 实验流程

### 构建模拟环境并转化机器码

实验中，我们需要在linux环境下利用spim模拟器转化机器码，这个过程中需要修改asm2hex文件，使得能正常调用Qtspim模拟器，经一番周折与配置，我们成功获取了给出的所有input文件的机器码并保存为.x文件。同时，有一些汇编指令出现了溢出，spim帮我们做了效果相同的其他指令的替代。(下图的69行)

```

User Text Segment [00400000]..[00440000]
[00400000] 2402000a addiu $2, $0, 10 ; 7: addiu $v0, $zero, 0xa
[00400004] 24030001 addiu $3, $0, 1 ; 10: addiu $3, $zero, 1
[00400008] 2404ffff addiu $4, $0, -1 ; 11: addiu $4, $zero, -1
[0040000c] 24051234 addiu $5, $0, 4660 ; 14: addiu $5, $zero, 0x1234
[00400010] 08100007 j 0x0040001c [l_1] ; 17: j l_1
[00400014] 00bf2821 addu $5, $5, $31 ; 19: addu $5, $5, $ra
[00400018] 10000004 beq $0, $0, 16 [l_2-0x00400018]; 20: beq $zero, $zero, l_2
[0040001c] 24a50007 addiu $5, $5, 7 ; 22: addiu $5, $5, 7
[00400020] 0c100005 jal 0x00400014 [l_0] ; 23: jal l_0
[00400024] 08100016 j 0x00400058 [l_8] ; 24: j l_8
[00400028] 24a50009 addiu $5, $5, 9 ; 26: addiu $5, $5, 9
[0040002c] 14640003 bne $3, $4, 12 [l_4-0x0040002c]; 27: bne $3, $4, l_4
[00400030] 24a50005 addiu $5, $5, 5 ; 30: addiu $5, $5, 5
[00400034] 04010005 bgez $0 20 [l_6-0x00400034]; 31: bgez $zero, l_6
[00400038] 24a5000b addiu $5, $5, 11 ; 34: addiu $5, $5, 11
[0040003c] 1860ffff blez $3 -12 [l_3-0x0040003c]; 35: blez $3, l_3
[00400040] 24a50063 addiu $5, $5, 99 ; 38: addiu $5, $5, 99
[00400044] 1c60ffff bgtz $3 -20 [l_3-0x00400044]; 39: bgtz $3, l_3
[00400048] 24a5006f addiu $5, $5, 111 ; 42: addiu $5, $5, 111
[0040004c] 03e00008 jr $31 ; 43: jr $ra
[00400050] 24a500c8 addiu $5, $5, 200 ; 47: addiu $5, $5, 200
[00400054] 0000000c syscall ; 49: syscall
[00400058] 24a500d7 addiu $5, $5, 215 ; 51: addiu $5, $5, 215
[0040005c] 0c10001a jal 0x00400068 [l_10] ; 52: jal l_10
[00400060] 24a50001 addiu $5, $5, 1 ; 55: addiu $5, $5, 1
[00400064] 0000000c syscall ; 56: syscall
[00400068] 00a62821 addu $5, $5, $6 ; 58: addu $5, $5, $6
[0040006c] 04900003 bltzal $4 12 [l_12-0x0040006c]; 59: bltzal $4, l_12
[00400070] 24a50190 addiu $5, $5, 400 ; 62: addiu $5, $5, 400
[00400074] 0000000c syscall ; 63: syscall
[00400078] 00a62821 addu $5, $5, $6 ; 65: addu $5, $5, $6
[0040007c] 0491ffff bgezal $4 -12 [l_11-0x0040007c]; 66: bgezal $4, l_11
[00400080] 3c01beb0 lui $1, -16720 ; 69: addiu $5, $5, 0xbeb0063d
[00400084] 3421063d ori $1, $1, 1597
[00400088] 00a12821 addu $5, $5, $1
[0040008c] 0000000c syscall ; 70: syscall

```

## 模拟实验过程

实验的shell.c文件构建了一个终端用于读入并执行控制台输入的一系列命令，包括，而sim.c实则完成了其中的一个函数process\_instruction()用于处理指令。

对作用过程的更深入理解，我们需要阅读shell.c和shell.h文件。

模拟过程首先执行main函数，其中在go()和run()两个函数中调用了cycle()，模拟了指令的执行流。

程序定义了一个CPU状态的结构体，内部存储了pc、32个寄存器和用于乘除法的HI和LO两个特殊寄存器：

```
typedef struct CPU_State_Struct {
```

```
uint32_t PC;           /* program counter */
uint32_t REGS[MIPS_REGS]; /* register file. */
uint32_t HI, LO;       /* special regs for mult/div. */
} CPU_State;
```

以及两个CPU状态CURRENT\_STATE, NEXT\_STATE。

在cycle()函数中可以看出，程序先执行指令处理函数process\_instruction()，然后将下一CPU状态赋值给当前CPU状态，指令计数器+1，完成了一个周期的工作。

这里需要注意的是，reg寄存器里面的类型是uint32\_t为无符号数，在处理指令过程中需要区分有无符号，即进行有符号的比较时需要进行强制类型转换。

因此接下来我们的工作就是补全指令处理函数。

## 指令处理函数的编写

指令执行过程中，先调用mem\_read\_32()获取指令对应的机器码。

再对机器码执行处理，先把一条32位指令按指令类型分解成多个指令字段，再按照指令的类型进行分类，进行不同case下的执行。

本次实验中，我们按字段大小排序完成了各个指令的编写。下面结合几个指令说明需要注意的点和遇到的问题。

- mem\_read\_32() 注意参数地址是32位，而立即数加载的数是16位。
- addiu 不检测溢出（u不代表无符号数，而是不检测溢出）由于加的立即数过大，超出了16位有符号数的表示范围，qtspim会帮我们将这些导致溢出的addiu指令转换为ori指令，同时保证其运算结果在不检验溢出的情况下的正确性。**同时需要采用符号扩展！**
- syscall 在本次实验中，只需要实现值为0xA时，调整RUN\_BIT=0结束模拟的过程。
- lbu 从指定位置内存中加载一个字节（8位），并将其零扩展为32位，然后存储在寄存器rt中，注意地址是基地址Base+偏移量Offset。这里采用位运算将取出的32位按位与上0xff即获得了我们需要的字节。其他的加载操作同理。
- sb 将寄存器rt中的最低字节存储到内存中的指定地址。与读取指令同理，但这里我们需要把原地地址数据的其他不需要存储的位保留，因此需要读取原地地址数据，按位与获取其余不需要存储的位，和需要存储的值按位或拼接起来，存回原来的位置。
- 分支和跳转指令。因为处理器按照字节寻址，二指令存储器每个地址是一个32bit字，所以要给指令中的立即数乘4，即左移两位。在判断完满足条件跳转后，带AL的指令还需要将下一条指令的地址存储到31号寄存器。

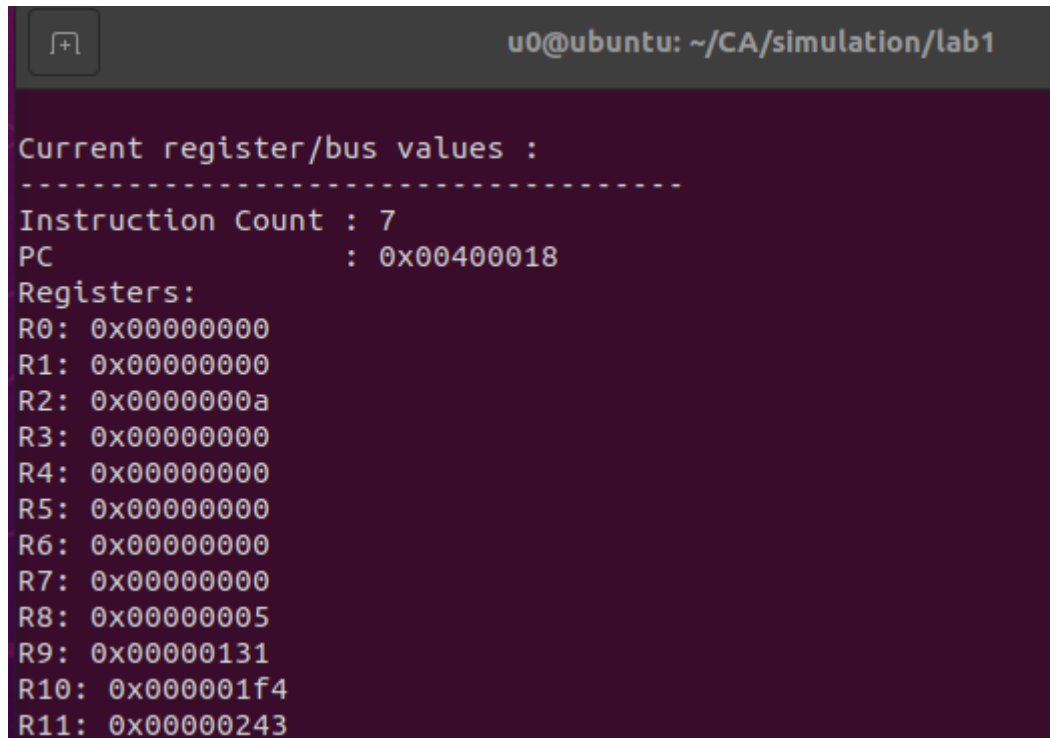
除了分支和跳转指令之外，其他指令执行完后PC值都会增加4，我们封装了一个函数pc\_add()来执行这条指令。

# 仿真验证

我们可以通过在控制台输入rdump返回所有储存的寄存器的内容，mdump返回内存中的内容，以此验证是否对应的机器码都可以成功执行。

首先是算术语句的两个样例。

- 模拟addiu的结果，可以看出都正常执行了addiu和syscall语句。



```
u0@ubuntu: ~/CA/simulation/lab1

Current register/bus values :
-----
Instruction Count : 7
PC                  : 0x00400018
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000000
R6: 0x00000000
R7: 0x00000000
R8: 0x00000005
R9: 0x00000131
R10: 0x000001f4
R11: 0x00000243
```

- arithtest的结果，依照汇编语句的运算顺序和结果正常执行了，其中的一些运算如上文Qtspim进行了等价代替。

```
u0@ubuntu: ~/CA/simulation/lab1
jr
op=2,rs=0,rt=10,rd=0,sa=0,funct=16,imm=22,
j
current_pc=0x 400024
next_pc=0x 400058
op=9,rs=5,rt=5,rd=0,sa=3,funct=17,imm=215,
addiu
op=3,rs=0,rt=10,rd=0,sa=0,funct=1a,imm=26,
jal
current_pc=0x 40005c
next_pc=0x 400068
op=0,rs=5,rt=6,rd=5,sa=0,funct=21,imm=10273,
adduop=1,rs=4,rt=10,rd=0,sa=0,funct=3,imm=3,
bltzal
op=0,rs=5,rt=6,rd=5,sa=0,funct=21,imm=10273,
adduop=1,rs=4,rt=11,rd=1f,sa=1f,funct=3d,imm=65533,
bgezal
op=f,rs=0,rt=1,rd=17,sa=1a,funct=30,imm=48816,
op=d,rs=1,rt=1,rd=0,sa=18,funct=3d,imm=1597,
op=0,rs=5,rt=1,rd=5,sa=0,funct=21,imm=10273,
adduop=0,rs=0,rt=0,rd=0,sa=0,funct=c,imm=12,
Simulator halted

MIPS-SIM>
```

然后是分支测试。我们以第一个样例brtest0为例进行解释。

- 如图，按照语句的语义执行的结果，指令流应该是进行了如图的跳转，在bne处判断条件不成功，因此不做跳转。

```

1      # Basic branch test
2      .text
3
4  main:
5      addiu $v0, $zero, 0xa
6  l_0:
7      addiu $5, $zero, 1
8      j l_1
9      addiu $10, $10, 0xf00
10     ori $0, $0, 0
11     ori $0, $0, 0
12     addiu $5, $zero, 100
13     syscall
14 → l_1:
15 ✕ bne $zero, $zero, l_3
16     ori $0, $0, 0
17     ori $0, $0, 0
18     addiu $6, $zero, 0x1337
19 l_2:
20     beq $zero, $zero, l_4
21     ori $0, $0, 0
22     ori $0, $0, 0
23     # Should not reach here
24     addiu $7, $zero, 0x347
25     syscall
26 l_3:
27     # Should not reach here
28     addiu $8, $zero, 0x347
29     syscall
30 → l_4:
31     addiu $7, $zero, 0xd00d
32     syscall
33
34

```

- 执行的结果，正常执行了对应的语句并在寄存器R7中存储了正确的值0xd00d。



```
u0@ubuntu: ~/CA/simulation/lab1
Simulating...
Simulator halted
MIPS-SIM> rdump

Current register/bus values :
-----
Instruction Count : 10
PC                : 0x00400050
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000001
R6: 0x00001337
R7: 0x0000d00d
R8: 0x00000000
R9: 0x00000000
R10: 0x00000000
R11: 0x00000000
```

执行的结果，正常执行了对应的语句并在寄存器中存储了正确的值。

brtest1和brtest2如下。

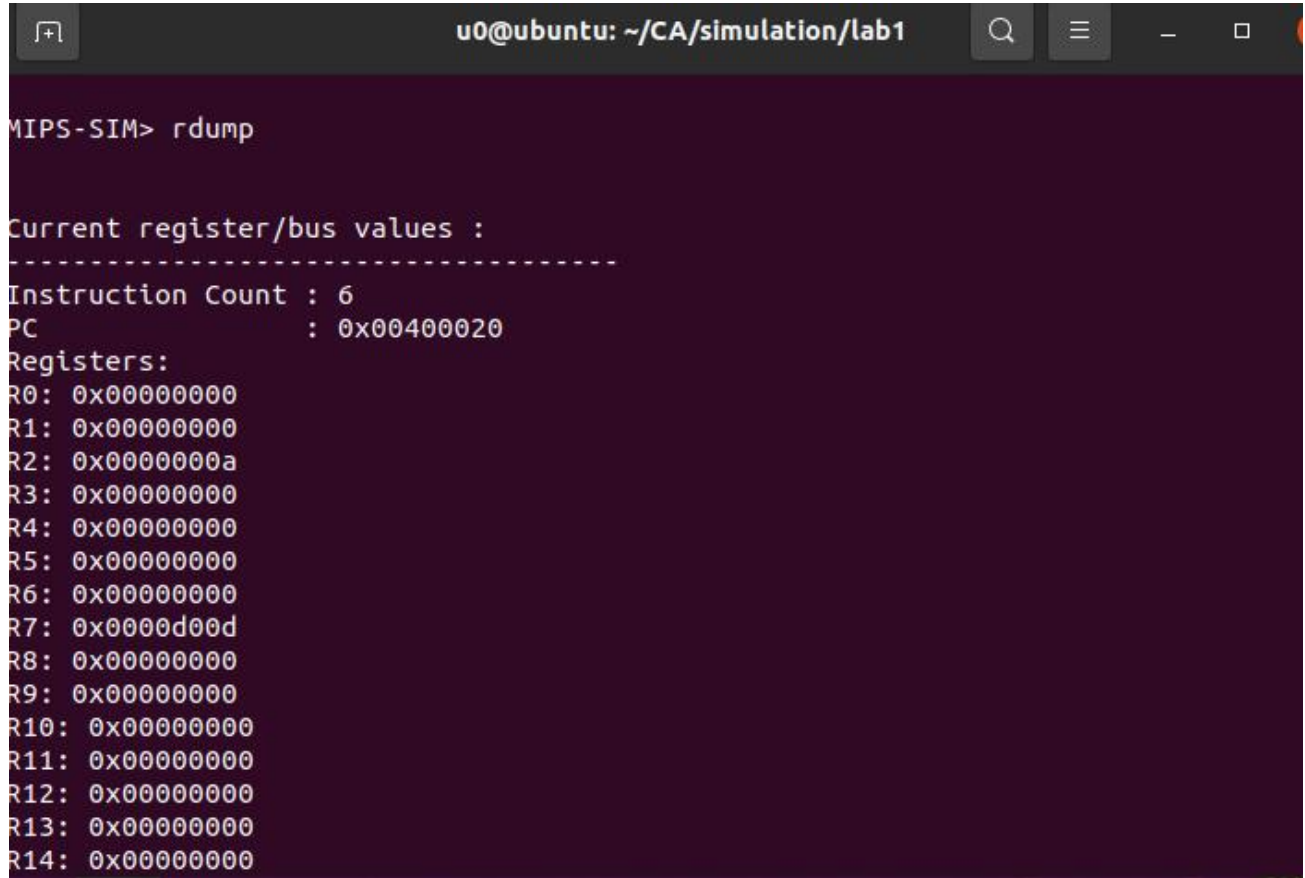
- brtest1：在R5中存储了对应的值0xbef01a5e，过程中也正确进行了每一步的运算，保存了合理的ra值，说明成功正确运行到了正确的结尾。

```
u0@ubuntu: ~/CA/simulation/lab1
Instruction Count : 30
PC                : 0x0040008c
Registers:
R0: 0x00000000
R1: 0xbef0063d
R2: 0x0000000a
R3: 0x00000001
R4: 0xffffffff
R5: 0xbef01a5e
R6: 0x00000000
R7: 0x00000000
R8: 0x00000000
```

```
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00400070
HI: 0x00000000
LO: 0x00000000

MIPS-SIM> 
```

- brtest2: 执行成功时在R7中存储了0xd00d。

A terminal window titled 'u0@ubuntu: ~/CA/simulation/lab1' showing the output of the 'rdump' command in MIPS-SIM. The output displays current register and bus values, instruction count, PC, and a list of registers R0 through R14. Register R7 is highlighted in green and contains the value 0xd00d.

```
u0@ubuntu: ~/CA/simulation/lab1
MIPS-SIM> rdump

Current register/bus values :
-----
Instruction Count : 6
PC                  : 0x00400020
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000000
R6: 0x00000000
R7: 0x0000d00d
R8: 0x00000000
R9: 0x00000000
R10: 0x00000000
R11: 0x00000000
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
```

然后是访存的两个样例。

- 第一个样例test0的执行结果如下，R17存储的值为0x881d。



```
u0@ubuntu: ~/CA/simulation/lab1
Instruction Count : 32
PC : 0x0040007c
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x10000004
R4: 0x00000000
R5: 0x000000ff
R6: 0x000001fe
R7: 0x000003fc
R8: 0x0000792c
R9: 0x000000ff
R10: 0x000001fe
R11: 0x000003fc
R12: 0x0000792c
R13: 0x000000ff
R14: 0x000000ff
R15: 0x000001fe
R16: 0x000003fc
R17: 0x0000881d
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
```

- test1主要是取半字和取字节，最终在寄存器中都存储了相应的值进行对应，正确运行了结果。

```
u0@ubuntu: ~/CA/simulation/lab1
Instruction Count : 32
PC : 0x0040007c
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x10000004
R4: 0x00000000
R5: 0x0000cafe
R6: 0x0000feca
R7: 0x0000beef
R8: 0x0000efbe
R9: 0x000000fe
R10: 0x000000ca
R11: 0xffffffff
R12: 0xffffffff
R13: 0x0000cafe
R14: 0x0000feca
R15: 0x000000ef
R16: 0x000000be
R17: 0x0001ccea
R18: 0x00000000
R19: 0x00000000
```

## 小结

本次实验我们在一个shell框架下，完成了对应的mips指令集的简单复现，同时结合了测试用例进行了简单的逻辑测试，不仅锻炼了阅读一个新框架和项目的能力，也学习了更多的mips指令集的实现细节，收获颇丰。