MIPS模拟机 技术文档

MIPS Simulator Technical Document

项目经理：王子腾

组员：徐晓丹 马静怡

Date: 2020-6-30

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

2.1 Main：项目入口

模块介绍

本模块为系统入口模块，用于搭建与用户的接口，用户可在`CLI`界面内与系统进行命令交互。

路由模块

Main模块中实现路由与交互操作的核心部分代码如下：

```
while(1)            // 退出条件：用户输入 exit
{
    cout << "--> ";
    cin >> option;          // 读入指令
    if(option=="R"){
        mips.Regshow();     // 打印寄存器
    }
    else if(option=="D"){
        mips.Memshowdata(); // 数据方式看内存
    }
    else if(option=="U"){
        mips.Memshowins();  // 指令方式看内存
    }
    else if(option=="A"){
        mips.Addins();      // 添加单条指令
    }
    else if(option=="T"){
        mips.execute();     // 单步执行
        cout << endl;
    }
    else if(option=="exit"){
        return 0;           // 结束程序
    }
    else{
        cout << "--> Wrong input. " << endl;
        continue;           // 错误输入
    }
}
```

2.2 MIPS 类：创建一个模拟机

模块介绍

MIPS类为系统核心模块，起到整合其余模块并构建与**Main**模块进行信息沟通的作用。

本模块的启动标志着系统创建一个新初始化的 `32` 位***MIPS***模拟机，模拟机内部包含了一个`32-bit`虚拟寄存器，一个`1024KB`虚拟内存，一个指令存储器，以及一个程序计数器。

本模块提供了程序所需的所有5个外部接口，并通过实例化各虚拟模块调用其内部的接口，最终实现中间路由和整合的功能。

重要变量

```
int PC;          // 程序计数器，指向当前所在指令地址
int num;         // 记录当前指令条数
Memory m;        // 模拟内存
Register reg;    // 虚拟寄存器组
Instruction ins;// 虚拟指令系统
```

构造函数

```
MIPS::MIPS(int type): ins(type), PC(0){}
```

函数调用

```
void Regshow();      // 打印寄存器状态
void Memshowdata(); // 数据方式看内存
void Memshowins();  // 指令方式看内存
void Addins();       // 添加指令
void execute();      // 单步执行
```

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: 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
-exit: 退出程序
```

2.3 Register类：虚拟寄存器

模块介绍

Register 类包含了一个虚拟寄存器组，其中存放了从$zero依次到$ra的32个寄存器，编号为0~31。

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

```
// 名称索引
const string reg[32] = {"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"};
int val[32];    // 存放寄存器值
```

函数调用

```
int regindex(string);        // 根据名称获得下标索引
string regname(int);         // 根据索引获取名称
void setval(int index, int v);  // 设置寄存器值
int getval(int);             // 获取寄存器值
void show();                 // 打印寄存器组
```

2.4 Memory类：模拟内存

模块介绍

Memory 类包含了一个模拟的内存，其中存放了从$zero依次到$ra的32个寄存器，编号为0~31。

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

```
const int capacity = 1024;  // 内存大小为1KB
const int PAGESIZE = 65536; // 模拟内存的每一页为64KB
const int PAGENUM = 16;     // 虚拟内存占1M空间
int ptr = 0;                // 当前指向内存位置
string memory[1024];        // 模拟内存
int record[1024];           // 标记存储位点
```

函数调用

```
void writedata(int pos, int type, string val);  //写数据覆盖到内存的4个字节
void writehalf(int pos, bool location, string val);//写数据覆盖到内存的2个字节
string read(int pos);           // 读取内存字段
int showadd(int pos);           // 打印内存地址
void showdata();                // 打印内存数据
```

2.4 Instruction类：指令容器

模块介绍

Instruction 类内存放了从文件中读入的所有指令，并通过正则表达式将其拆分为**Label**、**Option**、**Func**等字段，进行后续的匹配、编译、执行工作。

本类为每一条指令（包含伪指令）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、反汇编指令，本模块可将字段传递给***MIPS***模块，供其进行指令的执行、内存的读写操作。

重要变量

```
vector<string> sins;    // 文本模式存储指令
vector<string> bins;    // 二进制模式存储指令
vector<string> label;   // 指令标签集
```

函数调用

```
vector<string> option;  // 指令操作码
Instruction(int type);  // 构造函数，根据type确定编译或反编译

string getins(int type, int pos);   // 根据type获取文本/二进制指令
int getsize();          // 获取当前文本指令数目
int getbsize();         // 获取当前二进制指令数目（伪代码编译造成两者差异）
void showins(int);      // 指令方式看内存
void addins(string);    // 添加单条指令
void compile(int,string);   // 编译文本指令
void reverse(int);      // 反编译二进制指令
int findlabel(string);  // 搜索标签的PC位点
```

3 架构框图与指令集合

3.1 项目架构

本项目共由6个文件、5个模块类组成，其中主要架构与相互调用关系如下图所示：

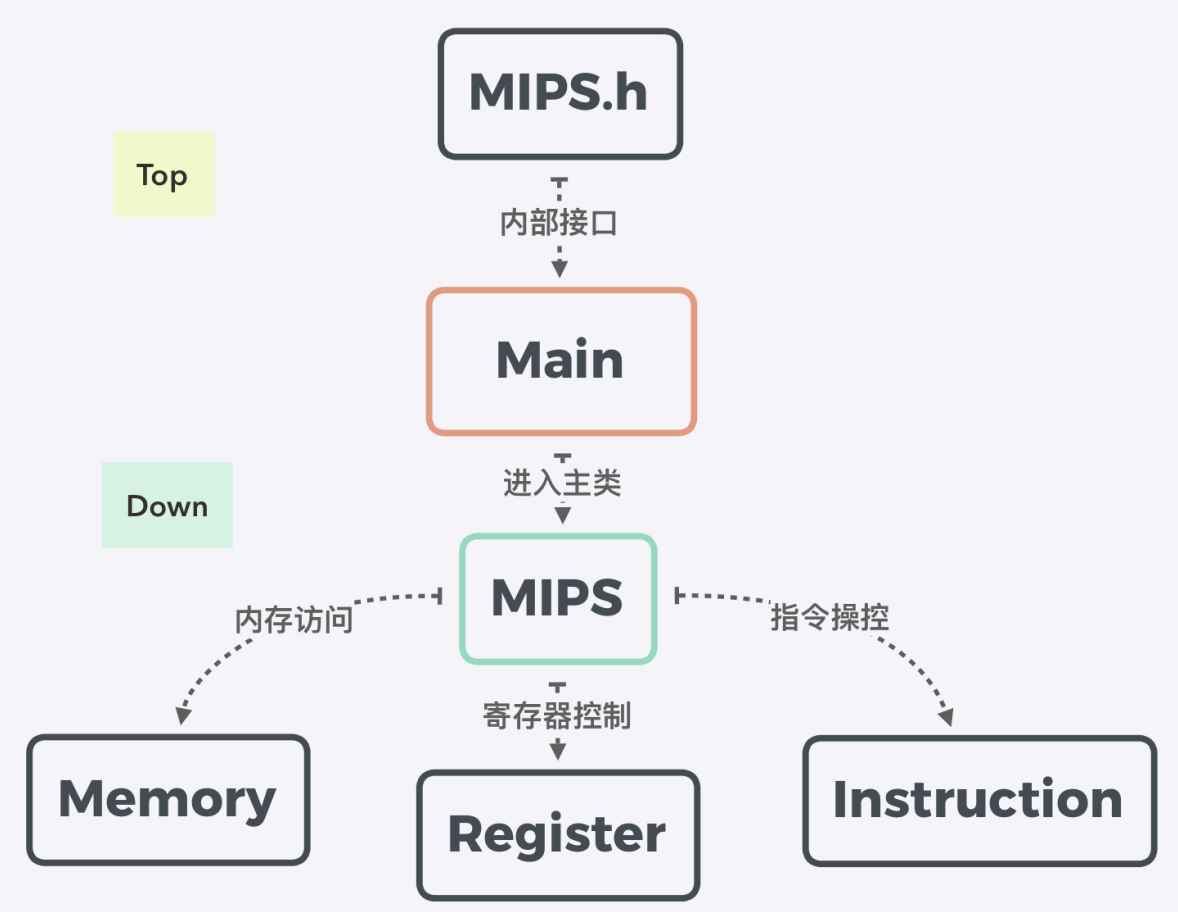

图3.1.1 MIPS模拟器系统架构

3.3 当前支持指令集

当前支持的`R`类型指令

```
add rd rs rt
sub rd rs rt
mul rd rs rt
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
and rd rs rt
or rd rs rt
xor rd rs rt
nor 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
slti rt rs ofs
andi rt rs ofs
ori rt rs ofs
xori rt rs ofs
lui rt ofs
lh rt ofs(rs)
lw 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
```

4 算法分析

4.1 指令预处理

通过正则表达式匹配指令标签，如果指令带有标签则将标签部分从指令中删去，并将标签存储到标签集中，同时记录映射的指令位点。

```
regex e("^([a-z 0-9]+):");          // 构建正则表达式
smatch m;
bool found = regex_search(instruction,m,e); // 寻找标签是否存在
if(found){          //存在标签则将它从指令中删去，并做记录
    lab = m.str();
    lab.erase(lab.end()-1);
    int i = 0;
    while(instruction[i]!=':')
    i++;
    instruction.erase(0,i+1);   // 指令分离
}
else lab = "";
label.push_back(lab);           // 存储标签
```

记录标签信息后，去掉指令中多余的空格，将指令中的字母全部转换成小写字符，之后分割出操作码部分，并存储进`option`向量中、

```
while(instruction[0]==' ')
    instruction.erase(instruction.begin());     //清除空格
transform(instruction.begin(),instruction.end(),instruction.begin(),::tolower
);          // 指令转小写字符
string op;
int i;
// 获取操作码
for(i = 0; instruction[i]!=' '; i++)
    op+=instruction[i];
option.push_back(op);   // 存储操作码
```

4.2 编译模块

函数声明

```
// index表示指令位点， op表示预处理得到的操作码
void Instruction::compile(int index, string op);
```

重要变量

```
// 可能出现的各字段
string R1, R2, R3, str, code;
int r1, r2, r3, ofs;
// 寄存器实例，用于寄存器码辨识操作
Register r;
```

指令选择

函数内部通过辨析指令的操作码依次选择不同的指令进行编译，语句处理内部使用字符串搜索与匹配功能进行定位与截断，从而获取不同性质的数据，以下为编译样例：

`R` 类型指令编译（例：`sll rd rt sa`）

```
else if(op=="sll"){
    code="000000";              // 操作码
    int i = 0, j = 0;
    // rd
    i = sins[index].find('$');
    j = sins[index].find(',', i+1);
    R1 = sins[index].substr(i+1, j-i-1);
    r1 = r.regindex(R1);        // 获取rd寄存器索引
    // rs
    i = sins[index].find('$', j);
    j = sins[index].find(',', i+1);
    R2 = sins[index].substr(i+1, j-i-1);
    r2 = r.regindex(R2);        // 获取rs寄存器索引
    // sa
    str = sins[index].substr(j+1);
    int num = dectonum(str);    // 十进制字符串转整数

    code += numtobin(r2, 5);    // 整数转二进制5-bit字符串
    code += "00000";
    code += numtobin(r1, 5);
    code += numtobin(num, 5);
    code += "000000";
}
```

`I` 类型指令编译（例：`beq rs rt label`）

```
else if(op=="beq"){
    code="000100";              // 操作码
    int i = 0, j = 0;
    // rs
    i = sins[index].find('$');
    j = sins[index].find(',', i+1);
    R1 = sins[index].substr(i+1, j-i-1);
    r1 = r.regindex(R1);        // 获取rd寄存器索引
    // rt
    i = sins[index].find('$', j);
    j = sins[index].find(',', i+1);
    R2 = sins[index].substr(i+1, j-i-1);
    r2 = r.regindex(R2);        // 获取rs寄存器索引

    code += numtobin(r1, 5);
    code += numtobin(r2, 5);
    // ofs
    str = sins[index].substr(j+1);
    ofs = findlabel(str)-index; // 偏移
    code += numtobin(ofs, 16);
}
```

`J` 类型指令编译（例：`j label`）

```
else if(op=="j"){
    code="000010";
    str = sins[index].substr(1);     // 获取标签值
    ofs = findlabel(str);            // 偏移量
    code+=numtoubin(ofs, 26);
}
```

伪指令编译（例：`move rd rs`）

伪指令的编译思路是将伪指令转化为等效普通指令集合后，依次插入二进制指令向量中。

```
else if(op=="move"){            // 等效：or rd rs $zero
    code = "000000";
    int i = 0, j = 0;
    // rd
    i = sins[index].find('$');
    j = sins[index].find(',', i+1);
    R1 = sins[index].substr(i+1, j-i-1);
    r1 = r.regindex(R1);        // 获取rd寄存器索引
    // rs
    i = sins[index].find('$', j);
    j = sins[index].find(',', i+1);
    R2 = sins[index].substr(i+1, j-i-1);
    r2 = r.regindex(R2);        // 获取rs寄存器索引
    // zero
    R3 = "zero";
    r3 = r.regindex(R3);        // $zero

    code += numtobin(r2, 5);
    code += numtobin(r3, 5);
    code += numtobin(r1, 5);
    code += "00000100101";
}
```

4.3 反汇编模块

对二进制码进行切分，如果`op`为全0则做对R类型指令的`func`码进行识别并反汇编处理，否则根据不同`op`做相应类型的反汇编处理。

反汇编 （例：`andi rt rs ofs`）

```
else if(op == "001100"){//andi
    ins += "andi ";
    //rs
    R1 = bins[index].substr(6,5);
    R1 = "$"+reg.regname(bintounum(R1));    // 获取rs寄存器名称
    //rt
    R2 = bins[index].substr(11,5);
    R2 = "$"+reg.regname(bintounum(R2));    // 获取rt寄存器名称
    str = bins[index].substr(16);           // 获取偏移值
    temp = bintonum(str);

    stringstream ss;
    ss << temp;
```

```
    ss >> str;
    ins += R2 + "," + R1 + "," + str;       // andi rt, rs, ofs
}
```

4.4 内存模拟

读内存的操作通过直接返回对应内存位点的数据实现，并交由***MIPS***进一步处理，其中读取前16位或后16位则有对应程序二次分割处理。

```
string Memory::read(int pos){
    return memory[pos];     // 返回整字段
}
```

写内存的操作分为写入完整字段与写入半字段。

在***Memory***类中储存有`memory`与`record`，分别用于存储与标记操作，其中`record`内的值类型可以被标记为1或2，分别对应着储存有指令与储存有数据，该细节将在***Main***模块的指令方式/数据方式看内存中体现出作用。

```
void Memory::writedata(int pos, int type, string val)//ins:1 data:2
{
    if(type == 1){
        memory[ptr] = val;      // 存储字段
        record[ptr] = 1;        // 将该处标记为存储指令
        ptr++;                  // 指令指针步进
    }
    else{
        memory[pos] = val;
        record[pos] = 2;
    }
}
```

在写入半字段操作中，由于存在前后两部分写入选择，因此对不同模式下的插入进行了标记，存储于`location`布尔变量中。

```
void Memory::writehalf(int pos, bool location, string val)
{
    int value = bintonum(val);
    val = numtobin(value, 16);
    if(location == false) // left half
        memory[pos] = val + memory[pos].substr(16);
    else
        memory[pos] = memory[pos].substr(0,16) + val;
    record[pos] = 2;
}
```

5 使用手册与使用实例

本部分将使用如下程序对MIPS模拟机的各项功能进行验证:

```
31
cal:    addi $t0,$zero,28
        addi $t1,$zero,-1
        add $t2,$t0,$t1
        sub $t3,$t0,$t1
        mul $t4,$t0,$t1
slide:  sll $t7,$t1,1
        srl $t8,$t1,1
        sra $t9,$t1,1
logic:  and $t1,$t0,$zero
        andi  $t2,$t0,-1
        or $t3,$t0,$zero
        ori  $t4,$t0,-1
        xor $t5,$t0,$zero
        xori  $t6,$t0,-1
        nor $t7,$t0,$zero
        slt $t8,$t0,$t1
        j jump
test1:  sub $t2,$zero,$zero
        sub $t3,$zero,$zero
jump:   beq $t2,$t3,load
        bne $t2,$t3,exit
load:   sw $t0,40($t1)
        lw $s0,40($t1)
        addi $t1,$t1,30
        jr $t1
test2:  addi $s6,$zero,30000
        sw $t1,0($s6)
        lw $a0,0($s6)
        addi $v0,$zero,1
psuedo: swap $t0,$t1
        move $t1,$t0
```

5.0 启动系统

启动工程文件或可执行文件，根据当前汇编码，选择进入汇编模式：

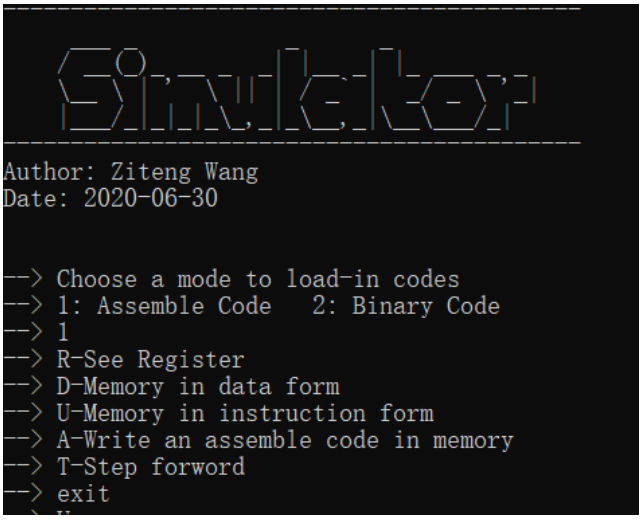

图5.0 MIPS Simulator启动界面

5.1 指令方式看内存

命令行中输入`U`指令，以指令的方式查看当前内存。

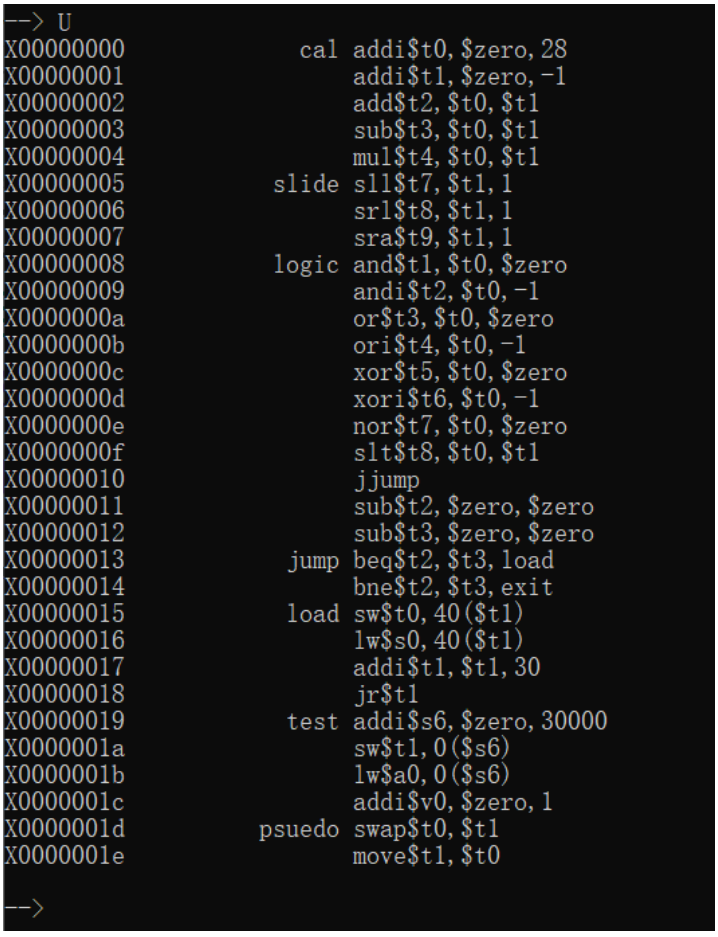

图5.1 MIPS Simulator指令方式读内存

对比可看出所有指令均正确分割并有序插入。

5.2 数据方式看内存

命令行中输入`D`指令，以数据的方式查看当前内存。

```
--> D
X00000000       00100000000010000000000000011100
X00000001       00100000000010011111111111111111
X00000002       00000001000010010101000000100000
X00000003       00000001000010010101100000100010
X00000004       01110001000010010110000000000010
X00000005       00000001001000000111100001000000
X00000006       00000001001000001100000001000010
X00000007       00000001001000001100100001000011
X00000008       00000001000000000100100000100100
X00000009       00110001000010101111111111111111
X0000000a       00000001000000000101100000100101
X0000000b       00110101000011001111111111111111
X0000000c       00000001000000000110100000100110
X0000000d       00111001000011101111111111111111
X0000000e       00000001000000000111100000100111
X0000000f       00000001000010011100000000101010
X00000010       00001000000000000000000000010011
X00000011       00000000000000000101000000100010
X00000012       00000000000000000101100000100010
X00000013       00010001010010110000000000000010
X00000014       00010101010010111111111111101011
X00000015       10101101000010010000000000101000
X00000016       10001110000010010000000000101000
X00000017       00100001001010010000000000011110
X00000018       00000001001000000000000000001000
X00000019       00100000000101100111010100110000
X0000001a       10101101001101100000000000000000
X0000001b       10001100100101100000000000000000
X0000001c       00100000000000100000000000000001
X0000001d       00000001000010010100000000100110
X0000001e       00000001000010010100100000100110

-->
```

图5.2 MIPS Simulator数据方式读内存

对比可看出所有指令均正确编译并成功录入。

5.3 Debug模式

5.3.1 Calculator计算模式

首先单步完成前五条指令，并根据寄存器状态确定指令执行情况：

```
--> T
PC: 0

--> T
PC: 1

--> T
PC: 2

--> T
PC: 3

--> T
PC: 4

--> R
| Register |  Value | Register |  Value | Register |  Value | Register |  Value |
|    $zero |      0 |      $at |      0 |      $v0 |      0 |      $v1 |      0 |
|      $a0 |      0 |      $a1 |      0 |      $a2 |      0 |      $a3 |      0 |
|      $t0 |     28 |      $t1 |     -1 |      $t2 |     27 |      $t3 |     29 |
|      $t4 |    -28 |      $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 |

-->
```

图5.3.1 MIPS Simulator计算模式运行结果

根据汇编指令与对应的执行情况，通过寄存器状态的变化可得出，所有计算指令均成功运行。

5.3.2 Slide移位模式

单步完成接下来的三步操作，分别为逻辑左移、逻辑右移、算术右移：

```
--> T
PC: 5

--> T
PC: 6

--> T
PC: 7

--> R
| Register|   Value| Register|   Value| Register|   Value| Register|   Value|
|    $zero|       0|      $at|       0|      $v0|       0|      $v1|       0|
|      $a0|       0|      $a1|       0|      $a2|       0|      $a3|       0|
|      $t0|      28|      $t1|      -1|      $t2|      27|      $t3|      29|
|      $t4|     -28|      $t5|       0|      $t6|       0|      $t7|      -2|
|      $s0|       0|      $s1|       0|      $s2|       0|      $s3|       0|
|      $s4|       0|      $s5|       0|      $s6|       0|      $s7|       0|
|      $t8|2147483647|      $t9|      -1|      $k0|       0|      $k1|       0|
|      $gp|       0|      $sp|       0|      $fp|       0|      $ra|       0|

-->
```

图5.3.2 MIPS Simulator移位模式运行结果

其中`$t8`存储了$2^{31}-1$，表明-1的逻辑右移会导致整体符号位右移，发生了溢出。

根据汇编指令与对应的执行情况，通过寄存器状态的变化可得出，所有移位指令均成功运行。

5.3.3 Logic逻辑运算模式

单步执行之后的八条指令，并观察`$t1`~`$t8`的变化：

```
--> T
PC: 8

--> T
PC: 9

--> T
PC: 10

--> T
PC: 11

--> T
PC: 12

--> T
PC: 13

--> T
PC: 14

--> T
PC: 15

--> R
| Register | Value | Register | Value | Register | Value | Register | Value |
|    $zero |     0 |      $at |     0 |      $v0 |     0 |      $v1 |     0 |
|      $a0 |     0 |      $a1 |     0 |      $a2 |     0 |      $a3 |     0 |
|      $t0 |    28 |      $t1 |     0 |      $t2 |    28 |      $t3 |    28 |
|      $t4 |    -1 |      $t5 |    28 |      $t6 |   -29 |      $t7 |   -29 |
|      $s0 |     0 |      $s1 |     0 |      $s2 |     0 |      $s3 |     0 |
|      $s4 |     0 |      $s5 |     0 |      $s6 |     0 |      $s7 |     0 |
|      $t8 |     0 |      $t9 |    -1 |      $k0 |     0 |      $k1 |     0 |
|      $gp |     0 |      $sp |     0 |      $fp |     0 |      $ra |     0 |

-->
```

图5.3.3 MIPS Simulator逻辑运算模式运行结果

根据汇编指令与对应的执行情况，通过寄存器状态的变化可得出，所有逻辑运算指令均成功运行。

5.3.4 跳转模式

单步执行`j jump`与`beq$t2,$t3,load`指令，观察PC变化：

```
--> T
PC: 16

--> T
PC: 19

--> R
| Register | Value | Register | Value | Register | Value | Register | Value |
|    $zero |     0 |      $at |     0 |      $v0 |     0 |      $v1 |     0 |
|      $a0 |     0 |      $a1 |     0 |      $a2 |     0 |      $a3 |     0 |
|      $t0 |    28 |      $t1 |     0 |      $t2 |    28 |      $t3 |    28 |
|      $t4 |    -1 |      $t5 |    28 |      $t6 |   -29 |      $t7 |   -29 |
|      $s0 |     0 |      $s1 |     0 |      $s2 |     0 |      $s3 |     0 |
|      $s4 |     0 |      $s5 |     0 |      $s6 |     0 |      $s7 |     0 |
|      $t8 |     0 |      $t9 |    -1 |      $k0 |     0 |      $k1 |     0 |
|      $gp |     0 |      $sp |     0 |      $fp |     0 |      $ra |     0 |

--> T
PC: 21
```

图5.3.4 MIPS Simulator跳转模式运行结果

根据汇编指令与对应的执行情况，结合`$t2`与`$t3`相等的情况，可得知所有跳转操作均成功执行。

5.3.5 读写内存模式

单步执行后两步操作`sw $t0,40($t1)`与`lw $s0,40($t1)`，观察寄存器和内存变化：

```
--> T
PC: 22

--> T
PC: 23

--> R
| Register |  Value | Register |  Value | Register |  Value | Register |  Value |
|    $zero |      0 |      $at |      0 |      $v0 |      0 |      $v1 |      0 |
|      $a0 |      0 |      $a1 |      0 |      $a2 |      0 |      $a3 |      0 |
|      $t0 |     28 |      $t1 |     30 |      $t2 |     28 |      $t3 |     28 |
|      $t4 |     -1 |      $t5 |     28 |      $t6 |    -29 |      $t7 |    -29 |
|      $s0 |     28 |      $s1 |      0 |      $s2 |      0 |      $s3 |      0 |
|      $s4 |      0 |      $s5 |      0 |      $s6 |      0 |      $s7 |      0 |
|      $t8 |      0 |      $t9 |     -1 |      $k0 |      0 |      $k1 |      0 |
|      $gp |      0 |      $sp |      0 |      $fp |      0 |      $ra |      0 |
```

图5.3.5-1 MIPS Simulator读写内存模式寄存器运行结果

```
--> D
X00000000       00100000000010000000000000011100
X00000001       00100000000010011111111111111111
X00000002       00000010000100101010000000100000
X00000003       00000010000100101011000000100010
X00000004       01110001000010010110000000000010
X00000005       00000001001000000111100001000000
X00000006       00000001001000001100000001000010
X00000007       00000001001000001100100001000011
X00000008       00000001000000000100100000100100
X00000009       00110001000010101111111111111111
X0000000a       00000001000000000101100000100101
X0000000b       00110101000011001111111111111111
X0000000c       00000001000000000110100000100110
X0000000d       00111001000011101111111111111111
X0000000e       00000001000000000111100000100111
X0000000f       00000001000010011100000000101010
X00000010       00001000000000000000000000010011
X00000011       00000000000000000101000000100010
X00000012       00000000000000000101100000100010
X00000013       00010001010010110000000000000010
X00000014       00010101010010111111111111101011
X00000015       10101101000010010000000000101000
X00000016       10001110000010010000000000101000
X00000017       00100001001010010000000000011110
X00000018       00000001001000000000000000001000
X00000019       00100000000101100111010100110000
X0000001a       10101101001101100000000000000000
X0000001b       10001100100101100000000000000000
X0000001c       00100000000000100000000000000001
X0000001d       00000001000010010100000000100110
X0000001e       00000001000010010100100000100110
X00000028       00000000000000000000000000011100

-->
```

图5.3.5-2 MIPS Simulator读写内存模式内存运行结果

根据汇编指令与对应的寄存器、内存执行情况，可得知所有读写内存操作均成功执行。

5.3.6 伪代码模式

单步执行后两步操作`sw $t0,40($t1)`与`lw $s0,40($t1)`，观察寄存器和内存变化：

```
--> T
PC: 32

--> R
| Register|   Value| Register|   Value| Register|   Value| Register|   Value|
|    $zero|       0|      $at|       0|      $v0|       0|      $v1|       0|
|      $a0|       0|      $a1|       0|      $a2|       0|      $a3|       0|
|      $t0|      28|      $t1|      28|      $t2|      28|      $t3|      28|
|      $t4|      -1|      $t5|      28|      $t6|     -29|      $t7|     -29|
|      $s0|      28|      $s1|       0|      $s2|       0|      $s3|       0|
|      $s4|       0|      $s5|       0|      $s6|       0|      $s7|       0|
|      $t8|       0|      $t9|      -1|      $k0|       0|      $k1|       0|
|      $gp|       0|      $sp|       0|      $fp|       0|      $ra|       0|

--> T
You Have Run All Codes
```

图5.3.6 MIPS Simulator伪代码模式内存运行结果

根据汇编指令与对应的执行情况，通过寄存器状态的变化可得出，所有伪代码指令均成功运行。

到此完成了所有的测试。

6 开发心得与展望

本次***MIPS***模拟机的开发过程中，遇到了很多问题，也有了很多感想，分为以下几条：

1. **系统的架构**

   项目最开始的架构往往决定了开发的质量，本着还原、仿真、模拟的原则，我在设计本模拟机的过程中尽可能根据计算机指令运行的真实架构进行了构思，仿照数据通路的接口，构建了指令集、寄存器组、模拟内存、模拟机**API**等模块，在降低耦合度的同时，能够集中精力对单一部分进行完善。

2. **指令的执行**

   这一部分的位置安排和执行原理都有值得推敲的地方，起初我打算将这一部分在***Instruction***指令模块中实现，从而构成指令的闭合操作，但在实现中却遇到了问题：由于指令的执行涉及到对寄存器、内存模块的访问操作，如果在指令集合中新建两个类的实例则会在实际使用过程中与***MIPS***模块发生冲突（存在两个正交的实例）。经过重新分析架构，我认为将指令的执行安排在***MIPS***类内更加合适，在真实的计算过程中，指令在**CPU**的操作便是作为不同模块间的中央单元，因此**软CPU**的实现也应当遵循同样的规律。

3. **指令的解析**

   指令的解析也是本部分的重点，如何在编译过程中妥善处理标签的位点、指令的区分、寄存器或立即数的识别等问题是这一部分的核心问题，在指令的处理过程中，由于标签的格式多样化，我使用了正则表达式进行标签的识别，并将其存储在**Label**向量中，对于每一条指令，分别分配一个**标签**、**文本指令**、**二进制指令**的存储空间，在解析时填写标签，编译时填写二进制指令，反汇编时填写文本指令等，至此可将指令字段化，并在函数中通过定位关键字符如‘$’、‘,’以获取相应开始和结束位点，并使用字符串的截取和相应的自定义函数进行解析。

在今后的学习工作中，我认为本项目仍有许多可改进的地方，如将用户界面图形化，指令丰富化等，在将来有机会时可以将其一一实现。

7 小组成员与分工

王子腾：***MIPS***模拟器代码实现，功能测试，报告撰写

徐晓丹：***MIPS***汇编过程思路

马静怡：无