CMSC 132 PROJECT: A SIMPLE COMPUTER FOR 24-BIT INPUT

BY

Fuchsia Pink

PATRICIA JUNE CANCERAN

MARION MAULEON

MARIE LORAINE RACASA

ERICA MAE YEBAN

BS COMPUTER SCIENCE

Submitted to Kristine Elaine Bautista in Partial Fulfillment of the Requirements of

CMSC 132 (Computer Architecture)

Second Semester, AY 2014-2015

MAY 2015

I. Introduction and Statement of the Problem

The project is an instruction set simulator designed after WinMIP64. It supports a 24-bit input with the first 8-bit allocated for instruction and the 16-bits for the two operands with 8-bit each. There are 8 general purpose 8-bit registers, from r0 to r7, two 8-bit memory address registers, MAR0 and MAR1, one 8-bit program counter, one 8-bit instruction register and one 8-bit flag.

The application produced in this project can be viewed as a window with six sub windows inside which are Code Window, Cycle Window, Pipeline Window, Register Window, Data Window and Input Textbox. These work together as one in order to give user a systematic view of how instructions and registers are executed.

In order to produce a fully functional program the following problems have to be solved:

- How to provide the mapping of opcode to the machine code?
- What register number should be assigned to a register number?
- What will be our ALU design, Data path design and Control Unit design?
- How will be the implementation in a chosen programming language in order to reach the goal of the project?

If all the problems are solved, a successful program is expected to be produced.

II. Mapping of opcode to machine code

INSTRUCTIONS	INSTRUCTION CODE	BINARY CODE
DATA TRANSFER INSTRUCTIONS		
LOAD	LOAD	0000 0000
STORE	STORE	0000 0001
SAVE	SAVE	0000 0010
ARITHMETIC INSTRUCTIONS		
INC	INC	0010 0000
DEC	DEC	0010 0001
ADD	ADD	0010 0010
SUB	SUB	0010 0011
MUL	MUL	0010 0100
DIV	DIV	0010 0101
COMPARISON OPERATION		
CMP	CMP	0100 0000
LOGIC INSTRUCTIONS		
AND	AND	0100 0000
OR	OR	0100 0001
NOT	NOT	0100 0010
XOR	XOR	0100 0011
PROGRAM FLOW INSTRUCTIONS		
JE	JE	1000 0000
JG	JG	1000 0001
JL	JL	1000 0010
JMP	JMP	1000 0011

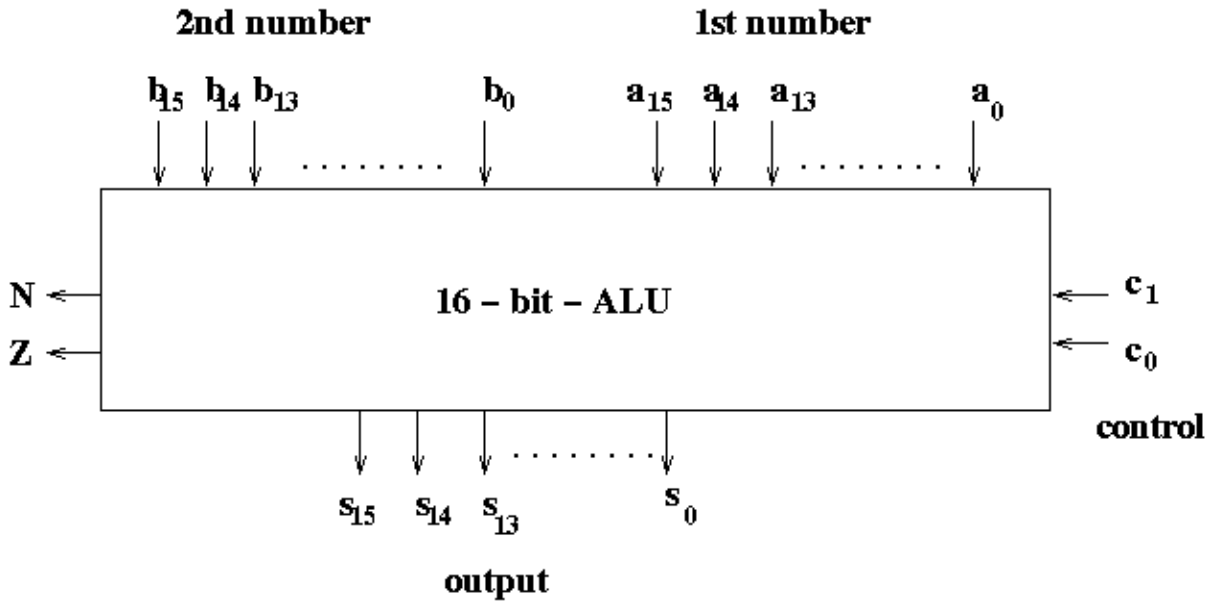
III. Register Assignments

	REGISTER CODE	**BINARY CODE**
REGISTERS	R0	0000 0000
	R1	0000 0001
	R2	0000 0010
	R3	0000 0011
	R4	0000 0100
	R5	0000 0101
	R6	0000 0110
	R7	0000 0111
MEMORY ADDRESS REGISTERS	MAR0	0010 0000
	MAR1	0010 0001
PROGRAM COUNTER	PC	0100 0000
INSTRUCTION REGISTER	IR	0110 0000
FLAG	FLAG	1000 0000

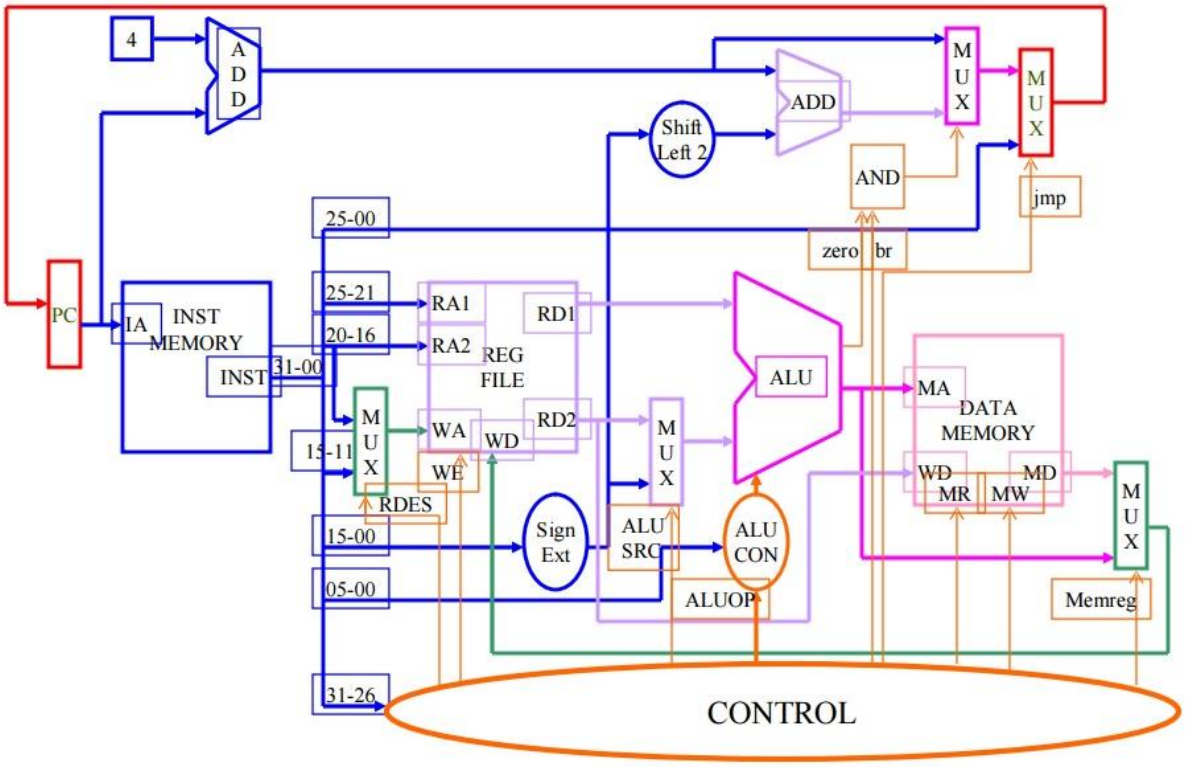
INSTRUCTIONS	**SYNTAX**
Data Transfer Instructions	
Load	LOAD <register> <memory address>
Store	STORE <memory address> <register>
Save	SAVE <register> <immediate value>
Arithmetic Instructions	
Inc	INC <register> <register>
Dec	DEC <register> <register>
Add	ADD <register> <register>

Sub	SUB <register> <register>
Mul	MUL <register> <register>
Div	DIV <register> <register>
Comparison Operations	
Cmp	CMP <register> <register>
Logic Instructions	
And	AND <register> <register>
Or	OR <register> <register>
Not	NOT <register> <register>
Xor	XOR <register> <register>
Program Flow Instructions	
JE	JE <label>
JG	JG <label>
JL	JL <label>
JMP	JMP <label>

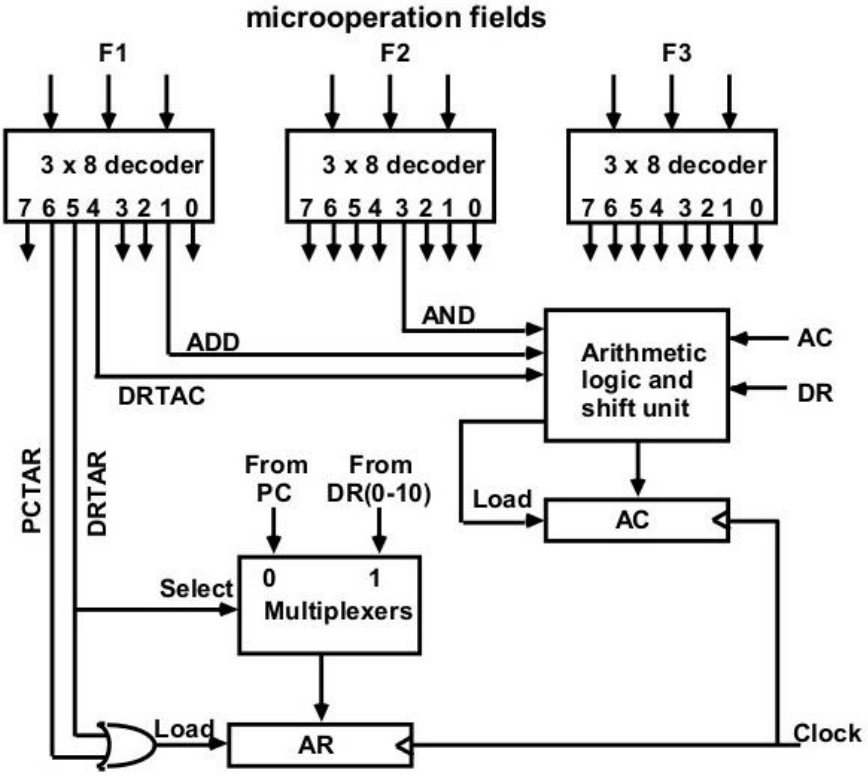
IV. ALU Design



V. Data Path Design



VI. Control Unit Design



VII. Programming Language and Implementation

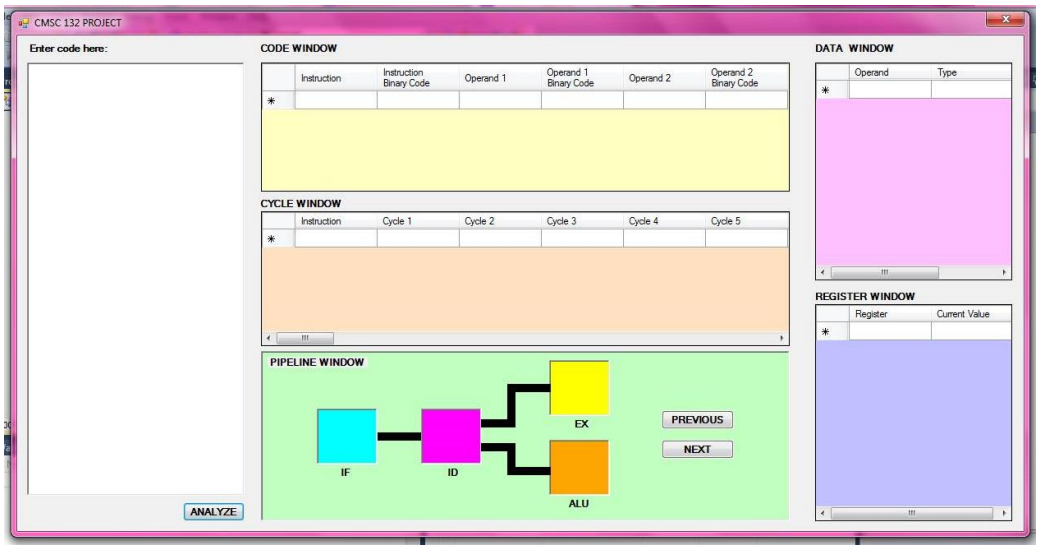
In this project, we used Visual C# as our programming language.

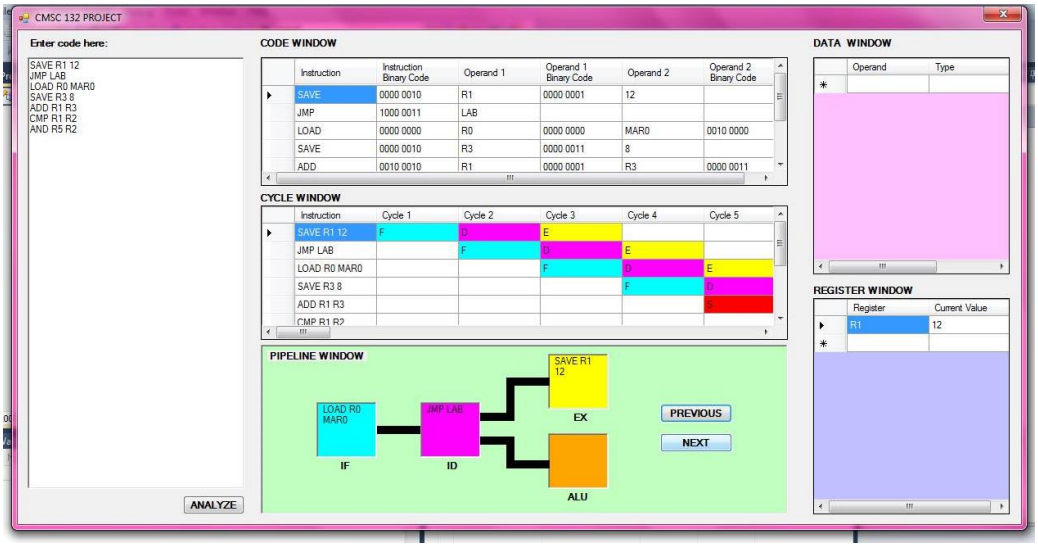
After the analyze button has been pressed, the input is parsed by line. There are three initialized linked lists for the Cycle window named as F, D and E. These linked lists together with Code, Cycle and Register Window will be cleared at the beginning. A set of regex is initialized for the checking of the lines parsed. Every matched regex and line from the input will be added into the code window. The information added are already separated into columns of instruction, operand1 and operand2. These lines are also already classified based on the type of instruction. (e.g. Arithmetic instruction, Logic instruction, Comparison Instruction, etc.)

The next part of the process done after pressing Analyze button is using the data inside the cells of code window in order to see if there are any hazards which are existing from the input. The operand1 of a line and operand2 of another line is compared to see if there is a RAW, WAW or WAR hazard. The variable "lastDep", initialized as zero, is updated to keep track where the fetch-decode-execute cycle in cycle window should start. If lastDep is still equal to zero it means that there is no dependency or hazard present in the line and a normal addition of fetch-decode-execute cycle in cycle window will be done. If there is a hazard detected, the value of "lastDep" will be the reference of where the current instruction should start the cycle. In arithmetic and logic instructions decode and execute are added twice. Stalls will also be added if necessary.

The next and previous button is for the pipeline and register window. The pipeline window shows the step by step execution of every line of instruction. We implemented it by checking every columns of cycle window. Every press of next button is counted and the count correspond to the column number to be scanned. Also the value of the register in register window is changed every time the next button is pressed depending on the instruction accessed. The previous button allows the user to view the previous state of pipeline and register window.

VIII. Screenshots





IX. Conclusion

The program produced in this project is an effective and simple way of showing to the user how instructions are processed.

X. References

Basic computer programming and micro programmed control. (n.d.). Retrieved May 28, 2015, from http://www.slideshare.net/raiuniversity/mca-iu32basic-computer-programming-and-micro-programmed-control

CS 355 - Computer Organization/Architecture II Project 3 (n.d.). Retrieved May 28, 2015, from http://www.mathcs.emory.edu/~cheung/Courses/355/Projects/pj3.html

Data Path and Control Design. (n.d.). Retrieved May 28, 2015, from http://class.ece.iastate. edu/arun/Cpre381_Sp06/lectures/MIPS_SC.pdf

XI. Appendix B: Code of the Program

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Text.RegularExpressions;
using System.Collections;
namespace WindowsFormsApplication1
{
  public partial class Form1 : Form
  {
    static Hashtable opI = new Hashtable();    //INSTRUCTIONS
    static Hashtable opR = new Hashtable();    //REGISTERS
    static Hashtable reg = new Hashtable();    //REGISTERS
    static int indicator = 1, last = 1, aflag = 0, matched = -1;  //FLAGS

    //--- REGEX DECLARATIONS ---
    static string immediate = @"(?<immediate>(?<floatliteral>-?[0-9]+(\.[0-9]+))|(?<numberliteral>-?[0-9]+))";
    static string registers = @"(R0|R1|R2|R3|R4|R5|R6|R7)";
    static string mar = @"(MAR0|MAR1)";
    static string arithmeticInstructions = @"(INC|DEC|ADD|SUB|DIV|MUL)";
    static string logicInstructions = @"(AND|OR|NOT|XOR)";
    static string jumpsInstructions = @"(JE|JG|JL|JMP)";
    static string jumpLabel = @"[a-zA-Z]\w*";

    //--- LINKED LISTS FOR F, D, E ---
    static LinkedList<int> F = new LinkedList<int>();
    static LinkedList<int> D = new LinkedList<int>();
    static LinkedList<int> E = new LinkedList<int>();

   //--- DATA TRANSFER INSTRUCTIONS ---
   Regex load = new Regex(@"(?<instruction>LOAD)\s+(?<operand1>" + registers + @")\s+(?<operand2>" + mar +
")");
   Regex save = new Regex(@"(?<instruction>SAVE)\s+(?<operand1>" + registers + @")\s+(?<operand2>" +
immediate + ")");
   Regex store = new Regex(@"(?<instruction>STORE)\s+(?<operand1>" + mar + @")\s+(?<operand2>" + registers +
")");

   //--- ARITHMETIC INSTRUCTIONS ---
   Regex arithmetic = new Regex(@"(?<instruction>" + arithmeticInstructions + @")\s+(?<operand1>" + registers +
@")\s+(?<operand2>" + registers + ")");

   //--- COMPARISON OPERATIONS ---
   Regex compare = new Regex(@"(?<instruction>CMP)\s+(?<operand1>" + registers + @")\s+(?<operand2>" +
registers + ")");

   //--- LOGIC INSTRUCTIONS ---
   Regex logic = new Regex(@"(?<instruction>" + logicInstructions + @")\s+(?<operand1>" + registers +
@")\s+(?<operand2>" + registers + ")");

   //--- PROGRAM FLOW INSTRUCTIONS ---
   Regex jump = new Regex(@"(?<instruction>" + jumpsInstructions + @")\s+(?<operand1>" + jumpLabel + @")");


    public Form1()
    {
      InitializeComponent();
      //BINARY CODES
      // --- DATA TRANSFER INSTRUCTIONS ---
      opI.Add("LOAD", "0000 0000");
      opI.Add("STORE", "0000 0001");
      opI.Add("SAVE", "0000 0010");
      // --- ARITHMETIC INSTRUCTIONS ---
      opI.Add("INC", "0010 0000");
      opI.Add("DEC", "0010 0001");
      opI.Add("ADD", "0010 0010");
      opI.Add("SUB", "0010 0011");
      opI.Add("MUL", "0010 0100");
      opI.Add("DIV", "0010 0101");
      // --- COMPARISON INSTRUCTIONS ---
      opI.Add("CMP", "0100 0000");
      // --- LOGIC INSTRUCTIONS ---
      opI.Add("AND", "0100 0000");
      opI.Add("OR", "0100 0001");
      opI.Add("NOR", "0100 0010");
      opI.Add("XOR", "0100 0011");
      // --- PROGRAM FLOW INSTRUCTIONS ---
      opI.Add("JE", "1000 0000");
      opI.Add("JG", "1000 0001");
      opI.Add("JL", "1000 0010");
      opI.Add("JMP", "1000 0011");
      // --- REGISTERS ---
      opR.Add("R0", "0000 0000");
      opR.Add("R1", "0000 0001");
      opR.Add("R2", "0000 0010");
      opR.Add("R3", "0000 0011");
      opR.Add("R4", "0000 0100");
      opR.Add("R5", "0000 0101");
      opR.Add("R6", "0000 0110");
      opR.Add("R7", "0000 0111");
      // --- MEMORY ADDRESS REGISTERS ---
      opR.Add("MAR0", "0010 0000");
      opR.Add("MAR1", "0010 0001");
      // --- PROGRAM COUNTER ---
      opR.Add("PC", "0100 0000");
      // --- INSTRUCITON REGISTER ---
      opR.Add("IR", "0110 0000");
      // --- FLAG ---
      opR.Add("FLAG", "1000 0000");

      //--- VALUE OF REGISTERS ARE INITIALIZED TO 0
      for(int i = 0; i<8; i++)
      {
        reg.Add("R" + i, 0.0);
      }
      reg.Add("MAR0", 0.0);
      reg.Add("MAR1", 0.0);

      //--- SETTING COLORS TO PIPELINE WINDOW PANELS
      IF.BackColor = System.Drawing.Color.Cyan;
      ID.BackColor = System.Drawing.Color.Fuchsia;
      EX.BackColor = System.Drawing.Color.Yellow;
      ALU.BackColor = System.Drawing.Color.Orange;
    }

    private void button1_Click(object sender, EventArgs e)  // ANALYZE BUTTON
    {
      //Clears the table contents every start
      codeWindow.Rows.Clear();
      cycleWindow.Rows.Clear();
      registerWindow.Rows.Clear();
      F.Clear();
      D.Clear();
      E.Clear();

      //SETS THE STARTING POINT OF THE FDE CYCLE
      F.AddLast(1);
      D.AddLast(1);
      D.AddLast(1);
      E.AddLast(1);
      E.AddLast(1);
      E.AddLast(1);

      //Parsing for each inputLines[i], separated by new inputLines[i]
      String[] inputLines = input.Text.Split('\n');

      //MAXIMUM NO OF COLUMNS PER ROW
      for (int i = 0; i < inputLines.Length*5; i++ )
      {
        F.AddLast(0);
        D.AddLast(0);
        E.AddLast(0);
      }

      if (input.Text.Length > 0)
      {
        for (int i = 0; i < inputLines.Length; i++  )
        {
          matched = 1;
          string instr = "";
          string op1 = "";
          string op2 = "";

          cycleWindow.Rows.Add();
          cycleWindow.Rows[i].Cells[0].Value = inputLines[i];

          // --- DATA TRANSFER INSTRUCTIONS ---

          //CHECK IF LOAD
          if (load.Match(inputLines[i]).Success)
          {
            instr = load.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = load.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = load.Match(inputLines[i]).Groups["operand2"].Value.ToString();

            codeWindow.Rows.Add(instr, (string)opI[instr], op1, (string)opR[op1], op2, (string)opR[op2]);
          }
          //CHECK IF SAVE
          else if (save.Match(inputLines[i]).Success)
          {
            instr = save.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = save.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = save.Match(inputLines[i]).Groups["operand2"].Value.ToString();
```

```
            codeWindow.Rows.Add(instr, (string)opl[instr], op1, (string)opR[op1], op2, (string)opR[op2]);
        }
        //CHECK IF STORE
        else if (store.Match(inputLines[i]).Success)
        {
            instr = store.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = store.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = store.Match(inputLines[i]).Groups["operand2"].Value.ToString();

            codeWindow.Rows.Add(instr, (string)opl[instr], op1, (string)opR[op1], op2, (string)opR[op2]);
        }
        // --- ARITHMETIC INSTRUCTIONS ---
        //CHECK IF INC,DEC,ADD,SUB,MUL,DIV
        else if (arithmetic.Match(inputLines[i]).Success)
        {
            instr = arithmetic.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = arithmetic.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = arithmetic.Match(inputLines[i]).Groups["operand2"].Value.ToString();

            codeWindow.Rows.Add(instr, (string)opl[instr], op1, (string)opR[op1], op2, (string)opR[op2]);
        }
        // --- COMPARISON INSTRUCTIONS ---
        //CHECK IF CMP
        else if (compare.Match(inputLines[i]).Success)
        {
            instr = compare.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = compare.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = compare.Match(inputLines[i]).Groups["operand2"].Value.ToString();

            codeWindow.Rows.Add(instr, (string)opl[instr], op1, (string)opR[op1], op2, (string)opR[op2]);
        }
        // --- LOGIC INSTRUCTIONS ---
        //CHECK IF AND,OR,NOT,XOR
        else if (logic.Match(inputLines[i]).Success)
        {
            instr = logic.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = logic.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = logic.Match(inputLines[i]).Groups["operand2"].Value.ToString();

            codeWindow.Rows.Add(instr, (string)opl[instr], op1, (string)opR[op1], op2, (string)opR[op2]);
        }
        // --- PROGRAM FLOW INSTRUCTIONS ---
        //CHECK IF JE,JG,JL,JMP
        else if (jump.Match(inputLines[i]).Success)
        {
            instr = jump.Match(inputLines[i]).Groups["instruction"].Value.ToString();
            op1 = jump.Match(inputLines[i]).Groups["operand1"].Value.ToString();
            op2 = jump.Match(inputLines[i]).Groups["operand2"].Value.ToString();

            codeWindow.Rows.Add(instr, (string)opl[instr], op1, (string)opR[op1], op2, (string)opR[op2]);



        }

        else
        {
            matched = -1;
        }

        if (matched != -1)
        {
          //  if (!jump.Match(inputLines[i]).Success)
          // {
                int lastDep = 0;
                for (int j = 0; j < codeWindow.Rows.Count - 1; j++)
                {
                    // WAW
                    if (codeWindow.Rows[j].Cells[2].Value != null &&
codeWindow.Rows[j].Cells[2].Value.ToString().Equals(op1))
                    {
                        int k;
                        for (k = cycleWindow.Rows[0].Cells.Count - 1; k >= 0; k--)
                        {
                            if (cycleWindow.Rows[j].Cells[k].Value != null &&
cycleWindow.Rows[j].Cells[k].Value.ToString().Equals("E"))
                            {
                                break;
                            }
                        }
                        if (lastDep < k)
                        {
                            lastDep = k + 1;
                        }
                    }
                    // WAR
                    else if (codeWindow.Rows[j].Cells[4].Value != null &&
(codeWindow.Rows[j].Cells[4].Value.ToString().Equals(op1) || op1.Equals(op2)))
                    {
                        int k;
                        for (k = cycleWindow.Rows[0].Cells.Count - 1; k >= 0; k--)
                        {
                            if (cycleWindow.Rows[j].Cells[k].Value != null &&
cycleWindow.Rows[j].Cells[k].Value.ToString().Equals("E"))
                            {
                                break;
                            }
                        }
                        if (lastDep < k)
                        {
                            lastDep = k + 1;
                        }
                    }
                    //RAW
                    else if (codeWindow.Rows[j].Cells[2].Value != null &&
codeWindow.Rows[j].Cells[2].Value.ToString().Equals(op2))
                    {
                        int k;
                        for (k = cycleWindow.Rows[0].Cells.Count - 1; k >= 0; k--)
                        {
                            if (cycleWindow.Rows[j].Cells[k].Value != null &&
cycleWindow.Rows[j].Cells[k].Value.ToString().Equals("E"))
                            {
                                break;
                            }
                        }
                        if (lastDep < k)
                        {
                            lastDep = k + 1;
                        }
                    }
                }

                // No dependence
                if (lastDep == 0)
                {

                    int j;
                    // F
                    for (j = 1; j < cycleWindow.Rows[0].Cells.Count; j++)
                    {
                        int k;
                        for (k = 0; k < cycleWindow.Rows.Count; k++)
                        {
                            if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("F"))
                            {
                                break;
                            }
                        }
                        if (k >= cycleWindow.Rows.Count)
                        {
                            cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "F";
                            cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Cyan;
                            break;
                        }
                    }
                    //cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[cycleWindow.Rows.Count - 1].Value = "F";
                    //cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[cycleWindow.Rows.Count -
1].Style.BackColor = System.Drawing.Color.Cyan;



                    // D
                    for (j += 1; j < cycleWindow.Rows[0].Cells.Count; j++)
                    {
                        int k;
                        for (k = 0; k < cycleWindow.Rows.Count; k++)
                        {
                            if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("D"))
                            {
                                break;
                            }
                        }
                        if (k >= cycleWindow.Rows.Count)
                        {
                            cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "D";
                            cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Fuchsia;
                            break;
                        }
                    }

                    // Second D
                    if (arithmetic.Match(inputLines[i]).Success || logic.Match(inputLines[i]).Success)
                    {
                        for (j += 1; j < cycleWindow.Rows[0].Cells.Count; j++)
                        {
                            int k;
                            for (k = 0; k < cycleWindow.Rows.Count; k++)
                            {
                                if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("D"))
                                {
                                    break;
                                }
                            }
                            if (k >= cycleWindow.Rows.Count)
                            {
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "D";
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Fuchsia;
                                break;
                            }
                        }
                    }

                    // E
                    for (j += 1; j < cycleWindow.Rows[0].Cells.Count - 1; j++)
                    {
                        int k;
                        for (k = 0; k < cycleWindow.Rows.Count; k++)
                        {
                            if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("E"))
                            {
                                break;
                            }
                        }
                        if (k >= cycleWindow.Rows.Count)
                        {
                            cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "E";
                            cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Yellow;
                            break;
                        }
                    }
```

```
                    // Second E
                    if (arithmetic.Match(inputLines[i]).Success || logic.Match(inputLines[i]).Success)
                    {
                        for (j += 1; j < cycleWindow.Rows[0].Cells.Count - 1; j++)
                        {
                            int k;
                            for (k = 0; k < cycleWindow.Rows.Count; k++)
                            {
                                if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("E"))
                                {
                                    break;
                                }
                            }
                            if (k >= cycleWindow.Rows.Count)
                            {
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "E";
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Yellow;
                                break;
                            }
                        }
                    }

                    // DEPENDENCY
                    else
                    {
                        cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[0].Value = inputLines[i];
                        int j;

                        // F
                        for (j = lastDep; j < cycleWindow.Rows[0].Cells.Count; j++)
                        {
                            int k;
                            for (k = 0; k < cycleWindow.Rows.Count; k++)
                            {
                                if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("F"))
                                {
                                    break;
                                }
                            }
                            if (k >= cycleWindow.Rows.Count)
                            {
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "F";
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Cyan;
                                break;
                            }
                        }

                        //cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[lastDep].Value = "F";
                        //cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[lastDep].Style.BackColor =
System.Drawing.Color.Cyan;

                        // D
                        for (j += 1; j < cycleWindow.Rows[0].Cells.Count; j++)
                        {
                            int k;
                            for (k = 0; k < cycleWindow.Rows.Count; k++)
                            {
                                if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("D"))
                                {
                                    break;
                                }
                            }
                            if (k >= cycleWindow.Rows.Count)
                            {
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "D";
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Fuchsia;
                                break;
                            }
                        }

                        // Second D
                        if (arithmetic.Match(inputLines[i]).Success || logic.Match(inputLines[i]).Success)
                        {
                            for (j += 1; j < cycleWindow.Rows[0].Cells.Count; j++)
                            {
                                int k;
                                for (k = 0; k < cycleWindow.Rows.Count; k++)
                                {
                                    if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("D"))
                                    {
                                        break;
                                    }
                                }
                                if (k >= cycleWindow.Rows.Count)
                                {
                                    cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "D";
                                    cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Fuchsia;
                                    break;
                                }
                            }
                        }

                        // E
                        for (j += 1; j < cycleWindow.Rows[0].Cells.Count - 1; j++)
                        {
                            int k;
                            for (k = 0; k < cycleWindow.Rows.Count; k++)
                            {
                                if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("E"))
                                {
                                    break;
                                }
                            }
                            if (k >= cycleWindow.Rows.Count)
                            {
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "E";
                                cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Yellow;
                                break;
                            }
                        }

                        // Second E
                        if (arithmetic.Match(inputLines[i]).Success || logic.Match(inputLines[i]).Success)
                        {
                            for (j += 1; j < cycleWindow.Rows[0].Cells.Count - 1; j++)
                            {
                                int k;
                                for (k = 0; k < cycleWindow.Rows.Count; k++)
                                {
                                    if (cycleWindow.Rows[k].Cells[j].Value != null &&
cycleWindow.Rows[k].Cells[j].Value.Equals("E"))
                                    {
                                        break;
                                    }
                                }
                                if (k >= cycleWindow.Rows.Count)
                                {
                                    cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Value = "E";
                                    cycleWindow.Rows[cycleWindow.Rows.Count - 2].Cells[j].Style.BackColor =
System.Drawing.Color.Yellow;
                                    break;
                                }
                            }
                        }
                    }
                }

            }

            for (int i = 0; i < inputLines.Count(); i++)    //STALL in between FDE
            {
                int j, k;
                for (j = (cycleWindow.Rows[i].Cells.Count) - 1; cycleWindow.Rows[i].Cells[j].Value == null && j > i; j--) { } // j =
last E of the current row

                for (k = 1; k < j && i != 0; k++) {
                    int l;
                    for (l = i; l > 2; l--)
                    {
                        if (cycleWindow.Rows[l].Cells[k].Value != null && cycleWindow.Rows[l].Cells[k].Value.Equals("F"))
                        {
                                                    break;
                        }
                    }
                    if (l <= 0) // f not found
                    {
                        break;
                    }
                } // first position of F, D, E, or S of the row
                for (; k < j; k++)
                {
                    if (cycleWindow.Rows[i].Cells[k].Value == null)
                    {
                        cycleWindow.Rows[i].Cells[k].Value = "S";
                        cycleWindow.Rows[i].Cells[k].Style.BackColor = System.Drawing.Color.Red;
                    }
                }
                last = 1;
                for (int j = 0; j < cycleWindow.Rows[0].Cells.Count; j++)   //GETS THE LAST CYCLE
                {
                    int i;
                    for (i = 0; i < cycleWindow.Rows.Count; i++)
                    {
                        if (cycleWindow.Rows[i].Cells[j].Value != null)
                        {
                            break;
                        }
                    }
                    if (i >= cycleWindow.Rows.Count)
                    {
                        last = j - 1;
                        break;
                    }
                }
                indicator = 1;

                IF_LABEL.Text = "";
                ID_LABEL.Text = "";
                EX_LABEL.Text = "";
                ALU_LABEL.Text = "";

                //PRINTS THE INSTRUCTION IN THE PIPELINE WINDOW
                for (int i = 0; i < cycleWindow.Rows.Count; i++)
                {
                    if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("F"))
                    {
                        IF_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
                    }
                    else if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("D"))
                    {
                        ID_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
                    }
```

```
            else if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("E"))
            {
               EX_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
               if (arithmetic.Match(cycleWindow.Rows[i].Cells[0].Value.ToString()).Success ||
logic.Match(cycleWindow.Rows[i].Cells[0].Value.ToString()).Success)
               {
                  ALU_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
               }
            }
         }


      }


      private void input_TextChanged(object sender, EventArgs e)
      {

      }

      private void cycleWindow_Paint(object sender, PaintEventArgs e)
      {

      }

      private void pipelineWindow_Paint(object sender, PaintEventArgs e)
      {

      }

      private void Form1_Load(object sender, EventArgs e)
      {

      }

      private void cycleWindow_CellContentClick(object sender, DataGridViewCellEventArgs e)
      {

      }

      private void codeWindow_CellContentClick(object sender, DataGridViewCellEventArgs e)
      {

      }

      private void flowLayoutPanel2_Paint(object sender, PaintEventArgs e)
      {

      }

      private void previous_Click(object sender, EventArgs e) //PRINTS THE PREVIOUS STATE OF REGISTERS IN
REGISTER WINDOW AND PREVIOUS INSTRUCTIONS IN PIPELINE WINDOW
      {
         if (indicator > 1)
         {
            indicator--;

            IF_LABEL.Text = "";
            ID_LABEL.Text = "";
            EX_LABEL.Text = "";
            ALU_LABEL.Text = "";

            for (int i = 0; i < cycleWindow.Rows.Count; i++)
            {
               if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("F"))
               {
                  IF_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
               }
               else if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("D"))
               {
                  ID_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
               }
               else if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("E"))
               {
                  EX_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
                  if (arithmetic.Match(cycleWindow.Rows[i].Cells[0].Value.ToString()).Success ||
logic.Match(cycleWindow.Rows[i].Cells[0].Value.ToString()).Success)
                  {
                     ALU_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
                  }
               }
            }

            if (save.Match(EX_LABEL.Text).Success)  //PRINTS THE VALUE OF THE REGISTER IN SAVE INSTRUCTION
            {
               string op1 = save.Match(EX_LABEL.Text).Groups["operand1"].Value;
               string op2 = save.Match(EX_LABEL.Text).Groups["operand2"].Value;
               int flag = 0;

               reg[op1] = float.Parse(op2);
               float op1value = (float)reg[op1];

               for (int j = 0; j < registerWindow.Rows.Count; j++)
               {
                  if (registerWindow.Rows[j].Cells[0].Value != null &&
op1.Equals(registerWindow.Rows[j].Cells[0].Value.ToString()))
                  {
                     registerWindow.Rows[j].Cells[1].Value = op1value;
                     flag = 1;
                     break;
                  }
               }
               if (flag == 0)
               {
                  registerWindow.Rows.Add(op1, op1value);
               }
            }

            else if (arithmetic.Match(EX_LABEL.Text).Success)   //EVALUATES THE VALUE ON THE REGISTERS TO SOLVE
THE ARITHMETIC OPERATIONS AND STORES IT INSIDE THE REGISTER
            {
               if (aflag == 1)
               {
                  string instr = arithmetic.Match(EX_LABEL.Text).Groups["instruction"].Value.ToString();
                  string op1 = arithmetic.Match(EX_LABEL.Text).Groups["operand1"].Value;
                  string op2 = arithmetic.Match(EX_LABEL.Text).Groups["operand2"].Value;

                  float op1value = (float)reg[op1];
                  float op2value = (float)reg[op2];

                  if (instr.Equals("ADD"))
                  {
                     reg[op1] = op1value - op2value;
                  }
                  else if (instr.Equals("SUB"))
                  {
                     reg[op1] = op1value + op2value;
                  }
                  else if (instr.Equals("MUL"))
                  {
                     reg[op1] = op1value / op2value;
                  }
                  else if (instr.Equals("DIV"))
                  {
                     reg[op1] = op1value * op2value;
                  }
                  else if (instr.Equals("INC"))
                  {
                     reg[op1] = op2value - 1;
                  }
                  else if (instr.Equals("DEC"))
                  {
                     reg[op1] = op2value + 1;
                  }

                  int flag = 0;

                  for (int j = 0; j < registerWindow.Rows.Count; j++)
                  {
                     if (registerWindow.Rows[j].Cells[0].Value != null &&
op1.Equals(registerWindow.Rows[j].Cells[0].Value.ToString()))
                     {
                        registerWindow.Rows[j].Cells[1].Value = reg[op1].ToString();
                        flag = 1;
                        break;
                     }
                  }
                  if (flag == 0)
                  {
                     registerWindow.Rows.Add(op1, reg[op1].ToString());
                  }
                  aflag = 0;
               }
               else
               {
                  aflag = 1;
               }
            }
         }
      }


      private void next_Click(object sender, EventArgs e)  //PRINTS THE NEXT STATE OF REGISTERS IN REGISTER
WINDOW AND NEXT INSTRUCTIONS IN PIPELINE WINDOW
      {
         if (indicator >= 1 && indicator < last)
         {
            indicator++;

            IF_LABEL.Text = "";
            ID_LABEL.Text = "";
            EX_LABEL.Text = "";
            ALU_LABEL.Text = "";


            for (int i = 0; i < cycleWindow.Rows.Count; i++)
            {
               if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("F"))
               {
                  IF_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
               }
               else if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("D"))
               {
                  ID_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
               }
               else if (cycleWindow.Rows[i].Cells[indicator].Value != null &&
cycleWindow.Rows[i].Cells[indicator].Value.ToString().Equals("E"))
               {
                  EX_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
                  if (arithmetic.Match(cycleWindow.Rows[i].Cells[0].Value.ToString()).Success ||
logic.Match(cycleWindow.Rows[i].Cells[0].Value.ToString()).Success)
                  {
                     ALU_LABEL.Text = "" + cycleWindow.Rows[i].Cells[0].Value;
                  }
               }
            }


            if (save.Match(EX_LABEL.Text).Success)   //PRINTS THE VALUE OF THE REGISTER IN SAVE INSTRUCTION
            {
               string op1 = save.Match(EX_LABEL.Text).Groups["operand1"].Value;
               string op2 = save.Match(EX_LABEL.Text).Groups["operand2"].Value;
               int flag = 0;
```

```
                reg[op1] = float.Parse(op2);
                float op1value = (float)reg[op1];

                for (int j = 0; j < registerWindow.Rows.Count; j++)
                {
                    if (registerWindow.Rows[j].Cells[0].Value != null &&
op1.Equals(registerWindow.Rows[j].Cells[0].Value.ToString()))
                    {
                        registerWindow.Rows[j].Cells[1].Value = op1value;
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0)
                {
                    registerWindow.Rows.Add(op1, op1value);
                }
            }

            else if (arithmetic.Match(EX_LABEL.Text).Success)//EVALUATES THE VALUE ON THE REGISTERS TO SOLVE
THE ARITHMETIC OPERATIONS AND STORES IT INSIDE THE REGISTER

            {
                if (aflag == 1)
                {
                    string instr = arithmetic.Match(EX_LABEL.Text).Groups["instruction"].Value.ToString();
                    string op1 = arithmetic.Match(EX_LABEL.Text).Groups["operand1"].Value;
                    string op2 = arithmetic.Match(EX_LABEL.Text).Groups["operand2"].Value;
                    float op1value = 0, op2value = 0;

                    if (reg[op1] != null)
                    {
                        op1value = float.Parse(reg[op1].ToString());
                    }
                    if (reg[op2] != null)
                    {
                        op2value = float.Parse(reg[op2].ToString());
                    }

                    if (instr.Equals("ADD"))
                    {
                        reg[op1] = op1value + op2value;
                    }
                    else if (instr.Equals("SUB"))
                    {
                        reg[op1] = op1value - op2value;
                    }
                    else if (instr.Equals("MUL"))
                    {
                        reg[op1] = op1value * op2value;
                    }
                    else if (instr.Equals("DIV"))
                    {
                        reg[op1] = op1value / op2value;
                    }
                    else if (instr.Equals("INC"))
                    {
                        reg[op1] = op2value + 1;
                    }
                    else if (instr.Equals("DEC"))
                    {
                        reg[op1] = op2value - 1;
                    }

                    int flag = 0;

                    for (int j = 0; j < registerWindow.Rows.Count; j++)
                    {
                        if (registerWindow.Rows[j].Cells[0].Value != null &&
op1.Equals(registerWindow.Rows[j].Cells[0].Value.ToString()))
                        {
                            registerWindow.Rows[j].Cells[1].Value = reg[op1].ToString();
                            flag = 1;
                            break;
                        }
                    }
                    if (flag == 0)
                    {
                        registerWindow.Rows.Add(op1, reg[op1].ToString());
                    }
                    aflag = 0;
                }
                else
                {
                    aflag = 1;
                }
            }

        }
    }

    private void label5_Click(object sender, EventArgs e)
    {

    }

    private void dataWindow_CellContentClick(object sender, DataGridViewCellEventArgs e)
    {

    }

  }
}
```