



《计算机系统原理》实验报告

作业名称: 模拟器设计

小组成员: _____.

产品经理: _____.

指导老师: 楼学庆

2020-6-30

一 . 引言

本程序可以模拟一个 MIPS 虚拟机，有内存和寄存器文件，可以运行 MIPS 指令，并将结果反馈在内存和寄存器数据的变化上。

支持的指令有：42 条基本指令，包括：

R 型指令	add, sub, slt, and, or, xor, nor, sll, sllv, srl, srlv, sra, srav, jr, jalr, syscall, addu, subu,
I 型指令	lui, addi, sltu, sltiu, andi, ori, xori, lw, lwx, lh, lhx, lhu, lhux, sw, swx, sh, shx, beq, bne, bgezal, addiu, slti,
J 型指令	j, jal
C 型指令	

二 . 基本实验原理

程序设计主要基于 MIPS 汇编相关知识。

2.1 数据存储模式

采用大端规则进行存储，也就是一条长数据在存储时，低位地址存储高位数据。

2.2 通用寄存器

MIPS 架构有 32 个通用寄存器，将 32 个寄存器按 0-31 编号，编号顺序如下表所示。

Name	Register Number	Description
\$zero	0	始终为 0
\$at	1	为汇编保留，主要用于伪指令拓展
\$v0-\$v1	2-3	子程序返回
\$a0-\$a3	4-7	子程序调用参数

四．详细设计说明

4.1 主要类

指令类：

```
class Instr{
public:
    string all;
    vector<string> args;
public:
    Instr();
    ~Instr();
    Instr(int bin);
    Instr(string s);
    int toBin();
};
```

其中，成员变量 all 是指令的完整内容，args 是指令的操作符以及参数。

可以通过一条指令字符串构造指令，也可以通过将一个 32 位机器码进行反汇编来构造指令，toBin 方法可以提供指令经过汇编后的 32 位机器码。

模拟器类：

```
class Sim{
private:
    unsigned char Memory[MEM_SIZE];
    int regs[REG_NUM];
    int PC;
    int entry;

public:
    Sim();
    ~Sim();

    void Start();
    void LoadInstr(vector<int> bin32s, int ofs);
    void LoadBin(vector<unsigned char> bin8s, int ofs);

    void Step();
    void Run();
    void ShowStatus();
    void PrintRegs();
};
```

```

    void Again();
    void ShutDown();

    Instr FetchInstr();
    Instr GetNextInstr();

    bool Execute(Instr ins);
    bool executeR1(Instr ir);
    bool executeR2(Instr ir);
    bool executeR3(Instr ir);
    bool executeI1(Instr ir);
    bool executeI2(Instr ir);
    bool executeI3(Instr ir);
    bool executeJ(Instr ir);
    bool executeC(Instr ir);
};

```

其中，成员变量 `Memory` 代表内存，可以加载指令和数据，内存寻址单位为一个字节（8 位二进制码），一条指令占用四个寻址单位。成员变量 `regs` 指寄存器文件，每个寄存器均为 32 位。成员变量 `PC` 是程序计数器，在类中是 `Memory` 数组的索引下标，始终指向将要运行的下一条指令。成员变量 `entry` 保存了程序的入口，便于多次运行程序。

`Start` 方法可以开启模拟器，用户可以开始输入命令对模拟器进行操作，支持的操作有：加载指令（由 `loadInstr` 方法实现）或者数据（由 `LoadBin` 方法实现），运行程序（由 `Run` 方法实现），单步运行（由 `Step` 方法实现），重新开始或者回到程序起点（由 `Again` 方法实现），显示寄存器状态（由 `ShowStatus` 方法实现），以及退出模拟器（由 `ShutDown` 方法实现）。

`ShowStatus` 方法在显示寄存器状态的时候，还会显示下一条要运行的指令，由 `GetNextInstr` 方法获得下一条指令，即 `PC` 所指向的指令的内容。

单步执行时，由 `FetchInstr` 方法获取下一条要执行的指令并使 `PC` 自增，然后用 `Execute` 方法执行指令，`Execute` 方法会先判断指令的类型，然后再调用不同类型的方法去执行指令。

从 `Memory` 中获取到的指令是 32 位的二进制机器码，要将其先转化为标准的 MIPS 指令形式，这里涉及到了反汇编的相关模块。

`Run` 方法通过一直调用 `FetchInstr` 直到程序结束，来一次性运行所有指令。

4.2 汇编模块

汇编模块主要用于处理指令，比如将指令加载进内存的时候，要先将指令翻译成 32 位的机器码。

```

int translate(vector<string> opm);

```

```
int funcNum(string op);
int translateR(int op, vector<string> instr);
int translateI(int op, vector<string> instr);
int translateJ(int op, vector<string> instr);
int translateC(int op, vector<string> instr);

int isRIJC(string op);
int opcode(string op);
```

translate 函数将从指令中提取出来的参数进行处理，判断指令的类型，然后调用其他函数进行翻译，其中 translateR 函数用来翻译 R 型指令，translateI 函数用来翻译 I 型的指令，translateJ 函数用来翻译 J 型指令，而 translateC 函数用来翻译 C 型指令。

isRIJC 函数用来判断指令的类型，funcNum 函数用来获取指令中 function 字段的内容，opcode 函数用来获取指令的操作码。

4.3 反汇编模块

反汇编模块主要用于处理机器码，比如在运行指令时需要先从内存中获取机器码，然后翻译成一般规范的 MIPS 指令，再执行。

```
string reasmC(int code);
string reasmJ(int code);
string reasmR(int code);
string reasm(int code);
```

函数 reasm 用来将一条 32 位的机器码反汇编为一条 MIPS 指令，它会先判断指令的类型，然后调用其函数，reasmR 函数可以将对 R 类型的指令进行反汇编操作，reasmJ 函数可以对 J 类型的指令进行反汇编操作，而 reasmC 函数可以对 C 类型的指令进行反汇编操作，对 I 类型指令的反汇编操作被兼容再 reasm 函数中。

进行反汇编处理的时候，函数会先确定指令的类型从而确定指令的机器码中包含哪些字段，指令有哪些参数，然后从机器码中获取个字段，并将其转化为指令的参数或者操作符，再进行组合，最后得到一条一般规范的 MIPS 指令（字符串的形式）。

4.4 其他

还有其他的一些函数和变量，主要用于辅助处理，比如指令翻译前的先行处理，使指令格式化以便于翻译。

```
map<int, string> regName = {
    {0, "$zero"}, {1, "$at"}, {2, "$v0"}, {3, "$v1"},
    {4, "$a0"}, {5, "$a1"}, {6, "$a2"}, {7, "$a3"},
```

```

        {8, "$t0"}, {9, "$t1"}, {10, "$t2"}, {11, "$t3"},
        {12, "$t4"}, {13, "$t5"}, {14, "$t6"}, {15, "$t7"},
        {16, "$s0"}, {17, "$s1"}, {18, "$s2"}, {19, "$s3"},
        {20, "$s4"}, {21, "$s5"}, {22, "$s6"}, {23, "$s7"},
        {24, "$t8"}, {25, "$t9"}, {26, "$k0"}, {27, "$k1"},
        {28, "$gp"}, {29, "$sp"}, {30, "$fp"}, {31, "$ra"},
    };

    map<string,int> regIndex = {
        {"zero", 0}, {"$at", 1}, {"$v0", 2}, {"$v1", 3},
        {"$a0", 4}, {"$a1", 5}, {"$a2", 6}, {"$a3", 7},
        {"$t0", 8}, {"$t1", 9}, {"$t2", 10}, {"$t3", 11},
        {"$t4", 12}, {"$t5", 13}, {"$t6", 14}, {"$t7", 15},
        {"$s0", 16}, {"$s1", 17}, {"$s2", 18}, {"$s3", 19},
        {"$s4", 20}, {"$s5", 21}, {"$s6", 22}, {"$s7", 23},
        {"$t8", 24}, {"$t9", 25}, {"$k0", 26}, {"$k1", 27},
        {"$gp", 28}, {"$sp", 29}, {"$fp", 30}, {"$ra", 31},
    };

    unsigned char hexstr2int8(string s);
    string prepare(string s);
    vector<string> strSplit(string s,char c);
    string strInBrackets(string s);

```

两个 map 分别储存了寄存器编号和寄存器名之间的对应关系，便于查找。

hexstr2int8 函数可以把一个字符串转化为一个 8 位的二进制数(unsigned char)，主要在加载数据的时候用到。

prepare 函数可以对一条指令字符串进行预处理，去掉多余的空白符，将参数用逗号分隔。

strSplit 函数可以将一个字符串按照某个字符分成许多段，可以处理经过 prepare 函数处理的字符串以得到指令的操作符以及参数。

strInBrackets 函数可以获取一个含有括号的字符串中括号内的子串，在指令中可能会有参数位于括号中，可以通过这个函数进行处理。

4.5 指令执行处理

指令的执行主要通过操作寄存器和内存来体现。

```

bool Execute(Instr ins);
bool executeR1(Instr ir);
bool executeR2(Instr ir);

```

```
bool executeR3(Instr ir);
bool executeI1(Instr ir);
bool executeI2(Instr ir);
bool executeI3(Instr ir);
bool executeJ(Instr ir);
bool executeC(Instr ir);

int RIJC(string op);
```

RIJC 函数为辅助函数，用来判断指令的具体类型，不只限于判断是 R、I、J、C 四种类型中的哪一种。比如如果是 R 类型指令，还判断是算术运算指令还是逻辑运算指令，还是跳转指令或者系统调用，如果是 I 类型的指令，还会判断是立即数运算指令，还是数据读写指令，还是跳转指令。

执行指令时，函数会根据指令中的参数对相应的寄存器和内存块进行操作，如果涉及到了跳转指令，函数会改写 PC 的值。

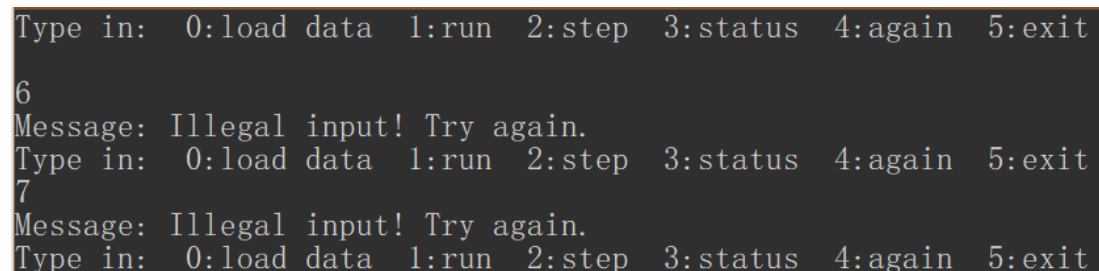
五．使用说明

5.1 编译及运行

可以直接运行 make.bat 文件对源码进行编译，然后运行生成的可执行文件，也可以直接运行 main.exe，这两种选择的结果并不会有什么不同。

5.2 基本操作及样例

运行程序，会提示输入命令，如图所示，0 代表加载数据，1 代表运行程序，2 代表单步执行程序，3 代表显示寄存器状态，4 代表回到程序入口，5 代表退出。



```
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
6
Message: Illegal input! Try again.
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
7
Message: Illegal input! Try again.
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
```

如果输入了其他的命令，则会被模拟器判定为不合法输入，并再次提示输入。

```

Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
3
$zero: 0x00000000    $t0: 0x00000000    $s0: 0x00000000    $t8: 0x00000000
  $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
  $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
  $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
  $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
  $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
  $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
  $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: addi $s1,$zero,3
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit

```

显示寄存器时下方还会有下一条指令的内容，便于调试。

一条命令执行结束后模拟器会提醒输入下一条命令，直到退出。

```

Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
2
->step
$zero: 0x00000000    $t0: 0x00000000    $s0: 0x00000000    $t8: 0x00000000
  $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000003    $t9: 0x00000000
  $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
  $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
  $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
  $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
  $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
  $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: xor $t0,$t0,$t0
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit

```

选择单步执行命令时会出现命令回显，执行结束后也会有更新的寄存器状态和下一条指令内容。

```

Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
1
->run
Finished
$zero: 0x00000000    $t0: 0x00000006    $s0: 0x00000006    $t8: 0x00000000
  $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
  $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
  $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
  $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
  $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
  $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
  $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: sll $zero,$zero,0
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit

```

选择运行命令后，也会出现命令回显，程序顺利结束时会有“Finished”提示，并显示更新后的寄存器状态，此时下一条指令为无效指令。

```

Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
4
Message: PC has been reset successfully!
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
1
->run
Finished
$zero: 0x00000000    $t0: 0x00000006    $s0: 0x00000006    $t8: 0x00000000
    $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
    $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
    $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
    $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
    $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
    $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
    $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: sll $zero,$zero,0
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit

```

选择重新开始命令后，会出现 PC 被重置的消息提示，此时可再次运行程序。

```

Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
4
Message: PC has been reset successfully!
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
3
$zero: 0x00000000    $t0: 0x00000006    $s0: 0x00000006    $t8: 0x00000000
    $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
    $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
    $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
    $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
    $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
    $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
    $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: addi $s1,$zero,3
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit

```

可以看到，PC 被重置后，下一条指令的内容变成了程序入口处的指令。

还可以选择加载新的数据，先选择要加载数据的类型，数据类型可以是指令，也可以是二进制码，当选择指令时，先输入要将指令加载的位置，再输入想要加载的指令（可以输入多条指令）并以一个单独的`#`结束，就可以实现指令的加载。

指令加载成功后会出现提示信息，在图中可以看到，当在程序入口处加载了一条指令后，再次显示模拟器状态，此时程序入口处第一条指令已经变成了刚加载的指令。

```

The next instruction: addi $s1,$zero,3
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
0
Data type: 0:Bin  1:Instruction
1
Position: 0
Instructions end with "#":
addi $s1, $zero, 4
#
Message: Successfully loaded!
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
3
$zero: 0x00000000    $t0: 0x00000000    $s0: 0x00000000    $t8: 0x00000000
    $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
    $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
    $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
    $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
    $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
    $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
    $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: addi $s1,$zero,4
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit

```

再次运行，得到了新的结果（在寄存器\$s0）中。

```
The next instruction: addi $s1,$zero,4
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
1
->run
Finished
$zero: 0x00000000    $t0: 0x0000000A    $s0: 0x0000000A    $t8: 0x00000000
  $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
  $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
  $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
  $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
  $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
  $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
  $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: sll $zero,$zero,0
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
```

如果选择加载二进制码，先输入要加载的位置，再依次输入 16 进制数（每个不小于 0，不大于 255），以回车分隔，以一个单独的 '#' 结束，在图中可以看到，加载后下一条要执行的指令变成了刚加载的 4 个 8 位二进制码所表示的指令。

```
The next instruction: addi $s1,$zero,3
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
0
Data type: 0:Bin  1:Instruction
0
Position: 0
Hexadecimal int8s end with "#":
20
11
00
05
#
Message: Successfully loaded!
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
3
$zero: 0x00000000    $t0: 0x00000000    $s0: 0x00000000    $t8: 0x00000000
  $at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
  $v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
  $v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
  $a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
  $a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
  $a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
  $a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: addi $s1,$zero,5
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
```

如果在选择数据类型的时候输入了其他数字，就会提示输入不合法，然后直接结束当前的数据加载处理。

```
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
0
Data type: 0:Bin  1:Instruction
3
Position: 3
Message: Illegal input.
Type in:  0:load data  1:run  2:step  3:status  4:again  5:exit
-
```

选择退出，程序结束。

```

Type in: 0:load data 1:run 2:step 3:status 4:again 5:exit
1
->run
Finished
$zero: 0x00000000    $t0: 0x0000000A    $s0: 0x0000000A    $t8: 0x00000000
$at: 0x00000000    $t1: 0x00000000    $s1: 0x00000000    $t9: 0x00000000
$v0: 0x00000000    $t2: 0x00000000    $s2: 0x00000000    $k0: 0x00000000
$v1: 0x00000000    $t3: 0x00000000    $s3: 0x00000000    $k1: 0x00000000
$a0: 0x00000000    $t4: 0x00000000    $s4: 0x00000000    $gp: 0x00000000
$a1: 0x00000000    $t5: 0x00000000    $s5: 0x00000000    $sp: 0x00000000
$a2: 0x00000000    $t6: 0x00000000    $s6: 0x00000000    $fp: 0x00000000
$a3: 0x00000000    $t7: 0x00000000    $s7: 0x00000000    $ra: 0x00000000
The next instruction: sll $zero,$zero,0
Type in: 0:load data 1:run 2:step 3:status 4:again 5:exit
5
Bye~

```

在以上的示例中，程序预先加载了一个可以求前 n 项自然数和的程序，第一条指令设置输入，即 n 的值，保存在寄存器 $s1$ 中，最后的结果会被保存在寄存器 $s0$ 中。

六．总结与感悟

基本实现了模拟器要求的功能，但是也有一些遗憾，比如对伪指令的支持不够完善，没有给程序一个比较好看的图形界面。

在设计的过程中，不仅用到了 MIPS 相关的内容，也用到了 CPU 的一些知识，课堂上的内容得到了复习和巩固，我们对计算机底层原理的理解也更加深刻了。