German University in Cairo

Faculty of Media Engineering and Technology

Project Report

(Computer System Architecture)

Course Name : CSEN 601

<u>Team Number (Name) :-</u>

(4) (The Best ISA)

<u>Introduced By:-</u>

Ali Kabeel	43-1172	T-15
Salma Khaled	43-2735	T-8
Mohammed Khaled	43-16053	T-16
Aman Allah Rafat	43-17354	T-15
Sarah Ahmed	43-17401	T-10

<u>Prof :-</u>

Hassan Soubra

Sections:

About the Project

The Project is a Java-Based Simulation of the microcontroller architecture implementation with reduced instruction set covering only certain instructions. Throughout this report we will refer to the ISA as B.ISA to be able to compare with MIPS to tackle the pros and cons of this design as compared to MIPS design.

A. Microarchitecture:

The chosen microarchitecture is the Von Neumann architecture having one memory for both data and instructions. In order to protect and handle our memory safely we adopted the no-execute bit approach which is an array of bits (Booleans) to differentiate the data from the instructions to prevent data from being executed. This would save the day in case of buffer flow attacks for instance.

B. Size of Instruction & Data Memory:

The instruction and data memory combined have a size of 2048 words (2048 * 32 bits) which is shared between both instructions and data with no separation as dictated by the Von Neumann architecture.

C. Total number of registers:

Just like MIPS R2000 we have 32 general purpose registers of size 32 bit each but our implementation does not have the zero register. We believe having something hardwired to 0 although useful wasted one of the precious registers so we are giving the program writer the full potential of the 32-bit registers. Removing the zero register was just a matter of design decision that can’t be judged to be better/worse than MIPS since they nearly balance in terms of pros and cons.

D. Instruction Set Chosen:

I) Arithmetic Instructions:
1. Sub.
2. Add.
3. Add immediate.
4. Multiply.

II) Logical Instructions:
1. Or.
2. And immediate.
3. Shift right logical.
4. Shift left logical.

III) Data Transfer Instructions:
1. Load word.
2. Store word.

IV) Conditional Branch Instructions:
1. Branch on equal.
2. Branch on less than.

V) Comparison Instructions:
1. Set on less than immediate.

VI) Unconditional Jump Instructions:
1. Jump Register.

E. Instruction Formats:

We added some changes to MIPS implementation like changing the place of the destination register to be always the first register. In case of Branch and Store Word, we handle those with special treatment since they will use the destination register as source register using flags in the implementation (MemWrite & Branch) to handle the special treatment. The instruction General formats are as follows:

R-Type Instructions Format					
Type	Destination	Source 1	Source 2	Shamt	Opcode
31 30	29 25	24 20	19 15	14 4	3 0

I-Type Instruction Format				
Type	Destination	Source	Immediate	Opcode
31 30	29 25	24 20	19 4	3 0

J-Type Instruction Format			
Type	Source	Not Used	Opcode
31 30	29 25	24 4	3 0

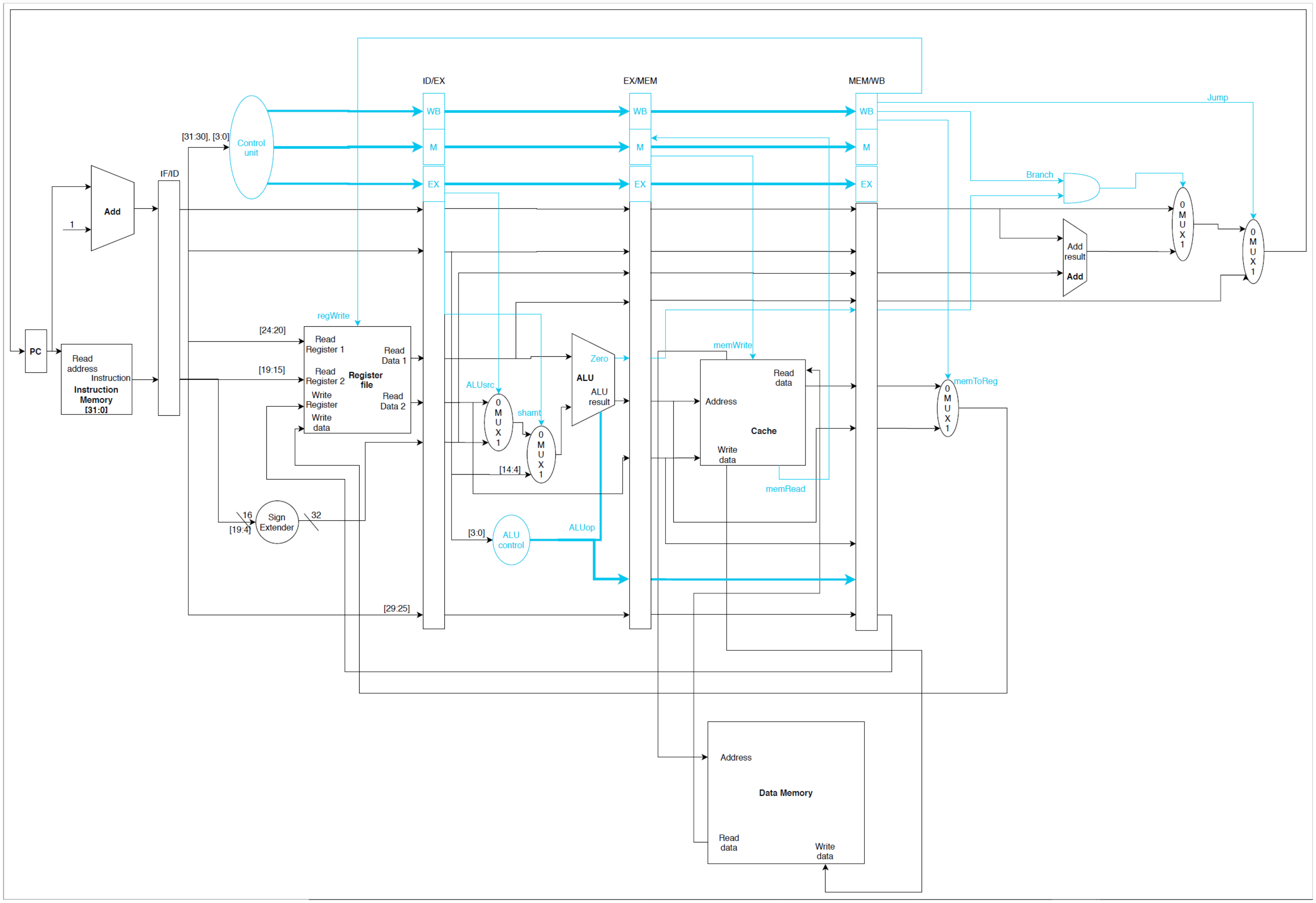
You can also notice that the Instructions have some compromises like having not used part in the J-Type format, increasing the shamt field amount. These are design compromises following the rule “Good design requires good compromises”.

Our new formats can’t be judged to be better than MIPS but would be at least as good as the original MIPS instruction formats. Since destination is now always in the same place, we no longer need the RegDest signal used in MIPS to choose rt between rd.

F. Cache Size and Replacement Policy:

Our Cache is 512 words in size and use the direct mapping replacement policy using tag and index and valid bits. Since the size of cache is chosen to be big, direct mapping would not be a problem in fact it can be optimal since around only 4 words map to same cell as we have 2048 words memory.

Datapath Graphical Representation



Code Flow & Structure

A. PipeLineRegsiter Class:

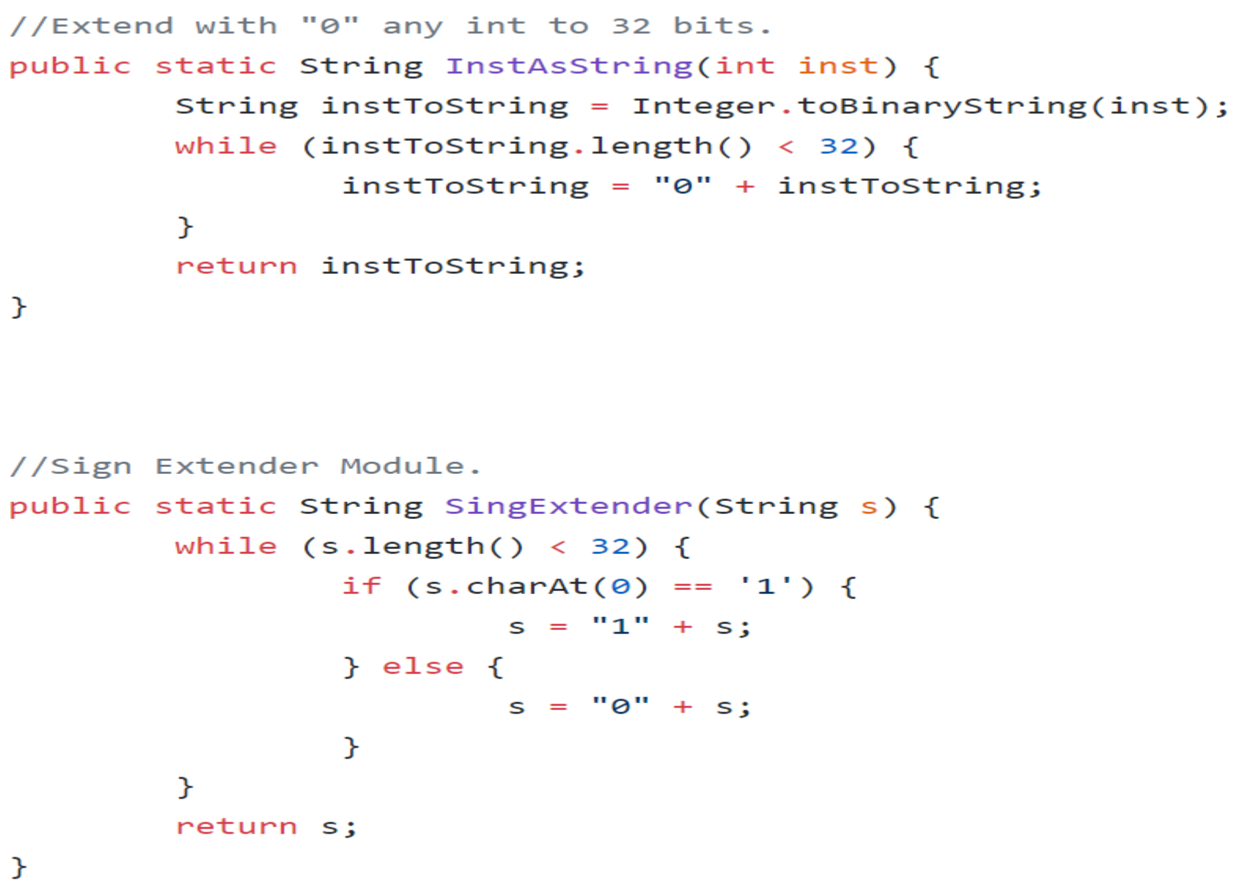
The hero behind the whole migration from single cycle to the pipelined implementation is this class. Although having no methods. Its instance variables carry every single piece of information that might be needed by any stage to complete its work which is what pipeline registers are all about. So basically, this class imitates the pipeline registers in our code.

```
class PipeLineRegister {
        boolean  branch, memRead, memToReg, memWrite, ALUsrc, RegWrite, jump, shamt;
        String AlUop;
        String instruction;
        int PCval;
        int toMemoryVal;
        int src1Val, src2Val, ImmediateVal;
        int WBaddress, WBvalue;
        int jumVal;
        int ALUresult;
        boolean Zero;

}
```

B. RegisterFile Class:

This class simply represents our 32 general purpose registers as a Static array of Strings each of size 32 characters (each character is 0 or 1). We also used methods to ensure there are exactly 32 characters inside registers/memory as shown below:

```
public class RegisterFile {
        //Class representing the 32 registers in the register file.
        static int registers[];
        public RegisterFile() {
                registers = new int[32];
        }

}
/*The registers are named from 0 to 31 in binary respectively.
 * We no longer have a zero register like MIPS.
 * All 32 registers are general purpose and allow data and addresses to be stored in them.
 * No register has a special purpose except for the PC which is kept in the Memory for functionality issues.
 */
```



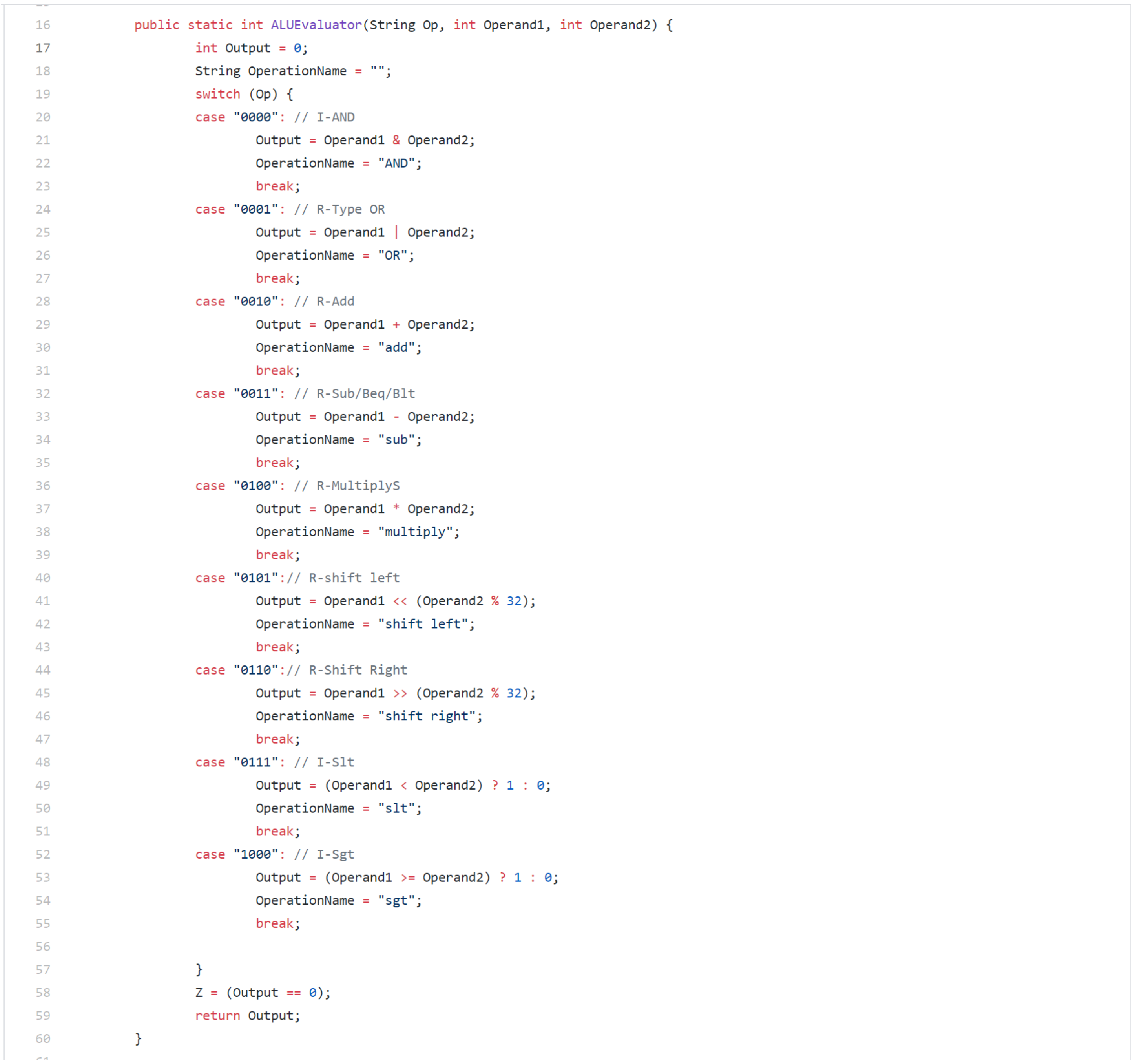
```
//Extend with "0" any int to 32 bits.
public static String InstAsString(int inst) {
        String instToString = Integer.toBinaryString(inst);
        while (instToString.length() < 32) {
                instToString = "0" + instToString;
        }
        return instToString;
}



//Sign Extender Module.
public static String SingExtender(String s) {
        while (s.length() < 32) {
                if (s.charAt(0) == '1') {
                        s = "1" + s;
                } else {
                        s = "0" + s;
                }
        }
        return s;
}
```

C. Controller Class:

This is the main class where everything happens. It handles the different stages (Fetch, Decode, Execute, Memory and Write Back). Each Stage is handled by a method that carries out all its tasks we have changed in the roles of the methods a little bit. For instance, once fetched the fetch method will pass the Opcode and Type fields to the Control method to set all needed signals in further stage. The decode method will store all needed info from the register file and the instruction in the appropriate fields of the pipeline register. The execute method will use the ALU class to execute the needed operation then store the result in the ALU pipeline register. Memory stage will read or write from Memory Pipeline register and will do that based on Memory flags. Finally, the write-back stage will handle writing back to register or branching or jumping (updating PC). In fact, we have 5 pipeline registers. Each stage has a register that it reads from and write to. This makes stages independent from each other and later the main methods handling the rotation of the contents of the registers before the start of each clock cycle.

D. ALU Class:

This class handles all the needed ALU operation and the result is stored in the appropriate pipeline register. The core method of this class is shown here:



```java
	public static int ALUEvaluator(String Op, int Operand1, int Operand2) {
		int Output = 0;
		String OperationName = "";
		switch (Op) {
		case "0000": // I-AND
			Output = Operand1 & Operand2;
			OperationName = "AND";
			break;
		case "0001": // R-Type OR
			Output = Operand1 | Operand2;
			OperationName = "OR";
			break;
		case "0010": // R-Add
			Output = Operand1 + Operand2;
			OperationName = "add";
			break;
		case "0011": // R-Sub/Beq/Blt
			Output = Operand1 - Operand2;
			OperationName = "sub";
			break;
		case "0100": // R-MultiplyS
			Output = Operand1 * Operand2;
			OperationName = "multiply";
			break;
		case "0101":// R-shift left
			Output = Operand1 << (Operand2 % 32);
			OperationName = "shift left";
			break;
		case "0110":// R-Shift Right
			Output = Operand1 >> (Operand2 % 32);
			OperationName = "shift right";
			break;
		case "0111": // I-Slt
			Output = (Operand1 < Operand2) ? 1 : 0;
			OperationName = "slt";
			break;
		case "1000": // I-Sgt
			Output = (Operand1 >= Operand2) ? 1 : 0;
			OperationName = "sgt";
			break;

		}
		Z = (Output == 0);
		return Output;
	}
```

E. Memory Class:

The memory class handles everything related to memory. From having the memory array and the cache to loading the program before start of execution and updating PC. The memory class has methods that will load the instructions from a text file named code.txt and then it will add it to our array. The class also has methods for updating PC, reading and writing data whilst handling the cache misses. Here is a part from the code that shows the instance attributes and some methods of the memory class:

```java
public class Memory {
        static final int cacheSize = 512;// no. in words;
        static final int memSize = 2048;// no in words;
        static int[] cache, mem, tag;
        static boolean[] valid;
        static boolean[] nonExecute;
        static int PC;
        static int nextInstruction;
        //Checks if there is a valid next instruction.
        public static boolean hasMoreInstruction() {
                return PC<nextInstruction;
        }
        // A function that reads the code from text file and add it to main memory.
        public static int readFile(String name) {
                String Data = "";
                int count = 0;
                File file = new File(name);
                try {
                        Scanner scan = new Scanner(file);
                        while (scan.hasNextLine()) {
                                Data = scan.nextLine().trim();
                                Memory.loadInstruction(Integer.parseUnsignedInt(Data, 2));
                                count++;
                        }
                        scan.close();
                } catch (FileNotFoundException e) {
                        System.out.println(e.getMessage());
                }
                return count;
        }
        //A function to read code from text file.
        public static int readCode() {
                return readFile("code.txt");
        }
```

F. Main Method:

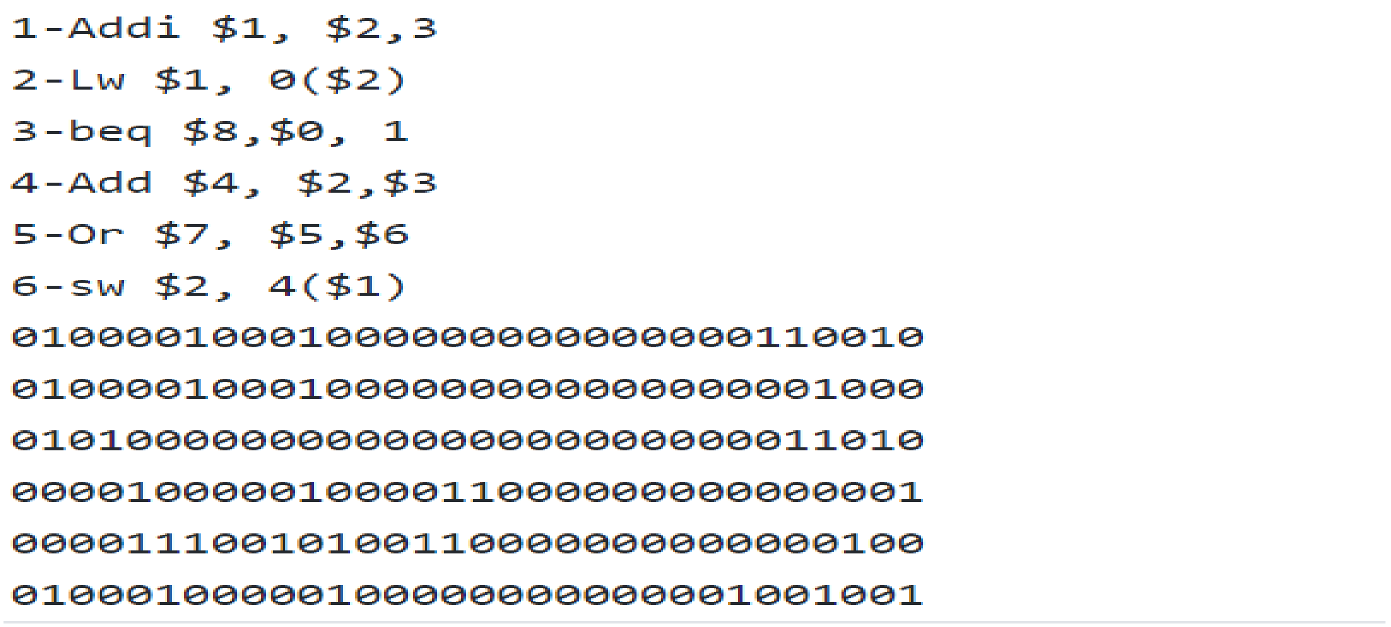
In our main method we handle the update of the cycle counter as well as shifting the content of the registers for instance in a stage we will move content to fetch to decode, decode to execute, execute to memory & memory to write back. We also use the isNull() method to check that pipeline still has content and is not empty. Also, the init() method will initialize all needed variables components before execution starts. Here is the implementation of our main method along with its helpers:

```java
public class Controller {
        static PipeLineRegister pipeline[];
        static int cycle;
        public Controller() {
                pipeline = new PipeLineRegister[5];
        }
        //Main method to simulate the program & control excution.
        //An Empty Cycle is printed out After execution is done.
        //We call this the Cleaning Cycle.It checks that the pipeline is empty & there are not other instructions.
        //The Cleaning Cycle has no effect at all in fact it is a mere check that everything went as expected.

        public static void main(String[] args) {
                init();
                Memory.readCode();
                while (Memory.hasMoreInstruction() || !isNull()) {
                        nextCycle();
                        System.out.println("After clock cycle: " + cycle);
                        System.out.println();
                        for (int i = 4; i >0; i--) {
                                pipeline[i] = pipeline[i - 1];
                        }
                        fetch();
                        decode();
                        execute();
                        Mem();
                        WB();
                }
        }
//Check if there anything in the pipeline before halting exction
public static boolean isNull() {
        for(int i =0;i<pipeline.length;i++) {
                if(pipeline[i]!=null)
                        return false;
        }
        return true;
}
//A function to initialize the components.
        public static void init() {
                new Memory();
                new RegisterFile();
                new Controller();


        }
        //Increment the variable cycle to simulate next cycle.
        public static void nextCycle() {
                cycle++;
        }
```

Running Sample Code & Output Analysis

A. Test Code:

Here is the sample code from the project test file translated into the ISA including some changes to registers to meet our naming with its translation to Binary which is our input format:



```
1-Addi $1, $2,3
2-Lw $1, 0($2)
3-beq $8,$0, 1
4-Add $4, $2,$3
5-Or $7, $5,$6
6-sw $2, 4($1)
01000010001000000000000000110010
01000010001000000000000000001000
01010000000000000000000000011010
00001000001000011000000000000001
00001110010100110000000000000100
01000100000100000000000001001001
```

B. Output samples:

The raw output is attached here as plain text copied from the console for easy tracing.

```
After clock cycle: 1

Fetch stage
Instruction: 01000010001000000000000000110010
next PC: 00000000000000000000000000000001
---------------------------
After clock cycle: 2

Fetch stage
Instruction: 01000010001000000000000000001000
next PC: 00000000000000000000000000000010
---------------------------
Decode stage
Instruction: 01000010001000000000000000110010
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000011
Next PC: 00000000000000000000000000000001
Src1: 00010
Src2: 00000
Dest: 00001
Shift amount: 00000000011
Jump Value: 00000000000000000000000000000000
```

```
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
EX control: ALUsrc-> 1, ALUop-> 0010, shamt->0
---------------------------
After clock cycle: 3

Fetch stage
Instruction: 01010000000000000000000000011010
next PC: 00000000000000000000000000000011
---------------------------
Decode stage
Instruction: 01000010001000000000000000001000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000000
Next PC: 00000000000000000000000000000010
Src1: 00010
Src2: 00000
Dest: 00001
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 1, RegWrite-> 1
Memory control: memRead-> 1, memWrite-> 0, branch-> 0, jump-> 0
EX control: ALUsrc-> 1, ALUop-> 0010, shamt->0
---------------------------
execute stage
Instruction: 01000010001000000000000000110010
Zero flag: 0
Branch Address: 00000000000000000000000000000100
ALU result/addres: 00000000000000000000000000000011
register value to write to memory: 00000000000000000000000000000000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000011
Next PC: 00000000000000000000000000000001
Src1: 00010
Src2: 00000
Dest: 00001
Shift amount: 00000000011
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
--------------------------------------
After clock cycle: 4

Fetch stage
Instruction: 00001000001000011000000000000001
next PC: 00000000000000000000000000000100
---------------------------
Decode stage
Instruction: 01010000000000000000000000011010
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000001
Next PC: 00000000000000000000000000000011
Src1: 00000
Src2: 00000
Dest: 01000
```

```
Shift amount: 00000000001
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
Memory control: memRead-> 0, memWrite-> 0, branch-> 1, jump-> 0
EX control: ALUsrc-> 0, ALUop-> 0011, shamt->0
---------------------------
execute stage
Instruction: 01000010001000000000000000001000
Zero flag: 1
Branch Address: 00000000000000000000000000000010
ALU result/addres: 00000000000000000000000000000000
register value to write to memory: 00000000000000000000000000000000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000000
Next PC: 00000000000000000000000000000010
Src1: 00010
Src2: 00000
Dest: 00001
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 1, RegWrite-> 1
Memory control: memRead-> 1, memWrite-> 0, branch-> 0, jump-> 0
--------------------------------------
Mem stage
Instruction: 01000010001000000000000000110010
ALU result/addres: 00000000000000000000000000000011
Register value to write to memory: do not care
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
------------------------
After clock cycle: 5

Fetch stage
Instruction: 00001110010100110000000000000100
next PC: 00000000000000000000000000000101
---------------------------
Decode stage
Instruction: 00001000001000011000000000000001
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000001100000000000
Next PC: 00000000000000000000000000000100
Src1: 00010
Src2: 00011
Dest: 00100
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
EX control: ALUsrc-> 0, ALUop-> 0010, shamt->0
---------------------------
execute stage
Instruction: 01010000000000000000000000011010
Zero flag: 1
Branch Address: 00000000000000000000000000000100
ALU result/addres: 00000000000000000000000000000000
```

```
register value to write to memory: 00000000000000000000000000000000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000001
Next PC: 00000000000000000000000000000011
Src1: 00000
Src2: 00000
Dest: 01000
Shift amount: 00000000001
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
Memory control: memRead-> 0, memWrite-> 0, branch-> 1, jump-> 0
--------------------------------------
Mem stage
Instruction: 01000010001000000000000000001000
ALU result/addres: 00000000000000000000000000000000
Register value to write to memory: do not care
Memory word read: 01000010001000000000000000110010
Control signals -->
WB control: memToReg-> 1, RegWrite-> 1
------------------------
WB stage
Instruction 01000010001000000000000000110010
----------------------------------
After clock cycle: 6

Fetch stage
Instruction: 01000100000100000000000001001001
next PC: 00000000000000000000000000000110
---------------------------
Decode stage
Instruction: 00001110010100110000000000000100
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000011000000000000
Next PC: 00000000000000000000000000000101
Src1: 00101
Src2: 00110
Dest: 00111
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
EX control: ALUsrc-> 0, ALUop-> 0001, shamt->0
---------------------------
execute stage
Instruction: 00001000001000011000000000000001
Zero flag: 1
Branch Address: 00000000000000000000000000000100
ALU result/addres: 00000000000000000000000000000000
register value to write to memory: 00000000000000000000000000000000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000011000000000000
Next PC: 00000000000000000000000000000100
Src1: 00010
Src2: 00011
Dest: 00100
Shift amount: 00000000000
```

```
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
--------------------------------------
Mem stage
Instruction: 01010000000000000000000000011010
ALU result/addres: 00000000000000000000000000000000
Register value to write to memory: do not care
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
------------------------
WB stage
Instruction 01000010001000000000000000001000
----------------------------------
After clock cycle: 7

Decode stage
Instruction: 01000100000100000000000001001001
Read data 1: 01000010001000000000000000110010
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000100
Next PC: 00000000000000000000000000000110
Src1: 00001
Src2: 00000
Dest: 00010
Shift amount: 00000000100
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
Memory control: memRead-> 0, memWrite-> 1, branch-> 0, jump-> 0
EX control: ALUsrc-> 1, ALUop-> 0010, shamt->0
---------------------------
execute stage
Instruction: 00001110010100110000000000000100
Zero flag: 1
Branch Address: 00000000000000000000000000000101
ALU result/addres: 00000000000000000000000000000000
register value to write to memory: 00000000000000000000000000000000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000001100000000000
Next PC: 00000000000000000000000000000101
Src1: 00101
Src2: 00110
Dest: 00111
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
--------------------------------------
Mem stage
Instruction: 00001000001000011000000000000001
ALU result/addres: 00000000000000000000000000000000
Register value to write to memory: do not care
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
------------------------
```

```
WB stage
Instruction 01010000000000000000000000011010
---------------------------------
After clock cycle: 8

Fetch stage
Instruction: 00001110010100110000000000000100
next PC: 00000000000000000000000000000101
---------------------------
execute stage
Instruction: 01000100000100000000000001001001
Zero flag: 0
Branch Address: 00000000000000000000000000001010
ALU result/addres: 01000010001000000000000000110110
register value to write to memory: 00000000000000000000000000000000
Read data 1: 01000010001000000000000000110010
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000100
Next PC: 00000000000000000000000000000110
Src1: 00001
Src2: 00000
Dest: 00010
Shift amount: 00000000100
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
Memory control: memRead-> 0, memWrite-> 1, branch-> 0, jump-> 0
--------------------------------------
Mem stage
Instruction: 00001110010100110000000000000100
ALU result/addres: 00000000000000000000000000000000
Register value to write to memory: do not care
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
------------------------
WB stage
Instruction 00001000001000011000000000000001
---------------------------------
After clock cycle: 9

Fetch stage
Instruction: 01000100000100000000000001001001
next PC: 00000000000000000000000000000110
---------------------------
Decode stage
Instruction: 00001110010100110000000000000100
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000011000000000000
Next PC: 00000000000000000000000000000101
Src1: 00101
Src2: 00110
Dest: 00111
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
EX control: ALUsrc-> 0, ALUop-> 0001, shamt->0
---------------------------
```

```
Mem stage
Instruction: 01000100000100000000000001001001
ALU result/addres: 01000010001000000000000000110110
Register value to write to memory: 00000000000000000000000000000000
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
------------------------
WB stage
Instruction 00001110010100110000000000000100
----------------------------------
After clock cycle: 10

Decode stage
Instruction: 01000100000100000000000001001001
Read data 1: 01000010001000000000000000110010
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000000000000000100
Next PC: 00000000000000000000000000000110
Src1: 00001
Src2: 00000
Dest: 00010
Shift amount: 00000000100
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
Memory control: memRead-> 0, memWrite-> 1, branch-> 0, jump-> 0
EX control: ALUsrc-> 1, ALUop-> 0010, shamt->0
---------------------------
execute stage
Instruction: 00001110010100110000000000000100
Zero flag: 1
Branch Address: 00000000000000000000000000000101
ALU result/addres: 00000000000000000000000000000000
register value to write to memory: 00000000000000000000000000000000
Read data 1: 00000000000000000000000000000000
Read data 2: 00000000000000000000000000000000
Immediate Value (Sign-Extended): 00000000000000000011000000000000
Next PC: 00000000000000000000000000000101
Src1: 00101
Src2: 00110
Dest: 00111
Shift amount: 00000000000
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
Memory control: memRead-> 0, memWrite-> 0, branch-> 0, jump-> 0
-------------------------------------
WB stage
Instruction 01000100000100000000000001001001
----------------------------------
After clock cycle: 11

execute stage
Instruction: 01000100000100000000000001001001
Zero flag: 0
Branch Address: 00000000000000000000000000001010
ALU result/addres: 01000010001000000000000000110110
register value to write to memory: 00000000000000000000000000000000
Read data 1: 01000010001000000000000000110010
Read data 2: 00000000000000000000000000000000
```

```
Immediate Value (Sign-Extended): 00000000000000000000000000000100
Next PC: 00000000000000000000000000000110
Src1: 00001
Src2: 00000
Dest: 00010
Shift amount: 00000000100
Jump Value: 00000000000000000000000000000000
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
Memory control: memRead-> 0, memWrite-> 1, branch-> 0, jump-> 0
--------------------------------------
Mem stage
Instruction: 00001110010100110000000000000100
ALU result/addres: 00000000000000000000000000000000
Register value to write to memory: do not care
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 1
------------------------
After clock cycle: 12

Mem stage
Instruction: 01000100000100000000000001001001
ALU result/addres: 01000010001000000000000000110110
Register value to write to memory: 00000000000000000000000000000000
Memory word read: do not care
Control signals -->
WB control: memToReg-> 0, RegWrite-> 0
------------------------
WB stage
Instruction 00001110010100110000000000000100
----------------------------------
After clock cycle: 13

WB stage
Instruction 01000100000100000000000001001001
----------------------------------
After clock cycle: 14
```

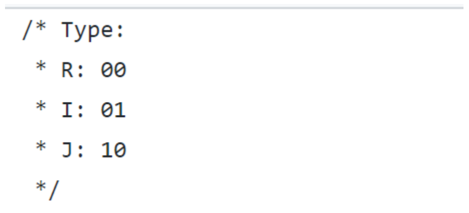
C. Output Analysis:

To ease testing, we have adopted exactly the format in the project output example while keeping our implementation identity. For instance, rd in our output is the destination register always, rs and rt are source 1 and source 2 respectively. The regDest Signal is no longer there since our implementation does not need it. Also notice that “After clock cycle:14” is printed. In fact, the execution took 13 cycle, the printing of that last line is intentional to make sure the execution went peaceful with no errors.

Final Notes

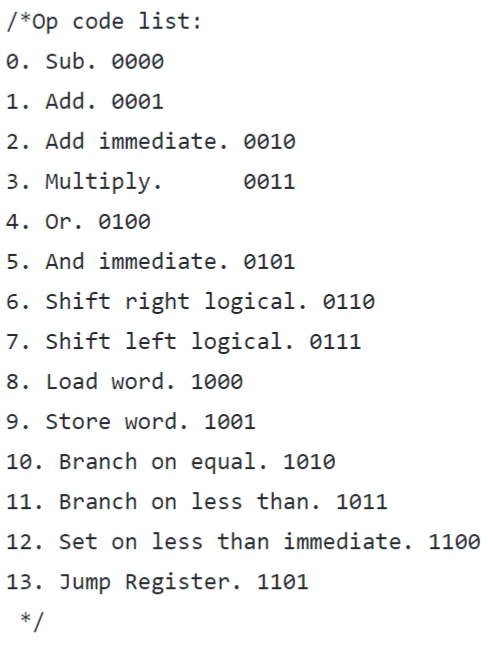
The whole code is commented, and the execution and testing should be very smooth. I am providing here a Cheat sheet on how to construct instructions for test by providing the Opcodes and Type codes of different instructions:

Types:



```
 /* Type:
  * R: 00
  * I: 01
  * J: 10
  */
```

Opcodes:



```
/*Op code list:
0. Sub. 0000
1. Add. 0001
2. Add immediate. 0010
3. Multiply.      0011
4. Or. 0100
5. And immediate. 0101
6. Shift right logical. 0110
7. Shift left logical. 0111
8. Load word. 1000
9. Store word. 1001
10. Branch on equal. 1010
11. Branch on less than. 1011
12. Set on less than immediate. 1100
13. Jump Register. 1101
 */
```

Finally:

The whole test code in the test code section is already loaded, so the submitted project is ready to run directly without having to write the instructions from scratch to save your time.

For any help or enquires please send: ali.abdelfattahkabil@student.guc.edu.eg