MIPS模拟机技术档案

组员：应承峻（MIPS模拟机）、贺婷婷、张佳瑶、王汀

1 功能描述

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。

2 程序运行原理

2.0 `Main`类：主类，启动GUI界面

类介绍：

`Main`类主要用于启动GUI界面，用户对指令进行编译时会创建一个新的模拟机。

2.1 `MipsSimulator`类：创建一个新模拟机

类介绍：

`MipsSimulator`类为系统创造一个新初始化的MIPS模拟机，它包含了一个`32`位寄存器，一个`8MB`虚拟内存，以及一个指令数量计数器和指令信息存储器。

重要变量：

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

构造函数：

```
public MipsSimulator(String textArea); //根据GUI界面文本输入框中的指令内容初始化一个模拟机
```

DEBUG模式使用方法：

```
-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
-I: 查看提示信息
```

2.2 `Register`类：寄存器

类介绍：

`Register`类包含了一个模拟寄存器，系统可以根据寄存器地址或名称获取或设置寄存器的值，也可以通过寄存器的名称得到寄存器的地址或通过地址得到寄存器的名称。

重要变量：

```
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：

```
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

类介绍：

`Memory`类包含了一个模拟的虚拟内存，系统可以根据该类的公有接口对数据进行读写操作。读写操作的具体实现通过单页映射的方式进行。

重要变量：

```
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
static final int PAGESIZE = 65536; //虚拟内存的每一页为64KB
static final int PAGENUM = 128;  //虚拟内存占8M空间
```

函数调用：

```java
public int readData(int addr); //从内存中addr处读取4个字节的数据
public void writeData(int addr, int val); //写数据到内存addr处的4个字节
public byte readByte(int addr);  //从内存中addr处读取1个字节的数据
public void writeByte(int addr, byte val); //写数据到内存addr处的1个字节
public short readHalf(int addr);//从内存中addr处读取2个字节的数据
public void writeHalf(int addr, short val);//写数据到内存addr处的2个字节
public boolean writeInstruction(int val);  //写指令到内存
```

2.4 `Expr`类：表达式计算

类介绍：

`Expr`类主要提供了对一个表达式进行计算的功能，用于实现形如`sw $rt,3*12+25($rs)`这样的指令的表达式计算功能。

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

类介绍：

`Instrction`类为每一条指令（伪指令除外，伪指令会先被拆分成多条普通指令再依次为为每一条指令创造一个实例。）创造一个实例。通过该类中提供的方法，可以对指令进行汇编、以及执行指令。也可以通过提供的静态方法对任意一个二进制指令进行反汇编。

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 void 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
```

当前支持的R类型指令：

```
sll rd  rt  sa
srl rd  rt  sa
sra rd  rt  sa
sllv    rd  rt  rs
srlv    rd  rt  rs
srav    rd  rt  rs
jr  rs
jalr    rd  rs
syscall
mult    rs  rt
multu   rs  rt
div rs  rt
divu    rs  rt
add rd  rs  rt
addu    rd  rs  rt
sub rd  rs  rt
subu    rd  rs  rt
and rd  rs  rt
or  rd  rs  rt
xor rd  rs  rt
nor rd  rs  rt
sltu    rd  rs  rt
slt rd  rs  rt
```

当前支持的I类型指令：

```
beq rs  rt  label
bne rs  rt  label
addi    rt  rs  ofs
addiu   rt  rs  ofs
slti    rt  rs  ofs
sltiu   rt  rs  ofs
andi    rt  rs  ofs
ori rt  rs  ofs
xori    rt  rs  ofs
lui rt  ofs
lb  rt  ofs(rs)
lh  rt  ofs(rs)
lw  rt  ofs(rs)
lbu rt  ofs(rs)
lhu rt  ofs(rs)
sb  rt  ofs(rs)
sh  rt  ofs(rs)
sw  rt  ofs(rs)
```

当前支持的J类型指令：

```
j label
jal label
```

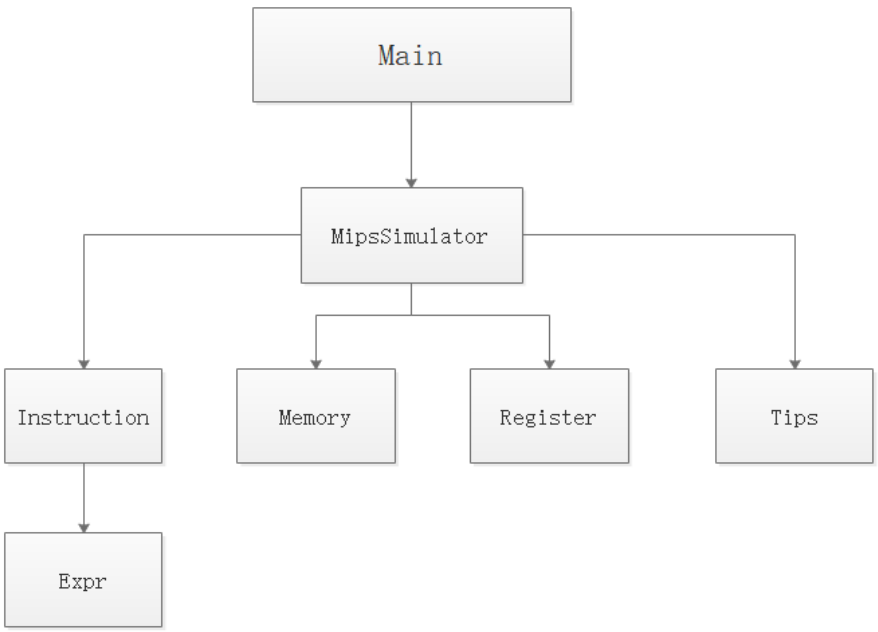
当前支持的伪指令：

```
move rd,rs
blt rs,rt,RR
bgt rs,rt,RR
ble rs,rt,RR
bge rs,rt,RR
swap rs,rt
sne rd,rs,rt
abs rd,rt
```

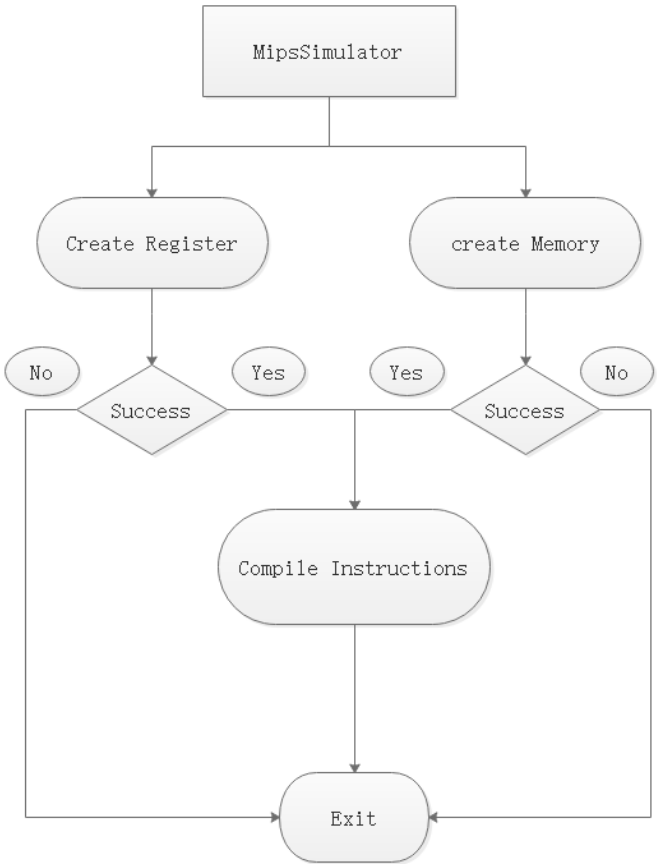
3 流程图与算法分析

3.0 程序架构与模拟器启动

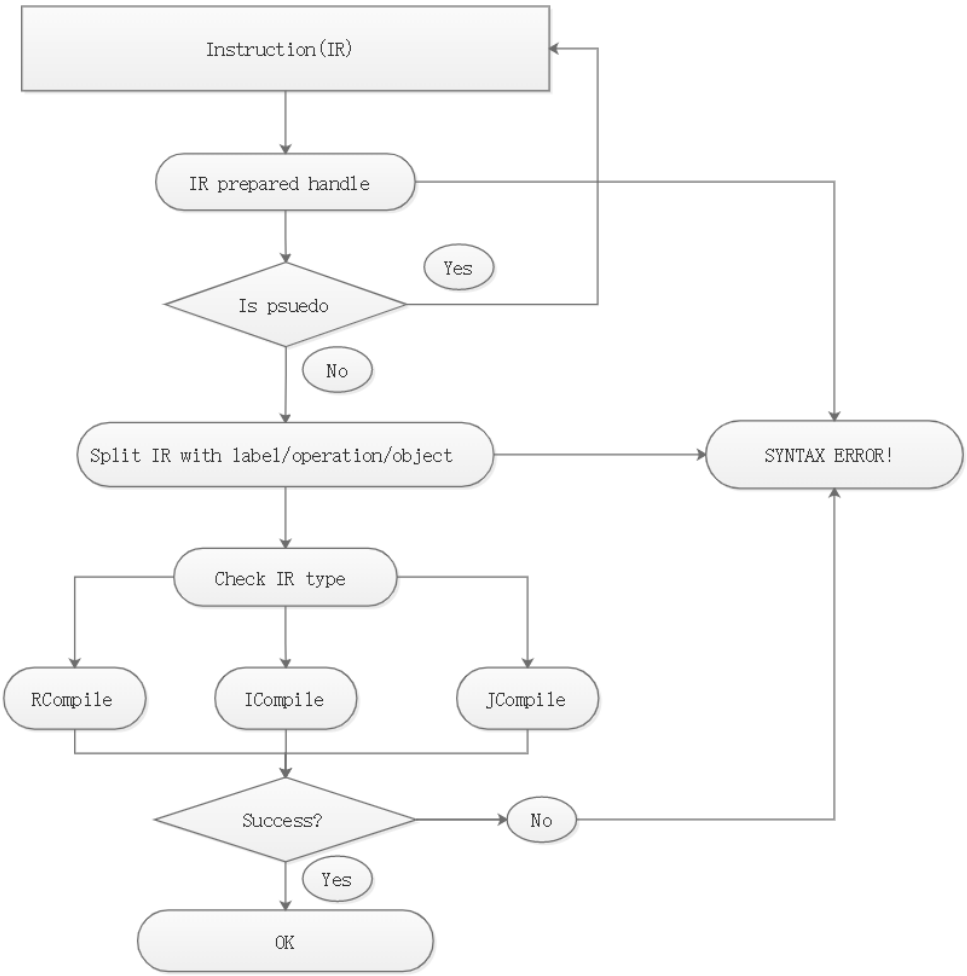
程序架构



模拟机启动：


MipsSimulator
Create Register
create Memory
No
Success
Yes
Yes
Success
No
Compile Instructions
Exit

3.1 指令编译处理



字符串预处理：将指令全部转换成小写字符，并将`|,|\t|($|$|`这几类符号全部替换成普通空格，然后去掉整个字符串首尾的空格

```
//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：通过正则表达式匹配指令标签，如果指令带有标签则将它从指令中删去，并将标签存储到哈希表中。如果标签已经存在则抛出异常。

```
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：根据`OperationCode`类中提供的接口判断指令属于哪种类型，并采用不同的编译方式。

```
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：先对指令的参数进行判断，如果指令参数不符合要求则抛出异常，否则生成对应的二进制码。

```
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;
```

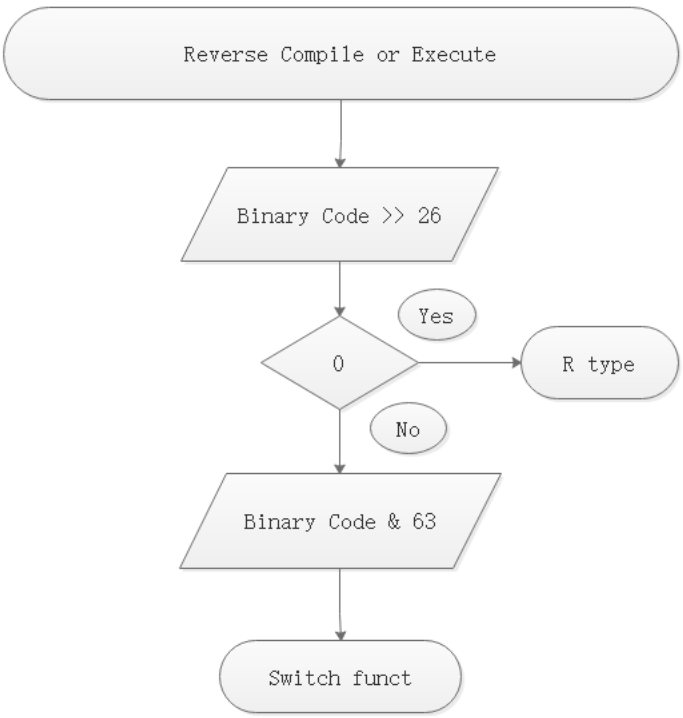
I指令编译样例：先对指令的参数进行判断，如果指令参数不符合要求则抛出异常，否则生成对应的二进制码。

```
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：先对指令的参数进行判断，如果指令参数不符合要求则抛出异常，否则生成对应的二进制码。

```
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：对二进制码进行切分，如果Op为0则做对R类型指令的反汇编处理，否则根据Funct做相应类型的反汇编处理。

```
int op = (bin >> 26) & 63;
```

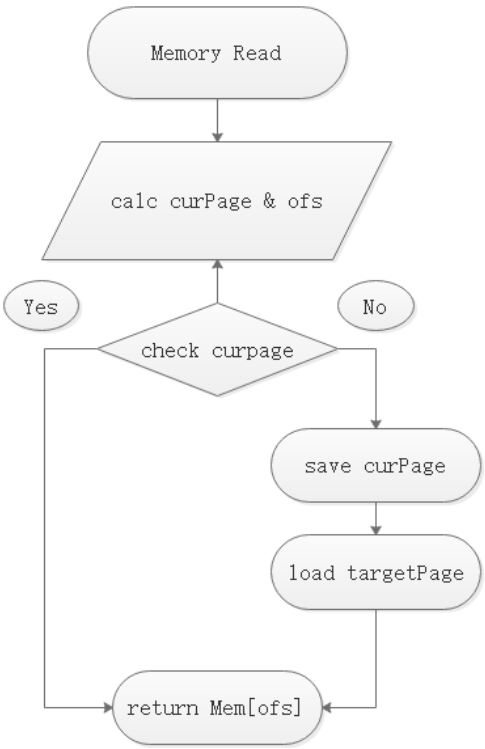
```
int rs = (bin >> 21) & 31;
int rt = (bin >> 16) & 31;
int rd = (bin >> 11) & 31;
int funct = bin & 63;
int immediate = bin & 0xFFFF;
int addr = bin & 0x3FFFFFF;
switch (op) {
    case 0:
        switch (funct) {
            case 32: // add
                str += "ADD $" + Register.regName(rd) + ",$" + 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;
```

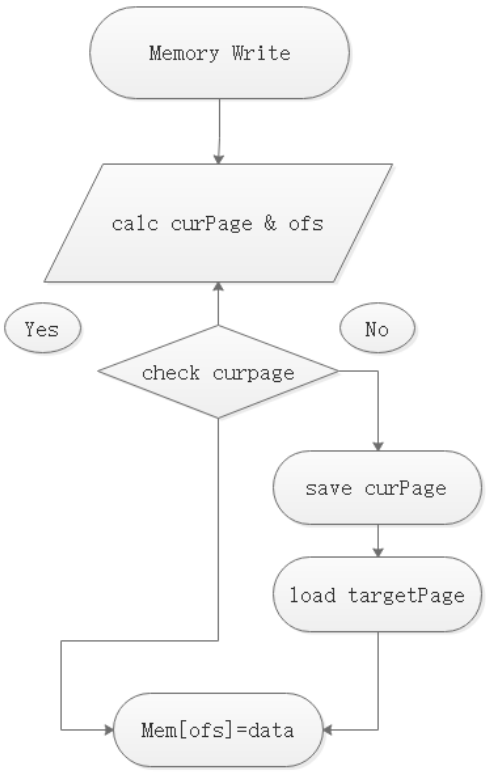
指令执行：类似于反汇编

```
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)))
            h new Exception("Failed to write data!");
        break;
    default:
        throw new Exception("invalid binary code!");
}
```

3.2 内存模拟





【1】内存模拟采取`Big-Endian`模式,高位放置在内存低位

【2】采用了单页映射的虚拟内存，每页为64KB，共128页，虚拟内存总计8MB

读数据：采用单页映射的方式进行读数据，如果待读数据的地址所对应的页在当前内存中，则直接返回，否则先将原来的页写回虚拟内存然后再将该页替换到内存中。根据`Big-Endian`的原则，假设起始地址为`addr`，则内存中的`addr~addr+8`是整数的最高一个字节，`addr+8~addr+16`是整数的次高一个字节，...，以此类推。

```
public int readData(int addr) throws Exception {
    int pag = addr / PAGESIZE;  //页号
    int ofs = addr % PAGESIZE;  //页偏移
    if (curPage != pag) {  //不在页内
        save(curPage);  //保存
        load(pag);  //重新加载
    }
    return readInt(ofs);
}

private int readInt(int offset) {
    if (offset + 4 > PAGESIZE) return 0;
    return ((int) data[offset] << 24) & 0xFF000000 |
        ((int) data[offset + 1] << 16) & 0x00FF0000 |
        ((int) data[offset + 2] << 8) & 0x0000FF00 |
        (int) data[offset + 3] & 0x000000FF;
}
```

写数据：采用单页映射的方式进行写数据时，如果待读数据的地址所对应的页在当前内存中，则直接写入，否则先将原来的页写回虚拟内存然后再将该页替换到内存中。由于内存过大，当用户想查看内存时输出全部的数据显然不合理，因此在这里将每次用户写入的地址加以记录，当查看内存时，只查看有数据写入的内存。

```java
public void writeData(int addr, int val) throws Exception {
    try {
        int pag = addr / PAGESIZE;
        int ofs = addr % PAGESIZE;
        if (curPage != pag) {
            save(curPage);
            load(pag);
        }
        writeInt(ofs, val);
    } catch (ArrayIndexOutOfBoundsException e) {
        throw new Exception("Relative address out of bound!");
    } catch (Exception e) {
        throw e;
    }
}

private void writeInt(int offset, int val) {
    if (offset + 4 > PAGESIZE) return  //溢出
    for (int i = 3; i >= 0; i--) {
        data[offset + i] = (byte) (val & 0xFF);    //依次写入每个自己
        val >>= 8;
    }
    if (!addrSet.contains(offset)) {  //记录地址
        addrSet.add(curPage * PAGESIZE + offset);
        addrSet.sort(Integer::compareTo);
    }
}
```

虚拟内存页的加载和保存：加载和保存页作为私有方法，只提供给公有接口进行调用。其实现原理是通过文件的定位与读写，将整个虚拟内存存储在一个文件中，当需要加载内存时，则通过`RandomAccessFile`类对块在文件中的位置进行定位，然后对文件进行读写。

```java
private void save(int page) throws Exception {
    File file = new File("DISKDATA");  //虚拟内存
    RandomAccessFile raf = null;
    try {
        raf = new RandomAccessFile(file, "rw");
        raf.seek(PAGESIZE * page);  //块读写
        raf.write(data, 0, PAGESIZE);
    } catch (Exception e) {
        throw e;
    } finally {
        try {
            if (raf != null) raf.close();
        } catch (Exception e) {
            throw e;
        }
    }
}

private void load(int page) throws Exception {
    File file = new File("DISKDATA");
    RandomAccessFile raf = null;
```

```
    try {
        raf = new RandomAccessFile(file, "rw");
        raf.seek(PAGESIZE * page);
        raf.read(data, 0, PAGESIZE);
        curPage = page;
    } catch (Exception e) {
        throw e;
    } finally {
        try {
            if (raf != null) raf.close();
        } catch (Exception e) {
            throw e;
        }
    }
}
```

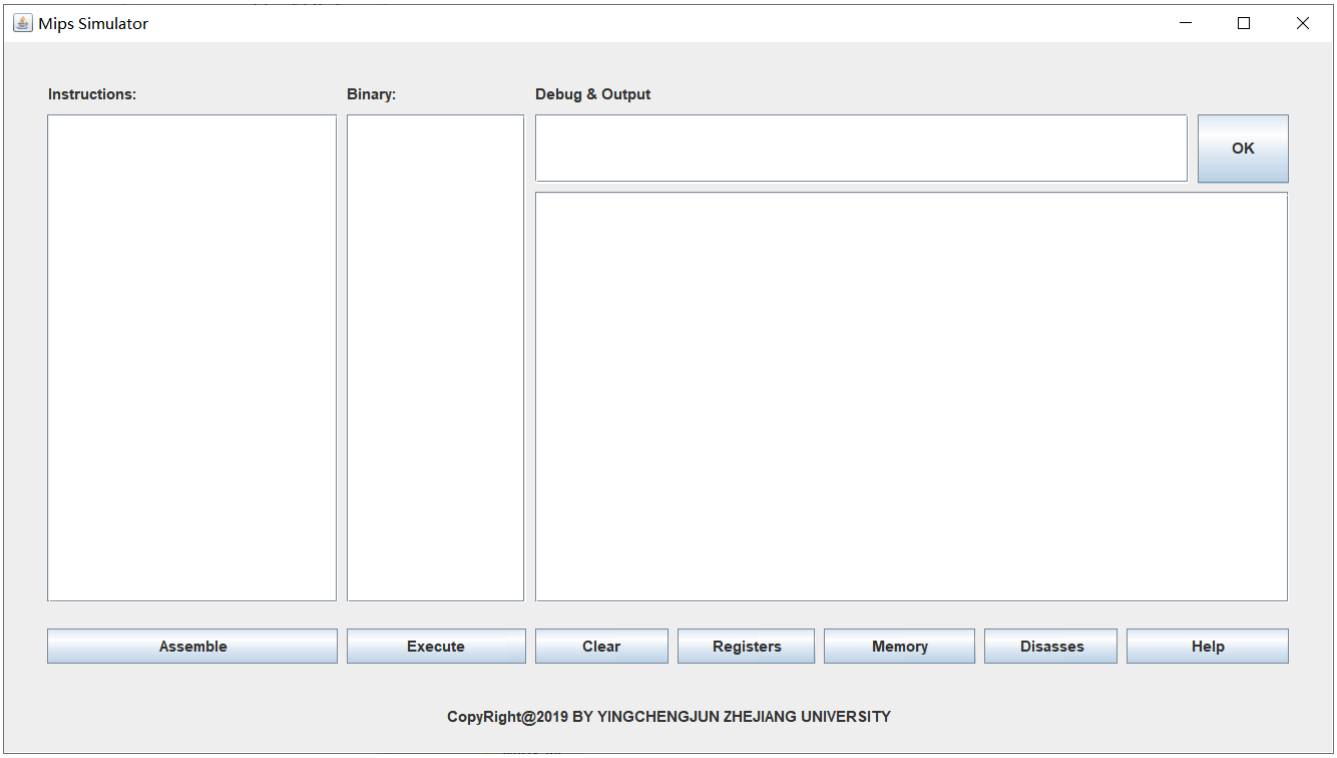
4 使用手册与使用实例

使用如下程序对MIPS模拟机的各项功能进行验证:

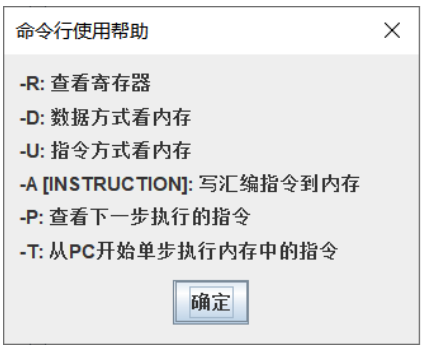
```
main:   addi $t0,$zero,28
        addi $t1,$zero,40
        addi $t9,$zero,-1
        add $t2,$t0,$t1
        sub $t3,$t0,$t1
        sll $t8,$t9,1
        srl $t7,$t9,1
        sra $t7,$t9,1
        j jump
        sub $t2,$zero,$zero
        sub $t3,$zero,$zero
jump:   beq $t2,$t3,label
        bne $t2,$t3,exit
label:  sw $t0,15*40+40($t1)
exit:   sw $t0,15*40+20($t1)
        lw $s0,620($t1)
test:   addi $s6,$zero,30000
        addiu $s6,$s6,60000
        addiu $s6,$s6,60000
        sw $t1,0($s6)
        lw $a0,0($s6)
        addi $v0,$zero,1
        syscall
psuedo: abs $t9,$t9
```

SO：在带`jar`包的文件夹中输入`java -jar Mips.jar`启动程序

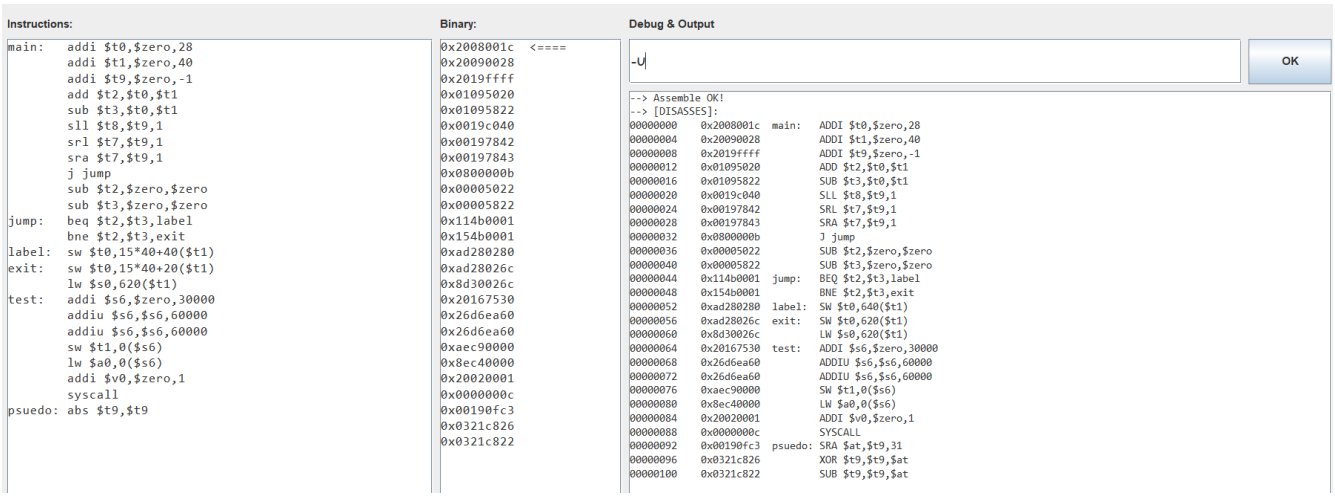
S1：界面主要分为`Instruction`（用于用户输入Mips指令），`Binary`（用于输出指令的汇编结果以及当前PC所指向的指令），`Debug`（命令行）以及`Output`（输出）。为了简化用户的使用成本，程序提供了一键清除输出（`Clear`），一键显示所有寄存器（`Registers`），一键显示内存（`Memory`）以及一键反汇编（`Disasses`）以及帮助的`Help`功能。



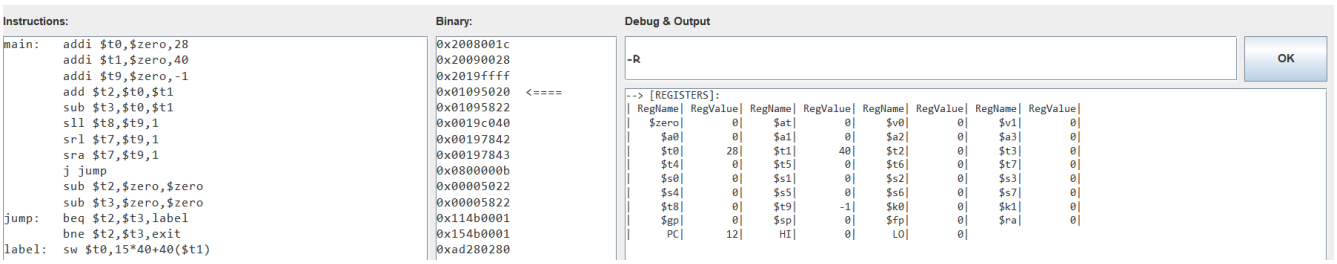
点击`Help`按钮可以看到命令行的使用帮助：



S2：在`Instruction`中输入已经准备好的指令，点击`Assemble`按钮进行汇编得到一串机器码，点击`Disasses`按钮或是在命令行中输入`-U`即可得到反汇编的结果，如下图所示：反汇编结果中依次显示指令的内存地址、指令的机器码以及指令。



S3：点击`Execute`按钮单步执行指令（或是通过命令行输入`-T`然后点击`OK`按钮）。当PC指向第4行时，点击`Registers`按钮或是在命令行中输入`-R`可以查看当前寄存器中存放的值，可以看到在执行了三条`addi`指令后，寄存器`$t0`、`$t1`、`$t9`的值分别被修改成了`28`、`40`和`-1`。



S4：继续执行指令，当PC指向第9行时（`j jump`）时，查看寄存器的值，此时

`68=$t2=$t0+$t1=28+40`，`-12=$t3=$t0-$t1=28-40`，`-2=$t8=$t9<<1=-1<<1`，

`-1=$t7=$t9>>>1=-1>>>1=-1`

```
--> Executed Successfully!
--> Executed Successfully!
--> Executed Successfully!
--> Executed Successfully!
--> Executed Successfully!
--> [REGISTERS]:
| 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|       -1|
|     $s0|        0|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|        0|     $s7|        0|
|     $t8|       -2|     $t9|       -1|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
|      PC|       32|      HI|        0|      LO|        0|
```

S5：继续执行下一步，发现`j jump`指令的功能实现，跳转到`jump`标签处。而由于`68=$t2!=$t3=-12`，因此`beq`指令不做跳转而`bne`指令跳转到`exit`处。

Instructions:	Binary:
main: addi $t0,$zero,28	0x2008001c
addi $t1,$zero,40	0x20090028
addi $t9,$zero,-1	0x2019ffff
add $t2,$t0,$t1	0x01095020
sub $t3,$t0,$t1	0x01095822
sll $t8,$t9,1	0x0019c040
srl $t7,$t9,1	0x00197842
sra $t7,$t9,1	0x00197843
j jump	0x0800000b
sub $t2,$zero,$zero	0x00005022
sub $t3,$zero,$zero	0x00005822
jump: beq $t2,$t3,label	0x114b0001 <====
bne $t2,$t3,exit	0x154b0001

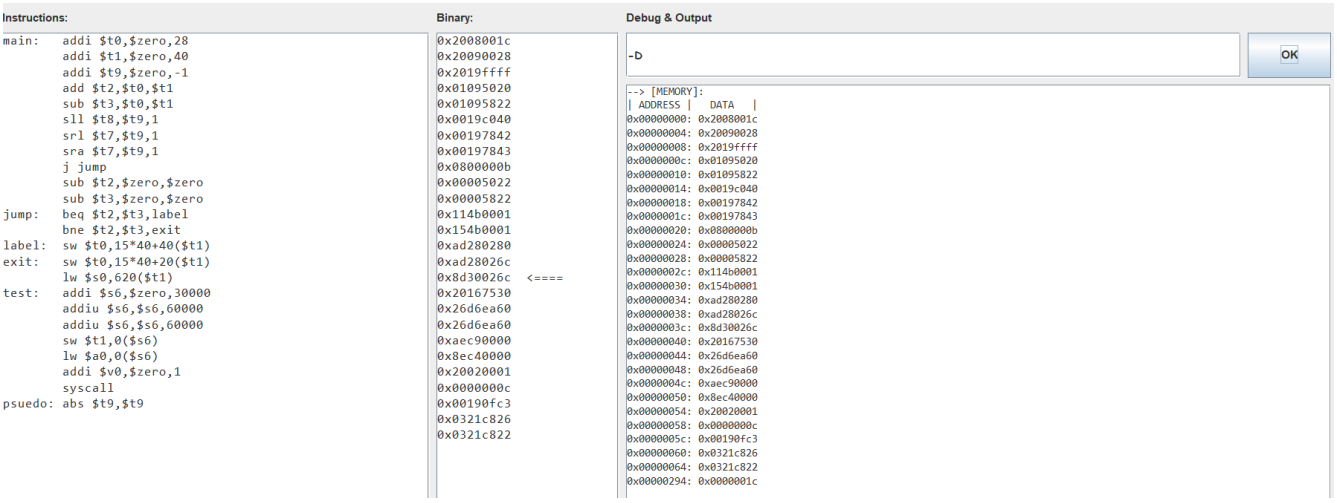
S6：可以看到PC跳转到`exit`处，此时将`$t0`的值存放到`$t1+ (15*40+40)`处，然后再讲该处的值取出放到寄存器`$s0`中。

sub $t3,$zero,$zero	0x00005822
jump: beq $t2,$t3,label	0x114b0001
bne $t2,$t3,exit	0x154b0001
label: sw $t0,15*40+40($t1)	0xad280280
exit: sw $t0,15*40+20($t1)	0xad28026c <====
lw $s0,620($t1)	0x8d30026c
test: addi $s6,$zero,30000	0x20167530

S7：再次查看寄存器，发现`$s0`中有了跟`$t0`一样的值。

```
--> Executed Successfully!
--> [REGISTERS]:
| 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|       -1|
|     $s0|       28|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|        0|     $s7|        0|
|     $t8|       -2|     $t9|       -1|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
|      PC|       64|      HI|        0|      LO|        0|
```

S8：点击`Memory`按钮或通过`-D`以数据方式查看内存，可以看到内存`0x00000294=620+40`处被存放了数据`0x0000001c=28`。



S9：为了进一步测试虚拟内存的功能，我们通过`addi`和`addiu`来将`$s6`的值加到`150000`，此时需要注意，由于`I`指令的立即数只有16位，因此在使用`addi`时，立即数不应超过$2^{15}-1=32767$，而在使用`addiu`时，立即数不应超过$2^{16}-1=65535$，否则将会出现溢出导致报错。此时查看寄存器，发现`$s6`的值满足要求。

```
--> Executed Successfully!
--> Executed Successfully!
--> Executed Successfully!
--> [REGISTERS]:
| 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|       -1|
|     $s0|       28|     $s1|        0|     $s2|        0|     $s3|        0|
|     $s4|        0|     $s5|        0|     $s6|   150000|     $s7|        0|
|     $t8|       -2|     $t9|       -1|     $k0|        0|     $k1|        0|
|     $gp|        0|     $sp|        0|     $fp|        0|     $ra|        0|
|      PC|       76|      HI|        0|      LO|        0|
```

S10：继续执行指令，将`$t1=40`处的值写入`$s6=150000`地址处，此时查看内存可以看到，内存`0x000249f0=150000`处被写入数据`0x28=40`，满足测试要求。

```
--> [MEMORY]:
| ADDRESS |   DATA   |
0x00000000: 0x2008001c
0x00000004: 0x20090028
0x00000008: 0x2019ffff
0x0000000c: 0x01095020
0x00000010: 0x01095822
0x00000014: 0x0019c040
0x00000018: 0x00197842
0x0000001c: 0x00197843
0x00000020: 0x0800000b
0x00000024: 0x00005022
0x00000028: 0x00005822
0x0000002c: 0x114b0001
0x00000030: 0x154b0001
0x00000034: 0xad280280
0x00000038: 0xad28026c
0x0000003c: 0x8d30026c
0x00000040: 0x20167530
0x00000044: 0x26d6ea60
0x00000048: 0x26d6ea60
0x0000004c: 0xaec90000
0x00000050: 0x8ec40000
0x00000054: 0x20020001
0x00000058: 0x0000000c
0x0000005c: 0x00190fc3
0x00000060: 0x0321c826
0x00000064: 0x0321c822
0x00000294: 0x0000001c
0x000249f0: 0x00000028
```

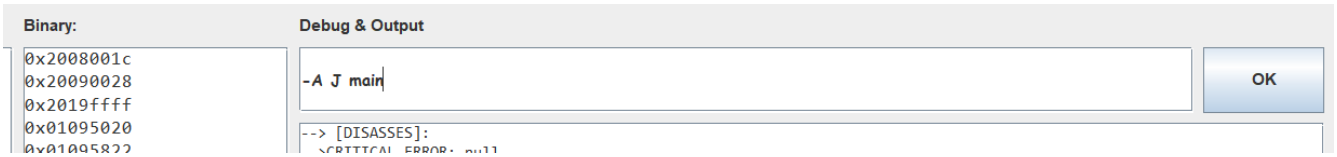
S11：接下来测试Syscall的功能，将1写入`$v0`以及将内存150000处的值写入`$a0`，进行系统调用`syscall`，可以看到在控制台中，值`40`被显示在控制台上。

```
PS D:\JavaWorkPlace\Mips\out\artifacts\Mips_jar> java -jar Mips.jar
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: cHRM chunk does not match sRGB
40
```

S12：然后测试伪指令的功能，`$t9`处原来存放的值为`-1`，在执行`abs`指令后，可以看到，`$t9`的值变成了1。

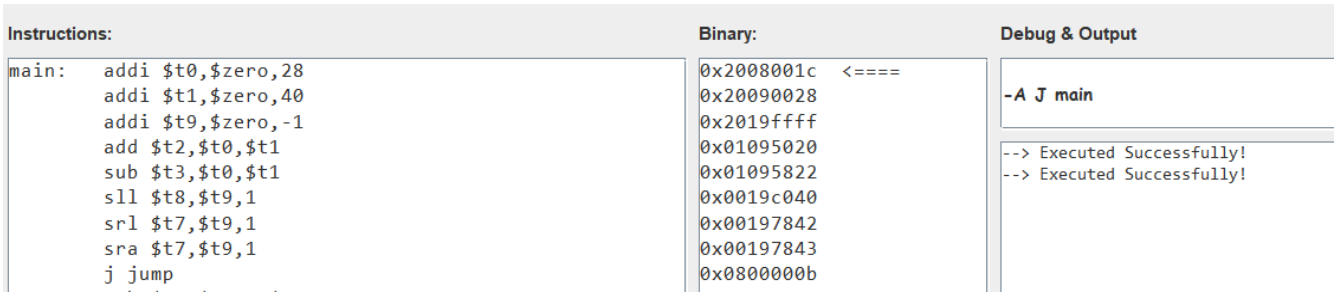
RegName	RegValue	RegName	RegValue	RegName	RegValue	RegName	RegValue
$zero	0	$at	-1	$v0	1	$v1	0
$a0	40	$a1	0	$a2	0	$a3	0
$t0	28	$t1	40	$t2	68	$t3	-12
$t4	0	$t5	0	$t6	0	$t7	-1
$s0	28	$s1	0	$s2	0	$s3	0
$s4	0	$s5	0	$s6	150000	$s7	0
$t8	-2	$t9	1	$k0	0	$k1	0
$gp	0	$sp	0	$fp	0	$ra	0
PC	104	HI	0	LO	0		

S13：最后通过`Debug`模式添加一条指令`J main`。



添加成功后，可以看到该指令被显示在指令框中。

继续指令，发现PC重新跳转回第一行。



到此完成了所有的测试。