MIPS模拟机

一、需求描述

以程序模拟MIPS运行，功能包括：

汇编器：输入汇编指令，转换成机器码模拟执行。

读入汇编好的机器码(二进制)，显示对应的汇编指令(反汇编)，并模拟执行。

1、模拟器运行界面设计：可以命令行或窗口界面。列表显示32个寄存器。（命令行版可参考DEBUG）

2、可执行多条指令。可观察寄存器、内存的变化。（命令行版可参考DEBUG）

DEBUG命令：

-->R-看寄存器，

-->D-数据方式看内存，

-->U-指令方式看内存，

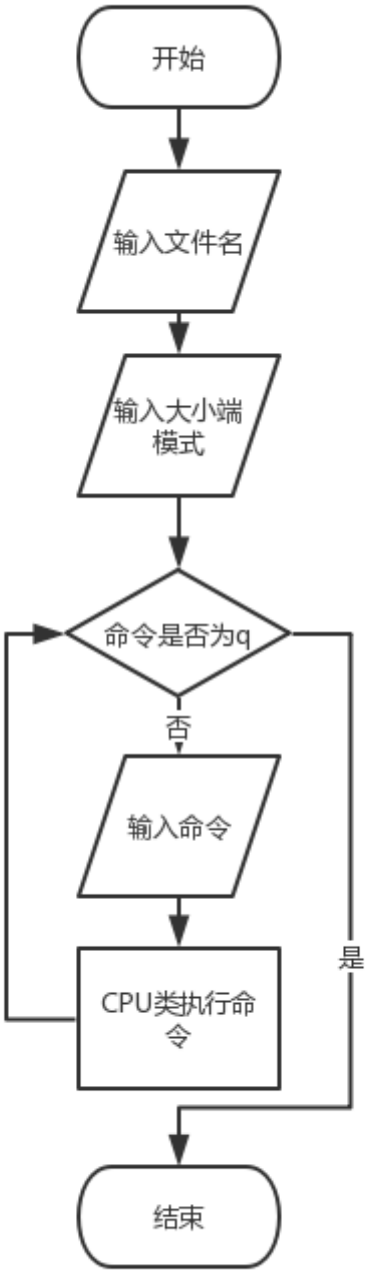
-->A-写汇编指令到内存，

-->T-单步执行内存中的指令

二、实现功能

1. 实现了共十条指令，分别是add, sub, and, or, addi, beq, bne, lw, sw, j.
2. 高仿debug界面，支持六条命令
    1. R/r-看寄存器
    2. D/d-数据方式看内存
    3. U/u-指令方式看内存
    4. A/a-写一条/多条指令到当前IP对应的内存中，直到输入空字符串
    5. T/t-单步执行内存中的指令,执行后输出寄存器中的值
    6. G/g-运行完整个程序，运行完后输出寄存器中的值
    7. Q/q-退出模拟器
3. 支持标号，可以使用灵活的指令格式与标号形式编程
4. 支持大头小头、支持对齐不对齐读写
5. lw, sw写入的内存与指令是同一个存储器，所以支持动态修改指令

三、程序流程图



四、实现思路

共设计有三个类，分别是CPU类，存储器单元MemoryUnit类，汇编反汇编器Assembler类。

4.1 CPU

CPU类是顶层类，为方便，并没有另外写一个模拟器类，而是让CPU类包含了MemoryUnit对象和Assembler对象。

CPU类的功能是通过调用MemoryUnit对象和Assembler对象完成模拟器的核心功能。

CPU类有七个公有方法，分别对应着模拟器的七个不同的功能， 私有成员有寄存器组、IP和汇编器与存储器。

4.1.1 构造函数

构造函数接收一个string类型的文件名，也就是要加载的程序，构造函数通过调用MemoryUnit类的sw方法，将程序加载入存储器，并计算出程序的长度以方便直接运行程序

伪代码：

```
CPU::CPU(string filename)
{
  openfile
  IP = 0
  while(file is not eof)
    code = file.read()
    MemoryUnit.sw(code)
    IP += 4
  maxIP = IP
  IP = 0
}
```

4.1.2 exec 单步执行函数

exec函数是单步执行函数，效果是从IP指向的存储器单元中读取指令并使得IP加4，之后解析指令完成对应的操作。

伪代码：

```
CPU::exec()
{
  code = MemoryUnit.lw(IP)
  IP += 4
  执行code对应的指令
}
```

4.1.3 showRegister 输出寄存器值函数

showRegister函数用于按照格式打印各个寄存器的值，用于模拟器的R/r指令。

伪代码：

```
void CPU::showRegister()
{
  for(each register in registerVec)
    print(register);
}
```

4.1.4 showData 数据方式看内存

showData函数用于按照debug格式打印当前IP后固定字节的数据情况，分别打印地址信息、数据信息、对应的字符信息。

伪代码:

```
void CPU::showData()
{
  for(int i = 0; i < NUMBER; i++)
    byte code = MemoryUnit.lw(IP + i);
    print(IP+i, code, char(code));
}
```

4.1.5 showInstruction 指令方式看内存

showInstruction函数用于按照debug格式打印当前IP后共八条指令，分别打印地址信息、数据信息、对应反汇编后的指令信息。

伪代码：

```
void CPU::showInstruction()
{
  for(int i = 0; i < 8; i++)
  {
    unsigned int code = MemoryUnit.lw(IP + i * 4);
    string instruction = Assembler.disassembler(code);
    print(IP + i * 4, code, instruction);
  }
}
```

4.1.6 modifyInstruction 修改当前IP对应的一条或多条指令

modifyIntruction函数用于修改当前IP对应的一条或多条指令，直到输入的指令为空为止。

伪代码：

```
void CPU::modifyInstruction()
{
  int i = 0;
  while(input >> instruction) {
    unsigned int code = Assembler.compile(instruction);
    MemoryUnit.sw(IP + i * 4, code);
    i++;
  }
}
```

4.2 MemoryUnit

MemoryUnit类是存储器单元，有一个数组私有成员，作为内存，还有一个布尔类型变量用于控制大端\小段规则。

四个公有成员函数用于操作内存。

4.2.1 lw 取出一个字

lw函数用于取出存储器中指定地址的一个字，参数为地址，返回值为字的值

代码:

```
unsigned int lw(size_t address) const
{
  unsigned int result = 0;
  // 大端规则，高地址存高位
  if(endian) {
    // i1: 高四位， i4: 低四位
    unsigned int i1, i2, i3, i4;
    i1 = memory[address + 3];
    i2 = memory[address + 2];
    i3 = memory[address + 1];
    i4 = memory[address];
```

```
    result |= (i1 << 24);
    result |= (i2 << 16);
    result |= (i3 << 8);
    result |= i4;
  }
  // 小端规则，高地址存低位
  else {
    unsigned int i1, i2, i3, i4;
    i1 = memory[address];
    i2 = memory[address + 1];
    i3 = memory[address + 2];
    i4 = memory[address + 3];
    result |= (i1 << 24);
    result |= (i2 << 16);
    result |= (i3 << 8);
    result |= i4;
  }
  return result;
}
```

4.2.2 sw 存储一个字

sw函数的作用是将一个字写入指定的内存单元，参数为地址和要写入的内容

代码：

```
void sw(size_t address, unsigned int number) {
  // 大端规则，高地址存高位
  if(endian) {
    memory[address] = (number & 0x000000FF);
    memory[address + 1] = ((number & 0x0000FF00) >> 8);
    memory[address + 2] = ((number & 0x00FF0000) >> 16);
    memory[address + 3] = ((number & 0xFF000000) >> 24);
  }
  // 小端规则，高地址存低位
  else {
    memory[address + 3] = (number & 0x000000FF);
    memory[address + 2] = ((number & 0x0000FF00) >> 8);
    memory[address + 1] = ((number & 0x00FF0000) >> 16);
    memory[address + 0] = ((number & 0xFF000000) >> 24);
  }
}
```

4.2.3 lb

lb函数用于取出存储器中指定地址的一个字节，参数为地址，返回值为该字节的值

4.2.4 sb

sw函数的作用是将一个字节写入指定的内存单元，参数为地址和要写入的内容

4.3 Assembler

Assembler类是汇编反汇编器类，用于汇编反汇编指令/文件

4.3.1 Compile 编译

Compile函数有两个重载，一个接收两个字符串，一个参数是输入文件，另一个参数是输出文件，这个重载用于编译文件.

另一个重载函数用于编译单条语句，参数是指令字符串，返回值是对应的机器码。

伪代码：

```
void Compile(string fileName, string outputName)
{
  file = open(fileName);
  // 源代码规格化，提取Label
  Formation(file);
  // 将Label替换为地址
  Substitute(file);
  // 将代码翻译为机器码
  Translate(outputName);
}
```

4.3.2 Disassembler 反编译

Disassembler函数同样有两个重载，一个接收两个字符串，一个参数是输入文件，另一个参数是输出文件，这个重载用于反汇编文件.

另一个重载函数用于反汇编单条语句，参数是机器码，返回值是对应的指令字符串。

伪代码：

```
string Disassembler(unsigned int code)
{
  opCode = code & 0xFA000000;
  switch(opCode)
  {
    case R:
      return processR(code);
      break;
    ...
  }
}
```

五、结果展示

5.1 测试准备

需要测试的指令共有十条，add, sub, and, or, addi, lw, sw, beq, bne, j.

需要测试的命令有六条，u, t, d, r, m, g

首先准备mips源文件，需要包括以上全部十条指令并能运行。此外，还要测试汇编器对于不规整的格式的处理和对标号的处理。

```
and    $s0,   $s0,      $zero
addi $t1   , $zero  , 3
and $t0, $t0, $zero
loop:
   add $s0    , $t0, $s0
```

```
   addi $t0, $t0, 1
   bne $t0, $t1 ,     loop
and $t2, $t2, $zero
addi $t2   , $zero   , 3
beq $s0, $t2, Corret
Incorret: addi $a0, $zero, 1
   j end
Corret:
add $a0, $zero, $s0
j end
add $t8, $s0, $s1
end:
addi $s2, $zero, 16708
 sub $s2, $s2, $a0
sw $s2, 96($zero)
lw $s1, 96  ($zero)
or $t5, $s2, $s0
```

以上mips翻译为C语言如下

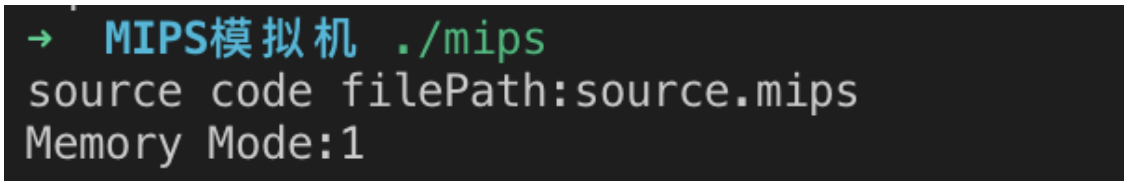
```
$s0 = 0;
$t1 = 3;
// $s0 = 0 + 1 + 2
for($t0 = 0; $t0 != $t1; $t0++) {
    $s0 = $s0 + $t0;
}
$t2 = 3;
// 检测计算结果是否正确，正确的话$a0保存计算结果
// 不正确$a0 = 1
if($s0 == $t2)
    $a0 = $s0;
else{
    $a0 = 1;
}
// 测试其他指令
// 可以在内存中看到两个A
$s2 = 0x4144 - 3;
Memory[70] = $s2;
$s1 = Memory[70];
$t5 = $s2 | $s0;
```

这个测试源文件覆盖了所有十条指令，并且写的极其不规整并使用了标号。

此外，为了验证D指令，特意写了两个A到内存中。

5.2 测试结果

1. 开启程序，输入源文件名和大小端模式，0是大端模式，其他是小端模式



   我们选择小端模式(小端模式实现要比大端复杂一点点点点)

2. 测试u指令

```
-u
0000:0000  02008024  and   $s0, $s0, $zero
0000:0004  20090003  addi  $t1, $zero, 3
0000:0008  01004024  and   $t0, $t0, $zero
0000:000C  01108020  add   $s0, $t0, $s0
0000:0010  21080001  addi  $t0, $t0, 1
0000:0014  15280003  bne   $t0, $t1, 3
0000:0018  01405024  and   $t2, $t2, $zero
0000:001C  200A0003  addi  $t2, $zero, 3
0000:0020  1150000B  beq   $s0, $t2, 11
0000:0024  20040001  addi  $a0, $zero, 1
0000:0028  0800000E  j     14
0000:002C  00102020  add   $a0, $zero, $s0
```

我们发现u指令的输出格式规整，反汇编完美，达成了u指令的目的。

3. t 命令单步运行

```
-t
$zero = 00000000   $at = 00000000   $v0 = 00000000   $v1 = 00000000   $a0 = 00000000   $a1 = 00000000   $a2 = 00000000   $a3 = 00000000
  $t0 = 00000000   $t1 = 00000000   $t2 = 00000000   $t3 = 00000000   $t4 = 00000000   $t5 = 00000000   $t6 = 00000000   $t7 = 00000000
  $s0 = 00000000   $s1 = 00000000   $s2 = 00000000   $s3 = 00000000   $s4 = 00000000   $s5 = 00000000   $s6 = 00000000   $s7 = 00000000
  $t8 = 00000000   $t9 = 00000000   $k0 = 00000000   $k1 = 00000000   $gp = 00000000   $sp = 00000000   $fp = 00000000   $ra = 00000000
-t
$zero = 00000000   $at = 00000000   $v0 = 00000000   $v1 = 00000000   $a0 = 00000000   $a1 = 00000000   $a2 = 00000000   $a3 = 00000000
  $t0 = 00000003   $t1 = 00000003   $t2 = 00000000   $t3 = 00000000   $t4 = 00000000   $t5 = 00000000   $t6 = 00000000   $t7 = 00000000
  $s0 = 00000000   $s1 = 00000000   $s2 = 00000000   $s3 = 00000000   $s4 = 00000000   $s5 = 00000000   $s6 = 00000000   $s7 = 00000000
  $t8 = 00000000   $t9 = 00000000   $k0 = 00000000   $k1 = 00000000   $gp = 00000000   $sp = 00000000   $fp = 00000000   $ra = 00000000
-t
$zero = 00000000   $at = 00000000   $v0 = 00000000   $v1 = 00000000   $a0 = 00000000   $a1 = 00000000   $a2 = 00000000   $a3 = 00000000
  $t0 = 00000003   $t1 = 00000003   $t2 = 00000000   $t3 = 00000000   $t4 = 00000000   $t5 = 00000000   $t6 = 00000000   $t7 = 00000000
  $s0 = 00000000   $s1 = 00000000   $s2 = 00000000   $s3 = 00000000   $s4 = 00000000   $s5 = 00000000   $s6 = 00000000   $s7 = 00000000
  $t8 = 00000000   $t9 = 00000000   $k0 = 00000000   $k1 = 00000000   $gp = 00000000   $sp = 00000000   $fp = 00000000   $ra = 00000000
```

t 指令也高仿debug，每运行一步，输出运行后的寄存器值。

前三条指令为

```
and    $s0,    $s0,       $zero
addi $t1    , $zero   , 3
and $t0, $t0, $zero
```

所以第二步运行后寄存器$t1的值应该为3，与程序运行相符，t指令完美运行。

4. udr 命令观察目前状态

```
-u
0000:000C  01108020  add   $s0, $t0, $s0
0000:0010  21080001  addi  $t0, $t0, 1
0000:0014  15280003  bne   $t0, $t1, 3
0000:0018  01405024  and   $t2, $t2, $zero
0000:001C  200A0003  addi  $t2, $zero, 3
0000:0020  1150000B  beq   $s0, $t2, 11
0000:0024  20040001  addi  $a0, $zero, 1
0000:0028  0800000E  j     14
0000:002C  00102020  add   $a0, $zero, $s0
0000:0030  0800000E  j     14
0000:0034  0211C020  add   $t8, $s0, $s1
0000:0038  20124144  addi  $s2, $zero, 16708
-d
0000:000C  20 80 10 01 01 00 08 21-03 00 28 15 24 50 40 01    . . . . . ! . . ( . $ P @ .
0000:001C  03 00 0A 20 0B 00 50 11-01 00 04 20 0E 00 00 08   . . .  . . P . . . .  . . . .
0000:002C  20 20 10 00 0E 00 00 08-20 C0 11 02 44 41 12 20    . . . . . . . . . . D A .
0000:003C  22 90 44 02 60 00 12 AC-60 00 11 8C 25 68 50 02   " . D . ` . . . ` . . . % h P .
0000:004C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:005C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:006C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:007C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
-r
$zero = 00000000   $at = 00000000   $v0 = 00000000   $v1 = 00000000   $a0 = 00000000   $a1 = 00000000   $a2 = 00000000   $a3 = 00000000
  $t0 = 00000003   $t1 = 00000003   $t2 = 00000000   $t3 = 00000000   $t4 = 00000000   $t5 = 00000000   $t6 = 00000000   $t7 = 00000000
  $s0 = 00000000   $s1 = 00000000   $s2 = 00000000   $s3 = 00000000   $s4 = 00000000   $s5 = 00000000   $s6 = 00000000   $s7 = 00000000
  $t8 = 00000000   $t9 = 00000000   $k0 = 00000000   $k1 = 00000000   $gp = 00000000   $sp = 00000000   $fp = 00000000   $ra = 00000000
```

可以看到程序运行到了第三条指令，通过u指令可以看到接下来的指令，d指令可以看到字符的显示。

5. g 命令直接运行到程序结束

```
-g
$zero = 00000000   $at = 00000000   $v0 = 00000000   $v1 = 00000000   $a0 = 00000003   $a1 = 00000000   $a2 = 00000000   $a3 = 00000000
  $t0 = 00000003   $t1 = 00000003   $t2 = 00000003   $t3 = 00000000   $t4 = 00000000   $t5 = 00004143   $t6 = 00000000   $t7 = 00000000
  $s0 = 00000003   $s1 = 00004141   $s2 = 00004141   $s3 = 00000000   $s4 = 00000000   $s5 = 00000000   $s6 = 00000000   $s7 = 00000000
  $t8 = 00000000   $t9 = 00000000   $k0 = 00000000   $k1 = 00000000   $gp = 00000000   $sp = 00000000   $fp = 00000000   $ra = 00000000
```

g 命令会直接将程序运行到结束，在运行结束后，会打印程序结束后的寄存器情况。

我们的程序的功能是计算0 + 1 + 2的值保存在$a0中，然后将内存中地址为96的字写为0x004141，也就是两个A。

上面寄存器的结果如同我们所料，接下来查看内存的结果。

6. d 命令数据形式查看内存

```
-d
0000:004C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:005C  00 00 00 00 41 41 00 00-00 00 00 00 00 00 00 00   . . . . A A . . . . . . . . . .
0000:006C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:007C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:008C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:009C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:00AC  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:00BC  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
-u
0000:004C  00000000  nop
0000:0050  00000000  nop
0000:0054  00000000  nop
0000:0058  00000000  nop
0000:005C  00000000  nop
0000:0060  00004141  nop
0000:0064  00000000  nop
0000:0068  00000000  nop
0000:006C  00000000  nop
0000:0070  00000000  nop
0000:0074  00000000  nop
0000:0078  00000000  nop
```

我们看到两个A如约而至，此外，由于我们存储器使用的是小端规则，所以低字节存高地址，可见两个A位于高地址上。

此外，由于程序已经通过g命令运行完毕，剩下的内存中的数据均为0，也就是nop

7. a命令修改内存指令

通过a命令可以修改一条或多条内存中当前IP对应下的指令。

下面我们写入一个sw指令来使地址为100的内存变为两个A

```
-a
0000:0050  sw $s1, 100($zero)
0000:0054
-u
0000:0050  AC110064  sw    $s1, 100($zero)
0000:0054  00000000  nop
0000:0058  00000000  nop
0000:005C  00000000  nop
0000:0060  00004141  nop
0000:0064  00000000  nop
0000:0068  00000000  nop
0000:006C  00000000  nop
0000:0070  00000000  nop
0000:0074  00000000  nop
0000:0078  00000000  nop
0000:007C  00000000  nop
-t
$zero = 00000000   $at = 00000000   $v0 = 00000000   $v1 = 00000000   $a0 = 00000003   $a1 = 00000000   $a2 = 00000000   $a3 = 00000000
  $t0 = 00000003   $t1 = 00000003   $t2 = 00000003   $t3 = 00000000   $t4 = 00000000   $t5 = 00004143   $t6 = 00000000   $t7 = 00000000
  $s0 = 00000003   $s1 = 00004141   $s2 = 00000000   $s3 = 00000000   $s4 = 00000000   $s5 = 00000000   $s6 = 00000000   $s7 = 00000000
  $t8 = 00000000   $t9 = 00000000   $k0 = 00000000   $k1 = 00000000   $gp = 00000000   $sp = 00000000   $fp = 00000000   $ra = 00000000
-d
0000:0054  00 00 00 00 00 00 00 00-00 00 00 00 41 41 00 00   . . . . . . . . . . . . A A . .
0000:0064  41 41 00 00 00 00 00 00-00 00 00 00 00 00 00 00   A A . . . . . . . . . . . . . .
0000:0074  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:0084  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:0094  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:00A4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:00B4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
0000:00C4  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
```

我们发现0x64(100)处的数据也变成了两个A.