Processor Design and Architecture

CSEN 601

A report submitted by:

Abdelrahman Mohamed Gamal - 55-3901, T-17
Adam Adham Abdelfattah - 55-3076, T-9
Ahmed Amr Abolfadl - 55-1221, T-9
Engy Sameh Fouad - 55-1361, T-15
Mariem Hatem Mobarak - 55-1178, T-14
Mohamed Amr Shaker - 55-0903, T-14

May 2024

0.1 Introduction

The purpose of this project is simulating a processor design and architecture with different types of instructions including the R, I, and J formats using C language. To simulate such a process, we were given an instruction table with 12 instructions inside a text file which we had to parse by converting them to binary.For any of these instructions to be implemented, they have to go through the 5 stages of the data paths which are fetch, decode, execute, memory access, and write back with each stage having a specific number of clock cycles to finish.These stages are in a specific pipeline running in parallel according to our project description . Every one of these stages code will be displayed and explained in this report.

0.2 Explanation of the parsing

0.2.1 enum Opcode

enum Opcode is an enumerator used to limit the values of Opcodes that can be declared to the opcodesgiven in the description to ADD,SUB,SLL,SRL,MULI,ADDI,BNE,ANDI,ORI,LW,SW,J.

0.2.2 Opcode getopcode()

Opcode getopcode() is a helper method for readInstructions() used as a parser that parses the string of the opcode part in the text file of instructions into an enumerater Opcode object to be used when executing the instructions, in case an invalid opcode is input it returns -1 which indicates an invalid opcode error.

0.2.3 void readInstructions()

void readInstructions() is the main method that parses the text file of instructions given.

The variable line is an array of chars that was set to a length of 1000 which is an arbitrary value and should be set according to the number of characters in the text file of instructions. The variable token is a pointer to char (in other words a String) that will point to the first char of every word in the text file to be loaded into memory. The variable reg_num is used to store the integer in the text file, for

example if the instruction is ADD R1 R2 R3 then reg_num will be set to each of 1, 2 and 3 temporarily when it's token word(string) is read, it also would read immediate values for example if the instruction is ADDI R1 R2 100 it will also store 100 temporarily when it's word is read, same applies with shamt in shift instructions and address in J instructions. The ptr variable is an optional variable for the function strtol() as it points to the last character of the token which is not needed in our implementation, since we do not need it we chose to set it to null therefore *ptr is a null pointer.

Firstly, the function loads the text file into the file variable. Then it reads the text file line by line so that the function can handle each instruction, for each line the function gets the opcode using the token of the first word as an input to the getOpCode helper method, then we initialized a uint32_t instruction variable that would store our instruction, after that the function determines the type of the instruction using the opcode whether R or I or J type using a switch operator that also determines the binary value of the opcode that is shifted and inserted into the instruction variable.

After determining the type, if it turned out to be of type R, we determine the values of addresses of rd and rs as it is the number after the letter R, however for the 3rd word token if it appears to start with an R it is stored as the rt otherwise it is stored as shift ammount. If it turned out to be of type I, then the first two words which would be stored into rd and rs in the same way as what we have done with the R-type, the last word is stored into the imm variable similar to what we have done with the shamt in the R-type. If it turned out to be of type J, then only the address is stored in it's dedicated variable. After these steps are taken we now have the full instruction in binary which will be stored in the next space in our memoryUnit. A counter is used to keep track of the specific address where we should insert the instruction. If counter is 1023 then instruction memory is full and it breaks since the instruction memory ranges from 0 to 1023.

0.3 Explanation of the 5 Stages

Variables used:

- *memoryUnit: represents the main memory which is of size 2048 (1024 for program instructions and 1024 for the data) and it's allocated in the main method
- PC: register which holds the address of the next instruction to be fetched, and it's initialized with zero
- R[32]: array which represents our 32 registers (31 GPRs, and the R0 register)
- instructions[4]: Array which represents our 4 instructions which is the max that can be done in parallel in our pipeline.
- instructionsStage[4]: Array which holds the stage of each of the 4 instructions we have, and it is initialized with zero
- instructionActive[4]: Boolean array which represents which instructions are active right now (in the pipeline)
- : instructionData: Structure which holds the data of each instruction (Opcode, R1, R2, R3, R1Address, R2Address, etc.)
- instructionDataArray[4]: Array of 4 objects where each object holds the data of a single instruction (Opcode, R1, R2, R3, R1Address, R2Address, etc.)

0.3.1 Fetch (1) and Decode (2)

To begin,we start with the first two stages which are the instruction fetch and instruction decode. Firstly the instruction fetch increments the instruction Stage which is a counter for the stage we are currently at. Subsequently, we retrieve the instructions to be executed from the memory, increment the program counter, and indicate which instruction is now in the fetch phase. Secondly the decode stage is responsible for decoding an instruction given by its index in the instructions array, The code extracts the number of needed bits for each section of the instruction according to the type

of the instruction and stores them in it's field of the instructionDataArray structure. Lastly we increment the instructionStage to show that we are in the second stage.

0.3.2 Execute (3)

In the third stage which is the execute we check the opcode of the instruction using switch cases for all 12 instructions to know which instruction is being executed and cross reference it with the given table of instructions which we parsed to determine what the instruction must do. To achieve this, we manipulate the destination register, two source registers, the immediate and the shift amount as needed. For example, we can add two values to two source registers and store the result in a destination register. For each switch case, we track the execution flow by printing the current and updated values of the relevant registers (R1) and program counter . After execution, we also update the instruction stage counter for the specified instruction index .

Flushing in executing (3.5)

A Flushing function was designed to manage the instruction pipeline when a branch/jump instruction is encountered. When a branch/jump instruction is identified by its index, the function records the clock cycle at which this branch/jump instruction entered. It then loops over all instructions in the pipeline, checking their entry clock cycles. For any instruction that is not the branch/jump instruction itself and entered the pipeline after the branch/jump instruction, the function flushes these instructions by setting their values to -1, resetting their stage to 0, and marking them as inactive. This ensures that any instructions following the branch/jump, which may have been fetched and executed based on incorrect branch prediction, are deleted and do not affect the pipeline.

0.3.3 Memory access (4) and Write back (5)

This memory function is responsible for handling memory operations such as loading from memory and storing to memory. For

LoadWord, It loads the value from memory at the specified memory location into a register R1. It then prints the previous and updated values of the register to track the memory access. For StoreWord, It stores the value from a register R1 into memory at the specified memory location. Moreover, it prints the previous and updated values of the memory location to track the memory write operation. Both cases increment the instruction stage counter for the given instruction index at the end of the memory operation.Lastly,the writeback checks if the instruction is one of the general register operations such as ADD, SUB, MULI, ADDI, ANDI, ORI, SLL, or SRL. If it is, it retrieves the destination register address from instructionDataArray and prints the current value of this register. If the register address is R0, it makes R0 equal 0 to be unchanged, as R0 is reserved as 0. Otherwise, it updates the register with the result from the ALU stored in instructionDataArray and prints the new value. The writeback also checks if the instruction is a PC instruction like a branch/jump and prints the current PC value. If the instruction involves branching and the branch condition is true, it updates the PC to the new address calculated by the ALU and prints the new value of the PC;else the PC remains unchanged. After completing the writeback, the function resets the data related to the instruction in instructionDataArray,setting the opcode and all related fields back to their default values and resets the stage of the instruction to 0 in instructionsStage, thus preparing the system for the next cycle of instructions.