MIPS模拟器

一、项目需求分析

1. 项目背景

本项目中，我们需要开发一个MIPS体系结构模拟器，实现汇编、反汇编、系统状态模拟、整数（补码）和浮点数的表示、转换和运算。该模拟器的开发可以帮助我们更好地理解MIPS体系结构的工作原

理。

2. 功能需求

- **汇编：**模拟器需要支持MIPS指令集的汇编功能，即将MIPS汇编代码转换为机器码。至少需要实现10种指令，例如add、and、xor、jal、beq等。
- **反汇编：**模拟器需要支持将机器码反汇编为MIPS汇编代码。
- **系统状态模拟：**模拟器需要模拟MIPS体系结构的系统状态，包括寄存器、内存等。需要能够显示当前状态，包括寄存器内容和内存内容。
- **整数（补码）的表示、转换和运算：**模拟器需要支持整数（补码）的表示、转换和运算功能，包括加、减、乘、除等运算。
- **浮点数的表示、转换和运算：**模拟器需要支持浮点数的表示、转换和运算功能，包括单精度和双精度浮点数的表示和运算。

3.非功能需求

- **用户友好性：**模拟器需要提供良好的用户体验，具有友好的界面和操作方式，方便用户进行操作。
- **高效性：**模拟器需要具有高效性能，能够快速地处理大规模的代码和数据。
- **可扩展性：**模拟器需要具有良好的可扩展性，可以方便地扩展新的指令和功能。

4.开发环境

- **开发语言：**本MIPS模拟器采用C++作为开发语言。C++是一种计算机高级程序设计语言，其既可以进行C语言的过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。
- **开发工具：**本项目采用开源的开发IDE， Visual Studio Code进行开发。 Visual Studio Code是Microsoft在2015年4月30日Build开发者大会上正式宣布一个运行于 Mac OS X、Windows和 Linux 之上

的，针对于编写现代Web和云应用的跨平台源代码编辑器，可在桌面上运行，并且可用于Windows，macOS和Linux。它具有对JavaScript，TypeScript和Node.js的内置支持，并具有丰富的其他语言（例如C++，C＃，Java，Python，PHP，Go）和运行时（例如.NET和Unity）扩展的生态系统。

二、总体设计

本MIPS模拟器总共实现10个MIPS指令：addi, andi, ori, xori, beq, jal, jr, add, and, slt

1.总体设计架构

程序入口：

程序从 Main 函数开始执行，其中创建了一个 UserInterface 类型的对象 ui，并通过调用 ui.run() 方法启动了用户界面。

用户界面：

UserInterfaceUserInterface 类实现了用户交互界面，包含了以下几个功能：

- 打印用户界面：printUI() 方法，根据用户当前的状态打印出相应的用户界面。
- 将汇编代码转换为二进制代码：AsmConvert() 方法，将输入的汇编代码转换为二进制代码，并打印输出。
- 将二进制代码转换为汇编代码：BiConvert() 方法，将输入的二进制代码转换为汇编代码，并打印输出。
- 读入并执行二进制代码：VmExecute() 方法，读入用户输入的二进制代码，并执行模拟器运行。
- 整数/浮点数转换：创建相应的UI界面（ConvertInterface类）实现整数浮点数转换表示、转换和运算

模拟器：

RegAndMemory 类实现了模拟器的功能

包含以下几个方法：

- 将二进制代码装载到模拟器内存：LoadCode() 方法，将输入的二进制代码装载到模拟器内存中。
- 执行指令：Execute() 方法，模拟器根据当前 PC 寄存器指向的指令执行相应的操作，并更新 PC 寄存器。
- 打印寄存器和内存状态：toString() 方法，将当前模拟器的寄存器和内存状态以字符串的形式返回。

汇编/反汇编：

Instruction类实现了汇编/反汇编的功能

通过AsmConvert() 方法以及BiConvert() 方法读入用户输入的指令，构建Instruction实体，通过调用Instruction类内成员函数实现汇编反汇编的转换。

整数的表示、转换和运算：

IntTyper类实现了整数的表示、转换和运算

其包含以下几个功能：

- 将二进制字符串转化为数值：BiStringToInt()方法，将输入的比特字符串转化为整数数值表达。
- 将整数转化为其二进制表示：IntToBiString()，将输入的整数数值转化为比特字符串表达。
- 整数加法：add()，实现整数之间的加法运算。
- 整数减法：sub()，实现整数之间的减法运算。
- 整数乘法：mul()，实现整数之间的乘法运算。
- 整数除法：div()，实现整数之间的除法运算。

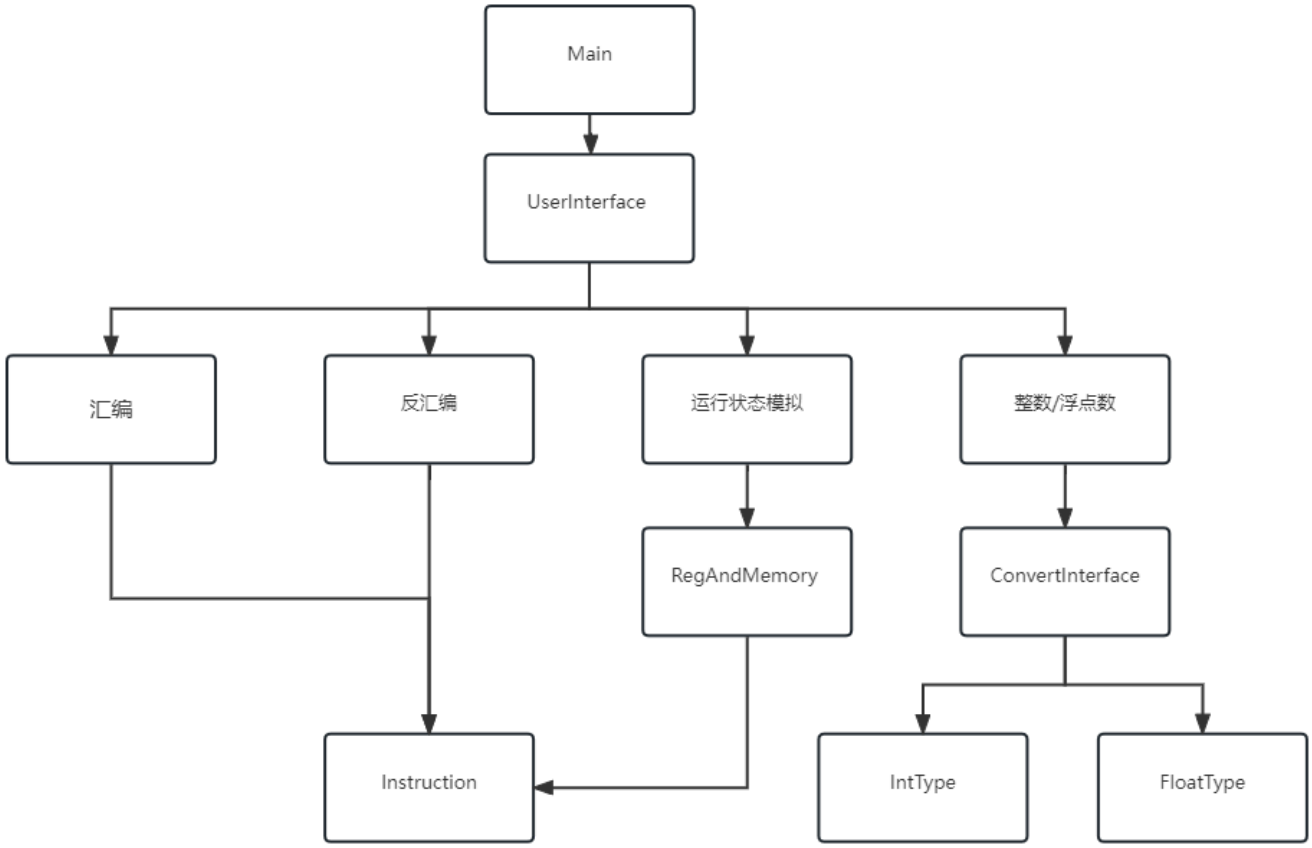
浮点数的表示、转换和运算：

FloatTyper类实现了浮点数的表示、转换和运算

其包含以下几个功能：

- 将二进制字符串转化为数值：BiStringToFloat()方法，将输入的比特字符串转化为整数数值表达。
- 将浮点数转化为其二进制表示：FloatToBiString()，将输入的整数数值转化为比特字符串表达。
- 浮点数加法：add()，实现浮点数之间的加法运算。
- 浮点数减法：sub()，实现浮点数之间的减法运算。
- 浮点数乘法：mul()，实现浮点数之间的乘法运算。
- 浮点数除法：div()，实现浮点数之间的除法运算。

2.总体设计概括图



三、具体设计

1. RegAndMemory类

RegAndMemory类是用于模拟寄存器和内存的操作。

成员变量

reg：一个大小为 REG_NUM(=32) 的 uint32_t 数组，用来模拟MIPS中的32个寄存器；
memo：一个大小为 MEMO_SIZE 的 char 数组，用来模拟内存；
pc：一个 uint32_t 类型的变量，用来记录程序计数器的值。

成员函数

void LoadCode(vector<uint32_t> code)：用于将一段指令代码加载到模拟器的内存中。它接收一个 vector<uint32_t> 类型的参数 code，代表一段指令代码。然后，将这段指令代码按照 4 字节为单位存储到内存中。

int Execute()：用于执行一段指令代码。这个函数返回一个 int 类型的值，代表执行结果。
void SetUint32(uint32_t value, uint32_t address)：用于将一个 uint32_t 类型的值存储到指定的内存地址中。它接收两个参数，分别是要存储的值 value 和要存储到的内存地址 address。
uint32_t GetUint32(uint32_t address)：用于从指定的内存地址中读取一个 uint32_t 类型的值。它接收一个参数 address，代表要读取的内存地址。
string toString()：用于将模拟器的状态以字符串形式返回。这个函数会返回一个 string 类型的值，代表模拟器的状态。具体来说，它会输出每个寄存器的二进制、十进制和十六进制表示，以及内存中的数据。其中，如果当前的地址是程序计数器的值，那么输出的数据前面会带有 << 符号。

总体来说，RegAndMemory类提供了模拟寄存器和内存的功能，同时提供了将指令代码加载到内存中、执行指令代码、读取和写入内存数据的接口。通过这些接口，我们可以使用这个类来模拟 MIPS 体系结构。

2.Instruction类

Instruction的类是用来代表了MIPS汇编程序中的一条指令。

成员变量

target:指向RegAndMemory对象的指针。
map_of_reg_num:一个静态std::map，将寄存器名(std::string)映射为寄存器号(int)。
map_of_reg_str:一个静态std::map，将寄存器数(int)映射到寄存器名(std::string)。
map_of_asm:一个静态std::map，将汇编指令(std::string)映射到操作码(uint32_t)。
map_of_opcode:一个静态std::map，将操作码(uint32_t)映射到汇编指令(如std::string)。
instruction_type:指示指令类型的enum类变量。
splt_instruction:一个std::vector，存储被分割成几部分的指令。
asm_size: int型，表示指令中部件的数量。
bi_instruction: uint32_t类型，存储指令的二进制表示。

成员函数

getUValueFromBits:一个从给定值的起始和结束索引的二进制代码中提取无符号整数值的函数。
getValueFromBits:一个从给定值的开始和结束索引的二进制代码中提取带符号整数值的函数。
InitMap:初始化静态映射map_of_reg_num、map_of_reg_str、map_of_asm和map_of_opcode的函数。
BitTypeConvert_xxx():一系列将比特形式的指令解读并相应设置map_of_reg_num、map_of_reg_str、map_of_asm、map_of_opcode等成员变量的函数。

AsmTypeConvert_xxx():一系列将汇编语言形式的指令解读并相应设置map_of_reg_num、map_of_reg_str、map_of_asm、map_of_opcode等成员变量的函数。
Exe_xxx():执行相应指令

总的来说，这个类提供了一种在c++中表示和操作MIPS汇编指令的方便方法。

3.CodeType类

CodeType类是一个虚拟类，该类包含了一些虚拟成员函数，包括add、sub、mul和div，以及一个toString函数。
这些函数都返回空指针或空字符串，并且可以在子类中被覆盖实现。这个类可以被用来作为其他特定类型的类的基类，这些类可以继承CodeType类并在其中实现自己的算法和操作。

4.IntType类

IntType类是 CodeType 的一个子类，用于表示Int类型的数值。

成员变量
int64_t value: 存储 IntType 类型的数值；
string res: 存储 IntType 类型的二进制表示（按照固定长度存储）；
static int length: 静态变量，表示二进制字符串的长度；
static int error_val: 静态变量，表示可能出现的错误值。

构造函数
IntType(int is_empty)：构造函数，如果传入的参数是非零值，则不做任何操作；
IntType(int64_t value, int length)：构造函数，根据传入的值和长度生成一个 IntType 实例；
IntType(string res, int length)：构造函数，根据传入的二进制字符串和长度生成一个 IntType 实例；**IntType()：**构造函数，根据用户输入的方式生成一个 IntType 实例。

成员函数
CodeType* add(CodeType* another)：将当前 IntType 实例和另一个 CodeType 类型的实例相加，返回相加的结果；
CodeType* sub(CodeType* another)：将当前 IntType 实例和另一个 CodeType 类型的实例相减，返回相减的结果；

CodeType* mul(CodeType* another)：将当前 IntType 实例和另一个 CodeType 类型的实例相乘，返回相乘的结果；
CodeType* div(CodeType* another)：将当前 IntType 实例和另一个 CodeType 类型的实例相除，返回相除的结果。
static int64_t BiStringToInt(string str, int length)：将二进制字符串转化为 IntType 类型的数值；
static string IntToBiString(int64_t value, int length)：将 IntType 类型的数值转化为二进制字符串。

该类的功能是实现 IntType 类型的数值和其二进制表示之间的转化，并支持 IntType 类型的加减乘除操作。成员函数和静态函数的作用在函数名和注释中均有解释。

5.FloatType类

FloatType是CodeType的子类。它的作用是实现浮点数类型，包括浮点数到二进制字符串的转换和四则运算的实现。

成员变量
float value：浮点数的值。
string res：浮点数的二进制字符串表示。

构造函数
FloatType(float value)：构造函数，用于根据给定的浮点数构造一个对象。在该函数中，将value设置为传入的value，并将res设置为对value进行FloatToBiString转换后的结果。
FloatType(string str)：构造函数，用于根据给定的字符串构造一个对象。在该函数中，将res设置为传入的str，并将value设置为对str进行BiStringToFloat转换后的结果。
FloatType()：构造函数，用于从控制台获取浮点数，并将其转换为对象。在该函数中，先在控制台打印一个提示，让用户选择用哪种方式（float或string）初始化对象，然后根据用户输入的选择从控制台获取数据，并调用适当的函数进行转换。

成员函数
static float BiStringToFloat(string str)：静态函数，用于将二进制字符串转换为浮点数。在该函数中，根据IEEE 754标准将二进制字符串转换为浮点数。
static string FloatToBiString(float value)：静态函数，用于将浮点数转换为二进制字符串。在该函数中，根据IEEE 754标准将浮点数转换为二进制字符串。
virtual CodeType* add(CodeType* another)：虚函数，用于实现加法运算。在该函数中，将当前对象的value与传入对象的value相加，然后创建一个新的FloatType对象，并将该对象的value设置为两数

之和，res设置为value的二进制字符串表示，最后返回该对象的指针。
virtual CodeType* sub(CodeType* another)：虚函数，用于实现减法运算。在该函数中，将当前对象的value与传入对象的value相减，然后创建一个新的FloatType对象，并将该对象的value设置为两数之差，res设置为value的二进制字符串表示，最后返回该对象的指针。
virtual CodeType* mul(CodeType* another)：虚函数，用于实现乘法运算。在该函数中，将当前对象的value与传入对象的value相乘，然后创建一个新的FloatType对象，并将该对象的value设置为两数之积，res设置为value的二进制字符串表示，最后返回该对象的指针。
virtual CodeType* div(CodeType* another)：虚函数，用于实现除法运算。在该函数中，将当前对象的value与传入对象的value相除，然后创建一个新的FloatType对象，并将该对象的value设置为两数之商，res设置为value的二进制字符串表示，最后返回该对象的指针。

6.ConvertInterface类

ConvertInterface类提供了一个接口函数run()，用于实现两种不同类型的数字转换和计算。

功能

- 当用户运行run()函数时，它会显示一个菜单，让用户选择要转换的数字类型（整型或浮点型）。
- 接下来，程序将要求用户输入两个数字，然后使用这些数字来创建两个CodeType对象。
- 一旦对象被创建，run()函数将调用它们的toString()函数，以显示这两个数字的值。
- 然后它将调用每个对象的add()、sub()、mul()和div()函数，来计算两个数字的和、差、积和商，并将结果显示出来。最后，程序等待用户按Enter键以继续。

成员变量

CodeType* code_1 和 CodeType* code_2 分别表示用户输入的两个数字所对应的CodeType对象，可以通过调用它们的成员函数来执行数字转换和计算。
int type：是用户选择的数字类型，1为整型，2为浮点型。

类内接口

void run() 是程序的主要接口函数，用于显示菜单、获取用户输入、执行数字转换和计算，以及等待用户输入Enter键以继续。

7.UserInterface类

UserInterface类是一个用户界面类，用于为一个虚拟机提供用户交互界面，该虚拟机可以运行汇编语言程序。

功能

- 打印用户界面，提供选项让用户选择要执行的操作。

- 将汇编代码转换为二进制代码并输出。
- 将二进制代码转换为汇编代码并输出。
- 加载二进制代码并运行。
- 将十进制数转换为二进制数。

成员变量

interface_state: 用于跟踪用户界面状态，确定执行哪个操作。

类内接口

printUI()：打印用户界面，获取用户输入以更改界面状态。

AsmConvert()：将输入的汇编代码转换为二进制代码并输出。

BiConvert()：将输入的二进制代码转换为汇编代码并输出。

GetCode()：获取用户输入的二进制代码并返回一个 uint32_t 类型的向量。

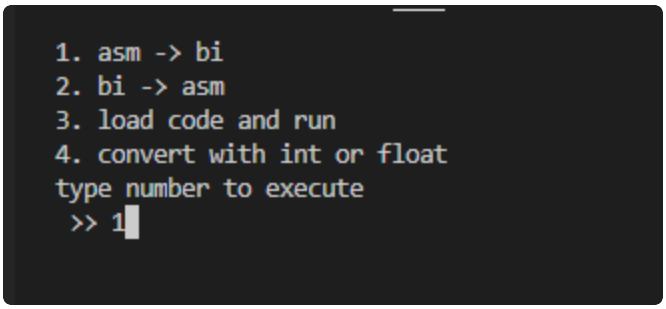
VmExecute()：加载用户输入的二进制代码并运行虚拟机。

run()：包含一个无限循环，根据用户界面的当前状态选择执行上述操作之一。如果界面状态不是上述状态之一，则无操作。

四、测试

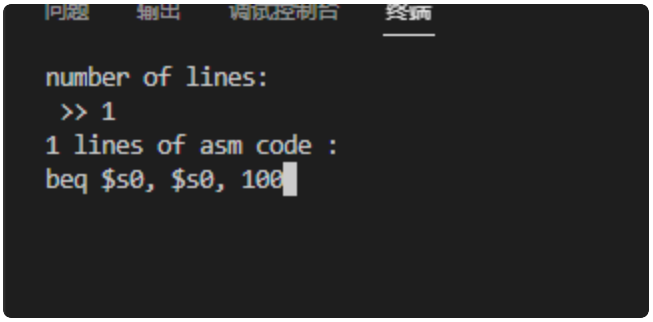
1.UI界面响应

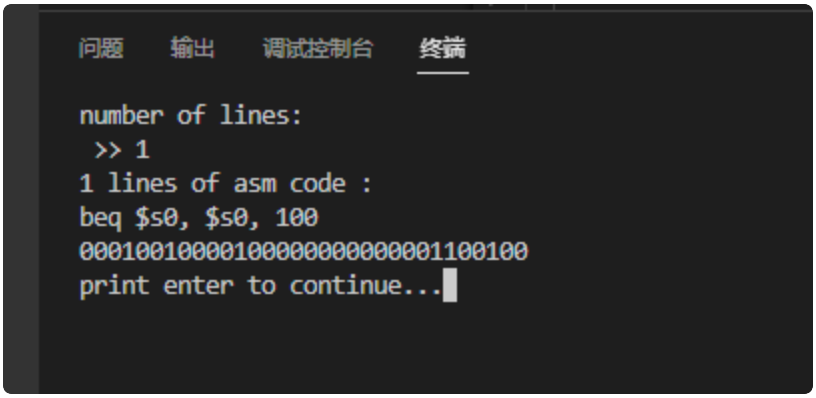
该测试用于测试UserInterface是否正常运行。编译运行，输入数字进入对应的部分。



2.汇编

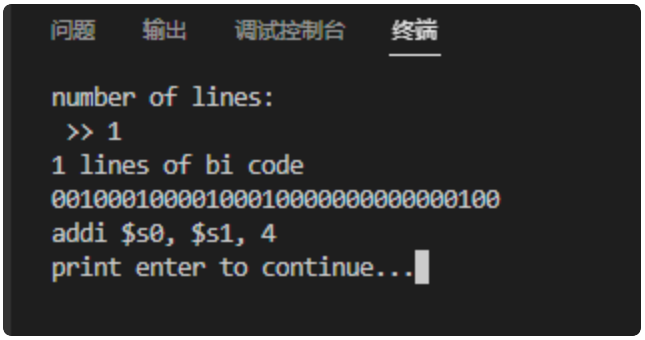
该测试测试汇编功能是否能正常运行。





3.反汇编

该测试测试反汇编功能是否能正常运行。



4.模拟运行

该测试测试输入一段指令，MIPS模拟器是否能够正常模拟运行该段代码。

```
lines of the code:
 >> 7
7 lines of bi code
00100010000100010000000000000100
00110010000100010000000000000000
00110110000100010000000000000000
00111010000100010000000000000001
00001100000000000000000000101000
00010010000100000000000001100100
00000010000000000000000000001000
```

```
        B D H
reg 0 : 0 0 0        reg 1 : 0 0 0        reg 2 : 0 0 0        reg 3 : 0 0 0
reg 4 : 0 0 0        reg 5 : 0 0 0        reg 6 : 0 0 0        reg 7 : 0 0 0
reg 8 : 0 0 0        reg 9 : 0 0 0        reg 10 : 0 0 0        reg 11 : 0 0 0
reg 12 : 0 0 0        reg 13 : 0 0 0        reg 14 : 0 0 0        reg 15 : 0 0 0
reg 16 : 0 0 0        reg 17 : 0 0 0        reg 18 : 0 0 0        reg 19 : 0 0 0
reg 20 : 0 0 0        reg 21 : 0 0 0        reg 22 : 0 0 0        reg 23 : 0 0 0
reg 24 : 0 0 0        reg 25 : 0 0 0        reg 26 : 0 0 0        reg 27 : 0 0 0
reg 28 : 0 0 0        reg 29 : 0 0 0        reg 30 : 0 0 0        reg 31 : 0 0 0
22   11   0   4
32   11   0   0    <<
36   11   0   0
3a   11   0   1
c   0   0   28
12   10   0   64
2   0   0   8
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0

press enter to run to next step
```

其中按回车可以单步运行，<<处即为当前指令的位置。

5.int和float的转换

Int类型转换

int支持变长，并且可以根据长度自动补全

```
length :
 >> 32
fisrt :
Start your int
1. start with an int
2. start with a string
 >> 1
value :
 >> 124
second :
Start your int
1. start with an int
2. start with a string
 >> 2
string :
 >> 10101001101010
```

```
first is : (123;00000000000000000000000001111011)
second is : (42149;00000000000000001010010010100101)
(123;00000000000000000000000001111011) + (42149;00000000000000001010010010100101) =
       (42272;00000000000000001010010100100000)
(123;00000000000000000000000001111011) - (42149;00000000000000001010010010100101) =
       (-42026;11111111111111111010110111010110)
(123;00000000000000000000000001111011) * (42149;00000000000000001010010010100101) =
       (5184327;00000000000000000000000000000000000000000001001111000110110100111)
(123;00000000000000000000000001111011) / (42149;00000000000000001010010010100101) =
       (0;00000000000000000000000000000000)
press enter to continue
```

Float类型转换

float必须输入32位并且无法自动补全

```
first :
Start your float
1. start with an float
2. start with a string
 >> 1
value :
 >> 3.14
second :
Start your float
1. start with an float
2. start with a string
 >> 2
string :
 >> 01000001000100100110101101111011
```

```
first is : (3.14;01000000010010001111010111000010)
second is : (9.15124;01000001000100100110101101111011)
(3.14;01000000010010001111010111000010) + (9.15124;01000001000100100110101101111011) =
       (12.2912;01000001010001001010100011101011)
(3.14;01000000010010001111010111000010) - (9.15124;01000001000100100110101101111011) =
       (-6.01124;11000000110000000101110000010011)
(3.14;01000000010010001111010111000010) * (9.15124;01000001000100100110101101111011) =
       (28.7349;01000001111001011110000100010000)
(3.14;01000000010010001111010111000010) / (9.15124;01000001000100100110101101111011) =
       (0.343123;00111110101011111010110111001101)
press enter to continue
```

6.单指令运行测试

该部分对10个实现的指令进行功能测试，以确保其能够正常运行

- add

add $s0, $s1, $s2，手动将reg2设置为5，reg3设置为3

```
00000010001100101000000000100000
reg1 : 16reg2 : 17reg3: 0
       B D H
reg 0 : 11 3 3        reg 1 : 0 0 0        reg 2 : 0 0 0        reg 3 : 0 0 0
reg 4 : 0 0 0        reg 5 : 0 0 0        reg 6 : 0 0 0        reg 7 : 0 0 0
reg 8 : 0 0 0        reg 9 : 0 0 0        reg 10 : 0 0 0        reg 11 : 0 0 0
reg 12 : 0 0 0        reg 13 : 0 0 0        reg 14 : 0 0 0        reg 15 : 0 0 0
reg 16 : 1000 8 8        reg 17 : 101 5 5        reg 18 : 0 0 0        reg 19 : 0 0 0
reg 20 : 0 0 0        reg 21 : 0 0 0        reg 22 : 0 0 0        reg 23 : 0 0 0
reg 24 : 0 0 0        reg 25 : 0 0 0        reg 26 : 0 0 0        reg 27 : 0 0 0
reg 28 : 0 0 0        reg 29 : 0 0 0        reg 30 : 0 0 0        reg 31 : 0 0 0
25 0   0   0   0
```

- addi

addi $s0, $s1, 4，手动将reg2设置为12

```
00100010000100010000000000000100
reg1 : 0reg2 : 16val: 4
        B D H
reg 0 : 10000 16 10          reg 1 : 0 0 0          reg 2 : 0 0 0          reg 3 : 0 0 0
reg 4 : 0 0 0          reg 5 : 0 0 0          reg 6 : 0 0 0          reg 7 : 0 0 0
reg 8 : 0 0 0          reg 9 : 0 0 0          reg 10 : 0 0 0          reg 11 : 0 0 0
reg 12 : 0 0 0          reg 13 : 0 0 0          reg 14 : 0 0 0          reg 15 : 0 0 0
reg 16 : 1100 12 c          reg 17 : 0 0 0          reg 18 : 0 0 0          reg 19 : 0 0 0
reg 20 : 0 0 0          reg 21 : 0 0 0          reg 22 : 0 0 0          reg 23 : 0 0 0
reg 24 : 0 0 0          reg 25 : 0 0 0          reg 26 : 0 0 0          reg 27 : 0 0 0
reg 28 : 0 0 0          reg 29 : 0 0 0          reg 30 : 0 0 0          reg 31 : 0 0 0
25 0   0   0   0
```

- and

and $s0, $s1, $s2，reg2=12, reg3=10，结果在reg1

```
00000010001100101000000000100100
reg1 : 16reg2 : 17reg3: 0
        B D H
reg 0 : 1010 10 a          reg 1 : 0 0 0          reg 2 : 0 0 0          reg 3 : 0 0 0
reg 4 : 0 0 0          reg 5 : 0 0 0          reg 6 : 0 0 0          reg 7 : 0 0 0
reg 8 : 0 0 0          reg 9 : 0 0 0          reg 10 : 0 0 0          reg 11 : 0 0 0
reg 12 : 0 0 0          reg 13 : 0 0 0          reg 14 : 0 0 0          reg 15 : 0 0 0
reg 16 : 1000 8 8          reg 17 : 1100 12 c          reg 18 : 0 0 0          reg 19 : 0 0 0
reg 20 : 0 0 0          reg 21 : 0 0 0          reg 22 : 0 0 0          reg 23 : 0 0 0
reg 24 : 0 0 0          reg 25 : 0 0 0          reg 26 : 0 0 0          reg 27 : 0 0 0
reg 28 : 0 0 0          reg 29 : 0 0 0          reg 30 : 0 0 0          reg 31 : 0 0 0
25 0   0   0   0
```

- andi

andi $s0, $s1, 0，将reg2值手动设置为16，和0按位与，输出到reg1

```
andi
00110010000100010000000000000000
reg1: 16 reg2: 17 imm: 0
        B D H
reg 0 : 0 0 0         reg 1 : 0 0 0         reg 2 : 0 0 0         reg 3 : 0 0 0
reg 4 : 0 0 0         reg 5 : 0 0 0         reg 6 : 0 0 0         reg 7 : 0 0 0
reg 8 : 0 0 0         reg 9 : 0 0 0         reg 10 : 0 0 0         reg 11 : 0 0 0
reg 12 : 0 0 0         reg 13 : 0 0 0         reg 14 : 0 0 0         reg 15 : 0 0 0
reg 16 : 0 0 0         reg 17 : 10000 16 10         reg 18 : 0 0 0         reg 19 : 0 0 0
reg 20 : 0 0 0         reg 21 : 0 0 0         reg 22 : 0 0 0         reg 23 : 0 0 0
reg 24 : 0 0 0         reg 25 : 0 0 0         reg 26 : 0 0 0         reg 27 : 0 0 0
reg 28 : 0 0 0         reg 29 : 0 0 0         reg 30 : 0 0 0         reg 31 : 0 0 0
25 0   0   0   0
```

- **beq**

beq $s0, $s0, 100，由于$s0 与 $s0 一定相等，因此在PC+4基础上偏移100，由于PC在测试时初始值为0，因此PC应设为104

```
00010010000100000000000001100100
reg1: 16 reg2: 16 offset: 100
        B D H
reg 0 : 0 0 0         reg 1 : 0 0 0         reg 2 : 0 0 0         reg 3 : 0 0 0
reg 4 : 0 0 0         reg 5 : 0 0 0         reg 6 : 0 0 0         reg 7 : 0 0 0
reg 8 : 0 0 0         reg 9 : 0 0 0         reg 10 : 0 0 0         reg 11 : 0 0 0
reg 12 : 0 0 0         reg 13 : 0 0 0         reg 14 : 0 0 0         reg 15 : 0 0 0
reg 16 : 0 0 0         reg 17 : 0 0 0         reg 18 : 0 0 0         reg 19 : 0 0 0
reg 20 : 0 0 0         reg 21 : 0 0 0         reg 22 : 0 0 0         reg 23 : 0 0 0
reg 24 : 0 0 0         reg 25 : 0 0 0         reg 26 : 0 0 0         reg 27 : 0 0 0
reg 28 : 0 0 0         reg 29 : 0 0 0         reg 30 : 0 0 0         reg 31 : 0 0 0
25 0   0   0   0
pc: 104
```

- **jal**

jal 40， PC在跳转至40，同时PC+4存入$ra中。因此PC=40，$ra = 4

```
00001100000000000000000000101000
 target: 40
        B D H
reg 0 : 0 0 0          reg 1 : 0 0 0          reg 2 : 0 0 0          reg 3 : 0 0 0
reg 4 : 0 0 0          reg 5 : 0 0 0          reg 6 : 0 0 0          reg 7 : 0 0 0
reg 8 : 0 0 0          reg 9 : 0 0 0          reg 10 : 0 0 0          reg 11 : 0 0 0
reg 12 : 0 0 0          reg 13 : 0 0 0          reg 14 : 0 0 0          reg 15 : 0 0 0
reg 16 : 0 0 0          reg 17 : 0 0 0          reg 18 : 0 0 0          reg 19 : 0 0 0
reg 20 : 0 0 0          reg 21 : 0 0 0          reg 22 : 0 0 0          reg 23 : 0 0 0
reg 24 : 0 0 0          reg 25 : 0 0 0          reg 26 : 0 0 0          reg 27 : 0 0 0
reg 28 : 0 0 0          reg 29 : 0 0 0          reg 30 : 0 0 0          reg 31 : 100 4 4
9 0   0   0   0
0   0   0   0    <<
15 0   0   0   0
pc: 40
```

- jr

jr $s0, 将PC的值替换为$s0中的值，由于$s0中的值为2，因此PC变为2

```
00000010000000000000000000001000
 reg1: 16
        B D H
reg 0 : 0 0 0          reg 1 : 0 0 0          reg 2 : 0 0 0          reg 3 : 0 0 0
reg 4 : 0 0 0          reg 5 : 0 0 0          reg 6 : 0 0 0          reg 7 : 0 0 0
reg 8 : 0 0 0          reg 9 : 0 0 0          reg 10 : 0 0 0          reg 11 : 0 0 0
reg 12 : 0 0 0          reg 13 : 0 0 0          reg 14 : 0 0 0          reg 15 : 0 0 0
reg 16 : 10 2 2          reg 17 : 0 0 0          reg 18 : 0 0 0          reg 19 : 0 0 0
reg 20 : 0 0 0          reg 21 : 0 0 0          reg 22 : 0 0 0          reg 23 : 0 0 0
reg 24 : 0 0 0          reg 25 : 0 0 0          reg 26 : 0 0 0          reg 27 : 0 0 0
reg 28 : 0 0 0          reg 29 : 0 0 0          reg 30 : 0 0 0          reg 31 : 0 0 0
0   0    << 0   0
24 0   0   0   0
pc: 2
```

- ori

ori $s0, $s1, 0，将reg2值手动设置为16，和0按位或，输出到reg1中

```
00110110000100010000000000000000
reg1: 16 reg2: 17 imm: 0
        B D H
reg 0 : 0 0 0         reg 1 : 0 0 0         reg 2 : 0 0 0         reg 3 : 0 0 0
reg 4 : 0 0 0         reg 5 : 0 0 0         reg 6 : 0 0 0         reg 7 : 0 0 0
reg 8 : 0 0 0         reg 9 : 0 0 0         reg 10 : 0 0 0         reg 11 : 0 0 0
reg 12 : 0 0 0         reg 13 : 0 0 0         reg 14 : 0 0 0         reg 15 : 0 0 0
reg 16 : 10000 16 10         reg 17 : 10000 16 10         reg 18 : 0 0 0         reg 19 : 0 0 0
reg 20 : 0 0 0         reg 21 : 0 0 0         reg 22 : 0 0 0         reg 23 : 0 0 0
reg 24 : 0 0 0         reg 25 : 0 0 0         reg 26 : 0 0 0         reg 27 : 0 0 0
reg 28 : 0 0 0         reg 29 : 0 0 0         reg 30 : 0 0 0         reg 31 : 0 0 0
25 0   0   0   0
```

- **slt**

slt $s0, $s1, $s2，手动将reg2设为1，reg3设为12

```
reg1 : 16reg2 : 17reg3: 0
        B D H
reg 0 : 1100 12 c         reg 1 : 0 0 0         reg 2 : 0 0 0         reg 3 : 0 0 0
reg 4 : 0 0 0         reg 5 : 0 0 0         reg 6 : 0 0 0         reg 7 : 0 0 0
reg 8 : 0 0 0         reg 9 : 0 0 0         reg 10 : 0 0 0         reg 11 : 0 0 0
reg 12 : 0 0 0         reg 13 : 0 0 0         reg 14 : 0 0 0         reg 15 : 0 0 0
reg 16 : 1 1 1         reg 17 : 1010 10 a         reg 18 : 0 0 0         reg 19 : 0 0 0
reg 20 : 0 0 0         reg 21 : 0 0 0         reg 22 : 0 0 0         reg 23 : 0 0 0
reg 24 : 0 0 0         reg 25 : 0 0 0         reg 26 : 0 0 0         reg 27 : 0 0 0
reg 28 : 0 0 0         reg 29 : 0 0 0         reg 30 : 0 0 0         reg 31 : 0 0 0
25 0   0   0   0
```

slt $s0, $s1, $s2，手动将reg2设置为12，reg3设置为10

```
00000010001100101000000000101010
reg1 : 16reg2 : 17reg3: 0
        B D H
reg 0 : 1010 10 a         reg 1 : 0 0 0         reg 2 : 0 0 0         reg 3 : 0 0 0
reg 4 : 0 0 0         reg 5 : 0 0 0         reg 6 : 0 0 0         reg 7 : 0 0 0
reg 8 : 0 0 0         reg 9 : 0 0 0         reg 10 : 0 0 0         reg 11 : 0 0 0
reg 12 : 0 0 0         reg 13 : 0 0 0         reg 14 : 0 0 0         reg 15 : 0 0 0
reg 16 : 0 0 0         reg 17 : 1100 12 c         reg 18 : 0 0 0         reg 19 : 0 0 0
reg 20 : 0 0 0         reg 21 : 0 0 0         reg 22 : 0 0 0         reg 23 : 0 0 0
reg 24 : 0 0 0         reg 25 : 0 0 0         reg 26 : 0 0 0         reg 27 : 0 0 0
reg 28 : 0 0 0         reg 29 : 0 0 0         reg 30 : 0 0 0         reg 31 : 0 0 0
25 0   0   0   0
```

- **xori**

xori $s0, $s1, 1，将reg2手动设置为16，与1按位异或后存储到reg1

```
00111010000100010000000000000001
reg1: 16 reg2: 17 imm: 1
        B D H
reg 0 : 0 0 0         reg 1 : 0 0 0         reg 2 : 0 0 0         reg 3 : 0 0 0
reg 4 : 0 0 0         reg 5 : 0 0 0         reg 6 : 0 0 0         reg 7 : 0 0 0
reg 8 : 0 0 0         reg 9 : 0 0 0         reg 10 : 0 0 0         reg 11 : 0 0 0
reg 12 : 0 0 0         reg 13 : 0 0 0         reg 14 : 0 0 0         reg 15 : 0 0 0
reg 16 : 10001 17 11         reg 17 : 10000 16 10         reg 18 : 0 0 0         reg 19 : 0 0 0
reg 20 : 0 0 0         reg 21 : 0 0 0         reg 22 : 0 0 0         reg 23 : 0 0 0
reg 24 : 0 0 0         reg 25 : 0 0 0         reg 26 : 0 0 0         reg 27 : 0 0 0
reg 28 : 0 0 0         reg 29 : 0 0 0         reg 30 : 0 0 0         reg 31 : 0 0 0
25 0   0   0   0
```