# 计算机组成课程实验报告—MIPS汇编软件实验

[计算机组成课程实验报告—MIPS汇编软件实验](#header-n593)  
 [1. 基本信息](#header-n596)  
 [1.1 小组成员信息](#header-n597)  
 [1.2 实验信息一览](#header-n614)  
 [编译器版本](#header-n615)  
 [使用已编译文件运行步骤](#header-n637)  
 [具体编译运行步骤](#header-n649)  
 [2. 题目要求](#header-n676)  
 [2.1 总体目标](#header-n677)  
 [2.2 具体要求](#header-n679)  
 [3. 实验原理](#header-n707)  
 [3.1 头文件与命名空间](#header-n708)  
 [3.2 C++面向对象特性的利用](#header-n710)  
 [MIPSAssembler类的属性](#header-n711)  
 [MIPSAssembler类的方法](#header-n752)  
 [MIPSAssembler的完整定义](#header-n803)  
 [3.3 宏的利用](#header-n805)  
 [3.4 AssembleMIPS函数的完整逻辑](#header-n818)  
 [3.5 核心思路](#header-n821)  
 [指令与寄存器](#header-n822)  
 [文件名（输入输出）](#header-n826)  
 [错误传递](#header-n836)  
 [字符串流的使用](#header-n847)  
 [4. 实验过程与结果](#header-n850)  
 [4.1 编写源码](#header-n851)  
 [4.2 编写测试数据并测试](#header-n862)  
 [正常输入](#header-n863)  
 [各种错误情况](#header-n877)  
 [Linux下使用Windows的CRLF换行方式](#header-n878)  
 [错误（不规范）的语法](#header-n887)

## 1. 基本信息

### 1.1 小组成员信息

|  |  |
| --- | --- |
| 姓名 | 学号 |
| 徐震 | 3180105504 |
| 麦昌楷 | 3180101982 |
| 钟添芸 | 3180103009 |
| 江昊翰 | 3180101995 |

### 1.2 实验信息一览

#### 编译器版本

注：

* 为了程序的完整性和跨平台性，我们在除了实验要求的平台外的其他平台做了相关的编译与运行测试。
* 为了让程序程序有更大的容错性，我们花了很大一部分精力来完善本程序的错误识别与错误处理系统。在完成基本要求：MIPS汇编器的基础上，我们的程序可以识别出用户输入中“可原谅”（例如某些位置的多余空格）以及“不可原谅”（语法错误，指令错误，寄存器错误等）的错误
  + 对于可原谅的错误，我们进行正常的输入输出。
  + 对于不可原谅的错误，我们停止程序运行，输出当前的汇编进度并进行错误报告，以便用户排查。

主要平台：

* Visual Studio 2019

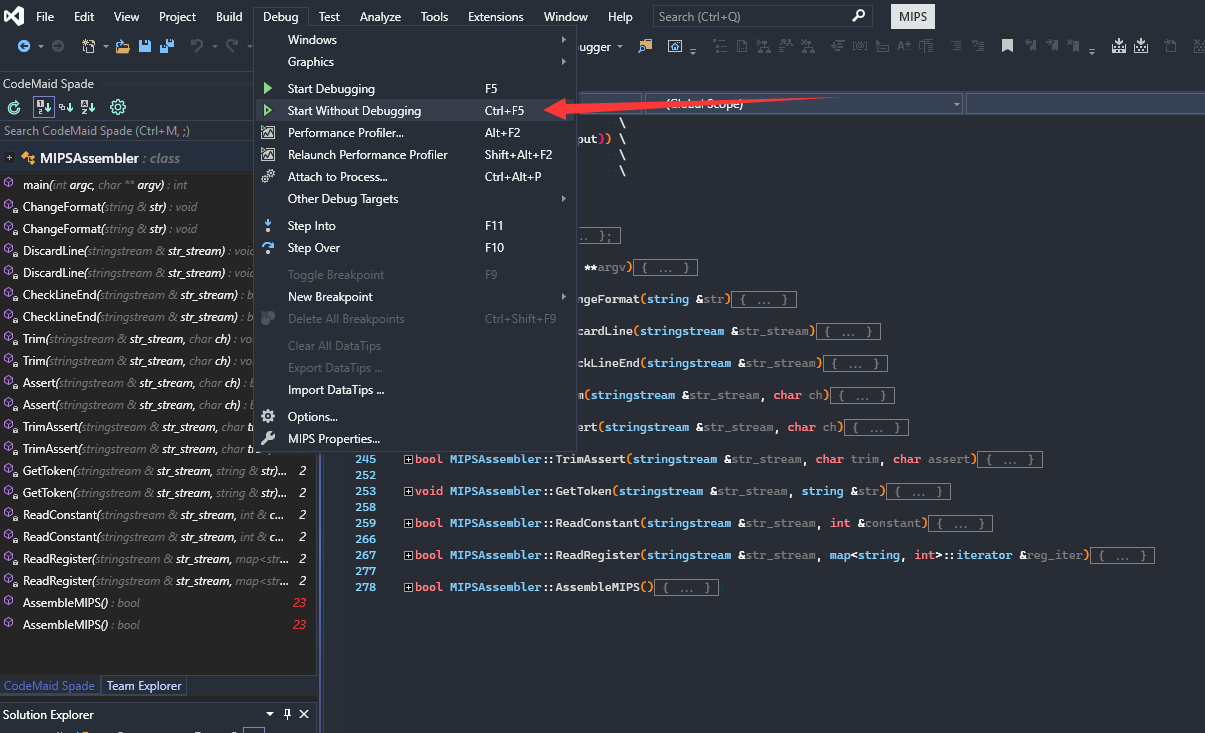
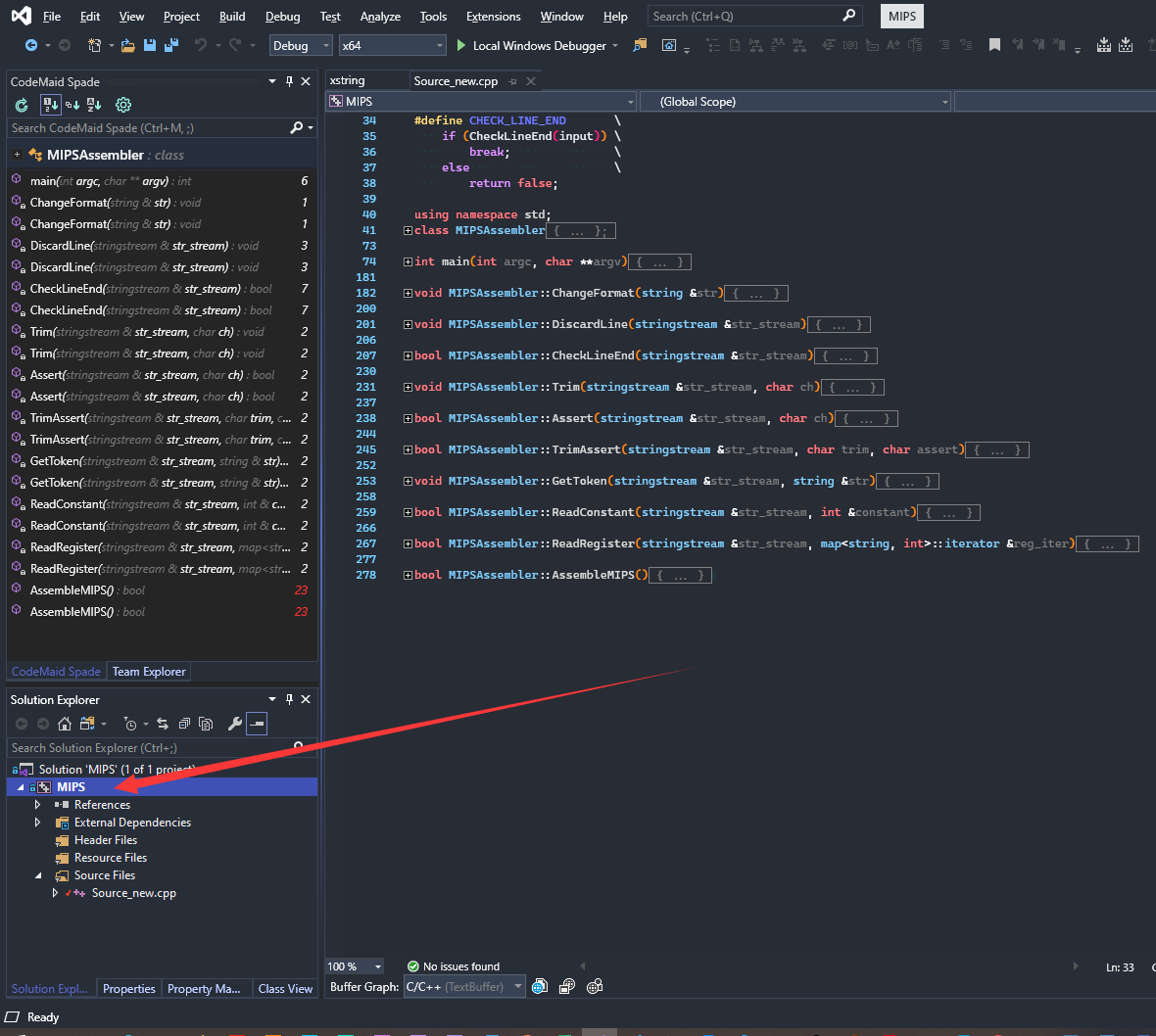
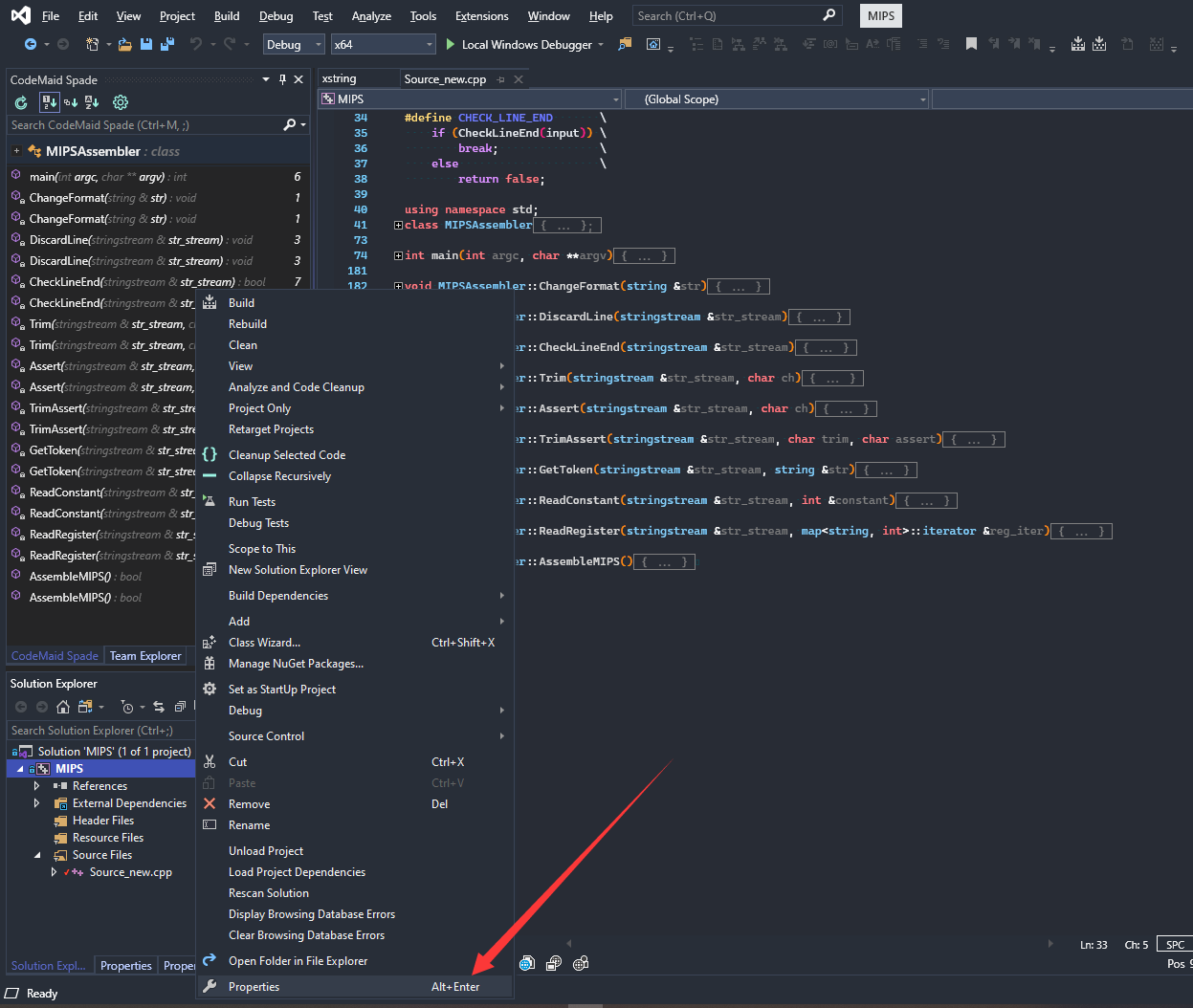
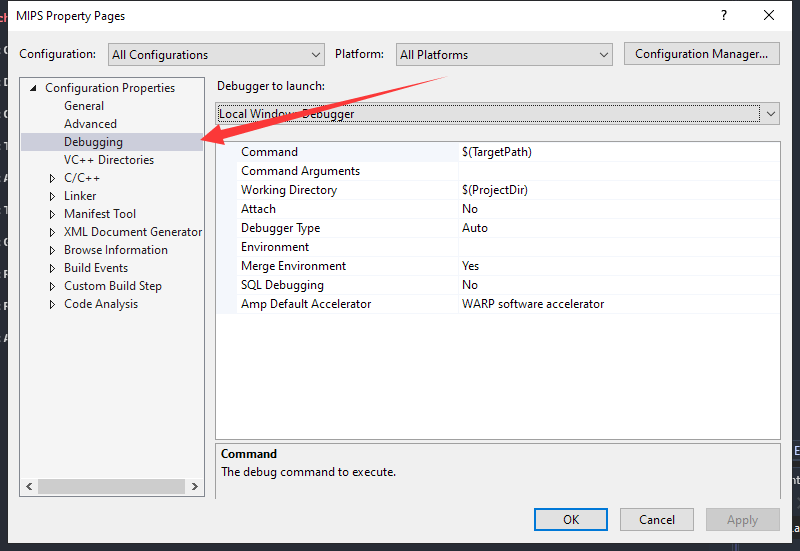
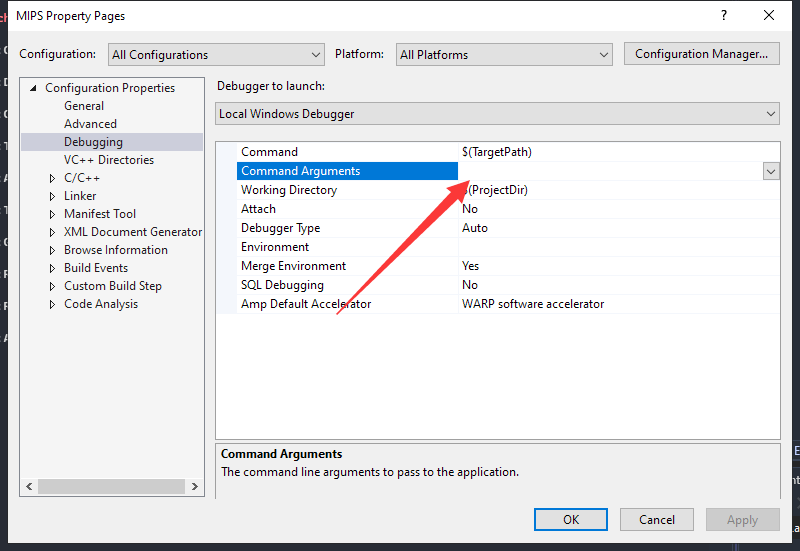
经过测试的其他平台：

* Windows: GCC 8.0.0(mingw-w64), CLANG 9.0.0(LLVM)
* Linux: GCC 7.4.0(WSL Ubuntu),GCC 9.2.1(KALI DEBIAN)

#### 使用已编译文件运行步骤

* 无命令行参数的情况下默认的输入文件名为：my1.asm，输出为my1.txt。
* 如果程序有两个命令行参数（两个以上则之取前两个），那么以第一个为输入文件名，第二个为输出文件名。
* 若是在Windows平台运行，可以直接在工程文件夹的Windows目录下打开cmd或其他命令行工具（如Windows Terminal），然后直接运行.exe文件（各种编译器的结果都可）。
* 若是在Linux平台运行，可以直接在shell环境下进入工程文件夹下的Linux目录，然后直接运行.out文件。
* 注意Windows和Linux平台使用的换行方式不一样，因此两者的对应的文件夹下的my1.asm和my1.txt对应的换行方式也不同。

#### 具体编译运行步骤

* 若使用Visual Studio 2019，可以在打开工程文件夹下的MIPS.sln文件后，直接按下快捷键Ctrl + F5编译并直接运行（不调试）。
* 
* 若使用较低版本的Visual Studio，可以新建一个空的工程，直接将工程文件夹下的Source.cpp、my1.asm、my1.txt复制到工程文件夹下，并在工程中添加Source.cpp文件，然后直接按下快捷键Ctrl + F5编译并直接运行（不调试）。
* 若要在Visual Studio环境下添加命令行参数，请采取以下步骤：
  1. 在Visual Studio的工程位置点击右键
  + 
  1. 在弹出的选项卡中选择属性一栏
  + 
  1. 在弹出的选项中选择调试
  + 
  1. 在右侧的命令行参数中输入需要的命令行参数
  + 
* 若使用其他命令行编译器如GCC，CLANG等，可以直接在源文件所在目录下输入形如如下指令的编译命令：
* # gcc  
  gcc Source.cpp -o a.out  
  # clang  
  clang Source.cpp -o a.out
* 然后直接运行：
* # 保证当前文件夹下有需要的my1.asm文件  
  # 且文件的换行方式符合系统要求：Linux LF, Windows CRLF  
  ./a.out  
  # 或者添加命令行参数  
  # 第一个参数为输入文件名  
  # 第二个参数为输出文件名  
  ./a.out my1.asm my1.txt

## 2. 题目要求

### 2.1 总体目标

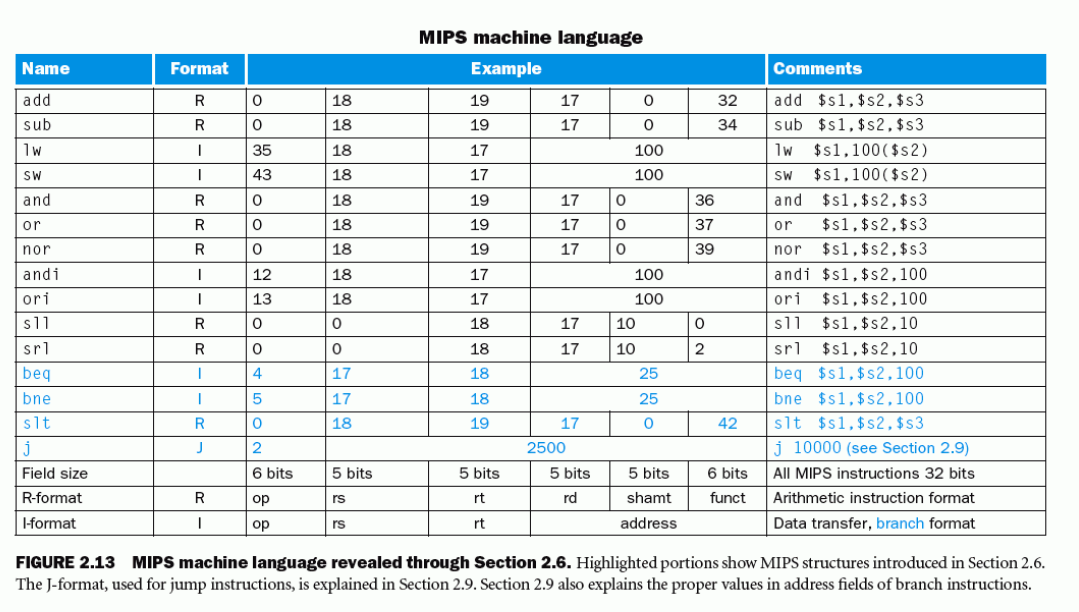
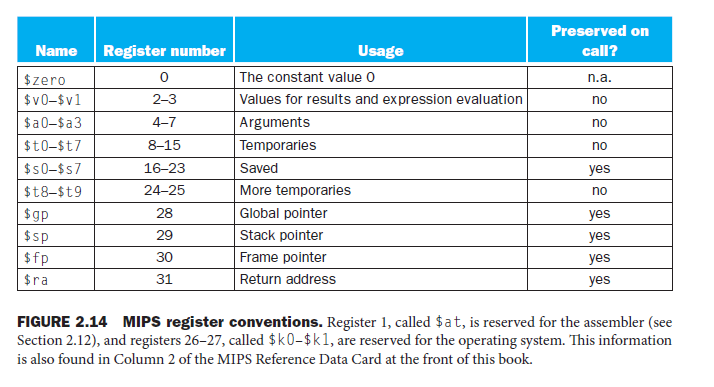
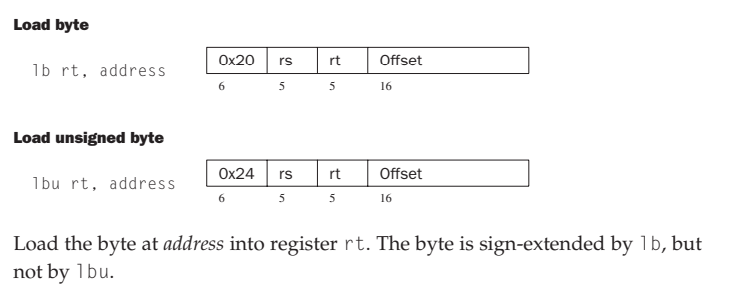
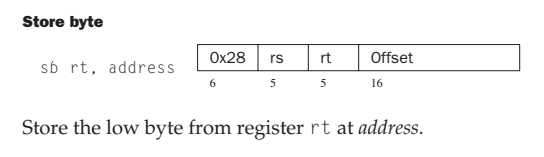
实现MIPS源程序的汇编器（MIPS源程序变机器码，MIPS源程序在文件my1.asm中，生成的机器码文件可放在my1.txt, my1.asm和my1.txt都是文本文件，不是二进制文件，放在你的可执行文件所在目录，下同。my1.asm和my1.txt这两个固定文件名固定不变。）

### 2.2 具体要求

1. 编译器：
   * visual studio 2010--2019版本的微软Visual Studio编译器，编写你的VC++程序。
   * 最近5年发布的Ubuntu Linux的GCC
   * 其他版本的编译器经老师和同学讨论后，决定并公告

* 实验报告第1章（或第1节）请说明你所用的编译器及版本，说明你的程序如何编译的各步骤说明。

1. 要求实现第三版教材图2-20的所有指令（见下面图1）、下面图2的addi指令、下面图3的lb（load byte）和sb（store byte）指令，你的汇编源程序可以无实际意义，但每条指令需要出现3次（3者的操作数差别要尽可能大）。对所有指令（共20条），除了系统专用的3个寄存器（$at,$k1,$k2）外的29个寄存器，都必须出现2次或以上。

* 
* 
* 
* 

1. 要求对实验结果截图，并说明。提交包含源程序的工程文件文件夹（删除obj文件及其他大于500KB的编译中间文件，但留下exe文件（对Windows系统的Visual Studio）或可执行文件（对Visual Studio之外的系统）、PDF或WORD格式的实验报告、my1.asm、my1.txt文件， 这些文件和目录压缩成一个压缩包文件。
2. my1.txt内容是文本，不是二进制形式的不可见字符（乱码）。My1.txt的典型格式如下：

* 1000:10001100 10001001 00000000 00000000 100011 00100 01001 00000 00000 000000  
  1004:00000000 00000000 01000000 00100000 000000 00000 00000 01000 00000 100000  
   ………
* 上面的标号1000是十进制（16进制也可以），接下去，左边32位是“8位-8位-8位-8位”的机器码，右边32位是“6位-5位-5位-5位-5位-6位”格式的机器码。

1. 你的程序需要能够由助教或老师试运行，会抽查，请提供可执行文件，典型数据文件，实验报告最后一节请提供程序如何运行的使用说明（最好有截图）。

## 3. 实验原理

### 3.1 头文件与命名空间

#include <bitset>  
#include <fstream>  
#include <iomanip>  
#include <iostream>  
#include <map>  
#include <set>  
#include <sstream>  
#include <string>  
#include <vector>  
  
using namespace std;

### 3.2 C++面向对象特性的利用

#### MIPSAssembler类的属性

我们在实验的实现过程中将汇编器抽象为一个对象。他有如下的属性：

* struct inst：一个用于方便的表示一条指令的小结构，包含
  + format：指令的结构代码（R，I或J）
  + op：指令的操作码
  + funct：指令的功能码
* // This small structure is for easily declaring different part of an instruction  
  struct inst {  
   char format; // Format or an instruction(R, I or J)  
   int op; // Opcode of an instruction  
   int funct; // Funct code of an instruction  
  };
* stringstream input, output：用于储存输入输出数据的字符串流
  + input：输入数据流
  + output：输出数据流
* stringstream input, output; // string stream for storing input and output
* map<string, inst> inst\_map：指令集，指令的字符串表示与指令的特征相对应
* map<string, inst> inst\_map; // inst\_map: Instruction Map, maps a string representation of an instruction to its corresponding format and opcode/function
* map<string, int> reg\_map：寄存器集，寄存器的字符串表示与起数字化表示相对应
* map<string, int> reg\_map; // reg\_map: Register Map, maps a string representation of a register to its corresponding register number
* int init\_line\_number = 0x1000：输出的起始行号，每编码一次指令自增4，以Byte为单位
* set<char> reserved\_char; // Reversed characters that should be considered an end for token recognition
* set<string> reserved\_char：保留的字符，本set中的所有元素会被认为是一个token的结尾（token不包括该元素）
  + 值得注意是，我们将“#”也作为保留字符，因为我们在asm文件中实现了注释（注释以“#”开头）
* set<char> reserved\_char; // Reversed characters that should be considered an end for token recognition
* bool Linux\_warning = false：与下面DiscardLine中提到的问题相对应
* bool Linux\_warning = false;
* int source\_line\_number = 1：用以统计源文件的行号，便于出错后信息的调试

#### MIPSAssembler类的方法

汇编器这一对象包含的方法有：

* void ChangeFormat(string &str)：将一组未拓展的数据拓展成实验要求的格式，例如
  + 从10001100100010010000000000000000到10001100 10001001 00000000 00000000 100011 00100 01001 00000 00000 000000
* void MIPSAssembler::ChangeFormat(string &str)  
  {  
   str.append(str); // Duplicate string  
   /\*\*  
   \* Add space correspondingly  
   \* Note that string::insert will insert things at the front of the postion specified  
   \* that is, inserting at string::end() is legal, while inserting at string::begin() is not  
   \*/  
   str.insert(8, 1, ' ');  
   str.insert(16 + 1, 1, ' ');  
   str.insert(24 + 2, 1, ' ');  
   str.insert(32 + 3, 6, ' ');  
   str.insert(38 + 9, 1, ' ');  
   str.insert(43 + 10, 1, ' ');  
   str.insert(48 + 11, 1, ' ');  
   str.insert(53 + 12, 1, ' ');  
   str.insert(58 + 13, 1, ' ');  
  }
* void DiscardLine(stringstream &str\_stream)：对字符串流进行操作，删除本行（行莫可能是回车或EOF）。
* void MIPSAssembler::DiscardLine(stringstream &str\_stream)  
  {  
   source\_line\_number++;  
   while ((str\_stream.get() != '\n') && (!str\_stream.eof()))  
   ;  
  }
* bool CheckLineEnd(stringstream &str\_stream)：检查当前的字符串流的行结尾是否符合要求。本方法在Linux与Windows平台的表现不同：
  + Windows下默认的回车换行模式是CRLF：Carrige Return Line Feed，在文件中储存为两个字符\r（0x0A）与\n（0x0D），但在各类涉及到字符串交互的操作中，Windows会讲\n解释为\r，\n两个字符。
  + Linux下的回车换行模式是LF：Line Feed，在文本中储存为\n（0x0D）一个字符，且Linux下的\n就会被解释为\n，因此在Linux下读取Windows中保存的文件时，如果换行方式为CRLF就会出现换行符的读取问题。
  + 在我们的程序中，如果试图在Linux中用Linux下编译的可执行文件读取以CRLF方式换行的文件，我们会给出警告，但是汇编工作会继续进行。
  + 如果某个文件是混合模式换行的（既有CRLF也有LF），在Linux下会得到警告，而Windows会忽略这个问题。
  + 同样的，Windows会将LF和CRLF模式的文件等同对待。
  + 注意，在DiscardLine中其实也会遇到类似的问题，但由于Windows和Linux对\n的解读方式不同，程序中这样的写法完全能胜任我们**删除本行**的要求。
* bool MIPSAssembler::CheckLineEnd(stringstream &str\_stream)  
  {  
   Trim(str\_stream, ' ');  
   if (str\_stream.peek() == '#' || str\_stream.peek() == '\n') {  
   DiscardLine(input);  
   return true;  
   } else if (str\_stream.peek() == '\r') {  
   if (!Linux\_warning) {  
   Linux\_warning = true;  
   cerr << "WARNING: Processing CRLF file on Linux system." << endl;  
   }  
   str\_stream.get();  
   if (str\_stream.peek() != '\n') {  
   cerr << "FATAL: Carrige Return encountered, however no Line Feed is found, file corrupted." << endl;  
   return false;  
   }  
   DiscardLine(input);  
   return true;  
   } else if (str\_stream.peek() != EOF) {  
   cerr << "FATAL: Unexpected line end." << endl;  
   return false;  
   } else   
   return true;  
  }
* void Trim(stringstream &str\_stream, char ch)：本方法会将参数中的字符串流接下来的所有ch清除掉，直到遇到不同于ch的字符。
* void MIPSAssembler::Trim(stringstream &str\_stream, char ch)  
  {  
   while (str\_stream.get() == ch) // Will consume an extra character, so we put it back  
   ;  
   str\_stream.unget();  
  }
* bool Assert(stringstream &str\_stream, char ch)：本方法会检测当前字符串的下一个字符是否为我们认为的，返回true如果是，如果不是返回false。
* bool MIPSAssembler::Assert(stringstream &str\_stream, char ch)  
  {  
   if (str\_stream.get() == ch) return true; // Get next character an validate it  
   cerr << "FATAL: Unable to read the character: " << ch << endl;  
   return false;  
  }
* bool TrimAssert(stringstream &str\_stream, char trim, char assert)：本方法用以处理正则表达式`\s\*,\s\*会匹配的内容，即空格，逗号，空格，通常见于寄存器间的分隔。也可以用于其他情况的处理，例如lw,sw等指令的左右括号。
* bool MIPSAssembler::TrimAssert(stringstream &str\_stream, char trim, char assert)  
  {  
   Trim(input, trim);  
   if (!Assert(str\_stream, assert)) return false;  
   Trim(input, trim);  
   return true;  
  }
* void GetToken(stringstream &str\_stream, string &str)：获取一个单字，注意，这个方法不会对空格等进行处理，所以若是没有删除多余的空格，str不会获得任何赋值而会被清空，因此本方法一般与Trim或者TrimAssert连用。
* void MIPSAssembler::GetToken(stringstream &str\_stream, string &str)  
  {  
   str.clear(); // Clear string buffer  
   while (!reserved\_char.count(str\_stream.peek())) str.push\_back(str\_stream.get()); // Get char if not reserved  
  }
* bool ReadRegister(stringstream &str\_stream, map<string, int>::iterator &reg\_iter)：GetToken并尝试将本寄存器映射到事先定义好的集合里，若失败则会报错。注意，同样不会清空空格，因此一般与Trim或者TrimAssert连用。
* bool MIPSAssembler::ReadRegister(stringstream &str\_stream, map<string, int>::iterator &reg\_iter)  
  {  
   string str;  
   GetToken(input, str);  
    
   // Return false if we cannot map the string to a register  
   if ((reg\_iter = reg\_map.find(str)) != reg\_map.end()) return true;  
   cerr << "FATAL: Unable to map register: " << str << " to int value." << endl;  
   return false;  
  }
* bool ReadConstant(stringstream &str\_stream, int &constant)：由于内部实现中用了流操作，因此会去除空格等，但在实际使用该方法的时候，我们还是将其与Trim或者TrimAssert连用。
* bool MIPSAssembler::ReadConstant(stringstream &str\_stream, int &constant)  
  {  
    
   if ((str\_stream >> constant)) return true; // Try interpreting the string into an integer  
   cerr << "FATAL: Unable to read the constant." << endl;  
   return false;  
  }
* bool AssembleMIPS()：核心方法。对输入和输出进行汇编。是本汇编器最主要的函数，通过调用上面列举的各种方法以实现我们需要的逻辑判断，字符匹配，错误处理等。也是本汇编器唯一对外暴露的方法。
  + 关于AssembleMIPS的具体实现会在下面宏定义的章节详细阐述。

#### MIPSAssembler的完整定义

class MIPSAssembler  
{  
 // This small structure is for easily declaring different part of an instruction  
 struct inst {  
 char format; // Format or an instruction(R, I or J)  
 int op; // Opcode of an instruction  
 int funct; // Funct code of an instruction  
 };  
 void ChangeFormat(string &str); // Convert tightly spaced string to the asked form  
 void DiscardLine(stringstream &str\_stream); // Discard current line until \n or EOF  
 bool CheckLineEnd(stringstream &str\_stream); // When the recognition of one full instruction is done, check current line for extra char  
 void Trim(stringstream &str\_stream, char ch); // Discard a certain character, usually space  
 bool Assert(stringstream &str\_stream, char ch); // return false if the next character is not what we expected  
 bool TrimAssert(stringstream &str\_stream, char trim, char assert); // Trim a certain char, assert if the next is not expected, then trim the char again. Usually for "\s\*,\s\*"  
 void GetToken(stringstream &str\_stream, string &str); // Get a token until reserved char is encountered. Usually a register  
 bool ReadRegister(stringstream &str\_stream, map<string, int>::iterator &reg\_iter); // Get a token and try to map it to an actual register. Returns false if map is unsuccessful  
 bool ReadConstant(stringstream &str\_stream, int &constant); // Read in a constant. Using stream  
 bool Linux\_warning = false;  
  
 public:  
 /\*\*  
 \* Using map can give us a clear interface for the instructions in case we want to add or delete or modify them  
 \* For instructions of format 'J' and 'I', we use 0 to fulfill the empty block that will not be used  
 \* We've considered using enum, but found it unnecessary  
 \*/  
 bool AssembleMIPS(); // Core method and the only method exposed. Returns false if the assembling is unsuccessful  
 stringstream input, output; // string stream for storing input and output  
 map<string, inst> inst\_map; // inst\_map: Instruction Map, maps a string representation of an instruction to its corresponding format and opcode/function  
 map<string, int> reg\_map; // reg\_map: Register Map, maps a string representation of a register to its corresponding register number  
 set<char> reserved\_char; // Reversed characters that should be considered an end for token recognition  
 int init\_line\_number = 0x1000; // The counter to add at the beginning of a line  
 int source\_line\_number = 1; // Counter for source code file line number  
};

### 3.3 宏的利用

在我们的MIPSAssembler对象中，最主要的方法是AssembleMIPS，然而为了能够正确处理各种不同的错误，本函数的逻辑变得十分复杂。在Visual Studio的Code Maid插件中本方法的复杂度已经到了20。

纵使我们已经利用了C++中的面向对象特性优化了部分复杂之处（本程序的第一个版本（未优化）AssembleMIPS的复杂度达到了50，是坚决不可接受的），这一核心函数仍旧有变得整洁的可能性，于是我们使用了C++语言中的宏来让代码变得整洁。

在AssembleMIPS中我们定义了如下的宏：

#define PROCESS\_REGISTER \  
 if (!ReadRegister(input, reg\_iter)) return false; \  
 output << bitset<5>((\*reg\_iter).second);  
  
#define PROCESS\_FIRST\_REGISTER \  
 Trim(input, ' '); \  
 PROCESS\_REGISTER  
  
#define PROCESS\_OTHER\_REGISTER(ch) \  
 if (!TrimAssert(input, ' ', (ch))) return false; \  
 PROCESS\_REGISTER  
  
#define PROCESS\_CONSTANT(width) \  
 if (!ReadConstant(input, constant)) return false; \  
 output << bitset<(width)>(constant);  
  
#define PROCESS\_FIRST\_CONSTANT(width) \  
 Trim(input, ' '); \  
 PROCESS\_CONSTANT(width)  
  
#define PROCESS\_OTHER\_CONSTANT(ch, width) \  
 if (!TrimAssert(input, ' ', (ch))) return false; \  
 PROCESS\_CONSTANT(width)

使用宏的原因如下：

* 宏就是文字内容的展开，没有额外的堆栈或参数传递问题，没有开销，并且可以对当前范围的变量进行操作。
* 最重要的是，宏可以将函数中包含返回指令的语句简化，这也是本程序经常需要用到的（检测到错误则返回false）。
* 用宏相互嵌套可以让函数的逻辑更清晰。剔除复杂的部分，提高程序的抽象程度。

### 3.4 AssembleMIPS函数的完整逻辑

程序的详细运行逻辑可以从注释/具体代码看到。

bool MIPSAssembler::AssembleMIPS()  
{  
#define PROCESS\_REGISTER ...  
#define PROCESS\_FIRST\_REGISTER ...  
#define PROCESS\_OTHER\_REGISTER(ch) ...  
#define PROCESS\_CONSTANT(width) ...  
#define PROCESS\_FIRST\_CONSTANT(width) ...  
#define PROCESS\_OTHER\_CONSTANT(ch, width) ...  
#define CHECK\_LINE\_END ...  
  
 int constant; // A constant  
 string str; // General purpose string  
 map<string, inst>::iterator inst\_iter; // Reserve the result of std::map::find  
 map<string, int>::iterator reg\_iter; // Reserve the result of std::map::find  
 while (1) {  
 // Skip comment and empty line and discard leading space  
 Trim(input, ' ');  
 if (input.peek() == '#' || input.peek() == '\n') {  
 DiscardLine(input);  
 continue;  
 } else if (input.peek() == '\r') {  
 if (!Linux\_warning) {  
 Linux\_warning = true;  
 cerr << "WARNING: Processing CRLF file on Linux system." << endl;  
 }  
 input.get();  
 if (input.peek() != '\n') {  
 cerr << "FATAL: Carrige Return encountered, however no Line Feed is found, file corrupted." << endl;  
 return false;  
 }  
 DiscardLine(input);  
 continue;  
 }  
  
 GetToken(input, str); // Try getting the first token(instruction)  
 if (str.empty() && input.peek() == EOF) return true; // If the EOF is encountered when trying to get a token, we should stop  
  
 output << setbase(16) << init\_line\_number << ':'; // Output line number  
 init\_line\_number += 4; // Increment the line number  
  
 if ((inst\_iter = inst\_map.find(str)) == inst\_map.end()) {  
 cerr << "FATAL: Unable to map instruction: \"" << str << "\" in instruction map." << endl;  
 return false;  
 } // Return false if we cannot map the string to an instruction  
 output << bitset<6>((\*inst\_iter).second.op); // Output opcode if instruction is found  
  
 switch ((\*inst\_iter).second.format) {  
 case 'R':  
 if ((\*inst\_iter).first == "jr") {  
  
 PROCESS\_FIRST\_REGISTER  
 for (int i = 0; i < 3; i++) output << bitset<5>(0); // Zero fill  
  
 } else if (((\*inst\_iter).first == "sll") || ((\*inst\_iter).first == "srl")) {  
  
 // Similar implementation as the I format(only these two instructions are of I format)  
 output << bitset<5>(0); // First five bit of sll and srl is filled with zero  
 PROCESS\_FIRST\_REGISTER  
 PROCESS\_OTHER\_REGISTER(',')  
 PROCESS\_OTHER\_CONSTANT(',', 5)  
  
 } else {  
  
 // Process the first register  
 PROCESS\_FIRST\_REGISTER  
  
 // Process two more register  
 PROCESS\_OTHER\_REGISTER(',')  
 PROCESS\_OTHER\_REGISTER(',')  
  
 output << bitset<5>(0); // Output shamt  
 }  
  
 // Output funct code of an instruction  
 output << bitset<6>((\*inst\_iter).second.funct);  
 break;  
  
 case 'I':  
  
 if (set<string>({"lw", "sw", "lb", "sb"}).count((\*inst\_iter).first)) {  
  
 PROCESS\_FIRST\_REGISTER  
 PROCESS\_OTHER\_CONSTANT(',', 0)  
 PROCESS\_OTHER\_REGISTER('(')  
 output << bitset<16>(constant);  
 if (!TrimAssert(input, ' ', ')')) return false;  
  
 } else {  
  
 PROCESS\_FIRST\_REGISTER  
 PROCESS\_OTHER\_REGISTER(',')  
 PROCESS\_OTHER\_CONSTANT(',', 16)  
 }  
 break;  
  
 case 'J':  
 Trim(input, ' ');  
 // Read in the constant(immediate value)  
 if (!ReadConstant(input, constant)) return false;  
 output << bitset<26>(constant);  
 break;  
 }  
 if (!CheckLineEnd(input)) return false;  
 // Splitting the machine code we've generated  
 output << endl; // Append a newline for getline to work  
 output.seekg(-32 - 1, ios::end); // Change stream input pointer  
 getline(output, str); // Store whole line  
 ChangeFormat(str); // Reformat  
 output.seekp(-32 - 1, ios::end); // Change stream output pointer  
 output << str; // Append the formatted string  
 output << endl; // previous newline character is consumed by getline  
 }  
 return true; // Assembly success  
}

### 3.5 核心思路

#### 指令与寄存器

为让我们的MIPSAssembler正常工作，我们在主程序中应首先将其实例化。然后为需要的常数进行赋值：

// Initializing constants  
A.init\_line\_number = 0x1000;  
A.reserved\_char = {',', '#', '(', ')', ' ', EOF, '\n', '\r'};  
A.inst\_map = {  
 {"add", {'R', 0, 32}},  
 {"sub", {'R', 0, 34}},  
 {"lw", {'I', 35, 0}},  
 {"sw", {'I', 43, 0}},  
 {"and", {'R', 0, 36}},  
 {"or", {'R', 0, 37}},  
 {"nor", {'R', 0, 39}},  
 {"andi",{'I', 12, 0}},  
 {"ori", {'I', 13, 0}},  
 {"sll", {'R', 0, 0}},  
 {"srl", {'R', 0, 2}},  
 {"beq", {'I', 4, 0}},  
 {"bne", {'I', 5, 0}},  
 {"slt", {'R', 0, 42}},  
 {"j", {'J', 2, 0}},  
 {"jr", {'R', 0, 8}},  
 {"jal", {'J', 3, 0}},  
 {"addi",{'I', 8, 0}},  
 {"lb", {'I', 32, 0}},  
 {"sb", {'I', 40, 0}},  
};  
A.reg\_map = {  
 {"$zero",0},  
 {"$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},  
};

注意，在inst\_map中我们对I类和J的指令的funct域也进行了赋值，虽然他们在实际的汇编过程中不会被使用。

#### 文件名（输入输出）

我们使用了命令行参数，当然，如果你选择不使用参数或者只使用一个，没有被赋值的文件名会采用默认值。

赋值的顺序是第一个参数对应输入文件名，第二个参数对应输出文件名。

他们的默认值分别为：

1. 输入：my1.asm
2. 输出：my1.txt

int main(int argc, char \*\*argv)  
{  
 string input\_file\_name = "my1.asm";  
 string output\_file\_name = "my1.txt";  
 if (argc >= 2) input\_file\_name = argv[1];  
 if (argc >= 3) output\_file\_name = argv[2];  
 MIPSAssembler A; // Instantiate the MIPS assembler  
 ifstream my1asm\_file(input\_file\_name); // open input file  
 if (!my1asm\_file.is\_open()) {  
 cerr << "FATAL: Unable to open the file specified";  
 return 2;  
 }  
 A.input << my1asm\_file.rdbuf(); // load filestream content  
 my1asm\_file.close(); // Close input file  
  
 // Open output file  
 ofstream my1txt\_file(output\_file\_name);  
 if (!my1txt\_file.is\_open()) {  
 cerr << "FATAL: Unable to open the file specified.";  
 return 3;  
 }  
 ...  
 ...  
}

#### 错误传递

在本程序中，错误是通过返回值（布尔值）传递的。例如：

* main函数发现AssembleMIPS返回false就知道汇编失败了，会进行DUMP等一系列操作：
* cerr << "FATAL: Illegal instruction or format. Check your syntax." << endl; // Error prompt  
   A.output.seekg(0, ios::beg); // Change output stream pointer position  
   cout << "DUMP: Current line of input stringstream is:" << endl; // Dump current input string stream  
   A.input.clear();  
   while (A.input.peek() != '\n' && A.input.tellg()) A.input.unget();  
   A.input.get();  
   string buffer;  
   getline(A.input, buffer);  
   cout << buffer << endl; // Dump current input line  
   cout << "DUMP: Current line number is: " << A.source\_line\_number << endl; // Dump current line number  
   cout << "DUMP: Current buffer of output stringstream is:" << endl; // Dump current output string stream  
   cout << A.output.rdbuf() << endl; // Dump current output string stream  
    
   output\_file << A.output.rdbuf(); // Dump the stringstream to file  
   output\_file.close(); // Close output file  
   return 1;
* 如果返回true则进行字符串流转换，关闭打开的文件等操作：
* A.output.seekg(0, ios::beg); // Change output stream pointer position  
  output\_file << A.output.rdbuf(); // Save output string stream to file stream  
  output\_file.close(); // Close output file  
  cout << "Assembly successful." << endl; // Successful prompt  
  cout << "Input file is: \"" << input\_file\_name << '\"' << endl;  
  cout << "Output file is: \"" << output\_file\_name << '\"' << endl;  
  return 0;
* 同样的，在AssembleMIPS函数中，各种不同的调用之间也通过bool类型的返回值进行交流。

#### 字符串流的使用

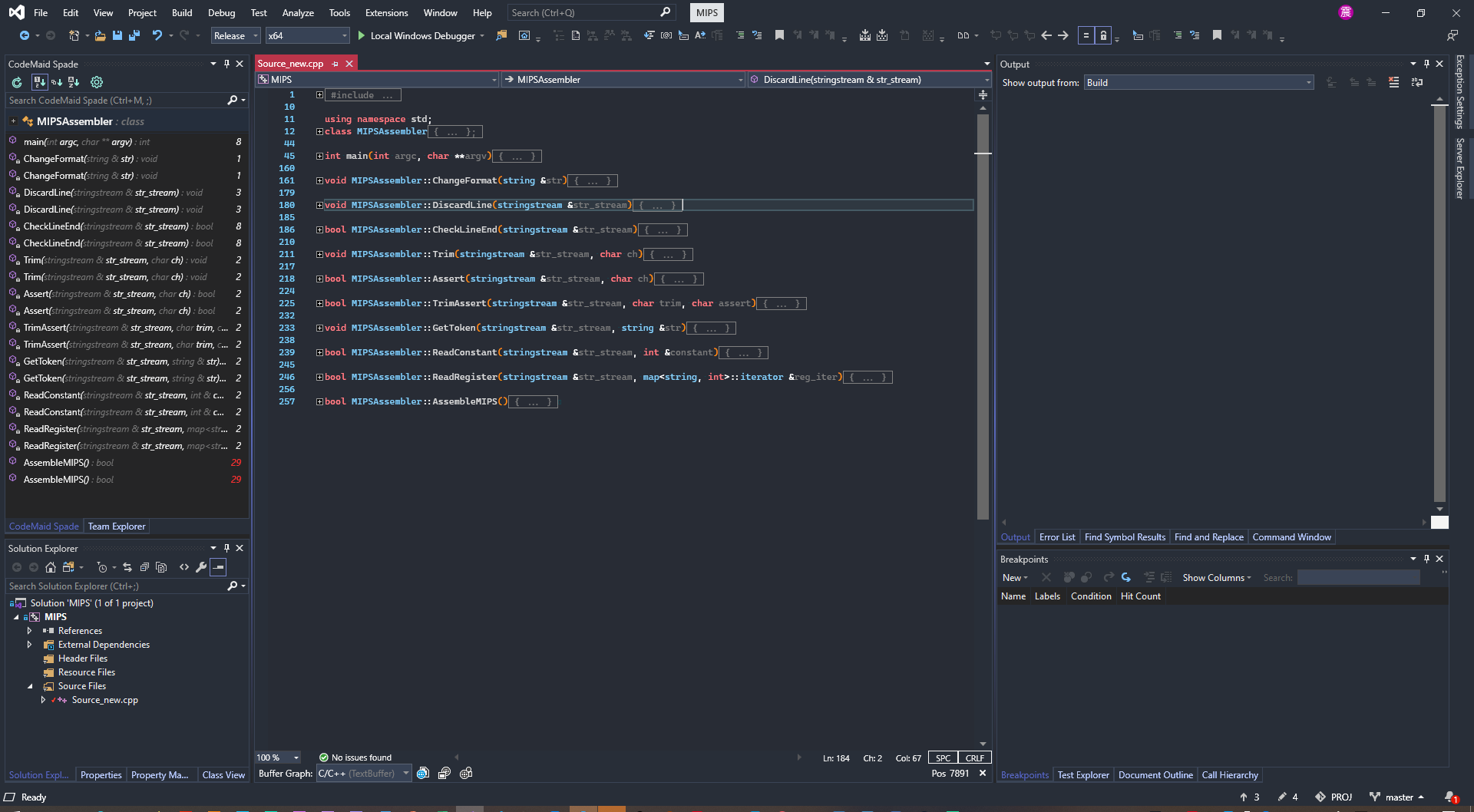
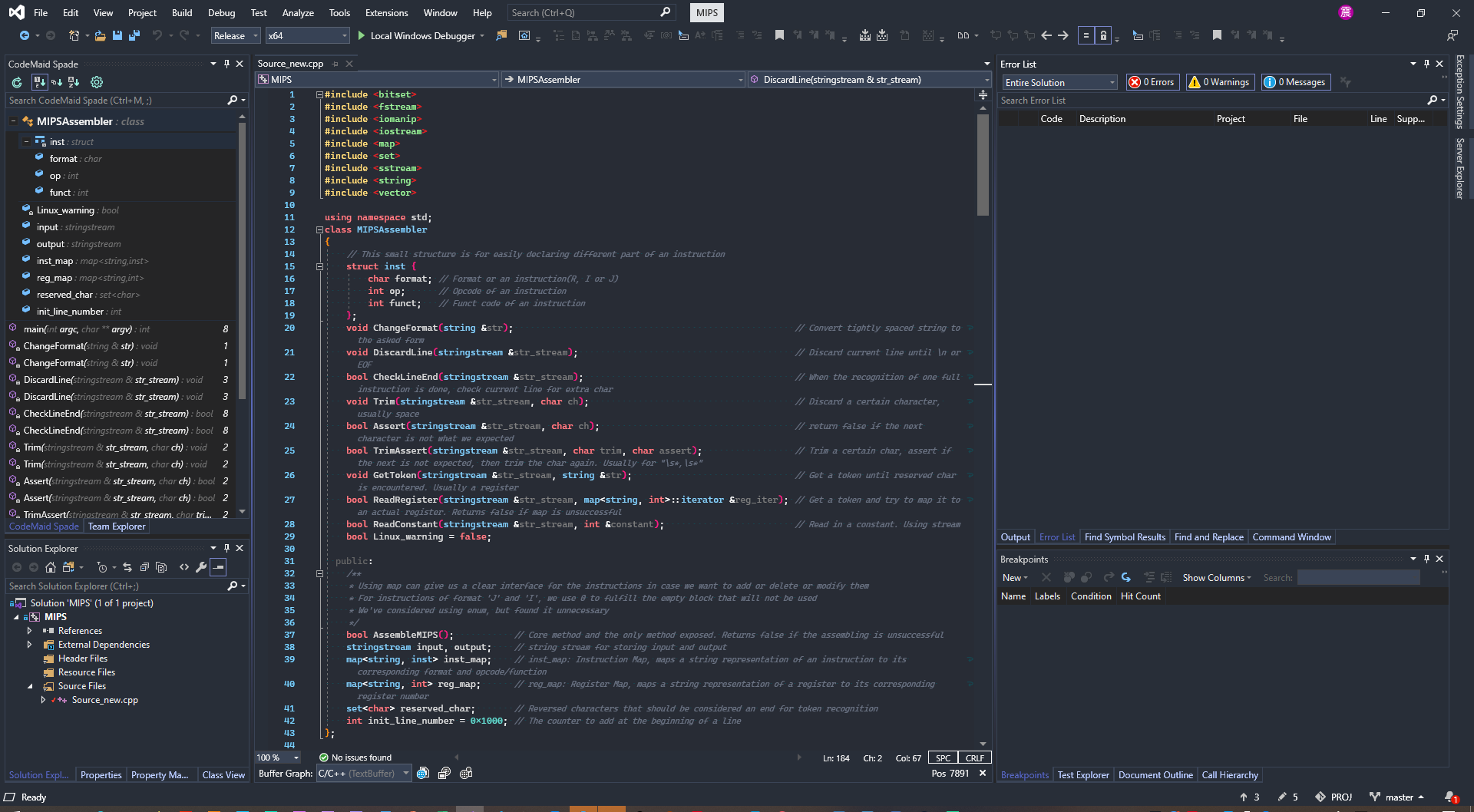
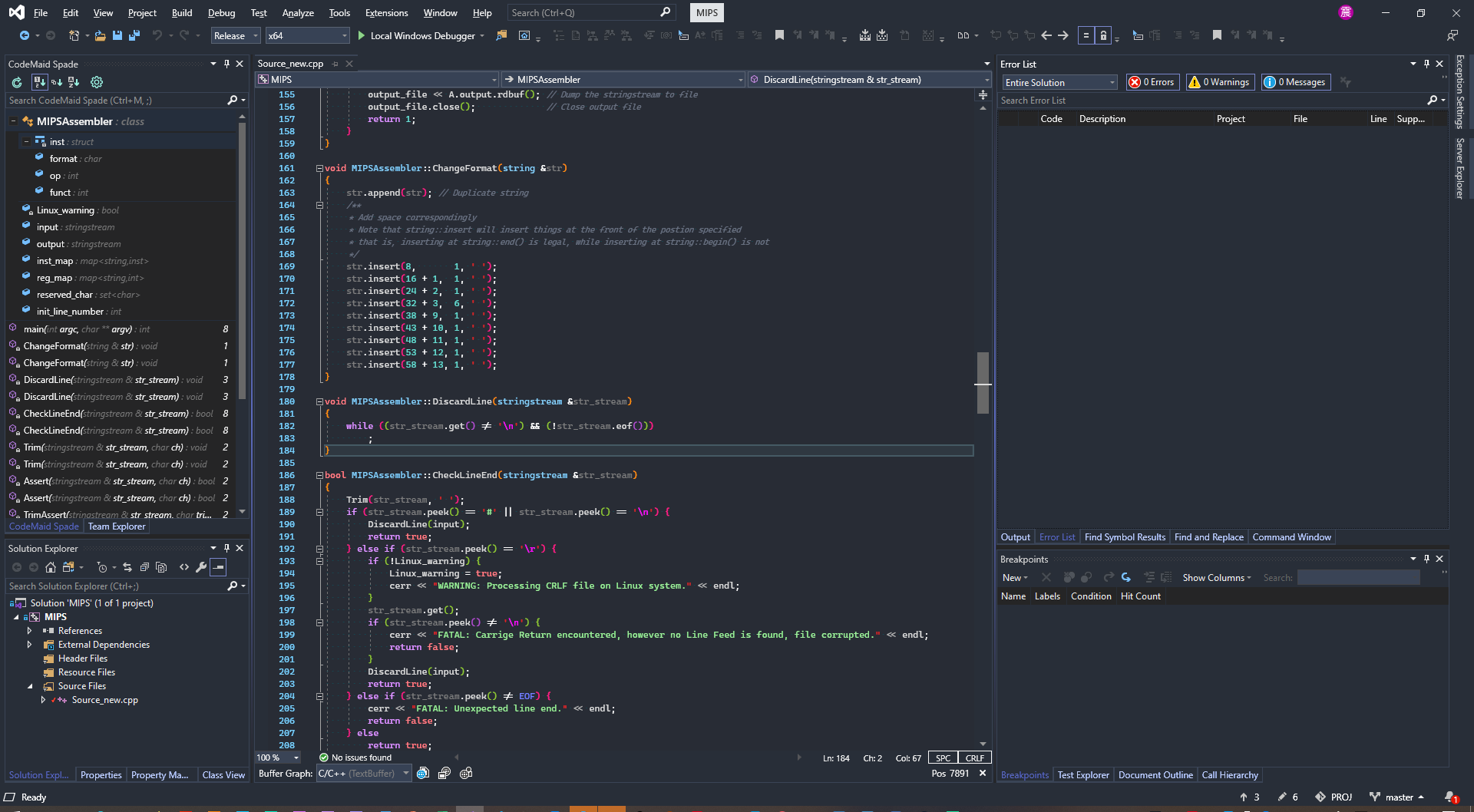
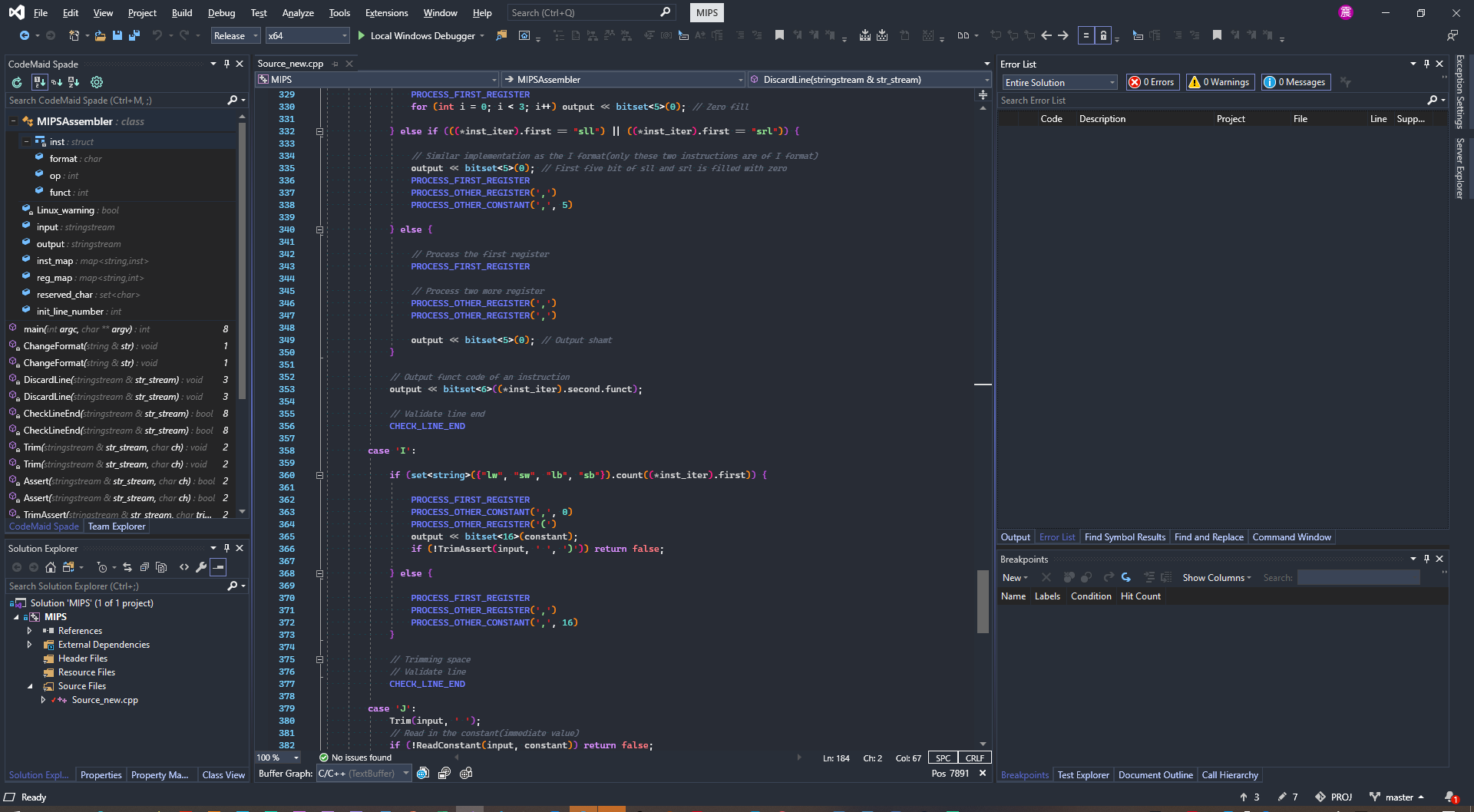
在本程序中，我们使用了字符串流这一种储存在内存中的对象。虽然它的各种操作和文件流类似，但是使用字符串流给了我们快速操作的可能。（读取内存中的内容远比读取硬盘上的文件快得多）

并且结合着面向对象的思想，我们完全可以在读取文件非常大的时候将输入在main函数中分割，并多次调用AssembleMIPS，这样就实现了大文件的处理。（在现在的版本中暂时没有进行这样的操作，但这种设计方式给了我们这样操作的可能性）

## 4. 实验过程与结果

### 4.1 编写源码

在代码编辑器中编写上述源码。

* 编写好代码框架
* 
* 添加具体内容
* 
* 
* 

### 4.2 编写测试数据并测试

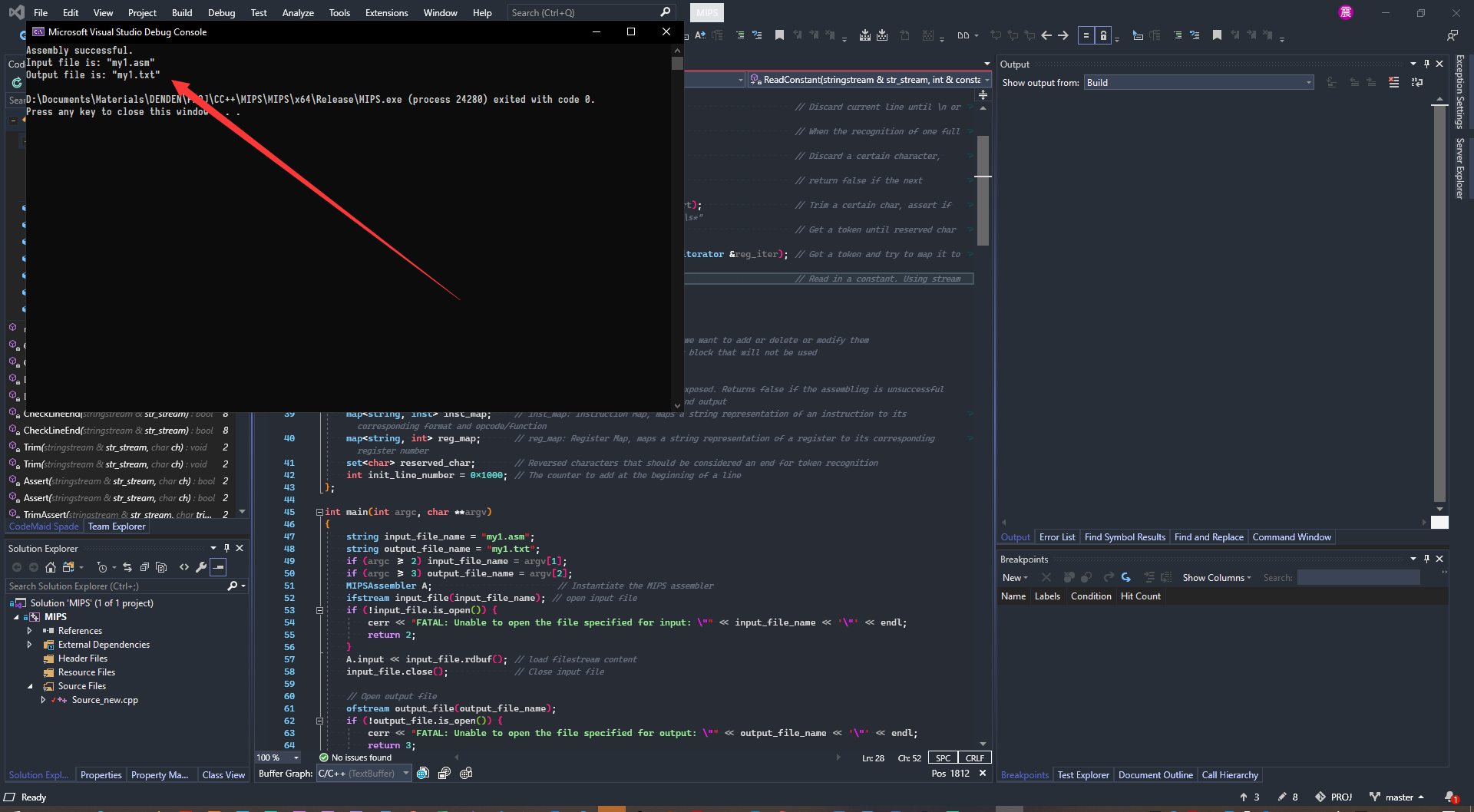
#### 正常输入

以普通格式测试所有涉及到的指令和寄存器以及常数输入。

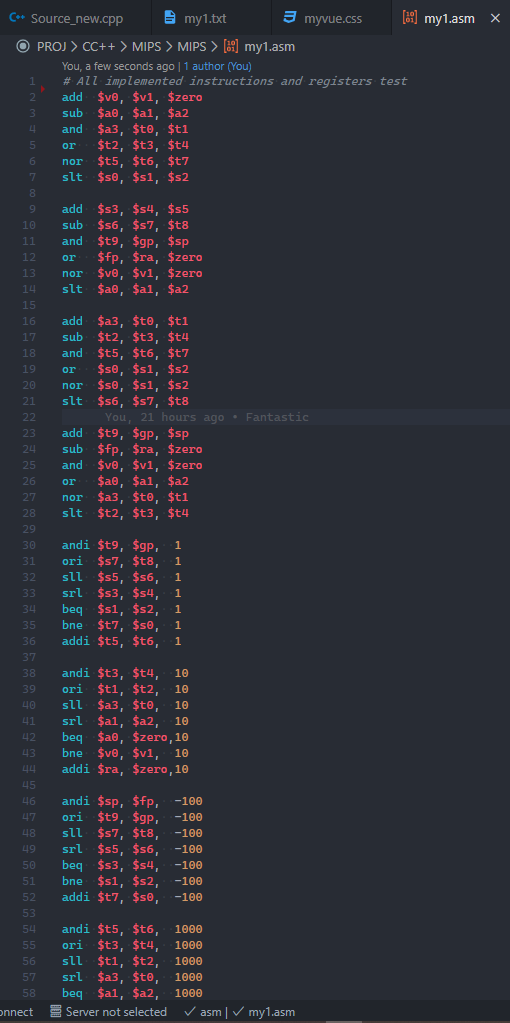
* 注意在此测试数据中，我们同时测试了空行和注释功能。

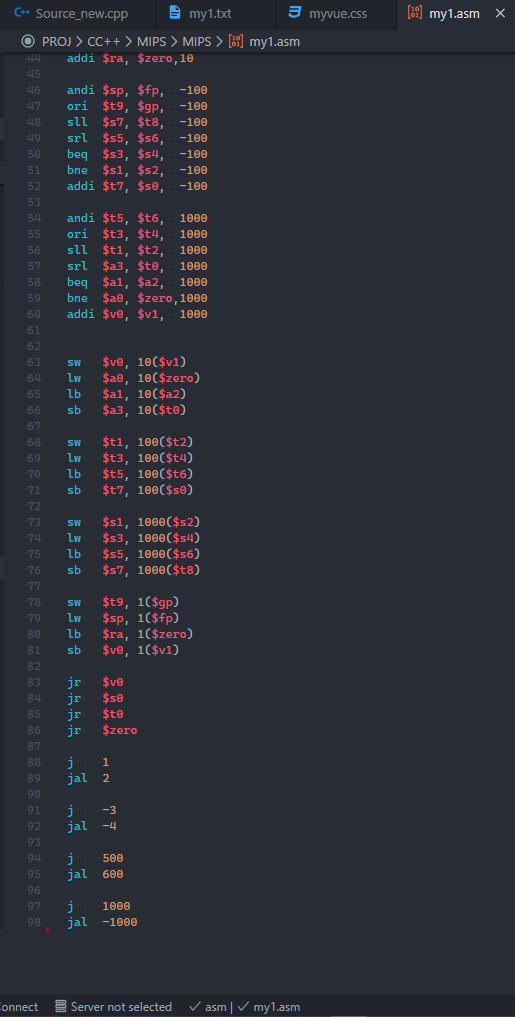
# All implemented instructions and registers test  
add $v0, $v1, $zero  
sub $a0, $a1, $a2  
and $a3, $t0, $t1  
or $t2, $t3, $t4  
nor $t5, $t6, $t7  
slt $s0, $s1, $s2  
  
add $s3, $s4, $s5  
sub $s6, $s7, $t8  
and $t9, $gp, $sp  
or $fp, $ra, $zero  
nor $v0, $v1, $zero  
slt $a0, $a1, $a2  
  
add $a3, $t0, $t1  
sub $t2, $t3, $t4  
and $t5, $t6, $t7  
or $s0, $s1, $s2  
nor $s0, $s1, $s2  
slt $s6, $s7, $t8  
  
add $t9, $gp, $sp  
sub $fp, $ra, $zero  
and $v0, $v1, $zero  
or $a0, $a1, $a2  
nor $a3, $t0, $t1  
slt $t2, $t3, $t4  
  
andi $t9, $gp, 1  
ori $s7, $t8, 1  
sll $s5, $s6, 1  
srl $s3, $s4, 1  
beq $s1, $s2, 1  
bne $t7, $s0, 1  
addi $t5, $t6, 1  
  
andi $t3, $t4, 10  
ori $t1, $t2, 10  
sll $a3, $t0, 10  
srl $a1, $a2, 10  
beq $a0, $zero,10  
bne $v0, $v1, 10  
addi $ra, $zero,10  
  
andi $sp, $fp, -100  
ori $t9, $gp, -100  
sll $s7, $t8, -100  
srl $s5, $s6, -100  
beq $s3, $s4, -100  
bne $s1, $s2, -100  
addi $t7, $s0, -100  
  
andi $t5, $t6, 1000  
ori $t3, $t4, 1000  
sll $t1, $t2, 1000  
srl $a3, $t0, 1000  
beq $a1, $a2, 1000  
bne $a0, $zero,1000  
addi $v0, $v1, 1000  
  
  
sw $v0, 10($v1)  
lw $a0, 10($zero)  
lb $a1, 10($a2)  
sb $a3, 10($t0)  
   
sw $t1, 100($t2)  
lw $t3, 100($t4)  
lb $t5, 100($t6)  
sb $t7, 100($s0)  
   
sw $s1, 1000($s2)  
lw $s3, 1000($s4)  
lb $s5, 1000($s6)  
sb $s7, 1000($t8)  
   
sw $t9, 1($gp)  
lw $sp, 1($fp)  
lb $ra, 1($zero)  
sb $v0, 1($v1)  
  
jr $v0  
jr $s0  
jr $t0  
jr $zero  
  
j 1  
jal 2  
  
j -3  
jal -4  
  
j 500  
jal 600  
  
j 1000  
jal -1000

运行截图：

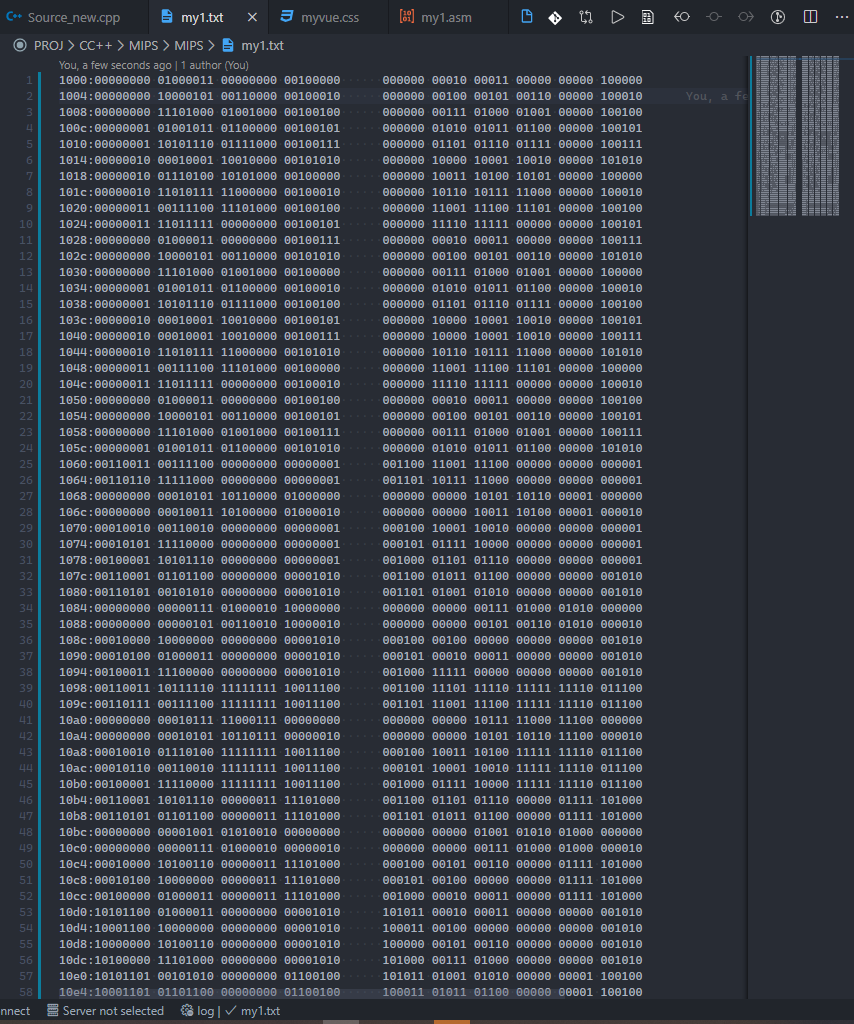


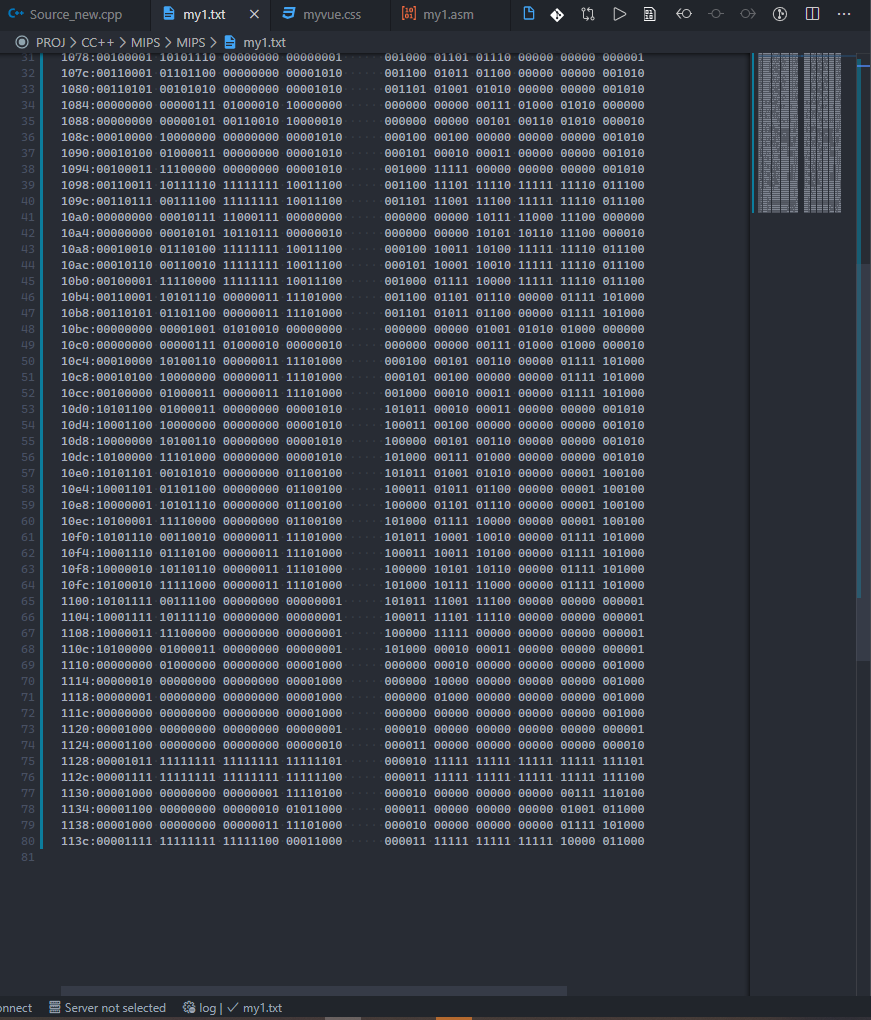
输入文件内容截图：





汇编结果截图：



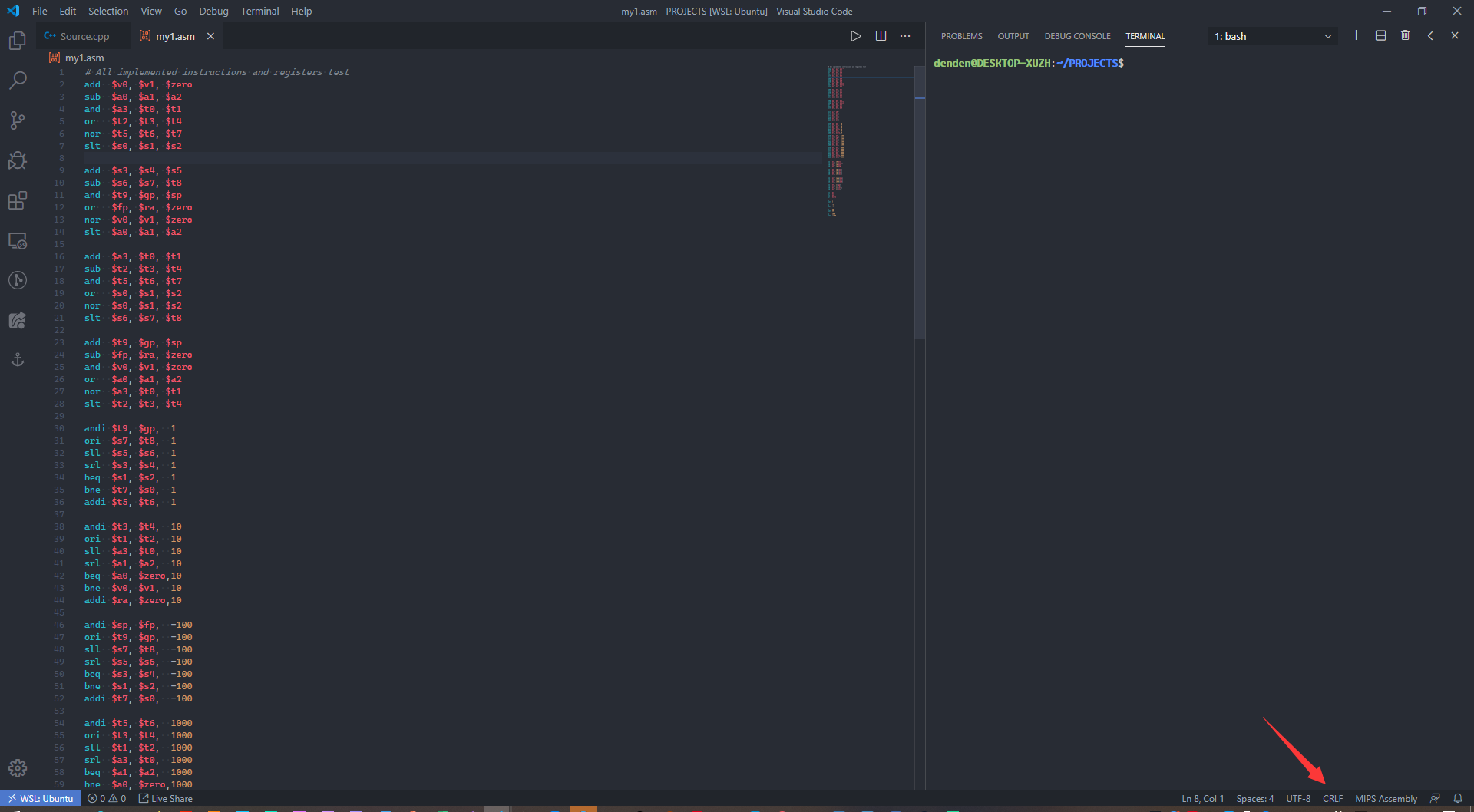


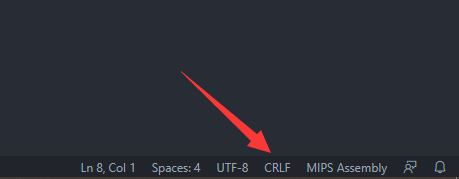
#### 各种错误情况

##### Linux下使用Windows的CRLF换行方式

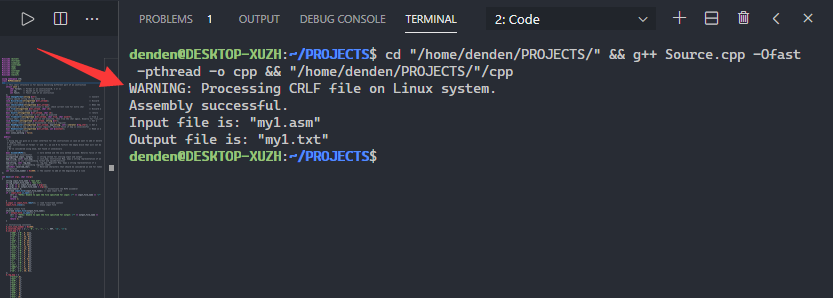
等级：WARNING

输入文件内容

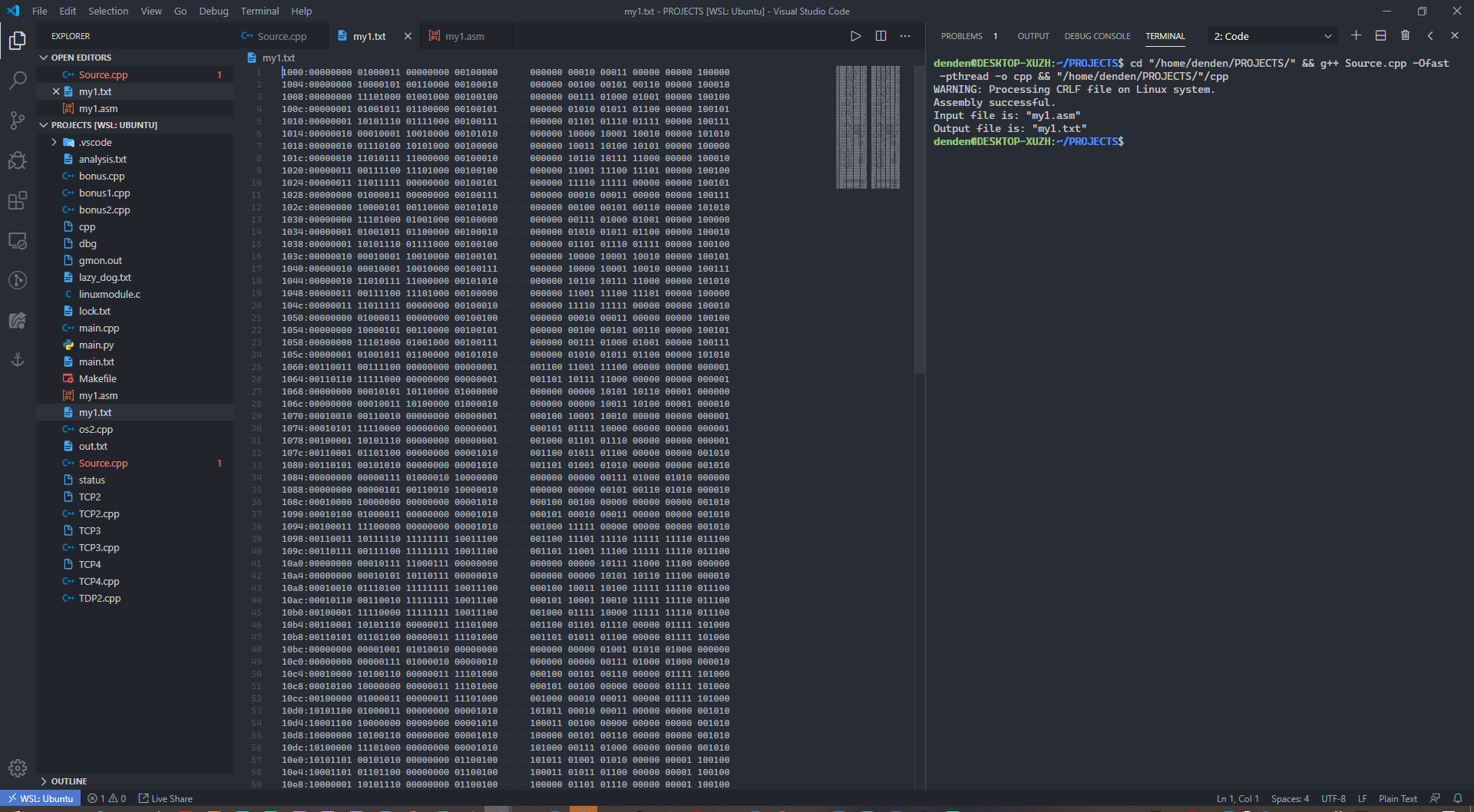




控制台输出



进行汇编后的文件内容

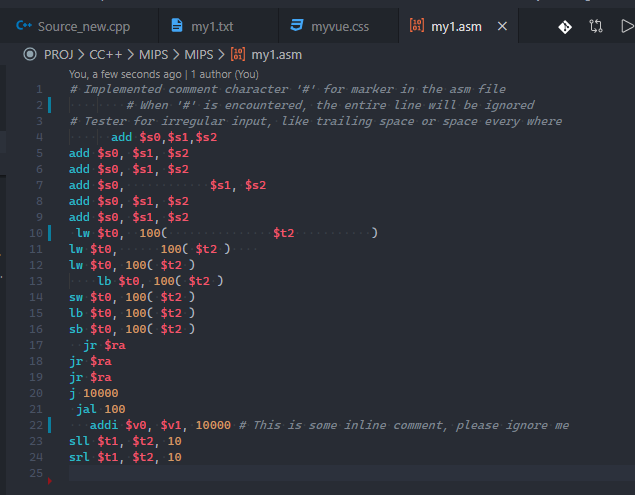


##### 错误（不规范）的语法

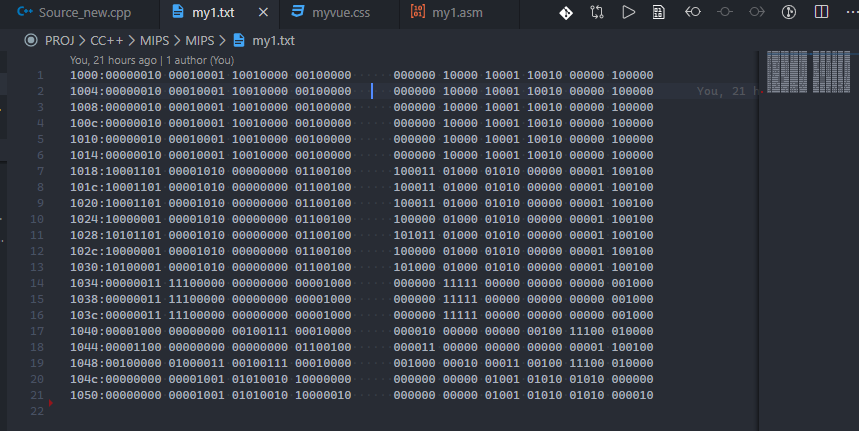
各类警告错误的列举

# Implemented comment character '#' for marker in the asm file  
 # When '#' is encountered, the entire line will be ignored  
 # Tester for irregular input, like trailing space or space every where  
 add $s0,$s1,$s2  
 add $s0, $s1, $s2  
 add $s0, $s1, $s2  
 add $s0, $s1, $s2  
 add $s0, $s1, $s2  
 add $s0, $s1, $s2  
 lw $t0, 100( $t2 )  
 lw $t0, 100( $t2 )   
 lw $t0, 100( $t2 )  
 lb $t0, 100( $t2 )  
 sw $t0, 100( $t2 )  
 lb $t0, 100( $t2 )  
 sb $t0, 100( $t2 )  
 jr $ra  
 jr $ra  
 jr $ra  
 j 10000  
 jal 100  
 addi $v0, $v1, 10000 # This is some inline comment, please ignore me  
 sll $t1, $t2, 10  
 srl $t1, $t2, 10

输入文件的截图

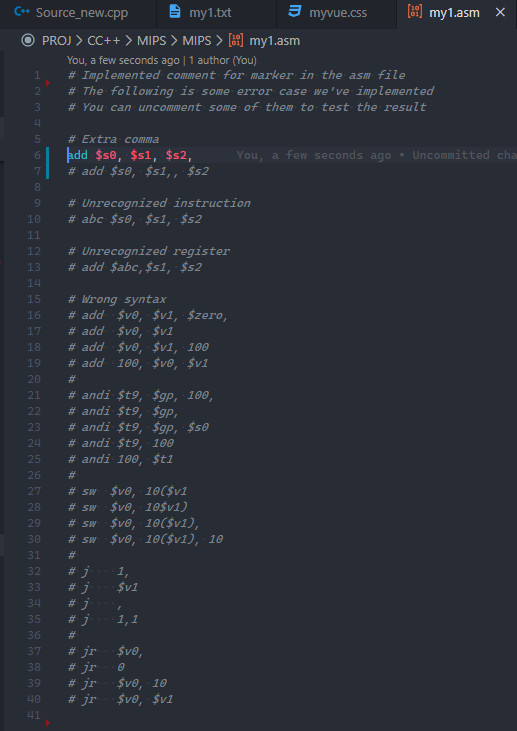
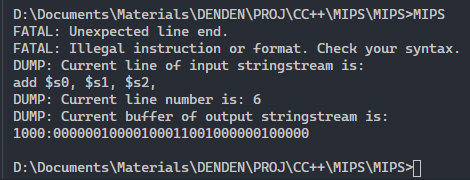
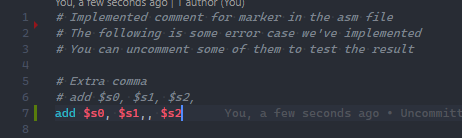
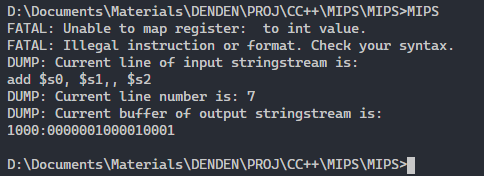
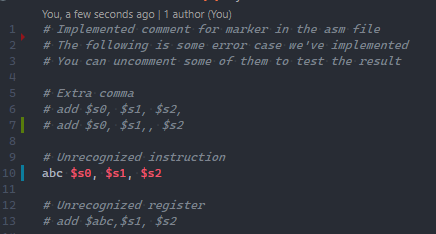
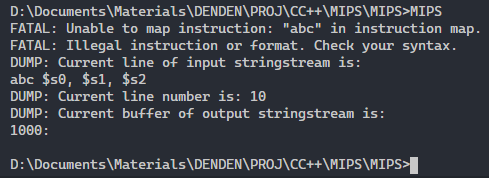
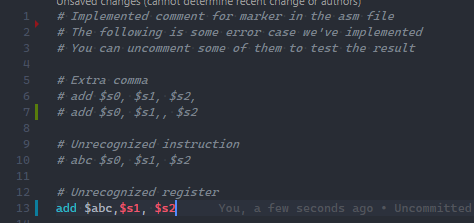
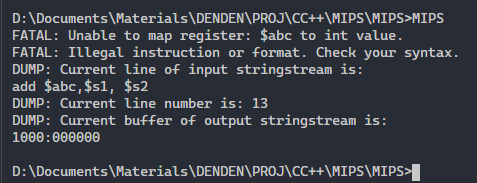
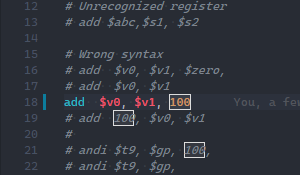
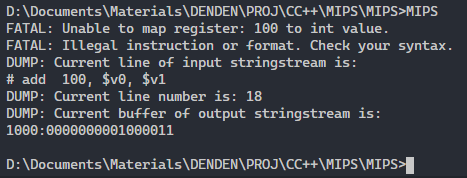
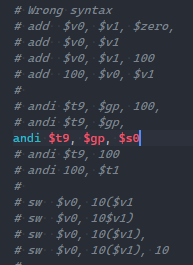
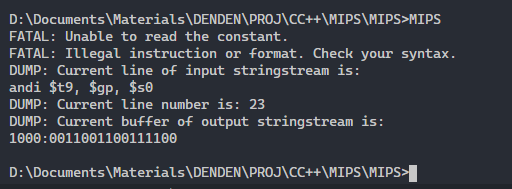
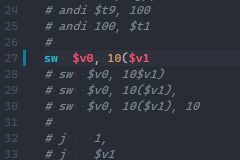
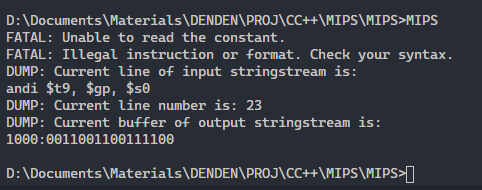
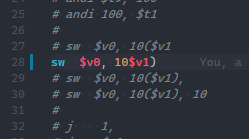
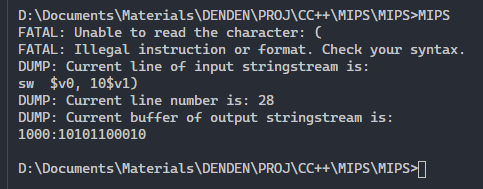
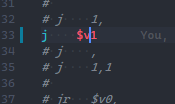
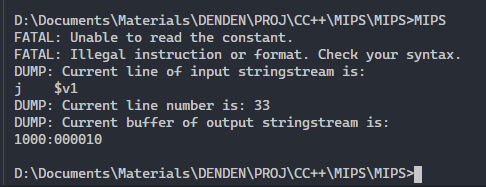
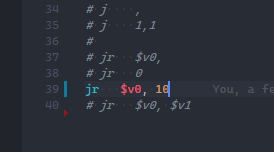
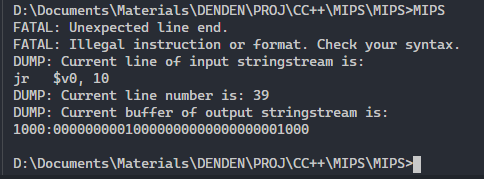


输出文件的截图



各类致命错误语法的列举

# Extra comma  
 # add $s0, $s1, $s2,  
 # add $s0, $s1,, $s2  
   
 # Unrecognized instruction  
 # abc $s0, $s1, $s2  
   
 # Unrecognized register  
 # add $abc,$s1, $s2  
   
 # Wrong syntax(Unable to match with already defined syntax)  
 # add $v0, $v1, $zero,  
 # add $v0, $v1  
 # add $v0, $v1, 100  
 # add 100, $v0, $v1  
 #   
 # andi $t9, $gp, 100,  
 # andi $t9, $gp,  
 # andi $t9, $gp, $s0  
 # andi $t9, 100  
 # andi 100, $t1  
 #   
 # sw $v0, 10($v1  
 # sw $v0, 10$v1)  
 # sw $v0, 10($v1),  
 # sw $v0, 10($v1), 10  
 #   
 # j 1,  
 # j $v1  
 # j ,  
 # j 1,1  
 #   
 # jr $v0,  
 # jr 0  
 # jr $v0, 10  
 # jr $v0, $v1

* 额外的逗号
* 
* 
* 
* 
* 无法识别的指令
* 
* 
* 无法识别的寄存器
* 
* 
* 错误的指令格式（部分）
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 
* 

## 5. 附录：实现框图

* 