**Simulation of a Non-Pipelined Processor**

**Project Design**

**Overall Design**

The simulation is run by the Main file, which is responsible for reading the configuration file, initializing all the objects (processor components) with respect to configurations, and running the instructions while the program ends while keeping the track of the program counter.

After the configuration file is read, the objects are initialized. The list of objects handled by the main file can be found below. First, instruction memory reads the program input file, converts all the instructions to binary encoding (with the use of ASMParser from Lab 5, with necessary changes made to support all the instructions for this project). Then program counter is initialized. Until program termination, in a while loop, the main class fetches the instruction from instruction memory with respect to current program counter and executes it (simulating all 5-stages using its objects).

In one iteration of the main loop, the process of executing a binary encoded instruction includes:

* (Instruction Decode) Passing the opcode to Control Unit, the Control Unit then updates the control values of other components with respect to this opcode.
* (Instruction Decode) Passing the required bits to Register File and executing read/write operation with the help of MUX1
* (Execute) Taking the 15-0 bits of the instructions and results from the Register File, passing the values and executing the specified operation using ALU3 with the help of ALU Control and MUX5
* (Memory Access) Passing the ALU result and write data to Data Memory object and executing its operation.
* (Writeback) Taking the result from the Data memory and writing data back to registers with the help of MUX4.
* (PC update) Updating the PC with the help of ALU1, ALU2, MUX2 and MUX3 with respect to instruction.

Note that even though all the corresponding functions will be called on each object, it doesn’t mean that their results will be used. For example, an instruction may not need access to Data Memory, in that case even though the function on Data Memory object will be called, the since the Control Unit would change its control value beforehand, the function would do nothing. Similarly, with a jump instruction, although all the functions will be called on the objects, only PC update part will be doing the actual execution of the instruction apart from the decoding done in Control Unit.

**Main**

The driver file of the simulation. Initializes the objects, sets the instruction memory and executes all the 5-stages on the instructions until the program termination.

Fields:

* InstructionMemory instMem
* RegisterFile regFile
* DataMemory dataMem
* ControlUnit cUnit
* ALU alu1, alu2, alu3
* ALUControl aluControl
* MUX mux1, mux2, mux3, mux4, mux5
* int programCounter
* bitset<32> currentInst

Methods:

* int main() - The main function of the program.
* int updatePC(int currentPC) - This is a helper method to calculate the next PC value, makes use of currentInst and cUnit fields.

**class InstructionMemory**

Instruction memory class is responsible of reading the input program, converting it to binary encoding using the ASMParser class and fetching the correct instruction for a given PC.

Private Fields:

* bitset<32> insturctions[100];
* int numInstructions;

Public Methods:

* InstructionMemory() - Default constructor of the class.
* void readInstructions(FILE program\_input) - This method reads the instructions from the program input file and decodes them and places them into the instructions array. It also counts the number of instructions.
* bitset<32> getInstruction(bitset<32> programCounter) - This method gets the instruction indicated by the program counter parameter.
* void printInstructionMemory(FILE output) - This method prints the instructions in the instruction memory in binary to the specified output file.
* bool isFinished(bitset<32> programCounter) - This method checks if all the instructions are run and the program needs to be terminated.

**class RegisterFile**

Register file class stores the registers, and simulates the behaviour of an actual register file with given input, storing the output readData to its fields. It’s operation is controlled by regWrite control value.

Private Fields:

* bitset<32> readData1
* bitset<32> readData2
* bitset<1> regWrite
* bitset<32> registers[32]

Public Methods:

* RegisterFile(FILE input) - The constructor of the class, initializes all the registers with the values from the input file.
* void execute(bitset<5> readRegister1, bitset<5> readRegister2, bitset<5> writeRegister, bitset<32> writeData) - This method simulates the behavior of register file with given parameters according to control value of register file (regWrite).
* void printRegisterFile(FILE output) - This method prints the content of the registers in the register file to the specified output file.
* void printDataAndControlValues(FILE output) - This method prints the read data fields and the regWrite control value to the given output file.
* bitset<32> getReadData1() - Accessor method for readData1
* bitset<32> getReadData2() - Accessor method for readData2
* bitset<32> getRegister(int i) - Acessor method for registers, returns the register numbered i.
* void setRegWrite(bitset<1> c) - Mutator method for regWrite

**class DataMemory**

Data memory class is responsible for storing values that are supposed to be in memory and simulate the behavior of data memory with given input and with respect to its control values memWrite and memRead.

Private Fields:

* bitset<32> memory[100]
* bitset<32> readData
* bitset<1> memWrite
* bitset<1> memRead

Public Methods:

* DataMemory(FILE input) - The constructor of the class, initializes all the values in the memory with the values from the input file.
* void execute(bitset<32> address, bitset<32> writeData) - This method simulates the behavior of the memory with given parameters according to control values of the data memory.
* void printDataMemory(FILE output) - This method prints the content of the memory to the specified output file.
* void printControlValues(FILE output) - This method prints the memRead and memWrite control values to the given output file.
* bitset<32> getReadData() - Accessor method for readData.
* void setMemWrite(bitset<1> c) - Mutator method for control value memWrite.
* void setMemRead(bitset<1> c) - Mutator method for control value memRead.
* bitset<32> getMemValue(int i) - Accessor method for memory values, returns ith value on the memory.

**class ControlUnit**

This class implements the control unit component which is responsible for setting control values of other components to let them execute properly.

Private Fields:

* bitset<1> regDst
* bitset<1>int jump
* bitset<1>int branch
* bitset<1>int memRead
* bitset<1>int memToReg
* bitset<2>int aluOp
* bitset<1>int memWrite
* bitset<1>int aluSrc
* bitset<1>int regWrite
* RegisterFile regFile
* DataMemory dataMem
* ALUControl aluControl
* MUX mux1, mux2, mux3, mux4, mux5

Public Methods

* ControlUnit(RegisterFile regFile, DataMemory dataMem, MUX mux1, MUX mux2, MUX mux3, MUX mux4, MUX 5, ALUControl aluControl)- Base constructor for ControlUnit
* void updateControlValues(bitset<6> opcode)-This function will edit the field values (and corresponding control values of other components) based on the six digit opcode.
* void printControlValues(FILE output) - This method prints the control values to the given output file.

**class ALU**

This class implements the main arithmetic & logic operation component of the simulation.

Private fields:

* bitset<32> ALUresult

Public Methods:

* void execute(bitset<32> data1, bitset<32> data2, bitset<5> ALUcontrol) - This function will simulate an ALU applied to the given information in data1 and data 2. Also the if ALUcontrol holds a value, the ALU will apply what the ALUcontrol function says to do on data1 data 2.
* bitset<32> getALUresult() - This method returns the current ALU result.

**class ALUControl**

This class implements the component that is responsible for controlling the ALU component. It decodes the ALU control value passed from the control unit and then sets the given ALU to do the corresponding operation.

Private fields:

* bitset<2>ALUOp
* bitset<6>opcode

Public methods:

* void setOpcode() - Mutator method for the opcode field.
* void setALUOp() - Mutator method for ALUOp field.
* void setALUControl(ALU alu3) - Sets the control value of given ALU with the current opcode.

**class MUX**

This class implements the multiplexer.

Public fields:

* bitset<1> controlValue
* vector<bool> result

Public methods:

* void execute(vector<bool> input1, vector<bool> input2) - This method chooses one input from two parameters based on the control value and set the variable result to that input.

**Testing**

Except the main file, all components have the same structure with accessor and mutator methods and an execute method. Thus, before bringing them together in the main file, all components’ accessor/mutator/execute methods will be tested individually with some values to see whether expected results are given. One exception to this is the InstructionMemory class, which will be tested with different input program files including all the possible instructions that can be given. Once all components are tested to work successfully, the main file will be tested with each type of instruction where every input to (output from) each component will be checked to be as expected.