Miguel Angel Triana Camacho

Single Cycle MIPS Implementation

August 28th, 2024

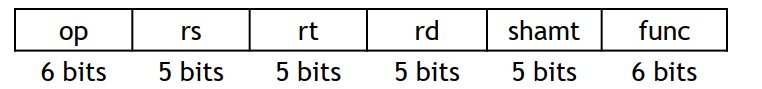
**Introduction**

The MIPS (Microprocessor without Interlocked Pipelined Stages) architecture is a family of RISC (Reduced Instruction Set) processor developed by MIPS technologies. This family consists of several versions of the MIPS processor (MIPS I, II, III, IV, V) and both 32 and 64 bits of width - the MIPS II was actually used in the PlayStation 2. This is a really simple core architecture, that is capable of doing complicated algorithms. This architecture also handles several extensions (optional), like FP based ones (MIPS-3D, MDMX).

\*Disclaimer: The word “core” and “CPU” are used interchangeably in this report.

MIPS uses a really simple instruction format, which consists on 3 types of instructions:

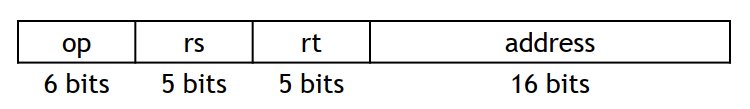
R-Type:



Used for most common arithmetic operations:

* op: operation code​
* rs, rt: first and second source regs​
* rd: destination reg​
* shamt: only for shifting​
* func: used for arithmetic functions

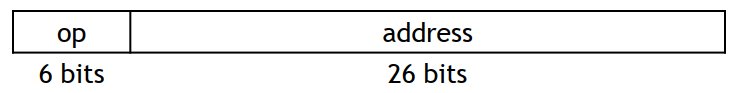
I – Type:



Used for immediate operations:

* op: operation code​
* rs: source reg​
* rt: destination reg (source for branches)​
* address: two’s complement value 16 bits

J – Type



* op: operation code​
* address: positive only value of 26 bits

Used for jump operations.

**Implementation:**

Single-cycle MIPS implementation.

Diagram, engineering drawing

Description automatically generated

Figure 1: MIPS Arch Implementation

This is a single cycle core, meaning that all instructions take one cycle to be completely executed. There are drawbacks from this implementation such as maximum frequency that are out of the scope of this report.

The architecture of this core is a Harvard Architecture, which means that we have independent buses for the instruction memory, and the for the data memory.

As a RISC core, this is also a register-based architecture, which means that all the data that will be executed, needs to be available in the register bank, for the ALU to process it. This also includes data for changing memory location. There is no data going

**Magic Under the Hood:**

Basically, the way this core works can be seen generally from left to right basing yourself in the figure 1.

We have a Program Counter (PC) that is counting the addresses of the Instruction Memory which stores the program that is executing. For this implementation this is a ROM memory.

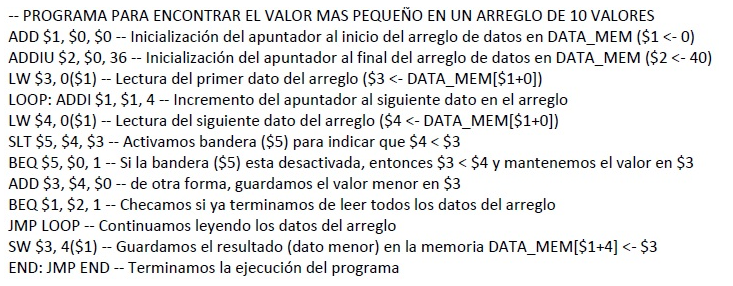
This PC sends the reading address to the Instruction Memory, which then sends the instruction to the decoder. In this arch the decoder is a mixture of elements that include the Control Logic, and the splitting of the instruction in different busses that go to the register bank, and the sign extend module.

The control logic is in charge of enabling/disabling the rest of the components of our core. These are:

* Register bank: This module stores the data for the ALU to process it. Really important due to the register-based architecture. It has two read ports, and one write port.
* ALU: This module is in charge of the arithmetic operations of the core. It is the processing unit. Has two source ports and a destination (or result) port. This ALU supports (ALU supports more operations than the actual core implementation):
  + Add
  + Subtract
  + Shift Left
  + Set Lower Than
  + Absolute Value
  + Xor
  + Shift Right
    - Logical
    - Arithmetic
  + Or
  + And
* Data Memory: This module stores all the data. It has one read port, and one write port.
* Misc Muxes: These are in charge of the data-path. These control where is the data going, there is one in the input of the write address port from the register bank, that selects where to take the address from in the instruction (immediate places), another one is in the source of the second port of the ALU to choose data from either the register bank, or immediate from the instruction. There are several in the PC path that choose the next address to access, these are in charge of branching and jumping. And finally, there is one in the output of the data memory, to choose the actual data going into the register bank, from the memory, or from the ALU.

Code Execution:

Algorithm:



Actual ASM code:

add $1, $0, $0

addi $2, $0, 36

lw $3, 0($1)

LOOP:

addi $1, $1, 4

lw $4, 0($1)

slt $5, $4, $3

beq $5, $0, SLT

add $3, $4, $0

SLT:

beq $1, $2, EXIT

j LOOP

EXIT:

sw $3, 4($1)

END:

addi $0, $0, 0

j END

Coded the memory to values from 0xA to 0x1:

//STORED PROGRAM

  uut.InstructionMemory.regData[0]  = 32'h00000000;

  uut.InstructionMemory.regData[1]  = 32'h00000820;

  uut.InstructionMemory.regData[2]  = 32'h20020024;

  uut.InstructionMemory.regData[3]  = 32'h8c230000;

  uut.InstructionMemory.regData[4]  = 32'h20210004;

  uut.InstructionMemory.regData[5]  = 32'h8c240000;

  uut.InstructionMemory.regData[6]  = 32'h0083282a;

  uut.InstructionMemory.regData[7]  = 32'h10a00001;

  uut.InstructionMemory.regData[8]  = 32'h00801820;

  uut.InstructionMemory.regData[9]  = 32'h10220001;

  uut.InstructionMemory.regData[10] = 32'h08100004;

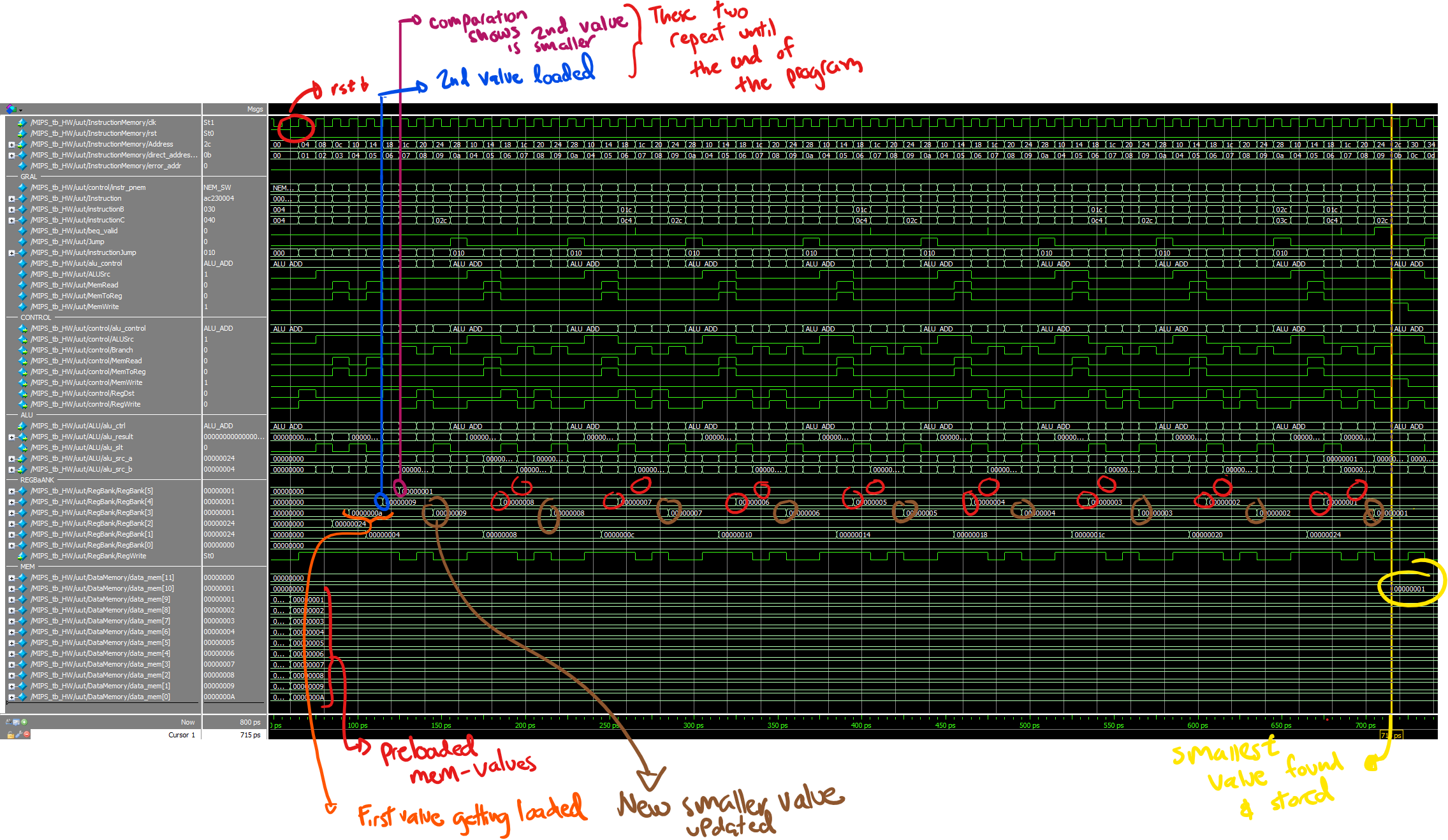
  uut.InstructionMemory.regData[11] = 32'hac230004;

  uut.InstructionMemory.regData[12] = 32'h20000000;

  uut.InstructionMemory.regData[13] = 32'h0810000c;

  uut.InstructionMemory.regData[14] = 32'h0;

Waveforms:

Waveform of the complete program run.

Two first iterations of the program:

A black screen with colorful text

Description automatically generatedRunning the testbench:

Open the ModelSim project inside the repo:

$REPO/MIPS\_GIT/ModelSim/MIPS\_ModelSim.mpf

Compile all the files

Open the waveform of the file MIPS\_tb2.sv

$REPO/MIPS\_GIT/src/MIPS\_tb2.sv

Note: The module name is called MIPS\_tb\_HW



Open the do file:

$REPO/MIPS\_GIT/ModelSim/wave\_hw.do

Done!

Modifying the values of the test:

Open and edit the file

$REPO/MIPS\_GIT/src/MIPS\_tb2.sv

Search the comment:

  // DATA MEMORY PRELOAD

  // EDIT LINES BELOW FOR PRELOADING THE MEMORY

Or line 53.

Done!