

# Table of Contents

Table of Figure.....	1
1 DESIGN .....	3
1.1 R-type instruction .....	5
1.2 I-Type Instruction.....	6
1.3 J-Type Instruction.....	7
1.4 Registers .....	7
1.5 Lookup Table .....	8
2 MipsToHex-STARTING .....	9
3 INTERACTIVE MODE .....	9
4 BATCH MODE .....	12
5 RUN MipsToHex .....	15
6 CONCLUTION.....	15
7 SOFTCOPY OF THE CODE.....	15
REFERENCES .....	37

# Table of Figure

FIGURE 1: BASIC INSTRUCTION FORMATS .....	4
FIGURE 2: R TYPE INSTRUCTIONS .....	5
FIGURE 3: I TYPE INSTRUCTIONS .....	6
FIGURE 4: J TYPE INSTRUCTION .....	7
FIGURE 5: LOOKUP_TABLE.....	8
FIGURE 6: BEQ-BNE INTERACTIVE MODE EXAMPLE .....	10
FIGURE 7: J-JAL INTERACTIVE MODE EXAMPLE .....	10
FIGURE 8: ADDI \$S1, \$S1, -17 INTERACTIVE MODE .....	11
FIGURE 9: INTERACTIVE MODE EXAMPLE .....	11
FIGURE 10: INTERACTIVE MODE EXAMPLE (II).....	11
FIGURE 11: INPUT SOURCE FILE .....	12
FIGURE 12: ERROR MESSAGE BATCH MODE EXAMPLE.....	13
FIGURE 13: INPUT-OUTPUT FILES WITH ERRORS IN BATHC MODE.....	14

## Introduction

The aim of the project is about writing a simple assembler using a high-level programming language such as C or C++, Python, Java, or a scripting language such as Perl or JavaScript to convert any MIPS assembly program containing some of the main MIPS instructions to hexadecimal machine language or object code. The assignment includes two parts which are interactive mode and batch mode. The interactive mode reads an instruction from command line, assembles it to hexadecimal, and outputs the result to the screen. The batch mode reads a source file with extension `‘.src’`, assembles to hexadecimal, and outputs the result to an object code file with extension `‘.obj’`. In addition to batch mode, a list of the instructions can be found in Fig. 2.27 (Page 139) and Fig. 2.25 (Page 134) of the textbook 5th Edition. Your assembler should also be able to handle the pseudo-instructions like `‘move’` provided in the same figure. The last assumption is that the first line of the code is stored at MIPS memory location `‘0x80001000’`. For interactive mode, the converting operation is done by starting address. Finally, the solution should support an interactive mode and a batch mode.

# 1 DESIGN

The project is completed in Java which is an object-oriented language, and Eclipse has been used as a compiler. The reason why Java is used is that there are lots of objects in the project, and creating objects and working on them are easy in Java. The focusing instructions in the assignment are

```
swap: sll $t1, $a1, 2
      add $t1, $a0, $t1
      lw $t0, 0($t1)
      lw $t2, 4($t1)
      sw $t2, 0($t1)
      sw $t0, 4($t1)
sort:  addi $sp, $sp, -20
      sw $ra, 16($sp)
      sw $s3, 12($sp)
      sw $s2, 8($sp)
      sw $s1, 4($sp)
      sw $s0, 0($sp)
      move $s2, $a0
      move $s3, $a1
      move $s0, $zero
for1tst:slt $t0, $s0, $s3
      beq $t0, $zero, exit1
      addi $s1, $s0, -1
for2tst:slti $t0, $s1, 0
      bne $t0, $zero, exit2
      sll $t1, $s1, 2
      add $t2, $s2, $t1
      lw $t3, 0($t2)
      lw $t4, 4($t2)
      slt $t0, $t4, $t3
```

```

    beq $t0, $zero, exit2
    move $a0, $s2
    move $a1, $s1
    jal exit2
    addi $s1, $s1, -1
    j for2tst
exit2: addi $s0, $s0, 1
    j for1tst
exit1: lw $s0, 0($sp)
    lw $s1, 4($sp)
    lw $s2, 8($sp)
    lw $s3, 12($sp)
    lw $ra, 16($sp)
    addi $sp, $sp, 20
    jr $ra

```

This instruction group is used in the batch mode as input.src file. Also, different instruction groups can be used. For the project, there are three type instruction which are R type, I type and J type. While designing the program, these instruction types are considered. In addition to them, a pseudo-instruction ‘move’ is designed by using ‘add’ instruction. In other words, ‘move’ is acting like add with rt = \$zero.

**BASIC INSTRUCTION FORMATS**

<b>R</b>	opcode	rs	rt	rd	shamt	funct
	31 26 25	21 20	16 15	11 10	6 5	0
<b>I</b>	opcode	rs	rt	immediate		
	31 26 25	21 20	16 15	0		
<b>J</b>	opcode	address				
	31 26 25	0				

Figure 1:Basic Instruction Formats

Also, a lookup table has been needed since the program can understand which instructions and registers are in the MIPS assembly language.

## 1.1 R-type instruction

Add, slt, sll, move and jr instructions are in R type format.

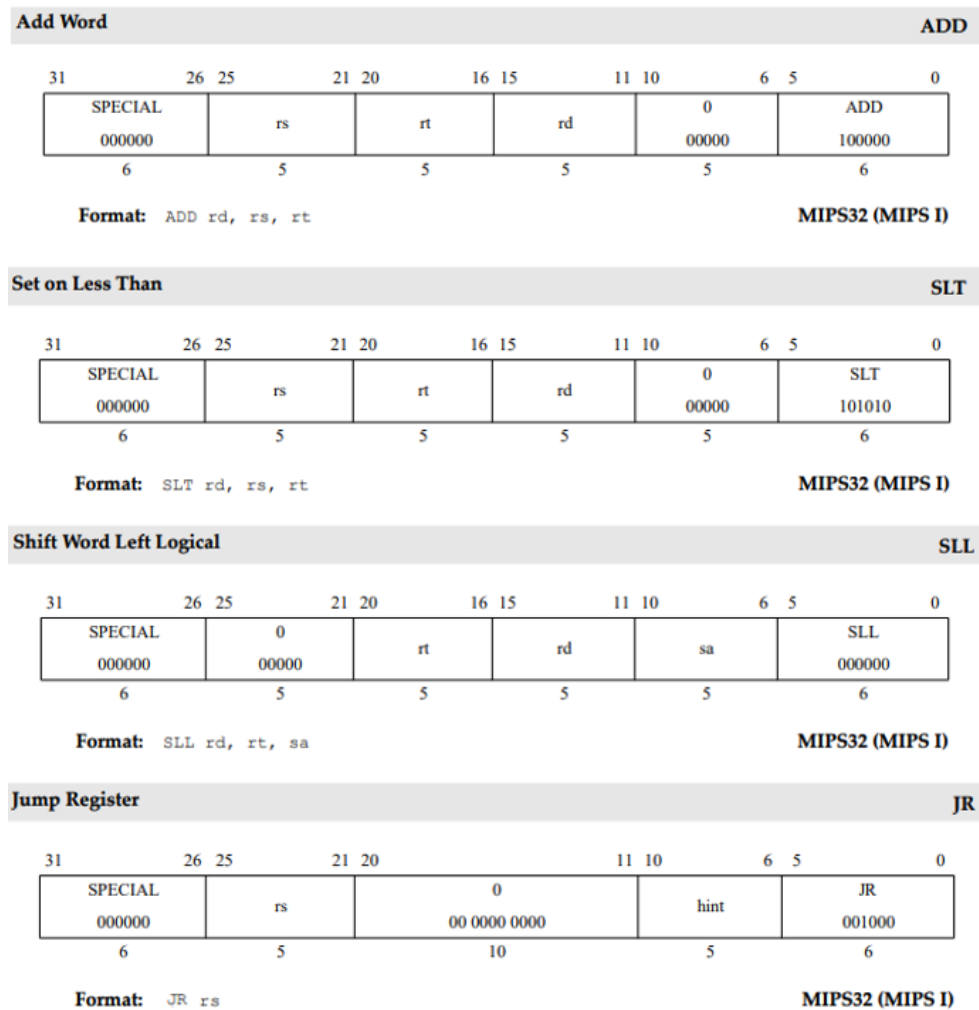


Figure 2: R type instructions

## 1.2 I-Type Instruction

addi ,sw ,lw ,slti ,beq and bne instructions are in I type format.

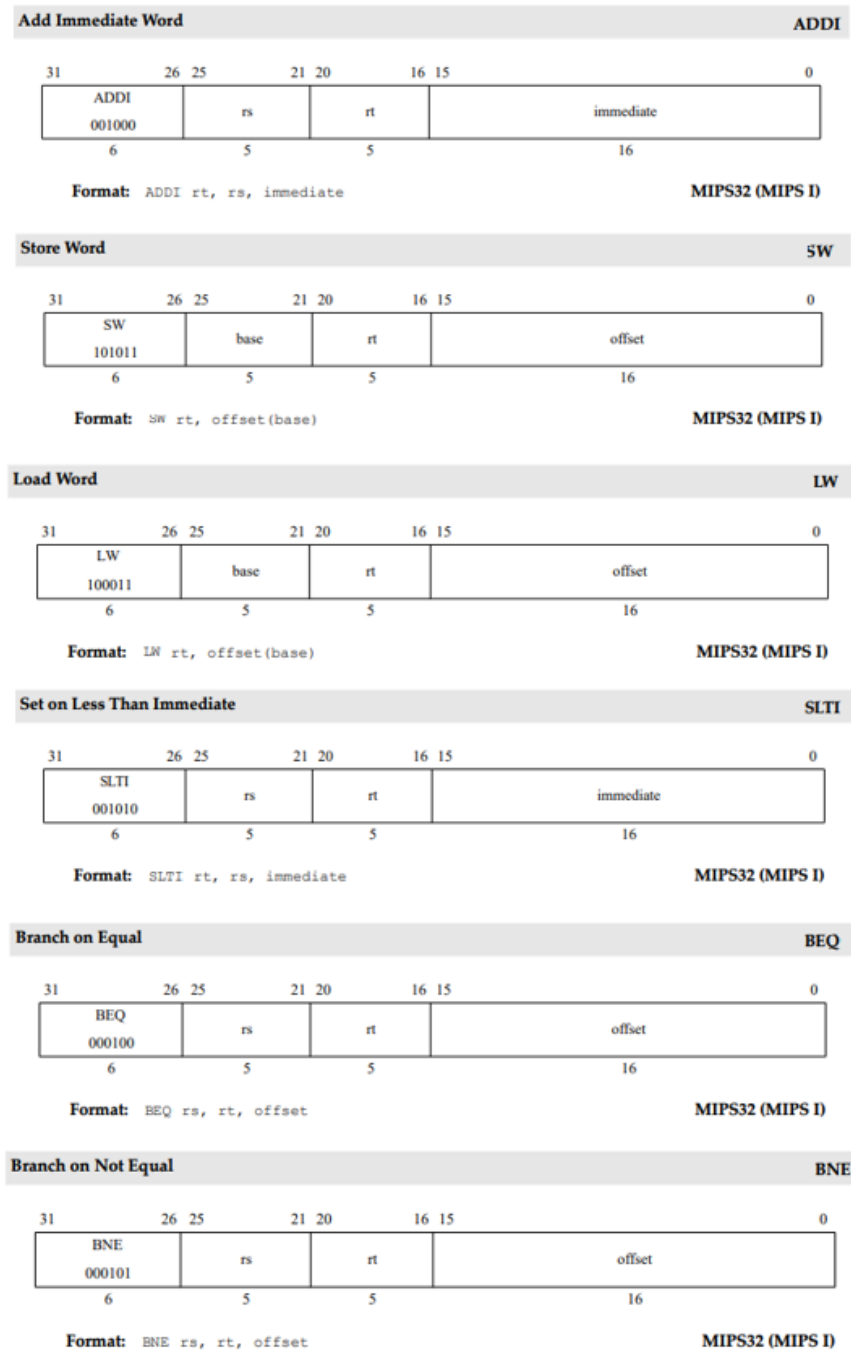


Figure 3: I type instructions

## 1.3 J-Type Instruction

jal and j instructions are in J type format.

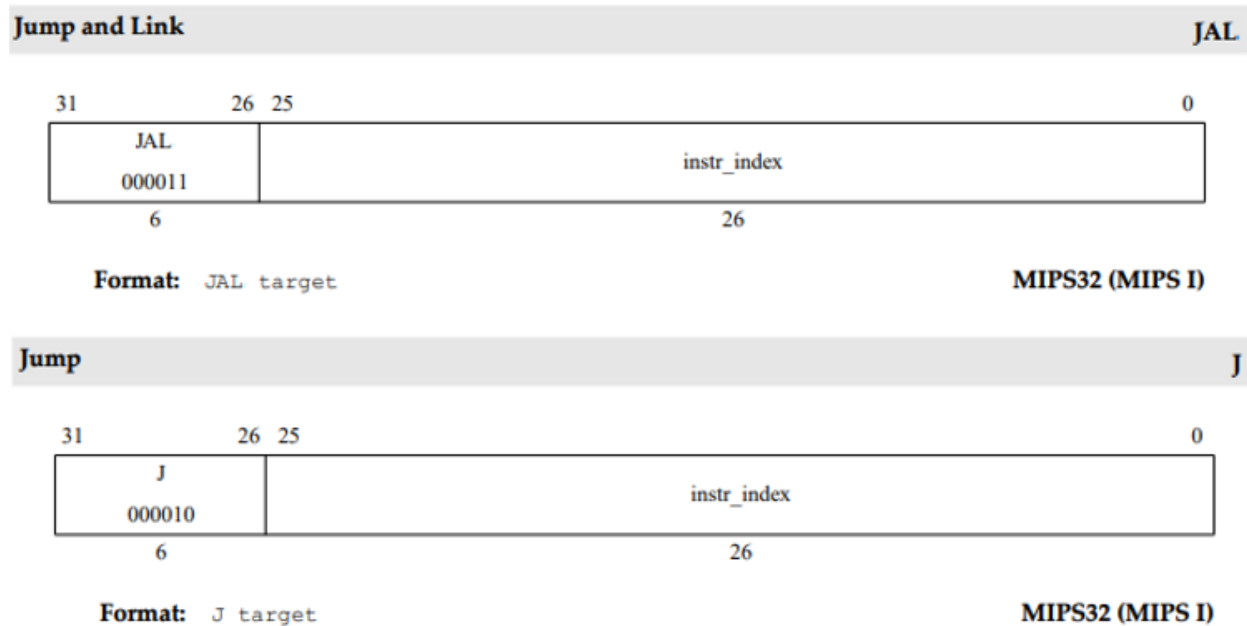


Figure 4: J type instruction

## 1.4 Registers

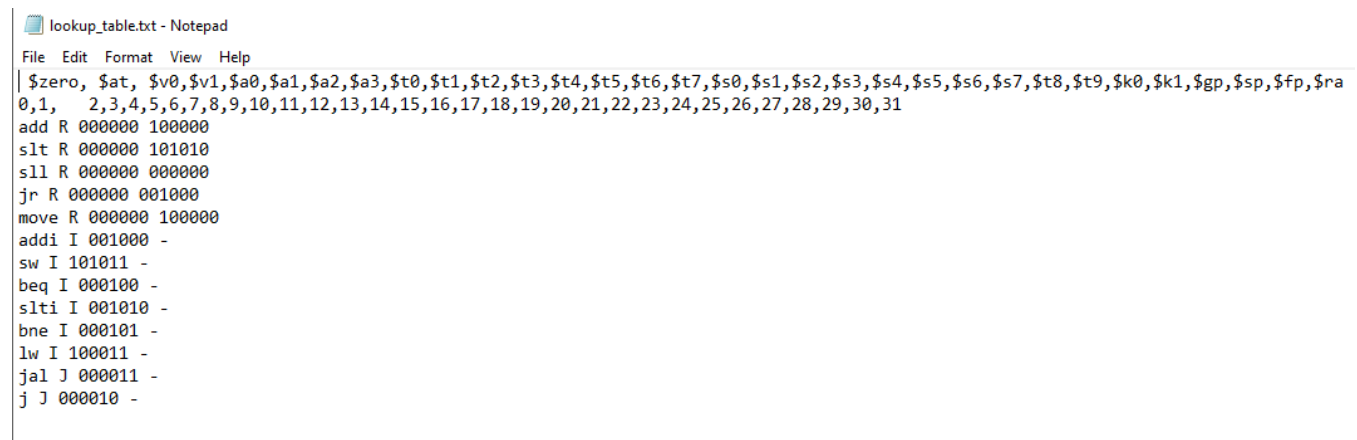
There are 32 registers, and all of them are 5 bits. These registers are;

Register 0: \$zero	Register 1: \$at	Register 2: \$v0
Register 3: \$v1	Register 4: \$a0	Register 5: \$a1
Register 6: \$a2	Register 7: \$a3	Register 8: \$t0
Register 9: \$t1	Register 10: \$t2	Register 11: \$t3
Register 12: \$t4	Register 13: \$t5	Register 14: \$t6
Register 15: \$t7	Register 16: \$s0	Register 17: \$s1
Register 18: \$s2	Register 19: \$s3	Register 20: \$s4
Register 21: \$s5	Register 22: \$s6	Register 23: \$s7
Register 24: \$t8	Register 25: \$t9	Register 26: \$k0
Register 27: \$k1	Register 28: \$gp	Register 29: \$sp
Register 30: \$s8	Register 31: \$ra	

## 1.5 Lookup Table

For understanding the MIPS assembler code, lookup table has been necessary. The table includes the information of instructions which are opcode and function in binary. Also, the register is included with their addresses in decimal. Because of sort calculation in converting part, and easy following the values, registers' information is in decimal, and instructions' is in binary.

The lookup table is a text file '.txt'. It's first row is for register names such as '\$zero', '\$s0' and '\$2', and these registers' values is placed in the second row. After registers' part, instructions' information start in the third row, and each instruction has one row for itself. Their row format is like 'NAME TYPE OPCODE FUNCTION'. For function part, only R type instructions has this part, so I and J types do not have function part. Due to sequence, '-' must be entered for these two instructions type. The lookup table can be seen below in figure 5, and a new instruction can be added by using the proper format.



```
lookup_table.txt - Notepad
File Edit Format View Help
| $zero, $at, $v0,$v1,$a0,$a1,$a2,$a3,$t0,$t1,$t2,$t3,$t4,$t5,$t6,$t7,$s0,$s1,$s2,$s3,$s4,$s5,$s6,$s7,$t8,$t9,$k0,$k1,$gp,$sp,$fp,$ra
0,1, 2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
add R 000000 100000
sll R 000000 101010
sll R 000000 000000
jr R 000000 001000
move R 000000 100000
addi I 001000 -
sw I 101011 -
beq I 000100 -
slti I 001010 -
bne I 000101 -
lw I 100011 -
jal J 000011 -
j J 000010 -
```

Figure 5: Lookup\_table



## 2 MipsToHex-STARTING

When the assembler is starting, a window is opened. The window has three buttons which are interactive, batch, and exit. For closing the assembler, select the exit button. The interactive button provides one line comment for the user. If the batch button is pressed, the assembler takes the input source file '.src', and convert it to the hexadecimal machine language, then save the outputs into output object file '.obj'.

## 3 INTERACTIVE MODE

In the interactive mode, the user can enter one line MIPS assembler comment, the mode converts it into the hexadecimal machine language to show on the screen.

### Restrictions:

- The lookup table must include instruction and registers.
- The input must be one line.
- The input must fulfill in the instruction format.

Beq, bne, j and jal needs a label to fulfill their formats. Yet, in one line code there is no way to use labels. Thus, when running any of these instructions in this mode, a row index must be entered. Beq and bne works like going forward +row index which is entered by the user. Since J and jal are not direct addressing mode, they work like that the label's address is calculated according to the start address '0x80001000', also, the label is at the row index which is entered by the user. To be clearer, you can check the below example.

### Example for beq and bne:

beq \$s1, \$s2, L	L: instruction
instruction	instruction
instruction	instruction
L: instruction	bne \$s1, \$s2, L
beq \$s1, \$s2, 3	bne \$s1, \$s2, -3
000100 10001 10010 0000 0000 0000 0011	000101 10001 10010 1111 1111 1111 1101
0x12320003	0x1632FFFD

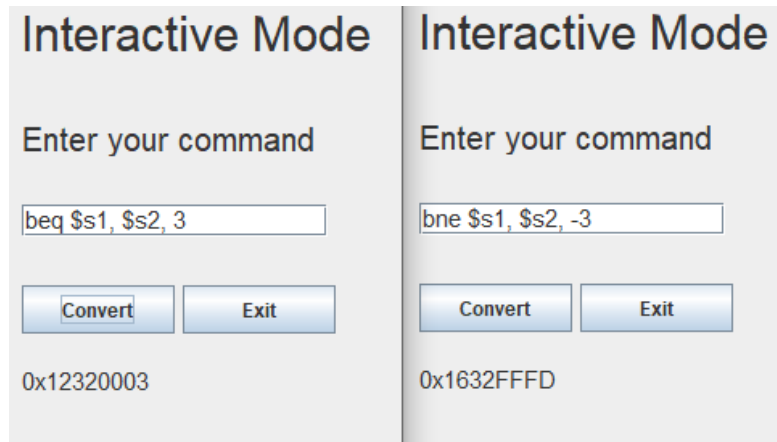


Figure 6: BEQ-BNE interactive mode example

Example for j and jal:

0x80001000	j L	L:	instruction
0x80001004	instruction		instruction
0x80001008	instruction		instruction
0x8000100C	L: instruction		jal L

0x8000100C  
 1000 0000 0000 0000 0001 0000 0000 1100  
 0000 0000 0000 0001 0000 0000 11 - 00  
 000000000000000010000000011  
 j 3  
 0000 10 00 0000000000000010000000011  
 0x08000403

0x80001000  
 1000 0000 0000 0000 0001 0000 0000 0000  
 0000 0000 0000 0001 0000 0000 00 - 00  
 000000000000000010000000000  
 jal 0  
 0000 11 00 0000000000000010000000000  
 0x0C000400

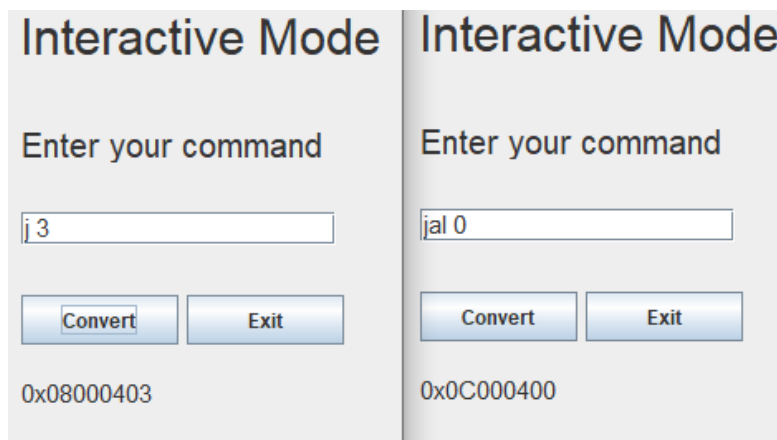


Figure 7: J-JAL interactive mode example

In additionally, since the mode handle white space errors, it can be said that the mode is user friendly. The asked input instruction which is “addi \$s1, \$s1, -17” is showed below with the version white space errors.

Interactive Mode	Interactive Mode
Enter your command	Enter your command
<input type="text" value="addi \$s1, \$s1, -17"/>	<input type="text" value="addi \$s 1, \$ s 1, - 17"/>
<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>
0x2231FFEF	0x2231FFEF

Figure 8: ADDI \$s1, \$s1, -17 interactive mode

The rest examples can be seen in figure 9 and figure 10.

Interactive Mode	Interactive Mode	Interactive Mode	Interactive Mode
Enter your command	Enter your command	Enter your command	Enter your command
<input type="text" value="move \$s1,\$s2"/>	<input type="text" value="add \$s1,\$s2 , \$zero"/>	<input type="text" value="lw \$t0, 4(\$t1)"/>	<input type="text" value="sw \$t0, 4(\$t1)"/>
<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>
0x02408820	0x02408820	0x8D280004	0xAD280004

Figure 9: Interactive mode example

Interactive Mode	Interactive Mode	Interactive Mode	Interactive Mode
Enter your command	Enter your command	Enter your command	Enter your command
<input type="text" value="jr \$ra"/>	<input type="text" value="sll \$s1,\$s2,2"/>	<input type="text" value="slt \$t0,\$t1,\$t2"/>	<input type="text" value="slti \$t0,\$t1,3"/>
<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>	<input type="button" value="Convert"/> <input type="button" value="Exit"/>
0x03E00008	0x00128880	0x012A402A	0x29280003

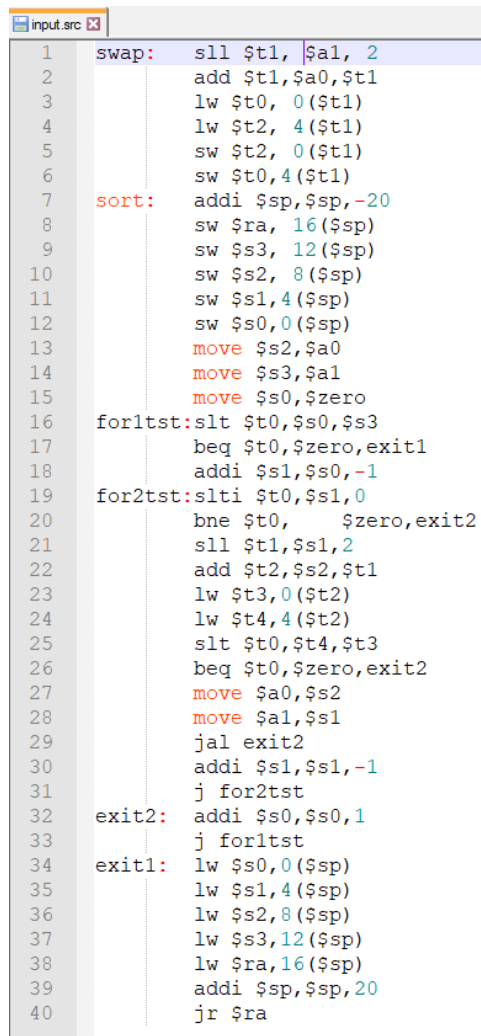
Figure 10: Interactive mode example (II)

## 4 BATCH MODE

In the batch mode, the assembler takes input from the source file named 'input' which is in project folder and then it creates an object file named 'output' to project folder and writes into it. If there is a file with the name 'output.obj', the MipsToHex write the outputs into that file. Our source file has .src format, and our output file has .obj format.

### Source file restriction:

- The file must be in the project folder.
- Labels must end with ':'
- Registers must be separated with ','
- The lines must include at least one instruction and register or label's name



```
1 swap: sll $t1, $a1, 2
2       add $t1, $a0, $t1
3       lw $t0, 0($t1)
4       lw $t2, 4($t1)
5       sw $t2, 0($t1)
6       sw $t0, 4($t1)
7 sort: addi $sp, $sp, -20
8       sw $ra, 16($sp)
9       sw $s3, 12($sp)
10      sw $s2, 8($sp)
11      sw $s1, 4($sp)
12      sw $s0, 0($sp)
13      move $s2, $a0
14      move $s3, $a1
15      move $s0, $zero
16 for1tst: slt $t0, $s0, $s3
17          beq $t0, $zero, exit1
18          addi $s1, $s0, -1
19 for2tst: slti $t0, $s1, 0
20          bne $t0, $zero, exit2
21          sll $t1, $s1, 2
22          add $t2, $s2, $t1
23          lw $t3, 0($t2)
24          lw $t4, 4($t2)
25          slt $t0, $t4, $t3
26          beq $t0, $zero, exit2
27          move $a0, $s2
28          move $a1, $s1
29          jal exit2
30          addi $s1, $s1, -1
31          j for2tst
32 exit2: addi $s0, $s0, 1
33          j for1tst
34 exit1: lw $s0, 0($sp)
35          lw $s1, 4($sp)
36          lw $s2, 8($sp)
37          lw $s3, 12($sp)
38          lw $ra, 16($sp)
39          addi $sp, $sp, 20
40          jr $ra
```

Figure 11: Input source file

Also, the batch mode is user friendly like the interactive mode. The mode can handle some error which are whitespace, completing the work with some errors and informing the errors. For instance, the assembler is worked with some white spaces and some errors in figure 12 and figure 13.

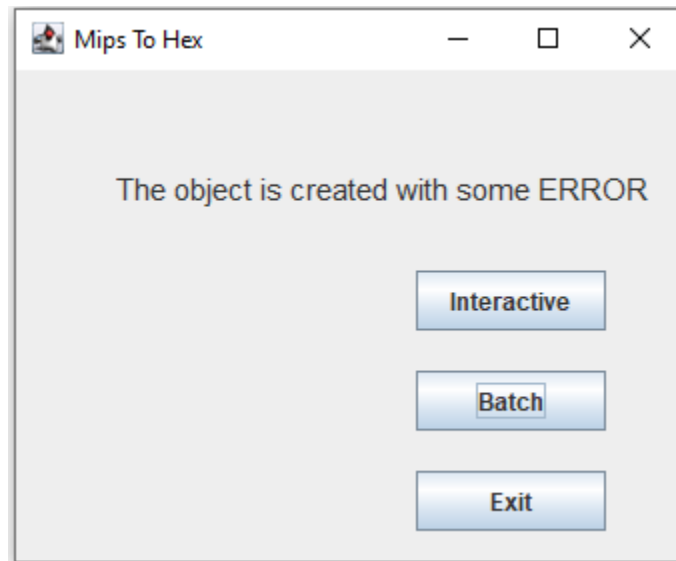


Figure 12: Error message batch mode example

The errors are caused by user's typing mistakes. These are in line 1 with whitespace, line 3 with operand, and line 5 with registers. The input file and the output file can be seen below with 1 fixed error line and 2 information about errors.

input.src	output.obj
1 swap: sll \$t1, \$a 1, 2	1 0x00054880
2 add \$t1,\$a0,\$t1	2 0x00894820
3 l \$t0, 0(\$t1)	3 The operand is unknown
4 lw \$t2, 4(\$t1)	4 0x8D2A0004
5 sw \$t, 0(\$t1)	5 Unknown register
6 sw \$t0,4(\$t1)	6 0xAD280004
7 sort: addi \$sp,\$sp,-20	7 0x23BDFFE0
8 sw \$ra, 16(\$sp)	8 0xAFBF0010
9 sw \$s3, 12(\$sp)	9 0xAFB3000C
10 sw \$s2, 8(\$sp)	10 0xAFB20008
11 sw \$s1,4(\$sp)	11 0xAFB10004
12 sw \$s0,0(\$sp)	12 0xAFB00000
13 move \$s2,\$a0	13 0x00809020
14 move \$s3,\$a1	14 0x00A09820
15 move \$s0,\$zero	15 0x00008020
16 for1tst:slt \$t0,\$s0,\$s3	16 0x0213402A
17 beq \$t0,\$zero,exit1	17 0x11000010
18 addi \$s1,\$s0,-1	18 0x2211FFFF
19 for2tst:slti \$t0,\$s1,0	19 0x2A280000
20 bne \$t0, \$zero,exit2	20 0x1500000B
21 sll \$t1,\$s1,2	21 0x00114880
22 add \$t2,\$s2,\$t1	22 0x02495020
23 lw \$t3,0(\$t2)	23 0x8D4B0000
24 lw \$t4,4(\$t2)	24 0x8D4C0004
25 slt \$t0,\$t4,\$t3	25 0x018B402A
26 beq \$t0,\$zero,exit2	26 0x11000005
27 move \$a0,\$s2	27 0x02402020
28 move \$a1,\$s1	28 0x02202820
29 jal exit2	29 0x0C00041F
30 addi \$s1,\$s1,-1	30 0x2231FFFF
31 j for2tst	31 0x08000412
32 exit2: addi \$s0,\$s0,1	32 0x22100001
33 j for1tst	33 0x0800040F
34 exit1: lw \$s0,0(\$sp)	34 0x8FB00000
35 lw \$s1,4(\$sp)	35 0x8FB10004
36 lw \$s2,8(\$sp)	36 0x8FB20008
37 lw \$s3,12(\$sp)	37 0x8FB3000C
38 lw \$ra,16(\$sp)	38 0x8FBF0010
39 addi \$sp,\$sp,20	39 0x23BD0014
40 jr \$ra	40 0x03E00008

Figure 13: Input-output files with errors in batch mode

## 5 RUN MipsToHex

There are a '.bat' and a '.jar' file in project folder. You can run executable '.jar' file via double click on '.bat' file which named "MipsToHex". Then menu will lead you.

## 6 CONCLUTION

The goal is converting mips assembler code to hexadecimal machine code in two different mode, and the small code works correctly. While working on the project, new java skills has been got Also, reading instructions, and understanding them have been improved. Finally, a convincing project has been completed.

## 7 SOFTCOPY OF THE CODE

```
public class main {  
    public static void main(String[] args) {  
        final String startPoint="100000000000000001000000000000"; // initial  
address 0x80001000  
        final String lookupTable="\\lookup_table.txt"; // lookup table's  
path  
        final String input="\\input.src"; // .src file's path  
        final String output="\\output.obj"; // .obj file's path  
        new MipsToHex(lookupTable, startPoint, input, output);  
    }  
}  
  
public abstract class Instruction {  
    private String machine; // machine code in hex  
    private String opcode; // instruction[31-25]  
    // the function convertHex take the 32 bit binary number and convert it to  
hexadecimal number  
    String convertHex(String binary) {  
        int digitNumber = 1;  
        int sum = 0;  
        String hex="0x";  
        for(int i = 0; i < binary.length(); i++){  
            if(digitNumber == 1)  
                sum += Integer.parseInt(binary.charAt(i) + "")*8;  
            else if(digitNumber == 2)
```

```

        sum += Integer.parseInt(binary.charAt(i) + "")*4;

    else if(digitNumber == 3)

        sum += Integer.parseInt(binary.charAt(i) + "")*2;

    else if(digitNumber == 4 || i < binary.length()+1){
        sum += Integer.parseInt(binary.charAt(i) + "")*1;
        digitNumber = 0;
        if(sum < 10)
            hex = hex + sum;
        else if(sum == 10)
            hex = hex + "A";
        else if(sum == 11)
            hex = hex + 'B';
        else if(sum == 12)
            hex = hex + "C";
        else if(sum == 13)
            hex = hex + "D";
        else if(sum == 14)
            hex = hex + "E";
        else if(sum == 15)
            hex = hex + "F";
        sum = 0;
    }

    digitNumber++;
}

return hex;
}

// set and get functions
public String getMachine() {
    return machine;
}

public void setmMachinen(String machine) {
    this.machine = machine;
}

public String getOpcode() {
    return opcode;
}

public void setOpcode(String opcode) {
    this.opcode = opcode;
}
}

public class Rtype extends Instruction{
    private String rs;           // instruction[24-20]
    private String rt;           // instruction[19-15]
    private String rd;           // instruction[14-10]

```



```

    private String shamt = "0"; // instruction[9-5] with default value
    private String functionCode; // instruction[4-0]

    public String RtypeConverter(LookupTable lookupTable, String operand, String[]
registers, int operandIndex) {
        if (registers.length == 3) {
            // if the 3. element of the register part is integer such as "sll
$t0,$t1,5"

            if(Character.isDigit(registers[2].charAt(0))){
                this.rd = registers[0];
                this.rt = registers[1];
                this.rs = "$zero"; // for zero value
                this.shamt = registers[2];
            }
            else {
                this.rd = registers[0];
                this.rs = registers[1];
                this.rt = registers[2];
            }
            // find registers value
            int indexRs=-1;
            int indexRt=-1;
            int indexRd=-1;
            int found = 0;
            for(int i = 0; i<lookupTable.registersName.length;i++)
            {
                if(this.rs.equals(lookupTable.registersName[i])){
                    found++;
                    indexRs = i;
                }

                if(this.rt.equals(lookupTable.registersName[i])) {
                    found++;
                    indexRt = i;
                }

                if(this.rd.equals(lookupTable.registersName[i])){
                    found++;
                    indexRd = i;
                }

                if (found == 3) break;
            }
            // if any of the register is unknown
            if((indexRs == -1) || (indexRt == -1 ) || (indexRd == -1 ))

                return "Unknown register";
            else {
                // set registers, opcode and function values
                this.setRt(lookupTable.registersValue[indexRt]);
                this.setRs(lookupTable.registersValue[indexRs]);
                this.setRd(lookupTable.registersValue[indexRd]);
                // convert values to binary numbers
                // convert shamt
                String binaryValue;

```

```

        int integerValue = Integer.parseInt(this.shamt);
        binaryValue = Integer.toBinaryString(integerValue);
        if (binaryValue.length() < 5){
            int max = 5-binaryValue.length();
            for(int i=0; i < max; i++)
                binaryValue = '0' + binaryValue;
        }
        this.setShamt(binaryValue);
        // convert rs
        integerValue = Integer.parseInt(this.rs);
        binaryValue = Integer.toBinaryString(integerValue);
        if (binaryValue.length() < 5){
            int max = 5-binaryValue.length();
            for(int i=0; i < max; i++)
                binaryValue = '0' + binaryValue;
        }
        this.setRs(binaryValue);
        // convert rt
        integerValue = Integer.parseInt(this.rt);
        binaryValue = Integer.toBinaryString(integerValue);
        if (binaryValue.length() < 5){
            int max = 5-binaryValue.length();
            for(int i=0; i < max; i++)
                binaryValue = '0' + binaryValue;
        }
        this.setRt(binaryValue);
        // convert rd
        integerValue = Integer.parseInt(this.rd);
        binaryValue = Integer.toBinaryString(integerValue);
        if (binaryValue.length() < 5){
            int max = 5-binaryValue.length();
            for(int i=0; i < max; i++)
                binaryValue = '0' + binaryValue;
        }
        this.setRd(binaryValue);
        // opcode and function code get binary numbers from the
lookup table
        this.setOpcode(lookupTable.opcodes.get(operandIndex));

        this.setFunctionCode(lookupTable.functions.get(operandIndex));
    }
    // registers.length == 3
    this.setmMachinen(convertHex(this.getOpcode() + this.getRs() +
this.getRt() + this.getRd() + this.getShamt() + this.getFunctionCode()));
    }
    // such as move instruction which is pseudo
    else if (registers.length == 2) {
        // also we need to add all the instruction which is same format
by using 'else if'
        if (operand.equals("move")) {
            this.rd = registers[0];
            this.rt = "$zero";
            this.rs = registers[1];
            // find registers value
            int indexRs=-1;

```

```

int indexRt=-1;
int indexRd=-1;
int found = 0;
for(int i = 0; i<lookupTable.registersName.length;i++)
{
    if(this.rs.equals(lookupTable.registersName[i])){
        found++;
        indexRs = i;
    }

    if(this.rt.equals(lookupTable.registersName[i])) {
        found++;
        indexRt = i;
    }

    if(this.rd.equals(lookupTable.registersName[i])){
        found++;
        indexRd = i;
    }

    if (found == 3) break;
}
// if any of the register is unknown
if((indexRs == -1) || (indexRt == -1 ) || (indexRd == -1
))

return "Unknown register";
else {
    // set registers, opcode and function values
    this.setRt(lookupTable.registersValue[indexRt]);
    this.setRs(lookupTable.registersValue[indexRs]);
    this.setRd(lookupTable.registersValue[indexRd]);
    // convert values to binary numbers
    // convert shamt
    String binaryValue;
    int integerValue = Integer.parseInt(this.shamt);
    binaryValue = Integer.toBinaryString(integerValue);
    if (binaryValue.length() < 5){
        int max = 5-binaryValue.length();
        for(int i=0; i < max; i++)
            binaryValue = '0' + binaryValue;
    }
    this.setShamt(binaryValue);
    // convert rs
    integerValue = Integer.parseInt(this.rs);
    binaryValue = Integer.toBinaryString(integerValue);
    if (binaryValue.length() < 5){
        int max = 5-binaryValue.length();
        for(int i=0; i < max; i++)
            binaryValue = '0' + binaryValue;
    }
    this.setRs(binaryValue);
    // convert rt
    integerValue = Integer.parseInt(this.rt);
    binaryValue = Integer.toBinaryString(integerValue);
    if (binaryValue.length() < 5){

```

```

        int max = 5-binaryValue.length();
        for(int i=0; i < max; i++)
            binaryValue = '0' + binaryValue;
    }
    this.setRt(binaryValue);
    // convert rd
    integerValue = Integer.parseInt(this.rd);
    binaryValue = Integer.toBinaryString(integerValue);
    if (binaryValue.length() < 5){
        int max = 5-binaryValue.length();
        for(int i=0; i < max; i++)
            binaryValue = '0' + binaryValue;
    }
    this.setRd(binaryValue);
    // opcode and function code get binary numbers from
the lookup table

    this.setOpcode(lookupTable.opcodes.get(operandIndex));

    this.setFunctionCode(lookupTable.functions.get(operandIndex));
    }
    else
        return "Reconsider instruction";

    // registers.length == 2
    this.setmMachinen(convertHex(this.getOpcode() + this.getRs() +
this.getRt() + this.getRd() + this.getShamt() + this.getFunctionCode()));
    }
    // such as jr instruction
    else if (registers.length == 1){
        // also we need to add all the instruction which is same format
by using 'else if'
        if (operand.equals("jr")) {
            // find registers value
            this.rs = registers[0];
            int indexRs=-1;
            for(int i = 0; i<lookupTable.registersName.length;i++)
            {
                if(this.rs.equals(lookupTable.registersName[i])) {
                    indexRs = i;
                    break;
                }
            }
            // if there is no rs register in the lookup table
            if((indexRs == -1))
                return "Unknown register";
            else {
                this.setRs(lookupTable.registersValue[indexRs]);
                this.setRt(lookupTable.registersValue[0]);
                this.setRd(lookupTable.registersValue[0]);
                // convert values to binary numbers
                // convert shamt
                String binaryValue;
                int integerValue = Integer.parseInt(this.shamt);

```

```

binaryValue = Integer.toBinaryString(integerValue);
if (binaryValue.length() < 5){
    int max = 5-binaryValue.length();
    for(int i=0; i < max; i++)
        binaryValue = '0' + binaryValue;
}
this.setShamt(binaryValue);
// convert rs
integerValue = Integer.parseInt(this.rs);
binaryValue = Integer.toBinaryString(integerValue);
if (binaryValue.length() < 5){
    int max = 5-binaryValue.length();
    for(int i=0; i < max; i++)
        binaryValue = '0' + binaryValue;
}
this.setRs(binaryValue);
// convert rt
integerValue = Integer.parseInt(this.rt);
binaryValue = Integer.toBinaryString(integerValue);
if (binaryValue.length() < 5){
    int max = 5-binaryValue.length();
    for(int i=0; i < max; i++)
        binaryValue = '0' + binaryValue;
}
this.setRt(binaryValue);
// convert rd
integerValue = Integer.parseInt(this.rd);
binaryValue = Integer.toBinaryString(integerValue);
if (binaryValue.length() < 5){
    int max = 5-binaryValue.length();
    for(int i=0; i < max; i++)
        binaryValue = '0' + binaryValue;
}
this.setRd(binaryValue);
// opcode and function code get binary numbers from

```

the lookup table

```

    this.setOpcode(lookupTable.opcodes.get(operandIndex));
    this.setFunctionCode(lookupTable.functions.get(operandIndex));
}
else
    return "Reconsider instruction";

// registers.length == 1
this.setmMachinen(convertHex(this.getOpcode() + this.getRs() +
this.getRt() + this.getRd() + this.getShamt() + this.getFunctionCode()));
}
else
    return "Reconsider instruction";

return this.getMachine();
}

```

```

// set and get functions
public String getRs() {
    return rs;
}

public void setRs(String rs) {
    this.rs = rs;
}

public String getRt() {
    return rt;
}

public void setRt(String rt) {
    this.rt = rt;
}

public String getRd() {
    return rd;
}

public void setRd(String rd) {
    this.rd = rd;
}

public String getShamt() {
    return shamt;
}

public void setShamt(String shamt) {
    this.shamt = shamt;
}

public String getFunctionCode() {
    return functionCode;
}

public void setFunctionCode(String functionCode) {
    this.functionCode = functionCode;
}
}

public class Itype extends Instruction{
    private String rs;           // instruction[24-20]
    private String rt;           // instruction[19-15]
    private String offset;       // instruction[14-0]

    public String ItypeConverter(LookupTable lookupTable, String operand, String[]
registers, int operandIndex) {
        if(registers.length==3) {
            // if the operand is a branch type that means its first character
is 'b'
            if(operand.charAt(0) == 'b'){

```

```

        this.rs = registers[0];
        this.rt = registers[1];
        this.offset= registers[2];
    }
    else{
        this.rt = registers[0];
        this.rs = registers[1];
        this.offset= registers[2];
    }
}

//For lw&sw type instructions
if(registers.length==2){
    // take first part for rt
    this.rt = registers[0];
    // Separate second part. before '(' for immediate and between two
    paranteses for rs
    String[] rest = registers[1].split("[()");
    this.offset = rest[0];
    this.rs = rest[1].substring(0, rest[1].length() - 1);
}

// find registers value
int indexRt=-1;
int indexRs=-1;
int found = 0;
for(int i = 0; i<lookupTable.registersName.length;i++)
{
    if(this.rs.equals(lookupTable.registersName[i])){
        found++;
        indexRs = i;
    }
    if(this.rt.equals(lookupTable.registersName[i])){
        found++;
        indexRt = i;
    }
    if (found == 2) break;
}
// if any of the register is unknown
if((indexRt == -1) || (indexRs == -1 ) )
    return "Unknown register";
// find immediate value
boolean sign;
//checking if immediate is negative or positive
if(this.offset.charAt(0) == '-'){
    this.offset = this.offset.substring(1);
    sign = false;
}
else
    sign = true;

try {
    int intImmediate=Integer.parseInt(this.offset);
    this.offset = Integer.toBinaryString(intImmediate);
}

```

```

}catch (Exception e) {
    return "Error: Immediate number";
}
// check Positive immediate
if((this.offset.length()>16) && (sign))
    return "Posivite immediates cannot be bigger than 16 bits";

// check Negative immediate
else if((this.offset.length()>15) && (!sign))
    return "Negative immediates cannot be bigger than 15 bits";
// if immediate number is valid
else {
    char[] immAr = new char[16];

    int j=1;
    for(int i = 15; i > -1 ; i--)
    {
        int len = this.offset.length()-j;
        //Fulfilling positive immediate to 16bits
        if(sign){
            if(len > -1) {
                immAr[i] = this.offset.charAt(len);
            }
            else
                immAr[i]='0';
            j++;
        }
        //Fulfilling negative immediate to 16bits
        else{
            if(len > -1) {
                immAr[i] = this.offset.charAt(len);
            }
            else{
                if (i == 0)
                    immAr[i]='1';
                else
                    immAr[i]='0';
            }
            j++;
        }
    }
    // 2's complement
    if(!sign) {
        boolean meetone = false;
        for(int i=immAr.length-1; i > 0; i--) {
            if(immAr[i] == '1'){
                if(meetone == false)
                    meetone =true;
                else
                    immAr[i] = '0';
            }
            else {
                if(meetone == true)
                    immAr[i] = '1';
            }
        }
    }
}

```



```

    }
}
// set rs, rt, offset and opcode vales
this.offset = new String(immAr); // binary
this.setRt(lookupTable.registersValue[indexRt]);
this.setRs(lookupTable.registersValue[indexRs]);
// convert rs
int integerValue = Integer.parseInt(this.rs);
String binaryValue = Integer.toBinaryString(integerValue);
if (binaryValue.length() < 5){
    int max = 5-binaryValue.length();
    for(int i=0; i < max; i++)
        binaryValue = '0' + binaryValue;
}
this.setRs(binaryValue);
// convert rt
integerValue = Integer.parseInt(this.rt);
binaryValue = Integer.toBinaryString(integerValue);
if (binaryValue.length() < 5){
    int max = 5-binaryValue.length();
    for(int i=0; i < max; i++)
        binaryValue = '0' + binaryValue;
}
this.setRt(binaryValue);
this.setOpcode(lookupTable.opcodes.get(operandIndex));

this.setmMachinen(convertHex(this.getOpcode() + this.getRs() +
this.getRt() + this.getOffset()));
}

return this.getMachine();
}

// set and get functions
public String getRs() {
    return rs;
}

public void setRs(String rs) {
    this.rs = rs;
}

public String getRt() {
    return rt;
}

public void setRt(String rt) {
    this.rt = rt;
}

public String getOffset() {
    return offset;
}

```

```

    public void setOffset(String offset) {
        this.offset = offset;
    }

}

public class Jtype extends Instruction{

    private String target;           // instruction[24-0]

    public String JtypeInteractionConverter(LookupTable lookupTable, String
operand, String[] target, int operandIndex, String startPoint) {
        if (target.length != 1)
            return "Error: Too many arguments";

        else {
            this.target = target[0];
            // find target value
            try {
                int intTarget=Integer.parseInt(this.target);
                if (intTarget > -1)
                    this.target = Integer.toBinaryString(intTarget);
                else
                    throw new Exception();

            }catch (Exception e) {
                return "Error: Target number";
            }
            // target number times 4
            int bits = Integer.parseInt(this.target, 2);
            this.target = Integer.toBinaryString(bits<<2);

            // calculate the final address
            this.target = addBinary(startPoint, this.target);
            // if the final address is 32 bits
            if(this.target.length() == 32) {
                char[] targ = new char[26];
                int addressStart = 4; // remove first four bits
                int addressEnd = 30; // remove last two bits
                int i = 0;
                for(; addressStart < addressEnd; addressStart++) {
                    targ[i] = this.target.charAt(addressStart);
                    i++;
                }
                // set opcode and target number
                this.target = new String(targ);
                this.setOpcode(lookupTable.opcodes.get(operandIndex));

                this.setmMachinen(convertHex(this.getOpcode() +
this.getTarget()));
            }
            // if the final address is not 32 bits

```

```

        else
            return "The address is too big";
    }
    return this.getMachine();
}

String addBinary(String a, String b) {
    // Initialize result
    String result = "";

    // Initialize digit sum
    int s = 0;

    // Traverse both strings starting
    // from last characters
    int i = a.length() - 1, j = b.length() - 1;
    while (i >= 0 || j >= 0 || s == 1)
    {

        // Compute sum of last
        // digits and carry
        s += ((i >= 0)? a.charAt(i) - '0': 0);
        s += ((j >= 0)? b.charAt(j) - '0': 0);

        // If current digit sum is
        // 1 or 3, add 1 to result
        result = (char)(s % 2 + '0') + result;

        // Compute carry
        s /= 2;

        // Move to next digits
        i--; j--;
    }

    return result;
}

// set and get functions
public String getTarget() {
    return target;
}

public void setTarget(String target) {
    this.target = target;
}

}

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

```

```

public class LookupTable {

    String[] registersName = null;
    String[] registersValue = null;
    ArrayList<String> instructions = new ArrayList<String>() ;
    ArrayList<String> opcodes = new ArrayList<String>() ;
    ArrayList<String> types = new ArrayList<String>() ;
    ArrayList<String> functions = new ArrayList<String>() ;

    LookupTable(String lookupTablePath){

        // reading lookup table start point
        Scanner scanner = null;
        try {
            scanner = new Scanner(new File(lookupTablePath));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        // read for registers
        String I = scanner.nextLine();
        I=I.replaceAll("\\s+", ""); //removing spaces
        String delims = "[,]+";
        registersName = I.split(delims.trim()); //Lookup table's first row has
        register's names. It supposed to separated with commas.
        I = scanner.nextLine();
        I=I.replaceAll("\\s+", "");
        registersValue = I.split(delims); //Lookup table's second row has
        corresponding register's numbers. It supposed to separated with commas.
        // read for instructions
        while (scanner.hasNextLine()) {

            I = scanner.nextLine();
            delims = "[ ]+"; //splitting instructions into tokens
            that we took from source file according to space.
            String[] tokens = I.split(delims);

            instructions.add(tokens[0]);
            types.add(tokens[1]);
            opcodes.add(tokens[2]);
            functions.add(tokens[3]);
        }
        // reading lookup table end point
        scanner.close();
    }

}

import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

```

```

public class MipsToHex {

    MipsToHex(String lookupTablePath, String startPoint, String inputFile, String
outputFile) {
        //taking path of absolute directory
        String mainPath = System.getProperty("user.dir");

        lookupTablePath = mainPath + lookupTablePath;

        // create the window items for main menu
        JFrame fr = new JFrame("Mips To Hex");

        JButton Interactive = new JButton("Interactive");
        JButton Batch = new JButton("Batch");
        JButton exit = new JButton("Exit");
        Interactive.setBounds(200,100,95,30);
        Batch.setBounds(200,150,95,30);
        exit.setBounds(200,200,95,30);

        JLabel message = new JLabel();
        message.setFont(new Font("Arial", Font.PLAIN, 15));
        message.setSize(1000, 20);
        message.setLocation(50, 50);
        // created the window items

        // add all items on the window
        fr.add(Interactive);
        fr.add(Batch);
        fr.add(exit);
        fr.add(message);

        fr.setSize(500,500);
        fr.setLayout(null);
        fr.setVisible(true);
        // end GUI

        // read lookup table from .txt file and create it
        LookupTable lt = null;
        try {

            lt = new LookupTable(lookupTablePath);

        }catch (Exception e) {
            e.printStackTrace();
            message.setText("The lookup-table cannot be read");
        }

        LookupTable lookupTable = lt;

        // exit button function
        exit.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {

```

```

        System.exit(0);
    }
});

// interactive mode
Interactive.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        new Interactive(lookupTable, startPoint);
        message.setText("");
    }
});

// batch mode
Batch.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String inputSourcePath = mainPath + inputFile;
        String outputObjPath = mainPath + outputFile;
        String msg = new
Batch().createObjFile(lookupTable, inputSourcePath, outputObjPath, startPoint);
        message.setText(msg);
    }
});

}

}

import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class Interactive {

    Interactive(LookupTable lookupTable, String startPoint){

        // create the window items for interactive mode
        JFrame fr = new JFrame("Mips To Hex");

        JLabel title = new JLabel("Interactive Mode");

        title.setFont(new Font("Arial", Font.PLAIN, 30));
        title.setSize(290, 30);
        title.setLocation(100, 30);

        JLabel input = new JLabel("Enter your command");
        input.setFont(new Font("Arial", Font.PLAIN, 20));
        input.setSize(190, 20);
        input.setLocation(100, 100);
    }
}

```

```

JTextField tinput = new JTextField();
tinput.setFont(new Font("Arial", Font.PLAIN, 15));
tinput.setSize(190, 20);
tinput.setLocation(100, 150);

JButton convert = new JButton("Convert");
JButton exit = new JButton("Exit");
convert.setBounds(100,200,95,30);
exit.setBounds(200,200,95,30);

JLabel message = new JLabel();
message.setFont(new Font("Arial", Font.PLAIN, 15));
message.setSize(1000, 20);
message.setLocation(100, 250);
// created the window items

// add all items on the window
fr.add(title);
fr.add(input);
fr.add(tinput);
fr.add(convert);
fr.add(exit);
fr.add(message);
fr.setSize(400,500);
fr.setLayout(null);
fr.setVisible(true);
// end GUI

// exit button function
exit.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        fr.dispose();
    }
});

// interactive mode function
convert.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // clear the message
        message.setText("");
        // if the input is empty
        if ((tinput.getText()).isEmpty())
            message.setText("Type a proper command");

        else {
            // tokens[0] is operation and tokens[1] is the rest
of the command
            String[] tokens =
tinput.getText().trim().split("\\s",2); // remove first white-spaces and divide the
instruction into 2 tokens.
            if(tokens.length==2) {
                tokens[1] = tokens[1].replaceAll("\\s+", "");
            }
            // remove all white spaces from the rest of the command

```

```

        // find the operation from the lookup table
        int instructionIndex = -1;
        for(int i = 0;
i<lookupTable.instructions.size();i++)
        {
            if(tokens[0].equals(lookupTable.instructions.get(i)))
            {
                instructionIndex = i;
                break;
            }
        }

        // if the operand is found in the lookup
table
        if (instructionIndex != -1)
        {
            String[] rest = tokens[1].split(",");
            // take all words (no operand) one by one according to the commas
            // if the instruction type is 'R'
            if
            (lookupTable.types.get(instructionIndex).equals("R")) {
                String msg = new
Rtype().RtypeConverter(lookupTable, tokens[0], rest, instructionIndex); // tokens[0]
means operand
                message.setText(msg);
            }
            // if the instruction type is 'I'
            else if
            (lookupTable.types.get(instructionIndex).equals("I")) {
                String msg = new
Itype().ItypeConverter(lookupTable, tokens[0], rest, instructionIndex); // tokens[0]
means operand
                message.setText(msg);
            }
            // if the instruction type is 'J'
            else if
            (lookupTable.types.get(instructionIndex).equals("J")) {
                String msg = new
Jtype().JtypeInteractionConverter(lookupTable, tokens[0], rest, instructionIndex,
startPoint); // tokens[0] means operand
                message.setText(msg);
            }
            else {
                message.setText("The type is
unknown");
            }
        }

        // if the operand is not in the lookup table
        else
            message.setText("The operand is
unknown");
    }
    // if the user input has wrong syntax

```



```

        else
            message.setText("Syntax error");
    }
}

}); // interactive mode function end

}

}

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

public class Batch {
    ArrayList<String> labels = new ArrayList<String>();
    ArrayList<Integer> labelsLines = new ArrayList<Integer>();
    ArrayList<String> sourceCode = new ArrayList<String>();
    ArrayList<String> operand = new ArrayList<String>();
    ArrayList<String> restCommand = new ArrayList<String>();
    ArrayList<String> converted = new ArrayList<String>();

    public String createObjFile(LookupTable lookupTable, String inputSourcePath,
String outputObjPath, String startPoint) {
        // if there is any error for converting, flag return true
        boolean Error = false;

        readSourceFile(inputSourcePath);
        if(operand.size() == 0)
            return "The source file cannot be opened";
        else {
            // create an obj file if it is not created
            createObjFile(outputObjPath);

            for(int i = 0; i < operand.size(); i++) {

                // find the operand from the lookupTable
                int operandIndex = -1;
                for(int j = 0; j < lookupTable.instructions.size(); j++)
                {
                    if(operand.get(i).equals(lookupTable.instructions.get(j)))
                    {
                        operandIndex = j;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        // if the operand is found in the lookup table
        if (operandIndex != -1)
        {
            String[] rest = restCommand.get(i).split(",");
            // if the operand type is 'R'
            if (lookupTable.types.get(operandIndex).equals("R"))
            {
                converted.add(new
Rtype().RtypeConverter(lookupTable, operand.get(i), rest, operandIndex)); //
tokens[0] means operand
            }
            // if the operand type is 'I'
            else if
(lookupTable.types.get(operandIndex).equals("I")) {
                // if operand is branch, calculate the label
value
                if(operand.get(i).charAt(0) == 'b') {
                    String branchValue;
                    int branchindex = -1;
                    for(int j = 0; j<labels.size();j++)
                    {
                        if(rest[2].equals(labels.get(j))){
                            branchindex = j;
                            break;
                        }
                    }
                    if(branchindex > -1)
                    {
                        if ((labelsLines.get(branchindex) - i )
> 0)
                            branchValue =
Integer.toString((labelsLines.get(branchindex) - i - 1));
                        else
                            branchValue =
Integer.toString((labelsLines.get(branchindex) - i));
                    }
                    else
                        branchValue = "Error: Label Name";

                    rest[2] = branchValue;
                }
                converted.add(new
Itype().ItypeConverter(lookupTable, operand.get(i), rest, operandIndex)); //
tokens[0] means operand
            }
            // if the operand type is 'J'
            else if
(lookupTable.types.get(operandIndex).equals("J")) {
                // find label's address
                int labelindex = -1;
                for(int j = 0; j<labels.size();j++)
                {

```

```

        if(rest[0].equals(labels.get(j))){
            labelindex = j;
            break;
        }
    }

    if(labelindex > -1)
        rest[0] =
Integer.toString(labelsLines.get(labelindex));
    else
        rest[0] = "-";

    converted.add(new
Jtype().JtypeInteractionConverter(lookupTable, operand.get(i), rest, operandIndex,
startPoint)); // tokens[0] means operand
    }
    else {
        converted.add("The type is unknown");
    }
}
// if the operand is not in the lookup table
else
    converted.add("The operand is unknown");

// if there is any error for converted
if(converted.get(i).charAt(0) != '0')
    Error = true;
writeObjFile(outputObjPath);
}

}

if(Error)
    return "The object is created with some ERROR";
else
    return "The object is created successfully";
}
private void writeObjFile(String outputObjPath) {

    try {
        BufferedWriter myWriter = new BufferedWriter(new
FileWriter(outputObjPath));

        for(int i = 0; i< converted.size(); i++) {
            myWriter.write(converted.get(i));
            myWriter.newLine();
        }
        myWriter.flush();
        myWriter.close();
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
}

```

```

private void createObjFile(String outputObjPath) {
    try {
        File outputObj = new File(outputObjPath);
        outputObj.createNewFile();
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}

private void readSourceFile(String inputSourcePath) {
    int indexLabel = -1;
    Scanner scanner = null;
    // read source file
    try {
        scanner = new Scanner(new File(inputSourcePath));

        //First read all the lines for detecting label names and
        corresponding index.
        while (scanner.hasNextLine()) {
            indexLabel++;
            String line = scanner.nextLine().trim();
            boolean isLabel = false;
            int i = 1;
            for(; i < line.length(); i++) {
                if(line.charAt(line.length() - i) == ':') {
                    isLabel = true;
                    break;
                }
                else isLabel = false;
            }
            if(isLabel) {
                labels.add(line.substring(0, line.length()-i));
                sourceCode.add((line.substring(line.length()-
i+1, line.length())));
                labelsLines.add(indexLabel);
            }
            else
                sourceCode.add(line);
        }
        scanner.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    // create operand and rest in the command
    for (int i = 0; i < sourceCode.size(); i++) {
        sourceCode.set(i, sourceCode.get(i).trim());
        String[] code = sourceCode.get(i).split("\\s", 2);
        operand.add(code[0]);
        restCommand.add(code[1].replaceAll(" ", ""));
    }

    } // end of the reading
}

```

# REFERENCES

Patterson, D. A., & Hennessy, J. L. (2016). Computer Organization and Design ARM Edition: The Hardware Software Interface. Morgan Kaufmann.

Mips32, R. (2001). Architecture for Programmers Volume ii: The Mips32 R Instruction Set.