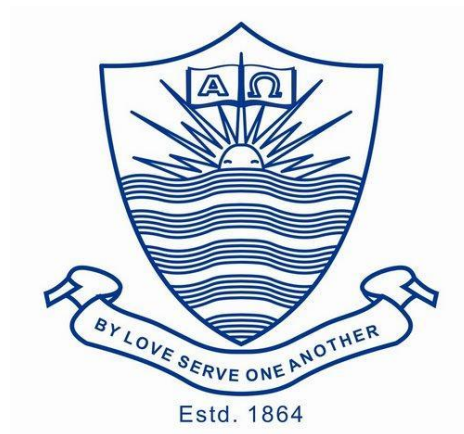


# **FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)**



**COMP451A**

**Fall 2022**

**Assignment 2**

**Abdullah Baig - 231485698**

## INTRODUCTION:

- This code is a program that converts MIPS assembly language instructions to their corresponding binary code. Assembly language is a low-level programming language that is used to write programs for computers and other devices. It consists of a set of mnemonic codes that represent machine instructions, as well as directives and data definitions. Assembly language is often used to write programs for microprocessors and other devices that have a simple instruction set, because it is easier to write and read than machine code, which consists of binary numbers that represent the machine instructions.
- An assembler is a program that converts assembly language code into machine code, which can be executed by a computer or other device. An assembler reads the assembly language code and generates the corresponding machine code, which is usually stored in a file or written to memory. The machine code generated by an assembler can be loaded into the memory of a computer or device and executed by the processor.
- The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a Reduced Instruction Set Computing (RISC) architecture that was developed by MIPS Technologies, Inc. in the 1980s. MIPS is a popular choice for teaching computer architecture and programming at the undergraduate level, as it has a simple and clean instruction set that is easy to understand and implement. The MIPS instruction set consists of about 200 instructions, which can be grouped into several categories, such as arithmetic, logical, load/store, branch, and other instructions.
- This program reads instructions from a file called "testFile.txt" and converts them one by one. The input file should contain a list of MIPS assembly language instructions, one per line. The program supports the following instructions: add, and, or, andi, ori, addi, sw, lw, and slt. The program converts each instruction to its corresponding machine code and stores the result in a file or writes it to memory.

- In the following section, we will describe the logic of the program in more detail, including the functions that are used to convert the MIPS instructions to their binary representation.

## **LOGIC:**

- The program begins by calling the main function, which is the entry point of the program. The main function first opens the input file "testFile.txt" using the fopen function from the stdio.h library. The "r" mode specifies that the file should be opened in read-only mode.
- Next, the main function checks if the file was successfully opened by checking if the file pointer returned by fopen is NULL. If the file could not be opened, the program prints an error message and returns 1 to indicate an error. Otherwise, the program proceeds to read the instructions from the file.
- The main function reads the instructions from the file one line at a time using the fgets function from the stdio.h library. fgets reads a line of text from the file and stores it in the line character array. The sizeof operator is used to determine the size of the line array, so that fgets knows how many characters to read.
- For each line read from the file, the main function tokenizes the line using the strtok function from the string.h library. strtok splits a string into tokens based on a specified delimiter. In this case, the delimiter is a space character, so strtok splits the line into the instruction and its operands.
- The main function declares several variables: rd, rs, rt, and offset. These variables are used to store the operands for the MIPS instructions. The meaning of each operand depends on the instruction. For example, in the add instruction, rd is the destination register, rs is the first source register, and rt is the second source register.

- After extracting the instruction and operands, the main function checks if the instruction is one of the load/store instructions (sw or lw). These instructions have a slightly different syntax than the other instructions, so they require special handling. If the instruction is a load/store instruction, the main function extracts the register operands (rt and rs) and the offset value from the line.
- The main function uses the getRegNo function to convert the register names to their corresponding numbers. getRegNo takes a string containing the register name and returns a long long representing the register number.
- The main function then checks the instruction and calls the appropriate function to convert the instruction to its binary representation. The functions add, and, or, andi, ori, addi, sw, lw, and slt are responsible for converting each of the respective instructions to their binary representation.
- The add function takes three long long arguments: rs, rt, and rd. These arguments represent the operands for the add instruction. The function converts these operands to their binary representation and returns the result.
- The and, or, andi, and ori functions work in a similar way. They take two or three long long arguments, depending on whether the instruction is a register-register or immediate instruction. These arguments represent the operands for the instruction. The function converts these operands to their binary representation and returns the result.
- The addi function is similar to the add, and, or, andi, and ori functions, but it takes an immediate value as one of its operands. An immediate value is a constant value that is encoded in the instruction itself, rather than being stored in a register.
- The sw and lw functions are used to store and load values from memory, respectively. These functions take three long long arguments: rs, rt, and offset. The rs and rt arguments represent the registers containing the memory address and the data to be

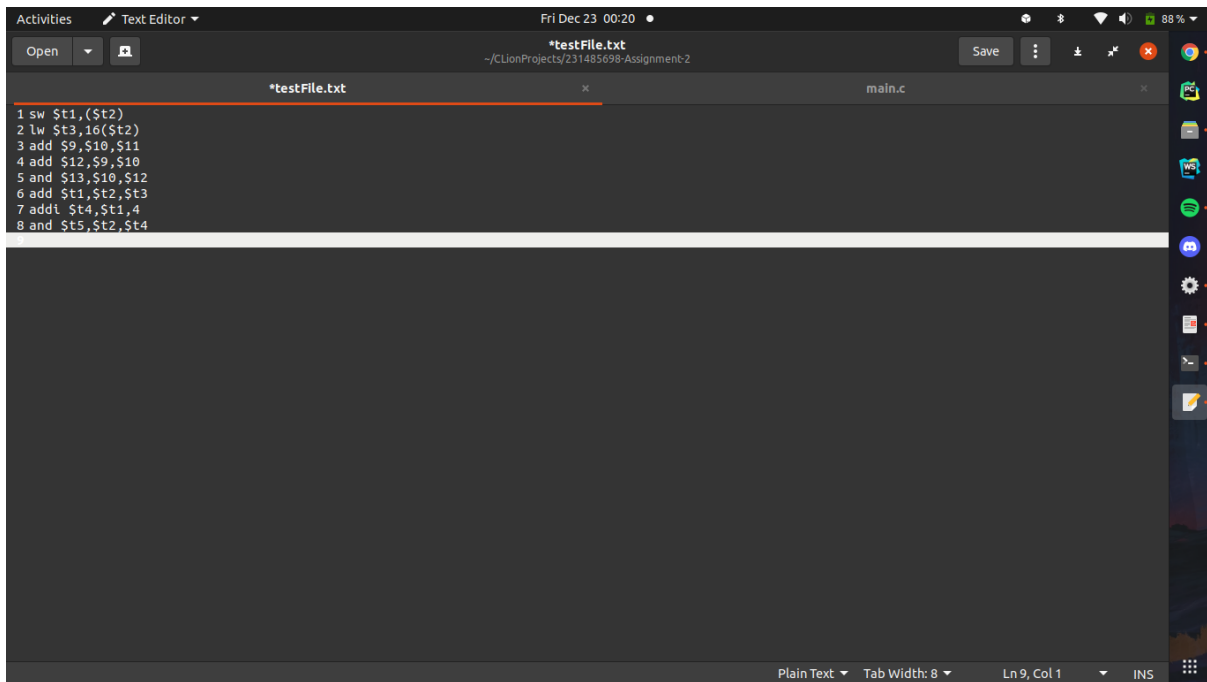
stored/loaded, respectively. The offset argument represents the displacement from the base address specified by the rs register.

- The slt function compares the values in two registers and stores the result in a third register. It takes three long long arguments: rs, rt, and rd. The rs and rt arguments represent the registers containing the values to be compared, and the rd argument represents the destination register where the result will be stored.
- The program also has a helper function called convert, which converts a decimal number to its binary representation. The function takes an int argument n and returns a long long representing the binary representation of n. If n is negative, the function first converts it to its two's complement before returning the binary representation.
- Finally, the main function closes the input file using the fclose function from the stdio.h library, and returns 0 to indicate success.

## **CONCLUSION:**

- This code is a program that converts MIPS assembly language instructions to their corresponding binary code. It reads instructions from an input file and converts them one by one, calling the appropriate function for each instruction. The program supports several MIPS instructions and has helper functions for converting numbers to their binary representation and for converting register names to their corresponding numbers.

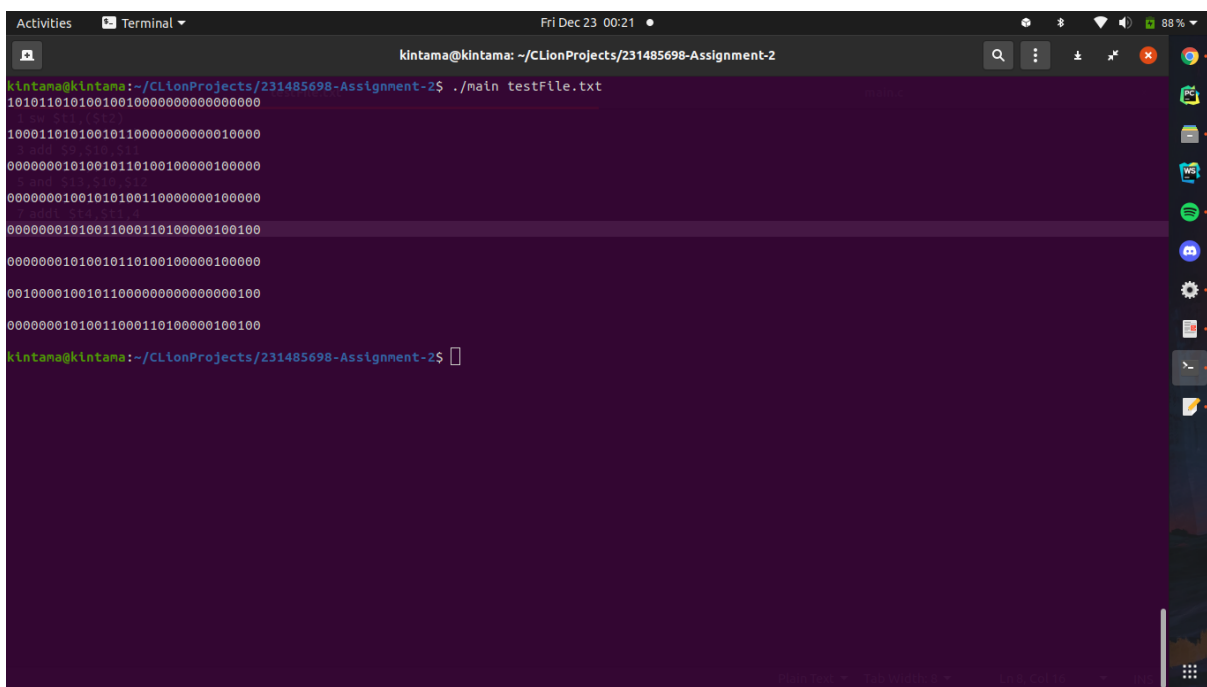
## SCREENSHOTS:



A screenshot of a text editor window titled "Text Editor" with a dark theme. The window shows two tabs: "\*testFile.txt" (active) and "main.c". The active tab contains assembly code for MIPS architecture. The code is as follows:

```
1 sw $t1,($t2)
2 lw $t3,16($t2)
3 add $9,$10,$11
4 add $12,$9,$10
5 and $13,$10,$12
6 add $t1,$t2,$t3
7 addi $t4,$t1,4
8 and $t5,$t2,$t4
```

The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 9, Col 1", and "INS".



A screenshot of a terminal window titled "Terminal" with a dark theme. The prompt is "kintama@kintama: ~/CLionProjects/231485698-Assignment-2". The command executed is `./main testFile.txt`. The output is a series of hexadecimal values, each preceded by a line number from 1 to 8, corresponding to the assembly code in the previous screenshot. The output is as follows:

```
1 10101101010010010000000000000000
2 10001101010010110000000000001000
3 000000010010110100100000100000
4 000000010010110100100000100000
5 000000010010101001000000000000
6 000000010010100110000000000000
7 000000010011000110100000100100
8 000000010010110100100000100000
```

The prompt is now `kintama@kintama: ~/CLionProjects/231485698-Assignment-2$`.

Activities Text Editor Fri Dec 23 00:20

Open \*testFile.txt Save

~/CLionProjects/231485698-Assignment-2

\*testFile.txt main.c

```
1 sw $t1,($t2)
2 lw $t3,16($t2)
```

Plain Text Tab Width: 8 Ln 3, Col 1 INS

Activities Terminal Fri Dec 23 00:20

kintama@kintama: ~/CLionProjects/231485698-Assignment-2

```
kintama@kintama:~/CLionProjects/231485698-Assignment-2$ ./main testFile.txt
10101101010010010000000000000000
100011010100101100000000000010000
kintama@kintama:~/CLionProjects/231485698-Assignment-2$
```

Activities Text Editor Fri Dec 23 00:19

testFile.txt  
~/CLionProjects/231485698-Assignment-2

testFile.txt x main.c x

```
1 add $9,$10,$11
2 add $12,$9,$10
3 and $13,$10,$12
```

Plain Text Tab Width: 8 Ln 4, Col 1 INS

Activities Terminal Fri Dec 23 00:19

kintama@kintama: ~/CLionProjects/231485698-Assignment-2

```
00000001010010110100100000100000
0000000100101010100110000000100000
0000000100101010100110000000100000
00000001001010001101000000100100
000000010010110100100000100000
10101101010010010000000000000000
00100001001011000000000000000100
10001101010010110000000000001000
000000010011000110100000100100
kintama@kintama:~/CLionProjects/231485698-Assignment-2$ ./main testFile.tx
kintama@kintama:~/CLionProjects/231485698-Assignment-2$ ./main testFile.txt
00000001010010110100100000100000
00000001001010100110000000100000
000000010011000110100000100100
kintama@kintama:~/CLionProjects/231485698-Assignment-2$
```



## CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Function prototypes
void add(long long rs, long long rt, long long rd);
void and(long long rs, long long rt, long long rd);
void or(long long rs, long long rt, long long rd);
void andi(long long rs, long long rt, long long imm);
void ori(long long rs, long long rt, long long imm);
void addi(long long rs, long long rt, long long imm);
void sw(long long rs, long long rt, long long offset);
void lw(long long rs, long long rt, long long offset);
void slt(long long rs, long long rt, long long rd);
long long convert(int n);
long long getRegNo(char* reg);

int main(int argc, char *argv[]) {
    // Open the input file
    FILE *fp = fopen("testFile.txt", "r");

    // Check if the file was successfully opened
    if (fp == NULL) {
        printf("Error opening file\n");
        return 1;
    }

    // Read the instructions from the file, one line at a time
    char line[64];
    while (fgets(line, sizeof(line), fp)) {
        // Tokenize the line to extract the instruction and its operands
        char *instruction = strtok(line, " ");
        long long rd, rs, rt, offset;

        if (strcmp(instruction, "sw") == 0 || strcmp(instruction, "lw") == 0) {
            rt = getRegNo(strtok(NULL, ","));
            char * temp = strtok(NULL, ",");

            if (temp[strlen(temp)-1] == '\n') {
                temp[strlen(temp)-1] = '\0';
            }

            if (strchr(temp, '(') != NULL) {
                if (temp[0] == '(') {
                    offset = 0;
                    temp = strtok(temp, "(");
                    temp[strlen(temp)-1] = '\0';
                }
            }
        }
    }
}
```

```

        rs = getRegNo(temp);
    } else {
        char *temp2 = temp;
        offset = atoi(strtok(temp2, "("));
        temp2 = strtok(NULL, "(");
        rs = getRegNo(temp2);
        offset = convert(offset);
    }
}

} else {
    rd = getRegNo(strtok(NULL, ","));
    rs = getRegNo(strtok(NULL, ","));
    rt = getRegNo(strtok(NULL, ","));
}

// Convert the instruction to binary code
if (strcmp(instruction, "add") == 0) {
    add(rs, rt, rd);
} else if (strcmp(instruction, "and") == 0) {
    and(rs, rt, rd);
} else if (strcmp(instruction, "or") == 0) {
    or(rs, rt, rd);
} else if (strcmp(instruction, "andi") == 0) {
    andi(rs, rt, rd);
} else if (strcmp(instruction, "ori") == 0) {
    ori(rs, rt, rd);
} else if (strcmp(instruction, "addi") == 0) {
    addi(rs, rt, rd);
} else if (strcmp(instruction, "sw") == 0) {
    sw(rs, rt, offset);
} else if (strcmp(instruction, "lw") == 0) {
    lw(rs, rt, offset);
} else if (strcmp(instruction, "slt") == 0) {
    slt(rs, rt, rd);
}
}

// Close the file
fclose(fp);

return 0;
}

long long convert(int n) {
    // Check if the number is negative
    // n = atoi(n + 1);
    if (n < 0) {
        // Convert the number to its two's complement representation
        n = ~n + 1;
    }
}

```

```

}

long long bin = 0;
int rem, i = 1;

while (n!=0) {
    rem = n % 2;
    n /= 2;
    bin += rem * i;
    i *= 10;
}

return bin;
}

long long getRegNo(char* reg) {
    if (reg[0] == ' '){
        while (reg[0] == ' ') {
            reg++;
        }
    }

    if (reg[strlen(reg)-1] == '\\n') {
        reg[strlen(reg)-1] = '\\0';
    }

    if (reg[strlen(reg)-1] == '\\') {
        reg[strlen(reg)-1] = '\\0';
    }

    if (strcmp(reg, "$zero") == 0) {
        return convert(0);
    } else if (strcmp(reg, "$at") == 0) {
        return convert(1);
    } else if (strcmp(reg, "$v0") == 0) {
        return convert(2);
    } else if (strcmp(reg, "$v1") == 0) {
        return convert(3);
    } else if (strcmp(reg, "$a0") == 0) {
        return convert(4);
    } else if (strcmp(reg, "$a1") == 0) {
        return convert(5);
    } else if (strcmp(reg, "$a2") == 0) {
        return convert(6);
    } else if (strcmp(reg, "$a3") == 0) {
        return convert(7);
    } else if (strcmp(reg, "$t0") == 0) {
        return convert(8);
    } else if (strcmp(reg, "$t1") == 0) {
        return convert(9);
    } else if (strcmp(reg, "$t2") == 0) {

```

```

        return convert(10);
    } else if (strcmp(reg, "$t3") == 0) {
        return convert(11);
    } else if (strcmp(reg, "$t4") == 0) {
        return convert(12);
    } else if (strcmp(reg, "$t5") == 0) {
        return convert(13);
    } else if (strcmp(reg, "$t6") == 0) {
        return convert(14);
    } else if (strcmp(reg, "$t7") == 0) {
        return convert(15);
    } else if (strcmp(reg, "$s0") == 0) {
        return convert(16);
    } else if (strcmp(reg, "$s1") == 0) {
        return convert(17);
    } else if (strcmp(reg, "$s2") == 0) {
        return convert(18);
    } else if (strcmp(reg, "$s3") == 0) {
        return convert(19);
    } else if (strcmp(reg, "$s4") == 0) {
        return convert(20);
    } else if (strcmp(reg, "$s5") == 0) {
        return convert(21);
    } else if (strcmp(reg, "$s6") == 0) {
        return convert(22);
    } else if (strcmp(reg, "$s7") == 0) {
        return convert(23);
    } else if (strcmp(reg, "$t8") == 0) {
        return convert(24);
    } else if (strcmp(reg, "$t9") == 0) {
        return convert(25);
    } else if (strcmp(reg, "$k0") == 0) {
        return convert(26);
    } else if (strcmp(reg, "$k1") == 0) {
        return convert(27);
    } else if (strcmp(reg, "$gp") == 0) {
        return convert(28);
    } else {
        // Extract the register number from the operand string
        //      long long reg_num = atoi(reg + 1);
        if (reg[0] == '$') {
            reg++;
        }

        // Convert the register number to binary
        return convert(atoi(reg));
    }
}

// add instruction: add $rd, $rs, $rt

```

```

void add(long long rs, long long rt, long long rd) {

    // Convert the instruction to binary code
    printf("000000%05lld%05lld%05lld00000100000\n", rs, rt, rd);

}

// and instruction: and $rd, $rs, $rt
void and(long long rs, long long rt, long long rd) {
    // Convert the instruction to binary code
    printf("000000%05lld%05lld%05lld00000100100\n", rs, rt, rd);
}

// or instruction: or $rd, $rs, $rt
void or(long long rs, long long rt, long long rd) {

    // Convert the instruction to binary code
    printf("000000%05lld%05lld%05lld00000100101\n", rs, rt, rd);
}

// andi instruction: andi $rt, $rs, imm
void andi(long long rs, long long rt, long long imm) {
    // Convert the instruction to binary code
    printf("001100%05lld%05lld%016lld\n", rs, imm, rt);
}

// ori instruction: ori $rt, $rs, imm
void ori(long long rs, long long rt, long long imm) {
    // Convert the instruction to binary code
    printf("001101%05lld%05lld%016lld\n", rs, imm, rt);
}

// addi instruction: addi $rt, $rs, imm
void addi(long long rs, long long rt, long long imm) {
    // Convert the instruction to binary code
    printf("001000%05lld%05lld%016lld\n", rs, imm, rt);
}

// sw instruction: sw $rt, offset($rs)
void sw(long long rs, long long rt, long long offset) {
    // Convert the instruction to binary code
    printf("101011%05lld%05lld%016lld\n", rs, rt, offset);
}

// lw instruction: lw $rt, offset($rs)
void lw(long long rs, long long rt, long long offset) {
    // Convert the instruction to binary code
    printf("100011%05lld%05lld%016lld\n", rs, rt, offset);
}

// slt instruction: slt $rd, $rs, $rt
void slt(long long rs, long long rt, long long rd) {

```

```
// Convert the instruction to binary code
printf("000000%0511d%0511d%0511d00000101010\n", rs, rt, rd);
}
```