The System Binary Tools

Supei Yang, Lan Qin, Yuqian Mao, Luyao Gao, Yao Tong, Zhiyuan Wan

May 31, 2020

Contents

1 Sunmary for clients

1.1 Overview

Our team achieved a simple but complete set of system binary tools for a high level programming language based on C++. We implemented components including editor, compiler, assembler, loader and simulator. Eventually, following the user manual, the user should be able to see our work and accomplishments.

Our final work presented itself as a Qt Mega project, which consists of several children projects that have seperate but connected functions. Upon running our Mega project, the user should have the following experiences. Firstly, an editor window will pop up and the user is indicated to enter his or her own program there. Secondly, when the user feels like finishing the editing, he or she can use the save button to store his program in a selected folder. Thirdly, our compiler is activated and pick up the stored program and begins compilation process, it will produce an equivalent program but in MIPS assembly language. Fourthly, assembler will read the file output by the compiler and convert it into a binary file. Thirdly, the program, now in binary format, will be loaded into the virtual memory and be executed by the simulator. This is the end of our flow.

1.2 User manual

Attention! Do not change the file folder name!

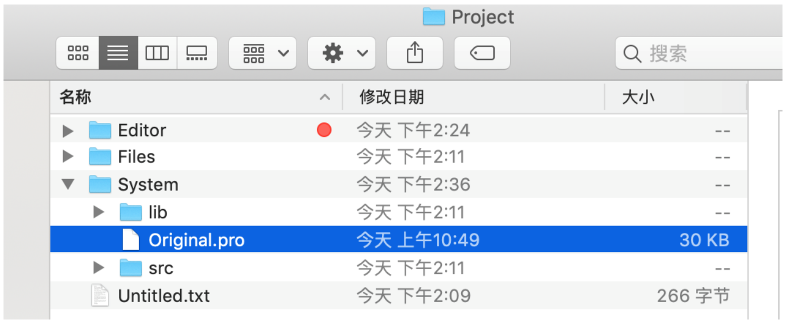
The following guides the user through the process of deploying our project, we aslo made a demo video available at

`https://cuhko365-my.sharepoint.com/:v:/g/personal/118010073_link_cuhk_edu_cn/EdASXJhSaB9HpB4hc1ux0nYBE6n1qX8jKQZitle58mAMjQ?e=ciH8pl`

- **Installation:** Download our Qt Mega project to your two personal computer. We have two versions available depending on your operating system, one for Windows 10, the other for MacOS. Folder for MacOS is named "Project", folder for Windows 10 is named "Project_Win". A Qt environment is required, version is expected to be the latest.

- **Prepare our project: (Mac as example)**

  1. Find and click Original.pro, and build this Qt project.

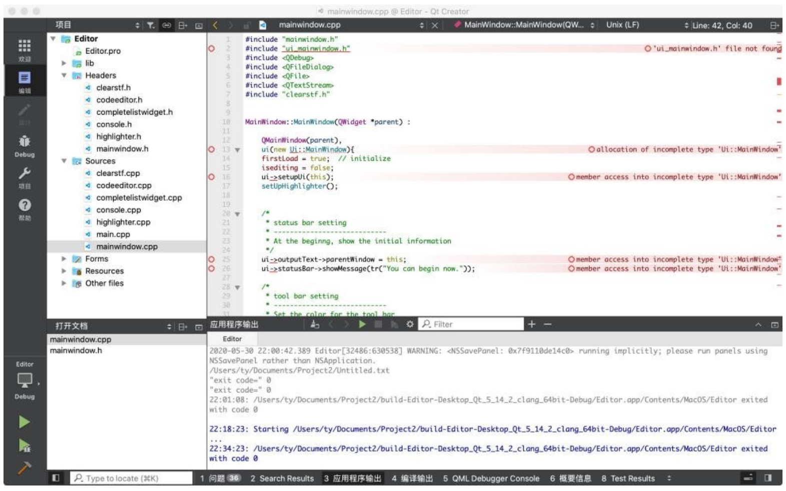


Project

名称	修改日期	大小
Editor	今天 下午2:24	--
Files	今天 下午2:11	--
System	今天 下午2:36	--
lib	今天 下午2:11	--
Original.pro	今天 上午10:49	30 KB
src	今天 下午2:11	--
Untitled.txt	今天 下午2:09	266 字节

Figure 1: Original.pro

Notice, due to a lack of Debug folder to optimize memory, the following scenario might occur, if so, do not worry, close it and build it again.



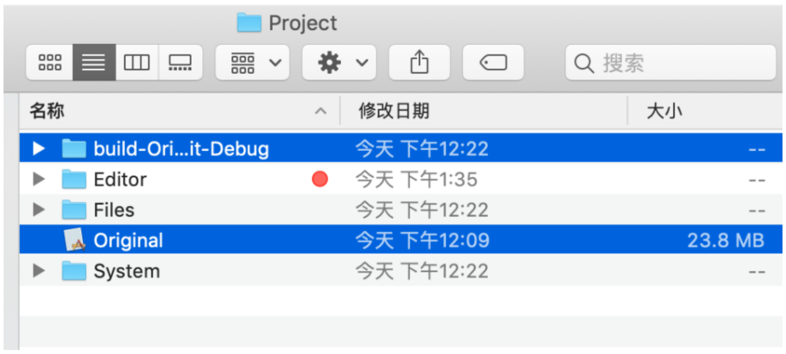
mainwindow.cpp @ Editor - Qt Creator

```
#include "mainwindow.h"
#include "ui_mainwindow.h"                          'ui_mainwindow.h' file not found
#include <QDebug>
#include <QFileDialog>
#include <QFile>
#include <QTextStream>
#include "clearstf.h"


MainWindow::MainWindow(QWidget *parent) :

    QMainWindow(parent),
    ui(new Ui::MainWindow){                         allocation of incomplete type 'Ui::MainWindow'
    firstLoad = true;  // initialize
    isediting = false;
    ui->setupUi(this);                              member access into incomplete type 'Ui::MainWindow'
    setUpHighlighter();


    /*
     * status bar setting
     * --------------------------
     * At the beginng, show the initial information
     */
    ui->outputText->parentWindow = this;            member access into incomplete type 'Ui::MainWindow'
    ui->statusBar->showMessage(tr("You can begin now."));  member access into incomplete type 'Ui::MainWindow'

    /*
     * tool bar setting
     * --------------------------
     * Set the color for the tool bar
```

```
2020-05-30 22:00:42.389 Editor[32486:630538] WARNING: <NSSavePanel: 0x7f9110de14c0> running implicitly; please run panels using
NSSavePanel rather than NSApplication.
/Users/ty/Documents/Project2/Untitled.txt
"exit code=" 0
"exit code=" 0
22:01:08: /Users/ty/Documents/Project2/build-Editor-Desktop_Qt_5_14_2_clang_64bit-Debug/Editor.app/Contents/MacOS/Editor exited
with code 0

22:18:23: Starting /Users/ty/Documents/Project2/build-Editor-Desktop_Qt_5_14_2_clang_64bit-Debug/Editor.app/Contents/MacOS/Editor
...
22:34:23: /Users/ty/Documents/Project2/build-Editor-Desktop_Qt_5_14_2_clang_64bit-Debug/Editor.app/Contents/MacOS/Editor exited
with code 0
```

Figure 2: Scenario 1

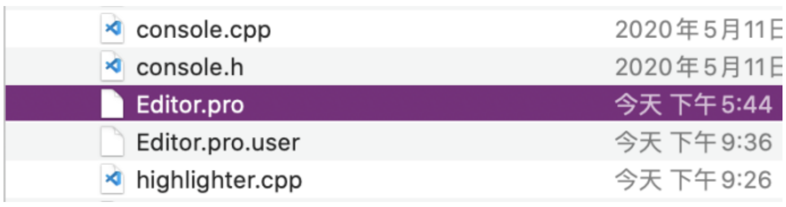
2. Under the Qt Debug folder, Find and move the Original.app (Original.exe for windows) into the Project(Project_Win for windows) folder.



名称	修改日期	大小
build-Ori...it-Debug	今天 下午12:22	--
Editor	今天 下午1:35	--
Files	今天 下午12:22	--
Original	今天 下午12:09	23.8 MB
System	今天 下午12:22	--

Figure 3: Move Original.app

3. Find and click Editor.pro, build and run it with Qt creator.



console.cpp	2020年5月11日
console.h	2020年5月11日
Editor.pro	今天 下午5:44
Editor.pro.user	今天 下午9:36
highlighter.cpp	今天 下午9:26

Figure 4: Editor.pro

- An editor window should pop up already. The user can then enter any program he or she wish to test it, note the following points.

  1. If the user simply copy and paste another program inside, the IDE will consider not editing is in procress, so make sure in this case, type something extra like a blank space to notify the IDE.
  2. The programming language we supoort is almost the same as C++. Minor differences are in ways of input and output, as well as our main program does not have a return value, libary inclusion and namespace. You can only edit under the main function. PS: You may find some prepared sample test files available under the project folder.

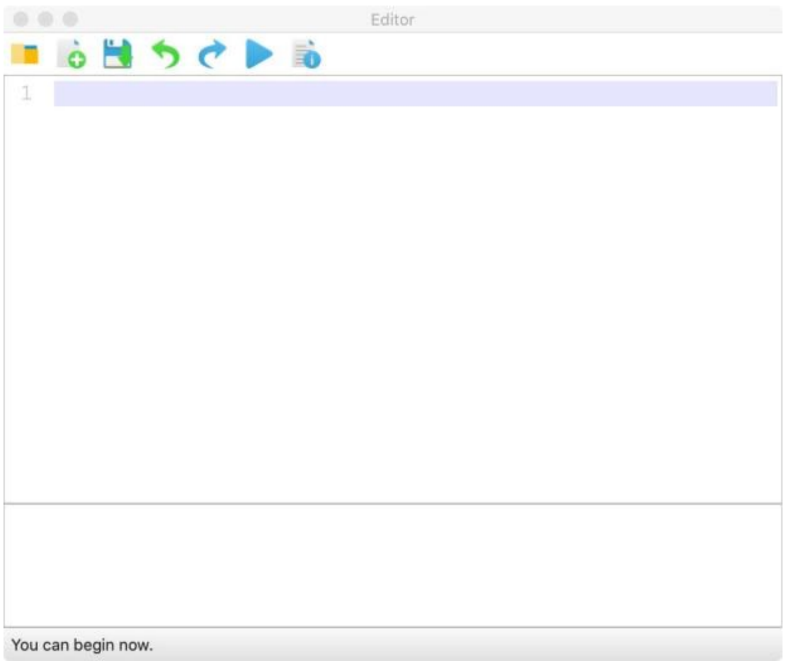


Figure 5: Editor.pro

3. The supported functions include all the essentials of empirical programming. For example, arithmetic at any level of complexties, logic operator(and/or), if, for and while loop, input and output. Four data types are supported: int, char, int array, char array (Like string).

   Notice, the user is recommended to look into appendix C for details about for functions of C++ we support and what differences we have with standard C++.
4. During editing, we implemented common features of C++ program editor that will bring convinience for the program. These features including, different color for diffrent syntax, auto-suggestion list of keywords, auto-completion of brackets, state bar for operation instructions etc. The user can look into our editor section to get more details.
5. When finishing the editing of the program. To execute the program, the user should press the run button directly and then he or she will be prompted to save the file in any directory and the execution follows.
6. Note, input must be entered in English Mode instead of Chinese or others to ensure the success of the execution.

- By now, we are capable of executing the program and the user should get the interaction or results he or she is expecting. However, note it is really hard to demonstrate our work of compiler, assembler and simulator, since they are mostly performing low-level software or even (virtual) hardware tasks, so they do not have any user friendly output. To shed more lights on these components, the user will see the next additional features other than the normal output of his or her program.

- First, the user can check the same folder he or she stored the original program, there will be two new files that are the corresponding MIPS assembly and binary equivalents, translated by compiler and assembler respectively.
- We believe the simulator is the most crucial component since it can leverage all the work done by the previous ones, from a high level language program, we now have a binary program that follows strict rules and syntax. Therefore, when executing the program, whenever an instruction pointed by the program counter is currently being executed, we will print the its location in memory. Whenever the content of any register is changed, we will print out this information. Whenever a branching occured, the destination will be printed. All data loading from the data segment of main memory will also be wise to the user.
- For user-friendly concerns, all helping information mentioned above will be printed out in a diffrent color then the actual output of the program.

2 Editor

2.1 Overview

The editor provides a visual graphical interface for the entire project, where all the interactions with the user in this project will take place. Through the editor, the user will be able to get an intuitive representation of the functions we have completed and achieve all the operations he needs. Overall, the design goal of our editor is user-friendly.

In the final project, our editor has implemented all the functions proposed in the proposal, such as file create, file open, save, save as (file operations) and cut, copy, paste, undo, redo (text editing functions) and so on. In addition, we have also implemented a "Run" button, added the result return area, and designed the application icon.

2.2 Related work

There have already been a number of well-established and fully functional text editors. When desgining functions of our editor, we mainly refer to the logic and methods of Notepad++ and Neatpad. We also took inspiration from the MAC's textedit and use it as a prototype.

As for the UI design, such as background color and text box display, as well as the realization of the connection between different buttons and their actual functions, we were inspired by the HJ editor[1] and refer to the Qt's open source library to see what kind of libraries are needed to build the editor.

The following is a list of the extended libaraires of Qt that we adopted in our implementation of editor. [2]

- $\langle QApplication\rangle$: manages the GUI application's control flow and main settings.
- $\langle QDebug\rangle$: provides an output stream for debugging information.
- $\langle QListWidget\rangle$: this is the super class of all user interface objects.
- $\langle QListWidgetItem\rangle$: this gives a layout item that represents a widget.
- $\langle QMainWindow\rangle$: provides a main application window.
- $\langle QMessageBox\rangle$: provides a modal dialog for informing the user or for asking the user a question and receiving an answer.
- $\langle QObject\rangle$: this is the super class of all Qt objects.
- $\langle QPainter\rangle$: performs low-level painting on widgets and other paint devices.
- $\langle QPlainTextEdit\rangle$: provides a widget that is used to edit and display plain text.
- $\langle QPainter\rangle$: performs low-level painting on widgets and other paint devices.
- $\langle QProcess\rangle$: is used to start external programs and to communicate with them.
- $\langle QRegularExpression\rangle$: provides pattern matching using regular expressions.
- $\langle QSyntaxHighlighter\rangle$: allows you to define syntax highlighting rules, and in addition you can use the class to query a document's current formatting or user data.
- $\langle QTextBlock\rangle$: provides a container for text fragments in a QTextDocument.
- $\langle QTextCharFormat\rangle$: provides formatting information for characters in a QTextDocument.
- $\langle QWidget\rangle$: this is the super class of all user interface objects.

2.3 End results of our implementation

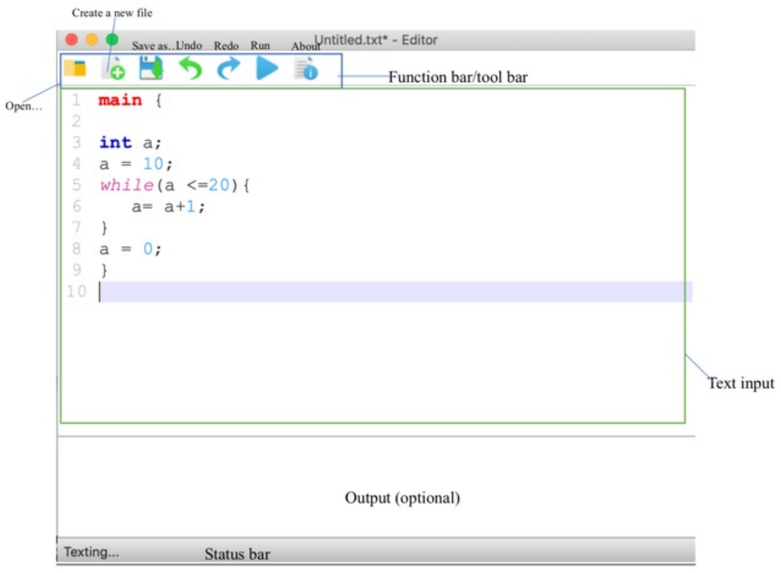
The following figure demonstrates what the user interface of our editor looks like.

2.3.1 Buttons

There are 7 function buttons in the tool bar at the top of our editor window, from left to right they represent file open, create a new file, file save as, undo, redo, run and about respectively. With these buttons, the user can open a file on this interface, create a new file, save the file, edit the file, run the file and get the basic information or user guide by clicking the About button.

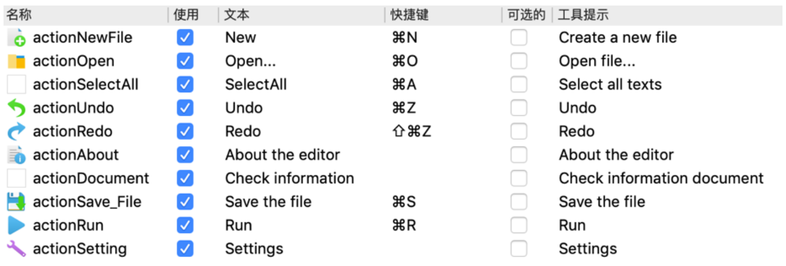
Figure 2 has shown all the button icons used in this editor.

Besides, a munu bar with options that are more or less the same as the buttons in the tool bar is also implemented for convinience, as shown below.



```
main {

int a;
a = 10;
while(a <=20){
    a= a+1;
}
a = 0;
}
```

Figure 6: Editor UI



名称	使用	文本	快捷键	可选的	工具提示
actionNewFile	✓	New	⌘N	☐	Create a new file
actionOpen	✓	Open...	⌘O	☐	Open file...
actionSelectAll	✓	SelectAll	⌘A	☐	Select all texts
actionUndo	✓	Undo	⌘Z	☐	Undo
actionRedo	✓	Redo	⇧⌘Z	☐	Redo
actionAbout	✓	About the editor		☐	About the editor
actionDocument	✓	Check information		☐	Check information document
actionSave_File	✓	Save the file	⌘S	☐	Save the file
actionRun	✓	Run	⌘R	☐	Run
actionSetting	✓	Settings		☐	Settings

Figure 7: Button Icons

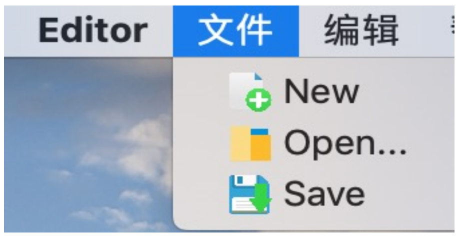


Figure 8: Menu Bar (a)

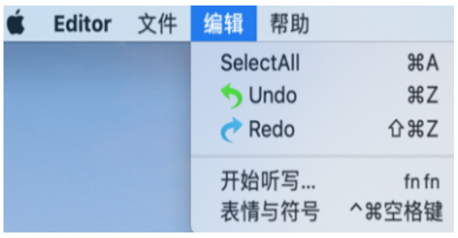


Figure 9: Menu Bar (b)

2.3.2 Text area and output area

Under the tool bar, there is a text input area, where the user can input his code. As shown in Figure 1, the user will see different colors of syntax highlighters for different words according to their usage. Besides, we have made associations for all the key words. Therefore, the user can choose the key word he wants in the alphabetically ordered popping-up completion-list. Moreover, the line index of the activated lines will be displayed on the left, with the current line highlighted.

When the user clicks the Run button, he will be asked to save his file as a .txt file in a specific position and this saved file will read by our next component, compiler, for compilation. Finally, we have an optional output area below the text area, which may receive results from the simulator and print them on the screen.

2.3.3 Status bar

At the bottom of our editor is a status bar. It will guide the user to use our project and provide simple hints for the user in each state. When the user opens our editor, he will receive the message “You can begin now.”. When the user is editing, the status bar will detect this action and display “Texting...”. After the user finish editing and run the code, the status bar will receive the change and output “processing...”. Finally, when all the process is finished and the results are outputted on the screen, the user will notice that the status bar will tell him “Ready”.

2.4 Implementation notes

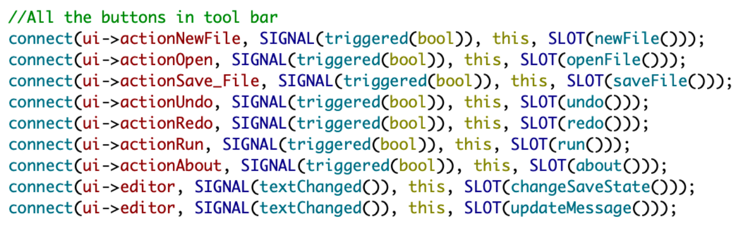
2.4.1 Main window

Step 1: Graphics

The QT designer is used to create a mainwindow. We first built a mainwindow UI, including the buttons and the layout. For the buttons, we downloaded the pngs from the Internet [3], adjusted them to a suitable size, and named the corresponding controls one by one. Then we created a qrc file (image.qrc) to store all the png files. For the layout, we designed the corresponding text box and placed it in the correct position. Finally, we got a file with the suffix “.ui”.

Step 2: Code

The file mainwindow.cpp realized the functions in the main window. Its header file is named as mainwindow.h. In this part, we referred to the Qt modules [4] and used the connect function to connect each setting (button or area) with its corresponding function.



```
//All the buttons in tool bar
connect(ui->actionNewFile, SIGNAL(triggered(bool)), this, SLOT(newFile()));
connect(ui->actionOpen, SIGNAL(triggered(bool)), this, SLOT(openFile()));
connect(ui->actionSave_File, SIGNAL(triggered(bool)), this, SLOT(saveFile()));
connect(ui->actionUndo, SIGNAL(triggered(bool)), this, SLOT(undo()));
connect(ui->actionRedo, SIGNAL(triggered(bool)), this, SLOT(redo()));
connect(ui->actionRun, SIGNAL(triggered(bool)), this, SLOT(run()));
connect(ui->actionAbout, SIGNAL(triggered(bool)), this, SLOT(about()));
connect(ui->editor, SIGNAL(textChanged()), this, SLOT(changeSaveState()));
connect(ui->editor, SIGNAL(textChanged()), this, SLOT(updateMessage()));
```

Figure 10: Button Codes

For the text input area, output area, there are three related functions: setUpHighlighter(), resizeEvent(), updateOutput(), updateError() and inputData(). The first function is used to set the font and highlighters for the text input area. The second function is used to set the size of input and output widgets. The third and fourth functions are used to show the output and error in the output area when the run process finishes. And the last function is used to receive the user input in the output area during the running process.

For the status area, there are two main functions: updateMessage() and runFinished(). The former is used to change the message in status area into "Texting..." when the user begins typing, while the latter is used to change the message into "Ready" when the run process finishes. However, the change of status message will also be triggered in other functions like run() when status changes.

For the file operations, there are three main functions newFile(), openFile() and saveFile() and a subsidiary function initFileData(). The initFileData() function initialize the file name ("Untitled.txt"), file path (Desktop) as well as file saving (saved) and running status (stop). Then the saveFile() function makes it possible for the user to choose the saving name and saving path of the file by himself, but he can also directly use the default setting assigned in the initFileData() function. The openFile() function will open a new file from the given path, but it will first check whether the current file is saved or not. The last function for file operations is newFile(), it will create a new editor window for the new file.

A run() function is written to realize the function of run button. It will ask the user save the file and then link to the compiler and receive the final results from simulator to print them in the output area. As for the redo and undo operations, two functions redo() and undo() defined under class MainWindow will directly link to the undo() function of the text area to realize the text operation.

2.4.2 Highlighter

The file highlighter.cpp file realized the functions in the main window. Its header file is named as highlighter.h. In this part, we set up several QTextCharFormats, including specific keywords, variables, single-line comments, multi-line fixes, and highlights of functions, according to the requirements of the supplier and subsequent parts. We added different types of highlighting statements to the corresponding Qlist, and set different colors for different lists.

For the highlighted format of comments, we used QRegularExpression to apply regular expressions. For the highlighted part of the function, we also set italics to highlight.

2.4.3 Code Editor

The file codeeditor.cpp realized the functions in the main window. Its header file is named as codeeditor.h. In this part, we set the text area color, text content and highlighter color, code line area design, words and symbols completion, based on the principle of user-friendly and referring to the design of Qt Creator.

For the code line area, there are 5 functions defined: widthOfLineNumberArea(), updateWidthOfLineNumberArea(), updataLineNumberArea(), highlightCurrentLine(), paintLineNumberArea(). The first two functions are used to arrange the line number area. One for the width of line according to the character numbers and another is for the space width before the code line area. The function updataLineNumberArea() is used to set line number area with scrolling. And to adjust the text area according to the code line change, a function resizeEvent() is used again. Finally, for the highlighter of the current operation line and the line numbers of activated lines, function highlightCurrentLine() and paintLineNumberArea() are used.

For the words and symbols completion, 5 functions are defined: keyPressEvent(), setCompleteList(), getWordOfCursor(), showCompleteWidget(), getCompleteWidgetX(). The first function is defined for the keyboard operations including the automatic completion of some symbols like "()", "", "¡¿" and "
", usages like while(), if(), for(), and key press like up, down and enter. And the last four functions are for the completion of key words. Function setCompleteList() defines all the key words need to be completed in a QStringList and showCompleteWidget() set the popping out keywords completion list and order the choices alphabetically. getWordOfCursor() and getCompleteWidgetX() are the subsidiary functions get the current word and decide the width, thereby helping set the size of completion list widget.

2.4.4 Auto-suggest list

The file completelistwidget.cpp realized the functions in the main window. Its header file is named as completelistwidget.h. In this part, we set the color, keyboard operations and completion of drop-down complete list widget. For the two functions in this part: keyPressEvent() and ldistance(), the former set the reaction of keyboard operations like up and down and so on, while the latter computes the distance between the source word and the target word use the matrix to give the proper completion to the current input.

2.4.5 console

The file console.cpp realized the functions in the main window. Its header file is named as console.h. This part completes the graphical console window related settings like editor background color, cursor and key actions. We first set the background color and built the connection of cursor position for real time change. Then in function keyPressEvent(), we specified the press action results of the backspace and enter key on the keyboard. Finally, we use the function resetCursorPosition() to reset the cursor's position to the new position.

2.5 Development beyond the proposal

- We have realized syntax highlighting for keywords, annotations and functions.
- We have implemented a relatively simple, colorful and bright UI, and built the connection with compiler and simulator through buttons.

- We have designed the icon of our application.
- In the text input area, we can display the number of activated lines of code. And when a keyword or variable appears, a drop-down menu can be used for associative selection.
- When user is editing code, we implemented tools that can automatically complete some symbols, like closing bracket, etc.

2.6 Reflections and difficulties

Because this editor can only operate on the entire file, it is difficult for us to achieve real-time dynamic feedback of grammatical errors that occur during the editing process. And limited to the conditions, we only stipulate the generation of files with the suffix .txt. Due to the Qt Creator version and special settings, some bugs are difficult to eliminate. The running speed will bring a certain bad experience. And it may take multiple attempts. Different bugs may appear when running on mac and win systems, this is due to the limitations brought by the system itself. Due to the different parts of the connection, the results return area does not display the running results every time.

3 Compiler

3.1 Overview

The objective of compiler is to translate a high-level language (C++/Java) into a low-level language (Assembly language). In this project, our goal is to translate a C++ like language into MIPS assembly language.

According to standard compiler design process, our compiler consists of 3 components (Lexical analysis, Syntax analysis and Intermediate Representation translation).

In lexical analysis process, compiler accepts the source code as a character stream, namely string. Then the compiler will analyze this stream and translate it into a "terminal token" stream.

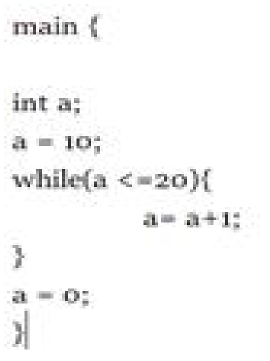
In Syntax analysis process, compiler will use the "terminal token" stream to finish syntax analysis using recursive-descent parsing algorithm. The semantic information will be recorded as Intermediate Representation (IR).

In IR translation, compiler will translate each IR into MIPS assembly language with a specific structure. The compiler allocate registers locally by loading and saving variables in each IR.

3.2 Implementation notes

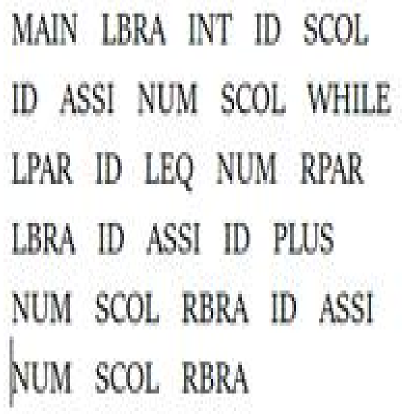
3.2.1 Lexical analysis

The key of lexical analysis is to implement a DFA (Deterministic Finite Automaton). In this project, the DFA is designed manually rather than using some DFA designing tool like Lex. The implementation of DFA is based on if-else judgment rather than a translation matrix. The terminal token stream will be stored in an array named "token_list".



```
main {

int a;
a = 10;
while(a <=20){
          a= a+1;
}
a = 0;
}
```

Figure 11: C++ like program



```
MAIN LBRA INT ID SCOL
ID ASSI NUM SCOL WHILE
LPAR ID LEQ NUM RPAR
LBRA ID ASSI ID PLUS
NUM SCOL RBRA ID ASSI
NUM SCOL RBRA
```

Figure 12: Token pattern

For Terminal definition, please refer to appendix E.

3.2.2 Syntax analysis

In this project, we use recursive-descent parsing algorithm. Thus, a LL(x) (Left-to-right parse, Leftmost-derivation, x-symbol lookahead) grammar should be designed first. (See Appendix F) Every non-terminal symbol is a function and terminals are obtained by lexical analysis. For every sentence derivation noted by "->" , the sematic information will be recorded and IR will be generated.

In this project, we use a linear IR quaternary (op,src1,src2,dst) since it fit MIPS assembly language well. The quaternary definition see appendix G. IR will be stored in a vector in this stage. During syntax analysis, a "symbol table" will be established. Symbol table will indicate the variable address in memory and its type. This table will be used in IR translation stage.

3.3 IR translation part

In this project, we first get the linear IR quaternary from syntax analysis. According to the opcode of IR quaternary, we design specific MIPS assembly language. MIPS instructions mainly include arithmetic and logical instruction, branch instructions, jump instructions, load instructions, store instructions and syscall. The following are the implementation of IR translation.

- **Arithmetic and logical instruction** Opcode of arithmetic and logical instructions include add, sub, mul and div. Related IR include "PLUS", "MINUS", “MULTI” and “DIV”. We first load variable address or value in register $t0and$t1, for example. Then, we do corresponding operation and store the result in \$t2. Finally, we save the result in the destination of IR.

- **Branch instructions** Opcode of branch instructions include beq, bne, bgt and bge. Related IR include “BEQ”, “BNE”, “B¿” and “B¿=”. We first load the variable address or value in register $t0and$t1, for example. Then, we enter branching instruction.

- **Jump instructions** Opcode of jump instructions include b. Related IR include “J”. In this part, we just put the destination label after “b”.

- **Load instructions** Opcode of load instructions include li, la, lw, lb. For integer, we use “li”. Related IR include “LOAD”, “ASSIGN”, “PUTCHAR” and “INPUT”. For variable address, we use “la”. For variable name, we use “lw”. For string or character, we use “lb”.

- **Store instructions** Opcode of store instructions include sw, sb. Related IR include “OUTPUT”. For integer, we use “sw”. For character, we use “sb”.

- **Syscall** Related “INPUT” and “OUTPUT”. We first load the argument in \$a0 if necessary. Then we load the code in \$v0 for syscall to use.

  Appendix H shows the details of IR translation.

3.4 Reflection and difficulties

Since we were sophomores with no knowledge about compiler principles, we spend a lot of time in reading the books and taking online courses. Finally, we understand the process of compiler and how to construct a simple compiler. One big problem during our compiler construction is how to analysis complex logic expression. Since in text book, this part is finished by LR analysis and we use LL analysis. We stop here for several days. We finally find the logic expression is something like arithmetical expression. We can consider a complex logic expression as the combination of “and” logic connected by “or”. And inside the “and” logic, it several “not” logic and “or” logic in parentheses connected by “and”. Appling this principle, we can analysis complex logic easily.

4 Assembler

4.1 Overview

Our final version of assembler presents itself as part of an independent Qt project, it will be incorporated into Qt mega project to be part of the complete system binary tools. Our assembler takes a file as input and output a file accordingly.

The achivements of us can be desribed in three parts. First, we accomplished the basic and minimum requirements of a MIPS language assembler, i.e. we are able to translate a MIPS instruction to a binary file in appropriate format, based on the principle idea of one-to-one correspondence. Second, we are able to support a variety of native and pseudo instructions, including most common instructions of R, I, and J format. Third, we go beyond a mere mapping between MIPS and bits by also adopting a single algorithms to assign all labels in the program a virtual memory address, which is helpful to our simulator.

We prepared four assembly language file to test the capabilities of our assembler. We will provide a detailed list of instructions that we support in Appendix, but in general, we are capable of dealing with programs that demands input from clients, doing convoluted arithemetics, outputing results to confole, diverging according to conditions, loops, etc.

4.2 Connection with compiler

The input of the assembler is precisely the output of compiler, an equivalent MIPS language program. Except for the obvious fact that the logic and correctness of the instructions are crucial to assembler, the layout and data summarized in the data segment also play a key role in determining the success of assembler.

The data segment consisting of all the variables declared in the main program, as well as some intermediate variables generated by compiler. The assembler demands two things from compiler: First, each instruction or variable (Label in data) occupies an independent line; Second, that the program follows MIPS syntax and the logic is correct.

4.3 Supported functions

We implemented an assembler that can translate a MIPS instruction file into the corresponding binary files. We support many native and psedo MIPS assembly instructions,, which includes basic arithemetic calculations of both signed and unsigned integers, load and store operations that move data between registers and main memory, as well as branching instructions. We also implemented all major Syscall functions based on the SPIM simulator, which includes simple print functions, read integers or strings from concole, and exit function.

Aside from the basics, our assembler also support simple string functions, most notably string indexing to get a character. This is achieved by supporting lb, rscr1, imm(rsrc2) MIPS instruction, which loads a byte at the right address to the right register, and the string is of course stored in the virtual memory in the form of a sequence of characters.

Supposedly, our input file follows strictly the syntax of the assembly language for MIPS CPU. consists of text and data directives, text directives contain instructions, data directive contains data and address of the data. The supported instructions in the text directives is listed above. Our data segment can contain simple ASCII zero terminated strings and integers.

4.4 Implementation notes

Our program scans the input file twice. In the first scan, all the labels for instructions or data are found out, and they are assigned an address in virtual memory based on three principles. First, labels in the text directives get addresses starting from the tail of the virtual memory, and then go back ward, with each instructions occupies eight bytes. Second, labels in the data directives was given addresses starting from the middle place of the virtual memory, and then go downward; character for example, occupies a byte, while integers occupites a word.

In the second scan, each statement in MIPS assembly language is broken down to its opcodes, register specifiers, and labels, and then their numeric equivalents, which are searched in a map data type, are combined to produce a legal binary instruction.

4.4.1 Important data structures

- *Symbol table:* This is a map data type in standard C++ template library. It records all labels found in the input asm file, which may contain both global and local references. The keys of the map are symbol names, the values are keys' addresses in the main memory. To obtain where in main memory (Here the main memory refers to the one allocated in the simulator, not actually the real memory in a computer) each label should go, our assembler has to keep track of the instruction size, which is an easy task given MIPS has fixed-length instructions.

- *Token scanner:* This is a scanner data type in stanford C++ library. It is used in our implementation of assembler mainly to examine line by line. More specifically, we use this to break down one assmembly language instruction to its componnets, like opCode, register names, immediate values etc.

- *Rname:* This is a map data type in standard C++ template library. It contains all the names for 32 general purpose registers as keys, values being their corresponding 5 bits numeric representation.

4.4.2 Important procudures

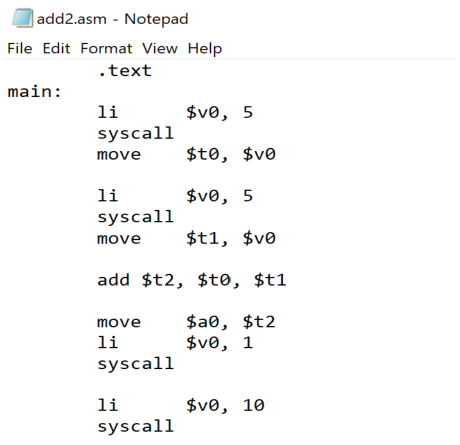
- *Decimal to binary:* This is a handy function that aids the process of getting the one-to-one correspondence between assembly instruction and binary representations. It takes two parameters, one if a decimal number, one is the output length of the binary number. This function is useful because the codes for MIPS instructions are often available in base 10 form, while we need to get the bianry form in appropriate length.

- *Operation kind:* This header file exports functions that come in the final process of scanning a complete assembly language file. It can recognize whether a line is a legitimate MIPS instructions or not, if it is, it will output the information that indicates which operation it performs exactly. After this set of procedures, we can call the functions mentioned below to perform individual translation.

- *R format:* This header file export a function that can translate a R Format MIPS instruction.
- *I format:* This header file export a function that can translate a I Format MIPS instruction.
- *J format:* This header file export a function that can translate a J Format MIPS instruction. Note that often the SymbolTable data structure need to be passed by reference as a parameter, because we will need the addresses of the labels in the translation. Note also, some supported J format instructions are pseduo-instructions, so we also implemented procedures that break down them into several native instructions as well.
- *Load and move format:* This header file deals with the load and move instructions occured in the input file, similar to J format, we need SymbolTable and specific attention is given to pseduo-instructions.

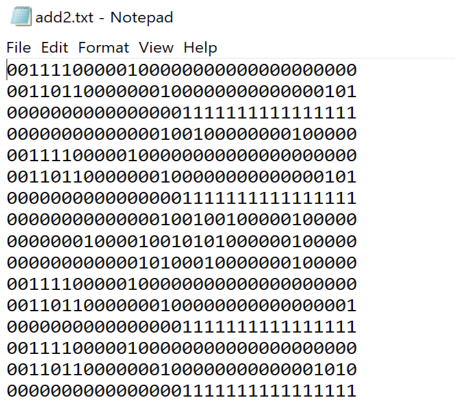
4.5 Test Results

We include one representative test among many we did on our assembler here for demonstration. The figure on the left is the input file we scanned, the figure on the right is the file we produced.



```
        .text
main:
        li      $v0, 5
        syscall
        move    $t0, $v0

        li      $v0, 5
        syscall
        move    $t1, $v0

        add $t2, $t0, $t1

        move    $a0, $t2
        li      $v0, 1
        syscall

        li      $v0, 10
        syscall
```

Figure 13: add2.asm file



```
00111100000100000000000000000000
00110110000000100000000000000101
00000000000000001111111111111111
00000000000000100100000000100000
00111100000100000000000000000000
00110110000000100000000000000101
00000000000000001111111111111111
00000000000000100100100000100000
00000001000010010101000000100000
00000000000010100010000000100000
00111100000100000000000000000000
00110110000000100000000000000001
00000000000000001111111111111111
00111100000100000000000000000000
00110110000000100000000000001010
00000000000000001111111111111111
```

Figure 14: add2.txt file

4.6 Reflection and difficulties

The assembler is implemented almost completely from sractch, guided only by the principle in one computer architecture book [5]. We are significantly benefited from the idea of top-down design, we divide the whole translation task into several processes, like generate symbol table, distinguish different kind of operations, and then translation for different kind of operations. There are no particular processes that would demand a lot of time, each of them are relatively simple enough to implement.

Compared to the proposal, one thing we ended up not doing is Linker, in other words, our assembler does not support global labels that may refer to a different file. This is mainly due to the surprising complexities coming along with our ambition to implement a complete set of system binary tools, from editor all the way to simulator. When we realized we could have used time more wisely on other more significant components like editor and compiler, the original plan for linker is then canceled. Nevertheless, our assembler can support a variety of programs, as will be demonstrated.

5 simulator

5.1 Overview

Load and simulator are the final components we need to achieve a complete set of system binary tools, and a crucial one as well, since they are the ones that can leverage the work done by previous components. More specifically, initially we have a relatively flexible language by which progrommer programs, after compiler and assembler, we get a rather rigourous binary file that follow strict rules and certain text at certain positions have fixed meanings, our simulator is supposed to take advantage of that and demonstrate our work in the form of simulating executing a program just like what a real computer would have done.

Eventually, our simulator supports just as many types instructions as our assembler, in other words, every line translated by assembler can be recognized by our simulator. Moreover, the execution process exactly mimics the real hardware (This is the whole point of a simulator), where we have registers, main memories, program counter that keeps track of which instrution to execute next, and other things.

5.2 Connection with assembler

In the end, simulator (Which includes a loader of course, for simplicity, we only mention simulator from now on) and assembler are in the form of one independent Qt project due the unsurprising close relationship between the two. The whole project takes a assembly language program and simulate the executing of it, and the equivalent binary program is a key intermediate product. The test results show that, as long as the algorithm logic is correct and instructions fall in our supporting set, the execution should be alright.

5.2.1 AddressLabel.h

This is a important header file that serves as an interface between assembler and simulator. It exports functions that scan the assembly language file, find all labels, and also recording how many instructions are there between each labels, and eventually produces the appropriate address each label should have in the virtual memory.

This is important to assembler, since to translate branching or loading instructions involving labels, their addresses have to be resolved. This is also important to simulator, since when executing the branching or loading instructions involving labels, we need to make sure the program counter jumps to the correct position or the right address is loaded, so that the whole execution would be coherent and produce expected results.

5.3 Implementation notes

5.3.1 Important data structures

- *Word:* This object represents the structure of a word, which consist of 32 bits or 4 bytes, this will be the building blocks for the following classes. Each byte of the word is addressable by means of member functions, this is similar as in the real computer, byte is the least addressable unit.

- *VirtualMemory:* This is a class the represents the main memory in a computer. One key instance variable of this class is a dynamic array of words, which is to imitate the real main memory. Other instance variable includes a program counter, which is simply an integer (means address) in this case; datasize, which conveys how many words the data segment occupies; size, which conveys how many words the text segment occupies.

- *registersMem:* This class represents the register files in a computer. In implementation, the key instance variable is simply a dynamic array of integers, the array index tells the indentity of the register, the array storage tells the content of registers.

5.3.2 Important procedures

- *binary to decimal:* Since the instructions read by simulator are all binary strings, it is sometimes convinient to have a handy function that tells the their decimal equivalents. For example, if we need to extract the information about registers from the binary instructions, we need to get the right substring, then use this handy function to know which register it means.

- *Decimal to Hexdecimal:* This function is used to output helping information in the simulating process, we know the address of the main memory is mostly in the form of hexdecimal numbers due to simplicity, while in our implementation, the addresses are decimal numbers, so the this function become necessary for our purpose.

- *Simulate Operation Kind:* This purpose of this procudure is similar to the "Operation Kind" in assembler, but instead of reading a line of MIPS code, this reads a line of binary code, and then output what kind of operation the instruction alludes to. We use this function to classify all the instructions, so that we can call the right functions to each instructions.

- *Execute Syscalls:* This function simulates the executing of various system call instructions. When it is called, it first go check the number of register $v0, which stores paramter that tells what kind of system call this is. Then, it perform necessary tasks per different Syscalls. For example, it is a read integer syscall, it reads an integer from console and store the result to $v0; If it is a read string syscall, it needs to check the content of $a0, which gives the address to store the string read from console.
- *Execute R format:* This function executes R type instructions.
- *Execute J format:* This function executes J type instructions.
- *Execute L format:* This function executes load instructions.
- *Update register:* This is a member function of the "registerMem" class, it can update the register to a new value. It needs the index of register and new value as parameters. In addition, it will print the helping information of which register is updated to what value.
- *Fill Data:* The corresponding header file exports functions that are helpful in the process of filling data in the right address of the data segment in the main memory.

5.4 Reflection and difficulties

We adopted the methodology of object oriented programming a lot in our implementation. One of the major task is to set up the "hardware". After we have the memory in place, the rest of the simulation is relatively straightforward. All we have to do is to make sure two things, first we can recoginize and execute the instructions correctly; Second, we make a good track of addresses, so that when loop and branching are involved, we can keep the logic of the algorithm right.

We had some difficulties when trying to improve our simulator so as to be able to execute programs involving string manipulations, because for simplicity, originally the way string is stored in our data segment is not quite like how it is stored in a real main memory. So our way of addressing one character in a string is quite unnatrual and cubersome. Eventually, we made the decision to abandon the old way, and store the string data as a sequence of characters instead, this brings a lot of work but we make it work finnally.

Contribution:

The various components in this project turned out to be highly inter-connected and we as a group spent much time communicating and collabrating. Each component achived to a desired level and robustness and every one of us worked diligently to make them as good as possible. We give ourselves the following evaluations in terms of contribution individually.

Yao Tong 16.7% Luyao Gao 16.6%

Supei Yang 16.7%	Lan Qin 16.7%
Yuqian Mao 16.6%	Zhiyuan Wan 16.7%

6 Appendix

Appendix A List of suppported MIPS instructions

- add rd, rs, rt

  Put the sum of registers rs and rt into register rd.

- sub rd, rs, rt

  Put the difference of registers rs and rt into register rd.

- mul rd, rs, rt

  Put the low-order 32 bits of the product of rs and rt into register rd.

- divu src1, src2

  Divide src1 by src2, put the quotient in register lo, the remainder in register hi.

- div des, src1, src2 *pseudoinstruction*

  Divide src1 by src2, put the quotient in register des.

- addi rd, rs, imm

  Put the sum of registers rs and the immediate into register rd.

- subi rd, rs, imm

  Subtract the unsigned number imm from rs, and put the result into register rd.

- b label *pseudoinstruction*

  Unconditionally branch to the instruction at the label. Pseudoinstruction category indicates that assembler will be responsible to translate this instruction to several native MIPS instructions.

- beq rs, rt, label

  Conditionally branch to the instruction at the label, if contents of rs == rt.

- blez rs, label

  Conditionally branch to the instruction at the label, if register rs is less than or equal to 0.

- bgt rscr1, src2, label *pseudoinstruction*

  Conditionally branch to the instruction at the label if register rscr1 is greater than src2.

- bne src1, src2, label

  Conditionally branch to the instruction at the label, if register src1 is not equal to register src2.

- bge src1, src2, label *pseudoinstruction*

  Conditionally branch to the instruction at the label, if register src1 is greater than or equal to register src2.

- beqz src1, label *pseudoinstruction*

  Conditionally branch to the instruction at the label, if value of register src1 is zero.

- la rdest, address *pseudoinstruction*

  Load computed address, not the contents of the address, into register rdest.

- li rdest, imm *pseudoinstruction*

  Loads an immediate value into a register.

- lb rdest, (rs)

  Load the byte stored at address rs into rdest.

- lw rdest, imm(label)

  Load the word stored at address label+imm into rdest.

- sw rdest, imm(label)

  Store the contetn of rdest into the word at address label+imm

- move rdest, rscr *pseudoinstruction* Move the contents in register rscr to rdes.

Appendix B Directives and system calls

Instruction *syscall* provides several useful system serveices, register \$v0 contains the code to determine what kind of syscall is used. \$a0 contains the argument to provide for the syscall.

- \$v0 = 10, exit
- \$v0 = 1, print integer contained in \$a0
- \$v0 = 4, print string pointed by the address in \$a0
- \$v0 = 5, read an integer input from console, stored back in \$v0; Or, if the input is a char, then its ascii value is stored in \$v0
- \$v0 = 8, read a string input from console, store at address \$a0, \$a1 gives the length of the allocated string space.

Directives

- *.text* Subsequent items are put in the text segment. These items may only be instructions.
- *.data* Subsequent items are stored in the data segment.
- *.asciiz* Store strings in memory and null-terminate it. Usage:

  Name: .asciiz str

  the label name contains the address of where the string is stored.
- *.space* Store strings in memory and null-terminate it. Usage:

  Name: .space number

  Starting at label ”name”, allocates ”number” of empty bytes for later usage.

Appendix C List of supported “C++” language instructions

This project supports a c-like language. (note that this language is case-sensitive)

- The main function must be declared as follows: main {. . . }

  In other words, there is no return value.
- Variable declaration is the same as C++. We support **int**, **char**, as well as their arrays.

  Note, however, the variable cannot be declared with a default value. Its value must be assigned separately.

  e.g. int a; int b[10]; char c; char d[10];
- We support arithemetical expressions at any level of complexities.

  e.g. a = (2*(10+b)+a*7*b)*2+1;
- The continuous assignment for char array is valid. For example, char c[30];c“hello world!”;
- We support For and While Loop that follows C++ syntax.

  for (initialize; condition; step) {. . . }

  Note, “{” and “}” can be omitted; Also, the iteration variable should be declared in advance.

  e.g. for (i = 0; i <= 10; i = i + 1) {

  b = b + 1; c = c + 1;

  }

  Loop can be convoluted, for instance:

  while (a >= 10) {

while (b < 100) c = c + 1;

}

- We support If, Else sentence, with the same syntax as C++:

  if (condition) {. . .} else {. . .} (else is optional). For example,

  if ((a==10 and b==5 or c==1) or d==6) { a = 1;

  if (a == 1) a = 3; } else {

  a = 2;

  }

- Input/output sentence. In this aspect, we use "input()" and "output()" to replace "cin" and "cout" in C++; For example,

  int a;

  intput(a); # Means cin >> a;

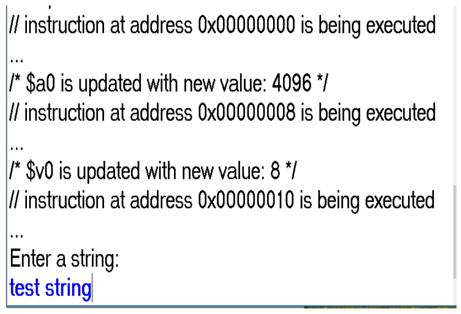
  output(a); # Means cout << a;

  output("Hello world!"); # Means cout << "Hello world!";

- We support C++ style commenting. In other words, both // and /* */ are accepted.

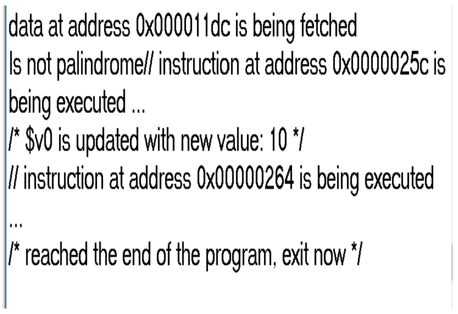
Appendix D Sample output

We use one program as an example here, which takes a string input from user and then returns whether it is a palindrome or not. Figure 15 shows the the beginning of the simulation, note the first line is always the same, since it is by design that the first instruction is stored in address zero. Figure 16 shows the end of the simulation, note the last two helping information is the same for all programs, since they are the execution of exit syscall.



```
// instruction at address 0x00000000 is being executed
...
/* $a0 is updated with new value: 4096 */
// instruction at address 0x00000008 is being executed
...
/* $v0 is updated with new value: 8 */
// instruction at address 0x00000010 is being executed
...
Enter a string:
test string
```

Figure 15: The beginning of execution



```
data at address 0x000011dc is being fetched
Is not palindrome// instruction at address 0x0000025c is
being executed ...
/* $v0 is updated with new value: 10 */
// instruction at address 0x00000264 is being executed
...
/* reached the end of the program, exit now */
```

Figure 16: The end of execution

Appendix E Token definition

Terminal name	Terminal Pattern		
FOR	for	ASSI	=
IF	if	GEQ	>=
ELSE	else	GRE	>
WHILE	while	LEQ	<=
ID	[a-z—_][a-z0-9]*	LES	<
NUM	[0-9]+	COMMA	,
INT	int	SCOL	;
LPAR	(	PLUS	+
RPAR	)	MINUS	-
LBRACK	[	TIMES	*
RBRACK	]	DIV	/
EQ	==		

Appendix F LL(x) context free grammar

Note: ⟨ ⟩ means derivation sentence, “” means terminal symbol, {}means the patten may repeat 0 or n times.

Entire program:

⟨whole⟩ -⟩ “main” “{“ ⟨block sentence⟩ “}”

Block sentence:

⟨block sentence⟩ -⟩ “ID” ⟨assign⟩ ⟨block sentence⟩

⟨block sentence⟩ -⟩ “IF” ⟨if sentence⟩ ⟨block sentence⟩

⟨block sentence⟩ -⟩ “char”—”int” ⟨variable declaration⟩ ⟨block sentence⟩

⟨block sentence⟩ -⟩ “while” ⟨while sentence⟩ ⟨block sentence⟩

⟨block sentence⟩ -⟩ “for” ⟨for sentence⟩ ⟨block sentence⟩

⟨block sentence⟩ -⟩ “input” ⟨input sentence⟩ ⟨block sentence⟩

⟨block sentence⟩ -⟩ “output” ⟨output sentence⟩ ⟨block sentence⟩

Variable declaration part:

⟨variable statement⟩ -⟩ ⟨variable declaration⟩ “;”

⟨variable declaration⟩ -⟩ “int”—”char” “ID”— (“ID” “[“ “NUM” “]”) ⟨multiple variable declaration⟩

⟨multiple variable declaration⟩ -⟩ “,” “ID” ⟨multiple variable declaration⟩

⟨multiple variable declaration⟩ -⟩ “;”

Arithmetic expression part:

⟨expression⟩ -⟩ “+” — “-“ — ϵ ⟨term⟩ {“+”—”-“ ⟨term⟩}

⟨term⟩ -⟩ ⟨factor⟩ “*”—”/” ⟨term⟩

⟨term⟩ -⟩ ⟨factor⟩

⟨factor⟩ -⟩ “ID” — “NUM” — “(“ ⟨expression⟩ “)”

Variable assign part:
⟨assign⟩ -⟩ "=" ⟨expression⟩
Boolean logic part:
⟨or logic⟩ -⟩ ⟨and logic⟩ "or" ⟨or logic⟩
⟨and logic⟩ -⟩ "not" ⟨and logic⟩
⟨and logic⟩ -⟩ "(" ⟨or logic⟩ ")"
⟨and logic⟩ -⟩ ⟨expression⟩ "=="—"⟩="—"⟩"—"⟨="—"⟨"—"!=" ⟨expression⟩ { "and" ⟨and logic⟩}
For-loop part:
⟨for sentence⟩ -⟩ "(" ⟨assign⟩ ";" ⟨or logic⟩ ";" ⟨assign⟩ ")" ⟨block sentence⟩
While-loop part:
⟨while sentence⟩ -⟩ "(" ⟨or logic⟩ ")" "{" ⟨block sentence⟩ "}"
If part:
⟨if sentence⟩ -⟩ "(" ⟨or logic⟩ ")" "{" ⟨block sentence⟩ ϵ — ("else" "{" ⟨block sentence⟩ "}")
Input part:
⟨input sentence⟩ -⟩ "(" "ID" {"," "ID"} ")"
Output part:
⟨output sentence⟩ -⟩ "(" "ID"—"string" ")"

Appendix G The IR definition

Op	Src1	Src2	Dst	Meaning
PLUS	T1	T2	S	S = T1 + T2
MINUS	T1	T2	S	S = T1 - T2
MULTI	T1	T2	S	S = T1 * T2
INT			a	Int a
CHAR			a	Char a
ASSIGN	v	offset	a	a[offset]=v
L			Label	Set a label for next Mid-codes
BEQ	T1	T2	Label	Branch to label if T1 == T2
B>	T1	T2	Label	Branch to label if T1 >T2
B>=	T1	T2	Label	Branch to label if T1 >= T2
BNE	T1	T2	Label	Branch to label if T1 != T2
INPUT			a	cin >>a
OUTPUT	a	STR/INT/CHAR		cout <<a
PUTCHAR	c	offset	a	a[offset] = 'c'

Appendix H IR to MIPS

- PLUS, T1, T2, S   li/lw, $t1, T1;   li/lw, $t2, T2;

add, $t3, $t2, $t1;   sw, $t3, S

- MINUS, T1, T2, S   li/lw, $t1, T1;   li/lw, $t2, T2;

  sub, $t3, $t2, $t1;   sw, $t3, S

- MULTI, T1, T2, S   li/lw, $t1, T1;   li/lw, $t2, T2;

  mul, $t3, $t2, $t1;   sw, $t3, S

- DIV, T1, T2, S   li/lw, $t1, T1;   li/lw, $t2, T2;

  div, $t3, $t2, $t1;   sw, $t3, S

- LOAD, T1, T2, S   INT: lw, $t0, T1;   li/lw, $t2, T2;   sw, $t2, T2

  CHAR: la, $t1, T2;   lb, $t0, (T2)$t1;

- BEQ, T1, T2, L   li/lw, $t1, T1;   lw, $t2, T2;

  beq, $1, $t2, L

- BNE, T1, T2, L   li/lw, $t1, T1;   lw, $t2, T2;

  bne, $1, $t2, L

- B>, T1, T2, L   li/lw, $t1, T1;   lw, $t2, T2;

  bge, $1, $t2, L

- ASSIGN, v, offset, a

  INT: li/lw, $t1, v;   sw, $t1, (offset)a

  CHAR: la, $t2, a;   sb, $t0, $t2

- PUTCHAR, v, offset, a   li, $t0, v;   la, $t1, a;   sb, $t0, (offset)$t1

- INPUT, a

  INT: li, $v0, 5;

  CHAR: la, $a0, a;   li, $v0, 8

- OUTPUT, a, STR/INT/CHAR

  INT: la, $a0, a;   li, $v0, 8;

  CHAR: la, $a0, a;   li, $v0, 4

References

[1] https://github.com/m-iDev-0792/HJ-Editor

[2] https://doc.qt.io/archives/qt-5.7/classes.html

[3] https://github.com/alecive/FlatWoken

[4] https://doc.qt.io/qtforpython/PySide2/QtCore/QProcess.html

[5] https://ict.iitk.ac.in/wp-content/uploads/CS422-Computer-Architecture-ComputerOrgani
pdf