Final Project – Microprocessor without Interlocked Pipelined Stages (MIPS)

Misael Lopez, *Student, University of Washington*

*Abstract*—This document presents the design, implementation, and results of the final project for EEP 598 Digital Design with FPGAs.

Keywords—Field Programmable Gate Array (FPGA), Register, System Verilog, Combinational Logic, Computer Processing Unit (CPU), Sequential, Logic, Read Only Memory (ROM), Random Access Memory (RAM), Control Unit, Data Path, Microprocessor without Interlocked Pipelined Stages (MIPS) (key words)

# Introduction

The final project for EEP 598 Digital Design with FPGAs requires all concepts seen throughout the class (combinational logic, sequential logic, memory access, etc.) to be applied in one system and loaded onto an FPGA platform. For this reason, a Microprocessor without Interlocked Pipelined Stages (MIPS) that can implement a Fibonacci Sequence was chosen as a final project.

This document presents the design, implementation, and results of utilizing SystemVerilog to design and deploy the MIPS microarchitecture running the Fibonacci sequence on a DE1\_SoC FPGA board. First, a MIPS architecture overview is provided. Then, the design methodology is presented which was used to determine the minimum instruction set needed and describes the main modules coded in SystemVerilog. Finally, the simulation and test results are shown, proving the functionality of the project.

# MIPS Overview

MIPS is an architecture developed by John Hennessy [1] and is commonly utilized by major companies. The following top-level diagram shows the main components of a typical MIPS microarchitecture [2]:

Diagram

Description automatically generated

1. MIPS Top Level Microarchitecture.

For this project, the architectural state is 32-bits and utilizes 32 registers (also known as the Register File). The following MIPS instruction subset is considered for this implementation:

* R-type Instructions
* Branches

# Design

## High Level Fibonacci Implementation

To implement the Fibonacci sequence that returns the Nth number given by a user, the following flow chart was developed:

Diagram

Description automatically generated

1. Fibonacci Sequence Flow Chart.

This flow chart determines that five variables are needed to return the Nth number of the sequence:

* n: user input dictating the Nth number to return from the sequence
* cnt: counter variable that keeps track of which Fibonacci number is being calculated
* a: N-1 Fibonacci number in the sequence
* b: N-2 Fibonacci Number in the sequence
* t: temporary variable used to calculate the next number in the sequence (*e.g.,* a = b + t).
* result: stores the number for the Fibonacci sequence to return.

## Registers and Instruction Set

Using the variables defined above, the following MIPS registers are assigned to each variable:

1. MIPS Registers Utilized

|  |  |  |
| --- | --- | --- |
| **Register** | **Register Type** | **Variable** |
| $a0 | Argument | n |
| $t1 | Temporary | cnt |
| $s0 | Saved | b |
| $s1 | Saved | a |
| $t0 | Temporary | t |
| $v0 | Return Value | result |

[insert table]

To follow the flow chart, the following MIPS instruction subset is chosen:

* ADD: add two registers and store in result register.
* ADDI: add immediate contents to register and store in result register.
* BEQ: branch to address if registers are equal.
* J: jump to address.

## Module: Program Counter

The Program Counter module is a 32-bit counter that starts at value 0x00000000h and counts to 0xFFFFFFFFh. Its primary purpose is to contain the address of the instruction to be executed in the instruction memory. While the first instruction to be executed in a MIPS architecture is located at address 0x00400004h, address 0x00000000h is used as the first instruction address for simplicity.

## Module: Instruction Memory

The Instruction Memory module is a RAM module containing 64 addressable 32-bit instructions. The instruction to load is fetched by the Program Counter, and the Instruction Memory module outputs the instruction to execute.

## Module: Register File

The Register File module contains a collection of 32 32-bit registers (as defined by the MIPS architecture). There are two address ports to read two registers, and one addressable write port to store results and/or values in a register. For this project, an additional read port has been added to examine the contents of all registers in real time on the DE1\_SoC board.

## Module: Sign Extend

The Sign Extend module receives instruction bits [15:0] and extends the sign of the 16-bit number of the word to be loaded. This module is primarily used for the Load Word instruction.

## Module: Arithmetic Logic Unit (ALU)

The ALU module takes two 32-bit source operands and executes one of the following operations based on an ALU control signal (ALUControl):

* Add, A + B
* Subtract, A - B
* And, A & B
* Or, A ^ B
* Set Less Than (If A < B, 1, else 0)

The output result is based on one of the operations executed above. There is also a Zero flag set when the resulting operation results in a 0. Additionally, this project adds a carry out flag from the ALU.

## Module: Data Memory

The Data Memory module is exactly like the Instruction Memory module with additional functionality to load data onto the Memory Module using a Write Data port.

## Module: Mux 2

Various Mux 2 modules are used in the MIPS architecture. The Mux 2 module is a 32-bit selector that selects between two inputs based on a 1-bit control signal.

Three Mux 2 modules are implemented directly in the design of this processor:

* Selecting between Sign Extend and Register File Data Port 2.
* Selecting Register File register destination.
* Selecting between ALU result and Data Memory output.

For the rest of the Mux 2 modules, the ternary operator “?” is used a shorthand method to select the next value for the Program Counter module.

## Module: Control Unit

The Control Unit module is a combination of two smaller modules: the Main Decoder module and the ALU Decoder module.

The Main Decoder Module uses the Opcode (bits [31:26]) to determine which registers to write to and which Mux 2s to set to 0 or 1 to fulfill the instruction set mentioned previously. The Main Decoder module has the following outputs:

* MemtoReg: selects Data Memory module output to write to register file.
* MemWrite: enables write for Data Memory module.
* Branch: utilizing the Zero bit from the ALU, sets the next Program Counter to a branch value if a branch condition is met.
* ALUControl: 3-bit output that dictates what operation the ALU should execute.
* ALUSrc: selects between either the Register File or Sign Extend result as Source B operand for the ALU.
* RegDst: selects between the instruction register write destination.
* RegWrite: enables write for the Register File.
* Jump: selects the next Program Counter value to be a jump address.

The ALU Decoder module utilizes the instruction’s Function field (bits [5:0]) and the Main Decoder’s ALUOp output to instruct the ALU which operation to execute.

# Simulation

## Assembly Code to Machine Code

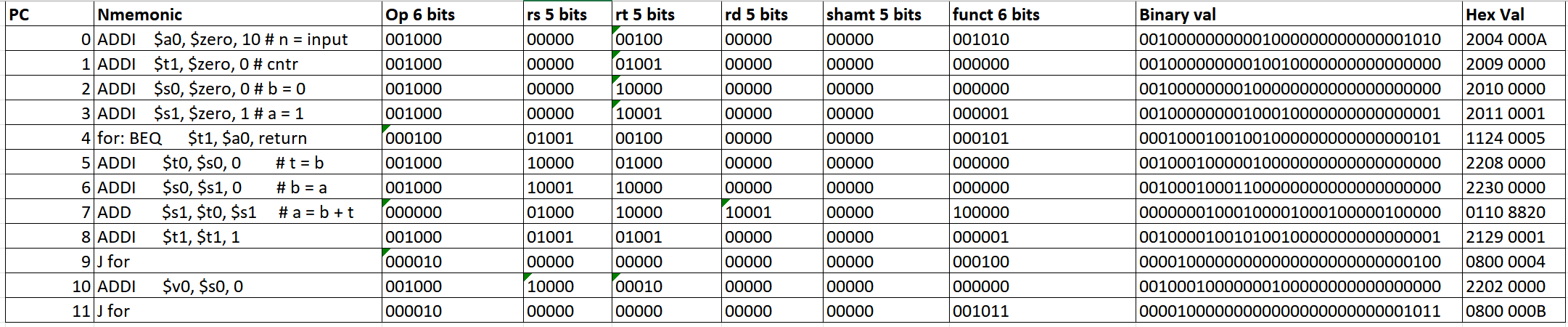
To evaluate whether or not the minimum MIPS instructions chosen can execute the Fibonacci sequence, an Assembly Language simulator was used [3]. The following code was run in the simulator to prove the minimum instruction set works:

A picture containing scatter chart

Description automatically generated

1. Fibonacci Sequence Assembly Code

After evaluating, the following machine as run in the simulator to prove the minimum instruction set works, the following machine code developed from the instruction set:



1. Fibonacci Machine Instruction Code

## ModelSim Results

The MIPS microarchitecture was simulated using ModelSim. Using the $display function, the Fibonacci sequence was tested and proven, producing the following results:

A picture containing text

Description automatically generated

1. MIPS Initial Register File.

Text

Description automatically generated

1. MIPS Fibonacci sequence in progress.

Text

Description automatically generated

1. MIPS Fibonacci sequence completed, result stored in Register V0.

The Wave display for the sequence can be seen in the following images:

A picture containing diagram

Description automatically generated

1. MIPS Fibonacci sequence, initial clock cycle.

A picture containing chart

Description automatically generated

1. MIPS Fibonacci sequence completed, result stored in Register $v0.

In the Wave images, it is important to note that sw\_in represents a DE1\_SoC switch input used to read the contents of the Register File. The output rfile\_rd3 represents the output of the contents pointed to by sw\_in. In this case, because the result is stored in register $v0, sw\_in is set to 00010b (2 dec).

# Results

The SystemVerilog code is provided in the project submission. The MIPS architecture was loaded onto a DE1\_SoC board to prove functionality. Switches SW[4:0] are utilized to inspect the contents of the MIPS register file. Hex Displays 0 to 5 are utilized to display the contents of the register in hexadecimal.

The processor was presented to Professor Sep Makhsous and teaching assistant Max McKelvey. Utilizing cases of n = 15 and n = 16, the following results where produced:

1. DE1\_SoC MIPS Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n** | **a** | **b** | **t** | **result** |
| 15 | 987 | 610 | 377 | 610 |
| 16 | 1597 | 987 | 610 | 987 |

# References

1. Harris, David Money and Harris, Sarah L., Digital Design and Computer Architecture, 2nd ed. Morgan Kaufmann, 2013.
2. H. Huang, “A single-cycle MIPS processor,” p. 25.
3. “WeMips: Online Mips Emulator.” https://rivoire.cs.sonoma.edu/cs351/wemips/ (accessed Dec. 09, 2022).