**MIPS模拟器技术报告报告**

1. **功能介绍**

该项目实现了四大功能：

1.汇编器：将汇编指令程序转换成机器码。

2.反汇编：将32位机器码转换成汇编指令

3.MIPS模拟器：能模拟执行MIPS机器码，并提供debug功能（显示执行后的寄存器的值和内存空间的值）。执行可以分为单步执行和一键执行，同时还具有设定指定地址的内存空间的值的功能。

4.整数（补码）、浮点数的表示、转换和运算，具体包括：有符号整数的原码、反码、补码表示、32位二进制补码转有符号整数、单精度浮点数的IEEE754表示、二进制编码转单精度浮点数（float）、双精度浮点数的IEEE754表示、二进制编码转单精度浮点数（double）、进制转换器，且所有功能均包含对错误输入的检测，即能够给出异常反馈。

对以上四大功能的整合及用户界面的制作用到了Qt Creator，所有的前端界面都是基于Qt Creator实现的。

下面将根据四大功能依次介绍其主要模块功能和实现原理。

1. **MIPS模拟器**

***模块功能介绍***

**1. 模拟执行&显示寄存器功能**

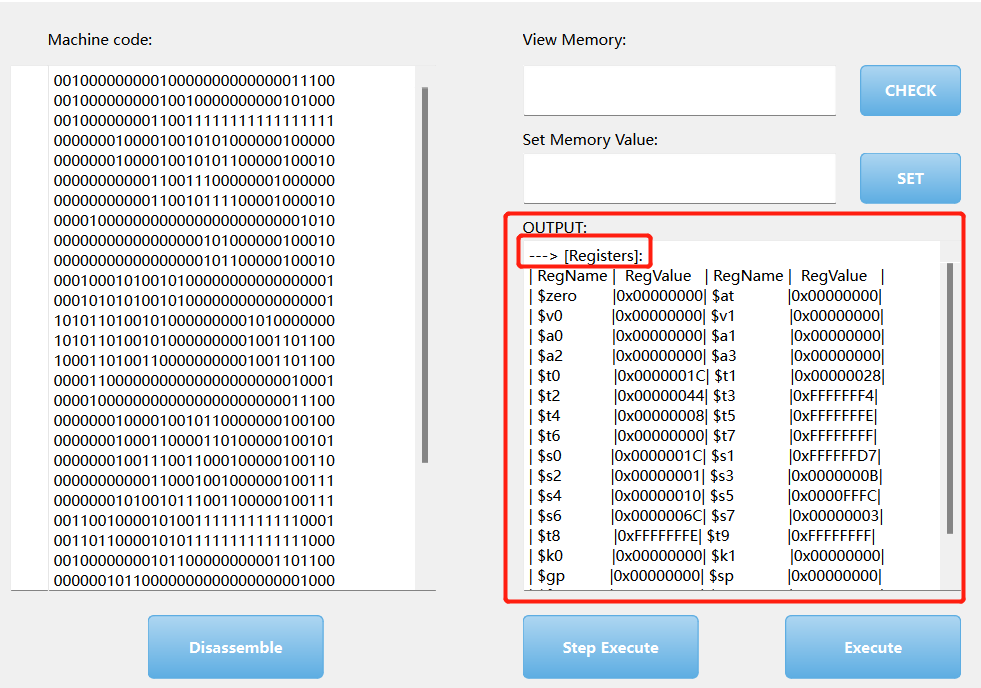
MIPS模拟器Simulator模块主要的功能是对输入的32位机器码进行翻译、执行，并将执行结果以用户可见的方式呈现出来。该Simulator模块已经实现并支持的汇编指令如下：

***R-type：ADD SUB AND OR XOR NOR JR SLT SLL SRL***

***I-type：BEQ BNE ADDI ANDI ORI LW SW***

***J-type：J JAL***

子功能1执行效果示例：



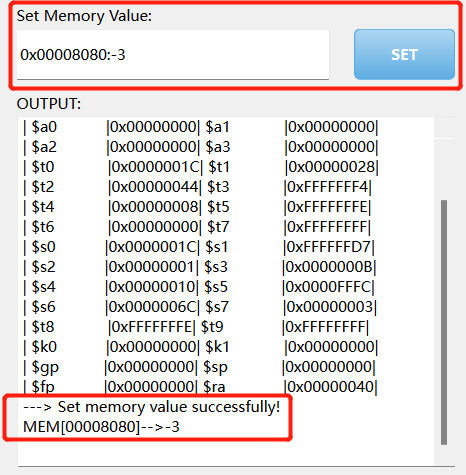
其中，当用户按下Execute（一键执行）或Step Execute（单步执行）按钮后，所有32位寄存器的值都将显示在红色的框中，方便用户检查和调试。

1. **给指定内存空间设值**

考虑到在编写汇编程序的过程中，可能会遇到希望将某些特定的常数（如数组或全局变量等）存储在一块或多块内存地址以便赋值给变量的情况，该模块特地增加了给指定内存空间设值的功能。用户只需要按照指定的格式就可以将某个特定的整数赋值给一块内存空间（一块内存空间指连续的4个byte空间，即连续的32位数据存储空间）。

**输入格式示例：0x00000004:31 ——给地址为0x00000004的内存空间设值为31 （前面10位以0x为前缀的8位16进制数表示指定的内存空间地址（要确保是4的倍数），冒号后面跟的是signed int类型的整数，支持负数形式的输入）**

子功能2执行效果示例：



当用户按上文指定的输入格式输入后，点击SET按钮，就会在下面的文本框中跳出已经成功为内存空间设定值的交互消息提示。

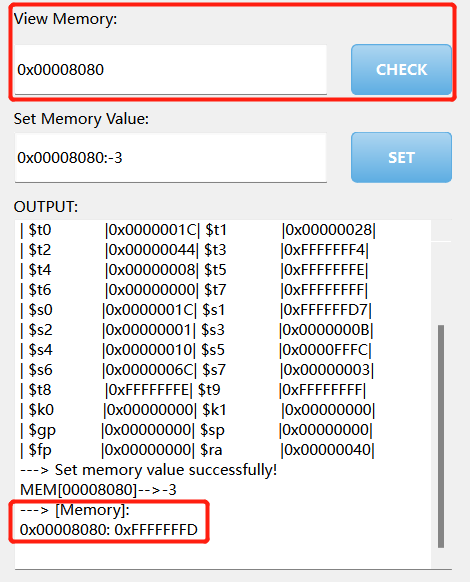
1. **查看内存**

除了支持寄存器的查看外，该模块还支持对内存的查看。

**输入格式示例：0x00008080 ——查询地址为0x00008080的内存空间的值**

**（输入须严格按照以0x为前缀8位16进制数进行，否则会出错）**

子功能3执行效果示例：

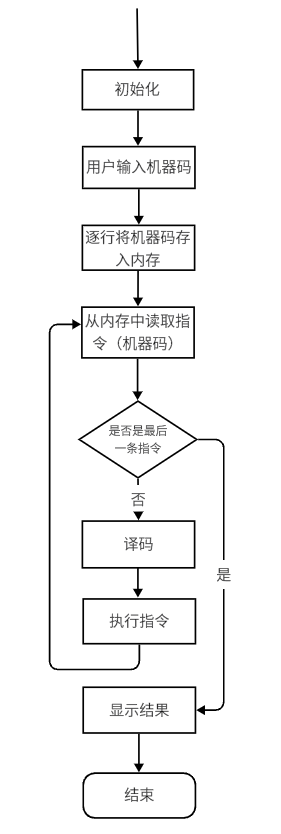


当用户按照上文格式输入需要查看的内存地址后，点击CHECK按钮，在下面的文本框中就会给出对应内存空间的值以及相应的交互信息。（如图所示，0x00008080地址的内存空间已经成功被改成了-3，输出形式是8位16进制补码）

***核心算法实现***

1. **实现原理及流程图**

**该模拟器模块实现的基本思路是：先将用户输入的32位二进制机器码全部读取，并依次存储到一块连续的内存空间当中（默认从内存地址0开始存储，4bytes一条机器码），然后从存储第一条机器码的内存空间开始（默认从内存地址0开始）依次读取机器码进入译码环节，然后针对不同类型的指令执行不同的操作，直到读取完最后一条指令（机器码）。**



1. **全局变量**

在实现模拟机（Simulator）的过程中，PC、内存、寄存器是非常重要的三个数据单 元，考虑到所有指令都会涉及到这三者，所以在程序中，这三者都被设置成全局变量的 形式，下面是有关这三者的定义和初始化函数。

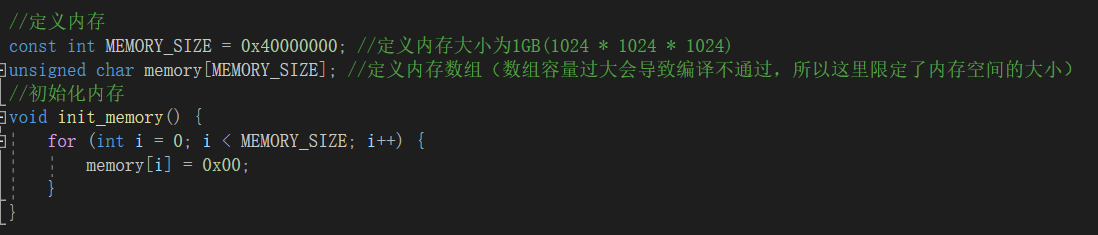


Figure 内存定义和初始化

**需要注意的是内存数组是unsigned char类型，也就是最多存储8位二进制，所以一条指令（32位二进制机器码）占4个连续的内存空间。**

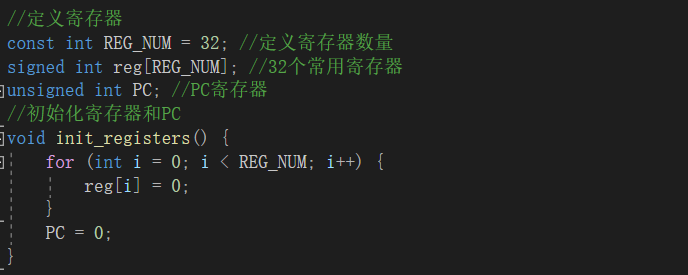


Figure 寄存器、PC定义和初始化

**其中PC存储的是下一条要读取的指令所对应的内存地址，因为一条指令由32位二进制机器码组成，而内存数组的大小为8位（unsigned char类型），因此PC必须是4的倍数，否则会出错！**

1. **重要函数**

**·3.1内存读/取函数**

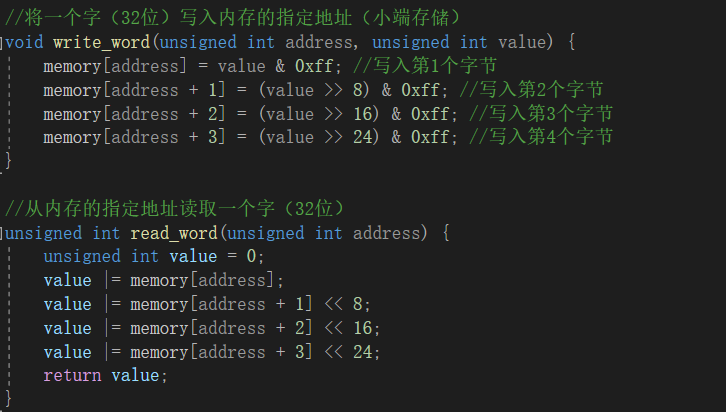


Figure 内存读/取函数

需要注意的是，该程序内存的存储模式是小端存储。内存的存储形式是unsigned int类型，即将32位2进制转换成unsigned int进行存储，这样存储的好处是可以通过移位运算和位逻辑运算很快地对读取的指令进行译码。

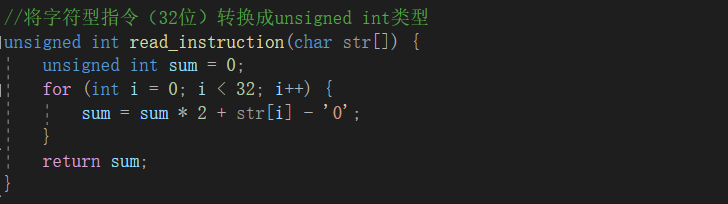


Figure 指令转换函数

在读取机器码并存到内存环节，由于用户输入的机器码是用string类型接收的，所以需要用到string转换为unsigned int 类型的函数，该函数也间接地属于内存读/取函数范畴。

**·3.2指令函数**

**·3.2.1 R-type类型指令**

已经实现的R-type类型指令有：ADD SUB AND OR XOR NOR JR SLT SLL SRL

**SLL & SRL:**

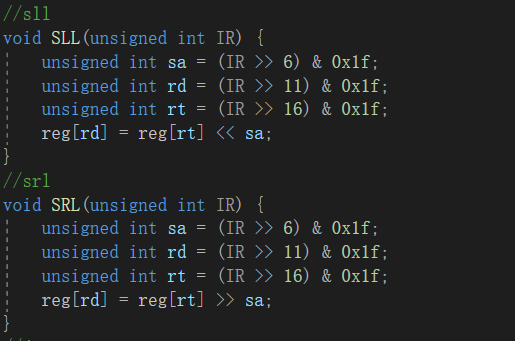


Figure SLL和SRL函数

实现思路：按照指令格式用移位运算和位逻辑运算读取寄存器编号和移位位数，然后对寄存器执行相应的移位。

**JR:**

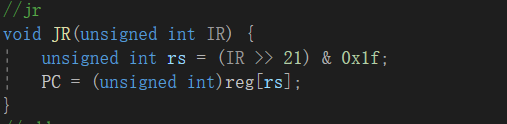


Figure JR函数

实现思路：按照指令格式用移位运算和位逻辑运算读取寄存器编号，然后更新PC的值为相应寄存器编号的值。

**ADD & SUB & AND & OR & XOR & NOR:**

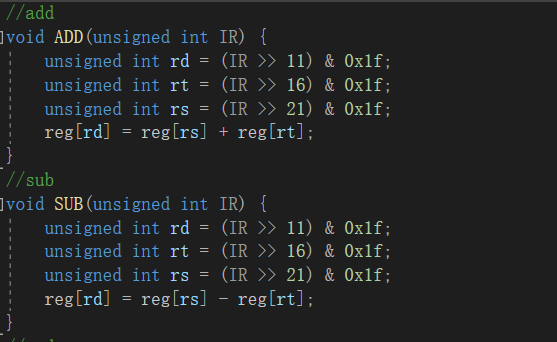


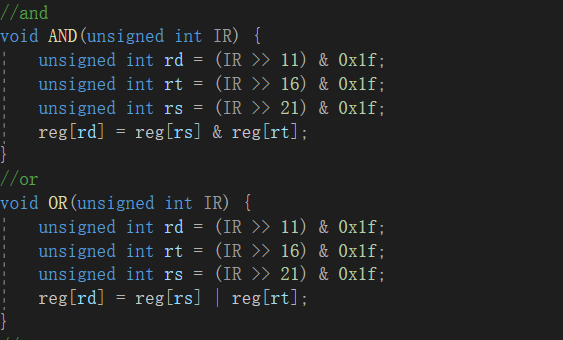
Figure ADD和SUB函数

Figure AND和OR函数

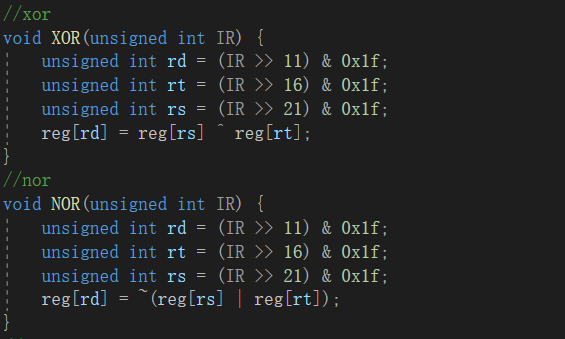


Figure XOR和NOR函数

这几个函数属于R型算术运算和逻辑运算，其实现机理都差不多，都是按照指令格式用移位运算和位逻辑运算读取寄存器编号，然后对相应的寄存器执行算术运算或逻辑运算。

**SLT:**

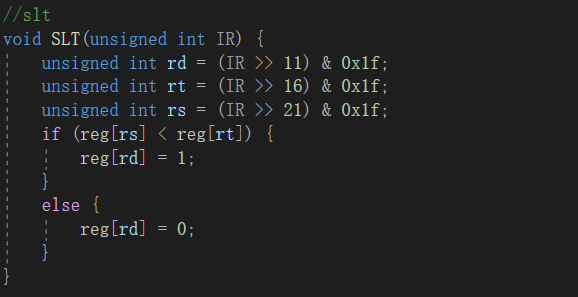


Figure SLT函数

实现思路：同前面的指令一样，先按照指令格式用移位运算和位逻辑运算读取寄存器编号，然后比较读取出来的rs寄存器和rt寄存器的大小，如果rs寄存器小于rt寄存器，那么就将rd寄存器的值设为1，反之则设为0。

**R\_decode（R型指令编码函数）:**

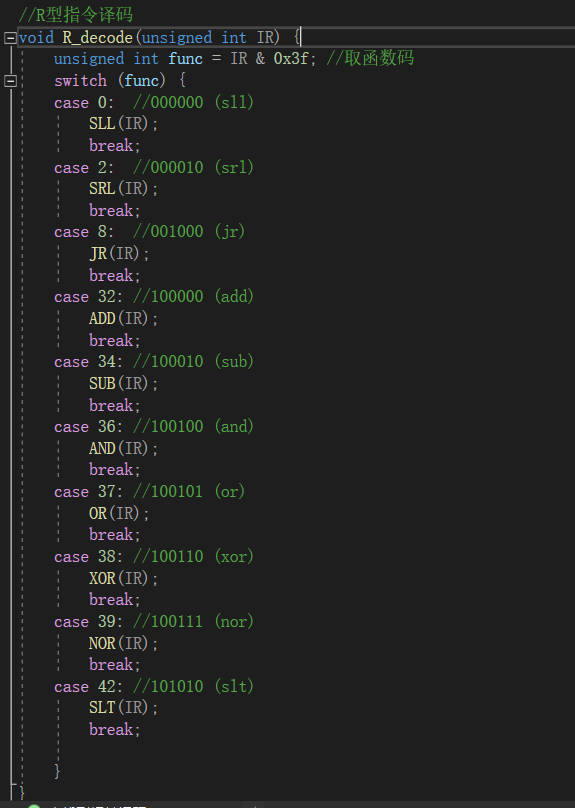


Figure R\_decode函数

R型指令译码函数的实现原理：由于R-type指令的opcode都是000000，所以只需对其最后6位func（函数码）进行判断即可。这里先将指令与0x3f（111111）按位与取出func（函数码），然后再用switch语句进行判断即可。

**·3.2.2 I-type类型指令**

已经实现的I-type类型指令有：BEQ BNE ADDI ANDI ORI LW SW

**BEQ & BNE:**

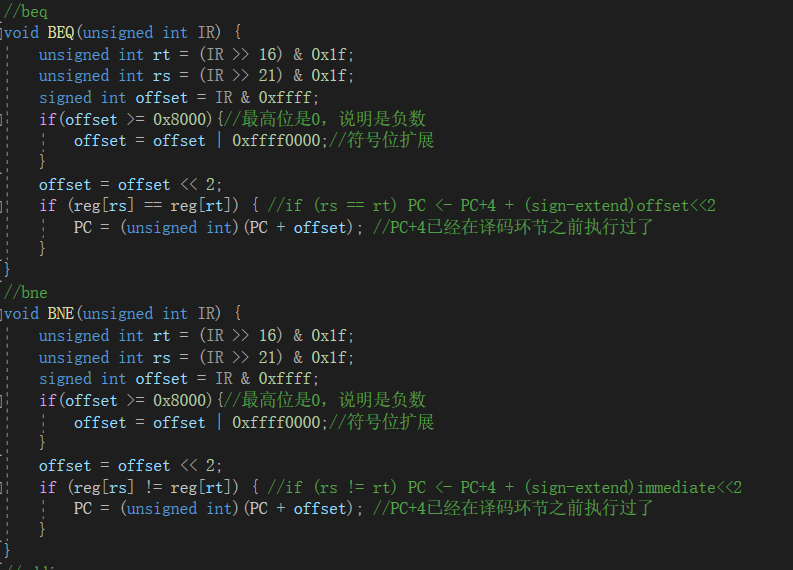


Figure BEQ和BNE函数

实现思路：先按照指令格式用移位运算和位逻辑运算读取寄存器编号和偏移量offset（**偏移量是补码类型**）。然后先判断offset最高位是不是1，若是1，则偏移量为负数，需要进行符号位拓展（将高位全部置为１）；若不为１，则偏移量是正数，不需任何操作。对于**BEQ**函数，若rs寄存器的值等于rt寄存器的值，则将现在的PC加上偏移量offset，而**BNE**函数的实现思路则相反。

**需要注意的是:由于取指令环节PC已经+4.所以在指令执行阶段，现在的PC指的是该指令对于的PC再加上4（PC+4）。**

**ADDI & ANDI & ORI :**

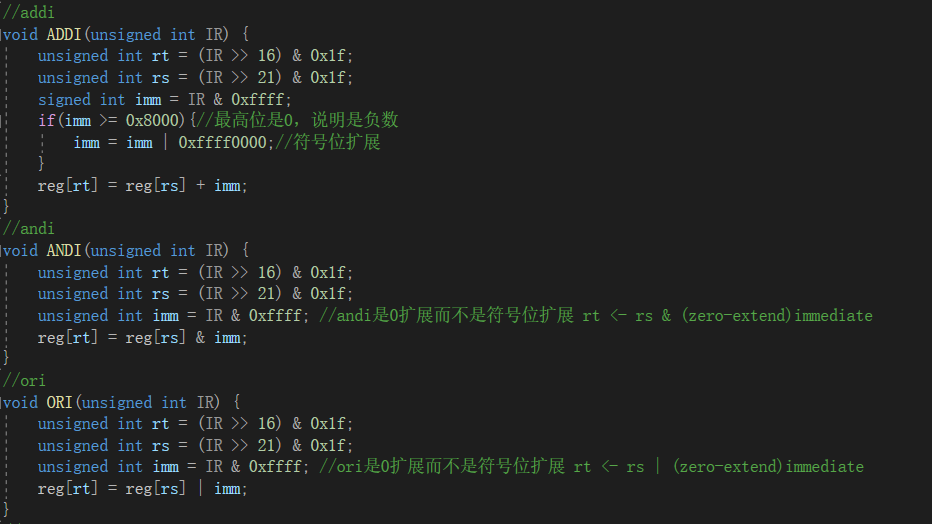


Figure ADDI、ANDI和ORI函数

这三个函数的实现思路同ADD AND OR十分相似，唯一的不同点是，某一个寄存器的值换成了立即数（imm）。同前面的指令函数一样，先按照指令格式用移位运算和位逻辑运算读取寄存器编号和偏移量offset（**ADDI的偏移量是补码类型，ANDI和ORI的偏移量不是，所以ADDI的立即数要进行正负数判断和最高位扩展，ANDI和ORI则不用**），然后将进行rs寄存器和立即数imm的算术运算或者逻辑运算，并将结果存储到rd寄存器。

**LW & SW:**

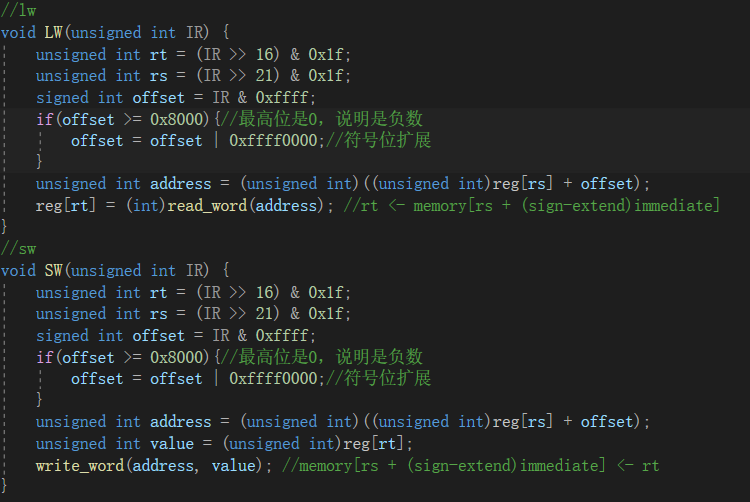


Figure LW和SW函数

LW和SW的实现思路：先按照指令格式用移位运算和位逻辑运算读取寄存器编号和偏移量offset（**偏移量是补码类型**），然后对偏移量进行最高位判断和符号位扩展。LW指令函数将rs寄存器的值加上offset之后的结果作为内存地址，读取该内存地址空间的值并赋值给rt寄存器；SW指令函数也将rs寄存器的值加上offset之后的结果作为内存地址，然后将rd寄存器的值赋值给该内存地址对应的内存空间。

**·3.2.3 J-type类型指令**

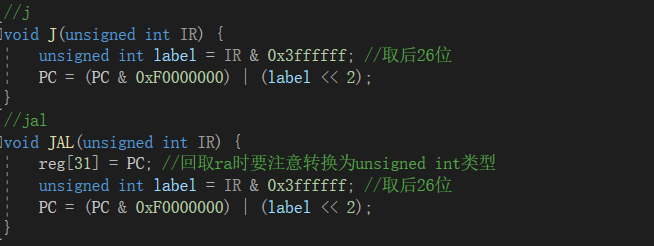


Figure J和JAL函数

J和JAL指令的实现思路：先按照指令格式用移位运算和位逻辑运算读取label（要跳转地址的后28位取前26位）值。因为考虑到PC一定是4的倍数，所以任何跳转地址的后2位必定是00，所以只需用26位即可表示28位（往26位后面添加00即可）。读取完label后，将现在PC的前4位保留，后28位置为label左移2位后的结果，这样得到的PC即为要跳转到的指令的内存地址。JAL和J相比，除了跳转之外，还要先把原来PC的值赋值给ra寄存器（以便返回）.

需要注意的是，同之前所讲的一样，在指令执行阶段的PC都是该指令对于的PC值加4（PC+4）。

Tips：考虑到I-type和J-type都可以用opcode区分，所以不用而外设置I型指令和J型指令的译码函数。

**·3.2.4 总的译码函数**

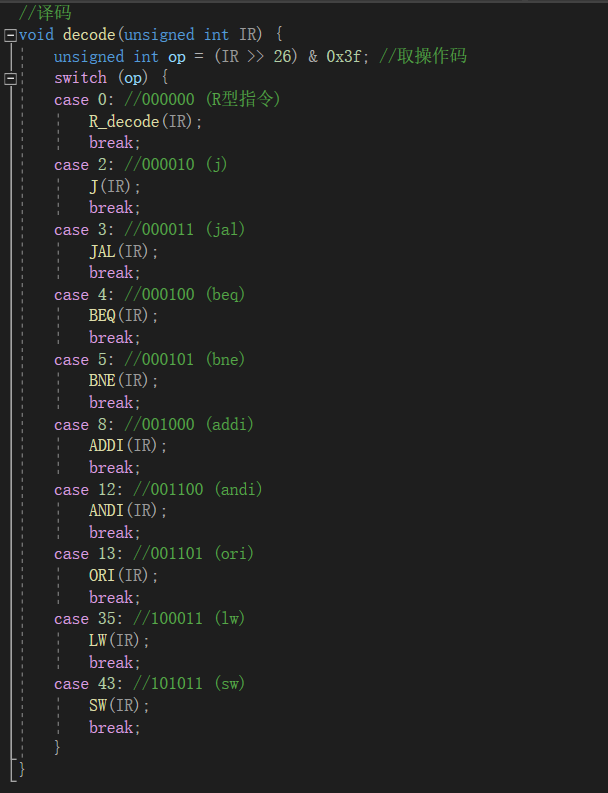


Figure 总的译码函数（decode）

decode函数的实现原理：先读取指令的前6位（opcode值），然后对opcode进行判断，执行相应的指令函数（**若是000000开头的R-type类型，则需要再进一步译码才能进入指令执行环节**）。

***GUI功能实现（基于Qt Creator）***

1. **Execute（一键执行）**

主要函数：void MainWindow::on\_Execute\_clicked()

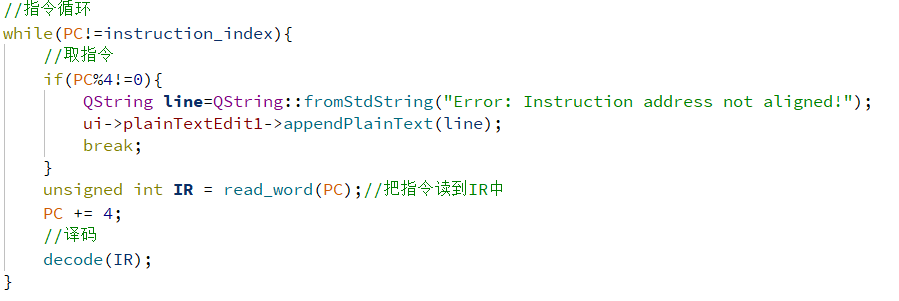
该函数实现过程：

①初始化内存、寄存器和PC

②将文本框内的机器码读取并存储到内存中（主要用到read\_instruction（）函数和write\_word（）函数，前面已介绍）。



③进行指令循环，依次读取指令并执行（主要用到read\_instruction（）函数和decode函数，前面已介绍）。



④OUTPUT（输出32位寄存器的值，主要用到RegOutput函数（），其实现原理较简易，这里不再展开描述）



1. **Step Execute（单步执行）**

主要函数：void MainWindow::on\_Step\_Execute\_clicked()

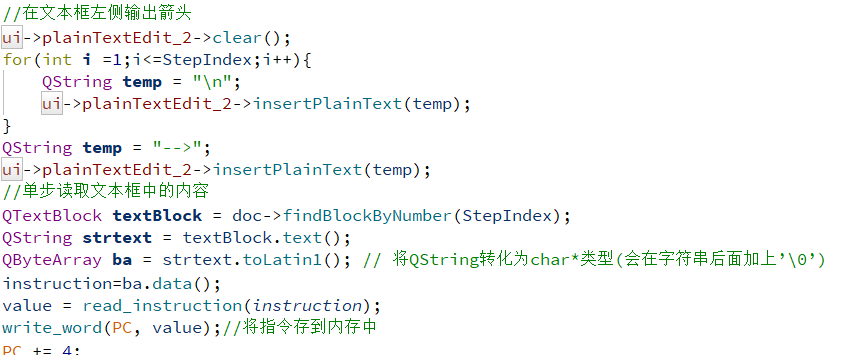
该函数实现过程：

①判断是否第一次执行

如果是则初始化内存，并读取文本框中的内容，否则清空输出框的内容，重置PC、StepIndex和flag（StepIndex：全局变量，用于判断当前读入的是文本框第几行，flag：用于判断是否是第一次执行）



②若不是第一次执行程序，则在文本框左侧画出指示性箭头（指示当前读取的行），并将读取到的32位机器码转化成unsigned int类型并存储到内存中



③执行指令并输出

和一键执行不同的是，单步执行只需要读一条指令，存一条指令，执行一条指令，而不需要先把指令存到内存，再从内存开始读取指令。

Tips：之所以要read\_word(PC-4)，是因为在取指令阶段PC已经加过4了，所以当前的指令存储地址是PC-4，所以要从内存地址为PC-4的地方读取指令。

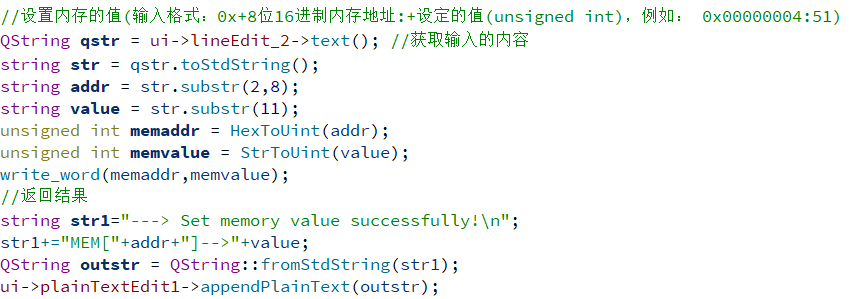


1. **SET MEMORY（设置内存空间的值）**

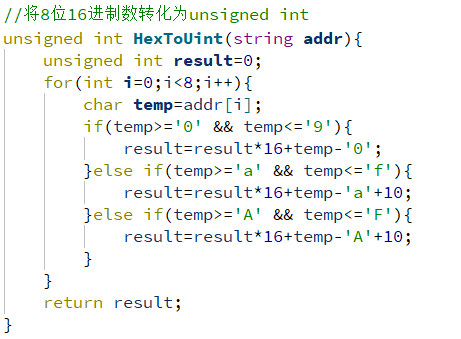
主要函数：void MainWindow::on\_CHECK\_clicked()

函数实现过程：

①读取用户输入的内存地址和要设置的值，调用写内存函数（write\_word（））给内存地址设值



其中用到了16进制形式的字符串转unsigned int函数HexToUint（）和string转unsigned int函数StrToUint（），它们的实现过程代码如下：





1. **CHECK（查看内存的值）**

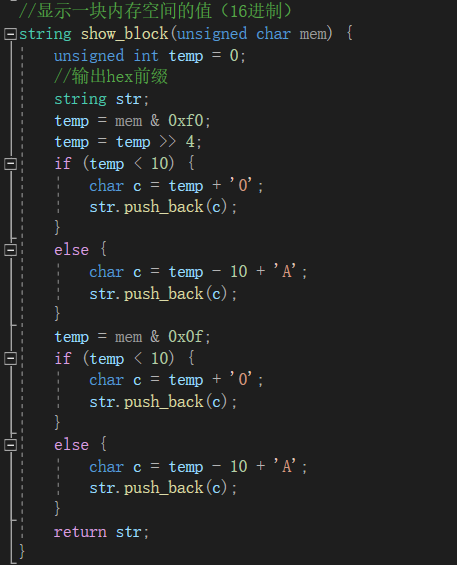
函数实现过程：

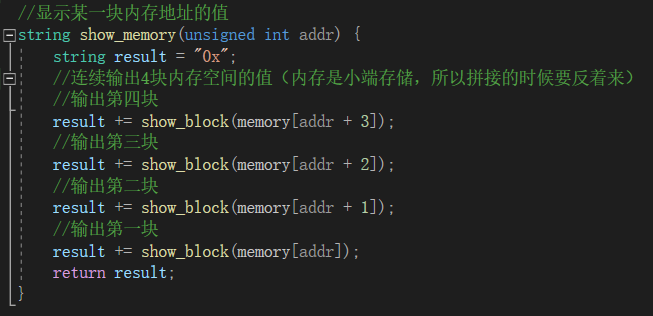
①获取用户输入的内存地址，并调用显示内存函数（show\_memory（））

其中内存显示函数show\_memory（）由子函数show\_block（）构成。

它们的实现代码和注释在下面的图中：







1. **汇编器**

***模块功能介绍***

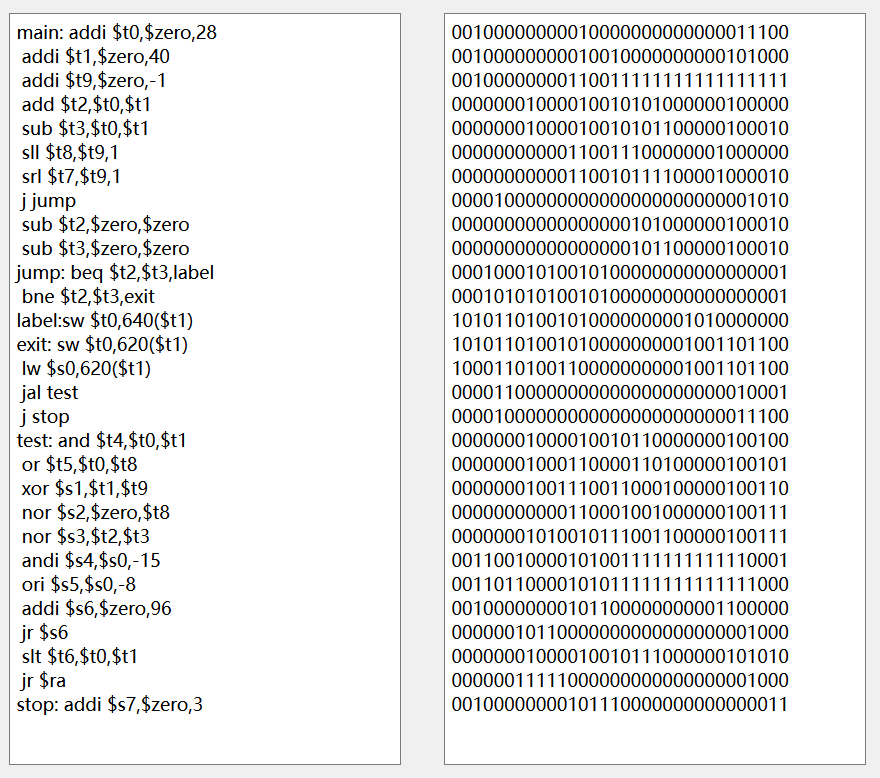
**1.编译功能**

MIPS编译器模块的目的是模拟MIPS编译器，将部分MIPS指令转换为二进制机器码并输出。在本模拟器中，实现了以下指令：

**ADD, ADDI, SUB, SUBI, AND, ANDI, OR, ORI, NOR, BEQ, BNE, J,JAL, JR, SLT, LW, SW, SRL,SLL,XOR**

同时也有所有寄存器的实现

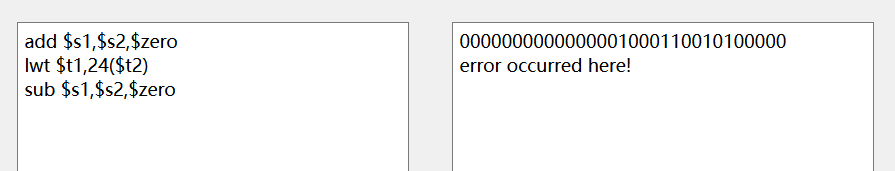
编译示例



**2.错误检测功能**

同时本模拟器还可以对指令的部分错误进行检测，能够检测的错误包括：

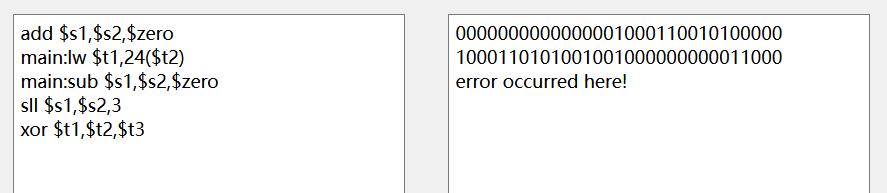
①该行指令不存在或不支持



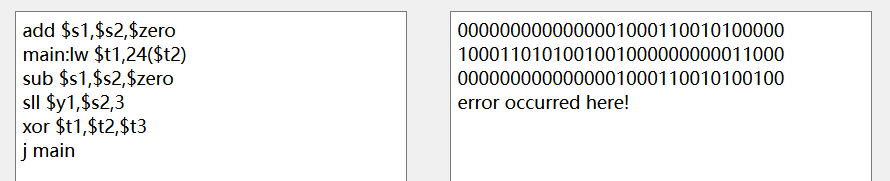
②指令格式不合法



③跳转标识重复



④寄存器错误

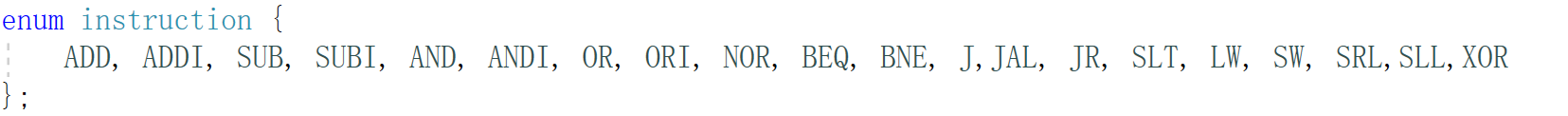


注：本编译器是按行编译的，当遭遇错误时，会在该MIPS指令的对应行中提示错误信息

***模块算法实现以及数据结构***

**1.数据结构设计**

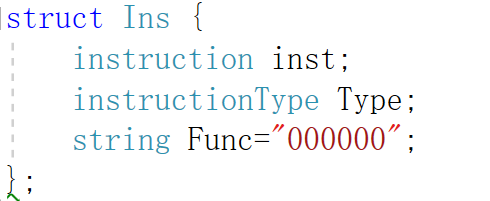
MIPS、指令中有两个非常重要而且会多次重复出现的组成元素：指令和寄存器。每个寄存器都有唯一的序列号，每个指令都有固定的格式和功能码。故可以在编译之前，生成指令和寄存器的相关数据存储在内存中，以便编译时调用。具体实现如下：



该枚举类型包含了所有支持的MIPS指令，用于对指令进行方便的唯一标识

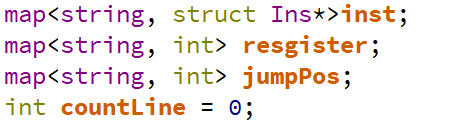


该枚举类型包含了三种指令类型



Ins结构包含了一个指令的所有信息，包括指令的唯一标识、指令类型以及功能码。

(此处功能码初始值设为“000000”因为大部分指令的功能码都为“000000”)



这里用map来根据指令字符串索引到相关指令的数据



这里用map来根据寄存器字符串索引到相关寄存器的序列号



这里用map来根据跳转标识符的字符串索引到标识符所在位置。

1. **重要函数以及算法设计**

**void MainWindow::on\_pushButton\_clicked()**

该函数用于响应图形界面上的编译按钮，是开始编译的入口。

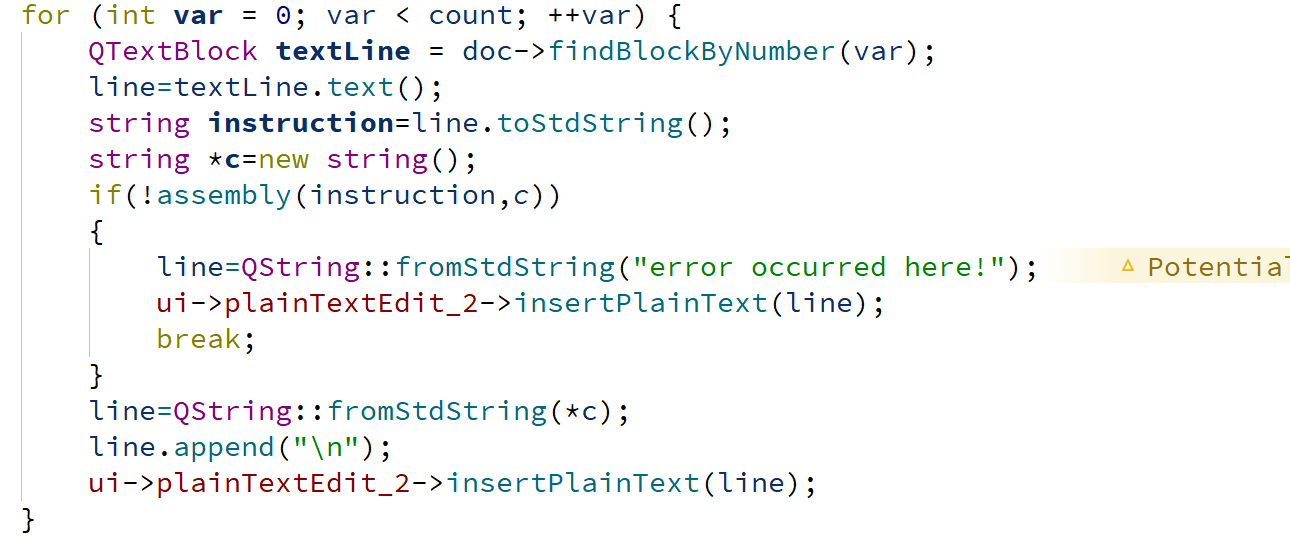
该函数的实现分为两个步骤：

①清空二进制机器码文本框中的内容

②逐一遍历每一行，找到并记录所有跳转标识符(jumpPos)的地址



③重新遍历每一行MIPS指令，生成相关的机器码，并将机器码逐行输出在机器码文本框中。



**int assembly(string L,string\*c)**

函数功能：是根据所给的指令字符串L生成相应的机器码，若生成成功，则将机器码存储到指针\*c中，并返回整数1；若生成失败，则返回0.

函数实现分为以下三个步骤：

①去除注释字段

判断改行是否存在注释字段‘#’，若有则去除注释字段

②处理跳转标记字段

判断是否有跳转标记字段，若有则消除该字段，没有则跳过



③判断指令类型调用不同的相关函数

这部分根据不同的指令生成不同的指令数据和寄存器数据，并传入相应的下一级处理函数（注：assemble传入下级函数的寄存器字符串中不包含指令字段，便于下一级进行处理）



若无异常则返回 1；反之则返回0

**int Rins(string line, struct Ins\* ins,string \*c)**

**（R型指令处理函数）**

函数功能：该函数根据上层函数传下来的寄存器字符串line和指令数据\*Ins生成相应的机器码存储到\*c中。

函数实现分为两个步骤：

①根据所给的寄存器字符串生成三个不同的寄存器存储在res1，res2，res3中：

（SLL和SRL指令没有res3，JR没有res2和res3，进行特殊处理）

比如说所给寄存器字符串为$s1,$t1,%zero，那么res1=”s1”,res2=”t1”,res3=”zero”

代码如下：



②根据所寄存器和相关指令构建机器码，这里由于SLL，SRL指令、JR指令与其他指令处理方式不同，共有两个分支。



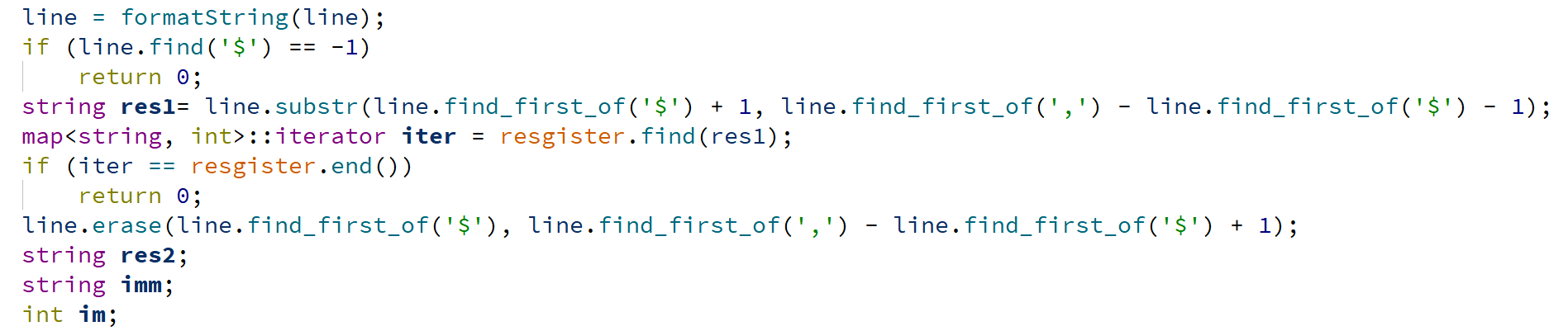
与这个函数类似的类型处理函数还有：

**int Iins(string line, struct Ins\*,string \*c);**

函数功能：该函数根据上层函数传下来的寄存器字符串line和指令数据\*Ins生成相应的机器码存储到\*c中。

函数实现分为两个步骤：

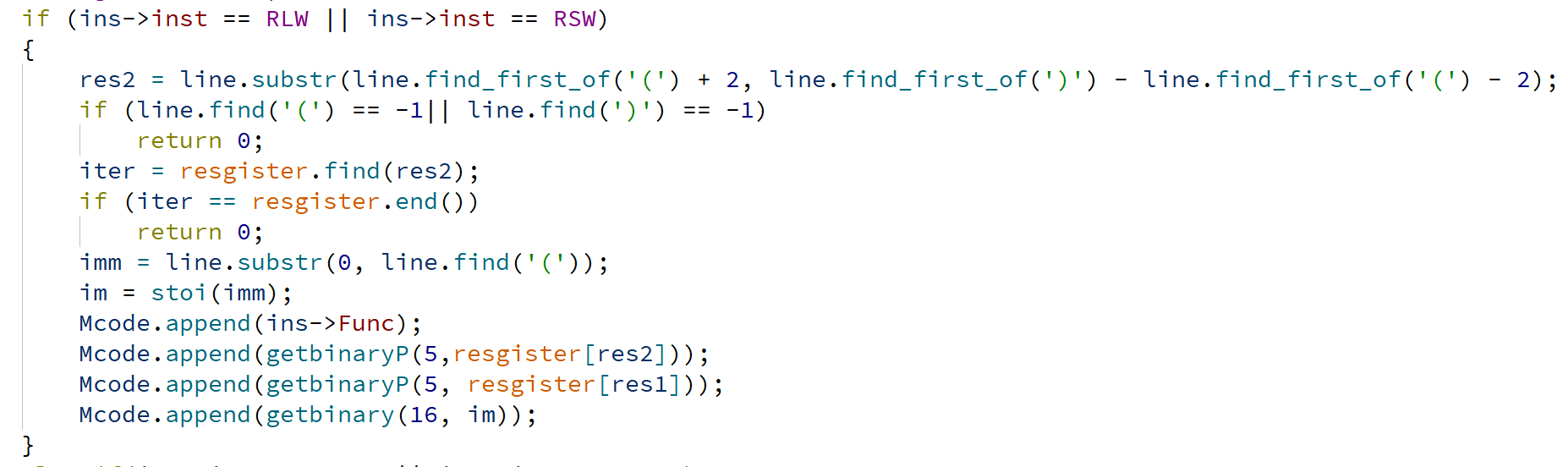
①根据所给的寄存器字符串生成寄存器存储在res1中：



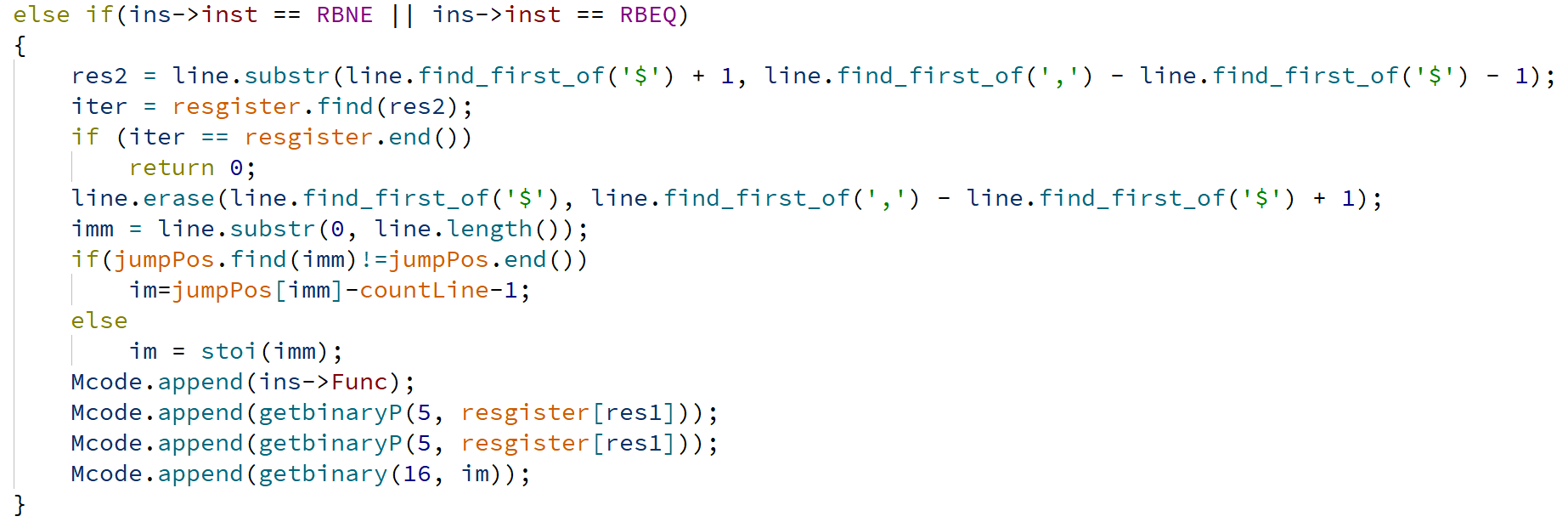
②由于不同I型指令的格式有较大差别，故在处理第二个寄存器res2时，则需要根据指令不同分别处理。处理完res2后，再获取立即数的值。

不同的I型指令大致可以分为3类：

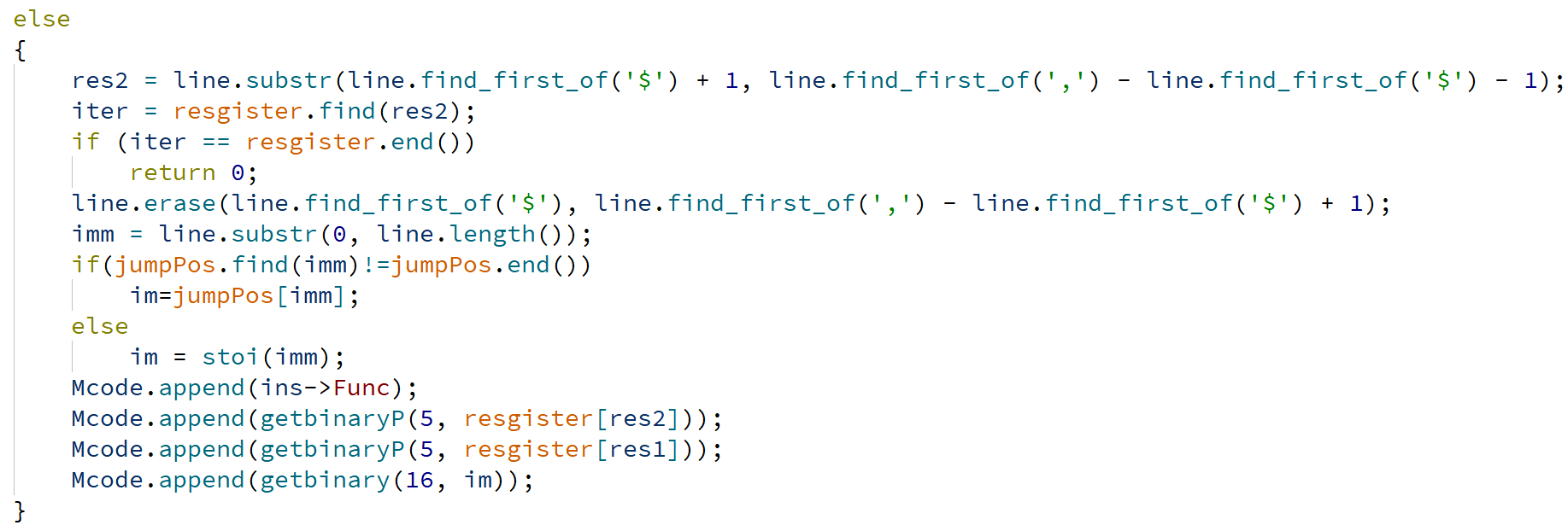
SW&LW



BEQ&BNE



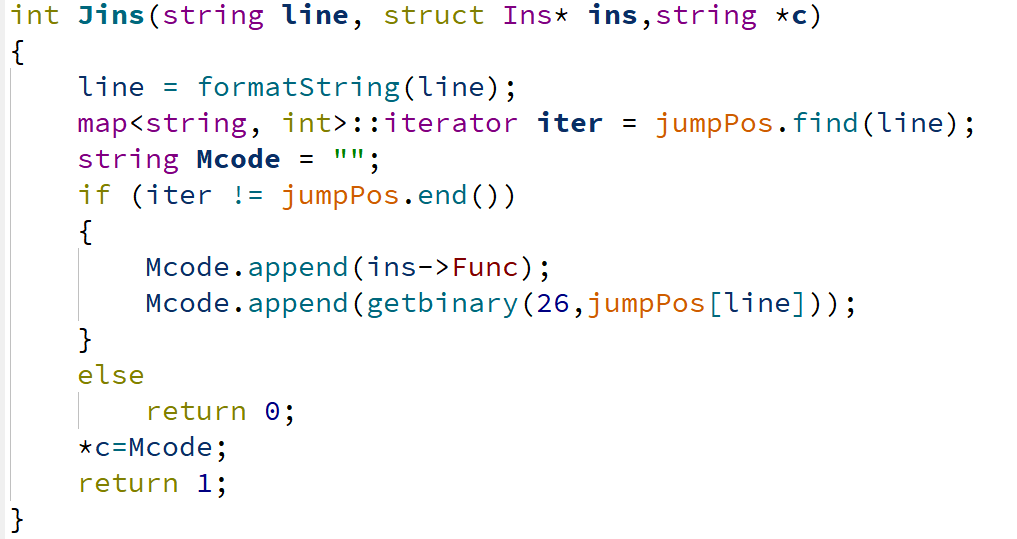
Other



**int Jins(string line, struct Ins\*,string \*c);**

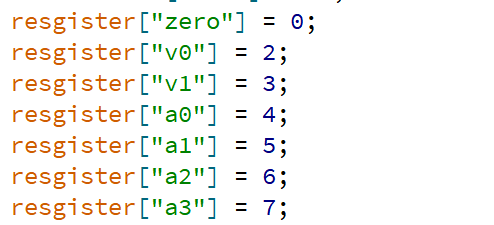
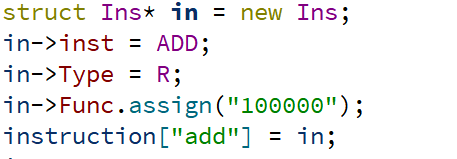
函数功能：该函数根据上层函数传下来的寄存器字符串line和指令数据\*Ins生成相应的机器码存储到\*c中。

该函数较为简单，只需要根据立即数和函数功能码直接生成机器码即可



**map<string, struct Ins\*> initial();**

该函数是一个初始化函数，对所有指令数据和寄存器数据进行初始化，并返回一个指令数据的map，部分代码示例如下：

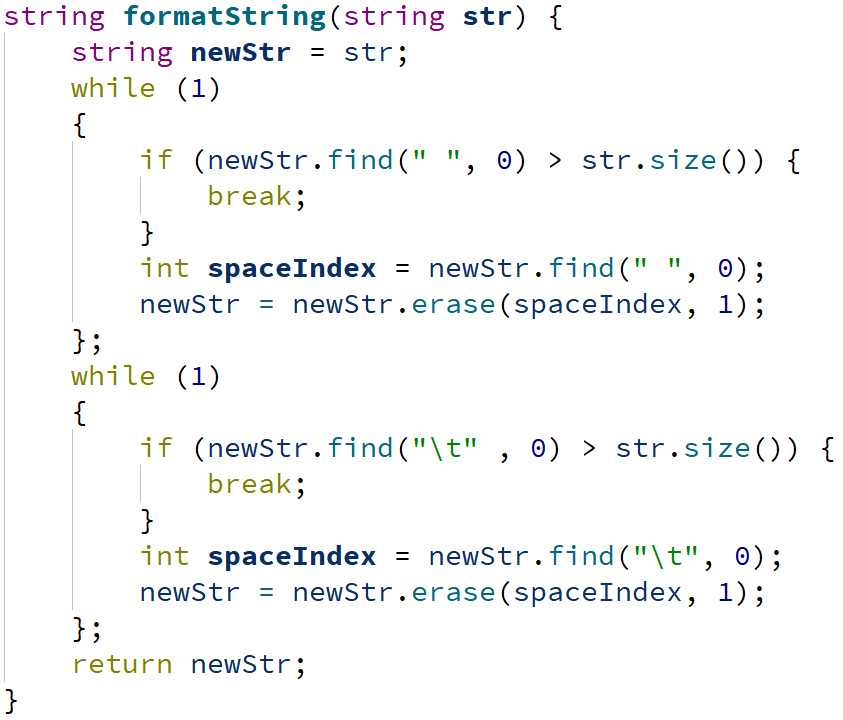


该函数能够在编译前做好初始化，方便编译操作

除此之外为了协助字符串操作，我还自己定义了一些字符串相关函数：

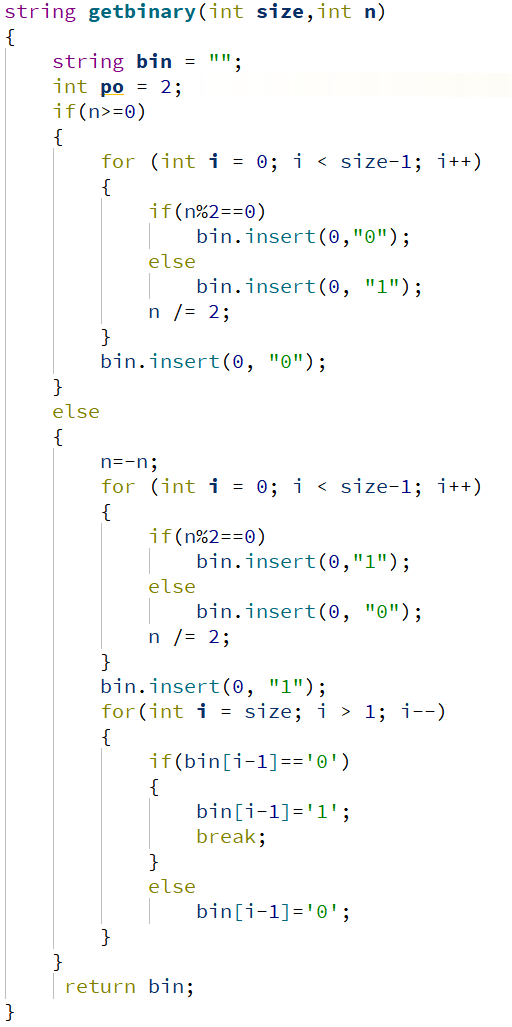
**string formatString(string str);**

该函数的功能是将字符串str中的所有空格和tab都去除



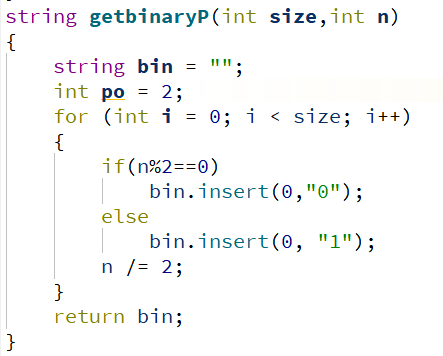
**string getbinary(int size, int n);**

该函数的功能是根据所给的参数，将n转换成一个长度为size的补码形式二进制数的字符串



**string getbinaryP(int size, int n);**

该函数的功能是根据所给的参数，将n转换成一个长度为size的原码形式二进制数的字符串



1. **整数（补码）、浮点数的表示、转换和运算。**

***模块功能介绍***

1.【有符号整数的原码、反码、补码表示】输入一个int范围内的整数，点击开始 转换，会生成相应的原码、反码、补码。

2.【32位二进制补码转有符号整数】输入一个32位二进制补码，点击开始转换， 会生成该补码对应的带符号整数。

3.【单精度浮点数的IEEE754表示】输入一个float数（若非float数会自动近似到最 邻近的float数），点击开始转换，会生成IEEE754标准下的二进制和十六进制表示。

4.【二进制编码转单精度浮点数（float）】输入一个32位IEEE754标准的二进制编 码，点击开始转换，会生成该编码表示的float数。</string>

5.【双精度浮点数的IEEE754表示】输入一个double数（若非double数会自动近似 到最邻近的double数），点击开始转换，会生成IEEE754标准下的二进制和十六进 制表示。

6.【二进制编码转单精度浮点数（double）】输入一个64位IEEE754标准的二进制 编码，点击开始转换，会生成该编码表示的double数。

7.【进制转换器】选择输入数的进制（二/十/十六），输入待转换的数，点击开始转 换，会生成相应的二、十、十六进制表示。若答案异常，可能是十进制数超过了long long int的范围。

* Tips：所有功能均包含对错误输入的检测，即能够给出异常反馈。

***模块算法实现以及数据结构***

**1.【有符号整数的原码、反码、补码表示】**

**void IntRepresentation(QString s,QString \*OC\_,QString \*RC\_,QString \*CC\_)**

输入：字符串s（表示整数）

输出：三种二进制形式的结果（OC、RC、CC）

1.声明变量i、x、temp、OC[33]、RC[33]、CC[33]（分别表示循环计数器、整数、中间变量、原码、反码、补码）；

2.将字符串s转化为int类型的x；

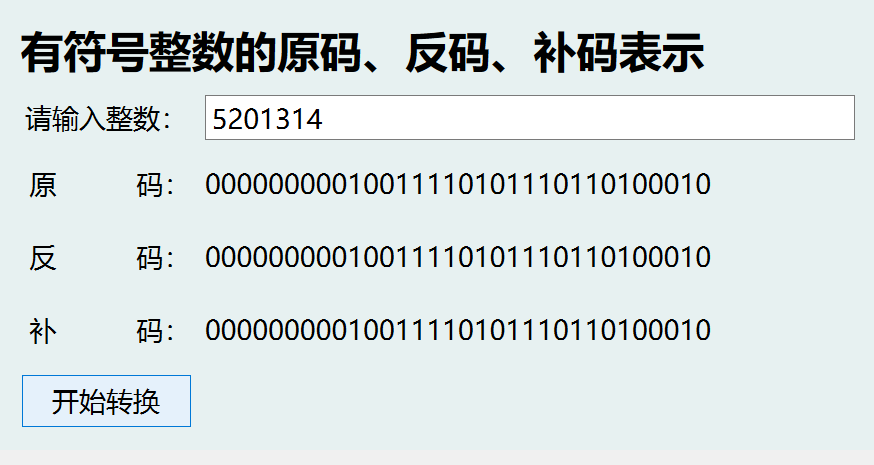
3.如果x等于0，则从0遍历到31，将OC、RC、CC的每一位都赋值为0；

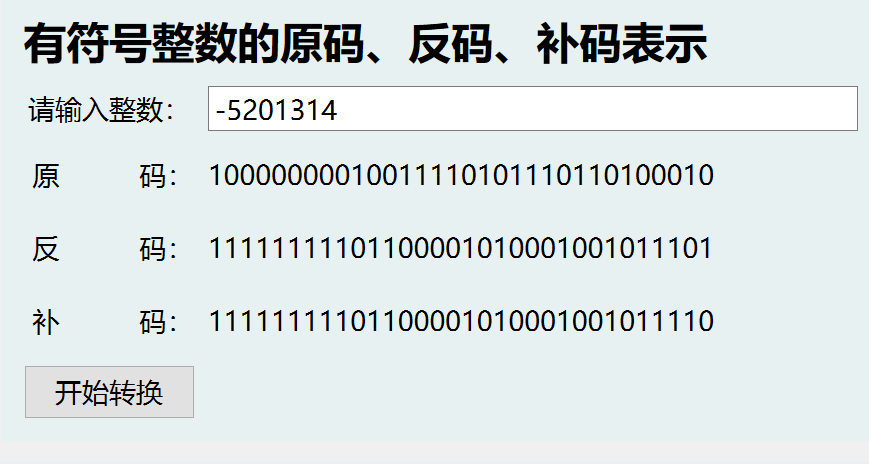
4.如果x大于0，则将x转化为二进制形式，并将结果赋值给OC、RC、CC；

5.如果x小于0，则将-x转化为二进制形式，并将结果赋值给OC、RC，将OC的符号位赋值为1，从0遍历到31，将RC中每一位都与0异或，得到RC的反码；最后将RC的结果加上1，得到CC的补码。

6.从0遍历到15，将OC、RC、CC中每一位对应地颠倒。

7.将OC、RC、CC分别以QString类型返回。





2. **【32位二进制补码转有符号整数】**

void ComplementToInt(QString s,int \*res\_)

输入：字符串s

输出：整数res

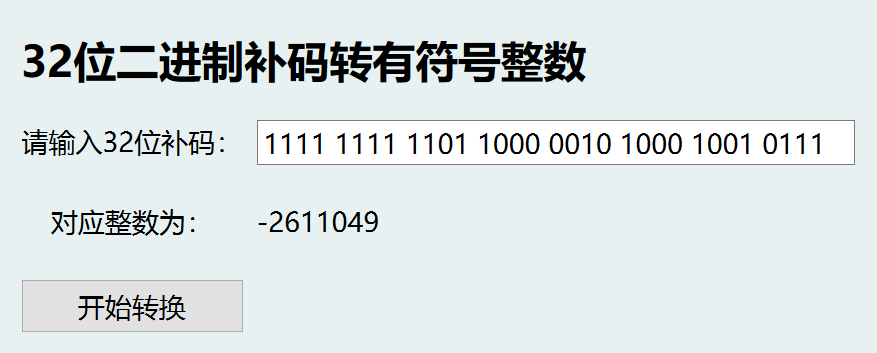
1.声明变量i、res、flag、c、code和code\_cp（分别表示循环计数器、结果、符号位、字符、指向字符串s的指针和指向code的指针）；

2.将字符串s转换为字符数组code，并将其地址赋值给code\_cp；

3.如果code的第一位为0，则flag等于1，否则flag等于-1；

4.如果flag等于1，则从31循环到0，将code中的第i位乘以2的31-i次幂并加到res上；

5.否则，将code\_cp减1得到它的反码，从31循环到0，将code\_cp的第i位取反并乘以2的31-i次幂并加到res上，最后将res的符号取反并赋值给res\_。



**3.【单精度浮点数的IEEE754表示】**

void floatToBinaryAndHex(float num, char\* binary, char\* hex)

输入：浮点数num

输出：二进制形式的字符串binary，十六进制形式的字符串hex

1.声明变量i、j、hex\_cp（分别表示循环计数器、循环计数器、十六进制的副本）；

2.声明整型指针pInt，并将其指向num的地址；

3.将pInt指向的内容转化为十六进制格式，并按照格式"%08X"存储到hex中，再将hex的最后一个字符设置为'\0'，将hex赋值给hex\_cp；

4.从0循环到7，表示8个16进制数字，对于每个16进制数字，将其转化为二进制，存储在binary中；

5.将binary的最后一个字符设为'\0'，函数结束。

* **是否为float数的判断**

输入：双精度浮点数x

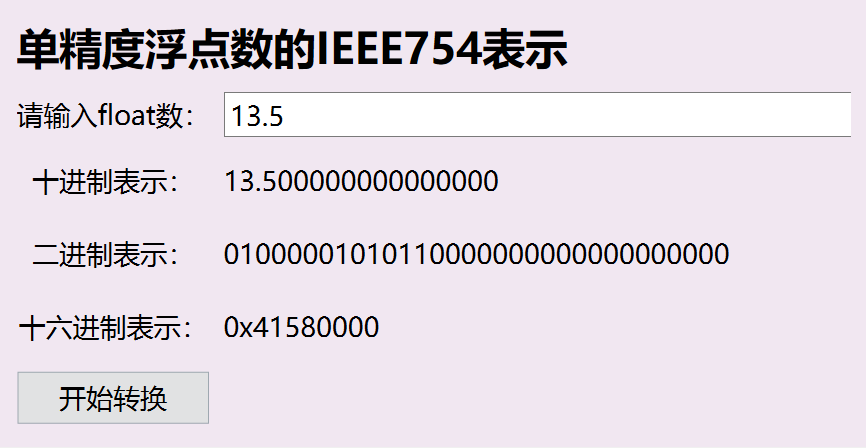
输出：整数1或0，表示x是否可以用单精度浮点数恢复

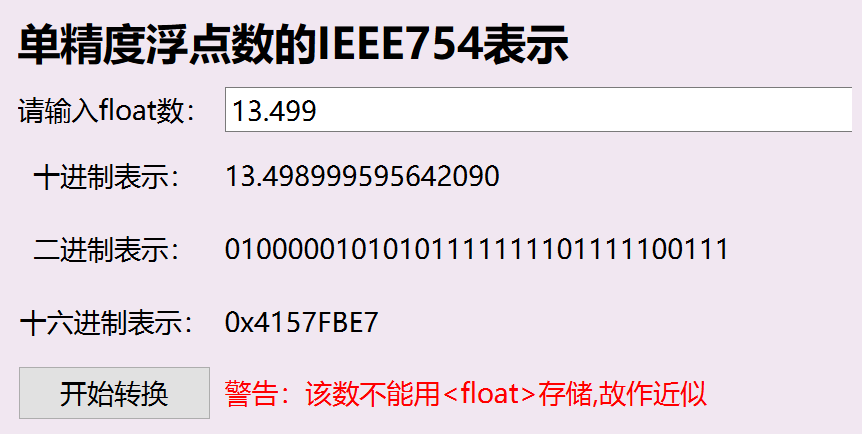
1.声明变量floatNum和recoveredNum（分别表示单精度浮点数和从单精度浮点数恢复的双精度浮点数）；

2.将x转换为floatNum，然后将floatNum转换为recoveredNum；

3.如果x与recoveredNum之间的差的绝对值小于1e-8，则返回1，表示x可以用单精度浮点数恢复；

4.否则，返回0，表示x无法用单精度浮点数恢复。





4.【二进制编码转单精度浮点数（float）】

void ComplementToFloat(QString s,double \*res)

输入：字符串s

输出：浮点数res

1.声明变量i、exponent和ans（分别表示循环计数器、指数和结果）；

2.声明字符变量c和指向字符串s的指针code；

3.将字符串s转换为字符数组code；

4.从8循环到1，计算指数exponent：将code中第i位的数值减去字符'0'，再乘以2的8-i次幂，并累加到exponent上；

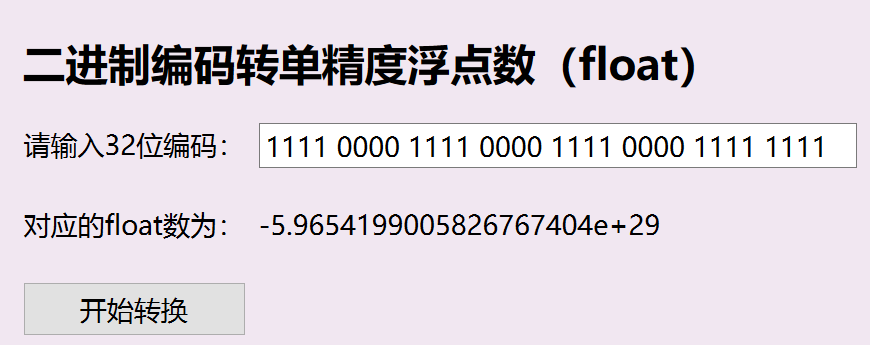
5.将exponent减去127，得到指数的真实值；

6.从31循环到9，计算小数部分的值ans：将code中第i位的数值减去字符'0'，再乘以2的-(i-8)+exponent次幂，并累加到ans上；

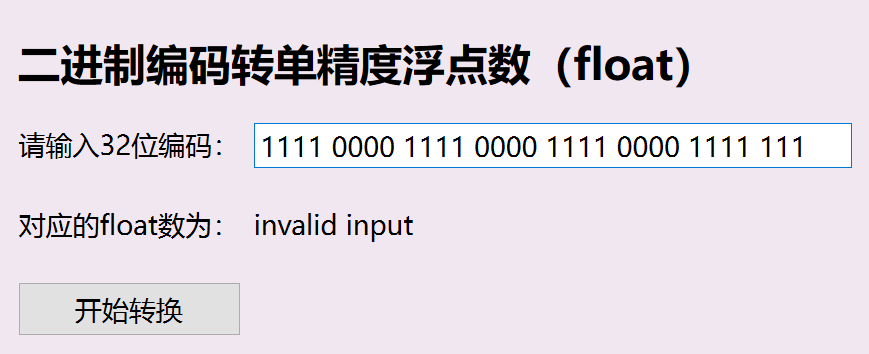
7.将2的exponent次幂加到ans中，得到最终的结果；

8.根据code的第0位确定结果的正负号，将其乘以-1或1；

9.将ans赋值给res，并将code的第32位设置为'\0'。



无效输入：（非32位/非01码）



**5.【双精度浮点数的IEEE754表示】**

**oid doubleToBinaryAndHex(double num, char\* binary, char\* hex)**

输入：双精度浮点数num

输出：二进制形式的字符串binary，十六进制形式的字符串hex

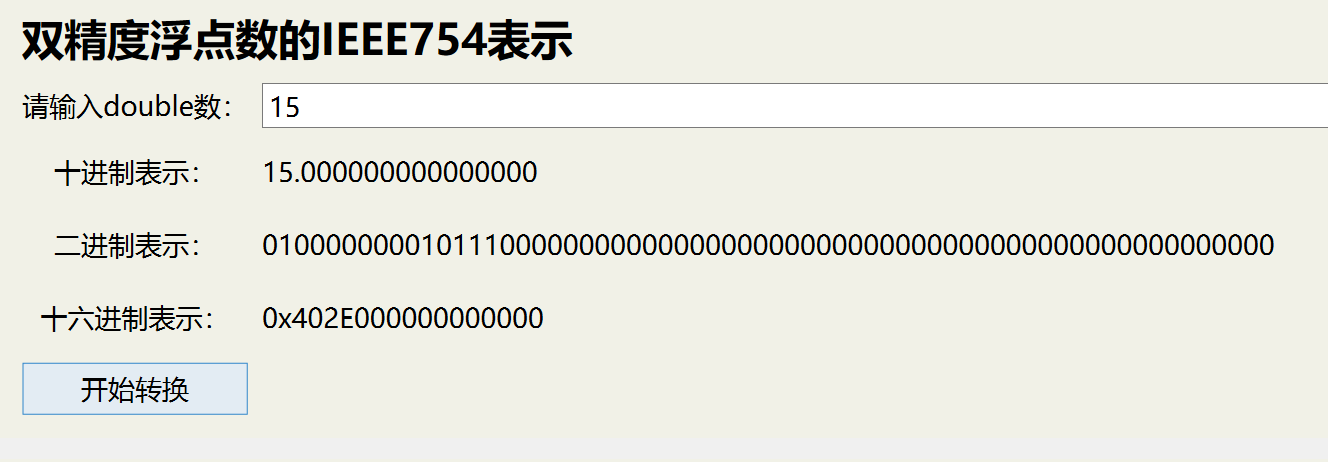
1.声明变量i、j、hex\_cp（分别表示循环计数器、循环计数器、十六进制的副本）；

2.声明long long int指针pInt，并将其指向num的地址；

3.将pInt指向的内容转化为十六进制格式，并按照格式"%016llX"存储到hex中，再将hex的最后一个字符设置为'\0'，将hex赋值给hex\_cp；

4.从0循环到15，表示16个16进制数字，对于每个16进制数字，将其转化为二进制，存储在binary中；

5.将binary的最后一个字符设为'\0'，函数结束

。

**6.【二进制编码转单精度浮点数（double）】**

**void ComplementToDouble(QString s,double \*res)**

输入：字符串s

输出：双精度浮点数res

1.声明变量i、exponent和ans（分别表示循环计数器、指数和结果）；

2.声明字符变量c和指向字符串s的指针code；

3.将字符串s转换为字符数组code；

4.从11循环到1，计算指数exponent：将code中第i位的数值减去字符'0'，再乘以2的11-i次幂，并累加到exponent上；

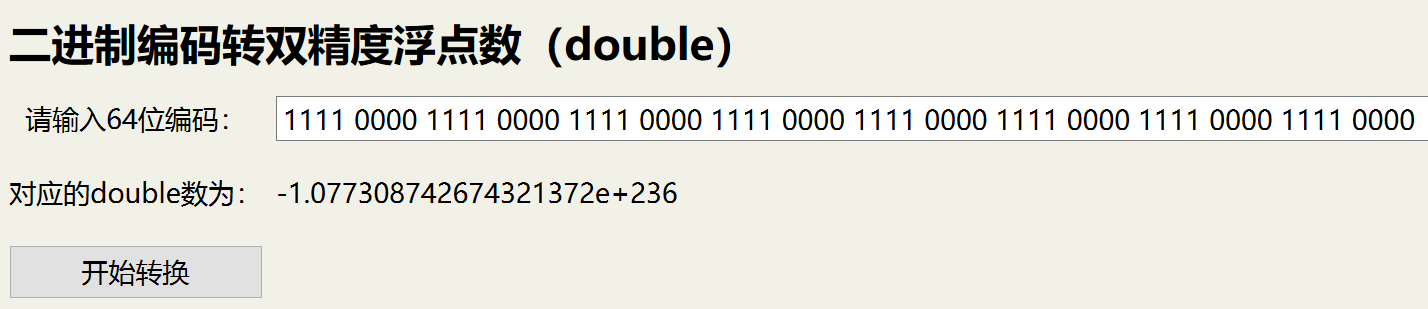
5.将exponent减去1023，得到指数的真实值；

6.从63循环到12，计算小数部分的值ans：将code中第i位的数值减去字符'0'，再乘以2的-(i-11)+exponent次幂，并累加到ans上；

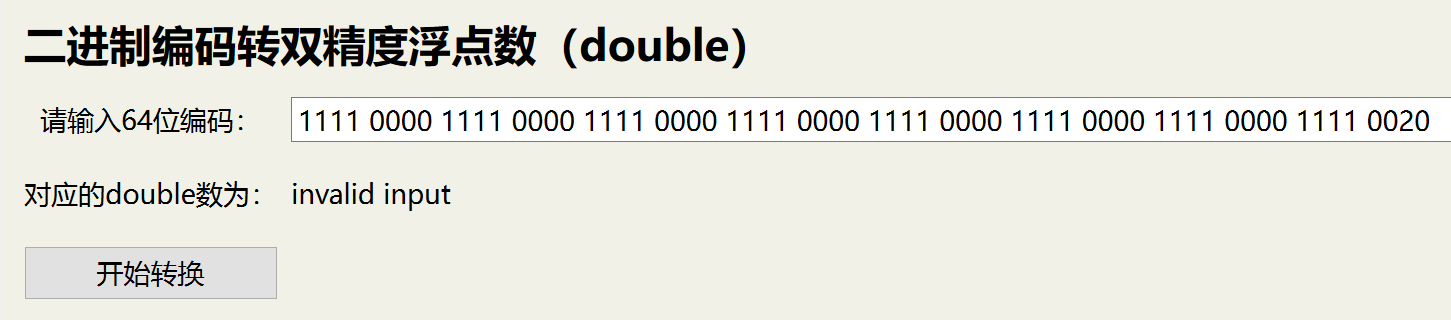
7.将2的exponent次幂加到ans中，得到最终的结果；

8.根据code的第0位确定结果的正负号，将其乘以-1或1；

9.将ans赋值给res，并将code的第65位设置为'\0'。



无效输入：（非64位/非01码）



7.【进制转换器】

* 二进制转十进制/十六进制：

void BinaryToDecHex(QString bin\_,QString \*dec\_,QString \*hex\_)

输入：二进制形式的字符串bin\_

输出：十进制形式的字符串dec\_，十六进制形式的字符串hex\_

实现步骤：

1. 声明变量c、i、j、k、n和flag（分别表示字符、循环计数器、循环计数器、循环计数器、字符串的长度和标志位）；

2. 声明long long int类型的变量dec并将其初始化为0；

3. 声明字符指针类型的变量hex和binary，并将他们分别初始化为动态内存分配(n/4+(n%4!=0)+1)个字节，并将hex的每个位置初始化为0；

4. 将bin\_转换为二进制形式的字符串并将其存储在binary中；

5. 将binary转换为十进制数并存储在dec中；

6. 从0循环到n/4+(n%4!=0)，表示每4位二进制数对应的一位十六进制数，对于每个十六进制数，将其转换成十六进制格式并存储在hex中；

7. 如果当前是第一个十六进制数，且$n\%4\neq 0$，则for循环中的代码将会特殊处理。将flag设置为0表示当前处理的是第一个十六进制数；从n%4-1开始循环到0，将最高位的n%4个二进制数转化成十六进制并存储到hex的第一个位置中，循环结束后直接进入下一个循环，不执行后面的代码；

8. 如果不是第一个十六进制数或$n\%4=0$，则从(i+flag)\*4+n%4-1开始循环到(i+flag)\*4+n%4-4，将这4个二进制数转化为十六进制格式并存储到hex的第i个位置；

9. 如果hex的第i个位置大于'9'，则将它转化为A到F中的一个字母；

10. 将十进制形式的字符串保存在dec\_中，将十六进制形式的字符串保存在hex\_中；

11. 释放hex所占用的动态内存。



* 十进制转二进制/十六进制：

void DecToBinaryHex(QString dec\_,QString \*bin\_,QString \*hex\_)

输入：十进制形式的字符串dec\_

输出：二进制形式的字符串bin\_，十六进制形式的字符串hex\_

实现步骤：

1. 声明变量i、j、k、n和flag（分别表示循环计数器、循环计数器、循环计数器、字符串的长度和标志位）；

2. 声明字符指针类型的变量binary和hex，并将他们分别初始化为动态内存分配n+1个字节和(n/4+(n%4!=0)+1)个字节，并将binary和hex的每个位置初始化为0；

3. 将dec\_转换为长整型的dec；

4. 如果dec为0，则n设为2；否则，n设为对数2的以dec为底的结果加1；

5. 从n-1循环到0，将每个二进制数存储在binary中；

6. 将binary转换成十六进制，并存储在hex中；

7. 如果当前是第一个十六进制数，且$n\%4\neq 0$，则for循环中的代码将会特殊处理。将flag设置为0表示当前处理的是第一个十六进制数；从n%4-1开始循环到0，将最高位的n%4个二进制数转化成十六进制并存储到hex的第一个位置中，循环结束后直接进入下一个循环，不执行后面的代码；

8. 如果不是第一个十六进制数或$n\%4=0$，则从(i+flag)\*4+n%4-1开始循环到(i+flag)\*4+n%4-4，将这4个二进制数转化为十六进制格式并存储到hex的第i个位置；

9. 如果hex的第i个位置大于'9'，则将它转化为A到F中的一个字母；

10. 将二进制形式的字符串保存在bin\_中，将十六进制形式的字符串保存在hex\_中；

11. 释放binary和hex所占用的动态内存。



* 十六进制转二进制/十进制：

void HexToBinaryDec(QString hex\_,QString \*bin\_,QString \*dec\_)

输入：十六进制形式的字符串hex\_

输出：二进制形式的字符串bin\_，十进制形式的字符串dec\_

实现步骤：

1. 声明变量i、j和n（分别表示循环计数器和字符串的长度）；

2. 声明字符变量c和指向字符串hex\_的指针hex；

3. 声明字符指针类型的变量binary，并将他们分别初始化为动态内存分配4\*n+1个字节，并将binary的每个位置初始化为0；

4. 将字符串hex\_转换为字符数组并将其存储在hex中；

5. 从0循环到n，表示16进制数中的每一位，将其转化为4位二进制数存储在binary中；

6. 将binary转化为十进制数，将结果存储在dec中；

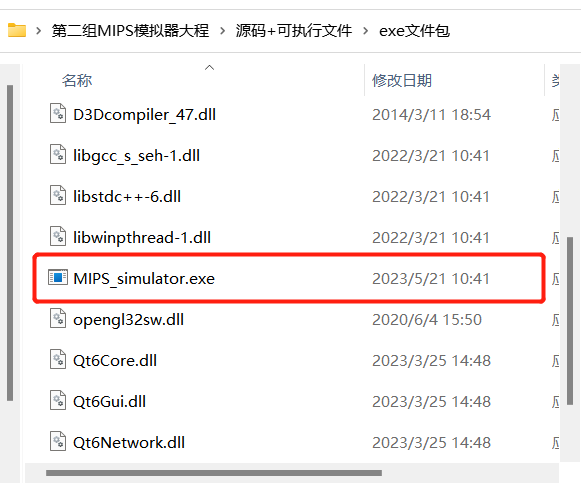
7. 将二进制形式的字符串保存在bin\_中，将十进制形式的字符串保存在dec\_中；

8. 释放binary所占用的动态内存。



1. **使用说明截图**

该项目实现了release版本的可执行文件，可在“源码+可执行程序包”下的“exe文件包”里找到，名为“MIPS\_simulator.exe”

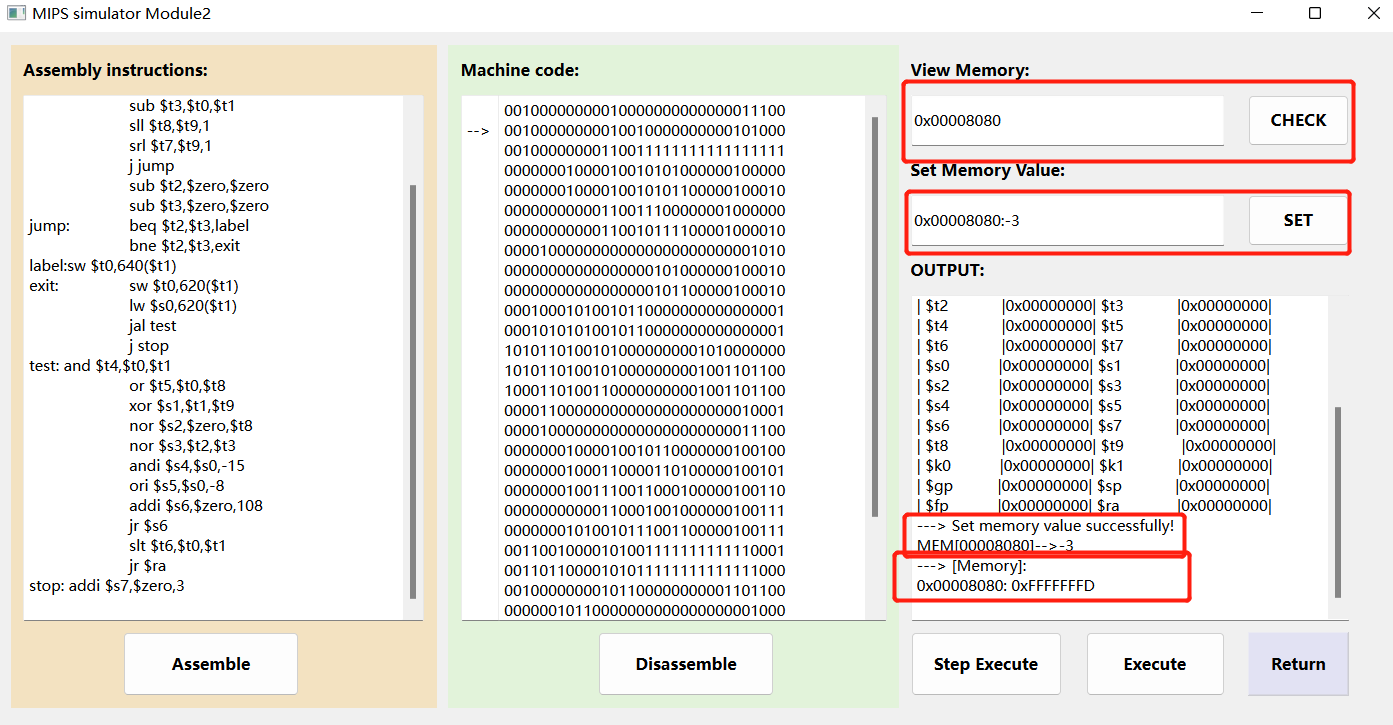


**主界面截图：**



其中点击Module1模块里的Go就会进入汇编、反汇编和模拟器的模块，点击Module2模块里的Go就会进入整数（补码）、浮点数的表示、转换和运算模块。

**Module1模块界面截图**



其中，Assemble按钮用于汇编功能（汇编指令转机器码），Disassemble按钮用于反汇编功能（机器码转汇编指令），Step Execute按钮用于单步执行（方便调试），Execute按钮用于一键执行（方便查看最后结果），SET按钮用于为指定地址的内存空间设值，CHECK按钮用于查看指定地址的内存空间的值，Return按钮用于返回主界面，其中SET 和CHECK功能都有严格的输入格式，否则会出错。

**Module2模块界面截图**



其中相关的功能介绍和使用说明在右侧的说明框里都有给出，且前面介绍该模块的时候也已经很详细的给出了用例，因而在此不再赘述。

1. **参考链接**

在实现汇编、反汇编和MIPS模拟器的时候，主要参考了CSDN上有关MIPS指令的规范和实现机理。下面是参考文章的链接：

https://blog.csdn.net/peachhhh/article/details/114376694

https://blog.csdn.net/wzbbbb/article/details/129971475

https://blog.csdn.net/weixin\_44984664/article/details/104970433