Computer Architecture and Mobile Processor

Project 4 – Simple Pipelined MIPS With Cache



Student Name : Jason Melvin Tedjokusumo

Student ID : 32205061

Major : SW 융합모바일시스템공학과

Professor : 유시환교수님

Submission Date : June 20, 2022

Free Days Left : 0 days

* Introduction

This project is the continuation of the 2nd project, moving on from single-cycle architecture of simple MIPS emulator, now we advanced to implement multiple-cycle in MIPS using 5 pipeline stages. Pipeline has marked a significant shift in the design of current microprocessor microarchitecture. Pipelining allows users to get the most out of their hardware by running many instructions at the same time. In 5 stages of pipeline, instruction is processed in 5 distinct stages: IF, ID, EX, MEM, WB. Each stages execute different instructions which follow the order from fetch, decode, execute, memory access (if needed), and write back within multiple clock cycles. In 5 stages pipeline, an instruction is completed within 5 cycles. Since it has multiple cycle execution in pipeline processor, 5 different instructions can be executed at the same time, this increases performance in terms of throughput. Therefore, we can maximize hardware utilization by concurrent execution.

* Important Concept

Each instruction in MIPS must go through five stages: fetch, decode, execute, memory access, and writeback result. Pipelined MIPS works on the concept that each stage takes one cycle. A fresh instruction is retrieved from memory at each clock step. After that, at the end of the instruction execution loop, cycle is updated. There is also temporary storage for each stage, which is referred to as the latch. The input/output values for the following cycle are stored in the latch, which in this case represents a new instruction to be processed in the next stage.

There are flow dependencies that must be managed in order to build an effective pipeline. Because the current instruction must wait for the prior instruction to finish its data read or write, data dependency must be addressed. Control dependency must be managed since it determines the control action based on the prior command. As a result, there are a variety of approaches to dealing with risks or reliance. Stalling, forwarding, and scoreboarding can all be used to alleviate data reliance. Stalling, branch delay slots, or branch prediction can all be used to solve control dependency. Forwarding is used in this project to deal with the data reliance.

What is the procedure for forwarding? It detects the requirement to forward first. It checks for data danger or dependency, then moves on if a certain if statement is met. It will proceed via the if statement if it is connected to ALU results. If the ALU result is stored in the pipeline registers, the ALU result is transmitted to following instructions. The value is grabbed from a specific latch if there is a dependency. Branch prediction is used to handle control dependency. It forecasts whether the branch will be chosen. The prediction was made using loop and if statement branches. The execution stage is where these flow dependencies are set up.

* Unique Considerations for Implementation

A second file called "header.h" is included in this code. The mnemonic of the instructions is defined as its opcode hexadecimal value at the beginning of the "header.h" file. Following that, some struct variables are constructed for usage in the latch implementation. The "header.h" file is included in the main file. The primary code begins with the definition of global variables. Struct types are declared after global variables, and they have three members of an array that is used as a latch. After that, the function variables are declared.

The main function begins on line 35. It begins with the declaration of local variables. It opens the binary input file sequentially and stores the binary input file's instructions in the instruction memory. The binary input file is closed after this step is completed. A while loop is built in line 66 to run through all instructions. Each instruction of the code will be read from memory and decoded, similar to the single cycle MIPS. The instruction is then executed, and memory access is carried out in accordance with the control logic. The writeback function is then used to write the instructions' results. Because it is connected to the pipeline notion, the writeback function is called after the IF function in this code. The latch instructions are then assigned to specified variables. The pc value is next examined; if it is 0xffffffff The counters for cycle and instruction count are continually incremented. The micro-architectural states that have changed are then printed. The final result and additional information of the executed instructions are reported after the while loop is closed. The total number of instructions, memory accesses, register operations, branch instruction count, taken branch count, not taken branch count, and jump instruction count are the additional pieces of information.

The IF function is created underneath the main function to get an instruction from memory and set the pc value. The CS function is then established for the control signal. Its purpose is to use an if-else if statement to set the value of the control signal according to its opcode value. The IDLatchUpdate function is then constructed to update the decode stage's pc and instruction value of the latch.

Following that, the ID function is constructed to decode each instruction based on its opcode value. The opcode and operands are decoded from the instruction. Then, to retrieve the sign extended instantaneous value of the command, an if statement is formed. In addition, the next line of code calculates the jump address's destination. The decode latch is updated in a sequential manner by invoking the IDLatchUpdate method. Following this function, the if-else if statement that runs according to the instruction's opcode is used to initialize the target of the j and jal instructions.

The EXE function is created below the ID function. To begin, the v0 and v1 latch values are stored in variables v0 and v1. The data dependency is then handled utilizing forwarding via the if-else if expression. After dealing with the data dependence, the idex latch value is used to initialize the v0 and v1 variables. Following that, the instruction is executed based on its control signal. The counter for a given variable is incremented during the execution process. To proceed to the next instruction, latch variables are continuously initialized one by one.

For memory access, the MEM function is generated after the EX function. This function is for load and store word instructions that require access to the memory register. Specific code is executed and the latch variables are initialized to continue to the next instruction based on the control signal (memread and memwrite).

Following the MEM function, the WB function is used to write the result using an if-else statement based on the memtoreg or writereg control signal. The regops counter variable will also be incremented if the registry operation is executed.

* Build Configuration / Environment

I am using C programming language in VScode. First of all, you need to install Vscode and mingw. Mingw is a C/C++ toolset, because Vscode doesn’t have a C compiler. After installing VScode, you have to download several extensions, which is C/C++ provided by Microsoft and its extensions and Code runner to run codes.

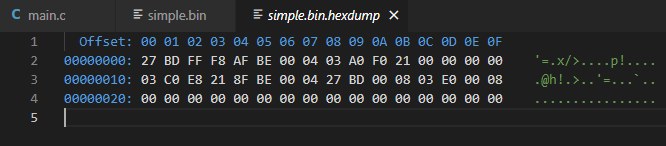
Sometimes when you want to use scanf() function, you need to insert something to the terminal, and we can use run in terminal to give the output in the terminal. Click on File > Preference > Settings > Extensions > Run Code Configuration > scroll down to Run in Terminal and check it.

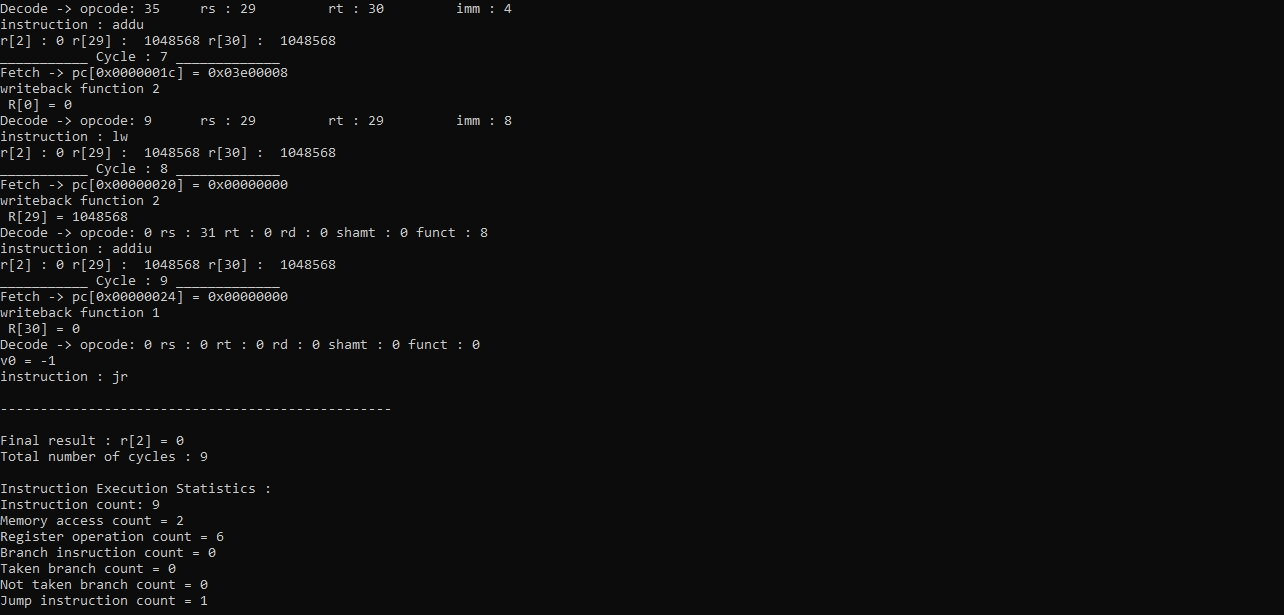
Because we will handle binary input file, so we have to install hexdump for Vscode in the extensions. This extension could show the binary input (which is impossible for human to read) in a unique hexadecimal format.

* Working Proofs

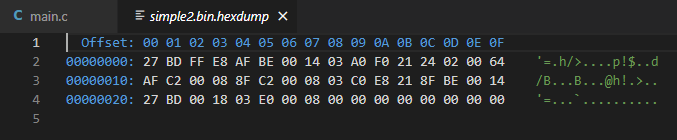
Below are the binary input files showed in hexadecimal dump file and the results of each binary input file. Because the results and some input binary files are too long, I will crop some of it and present the final results and print outputs.

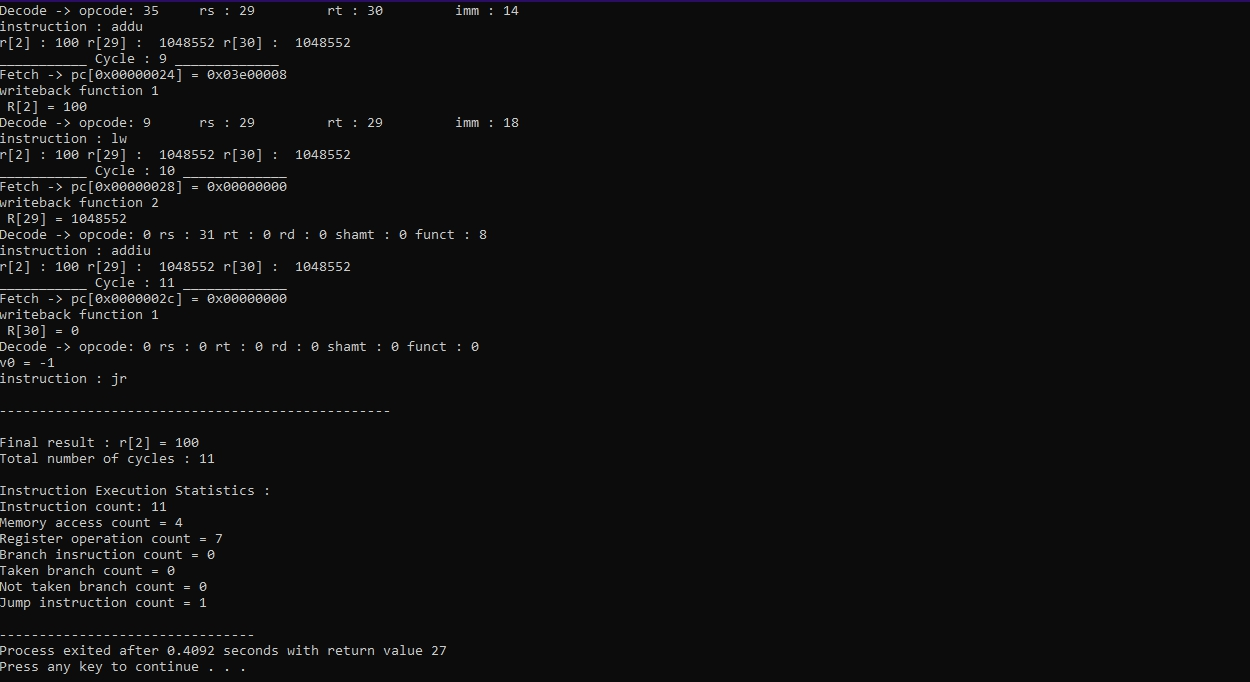
*Simple.bin*

**

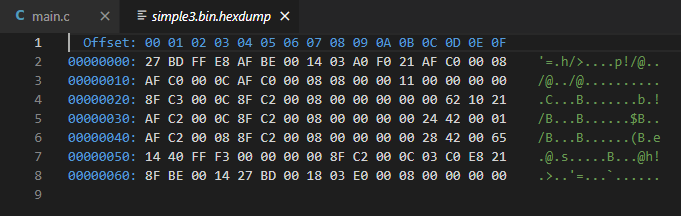


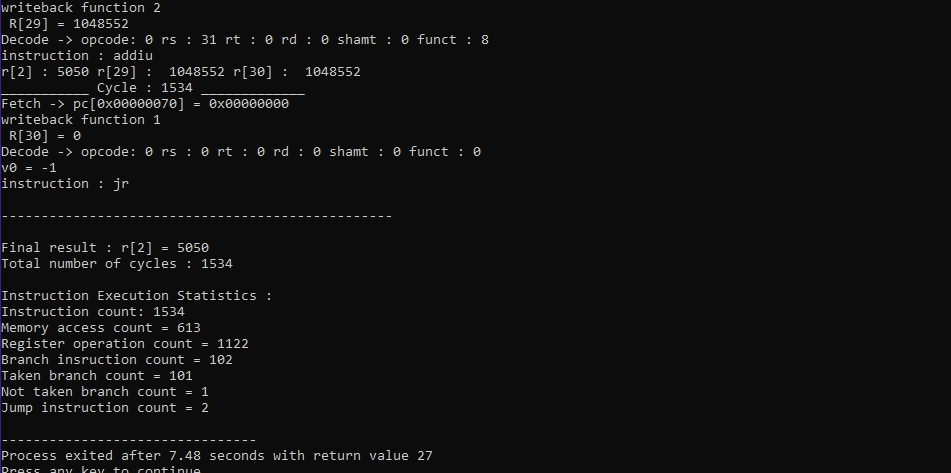
*Simple2.bin*



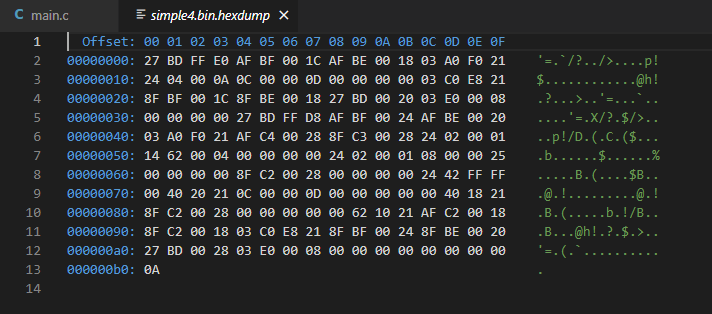


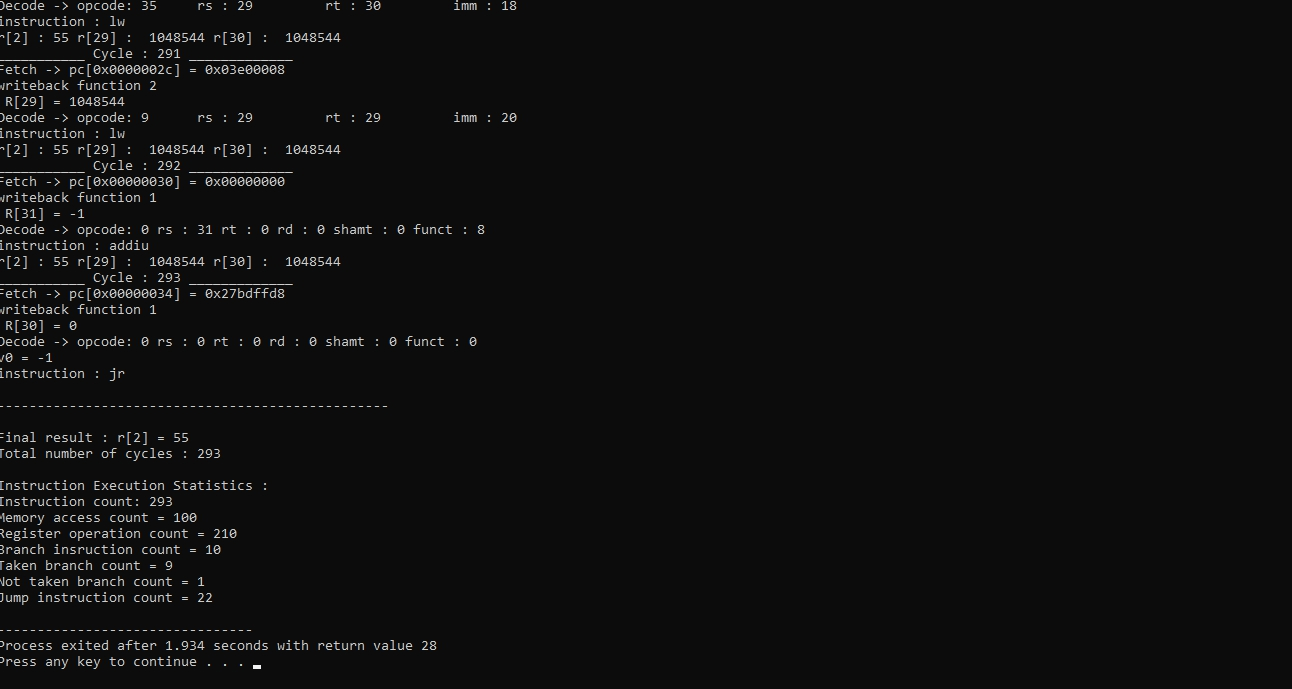
*Simple3.bin*

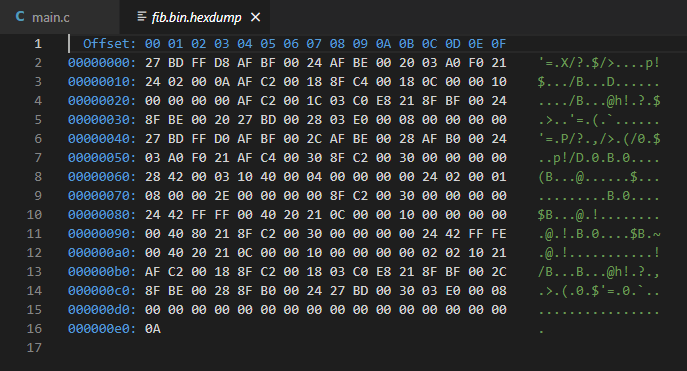


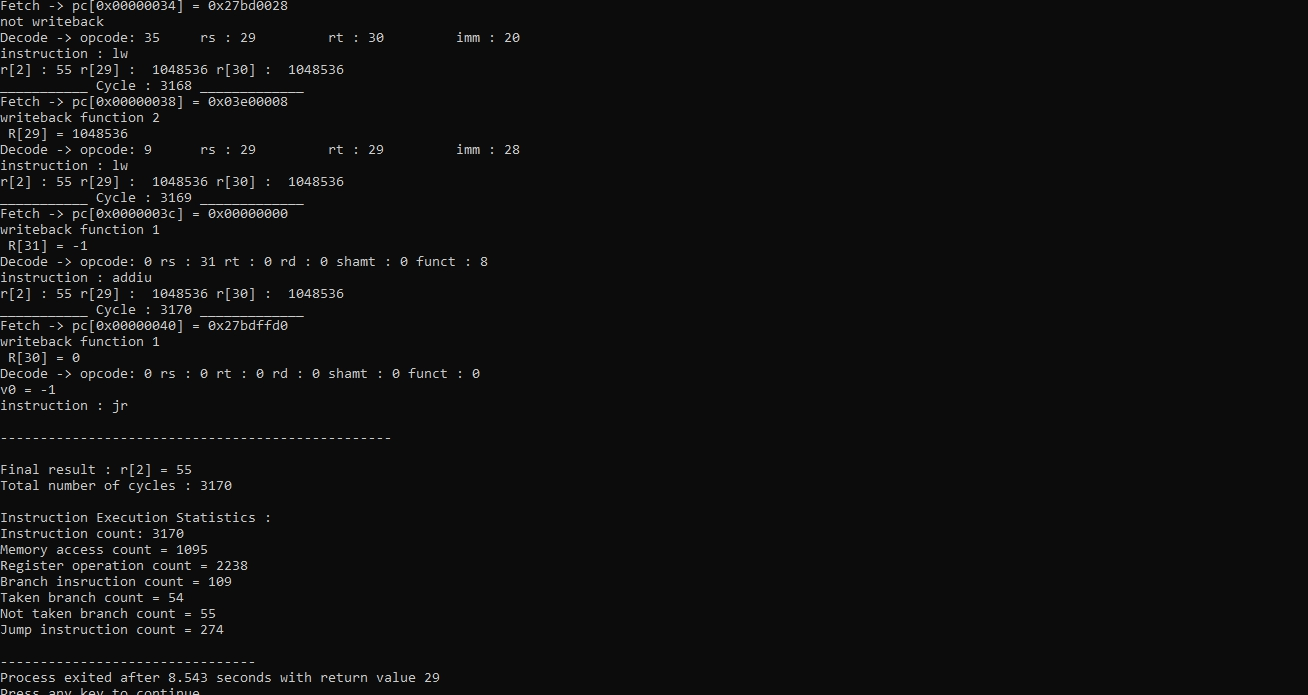


*Simple4.bin*

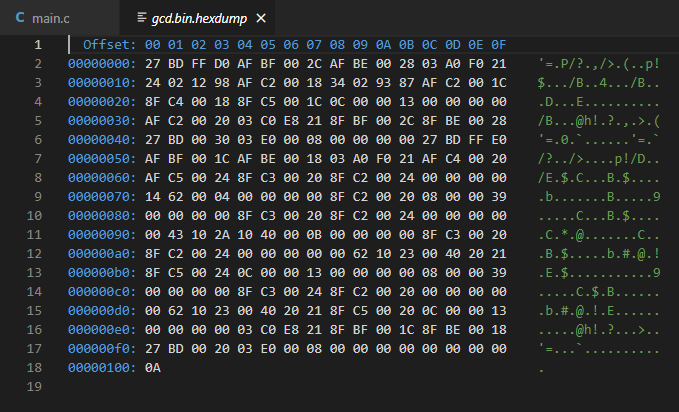
**

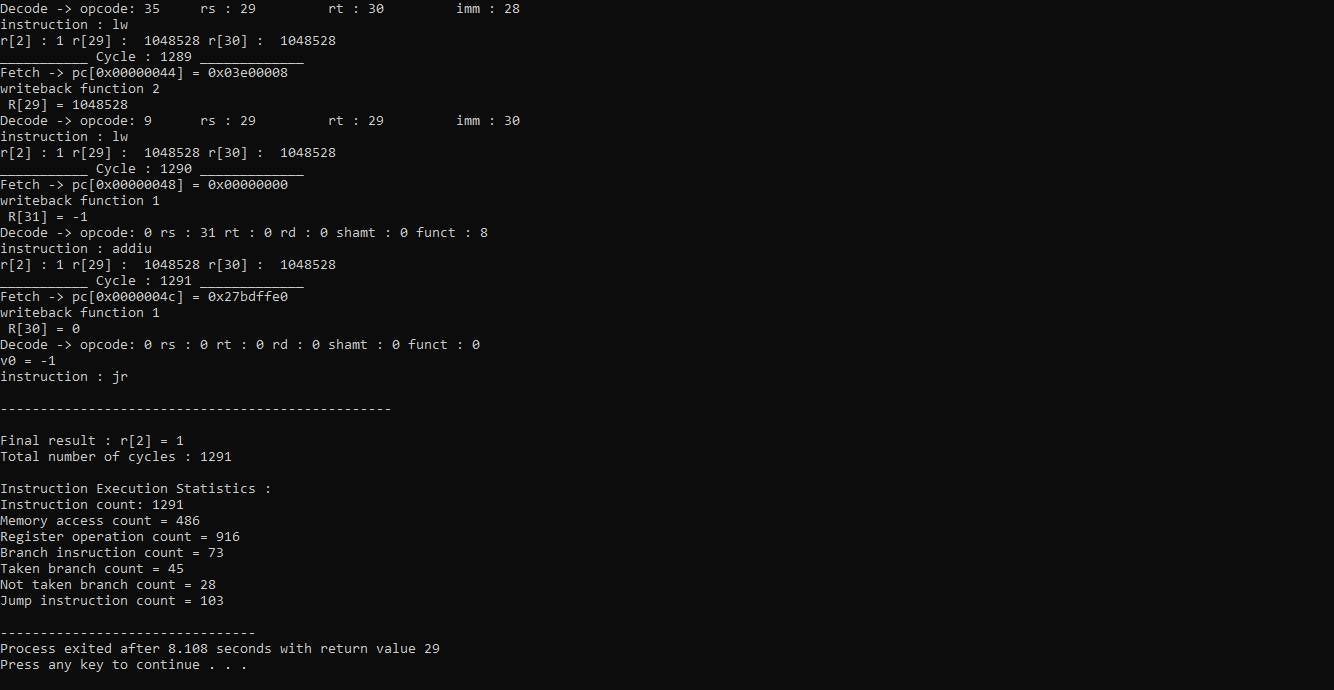


*Fib.bin*



*Gcd.bin*

**



* Trial & Errors

There is no end of problems in this project, in terms of pipeline structure, I think it’s still good thanks to single cycle project. However, in this project, implementing pipeline MIPS is harder, firstly because of the hazards, which are data dependency and control dependency. While working on data dependency problem, I always encounter multiple error, I think its because of the defect in my plan or structure for making the program to solve data dependency. There is always other scenarios when my program won’t work. And finding out what is missing is the hardest and time consuming one. Same goes to when dealing with control dependency, where we need to control how the branch works.

* Final Thoughts

At first, I thought that this continuation of the 2nd project, which now is 3rd project of pipeline MIPS would be absolutely worth for my experience and future goals. However, I feel very dejected when I encounter lots of problems that I couldn’t handle. I started to wonder, will I be able to pursue what I want or was this major which I picked is not fitted for me. I truly feel this way; but, thankfully I could finish this project. I hope that I won’t encounter this kind of work in my future, tho it is a good experience to know about microprocessor inner working systems.