Final Assessment Report

Computer Architecture & Organization

CSE112 – Group 5

Shehab El Din Adel Nassef Attia – 18P3863
Daniel Tarek Lewis Saad Hanna El Shabrawy- 18P1185
Omar Khaled Ossman – 18P1498
Abdelrahman Mohamed Shemies- 18P9565
Zyad Ahmed Yakan- 18P9538

Team Roles

Shehab Adel	Designed the structure of the MIPS simulator program, wrote several instructions
Daniel Tarek	Designed the structure of the MIPS simulator program, wrote several instructions
Omar Ossman	Wrote several instructions, made the binary conversion part
Abdelrahman Shemis	Integration of the classes with each other, wrote several instructions
Zyad Yakan	Pipeline Stages PowerPoint elaboration, wrote several instructions, worked with Data Structures

Contents

Introduction

MIPS Simulator is a program written in High-Level Languages like Java, C++, Python...etc which takes Assembly Language (MIPS Instructions) and convert them into Binary Language and execute the instructions. In this report we will show each step conducted in our program to make MIPS Simulator. We will elaborate the architecture of the program, how data structure helped us in making the program, the problems we faced and how did we solve it.

Note: There are some steps that do not need to be documented here as they are explained in the code blocks.

Glossary Terms

MIPS Architecture: Microprocessor without Interlocked Pipeline Stages is an Instruction Set Architecture developed by MIPS Technologies. This architecture allows the execution of instructions provided by the user through the computer's processor.

High-Level Language: Computer Programming Languages that are written on the computer to execute or build several software programs, like Java, C++, C, C#, Python, Perl, and a lot of other known programming languages.

Low-Level Language: Computer Programming Languages that is the bridge between High Level Language and Machine Code.

Machine Language: It is written in zeros and ones, Binary Language; it is the code in which is executed by the computer processor.

Assembler: Converts the Low-Level Language to Machine Language.

Compiler: Converts High Level Language to Low Level Language.

Data Structure: A way to organize, manage and store data which enables better access to it and editing. Data Structures collect data together and illustrates the relationships between them.

Maps: A data structure which holds data in key and value concept. For example, the ID and Name relationship, where ID acts as a special key for each Name (value).

String: A data type which consists of a list of characters combined. For example, "Hello World".

Integer: A data type which holds numerical values.

Setters: Functions that are used to set and change the value of an attribute inside a class

Getters: Functions that are used to get the value of an attribute inside a class

Constructor: It is more like a function which makes an object from a class.

Immediate: A constant numerical value.

How to Operate the Program

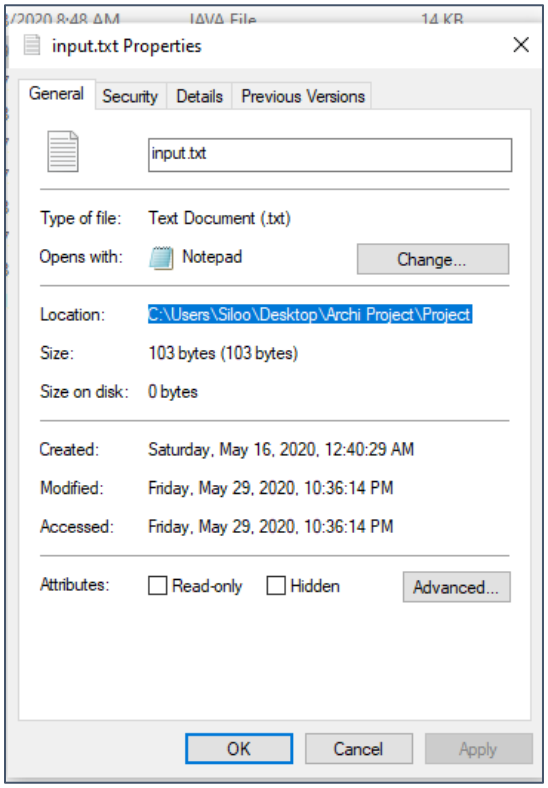
Note: It is better to use Visual Studio Code in running the program. Press File -> Open Folder -> Select the project's folder -> Press Select Folder

Step 1: Locate the input.txt file inside the project folder

Step 2: Right Click on it

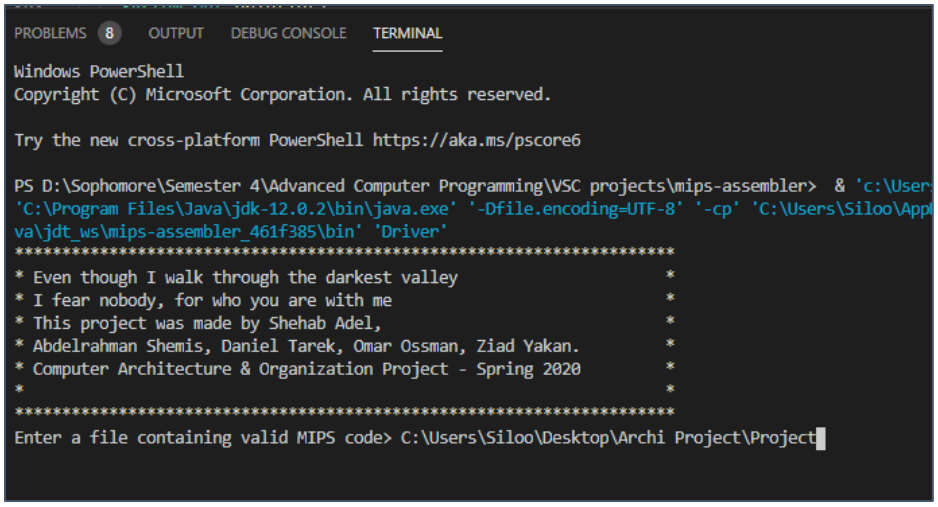
Step 3: Select "Properties"

Step 4: Copy the location's path

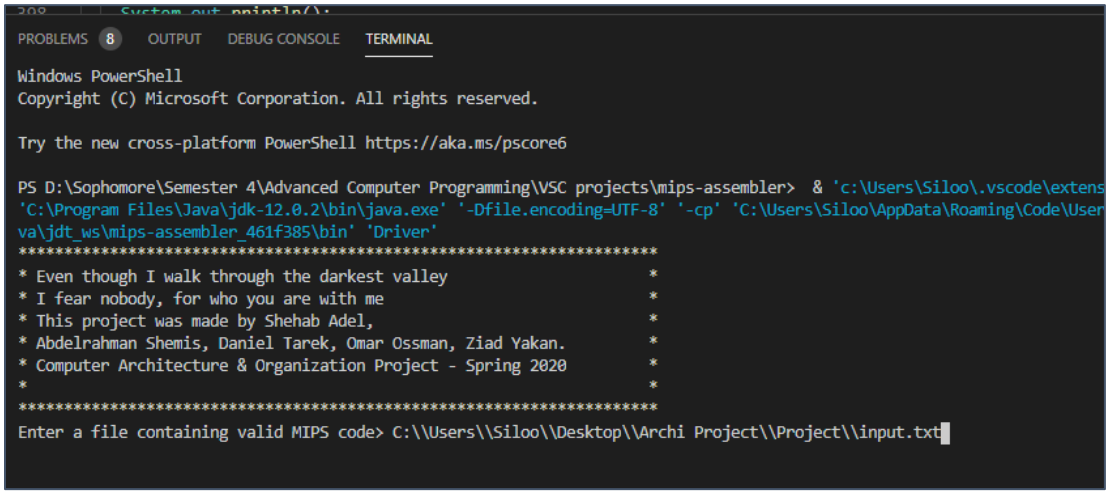


Step 5: Run the program

Step 6: Paste the location's path in the console



Step 7: add an additional backslash (\) at each backslash, and “input.txt ” at the end of the path, so it will be like that



Step 8: Press Enter

Program's Architecture

The program follows the Object-Oriented Architecture Style; this style organizes the program's code into several classes, where each class contains its attributes and functions. This allow reusability, organization, and security of data. In this program, we have five classes.

- Driver
- InstructionSet
- Instruction
- Registers
- AddressMap

Each of the above classes has a specific purpose where each holds data members that are used in building the assembler and the simulator.

Register Class

Starting with the Registers Class; it contains a single attribute which is a Map (registerMap) that associates the Registers names with their specific address. This is operated by invoking a function, where whenever the program executes, the function is called, and the Register map is initialized. registerMap contains a key of type String, and value of type Integer. We have an example of how it associates the register name with its address value.

```
        //Initiates the Register map and sets up the list.

    public static void init(){
       registerMap = new HashMap<>();

        registerMap.put("", 0);
        registerMap.put("$zero", 0);
        registerMap.put("$at", 1);
        registerMap.put("$v0", 2);
        registerMap.put("$v1", 3);
        registerMap.put("$a0", 4);
        registerMap.put("$a1", 5);
        registerMap.put("$a2", 6);
        registerMap.put("$a3", 7);
        registerMap.put("$t0", 8);
        registerMap.put("$t1", 9);
        registerMap.put("$t2", 10);
        registerMap.put("$t3", 11);
        registerMap.put("$t4", 12);
        registerMap.put("$t5", 13);
        registerMap.put("$t6", 14);
        registerMap.put("$t7", 15);
        registerMap.put("$s0", 16);
        registerMap.put("$s1", 17);
```

```
        registerMap.put("$s2", 18);
        registerMap.put("$s3", 19);
        registerMap.put("$s4", 20);
        registerMap.put("$s5", 21);
        registerMap.put("$s6", 22);
        registerMap.put("$s7", 23);
        registerMap.put("$t8", 24);
        registerMap.put("$t9", 25);
        registerMap.put("$k0", 26);
        registerMap.put("$k1", 27);
        registerMap.put("$gp", 28);
        registerMap.put("$sp", 29);
        registerMap.put("$fp", 30);
        registerMap.put("$ra", 31);
```

“put” is a method or function that inserts data to the map, as we see here when we call the function init(), we setup 32 registers providing their names and associated addresses.

The Register class contains two methods, get() and setValue()

- get()
  get() method allows us to get the corresponding value of a specific key, for example we say get(“$s2”); it will return 18 as we have in the block of code above. It is one of the most important methods, as we are using it a lot.
- setValue()
  setValue() method allows us to set a value for a specific key, in this method we must enter the name of the address (the key) in order to update or set it to have a new value, for example setValue(“$s2”, 2); now $s2 will have a value of 2 instead of 18.

AddressMap Class

We have a clone of Register class, but it is called AddressMap. It is the same as Register class which it contains an attribute of type Map and an initialization method, the only difference is that the Map holds two types of Integer and Integer, where it associates each address value from the Register map to have its own value inside.

So far if we have to illustrate the relationship between the two classes it will be like that, where the Register Map contains Address as a Value, but in the Address Map; it is considered as a Key.

So, in conclusion, each register has an address, inside each address there is a stored initial value which represents the register’s stored value.

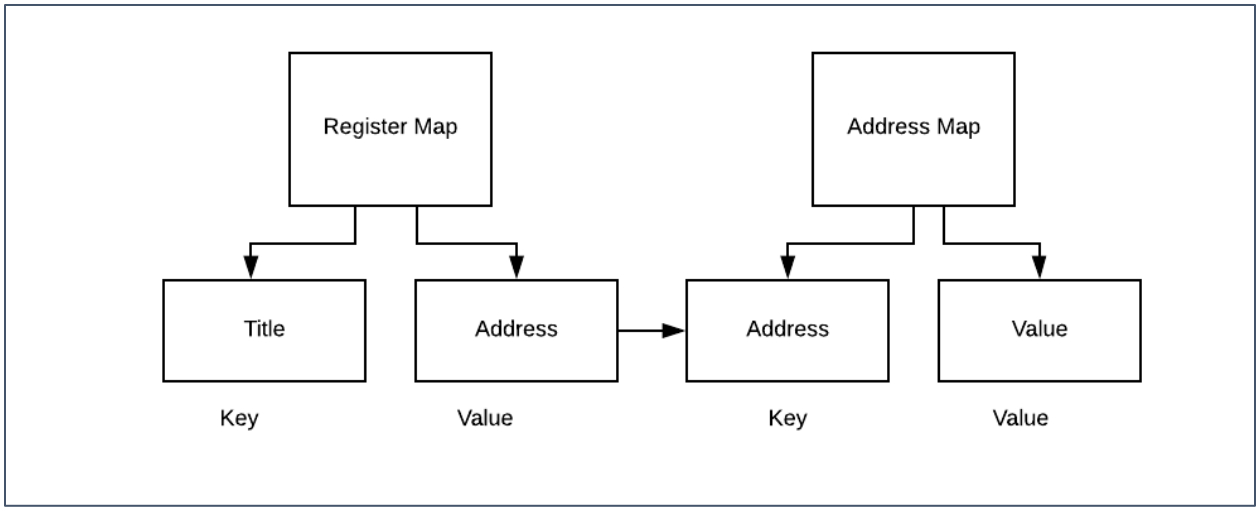


Figure 1 Relationship between Register and AddressMap

Instruction Class

In MIPS Instructions, each instruction must follow specific format to distinguish between it and any other formats. There are several factors that are used in formatting the instruction, the table below shows how instructions are formatted.

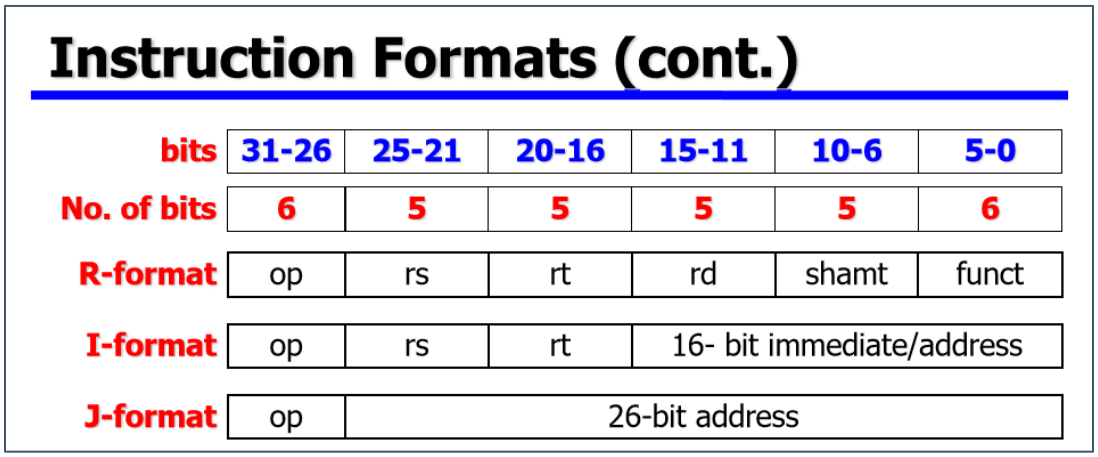


Figure 2 Instructions Format

We collected these factors and made them as attributes, in addition to some other attributes which will be explained later

```
class Instruction{
    //Set up data that the instruction should store.
    private int address, jumpAddress, immediate;
    private int shamt=0;
    private String instruction, source, target, destination, type, label;
```

Each instruction must contain a name (instruction attribute), type, and an opcode which is passed from another class InstructionStruct inner class. We have a constructor which takes the current virtual address and associate it with the instruction we currently working on. There are other several constructors.

```
  Instruction(int address){
        this.address = address;

    }
```

Then there are some Setters and Getters which set and get values of our attributes in the Instruction Class.

```
    public String getInstruction(Instruction ins)
    {
        return ins.instruction;
    }
    public String getSource(Instruction ins)
    {
        return ins.source;
    }
    public String getTarget(Instruction ins)
    {
        return ins.target;
    }
    public String getDest(Instruction ins)
    {
        return ins.destination;
    }
    public String getType(Instruction ins)
    {
        return ins.type;
    }
```

We have three methods that initialize the instruction based on its type as seen in Figure 2.

```
    public void setJType(String instruction, int jump){
        this.instruction = instruction;
        this.jumpAddress = jump;
        type = "J";
    }
```

```
//Sets up a r-type instruction.
    public void setRType(String instruction, String source, String target, String
 destination){
        this.instruction = instruction;
        this.destination = destination;
        this.source = source;
        this.target = target;
        type = "R";
    }
    public void setRType(String instruction, String source, String target, String
 destination, String shamt){
        this.instruction = instruction;
        this.destination = destination;
        this.source = source;
        this.target = target;
        this.shamt = Integer.parseInt(shamt);
        type = "R";


    }
```

The difference between the first setRType method and the second one, is they have different parameters where the second one receives shamt which is the Shift Amount. We created two methods in this because there are some instructions which does not have Shift Amount, that is why we initialized the Shift Amount equal zero.

Moving to the setIType method; it contains several if conditions to follow different paths if these conditions presented.

```
    public void setIType(String instruction, String source, String target, String
 immediate){
        //Initialize variables.
        if(instruction.equals("lui"))
        {
            setLui(instruction, target, immediate);
        }
        else if(instruction.equals("beq")||instruction.equals("bne")){
            this.instruction = instruction;
            this.source = source;
            this.target = target;
            this.immediate= Integer.parseInt(immediate);
        }
        else{
        this.instruction = instruction;
        this.source = source;
        this.target = target;
        try{
```

```
            this.immediate = Integer.parseInt(immediate);
        }catch(NumberFormatException e)
        {   System.out.println("Exception caught, by default the immediate = zero
");
            this.immediate = 0;
        }
        }
        type = "I";
    }
```

If the instruction is lui, it will only need the instruction, target, and immediate parameters. Else if, it was beq or bne instruction, it will need instruction, source, target, and immediate parameters, where immediate is converted to Integer from String type. The default case to accept instruction, source, target, and immediate. In case the immediate cannot be read, it will automatically be set to zero.

Then we have one of the most important methods in our project which is toBinary(). toBinary() method takes a parameter of type Instruction, it creates first an empty String temp, which will hold the binary value representation of the instruction.

```
    public String toBinary(Instruction ins){
        String temp="";
        String op=Integer.toBinaryString(InstructionSet.getOpCode(instruction));
        String opLead=String.format("%6s",op).replace(' ', '0');
        temp=opLead;
```

op attribute holds the string representation of the binary value of the instruction's opcode. This line `InstructionSet.getOpCode(instruction)` gets the opcode value of the instruction we provide, as we saved our instruction set in a Class called InstructionSet, which will be explained later. As we receive the opcode of the instruction, we put it into an attribute called opLead, which will hold the opcode in a special format which is six bits binary number, and then the temp holds the opLead string value.

```
        if(type=="R")
        {
       String rs= Integer.toBinaryString(AddressMap.get(Registers.get(source)));
//converting rs into binary
        String rsLead= String.format("%5s",rs).replace(' ', '0'); //formating it
into 5 bits
            temp=temp+rsLead;                                          //appending
 it to temp
```

In case the instruction type is R, we create a string attribute that will hold the value inside the register's rs address, which is converted into binary. (%5s) formats the string to be 5 bits string binary with replacing space with zeros and put inside rsLead. Finally, adding temp with rsLead. The illustration below will show how this operation works.

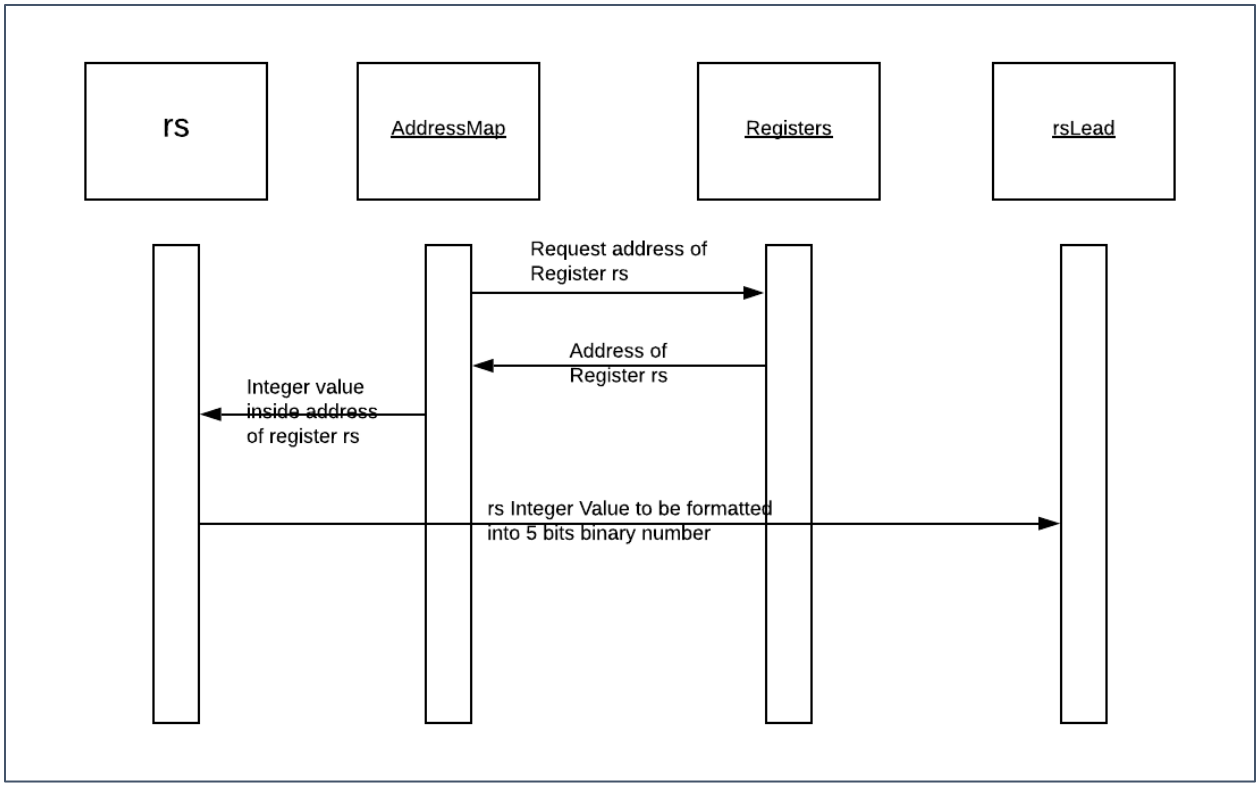


Figure 3 Setting the rs value to binary

And So on goes for the rt and rd registers, while for the Shift Amount and Function, they are directly associated inside the instruction, so there is no need to call for the AddressMap and Registers since that they are also not associated with Shift Amount and Function.

```
            String rt= Integer.toBinaryString(AddressMap.get(Registers.get(target
)));
            String rtLead= String.format("%5s",rt).replace(' ', '0');
            temp=temp+rtLead;
            String rd= Integer.toBinaryString(AddressMap.get(Registers.get(destin
ation)));
            String rdLead= String.format("%5s",rd).replace(' ', '0');
            temp=temp+rdLead;
            String shmt= Integer.toBinaryString(getShamt(ins));
            String shmtLead= String.format("%5s",shmt).replace(' ', '0');
            temp=temp+shmtLead;
            String func=Integer.toBinaryString(InstructionSet.getFunct(instructio
n));
            String funcLead=String.format("%6s",func).replace(' ','0');
            temp=temp+funcLead;
```

Same procedural goes for the I and J Type, but the parameters change. In the end the temp is returned holding the 32 bits binary representation of the instruction.

InstructionSet Class

This class has an attribute of type Map which has String as a key, and InstructionStruct as a value. The key here represents the Instruction title, while the value is a constructor which for InstructionStruct Class that several attributes of instruction's title, instruction's type, instruction's opcode, and instruction's function. That is why as mentioned before, we get the instruction's opcode from the InstructionSet class.

```
  //Set up all instructions into HashMap for lookup. Numeric values are decimal.
        instructionMap.put("add", new InstructionStruct("add", "R", 0, 32));
        instructionMap.put("sub", new InstructionStruct("sub", "R", 0, 34));
        instructionMap.put("addi", new InstructionStruct("addi", "I", 8, -2));
        instructionMap.put("lw", new InstructionStruct("lw", "I", 35, -2));
        instructionMap.put("sw", new InstructionStruct("sw", "I", 43, -2));
        instructionMap.put("slt", new InstructionStruct("slt", "R", 0, 42));
        instructionMap.put("slti", new InstructionStruct("slti", "I", 10, -2));
        instructionMap.put("beq", new InstructionStruct("beq", "I", 4, -2));
        instructionMap.put("bne", new InstructionStruct("bne", "I", 5, -2));
        instructionMap.put("j", new InstructionStruct("j", "J", 2, -2));
        instructionMap.put("andi", new InstructionStruct("andi", "I", 12, -2));
        instructionMap.put("and", new InstructionStruct("and", "R", 0, 36));
        instructionMap.put("or", new InstructionStruct("or", "R", 0, 37));
        instructionMap.put("ori", new InstructionStruct("ori", "I", 13, -2));
        instructionMap.put("nor", new InstructionStruct("nor", "R", 0, 39));
        instructionMap.put("sll", new InstructionStruct("sll", "R", 0, 0));
        instructionMap.put("srl", new InstructionStruct("srl", "R", 0, 2));
        instructionMap.put("sra", new InstructionStruct("sra", "R", 0, 3));
        instructionMap.put("lui", new InstructionStruct("lui", "I", 15, -2));
```

```
    //Acts as a struct to keep the instruction data togehter.
    private class InstructionStruct{
        public String instruction;
        public String type;
        public int opCode;
        public int function;
```

These are the instructions which our program can execute and assemble.

Moving to the core of the execution, we have two methods that are responsible for our execution. executeR() and executeI(). Each of them is responsible for the execution of specific type of instructions. Same procedural goes for both, where there is a switch case that executes on the instruction's title.

```
switch (ins.getInstruction(ins))
```

executeR() Method

In case of R type, we have nine cases that each represent a different instruction title. Starting with Shift Right Arithmetic, its logic is to move all bits in a word to right by n bits, filling the emptied bits with sign bit, which is equivalent to dividing by 2^n.

The syntax is rd = rt >> sh

First thing is getting the rt value through the AddressMap and RegisterMap as explained before, then convert it to 32 bits binary number in String format. Then, creating a variable that will hold the shift amount value provided from the instruction. Lastly, we create a variable that will hold the rd value following the syntax of the sra instruction which is rt >> shamt.

The procedural now is setting new value for the rd Register inside the AddressMap. Then printing out the rt value and its binary representation, shift amount, and the new rd value.

```
Integer rtValueSra=AddressMap.get(Registers.get(ins.getTarget(ins)));
String rtValueSraBinLead=String.format("%32s",Integer.toBinaryString(rtValueSra))
.replace(' ','0');

            Integer ShmtValueSra= Instruction.getShamt(ins);
            Integer rdValueSra = rtValueSra >>  ShmtValueSra;
            //Creating Binary String representation of rdValueSra
  String rdValueSraLead= String.format("%32s",Integer.toBinaryString(rdValueSra))
.replace(' ', '0');

            AddressMap.setValue(Registers.get(ins.getDest(ins)), rdValueSra);

         System.out.println("**************Execution**************\nLine Number:"
+(lineNo+1)+"\n"+
                ins.getTarget(ins)+" = "+rtValueSra+" And it's binary value is "+
rtValueSraBinLead+"\n"+
            "The Shift Amount = "+ Instruction.getShamt(ins) + "\n "
                    +
  The new value of register "+ins.getDest(ins)+" = "+AddressMap.get(Registers.get
(ins.getDest(ins)))
            +"\nIts binary value = "+rdValueSraLead
            );
```

The same procedural goes for srl and sll instructions, but they have different syntax as shown below.

sll: rd = rt << shamt

srl: rd = rt >>> shamt

Moving to the nor instruction where it is a logical bit-to-bit comparison; each rs bit is compared to the corresponding rt bit. The output is presented following this table of NOR outputs.

A	B	A **NOR** B
0	0	1
0	1	0
1	0	0
1	1	0

Figure 4 Nor Truth Table

```
            Integer rsValueNor= AddressMap.get(Registers.get(ins.getSource(ins)))
;//Holds the stored value in the register's address
            Integer rtValueNor= AddressMap.get(Registers.get(ins.getTarget(ins)))
;



            String rsBinNor = Integer.toBinaryString(rsValueNor); //String of Bin
ary for rs converted from the value of the register
            String rtBinNor= Integer.toBinaryString(rtValueNor);  //String of Bin
ary for rt
            String rsBinLeadNor=String.format("%32s",rsBinNor).replace(' ', '0');
 //formatting the rsBin into 32 bits binary
            String rtBinLeadNor=String.format("%32s",rtBinNor).replace(' ', '0');
 //formatting the rtBin into 32 bits binary
            BigInteger rsBigNor= new BigInteger(rsBinLeadNor,2);
            BigInteger rtBigNor= new BigInteger(rtBinLeadNor,2);
            BigInteger rdBigNor;
            String rdNor="";//the new rd string of binary characters
            rdNor=String.format("%32s",((rsBigNor.or(rtBigNor)) ).toString(2)).re
place(' ', '0');
            rdBigNor= new BigInteger(rdNor,2);
            for(int i=0;i<rdNor.length();i++)
            {
                rdBigNor=rdBigNor.flipBit(i); //flipping every single bit in the
Or result
            }
            String rdBinLeadNor =rdBigNor.toString(2); //converting flipped bits
to String in binary
            String rdBigNorLead="";
```

```
            rdBigNorLead= String.format("%32s",rdBinLeadNor).replace(' ', '0');
//in order to make sure it's staying 32 bits

            //Avoiding NumberFormatException which occures when we try to input
a decimal value
            //bigger than the max value of Integer
            try{
               //adding the new decimal value to the address of register rd
           AddressMap.setValue(Registers.get(ins.getDest(ins)), Integer.parseInt
(rdBigNorLead,2)); //adding decimal value converted from binary
            }catch(NumberFormatException e)
            {
                System.out.println("The decimal value for the instruction  "+ins
.getInstruction(ins) +" is so big to be an integer, it won't be added to the regi
ster.");
            }
```

We used a Big Integer class because it is the only type that can hold a 32 bits integer combined. In order to perform Nor logical instruction, there was no immediate function that would perform this logical operation, but there is a function that performs OR, which is the flipped version of NOR. So, we invoke OR method on rsBigNor and rtBigNor, and the result is put in rdBigNor. Then, we make a look where we flip every single bit in the rdBigNor. Finally, we set this new value in the rd register, and put it in a string for printing out to the user.

Same procedural goes for the AND, OR, ORI, and ANDI where they follow the table below in bit-to-bit comparisons

AND Truth Table

Inputs		Output
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

Figure 5AND and OR Truth Tables

The only difference between AND ANDI, and OR and ORI is that ANDI and ORI compare the value of register rt with binary representation of an immediate number.

Then moving to the add and sub instructions, which they add or subtract the rs and rt registers' values together and put the result inside the rd register. There is also an I-type instruction called addI with adds value inside register rt with an immediate number.

Finally, in the R type. The slt instruction; it sets the value inside rd to one if value of rs is less than value of rt. Else, set the value inside rd register to zero. There is also slti instruction in I-type, where the comparison between value of register rs and an immediate.

```
                //if rs<rt set value of 1 to rd, if not set value 0
                Integer slt= (AddressMap.get(Registers.get(ins.getSource(ins))) <
AddressMap.get(Registers.get(ins.getTarget(ins)))
                ? //unary operator
                1:0
                );
                //updating the value of rd
                AddressMap.setValue(Registers.get(ins.getDest(ins)), slt);
```

executeI() Method

Beq instruction compares value of register rs with value of register rt, if they are equal then head to the labeled instruction, else continue in the sequence of code. Bne is the same in comparison, but if they are not equal then head to the labeled instruction.

Beq and Bne instructions are executed in the Driver's class. Here, we just print if they are equal or not for the two instructions.

Lui instruction set the lower 16 bits of a register to zeros. The rs register is not used in this instruction, so it is equal to zero, but to operate the instruction, you must set the rs with zero manually in the instruction input file.

```
             Integer ImmValueLui= ins.getImmediate(ins);

            Integer rtValueLui= ImmValueLui << 16;
            String rtValueLuiBinLead=String.format("%32s",Integer.toBinaryString(
rtValueLui)).replace(' ','0');


            AddressMap.setValue(Registers.get(ins.getTarget(ins)), rtValueLui);
```

Driver Class

This is the main class, where the program runs from this class. We initialize some Maps and attributes which aid us in the running process.

```
    ArrayList<String> instructionsList = new ArrayList<>(); //Holds instructions
inputted from the input file(inputLine variable)
    LinkedHashMap<String, Integer> symbolTable = new LinkedHashMap<>(); //holds l
abels and their addresses
    LinkedHashMap<String, Integer> labelMap= new LinkedHashMap<>(); //holds label
s and their line numbers
```

```
    LinkedHashMap<Integer, String> addressMachine = new LinkedHashMap<>(); //hold
s addresses and machine code of the instructions inside

    String
    filename,
    inputLine, //holds string representation of every line of instructions in a w
hile loop
currentInstruction;

    String [] instructionArray;
    Integer address,ndx,labelNdx,labelLine=1;
    Boolean fileFound;
    //initiliazing the InstructionSet class for processing which contains
    //Instructions intializing, it contains Instruction types and associated data
 attributes
    InstructionSet mipsInstructionSet;
    //used for intializing the types of instructions
    Instruction mipsInstruction;
    Registers;
    FileReader;
    BufferedReader;
    AddressMap;

    fileFound = false;
    mipsInstructionSet = new InstructionSet();
    registers = new Registers();

    mipsInstructionSet.init();
    Registers.init();
```

fileFound initialized with false until the input file is found through the bufferedReader. We firstly enter a while loop which gets the file path through the filename we showed how to provide. Then, we put every single line inside a variable called inputLine which we later also add to the instructionsList. Since that, the inputLine is not empty, we start formatting the instructions provided.

First, we find out if contains a comment or not, so it will not be executed; we make the inputLine hold the start of the line till the # sign. Secondly, we determine if the line is empty in the first place, if true we do not add it to the instructionList. Then, we determine if the line contains a label, if true we put the label and associated address to symbolTable, and label and labelLine to the labelMap. Finally, we add the inputLine to the instructionsList and incrementing the address, and labelLine.

We go through a for loop which formats the instruction from the input line. It checks if the instructionList of current index contains “Exit:” label, it just adds the address next to the instruction in the instructionList for later printing, then break.

-Some block of codes is documented already in the Source Code, so there is no need to explain it here-

A String variable of currentInstruction holds the string representation of the instruction from instructionList of current index, to work on every instruction separately. We separate it from the labels in the start, then trim it, and finally split then add it into an instructionArray which holds the elements of the instruction separately. We make a switch case based on the instruction type through the first string which is instructionArray[0].

I Type Execution.

In case type I and it is beq, then we first calculate the offset which is equals the lines between the instruction and its label, by getting the label line from the labelMap and removing the ndx value and one.

Then we setup the instruction in mipsInstruction variable through setIType() method and the parameters are the instructionArray indicies and execute it by passing the created mipsInstruction and index number for printing out the instruction line.

In case the rs register value equals the rt register value, the new for loop's index will equal the index of the instruction's label brought from the AddressMap, else it will continue with no change.

```
ndx=labelMap.get(instructionArray[3])-1;
```

In the normal cases of other instructions, this is the default execution.

```
          mipsInstruction.setIType(instructionArray[0], instructionArray[2],
          instructionArray[1], instructionArray[3]);
          InstructionSet.exceuteI(mipsInstruction, ndx);
```

J Type Execution

In case of J-Type execution, which is the jump instruction only, we will calculate the offset from the label in the second index of the instructionArray, then get the label line corresponding to that label.

```
          //offsetJ represents the lines between the instruction and its correspo
nding label
          //labelMap returns line number of the label
          Integer offsetJ = ( labelMap.get(instructionArray[1]) - ndx - 1 );

          mipsInstruction.setJType(instructionArray[0], offsetJ);
          System.out.println("**************Execution**************"+ "\n"+"Line
Number:"+(ndx+1) + " The next line is "+ (labelMap.get(instructionArray[1])+1)+"\
n"+"*****************************\n" );
          //new line is the label line directly.
          ndx=labelMap.get(instructionArray[1])-1;
          break outerSwitch;
```

R Type Execution

In case we found srl, sll, or sra instructions, we follow this block of code. We set the rs register with zero, since it is not used.

```
        //Incase of SRL or SLL or SRA, Setup Instruction as Follow Instruction,
RD, RT, Shmt
            mipsInstruction.setRType(instructionArray[0], "0", instructionArray[2
], instructionArray[1],instructionArray[3]);
            InstructionSet.exceuteR(mipsInstruction,ndx);
```

The default execution is as following – you have to check the code to figure out the order of registers input here-

```
            mipsInstruction.setRType(instructionArray[0], instructionArray[2],
              instructionArray[3], instructionArray[1]);
              //Execution of the instruction
              InstructionSet.exceuteR(mipsInstruction,ndx);
              break;
```

Following that, we move to printing out the instruction and its corresponding machine code.

```
       System.out.println("Machine Code of the Instruction Of Line: "+(ndx+1)+"\t
"+mipsInstruction.toBinary(mipsInstruction));

       addressMachine.put(Integer.parseInt(String.valueOf(address)), mipsInstruct
ion.toBinary(mipsInstruction));
       instructionsList.set(ndx, instructionsList.get(ndx) + String.format("\t%08
x\t%s",address,mipsInstruction.toBinary(mipsInstruction)));
```

addressMachine map takes the integer value of the instruction's address as a key, and the string binary representation of the instruction. Then, in the current instruction in instructionList, we add the associated address and string binary representation for the instruction so it will as follow.

At the start of the loop the instruction was like that

Instruction

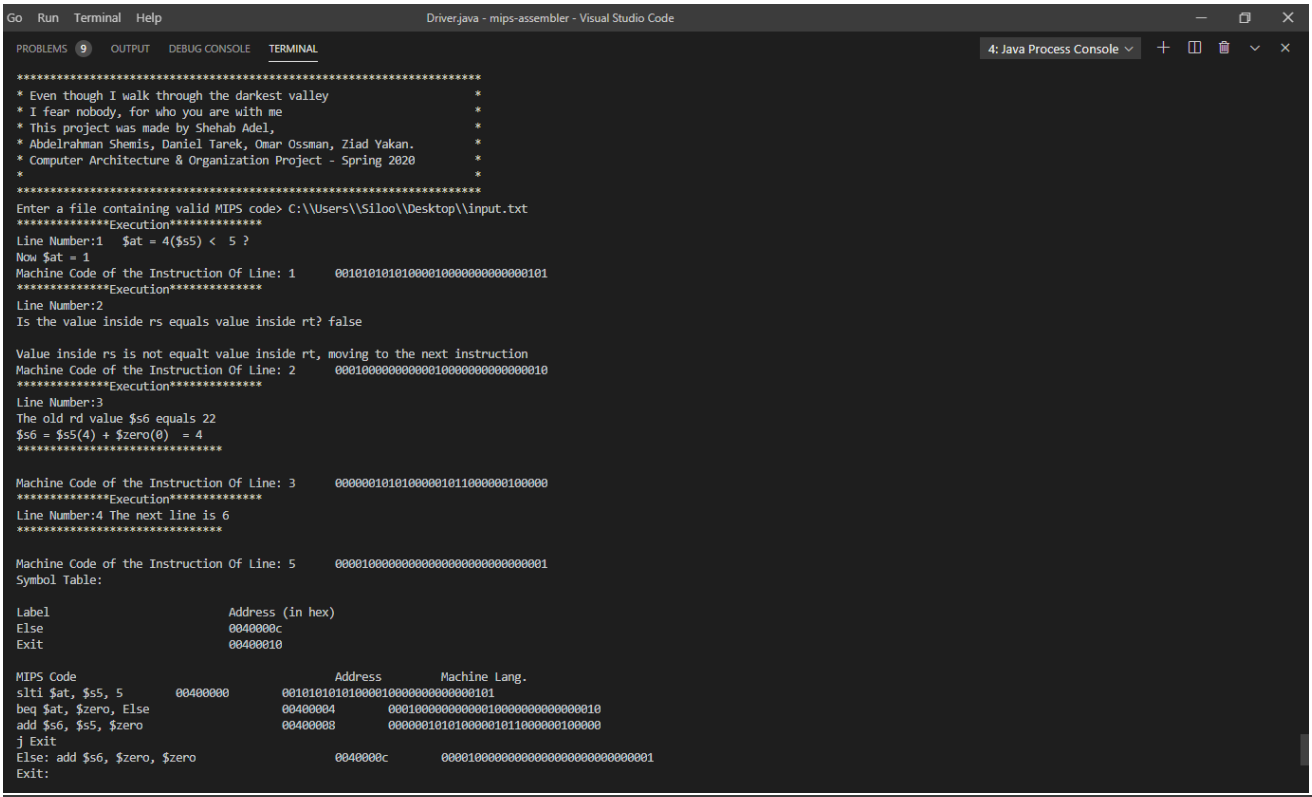
Then now it will be like that

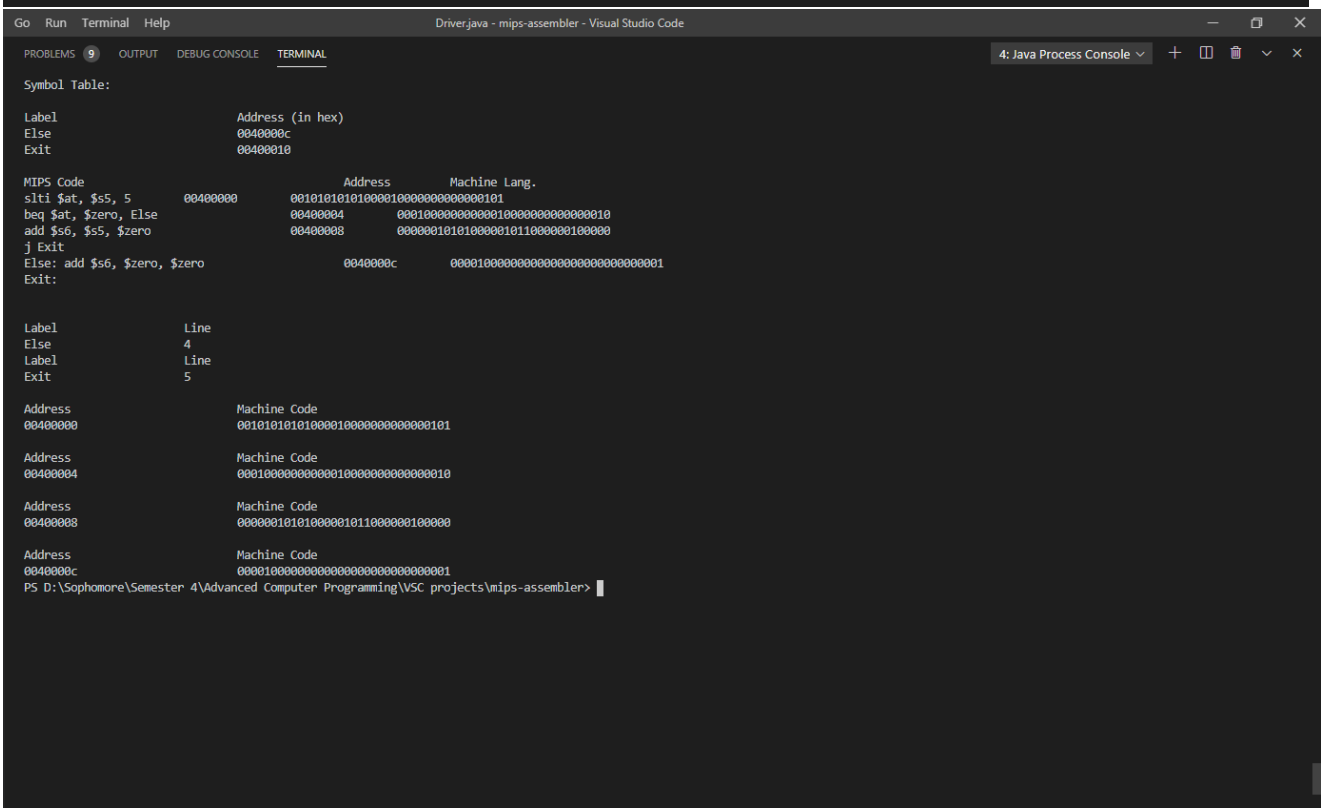
Instruction **Address** **Machine Language**

The loop continues until all the instructions in instructionList are executed.

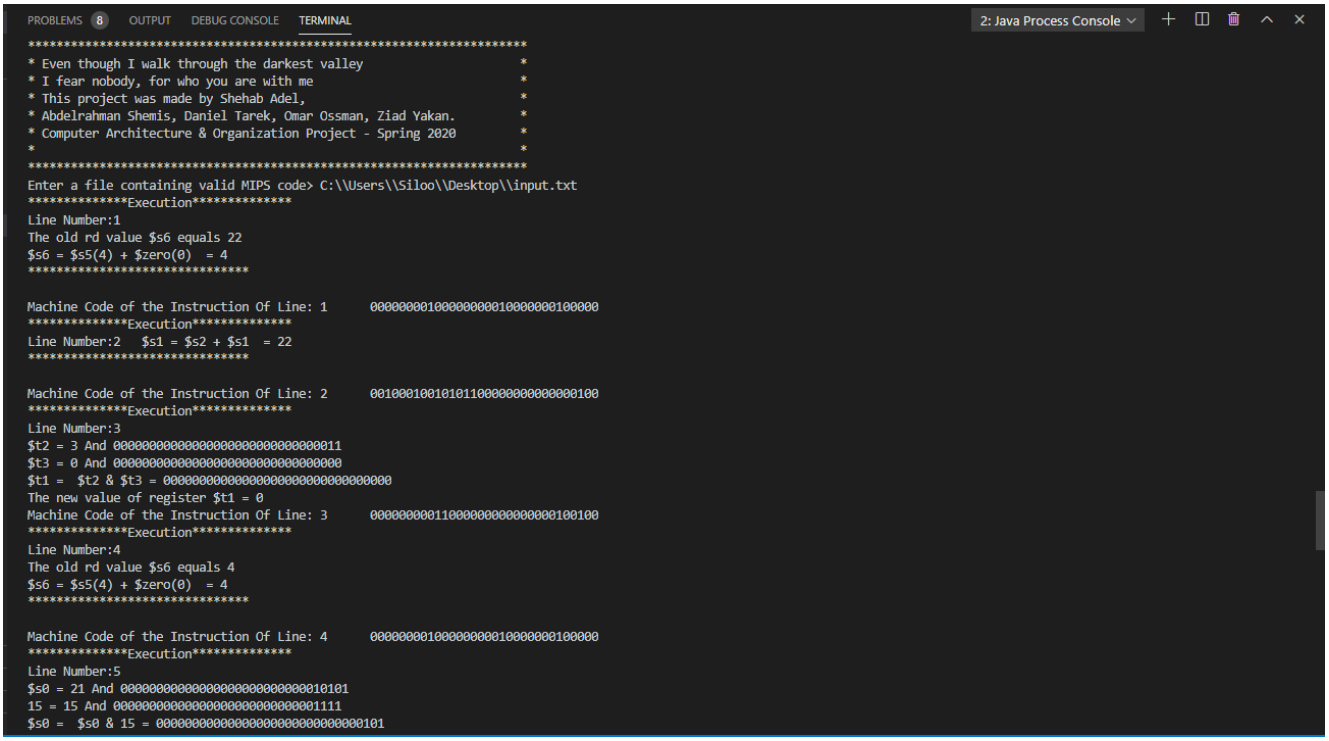
Printing

We start first by printing saved labels from symbolTable and its corresponding address, then print every single instruction from instructionsList. Furthermore, we print each label and its corresponding line from the labelMap. Finally, we print each address and the machine code inside it.

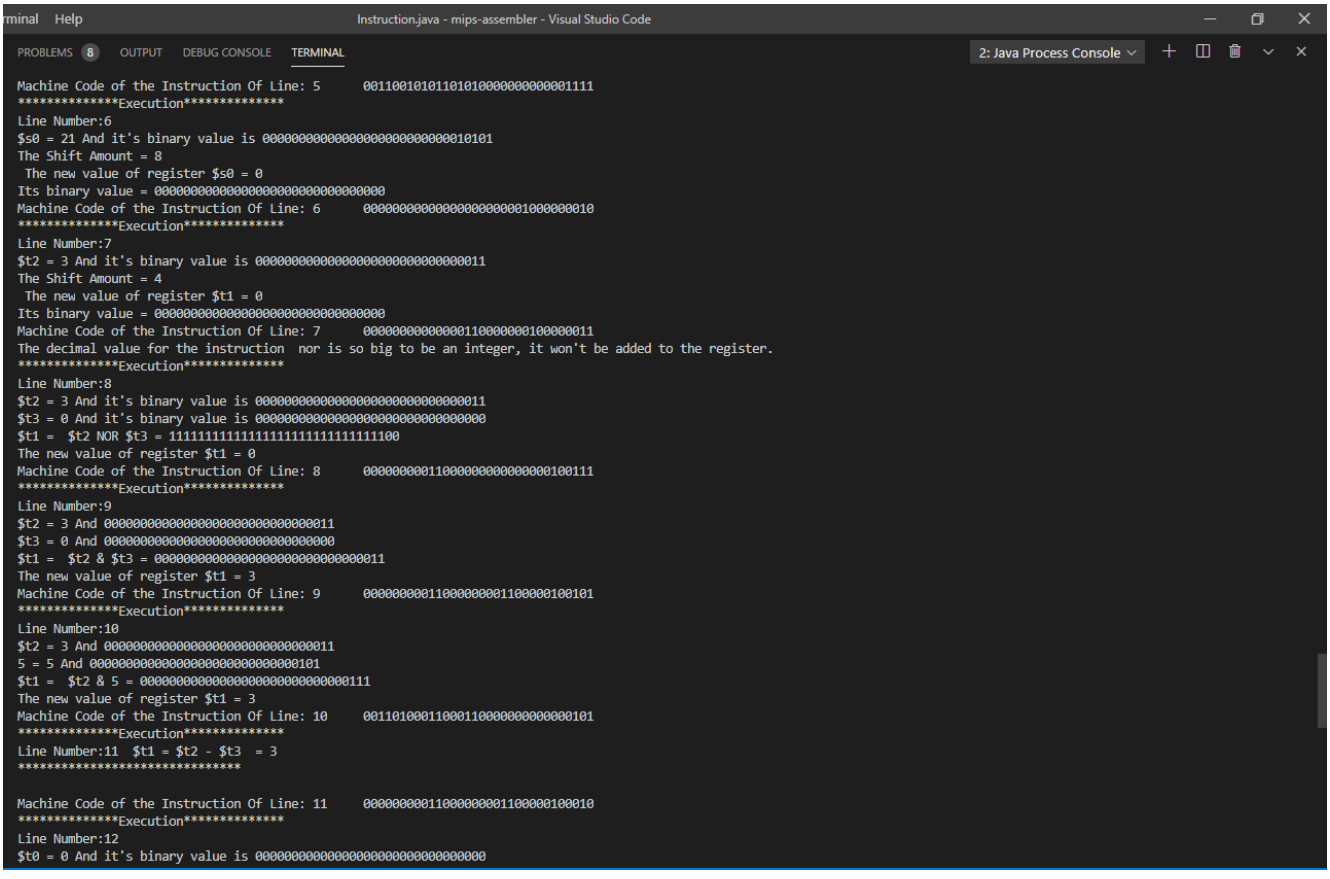




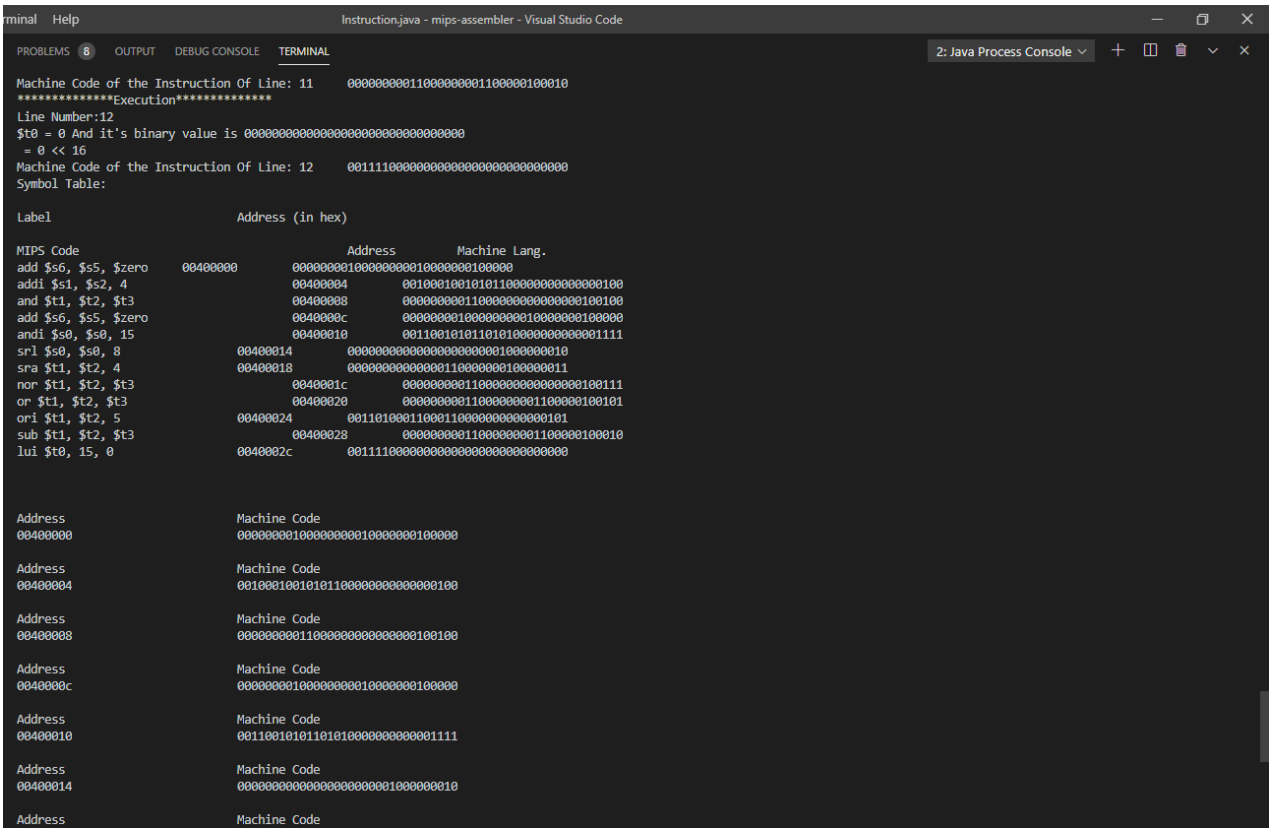
Another Test Cases

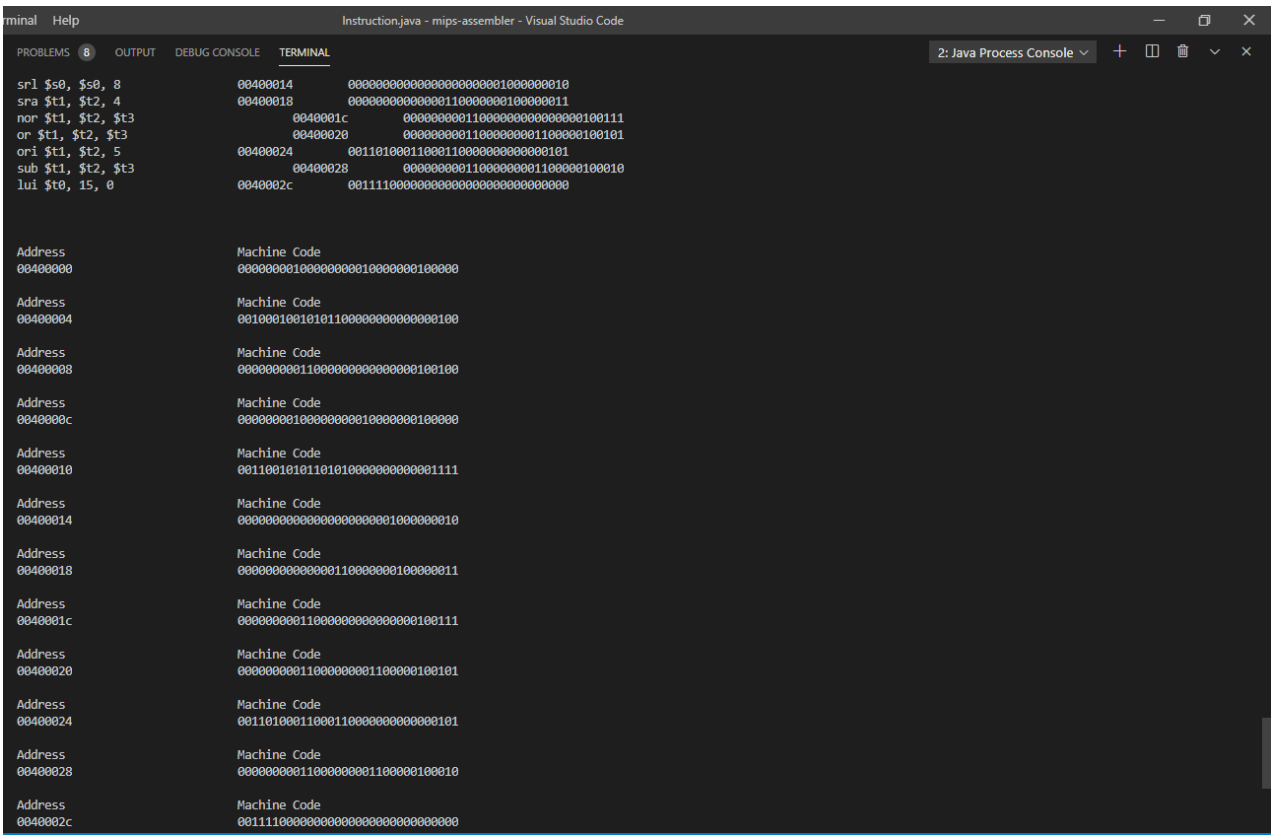


```
PROBLEMS 8  OUTPUT  DEBUG CONSOLE  TERMINAL                    2: Java Process Console
**********************************************************************
* Even though I walk through the darkest valley                      *
* I fear nobody, for who you are with me                             *
* This project was made by Shehab Adel,                              *
* Abdelrahman Shemis, Daniel Tarek, Omar Ossman, Ziad Yakan.         *
* Computer Architecture & Organization Project - Spring 2020         *
*                                                                    *
**********************************************************************
Enter a file containing valid MIPS code> C:\\Users\\Siloo\\Desktop\\input.txt
**************Execution**************
Line Number:1
The old rd value $s6 equals 22
$s6 = $s5(4) + $zero(0)  = 4
*******************************

Machine Code of the Instruction Of Line: 1      00000000010000000001000000100000
**************Execution**************
Line Number:2   $s1 = $s2 + $s1  = 22
*******************************

Machine Code of the Instruction Of Line: 2      00100010010101100000000000000100
**************Execution**************
Line Number:3
$t2 = 3 And 00000000000000000000000000000011
$t3 = 0 And 00000000000000000000000000000000
$t1 =  $t2 & $t3 = 00000000000000000000000000000000
The new value of register $t1 = 0
Machine Code of the Instruction Of Line: 3      00000000011000000000000000100100
**************Execution**************
Line Number:4
The old rd value $s6 equals 4
$s6 = $s5(4) + $zero(0)  = 4
*******************************

Machine Code of the Instruction Of Line: 4      00000000010000000001000000100000
**************Execution**************
Line Number:5
$s0 = 21 And 00000000000000000000000000010101
15 = 15 And 00000000000000000000000000001111
$s0 =  $s0 & 15 = 00000000000000000000000000000101
```



```
rminal  Help                Instruction.java - mips-assembler - Visual Studio Code
PROBLEMS 8  OUTPUT  DEBUG CONSOLE  TERMINAL                    2: Java Process Console
Machine Code of the Instruction Of Line: 5      00110010101101010000000000001111
**************Execution**************
Line Number:6
$s0 = 21 And it's binary value is 00000000000000000000000000010101
The Shift Amount = 8
 The new value of register $s0 = 0
Its binary value = 00000000000000000000000000000000
Machine Code of the Instruction Of Line: 6      00000000000000000000001000000010
**************Execution**************
Line Number:7
$t2 = 3 And it's binary value is 00000000000000000000000000000011
The Shift Amount = 4
 The new value of register $t1 = 0
Its binary value = 00000000000000000000000000000000
Machine Code of the Instruction Of Line: 7      00000000000000110000000100000011
The decimal value for the instruction  nor is so big to be an integer, it won't be added to the register.
**************Execution**************
Line Number:8
$t2 = 3 And it's binary value is 00000000000000000000000000000011
$t3 = 0 And it's binary value is 00000000000000000000000000000000
$t1 =  $t2 NOR $t3 = 11111111111111111111111111111100
The new value of register $t1 = 0
Machine Code of the Instruction Of Line: 8      00000000011000000000000000100111
**************Execution**************
Line Number:9
$t2 = 3 And 00000000000000000000000000000011
$t3 = 0 And 00000000000000000000000000000000
$t1 =  $t2 & $t3 = 00000000000000000000000000000011
The new value of register $t1 = 3
Machine Code of the Instruction Of Line: 9      00000000011000000001100000100101
**************Execution**************
Line Number:10
$t2 = 3 And 00000000000000000000000000000011
5 = 5 And 00000000000000000000000000000101
$t1 =  $t2 & 5 = 00000000000000000000000000000111
The new value of register $t1 = 3
Machine Code of the Instruction Of Line: 10     00110100011000110000000000000101
**************Execution**************
Line Number:11  $t1 = $t2 - $t3  = 3
*******************************

Machine Code of the Instruction Of Line: 11     00000000011000000001100000100010
**************Execution**************
Line Number:12
$t0 = 0 And it's binary value is 00000000000000000000000000000000
```


rminal Help
Instruction.java - mips-assembler - Visual Studio Code
PROBLEMS 8
OUTPUT
DEBUG CONSOLE
TERMINAL
2: Java Process Console
Machine Code of the Instruction Of Line: 11 00000000011000000001100000100010
*************Execution*************
Line Number:12
$t0 = 0 And it's binary value is 00000000000000000000000000000000
= 0 << 16
Machine Code of the Instruction Of Line: 12 00111100000000000000000000000000
Symbol Table:
Label Address (in hex)
MIPS Code Address Machine Lang.
add $s6, $s5, $zero 00400000 00000000100000000100000000100000
addi $s1, $s2, 4 00400004 00100010010101100000000000000100
and $t1, $t2, $t3 00400008 00000000011000000000000000100100
add $s6, $s5, $zero 0040000c 00000000100000000010000000100000
andi $s0, $s0, 15 00400010 00110010101101010000000000001111
srl $s0, $s0, 8 00400014 00000000000000000000001000000010
sra $t1, $t2, 4 00400018 00000000000000011000000100000011
nor $t1, $t2, $t3 0040001c 00000000011000000000000000100111
or $t1, $t2, $t3 00400020 00000000011000000001100000100101
ori $t1, $t2, 5 00400024 00110100011000110000000000000101
sub $t1, $t2, $t3 00400028 00000000011000000001100000100010
lui $t0, 15, 0 0040002c 00111100000000000000000000000000
Address Machine Code
00400000 00000000100000000100000000100000
Address Machine Code
00400004 00100010010101100000000000000100
Address Machine Code
00400008 00000000011000000000000000100100
Address Machine Code
0040000c 00000000100000000100000000100000
Address Machine Code
00400010 00110010101101010000000000001111
Address Machine Code
00400014 00000000000000000000001000000010
Address Machine Code


rminal Help
Instruction.java - mips-assembler - Visual Studio Code
PROBLEMS 8
OUTPUT
DEBUG CONSOLE
TERMINAL
2: Java Process Console
srl $s0, $s0, 8 00400014 00000000000000000000001000000010
sra $t1, $t2, 4 00400018 00000000000000011000000100000011
nor $t1, $t2, $t3 0040001c 00000000011000000000000000100111
or $t1, $t2, $t3 00400020 00000000011000000001100000100101
ori $t1, $t2, 5 00400024 00110100011000110000000000000101
sub $t1, $t2, $t3 00400028 00000000011000000001100000100010
lui $t0, 15, 0 0040002c 00111100000000000000000000000000
Address Machine Code
00400000 00000000100000000100000000100000
Address Machine Code
00400004 00100010010101100000000000000100
Address Machine Code
00400008 00000000011000000000000000100100
Address Machine Code
0040000c 00000000100000000100000000100000
Address Machine Code
00400010 00110010101101010000000000001111
Address Machine Code
00400014 00000000000000000000001000000010
Address Machine Code
00400018 00000000000000011000000100000011
Address Machine Code
0040001c 00000000011000000000000000100111
Address Machine Code
00400020 00000000011000000001100000100101
Address Machine Code
00400024 00110100011000110000000000000101
Address Machine Code
00400028 00000000011000000001100000100010
Address Machine Code
0040002c 00111100000000000000000000000000

Conclusion

This project aimed to show object-oriented programming skills in constructing and building a MIPS Simulator, and we succeeded in building it in a very small time. We advise in reaching the code because there are some details we did not mention as they are already explained in the Source Code. Refer to the pipeline.pptx file for the explanation of pipeline stages instruction execution.

References

Lafore, R. (2017). Data structures and algorithms in Java. Sams publishing.

Tinder, Richard F. (2000). Engineering digital design: Revised Second Edition. pp. 317–319. ISBN 0-12-691295-5. Retrieved 2008-07-04.

Harris, David Harris, Sarah (2007). Digital design and computer architecture (1st ed.). San Francisco,Calif.: Morgan Kaufmann.

David A. Patterson and John L. Hennessy. 2013. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface (5th. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.