CSEN 601 project report
Package 3: Fillet-O-Neumann with moves on the side

Team 87:

Hamza Hatem (55-1937) T07
Marly Maged (55-2587) T12
Mikhail George (55-1895) T18
Moaaz Samir (55-2296) T18

Supervised by:

Dr. Eng. Catherine M. Elias

May 2024

1 Introduction

The code for this project is written in C. It is meant to simulate a fictional processor design and architecture. The 'Fillet-O-Neumann with moves on the side' design is a Von-Neumann design where data and instructions are stored in the same memory. There are 3 instruction formats:

- R-type instructions: register to register operations such as arithmetic and logical operations
- I-type instructions: instructions dealing with immediate values such as loading and storing data, branching, and immediate arithmetic/logical operations
- J-type instructions: jump instructions

The code applies the logic of the RISC pipeline through 5 stages: Fetch, Decode, Execute, Memory access, Write-back. Depending on the instruction type, the logic of the stages in the pipeline can differ.

2 Methodology

The chosen format for instructions is int. Based on this, to handle a specific part of the instruction, bit-masking and shifting are used. For immediate values, sign extension is used to ensure that negative values are handled.

The memory and the registers (Zero register, general purpose registers (GPRs) & program counter (PC)) are initialised as structs for ease of referencing. This also allows us to keep track of the number of instructions within the memory in said struct. There are various methods to read from and write to both the memory and the registers.

The text file containing the Assembly instructions is first read then encoded into the relevant int representation. The encoded instructions are stored in the instruction memory to be read line by line in the fetch stage. The fetch stage returns the int containing the instruction to be used in the decode stage. During decoding, bit-masking and shifting are used to get the value of the various fields of each instruction. First, the opcode is obtained and checked to know instruction type. Then, based on the type, the needed fields are obtained and returned to be used in the execute stage. In the execute stage, results of the operations are calculated and returned to be used in the memory access and write-back stages.

Control hazards occur when the decision to execute one instruction is reliant on the result of another instruction. This happens in jump instructions (conditional or not). While the jump instruction is being executed, the next two instructions are being decoded and fetched regardless of whether they'll be executed or skipped. If they're going to be skipped, the pipeline is flushed. There are many ways to handle control hazards. The chosen method for this project is stalling.

Regarding pipe-lining, each method associated with the RISC pipeline (fetch, decode, execute, memory access & write-back) returns output to be used by the next stage. This allows us to call the methods based on clock cycles and ensure each stage is being executed in the correct number of clock cycles.

The decision to fetch, decode, execute, memory access or write-back is based on patterns found in clock cycles and whether the instruction has gone through the previous stage or not. For example, instructions are fetched in odd clock cycles. Instructions are decoded during both even and odd cycles given that they have been fetched. They are executed in both even and odd cycles given that they have been decoded and so on. This is done while also taking into consideration that instruction fetch and write-back can not happen in the same clock cycle since they are accessing the same physical memory.

3 Discussion

In one of the first iterations of the code, we were performing the operations of bit-masking and shifting to get certain values as two separate steps. During testing, the bit-masked value would get registered by the compiler as a signed int and messed up some values. This lead to performing both operations as one step/one line of code.

Before working on pipe-lining, the methods in our code were designed to call each other during execution. Upon considering pipe-lining, we decided it would be best to avoid this so that an instruction does not go through the entire RISC pipeline in one clock cycle. Instead, each method returns the input needed for the next stage. The methods are then called in the main method according to the clock cycle.

4 Code explanation

In this section, we will explain all the structures and functions used in the code.

Structure (Registers): This is the structure we created to define the registers used in our project. There is R0 that is constant as the value of this register is always zero and can't be changed. Then, we have an array of ints that has 31 indexes representing the general purpose registers. Then, we have an integer variable pc which represents the pc register.

Function (readGPR): This is the function that we will use when we want to read the value inside one of the registers.

Function (writeGPR): This is the function that we will use when we want to update the value of one of the registers.

Structure (Memory): This is the structure that we created to define the memory of our project. There is an int array of size 2048 which is the number of addresses available in the memory. Then, we have an int variable instructioncount that represents the number of instructions in the memory so far.

Function (initializeMemory): This is the function that we will use when we want to initialize all memory words to 0 at the beginning of the main method.

Function (initializeInstructions): This function initializes the value of the instructions to 0 inside the instructions part in the memory.

Function (initializeData): This function initializes the value of the data to 0 inside the data part of the memory.

Function (readMemory): This function iterates on the memory array until we get to the address we want to read the value from.

Function (writeMemory): This function iterates on all memory address until it gets to the address I want to write a new value in and it update the value in the memory

Function (writeInstruction): This function iterates on instruction part of the memory and adds the instruction we want to add to the memory

Function (writeData): This function iterates on the memory array until we get to the address we want to write the value into.

Function (addInstruction): This function calls writeInstruction function and increments the counter keeping track of the number of instructions into the memory.

Function (registerNumbers): This function extracts the numbers out of a line taken out of the mips text file.
For example: add r1 r2 500, this will extract 1, 2, 500

Function (fetch): This function reads an instruction from the instruction memory and returns the encoded int representation.

Function (decodeHelper): This function checks the opcode of the given instruction and returns instruction type (I, R or J).

Function (decode): This function calls decodeHelper. Based on the instruction type, using bit-masking and shifting it sets the relevant fields of the instruction type.

Function (RinstructionExecute): This function checks the opcode of the given R instruction and returns the result of the operation associated with that opcode.

Function (JinstructionExecute): This function checks the opcode of the given J instruction and returns the result of the operation associated with that opcode. In this case, the only instruction is the jump instruction.

Function (IinstructionExecute): This function checks the opcode of the given I instruction and returns the result of the operation associated with that opcode.

Function (execute): This function checks the opcode of the given instruction. Based on the instruction type, RinstructionExecute or JinstructionExecute or IinstructionExecute is called. The result of performing the operation is returned.

Function (memoryAccess): This function returns the value of move to register instruction and executes the move to memory instruction.

Function (writeBack): This function executes the move immediate function.

Function (encodeInstructions): This function reads the input file line by line and saves the int representation of each instruction in the instruction memory. This is done by checking the instruction type to account for the number of fields needed for it.

Function (pipeline): This function keeps track of the clock cycles and enforces pipe-lining. This is done by checking the clock cycle and calling the stage in the RISC pipe-line that should be executing. A stage can not start before the instruction has passed by the previous stage. This is the basic idea of pipe-lining, but it's also how the code functions (taking a stage's input from the last stage's output). It also handles control hazards. The jumpFlag is set to zero if the instruction is a jump instruction. The following instruction is stalled as needed. The variable tempInstructionNum keeps track of the actual instruction number in the pipe-line. This is needed since in the case of a jump instruction, the pc-1 is no longer the instruction to decode and so on.

Function (printMemory): This function iterates the memory and prints the location and value of all instruction and data.

Function (printRegisters): This function iterates the registers and prints the GPR number and the value it is holding.