**This is our project documentation: that is a document illustrating our design and our approach in implementing this design.**

**First: Our design**

Our instruction is 16-bit long, We classified the instructions into FOUR types (R , I , W ,(C+J)) so we have 2-bits (opCode) to differentiate between them

R-type: Four operations so we have another 2-bits(funct) to determine the operation

ADD -> 00 | SUB -> 01 | MUL -> 10 | AND -> 11

I-type: Four operations so we have another 2-bits(funct) to determine the operation

ADDI -> 00 | ORI -> 01 | SLL -> 10 | SRL -> 11

W-type: Two operations but we reserved 2-bits(funct) to determine the operation so that we have the Regularity feature in our design

LW -> 00 | SW -> 11

C+J TYPE:(For Conditional and Jump operations) Four operations so we have another 2-bits(funct) to determine the operation

BNE -> 00 | BGT -> 01 | SLT -> 10 | Jump -> 11

\_\_\_\_\_

To determine the operation needed to be done by the ALU we determine what instruction would be needed and give a code(ALUOp) for each one and they are:

ADD operation -> 000 , Instructions: ADD,ADDI,LOAD,STORE

SUB operation -> 001 , Instructions: SUB,BNE,SLT,BGT

MUL operation -> 010, Instructions: MUL

OR operation -> 011, Instructions: ORI

AND operation -> 100, Instructions: AND

SHIFT Right operation -> 101, Instructions: SHR

SHIFT Left operation -> 110, Instructions: SHL

\_\_\_\_

The control signals for each instruction are shown in the last table in the other Document named: Design.pdf

**Our design Constraints**

We have 16 register numbered from 0 to 15

First: For immediate, branch , memory instructions the user can only use registers $0,$1,$2,$3

For branch , memory instructions the immediate size is 8 bits

For jump instruction the immediate size is 12 bits

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Second: Our implementation**

We have the CPU.java class that acts as the engine for out project it contains the following:

1) FETCH: an instance from FetchStage class that fetches the next instruction from memory if there is.

2) DECODE: an instance from DecodeStage class that extracts the required information from the instruction that is currently in the decode stage.

3) EXECUTE: an instance from ExecuteStage class that does the appropriate execution operation for the instruction that is currently in the execution stage.

4) MEMACC: an instance from MemoryAccStage class that accesses the memory if the current instruction in the mem access stage needs to do so.

5) WB: an instance from WriteBackStage class that writes the value back if regWrite is set to 1 for the current instruction in the WB stage.

6) memUnit: That is the memory component for the microcontroller (Data and instruction memory (Von Neumannn))

7) pc: the program counter module

8) register file rf : the register file module (16 registers)

First we take the instruction count for the program to be executed second we take the program as input in Assembly language and then we have a class “Assembler” that contains the “convertToMachineCode” method that takes an assembly instruction and converts it to machine code according to our instruction format for each operation and then load the converted program to memory.

**We simulate the pipelining for the five stages by the following approach**

First of all we have an array of size 5, the cell at index 0 carries information about the instruction in fetch stage, the cell at index 1 carries information about the instruction in decode stage, the cell at index 2 carries information about the instruction in execute stage, the cell at index 3carries information about the instruction in memory access stage, the cell at index 4 carries information about the instruction in write back stage so after each cycle the content of the array is shifted one cell to the right and if no instruction in a specific stage then the corresponding cell is NULL

Second we have the pipeline registers:

We have exCurr , exNext are objects that simulates the process of pipeline registers between the decode and execute stage as follows: exCurr -> carries the required info for the current instruction in the ex stage | exNext -> carries the required info for the next instruction to be in the ex stage

We have memCurr , memNext are objects that simulates the process of pipeline registers between the d execute and memAccess stages as follows: memCurr -> carries the required info for the current instruction in the memAcc stage | memNext -> carries the required info for the next instruction to be in the memAcc stage

We have wbCurr , wbNext are objects that simulates the process of pipeline registers between the memAcc and write back stages as follows: wbCurr -> carries the required info for the current instruction in the wb stage | wbNext -> carries the required info for the next instruction to be in the wb stage

**In each cycle (iteration of loop)**

We fetch a new instruction and put it at cell 0, Decode the instruction at cell 1, Execute the instruction at cell 2, access memory for instruction at cell 3, write back the value computed for the instruction at cell 4, the finally shift the instructions one cell to the right and do the following tell no instruction in any of the five stages.

We used word addressing so for example if PC = 12 then it points to the 13th instruction in memory and so it is incremented by 1 after each cycle.