**FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)**

****

**COMP 451 (Compiler Construction)**

**2022 FALL**

**Assignment - 2**

**Gulraiz Noor Bari (231-525536)**

**Muhammad Hamza (231-450349)**

**INTRODUCTION:**

* **stdio.h** (standard input/output)is a header file that contains declarations for functions like, printf, scanf, etc.
* **stdlib.h** (standard library) is a header file that contains declarations of functions that involves memory allocation and process control. For example, in our program we used **exit(0)** at the end; which means to successfully terminate the program.
* **string.h** is a header file that contains declarations of functions that are used for working with strings (string manipulation).
* **main():** The main function serves as the starting point of the program execution in C language. User can pass any number of parameters depending upon the requirements of the program logic or structure. We have passed two parameters:
* **int argc:** Keeps count of the number of command-line arguments entered by the user that also includes the name of the program.
* **char \* argv[]:** Here argv (stores command-line arguments) is an array of pointers that is pointing to the characters of the command-line arguments entered by the user.
* **MAX\_LINE\_LENGTH** and **NUM\_OF\_REGS** (number of registers) are macros (variables that can be accessed from anywhere within a program) that hold values of 100 and 26
* **Struct** is a storage structure in C that can hold values of different datatypes. In our case, we use **Struct register** that stores the register name and its corresponding machine code.
* We initialize a list of registers using the previously made **register Struct** and store the register names and there corresponding machine code, which are in decimal digits and will converted into binary, later on.

**LOGIC/ALGORITHM:**

The code is a program that reads a file that contains an Assembly Language program and displays its corresponding machine code (not the actual output of the assembly language code) on the console.

* The program starts by checking whether two arguments were passed by the user in command line, which is done by argc (which has the count of arguments entered in the command-line). If the number of arguments is not two the function is not proceeded and displays a message “Invalid argument”, otherwise if user has entered two arguments; in our case the executable and file name that is to be read, the program proceeds.
* The file is read line by line in a loop using **fgets** function, which takes in three arguments:
  + **char \*str:** Pointer to an array of character where the string read will be copied.
  + **int n:** Maximum number of characters that can be copied.
  + **FILE \*stream:** Pointer to a file that will be read.

**Fgets (line, MAX\_LINE\_LENGTH, input\_file)**

**fgets** function stores first 100 (value of MAX\_LINE\_LENGTH) characters of the input file in **line** array, for each line of every iteration and it returns a string. The loop will run until null pointer is returned.

* Then we split the current line being read (in **line array)** in tokens using **strtok()** using commas (“ ,”) as delimiter. The first token of our Assembly Language program represents what operation will be performed.

**add $t1, $t2, $t3**

So, **“add”** will be stored as an operand and the remaining three (depending on type of instruction r-type or i-type) will be stored as register/immediate values.

* We will compare the current operand with being read and stored with all other operands available and this will be done using if-else statements. For example, we are comparing the current operand with, **“and”** the statement results as True. **“and”** is a r-type instruction so now we have to find the values of **rd** (destination register), **rs** (source register), **rt** (target register) in order to generate the machine code, and for this we will use the register table that we created using **Struct,** above.
* We run a loop on the register table and check if the current register being read is present in the table, if True then assign the machine code of the register to **rd.** Repeat this same process to get the values of **rs** and **rt.**
* Now in the next step we initialize the machine code variable that (in case of **“and”** r-type):
* First, takes the opcode of the operand and shifts it left to 26.

**000000**00000000000000000000000000

* + Then it takes **rs** (source register value, which is 10) and shifts it left to 21.

000000**01010**000000000000000000000

* + Then shifts **rt** (target register value, which is 11) to left 16 places.

00000001010**01011**0000000000000000

* + Then shifts **rd** (destination register value, which is 9) and shifts it left to 11 places.

0000000101001011**01001**00000000000

* In case of r-type instruction, next 5 bits are for shift amount, which is zero.

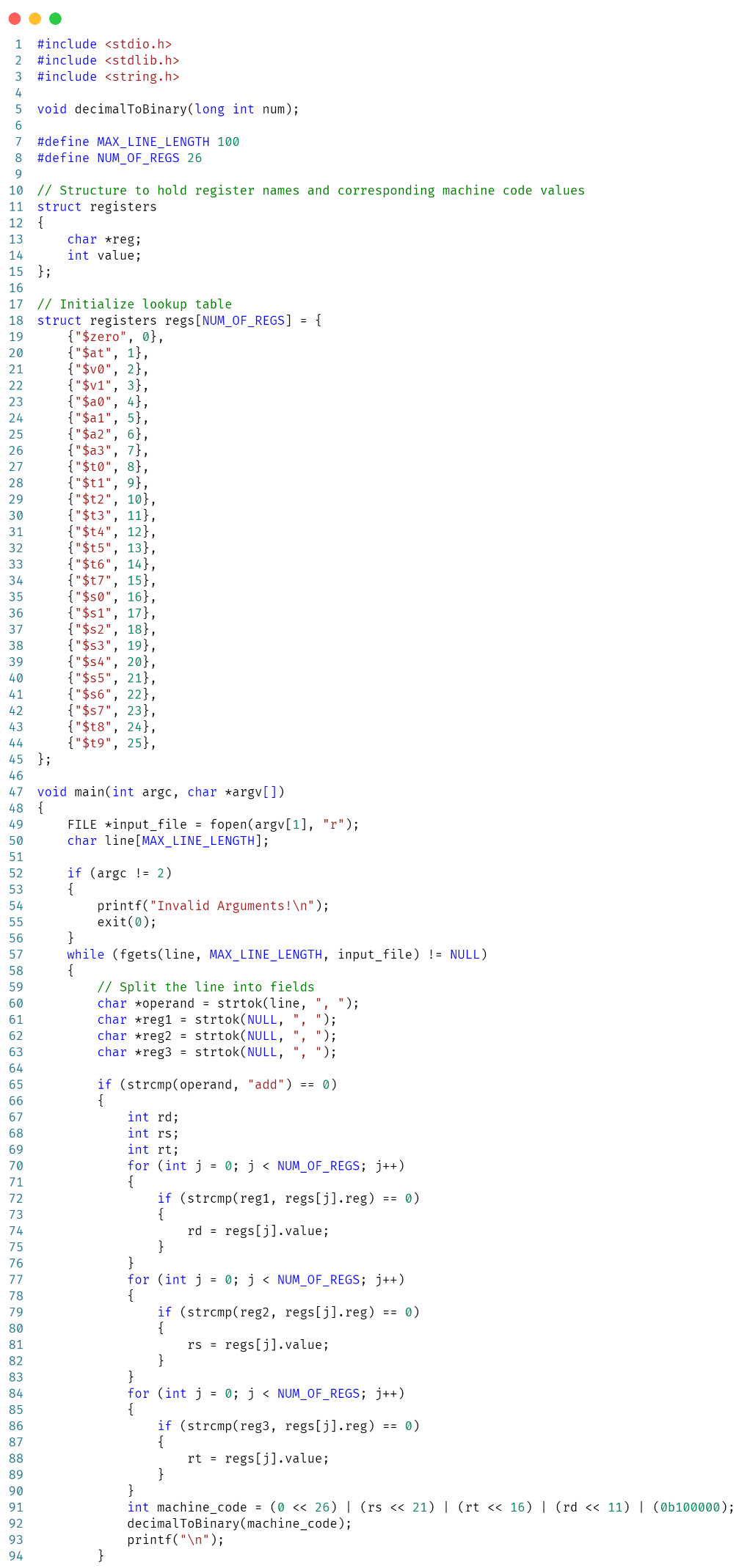
000000010100101101001**00000**000000

* The last six bits represent function code and is different for every operand (but specific to r-type instruction)

00000001010010110100100000**000000**

Finally we will convert the generated decimal number into binary using the function **decimalToBinary()**; that takes a long int decimal number and generates a equivalent binary number.

We will repeat the above steps for every Assembly language instruction line in the text file and generate the equivalent machine code and display it on the console.



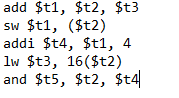
Table

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

**INPUT:**



**OUTPUT:**

Text

Description automatically generated