MIPS模拟机

学号：3170103456 姓名：应承峻

1 实验描述

以程序模拟MIPS运行，功能包括：

汇编器：将汇编程序转换成机器码。能有较灵活的格式，可以处理格式指令、表达式、有出错信息。

汇编反汇编：MIPS汇编指令与机器码的相互转换。

模拟器：根据机器码模拟执行可以运行简单MIPS程序。

1. 模拟器运行界面设计：可以命令行或窗口界面。可以执行指令的汇编、反汇编，可以单步执行指令观察寄存器、内存的变化。（命令行版可参考DEBUG）
2. 指令伪指令的汇编反汇编：将MIPS指令转换成二进制机器码，能够处理标号、变量。
3. MMU存储器管理单元：存储器存取模拟。大头小头，对齐不对齐，Cache，虚拟存储。
4. 格式指令表达式处理：对于汇编程序中的格式指令、表达式的处理。参考网页格式指令。

个人版模拟器应实现：

```
指令：R、LW、SW、BEQ、J五条；
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
```

2 程序架构分析

2.1 `Mips`类（主类）

重要变量：

```
int PC = 0; //程序计数器
int line = 0; //指令计数器，记录当前指令条数
Register r = new Register();    //初始化寄存器
Memory m = new Memory();   //初始化内存
ArrayList<Instruction> is = new ArrayList<>();   //存放所有指令
```

使用方法：

```
-R: 查看寄存器
-D: 数据方式看内存
-U: 指令方式看内存
-A [INSTRUCTION]: 写汇编指令到内,例: -A -A addi $t0,$zero,10
-T: 从PC开始单步执行内存中的指令
-P: 查看下一步执行的指令
-F: 从文件中读入指令,例:-F in.txt
-I: 查看提示信息
```

2.2 `Register`类：寄存器

重要变量：

```java
private static String[] reg = {"zero", "at", "v0", "v1", "a0", "a1", "a2", "a3", "t0", "t1", "t2", "t3", "t4","t5", "t6", "t7", "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "t8", "t9", "k0", "k1", "gp", "sp", "fp","ra"};  //存放寄存器名称
private int[] regValue = new int[32]; //存放寄存器值
```

函数调用：

```java
Register();  //构造函数，将所有寄存器初始化为0
public static int RegAddress(String str);  //根据寄存器名称获得寄存器地址
public static String regName(int id);   //根据寄存器地址获取寄存器名称
public boolean setRegValue(int id, int val);  //指定寄存器设置值
public boolean setRegValue(String str, int val);  //指定寄存器设置值
public int getRegValue(int id); //根据寄存器地址得到寄存器值
```

2.3 `Memory`类：内存

重要变量：

```java
private final int memSize = 1024;  //内存大小为1KB
private byte[] data = new byte[memSize];  //模拟内存
public ArrayList<Integer> addrSet = new ArrayList<>(); //存放写过数据的内存的地址
```

函数调用：

```java
public boolean writeData(int offset, int val); //写数据到内存offset处的4个字节
public int readData(int offset); //从内存中offset处读取4个字节的数据
public int readByte(int pos);   //从内存中pos处读取1个字节的数据
public boolean writeInstruction(int val);  //写指令到内存
```

2.4 `Expr`类：表达式计算

函数调用：

```java
public static boolean isValid(String expr); //判断表达式是否合法
public static int calc(String expr); //计算表达式的值
```

2.5`Instrction`类：指令

重要变量：

```java
public static HashMap<String, Integer> labelSet = new HashMap<>();  //存放指令标签
private static OperationCode op = new OperationCode();  //获取指令功能码
public int bin = 0xFFFFFFFF;  //指令二进制值，默认-1
public int line;  //当前是第几条指令
public String label = null; // 指令标签
public String[] items;  //分割指令后指令的各个部分
```

函数调用：

```java
Instruction(String[] items, int line, String label); //构造函数
Instruction(String str, int line); //构造函数
public int compile(); //编译指令
public static boolean isPsuedo(String str);  //判断是否是伪指令
public static ArrayList<Instruction> psuedoConvert(String str, int startline);//将伪指令转换成普通指令的集合
public static String initStr(String str);   //字符串初始化处理，过滤转义字符和多空格
public static String[] splitWithoutLabels(String str);  //分割字符串，并除去标签
public String reverseCompile();  //反汇编
public static int execute(int binary, Memory mem, Register reg, int PC); //执行二进制码
public static String fetchLabel(String str);  //得到字符串指令的标签
public static String matchKeyByValue(HashMap<String, Integer> map, int num);//根据哈希表的值获取键值对名称
private int rCompile(); //R类型指令编译
private int iCompile(); //I类型指令编译
private int jCompile(); //J类型指令编译
```

当前支持汇编的指令：

```
===================R类型=======================
0   sll rd  rt  sa
2   srl rd  rt  sa
3   sra rd  rt  sa
4   sllv    rd  rt  rs
6   srlv    rd  rt  rs
7   srav    rd  rt  rs
8   jr  rs
9   jalr    rd  rs
12  syscall
13  break
16  mfhi    rd
17  mthi    rs
18  mflo    rd
19  mtlo    rs
24  mult    rs  rt
25  multu   rs  rt
26  div rs  rt
27  divu    rs  rt
32  add rd  rs  rt
33  addu    rd  rs  rt
34  sub rd  rs  rt
```

```
35  subu    rd  rs  rt
36  and rd  rs  rt
37  or  rd  rs  rt
38  xor rd  rs  rt
39  nor rd  rs  rt
43  sltu    rd  rs  rt
42  slt rd  rs  rt
===================I类型=======================
bgez    rs  label       1   rt=1
bltz    rs  label       1   rt=0
beq rs  rt  label   4
bne rs  rt  label   5
blez    rs  label       6   rt=0
bgtz    rs  label       7   rt=0
addi    rt  rs  ofs 8
addiu   rt  rs  ofs 9
slti    rt  rs  ofs 10
sltiu   rt  rs  ofs 11
andi    rt  rs  ofs 12
ori rt  rs  ofs 13
xori    rt  rs  ofs 14
lui rt  ofs     15
lb  rt  ofs(rs)     32
lh  rt  ofs(rs)     33
lw  rt  ofs(rs)     35
lbu rt  ofs(rs)     36
lhu rt  ofs(rs)     37
sb  rt  ofs(rs)     40
sh  rt  ofs(rs)     41
sw  rt  ofs(rs)     43
lwcl    rt  ofs(rs)     49
swcl    rt  ofs(rs)     57
===================J类型=======================
j label
jal label
===================伪指令======================
move
blt
bgt
ble
bge
```

当前支持执行的指令：

```
add
sub
beq
bne
j
lw
sw
```

3 部分算法分析

3.1 指令编译处理

字符串预处理

```java
//字符串预处理
public static String initStr(String str) {
    return str.toLowerCase().replace(",", " ").replace("\t", " ").replace("($", " ").replace("$", " ").trim();  //去除多余的空格、\t转义字符、并将,和$替换成普通空格
}
```

指令模块分割：

```java
Matcher matcher = Pattern.compile("^([a-z 0-9]+):").matcher(str); //寻找标签是否存在
if (matcher.find()) { //存在标签则将它从指令中删去，并做记录
    str = str.replaceFirst("^[a-z0-9]+:", "").trim();
    label = matcher.group(1);
    if (!labelSet.containsKey(label)) {  //存储标签
        labelSet.put(label.trim(), line * 4 + startAddr);
    } else throw new Exception("Label " + label + " has been already used");
}
items = str.split(" +");  //指令分割
if (items.length == 0) throw new Exception("invalid instructions"); //异常抛出
```

编译选择模块：

```java
String str = items[0];   //得到操作码，并根据操作码决定选择哪种指令编译方式
if (OperationCode.r.containsKey(str)) bin = rCompile();
else if (OperationCode.j.containsKey(str)) bin = jCompile();
else if (OperationCode.i.containsKey(str)) bin = iCompile();
else if (items[0].trim().equals("")) bin = 0xFFFFFFFF;
else throw new Exception("Instruction does not exist!");
```

R指令编译样例：

```java
int rs = 0, rt = 0, rd = 0, sa = 0, func = 0;
switch (items[0]) {
    case "add":  //$rd,$rs,$rt
        if (items.length < 4) //异常抛出
            throw new Exception("Lack of Registers or Args!");
        else if (items.length > 4)
            throw new Exception("Too Many Registers or Args!");
        rd = Register.RegAddress(items[1]);
        rs = Register.RegAddress(items[2]);
        rt = Register.RegAddress(items[3]);
    break;
}
func = op.RFuncCode(items[0]);  //得到func码的值
return (rs << 21) | (rt << 16) | (rd << 11) | (sa << 6) | func;
```

I指令编译样例：

```java
int opcode = op.IFuncCode(items[0]);
int ofs = 0, rs = 0, rt = 0;
switch (items[0]) {
    case "addi":  //command $rt,rs,ofs
        if (items.length < 4)
            throw new Exception("Lack of Registers or Args!");
        else if (items.length > 4)
            throw new Exception("Too Many Registers or Args!");
        rs = Register.RegAddress(items[2]);
        rt = Register.RegAddress(items[1]);
        ofs = Expr.calc(items[3]) & 0xFFFF;
        break;
    case "lw": //command $rt,ofs($rs)
    case "sw":
        if (items.length < 4)
            throw new Exception("Lack of Registers or Args!");
        else if (items.length > 4)
            throw new Exception("Too Many Registers or Args!");
        rs = Register.RegAddress(items[3].replace(")",""));
        rt = Register.RegAddress(items[1]);
        ofs = Expr.calc(items[2]) & 0xFFFF;
        break;
    case "beq":
    case "bne":
        if (items.length < 4)
            throw new Exception("Lack of Register or Args!");
        else if (items.length > 4)
            throw new Exception("Too Many Registers or Args!");
        rs = Register.RegAddress(items[1]);
        rt = Register.RegAddress(items[2]);
        ofs = (labelSet.get(items[3]) - (startAddr + line * 4 + 4)) >> 2;
        break;
}
return (opcode << 26) | (rs << 21) | (rt << 16) | (ofs & 0xFFFF);
```

J指令编译样例：

```java
int ofs = 0, func = op.JFuncCode(items[0]);
switch (items[0]) {
    case "j":
    case "jal":
        if (items.length != 2)
            throw new Exception("Too Many Registers or Args!");
        if (labelSet.containsKey(items[1]))
            ofs = (labelSet.get(items[1])) >> 2;
        else throw new Exception("label " + items[1] + " does not exist!");
        break;
}
return (func << 26) | (ofs & 0x3FFFFFF);
```

反汇编：

```
int op = (bin >> 26) & 63;
int rs = (bin >> 21) & 31;
int rt = (bin >> 16) & 31;
int rd = (bin >> 11) & 31;
int funct = bin & 63;
int immediate = bin & 0xFFFF;
int addr = bin & 0x3FFFFFF;
switch (op) {
    case 0:
        switch (funct) {
            case 32: // add
                str += "ADD $" + Register.regName(rd) + ",$" + Register.regName(rs) + ",$" + Register.regName(rt);
                break;
            case 42: // slt
                str += "SLT $" + Register.regName(rd) + ",$" + Register.regName(rs) + ",$" + Register.regName(rt);
                break;
        }
        break;
    case 2: // j
        int realAddr = ((startAddr + 4 * line + 4) & 0xF0000000) | (addr << 2);
        str += "J " + matchKeyByValue(labelSet, realAddr);
        break;
    case 4: // beq
        realAddr = ((startAddr + 4 * line + 4) + (short) (immediate << 2));
        str += "BEQ $" + Register.regName(rs) + ",$" + Register.regName(rt) + "," + matchKeyByValue(labelSet, realAddr);
        break;
    case 8: // addi
        str += "ADDI $" + Register.regName(rt) + ",$" + Register.regName(rs) + "," + (short) immediate;
        break;
    case 35: // lw
        str += "LW $" + Register.regName(rt) + "," + (short) immediate + "($" + Register.regName(rs) + ")";
        break;
    case 43: // sw
        str += "SW $" + Register.regName(rt) + "," + (short) immediate + "($" + Register.regName(rs) + ")";
        break;
    default:
        throw new Exception("invalid binary code!");
}
return str;
```

指令执行：

```
switch (op) {
    case 0:
        switch (funct) {
            case 32: // add  $rd = $rs + $rt
                val = reg.getRegValue(rs) + reg.getRegValue(rt);
                reg.setRegValue(rd, val);
                break;
            case 34: // sub  $rd = $rs + $rt
                val = reg.getRegValue(rs) - reg.getRegValue(rt);
                reg.setRegValue(rd, val);
                break;
            case 42: // slt $rd = $rs < $rt ? 1 : 0
                val = reg.getRegValue(rs) < reg.getRegValue(rt) ? 1 : 0;
                reg.setRegValue(rd, val);
                break;
        }
        break;
    case 2: // j
        PC = (PC & 0xF0000000) | (addr << 2);
        break;
    case 4: // beq if ($rs==$rt) PC=PC+4+ofs else PC=PC+4
        if (rs == rt) PC += (short) (immediate << 2);
        break;
    case 5: // bne
        if (rs != rt) PC += (short) (immediate << 2);
        break;
    case 8: // addi $rt = $rs + ofs
        val = reg.getRegValue(rs) + (short) immediate;
        reg.setRegValue(rt, val);
        break;
    case 35: // lw  $rt = Memory[$rs+ofs]
        val = mem.readData(reg.getRegValue(rs) + (short) immediate);
        reg.setRegValue(rt, val);
        break;
    case 43: // sw Memory[$rs+ofs] = $rt
        if (!mem.writeData(reg.getRegValue(rs) + (short) immediate, reg.getRegValue(rt)))
            throw new Exception("Failed to write data!");
        break;
    default:
        throw new Exception("invalid binary code!");
}
```

3.2 内存模拟

【注】内存模拟采取`Big-Endian`模式,高位放置在内存低位

读数据:

```
public int readData(int offset) {  //读入四个字节
    if (offset + 4 > memSize) return 0;  //溢出
    return ((int) data[offset] << 24) & 0xFF000000 |  //24~31位
        ((int) data[offset + 1] << 16) & 0x00FF0000 | //16~23位
        ((int) data[offset + 2] << 8) & 0x0000FF00 |  //8~15位
        (int) data[offset + 3] & 0x000000FF; //0~7位
}
```

```

public int readByte(int pos) throws Exception { //读入一个字节
    if (pos < memSize) {
        return data[pos];
    }
    throw new Exception("invalid position" + pos);
}
```

写数据：

```
public boolean writeData(int offset, int val) {
    if (offset % 4 != 0) {  //目前只支持在4的整数倍地址写数据
        System.out.println("Address divided by 4 must remain 0!");
        return false;
    }
    if (offset + 4 > memSize) return false;  //溢出
    for (int i = 3; i >= 0; i--) {
        data[offset + i] = (byte) (val & 0xFF);  //写入每一个字节
        val >>= 8;
    }
    if (addrSet.contains(offset)==false)
            addrSet.add(offset); //标记该地址被写入过数据
    return true;
}

public boolean writeInstruction(int val) {  //写指令到内存
    boolean flag = writeData(pinst, val);
    pinst += 4;
    return flag;
}
```

4 实验验证

使用如下程序对MIPS模拟机的各项功能进行验证：

等价于

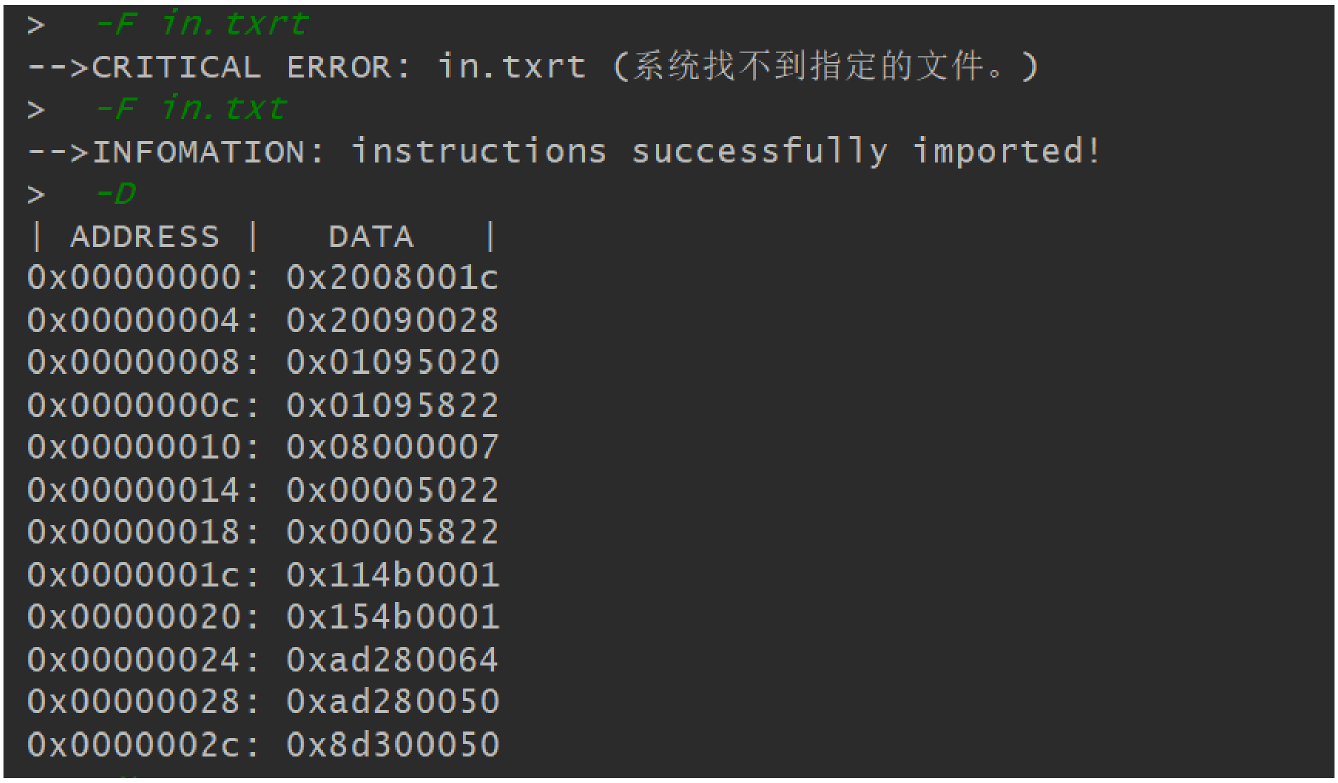
```
/*File: in.txt*/
main:   addi $t0,$zero,28
        addi $t1,$zero,40
        add $t2,$t0,$t1
        sub $t3,$t0,$t1
        j jump
        sub $t2,$zero,$zero
        sub $t3,$zero,$zero
jump:   beq $t2,$t3,label
        bne $t2,$t3,exit
label:  sw $t0,15*4+40($t1)
exit:   sw $t0,15*4+20($t1)
        lw $s0,80($t1)

```

```
end
```

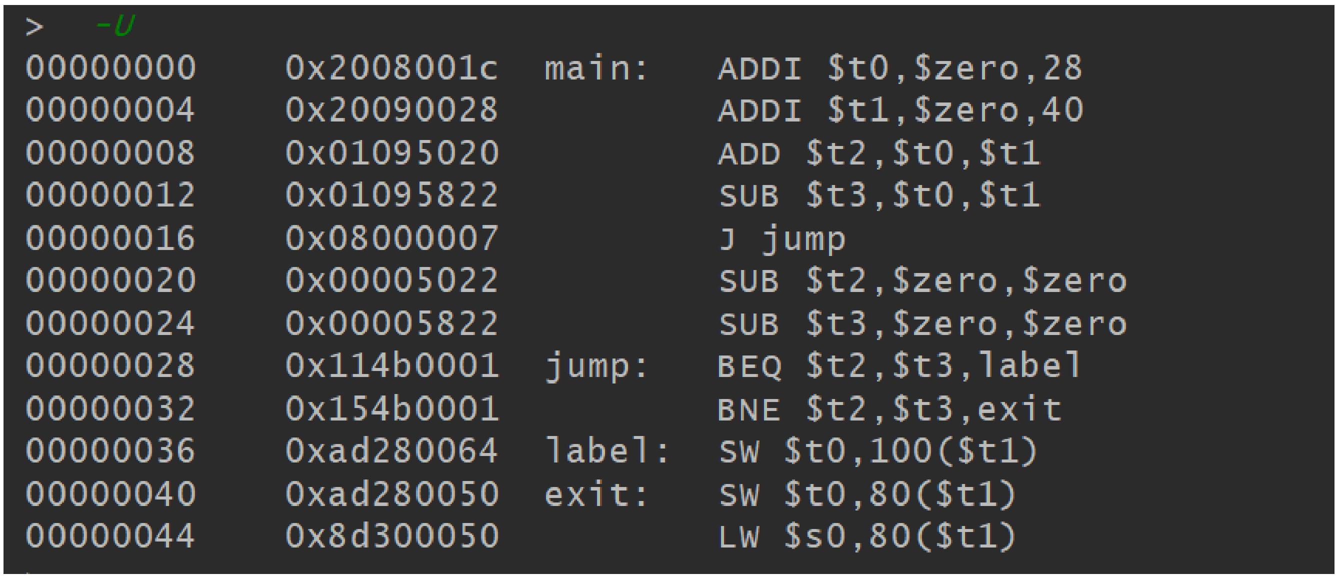
演示流程：

测试点1：测试异常抛出与指令批量导入功能。测试通过-D以数据方式看内存



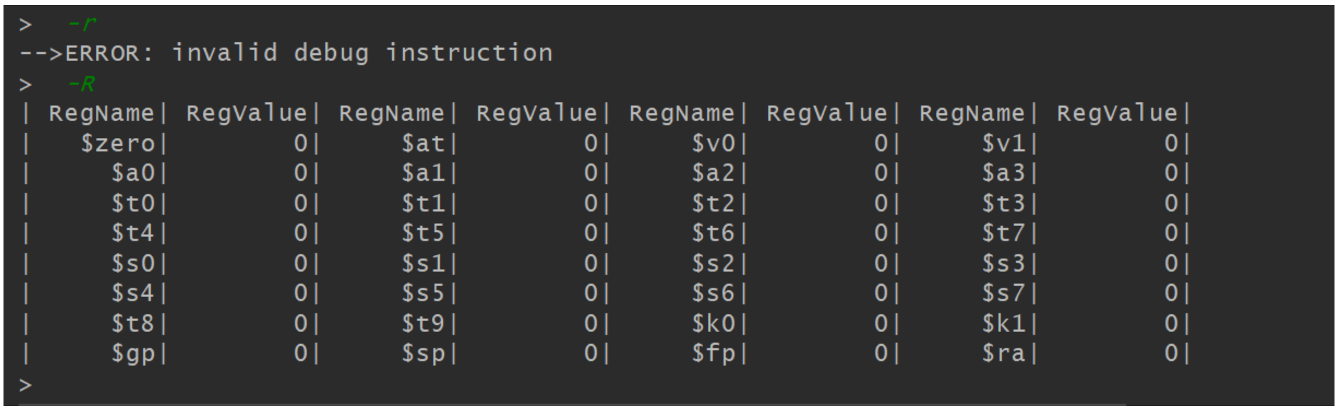
```
>  -F in.txrt
-->CRITICAL ERROR: in.txrt (系统找不到指定的文件。)
>  -F in.txt
-->INFOMATION: instructions successfully imported!
>  -D
| ADDRESS |   DATA    |
0x00000000: 0x2008001c
0x00000004: 0x20090028
0x00000008: 0x01095020
0x0000000c: 0x01095822
0x00000010: 0x08000007
0x00000014: 0x00005022
0x00000018: 0x00005822
0x0000001c: 0x114b0001
0x00000020: 0x154b0001
0x00000024: 0xad280064
0x00000028: 0xad280050
0x0000002c: 0x8d300050
```

测试点2：测试-U以指令方式看内存



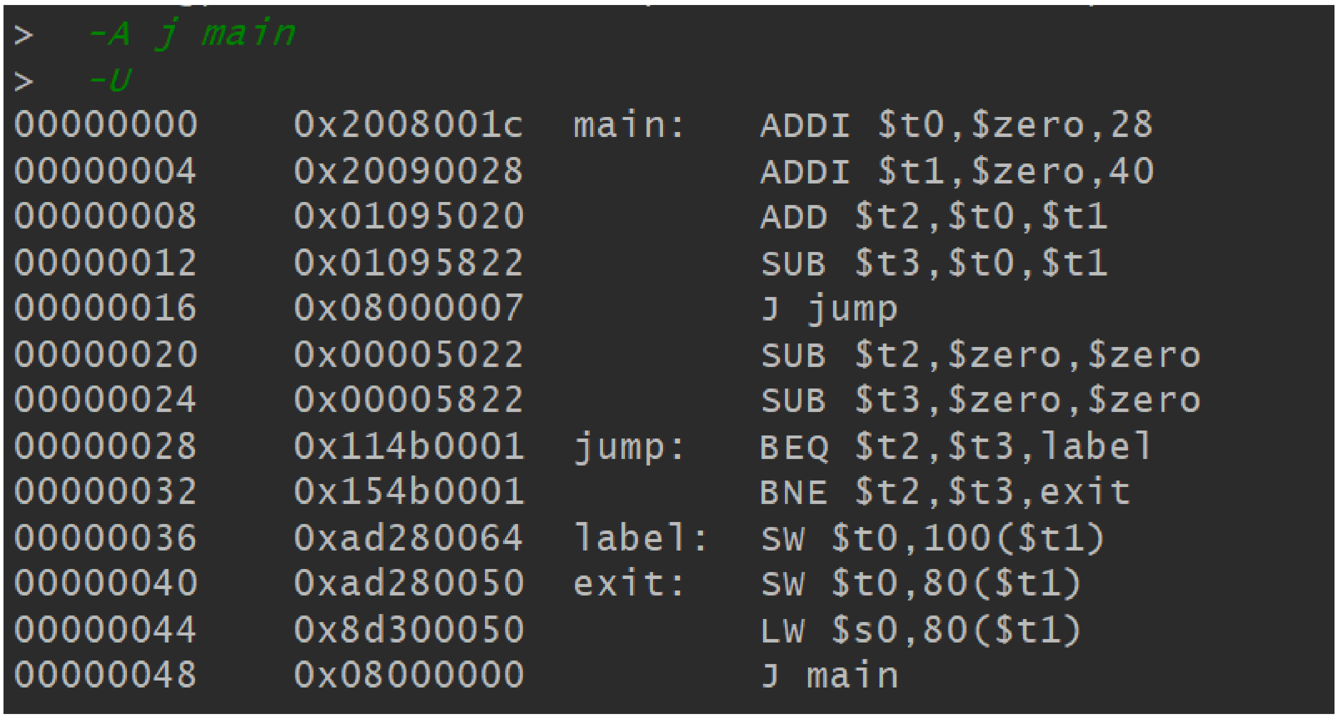
```
>  -U
00000000    0x2008001c  main:   ADDI $t0,$zero,28
00000004    0x20090028          ADDI $t1,$zero,40
00000008    0x01095020          ADD $t2,$t0,$t1
00000012    0x01095822          SUB $t3,$t0,$t1
00000016    0x08000007          J jump
00000020    0x00005022          SUB $t2,$zero,$zero
00000024    0x00005822          SUB $t3,$zero,$zero
00000028    0x114b0001  jump:   BEQ $t2,$t3,label
00000032    0x154b0001          BNE $t2,$t3,exit
00000036    0xad280064  label:  SW $t0,100($t1)
00000040    0xad280050  exit:   SW $t0,80($t1)
00000044    0x8d300050          LW $s0,80($t1)
```

测试点3：测试DEBUG指令的异常抛出，以及-R查看寄存器



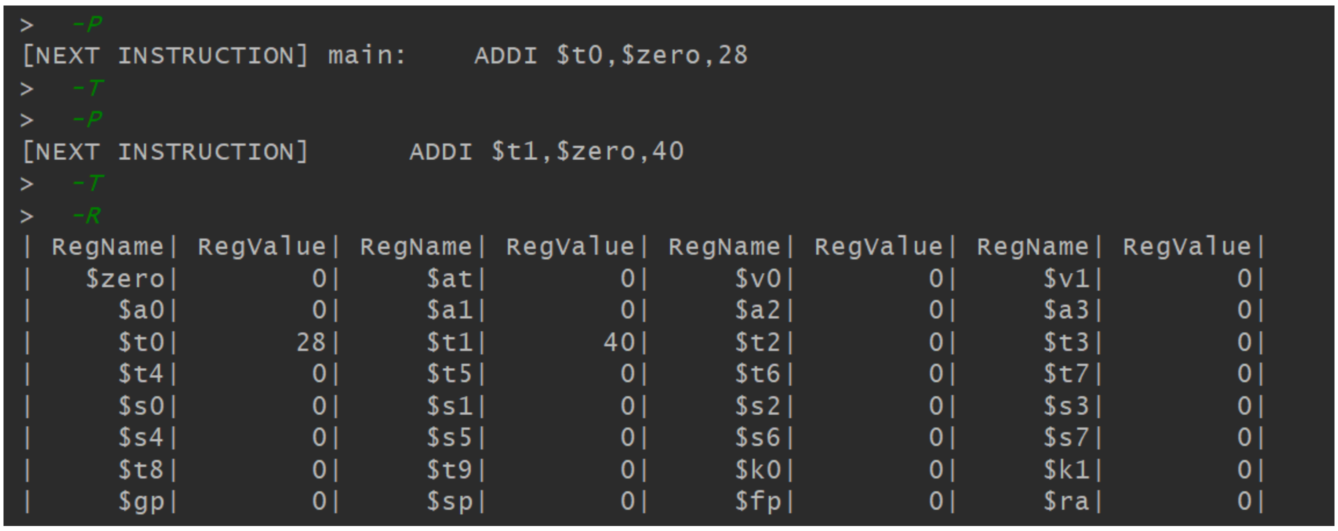
```
>  -r
-->ERROR: invalid debug instruction
>  -R
| RegName| RegValue| RegName| RegValue| RegName| RegValue| RegName| RegValue|
|   $zero|        0|     $at|        0|     $v0|        0|     $v1|        0|
|     $a0|        0|     $a1|        0|     $a2|        0|     $a3|        0|
|     $t0|        0|     $t1|        0|     $t2|        0|     $t3|        0|
|     $t4|        0|     $t5|        0|     $t6|        0|     $t7|        0|
|     $s0|        0|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|        0|     $s7|        0|
|     $t8|        0|     $t9|        0|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
>
```

测试点4：测试通过-A来往内存中写指令的功能



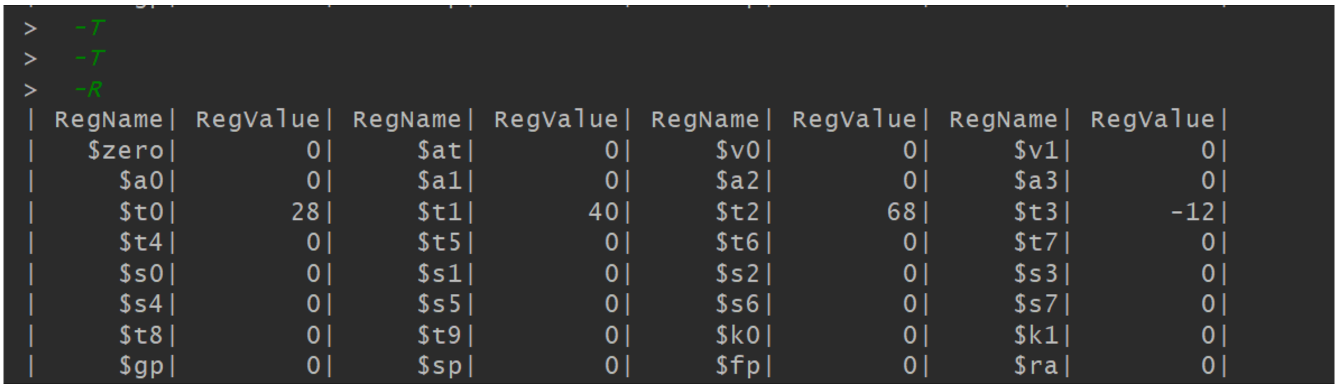
```
>  -A j main
>  -U
00000000    0x2008001c  main:   ADDI $t0,$zero,28
00000004    0x20090028          ADDI $t1,$zero,40
00000008    0x01095020          ADD $t2,$t0,$t1
00000012    0x01095822          SUB $t3,$t0,$t1
00000016    0x08000007          J jump
00000020    0x00005022          SUB $t2,$zero,$zero
00000024    0x00005822          SUB $t3,$zero,$zero
00000028    0x114b0001  jump:   BEQ $t2,$t3,label
00000032    0x154b0001          BNE $t2,$t3,exit
00000036    0xad280064  label:  SW $t0,100($t1)
00000040    0xad280050  exit:   SW $t0,80($t1)
00000044    0x8d300050          LW $s0,80($t1)
00000048    0x08000000          J main
```

测试点5：测试通过-P查看下一步需要执行的指令，-T执行一步指令



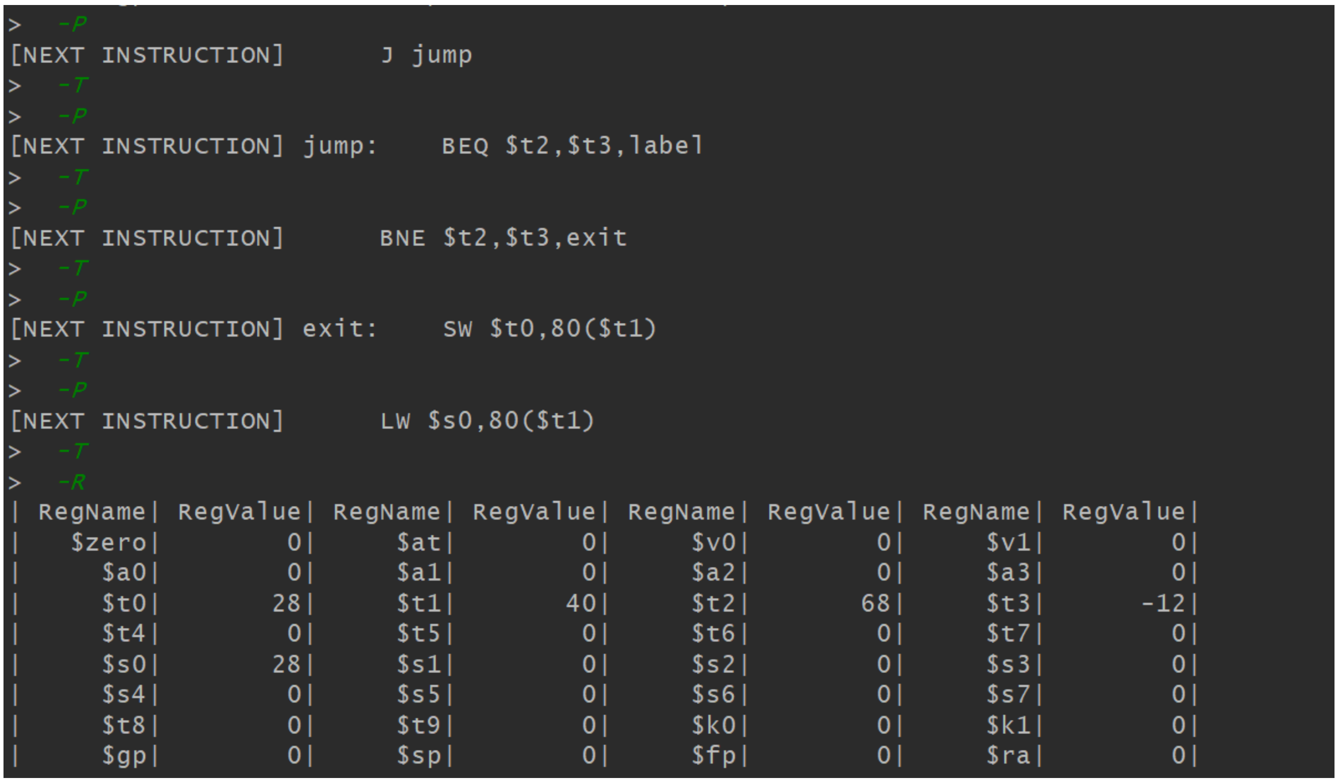
```
>  -P
[NEXT INSTRUCTION] main:    ADDI $t0,$zero,28
>  -T
>  -P
[NEXT INSTRUCTION]         ADDI $t1,$zero,40
>  -T
>  -R
| RegName| RegValue| RegName| RegValue| RegName| RegValue| RegName| RegValue|
|   $zero|        0|     $at|        0|     $v0|        0|     $v1|        0|
|     $a0|        0|     $a1|        0|     $a2|        0|     $a3|        0|
|     $t0|       28|     $t1|       40|     $t2|        0|     $t3|        0|
|     $t4|        0|     $t5|        0|     $t6|        0|     $t7|        0|
|     $s0|        0|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|        0|     $s7|        0|
|     $t8|        0|     $t9|        0|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
```

测试点6：测试运行结果是否正确



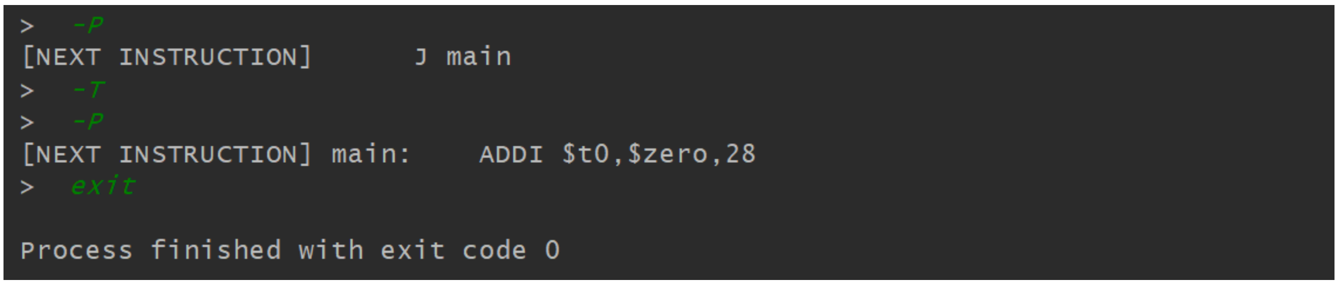
```
> -T
> -T
> -R
| RegName| RegValue| RegName| RegValue| RegName| RegValue| RegName| RegValue|
|   $zero|        0|     $at|        0|     $v0|        0|     $v1|        0|
|     $a0|        0|     $a1|        0|     $a2|        0|     $a3|        0|
|     $t0|       28|     $t1|       40|     $t2|       68|     $t3|      -12|
|     $t4|        0|     $t5|        0|     $t6|        0|     $t7|        0|
|     $s0|        0|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|        0|     $s7|        0|
|     $t8|        0|     $t9|        0|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
```

测试点7：测试运行结果是否正确



```
> -P
[NEXT INSTRUCTION]      J jump
> -T
> -P
[NEXT INSTRUCTION] jump:    BEQ $t2,$t3,label
> -T
> -P
[NEXT INSTRUCTION]      BNE $t2,$t3,exit
> -T
> -P
[NEXT INSTRUCTION] exit:    SW $t0,80($t1)
> -T
> -P
[NEXT INSTRUCTION]      LW $s0,80($t1)
> -T
> -R
| RegName| RegValue| RegName| RegValue| RegName| RegValue| RegName| RegValue|
|   $zero|        0|     $at|        0|     $v0|        0|     $v1|        0|
|     $a0|        0|     $a1|        0|     $a2|        0|     $a3|        0|
|     $t0|       28|     $t1|       40|     $t2|       68|     $t3|      -12|
|     $t4|        0|     $t5|        0|     $t6|        0|     $t7|        0|
|     $s0|       28|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|        0|     $s7|        0|
|     $t8|        0|     $t9|        0|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
```

测试点8：测试运行结果是否正确，以及通过exit结束程序



```
> -P
[NEXT INSTRUCTION]      J main
> -T
> -P
[NEXT INSTRUCTION] main:    ADDI $t0,$zero,28
> exit

Process finished with exit code 0
```