ECSE 425 Project

Design of a MIPS Processor

Justin Gelinas-Delisle (260461196), Mete Kemertas (260554854), Eric Liou (260473205), Martin Dorel (260419098), Andreas Brake (260451449)

Department of Electrical Engineering
McGill University
Montreal, Quebec, Canada

***Abstract*—This paper describes the process of designing and testing a fully functional pipelined MIPS processor implemented in VHDL. The system simulates a basic computer operation cycle with the use of five primary instruction stages and was modified with forwarding and branch hardware to allow pipelined execution. An assembler was also designed and used to compile text-based instructions into operable machine code.**

I. Introduction

The most basic functionality of any given computer processor is its ability to execute instruction cycles. These cycles comprise of locating a program instruction in memory, determining suitable action, and processing these actions. In simpler CPUs, instructions are executed sequentially: each operation is completed before the next one is started. Modern day processors are pipelined and contain further hardware to allow parallel execution of instructions.

The aim of this project was to design and implement a fully functional MIPS processor, capable of executing various instructions and to eventually add functionality to pipeline it. In particular, the processor must be able to fetch instructions from a given memory block, and in one or more cycles, successfully complete the operation. Instructions are compiled via an external assembler. This assembler will convert text-based MIPS instruction into operable machine code. Once designed and tested, pipeline functionality will be added, through the implementation of a variety of functions: hazard detection, branching, and branch resolution. ModelSim will be used to implement and simulate all VHDL generated components.

II. Design Overview

A. Assembler

The assembler, written in C, is broken down into three files, namely main.c, dictionary.c, and scheduler.c, which can be associated with a high level splitting of the tasks performed by the assembler. The first file, main.c, reads the input file and parses each line, identifying and isolating assembly components such as opcodes and registers and ignoring comments. These actions determine the type of instruction (which is required to determine which register types if any are used in the instruction) and perform conversions between the ASCII input to a binary output. The primary function of the dictionary file, as its name would suggest, is to perform conversions, specifically between ASCII opcodes to their binary equivalent along with decimal registers to binary values. In addition to this, the dictionary file also contains the logic that handles labels and their usage in jumps and branches. The scheduler file is the most recent and its functions are called after the normal operation of the assembler has completed, reorganizing the machine code to reduce or eliminate stalling in the processor.

B. MIPS Processor

The MIPS processor performs the execution of a single line of MIPS assembly code in 5 clock cycles, each corresponding to the following stages: instruction fetch(IF), instruction decode(ID), execute(EX), memory(MEM) and write-back(WB). A basic block diagram of the processor is shown in Figure 1.

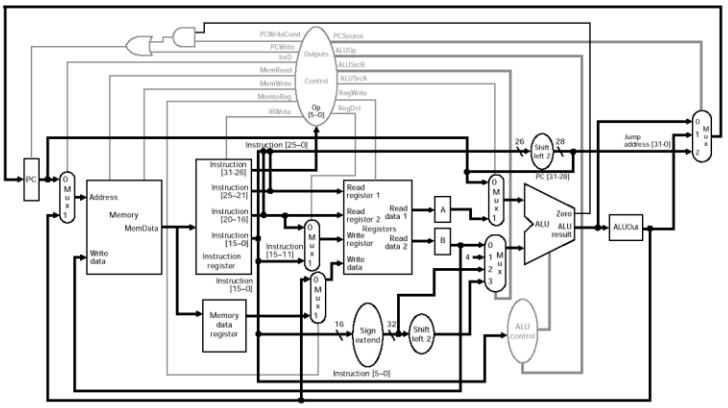

Figure 1: MIPS Processor Schematic [1]

C. Instruction Fetch

The instruction fetch stage uses the Program Counter(PC) value to get the instruction from the instruction memory. The instructions are stored in a Read-Only-Memory(ROM), which is linked to a .mif file holding the output of the assembler. The unit also includes an adder and a multiplexer. Once the execution of an instruction is completed, the adder increments the PC by 4. This value is then fed to the multiplexer, which decides whether to use PC+4 to fetch the next instruction from

the memory or branch out. The select signal for the multiplexer was initially received from the MEM stage and the value of the branch instruction was calculated in the Arithmetic Logic Unit (ALU). However, these were moved to the ID stage for pipelining purposes which will be explained later in the report. The 32 bit instruction received from the memory is fed as an input to the ID stage. A block diagram is provided below in Figure 2.

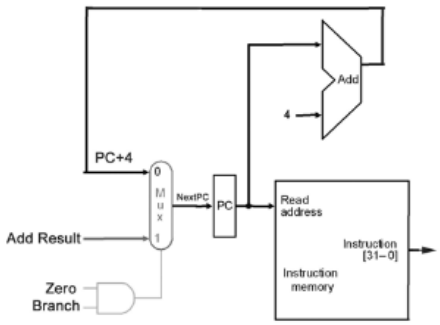

Figure 2: Instruction Fetch Block Diagram [2]

D. Instruction Decode

Instruction decode consists of a register block, a sign extend unit and a multiplexer. The incoming 32-bit instruction from IF is separated into smaller vectors and fed to the register block. The opcode bypasses ID and is fed directly from IF to the main control unit. Bits [25..21] and [20..16] are inputs to Read Register 1 and Read Register 2 consecutively. These two inputs are used commonly by all instructions. Based on control signal reg_dst, the multiplexer selects between bits [20..16] and [15..11] to use as a Write Register input. The Write Register input is used to determine which register to write to. On the other hand, the sign extend unit takes the least significant 16 bits of the instructions and resizes it to 32 bits. The sign extension may be used for sw and ld instructions as an offset. Since the ALU cannot handle 16 bit inputs, the offset is extended to 32 bits. Similarly, it may be used as a branch offset, in which case it must be shifted left by two bits in the EX stage, so that it is properly identified as a word offset. The instruction decode unit is illustrated below.

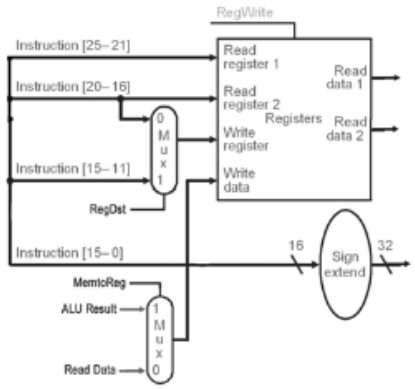

Figure 3: Instruction Decode Block Diagram [2]

E. The Main Control Unit

The control unit takes the Opcode of the current instruction, namely the most significant 6 bits, from IF and feeds several control signals to the rest of the processor. These control signals are reg_dst, jump, branch, branch_ne, mem_to_reg, alu_op, mem_write, mem_read, reg_write, alu_src, word_byte, and if_flush. reg_dst controls which register is written to the register file. Jump and branch select the jump and branch addresses that are sent to the PC. mem_to_reg selects either the data memory output or the ALU result to be written to the register file. alu_op contains three bits that determine which computation the ALU needs to make. The next section will cover alu_op code more in depth. mem_write is asserted when a register value is stored in the data memory during a sw instruction. Similarly mem_read is asserted when a register is loaded with content from the data memory during a lw instruction while reg_write is asserted when the register needs to be written. alu_src is used to select either the sign extend or Read Data to be used as the second input to the ALU. if_flush handles flushing of the instruction fetch. Below is a block diagram of a main control unit, note that the control signals illustrated do not correspond exactly to the ones we have implemented.

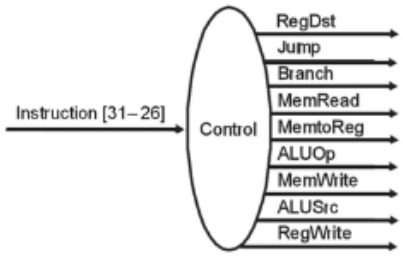

Figure 4: Control Unit [2]

F. Execute

The execution stage is encapsulates the ALU, ALU Control, Shift Left 2 and an adder. The adder and shift_left_2 are incorporated in a single component, while ALU and ALU Control are merged as well. This way the two components may execute in parallel and terminate the entire operation in one clock cycle. The adder is used to add the current PC+4 to the branch offset and the sum is fed to IF, where the address for the next instruction will be determined. The ALU itself takes two input on which it will operate. The first input is the first output of the ID stage, Read Data 1, in every case. The second input may be Read Data 2 from ID or sign-extended address offset to be used for load and store operations. This choice is made in a multiplexer, which takes its select signal from ALU Control. Below is a table of control signals ALU Control unit generates based on the signal ALUOp it receives from the main control unit according to Table 2.

ALUOp	Operation
010	Addition
110	Subtraction
000	And
001	Or
111	Set on less than

Table 1 - ALUOp signal and corresponding ALU operation

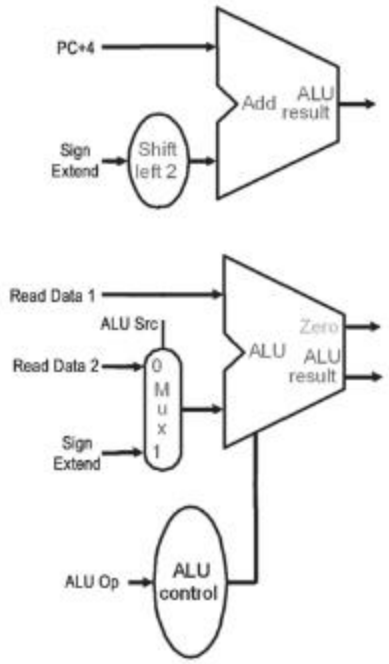

Figure 5: Instruction Execute Block Diagram [2]

G. Memory

The Data Memory unit is modeled with a vhdl block provided by our instructor and teaching assistant. The block is accessed only upon loading or storing operations, in which case the ALU has calculated the target address having its multiplexer selected the offset with ALUsrc signal asserted. As can be seen in Table 1, lw will have the control unit assert MemRead and the requested data will be sent back to registers in ID stage from the output. When sw is being processed, MemWrite will be asserted and the ALU result will determine the index to which the input will be written.

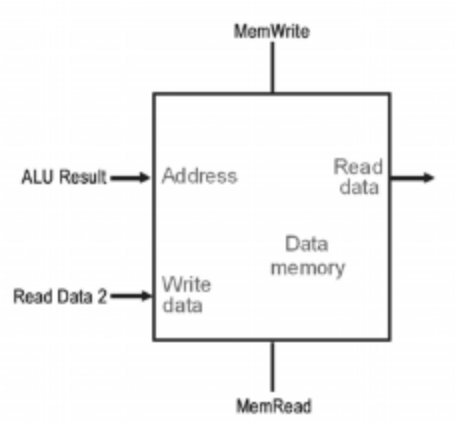

Figure 6: Memory Block Diagram [2]

III. PIPELINED IMPLEMENTATION

In pipelining, multiple instructions can be executed through different stages at the same time. This greatly reduces clock speed and overall performance. To implement the pipeline, control signals were added to simulate the registers needed to generate inputs to the various stages. For instance, when a 32-bit instruction is fetched from memory, the IF/ID registers will store this value as well as the PC+4 counter. On the next clock cycle, or when the instruction moves to the Decode state, these values are extracted from the pipeline register and used accordingly.

A. Hazard Detection

Being able to detect hazards is one of the primary features of pipelining - without it, the processor is unable to determine how to avoid them. The implementation of this feature is done in the control unit. As a primary hub for determining logic, hazard detection was added by storing the source and destination registers of the current and past two instructions. If there are matching registers, this indicates that there is a data hazard: either read-after-write or write-after-write. The control unit then sends out a stall to the Instruction Fetch stage, which stops the PC from incrementing, essentially fetching the same instruction again until the rest of the pipeline is cleared.

B. Forwarding

When pipelining, multiple instructions may cause hazards when requesting access to common variables. One method of reducing data hazards is by forwarding, by passing the output of any of the pipeline stages and into the functional unit (ALU) of future cycles. The implementation used in this processor was done by implementing additional hardware to control the forwarding of data. A forwarding block was implemented in the execution stage; this block primarily checks the status bits (reg_write) from the EX/MEM and MEM/WB registers as well as the source and destination registers of the current and past instructions. Depending on the conditions of these signals, a 2-bit signal is passed to the ALU

multiplexer. The inputs of this multiplexer are either the execution data of the current, previous, or two-instructions behind - the control signal from the forwarding unit essentially passes data values from previous instructions directly to the ALU. The status bits which control the logic in the forwarding unit are generated from the control unit for every instruction; as the current instruction goes through the pipeline, the status registers essentially follow it through the stages.

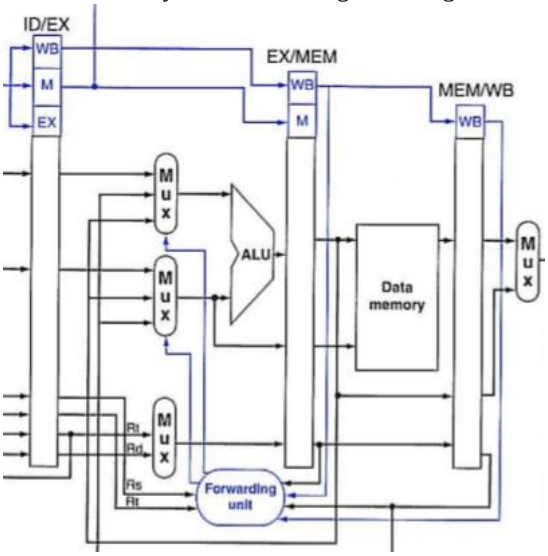

Figure 7: Forwarding Block Diagram [3]

C. Branching

Branch detection was designed in the control unit. The control unit sends a branch signal on any given branch instruction to an AND gate which has another input that compares the data in two registers of the current instruction. This represents the logic needed in branches, to check for different conditions such as not equals, equals, less than, etc. If they are equal, the PC counter is forced to the branch instruction value, and resumes fetching instructions from there.

IV. TESTING

Testing was implemented at several stages during our implementation of a pipelined processor. Each block was thoroughly tested during the initial construction of the MIPS processor. Testing was done in a controlled environment by manually feeding each block example inputs and observing the appropriate outputs were received. Careful attention was given to each block so that they would execute in one clock cycle. Afterwards the blocks were conglomerated and tested as a whole unit. This step was done by using the assembler to create machine code output of a program which was then fed into the system. Testing revealed that the processor works, but bugs within main memory still persist.

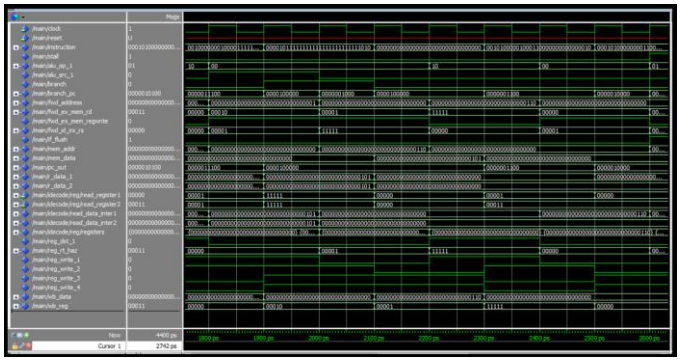

Figure 8: Timing Simulation

V. CONCLUSION

The aim of the project was to implement and optimize a pipelined MIPS processor. An initial design with basic functionalities was constructed. Pipelining was successfully implemented onto that design to improve the processors performance. The pipelined processor was further improved by introducing static instruction in the assembler.

REFERENCES

[1] Fourier.eng.hmc.edu, 'MIPS Processor (Multiple Cycle)', 2015. [Online]. Available: http://fourier.eng.hmc.edu/e85_old/lectures/processor/node6.html. [Accessed: 15- Apr- 2015].

[2] V. Rubio, 'A FPGA Implementation of a MIPS RISC Processor for Computer Architecture Education', Las Cruces, 2004.

[3] Ee.unt.edu, 2015. [Online]. Available: http://ee.unt.edu/public/guturu/MIPS_Pipeline_Hazard_Detection.jpg. [Accessed: 15- Apr- 2015].