Un-Pipelined Processor

ECE 5367 Term Report

Smithson Arrey, Matthew Van-Houten, Kyle Ramsey, Rediet Kebere,

Jorge Martinez, Caleb Rogers, Jiakang Du

ECE 5367

University of Houston

Houston, Texas

*Abstract*—This document presents the implementation of a 32-bit multi-cycle un-pipelined MIPS processor in C++. The design will help to improve the speed and performance of the processor. It has 5 stages of pipeline i.e., instruction fetch, decode, execute, memory access and write back all performed on different clock cycles. An analysis of the performance of the processor from a sample program is presented where particular attention was be paid to the reduction of clock cycles as well as to improve the speed of the processor. The work concludes that the multi-cycle implementation could be as much as 1.27 times faster (for a typical instruction mix) than a monocycle MIPS processor.

Keywords—un-pipelined; MIPS; multi-cycle; RISC; Computer Architecture; University of Houston; ECE 5367;

# **Introduction**

The acronym MIPS (Microprocessor without Interlocked Pipelines Stages) refers to the range of microprocessors developed by MIPS Technologies, with RISC architecture and general-purpose registers of register-register classification, in which most of the instructions do not access memory (except for the load/store instructions) and the processor instructions have two operands, the source and the target. MIPS follows the classic 5-stage RISC pipeline: Instruction Fetch (IF), Instruction/Object Decode (ID), Execute (EX), Memory Access (MEM), and Write (WB). MIPS is a load-store architecture, which means that to do arithmetic on data, values ​​must be explicitly read from memory with a special load instruction and written to memory with a store instruction; arithmetic instructions only operate on registers. MIPS microprocessors have been used for quite some time academically to show students how microprocessors in general work internally. There are many variations of implementations that can be made, ranging from a single-cycle processor to a multi-cycle processor with pipelining and support for floating-point operations.

The goal of this project is to implement and simulate a 32-bit multi-cycle un-pipelined MIPS processor in C++. With this we aim to show how well and fast the multi-cycle MIPS processor can process different instructions under the classic 5-stage RISC pipeline, as well as, to observe the advantages and disadvantages of a unpipelined MIPS compared to a pipelined MIPS.

# **methodology**

## **Defining the Logic**

After having defined the scope of the project we outlined the following objectives:

* To be able to read and load data from input text file
* Capable of decoding instructions in binary
* Execute MIPS’s I and R instructions
* Be able to record and write the processor clock cycles, and final contents of registers and memory into an output file

Before designing our logic for simulating the multi-cycle un-pipelined MIPS processor we needed identify the RISC 5-stages, and the required “components” for the processor:

* A PC (Program Counter)
* An ALU that operates with 32-bit word lengths
* An instruction memory (SW, LW, ADD, BNE, etc.)
* One data memory
* A sign extension unit
* 32 registers of 32-bits
* ALU control

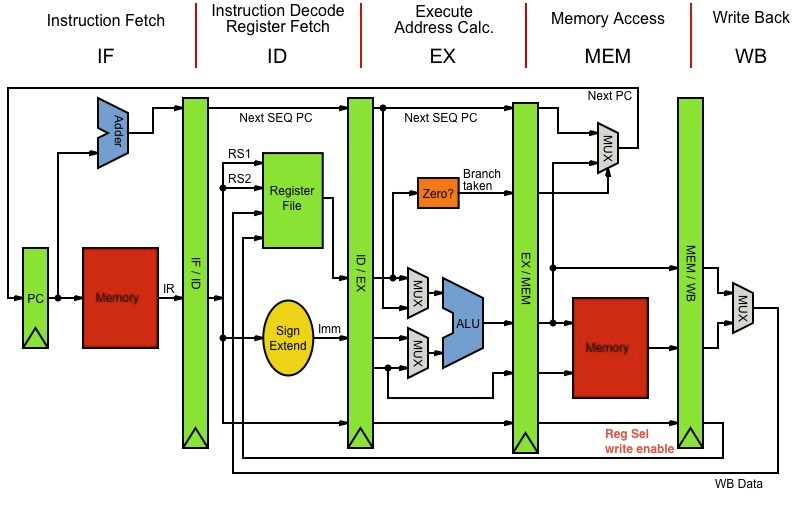


Figure 1: Example of a MIPS architecture processor divided into its 5 classic stages.

The fundamental idea behind the multicycle implementation is to divide the one long cycle of the single cycle implementation into 3 to 5 shorter cycles, depending on the instruction. Pipelining divides the execution of each instruction into several connected stages so that the output of one is the input of the next. As observed in Figure 2 below, a single clock cycle instruction can be broken into 3-5 shorter clock cycles with each cycle corresponding to a different RISC stage. This division into stages makes it possible to considerably reduce the clock period by making that even if a single instruction executes more slowly, the processor is capable of executing a higher number of instructions per second than a unicycle. Therefore, the multi-cycle implementation improves the performance (higher execution speed) due to reducing the average instruction time when there are more than a few instructions that do not need a memory access, or a register write.

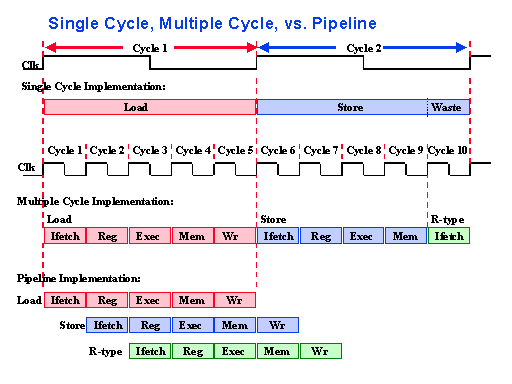


Figure 2: Single-cycle, multi-cycle, pipeline implementations.

Unlike the multi-cycle pipelined implementation, that can execute multiple instructions simultaneously as shown by Figure 2, the multi-cycle un-pipelined MIPS processor implemented in this work executes only one instruction at a time. Therefore, the execution takes more time or a greater number of cycles comparatively. Nevertheless, one advantage of a non-pipelining system is that its efficiency is not dependent on the CPU scheduler. In an un-pipelined processor, the CPU scheduler chooses the instruction from the pool of waiting instructions, when an execution unit gives a signal that it is free.

## **Code**

### **Overview**

Our primary focus with the code was to find the most efficient way to simulate an unpipelined MIPS processor in terms of clock cycle performance, RISC stages, and memory/register storage. Since we were writing the code in C++, this naturally let us to create an object class “MIPS” comprised of the class methods corresponding to the 5 RISC stages: Instruction Fetch (IF), Instruction/Object Decode (ID), Execute (EX), Memory Access (MEM), and Write (WB). This yields an incredibly easy way to manage and access the processor stages (class methods) and the data members (registers and memory). To facilitate the storage of the registers, a 32-bit data word array of size 32 (R0-R31) was created with its index corresponding to the register number. Consequently, for the memory, a struct array (MEMORY) of size 32 was created. The “MEMORY” struct comprises two 32-bit data members: memLocation and memContent, where the data member memLocation corresponds to one of the possible memory locations 0, 4..., 996.

### **Initializations**

The first steps towards simulating the MIPS processor were to first initialize all the register and memory contents to 0, read the contents of a provided input file, convert the input file text to the appropriate data, and store the data into the registers and memory accordingly. The first part of the input file starts with the registers and their content listed as **R# Content** followed by the text word **REGISTERS**. Next, the input file lists a set of memory locations along with their contents after the text word **MEMORY** in the following format **Location Content**. Finally, the input file ends with the word **CODE** followed by a list of 32-bit MIPS instructions in binary. For this last part, an additional array was created for storing each 32-bit instruction in binary string format where the index of the array works as the program counter (PC). To assist with the conversion and storage of the input text file contents to the appropriate, a set of functions were created to convert string number characters and string instructions in binary to integers. Additionally, a set of “MIPS” helper methods (get/set) were implemented to make the tasks of writing to registers/memory and retrieving data from registers/memory easier.

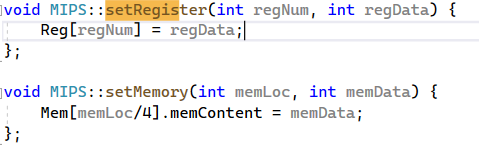


Figure 3: MIPS class set methods.

### **Logic**

With the register and memory initialized and the class methods ready for accessing the data of the registers and memory, we can implement the 5 RISC stages logic into our processor simulation.

1. ***Instruction Fetch (IF)***

The first stage in the pipeline is the instruction fetch (IF). In this stage, the 32-bit binary instruction of the current index (PC) is fetched from the created Code array (memory) and the program counter (PC) is incremented at the end of the cycle (assuming no branches for now). In addition to reading the 32-bit instruction, it is also responsible for sending the current instruction to the next stage in the pipeline.

1. ***Instruction Decode (ID)***

The next stage in our unpipelined implementation is the instruction decode (ID). As it names indicates, this function is responsible for decoding the 32-bit instruction and determining what values the control lines must be set to depending on the instruction type (R or I). See Figure 4 below. In our code, the Decode method splits the first six bits of the 32-bit instruction line (string) determines if the instruction is of R or I type by checking if the opcode (first 6-bits) is equal or not to zero respectively. If the instruction is R type, the method splits and decodes the remaining 26 bits as follows: source register (**rs**), bits 25 to 21; target register (**rt**), bits 20 to 16; destination register (**rd**) bits 15 to 11; shift amount (**shamt**), bits 10 to 6; and function (**funct**), bits 5 to 0. Alternatively, if the instruction is I type, then the remaining 26 bits are split and decoded as follows: source register (**rs**), bits 25 to 21; target register (**rt**), bits 20 to 16; and immediate/offset (**immediate**). This stage/method is also responsible for determining if the instruction requires to carry out the memory access (MEM) and/or register write back (WB) stages following the execution (EX) stage. In the code, the decoded control lines are then stored into different variables and fed to the corresponding components (register bank, sign extender, ALU, etc.) in the ID and EX stages. See Figure 1 above.

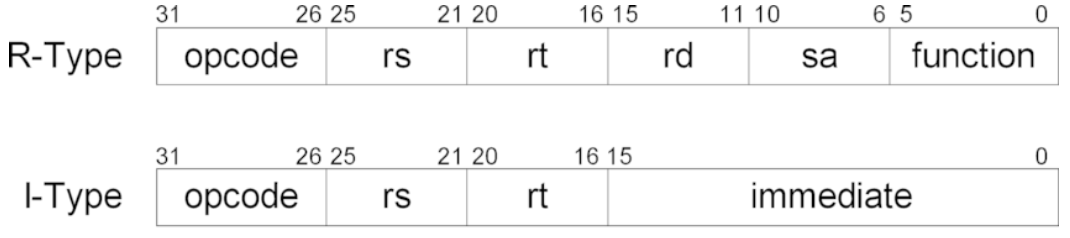


Figure 4: R and I type instruction formats in MIPS

1. ***Execute (EX)***

The third stage in our pipeline is Execute (EX). This stage is responsible for reading the ID/EX pipeline register to get values and control bits, and to perform the arithmetic logic unit (ALU) operation corresponding to the ALUop control line. In case of a branch operation, the branch address is calculated by adding the PC to the sign extended immediate field. Our un-pipelined MIPS simulation can execute a total of 8 different types of instructions: two data transfers: load word (LW), store word (SW); four arithmetic logic operations: addition (ADD), subtraction (SUB), set less than (SLT), immediate addition (ADDI) ; and two control-flow instructions: branch if equal (BEQ), and branch not equal (BNE). In our code, the logic of each instruction is defined outside of the Execute method but are called by it. For the control-flow instructions, the offset/immediate value is added to the PC when the branch condition is true. In case of a data transfer or arithmetic logic instruction, the result of the ALU is then transmitted to the MEM and/or WB stages.

1. ***Memory Access (MEM)***

This stage is only accessed by the load and store instructions. In our code, the load word (LW) instruction asserts the Boolean variable “doMEM” corresponding to the control line MemRead and uses the result of the ALU as an address to index the data memory. The resulting data is then subsequently written into the destination register. For the store word (SW) instruction, in the other hand, the stage unit asserts the Boolean variable “doWB” corresponding to the control line MemWrtite, and writes the content previously read from a register into the destination memory address.

1. ***Write Back (WB)***

Finally, the write back unit stage is executed if asserted by the instruction. This stage is where any computed values by the ALU are written back to their proper registers

### **Write to output**

The final feature of our multi-cycle un-pipelined MIPS consists of writing the timing of the instruction sequence as clock cycles and the final contents of the registers and memory into an output text file. The instruction and clock cycle numbers are immediately written into the file upon completing each unit stage, while the final contents of the register and memory are written at the very end. Upon completion, the user can restart the MIPS simulation with the command “Yes” and enter a new input and output file. Figure 5 below shows the output file after a complete simulation round.

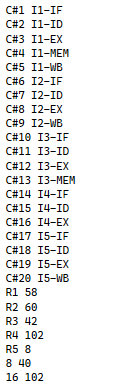


Figure 5: MIPS simulation output.

# **Conclusion**

Our multi-cycle un-pipelined MIPS processor project was completed successful. We wrote a C++ program that takes an input file from the user, reads the file, decodes different instructions in binary, processes the instructions following a MIPS/RISC approach, and writes the simulation results into a given output file. For this project, a total of 250 accessible memory locations were implemented but it could be easily scaled. Similarly, to keep the simulation simple some aspects of MIPS architecture were omitted such as the PC being incremented by one after each instruction (plus offset/ immediate) instead of adding 4 to the PC address, but it can be easily implemented too. The major reason for using arrays for the registers and memory data members instead of other types of data structures is because arrays result in easier calculable addresses which allows faster and simpler access to the element at each specific index.

The simulation results shown in Figure 5 corresponds to the execution of the instruction sequence: LW, ADD, SW, BNE, ADD. Assuming the clock cycle of a single-cycle processor is equal to five clock cycles of a multi-cycle processor, then for the instruction sequence above we would save a total of 5 clock cycles. Hence, by performing the multi-cycle implementation we were able to speed up the processor by a factor of 1.25.

# **Looking Forward and future Ideas**

The multi-cycle un-pipelined MIPS processor implemented in this work can be further improved to execute instructions faster and in fewer clock cycles by implementing complete pipelining. Compared to an un-pipelined MIPS processor, in a pipelined system, multiple instructions are overlapped during execution. Although this improves the overall performance of the processor, it introduces a new problem: hazards. If we consider for example the following instructions in MIPS:

**add $s0, $t0, $t1**

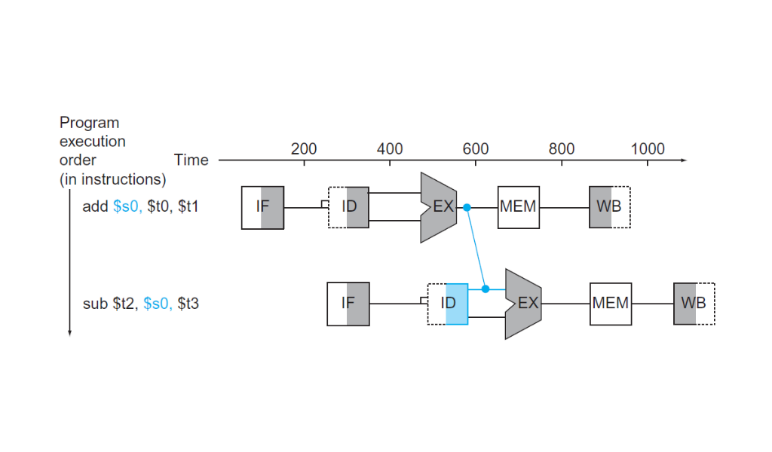
**sub $t2, $s0, $t3**

We see that the sub statement needs the result of the add to execute. The result of add appears at $s0 only in the fifth stage (WB), but the sub needs the correct value of $s0 in its third stage (EX) one clock cycle earlier. This is known as a **Data Hazard** and there are two more types of hazards to consider; structural and control hazards.

**Structural Hazard**: It is when the hardware cannot handle combinations of instructions that must be executed in the same cycle.

**Control Hazard**: They occur because the processor does not "know" enough. When conditional jumps and/or unconditional jumps occur.

To solve these hazards, a hazard detection unit must be implemented in the Decode (ID) stage. The easiest and most efficient way to resolve data risks is by using what is known as forwarding. By adding extra hardware to the CPU, we can pass the result of the ALU at the end of the EX stage to the ALU input for the next instruction.



**Figure 6: MIPS simulation output**

As shown by Figure 6 above, the result of the ADD instruction is passed to the EX stage of the SUB instruction well before the WB stage is executed.

This solution can only be applied if the destination stage is later than the source stage of the value we pass forward. In the diagram above, forwarding occurs from cycle 3 to 4. If the first instruction were a LW $s0, 20($t1) instead of ADD, no forwarding would be possible, because we cannot go back in time. In those cases, the only way to save the difficulty would be to stop the pipeline for one clock cycle, which is known as a pipeline stall.

Finally, a better way to simulate the multi-cycle un-pipelined MIPS processor implemented would be to develop the code in Verilog or VHDL where the clock cycles can be observed better in real time.

##### **Acknowledgment**

1. A special thanks to Dr. Abu Baker for teaching us the fundamentals of MIPS architecture.