Pre-Silicon Verification 2021

Instituto Nacional de Astrofísica Óptica y Electrónica

PROJECT:Description and Test of the Multicycle Processor in Verilog

Authors:
Abel Alejandro Rubín Alvarado.
Alejandro Torija Méndez.
Enya María Rendón Soto.
Isaías Páez Castañeda.

May 2021

Contents

1 Abstract

This document presents a multicycle MIPS processor described in Verilog with R-type, Load word, store word, addi, branch and jump instructions. The first part presents an introduction and technical information about how the MIPS processor is built. Next parts present the implementation of this architecture in Verilog and description of verification process with a bubble sort algorithm to test all instructions. Finally, last sections present results, a brief conclusion and some recommendations for future work.

2 Introduction

The multicycle MIPS processor executes instructions in a series of shorter cycles, simpler instructions executes in fewer cycles, and the complex instructions take more cycles.

Now the Control Unit generates signals for the instruction's current step and keeps track of the current step, it has extra registers to hold the result of one step for use in the next step.

Instruction: A *read* control signal is sent to the memory. The contents of the program counter (PC) are used as address. The instruction fetch is the same for all instructions.

Multiple cycle changes: A memory address can come from either the PC or the ALU. If the address comes from the ALU, there will be an offset of 1024 because the memory is divided in 2 parts: 0-1023 for the instructions and 1024-2047 for data, so it adds 1024 to write in the data section and avoid rewriting the instructions. In addition, the instruction must be held in IR for all other activities.

- Control signals
  - MemRead Activated to read the instruction from memory.
  - IorD 0 to use the program counter as the memory address.
  - IRWrite Activated to write the instruction to the instruction register.

Instruction Decode: Instruction decoding produces control signals for the datapath and memory. The inputs to control circuitry are the opcode and function fields of the instruction. It generates the following control signals:

- Read and write control signals for memory.
- Write control signals for registers.
- Multiplexer controls for routing data through the datapath.
- Control signals to select an appropriate ALU operation.
- Instruction decode is the same for all instructions.

Multicycle Changes: There are some changes in the control signals. The most important difference is that they are generated by a finite state machine (FSM) so that they can have different values in different states.

Control Signals: Instruction decode is automatic, requiring no control signals.

Program Counter Update: The PC gets a new value selected from the following.

- PC + 1 (most instructions)
- Branch target address (branch instructions)

- Jump target address from the instruction jump

Multicycle Changes: PC update done in two steps. The first step is a simple increment (PC $<-$ PC + 1) done automatically while the instruction is fetched. Later modifications are made for branches, jumps, and interrupts.

Control Signals:

- PCWrite :Asserted to capture an updated program counter value. Used for program counter increment and branch and jump completion.
- Branch: Asserted for branch instructions during the completion step where it is ANDed with the ALU Zero output.
- PCSrc: Selects the source for a program counter write from the ALU result (program counter increment), the contents of the ALUOut register (branch address), or the jump address.

Source Operand :The ALU is designed to combine two source operands to produce a result. The source operand fetch activity fetches the two source operands. One source operand is selected from the following.

- The register specified by the rs instruction field (R-type and I-type source operand)
- The program counter (PC increment and branch target base)

The other source operand is selected from the following.

- The register specified by the rt instruction field (R-type source operand)
- The sign extended immediate instruction field (I-type source operand)
- The sign extended immediate instruction field, shifted left 2 places (branch target offset)
- The constant 1 (PC increment)

Multicycle Changes: For the multicycle implementation, the ALU is also incrementing the program counter and computing branch target addresses. This requires a multiplexer for each of the source operands.
Control Signals

- ALUSrcA: Selects the first ALU input from either rs or the program counter (program counter increment or branch address computation).
- ALUSrcB: Selects the second ALU input from either rt, the immediate field, the constant 1 (program counter increment), or the shifted immediate field (branch address computation).

ALU Operation: For most instructions the ALU performs the operation suggested by the instruction mnemonic, which is coded into either the opcode or the function instruction field. For loads and stores the ALU computes the address, adding the sign extended immediate field of the instruction to the contents of the register specified by the rs field of the instruction. For branches the ALU can do a subtraction in order to compare two source operands, using the result to determine whether or not to do further a further update of the PC.
Multicycle Changes: The ALU performs different operations in different states, but no new control signals are required to do this.
Control Signals

- ALUOp determines the operation performed by the ALU: add, subtract, or decoded from function field.

3 Design Description

The following resources are required for this design:

- Program Counter (Register)
- Instruction/Data Memory
- Register File
- Sign Extend
- Arithmetic Logic Unit
- ALU Control
- FSM Control Unit
- AND Gate
- OR Gate
- (5) 32-bit Register
- (2) 32-bit 4 to 1 Mux
- (3) 32-bit 2 to 1 Mux
- 5-bit 2 to 1 Mux

4 Block Diagram

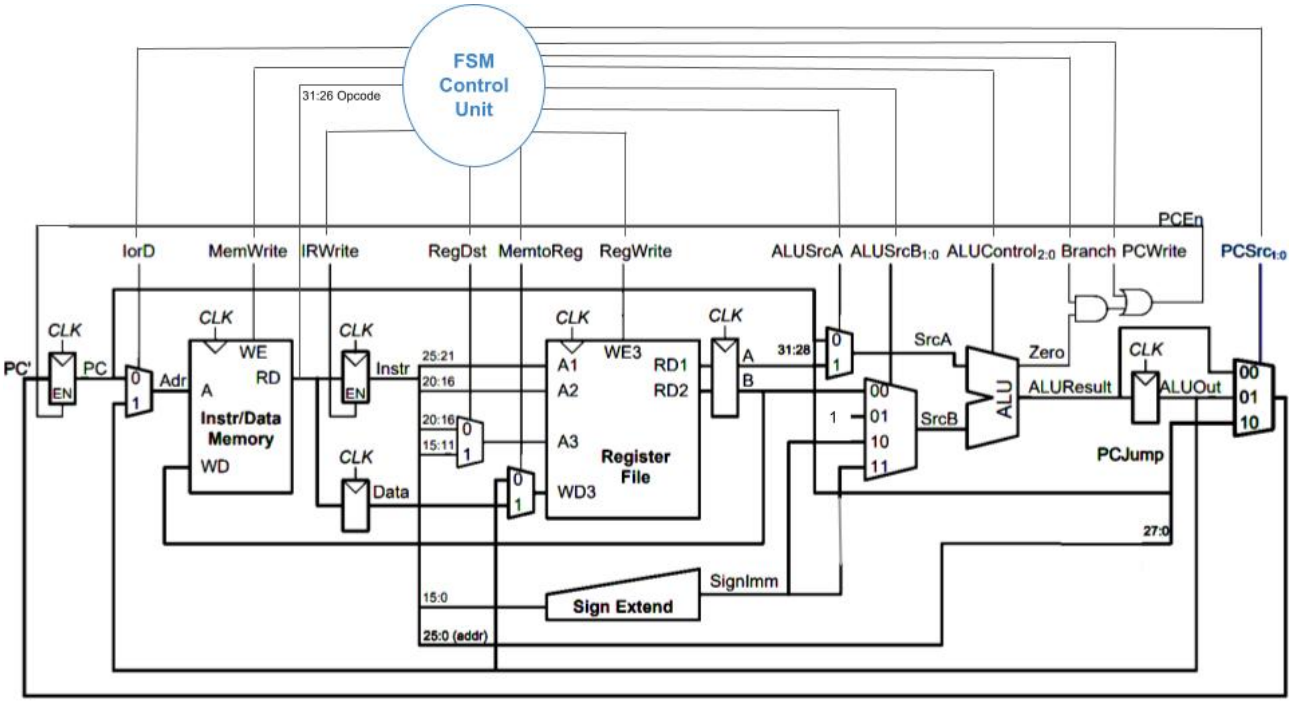


Figure 1: Complete multicycle MIPS processor

5 Implementation

MIPS Multicycle Implementation: The basic idea of the multicycle implementation is to split the long cycle of the single cycle implementation into 3 to 5 shorter cycles. The number of cycles depends on the instruction.

- Instruction fetch, program counter increment.
- Partial decoding of the instruction and calculation of the branch and jump target.
- Fetching the source operand, operating the ALU and updating the program counter for branches and jumps.
- Memory access, if necessary
- Writing to the register, if necessary

Part of the advantage of the multicycle implementation is better performance due to reduced total execution time for instructions that do not require a memory access or a write to the register. Multicycle processor implementations use Moore or Mealy finite state machines to generate control signals.

5.1 Finite State Machine Control

Finite State Machine is the unit to control data flow of each instruction and is presented in figure 3.

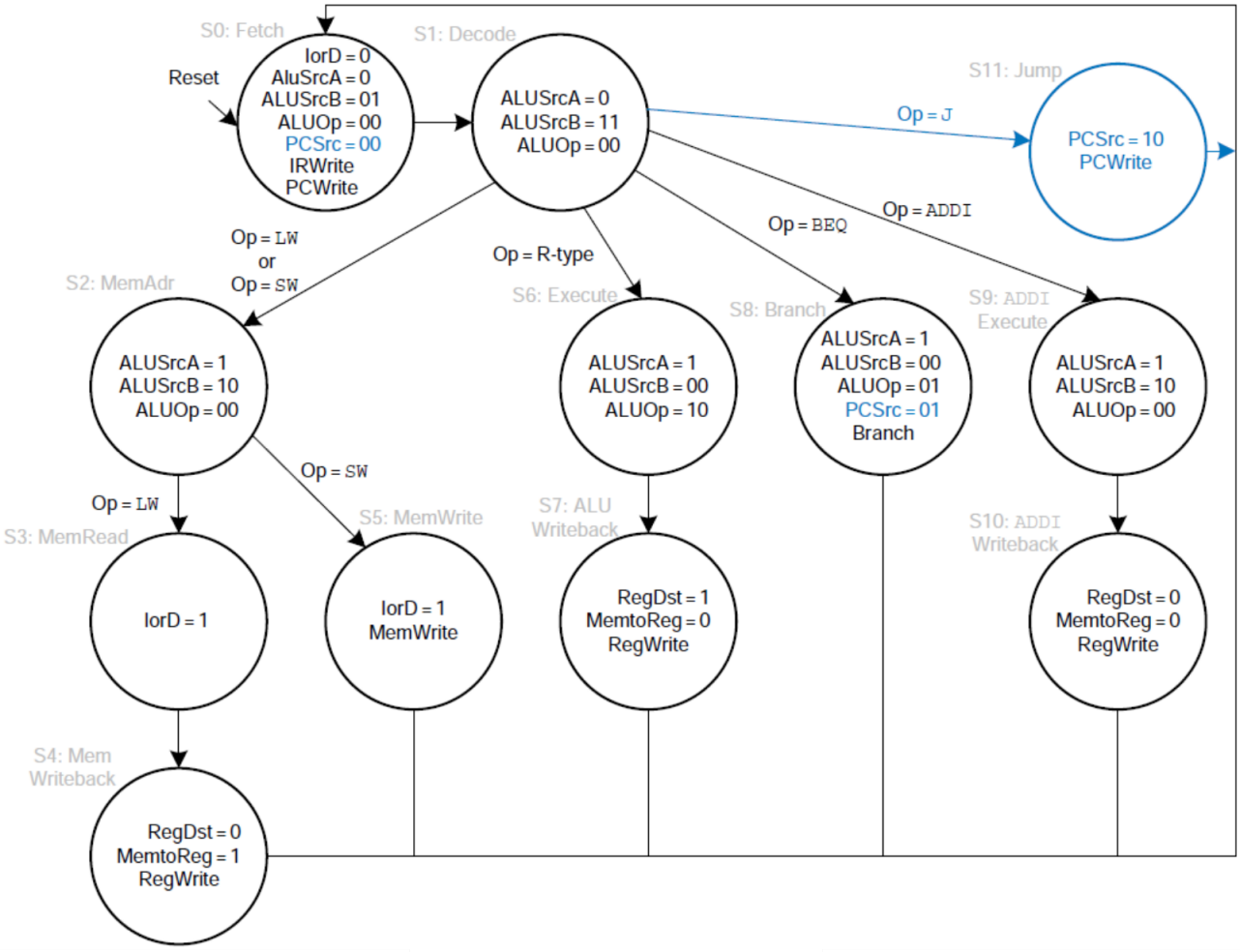


Figure 2: Finite State Machine to Control MIPS processor

5.2 HDL

```
module MC_MIPS(input clk, rst);
  //PC
  wire [31:0] PC_Plus1;
  wire [31:0] PC;
  //Mux Memory
  wire [31:0] Adr;
  wire [31:0] ALUOut;
  //Instr_Data_Mem
  wire [31:0] RD;
  //Reg Data
  wire [31:0] Data;
  //Register_File
  wire [31:0] Instr;
  wire [4:0] A1;
  wire [4:0] A2;
  wire [4:0] A3;
  wire [4:0] Dir_A3;
  wire [31:0] RD1;
  wire [31:0] RD2;
  wire [31:0] Data_WD3;
  wire [31:0]A;
  wire [31:0]B;
  //Sign_Extend
  wire [15:0] Imm;
  wire [31:0] SignImm;
  //ALU
  wire [31:0] SrcA;
  wire [31:0] SrcB;
  wire [31:0] ALUResult;
  //FSM Control
  wire [1:0] ALUOp;
  wire [5:0] op;
  wire IorD;
  wire MemWrite;
  wire IRWrite;
  wire RegDst;
  wire MemtoReg;
  wire RegWrite;
  wire ALUSrcA;
  wire [1:0] ALUSrcB;
  wire [3:0] ALUControl;
  wire branch;
  wire PCWrite;
  wire [1:0] PCSrc;
  wire [31:0] PC_prima;
  wire [5:0]funct;
  wire b_and;
  wire PCEN;
  wire [31:0]PCJump;
  //Assign
  assign A1 = Instr[25:21];
  assign A2 = Instr[20:16];
  assign A3 = Instr[15:11];
  assign op = Instr[31:26];
  assign Imm = Instr[15:0];
  assign funct = Instr[5:0];
  assign PCJump = {PC[31:26],Instr
    [25:0]};

  FSM_Control Ctrl(.clk(clk),.rst(rst),.
    opcode(op),.IorD(IorD),.MemWrite(
    MemWrite),.IRWrite(IRWrite),.RegDst(
    RegDst),.MemtoReg(MemtoReg),.RegWrite
    (RegWrite),.ALUSrcA(ALUSrcA),.ALUSrcB
    (ALUSrcB),.branch(branch),.PCWrite(
    PCWrite),.PCSrc(PCSrc),.ALUOp(ALUOp))
    ;

  PCreg PC1(.clk(clk),.rst(rst),.d(
    PC_prima),.q(PC),.EN(PCEN));

  Mux2_1 M1 (.I0(PC),.I1(ALUOut+32'd1024)
    ,.S(IorD),.out(Adr));

  Inst_Data_Memory I_D_Mem(.Address(Adr)
    ,.DataIn(B),.clk(clk),.ReadWrite(
    MemWrite), .DataOut(RD));

  Reg32 Reg_Inst(.D(RD),.Q(Instr),.EN(
    IRWrite),.clk(clk));

  Reg32 Reg_Data(.D(RD),.Q(Data),.EN(1'b1
    ),.clk(clk));

  Mux2_1 M2(.I0(ALUOut),.I1(Data),.S(
    MemtoReg),.out(Data_WD3));

  Mux5b_2_1 M5b(.I0(A2),.I1(A3),.S(RegDst
    ),.out(Dir_A3));

  Register_File RF(.A_rs(A1),.A_rt(A2),.
    A_rd(Dir_A3),.dataIn_rd(Data_WD3),.rw
    (RegWrite),.clk(clk),.rst(rst),.
    read_rs(RD1),.read_rt(RD2));

  Sign_Ext SE(.Din(Imm), .Dout(SignImm));

  Reg32 Reg_RF_RD1(.D(RD1),.Q(A),.EN(1'b1
    ),.clk(clk));

  Reg32 Reg_RF_RD2(.D(RD2),.Q(B),.EN(1'b1
    ),.clk(clk));

  Mux2_1 M3(.I0(PC),.I1(A),.S(ALUSrcA),.
    out(SrcA));

  Mux4_1 M4(.I0(B),.I1(32'h00000001),.I2(
    SignImm),.I3(SignImm),.S(ALUSrcB),.
    out(SrcB));

  ALU_ctrl_block ALU_ctrl_b(.ALUOp(ALUOp)
    ,.funct(funct),.ALUctrl(ALUControl));

  ALU ALU1(.a(SrcA),.b(SrcB), .ALUctrl(
    ALUControl),.ALU_res(ALUResult),.zero
    (zero));

  and(b_and,branch,zero);

  or(PCEN,b_and,PCWrite);

  Reg32 Reg_ALU(.D(ALUResult),.Q(ALUOut)
    ,.EN(1'b1),.clk(clk));

  Mux4_1 M5(.I0(ALUResult),.I1(ALUOut),.
    I2(PCJump),.I3(32'h00000000),.S(PCSrc
    ),.out(PC_prima));
endmodule
```

6 Description of Verification Process

6.1 Description of test program

Bubble Sort Algorithm

Searching and sorting are between the most common ingredients of programming systems. For this reason, Bubble sort algorithm was chosen to test multicycle MIPS processor and also that it uses all instructions implemented in this processor. Basic idea of bubble sorting is passing through of an array many times sequentially. Each step consists of comparing each element in array with its successor ($x[i]$ with $x[i+1]$) and the exchange of the two elements if their are not in the correct order[3]. Consider next array implemented in algorithm of MIPS processor:

57	46	35	50	11

In first step the following comparisons are made:

x[0]	with	x[1]	57 with 46	exchange
x[1]	with	x[2]	57 with 35	exchange
x[2]	with	x[3]	57 with 50	exchange
x[3]	with	x[4]	57 with 11	exchange

Table 1: First iteration of bubble sort

Then, after first step, the array is in the next order:

46	35	50	11	57

After first step bigger number is in its correct position into the array. In general x[n-1] will be in its proper position. Complete set of iterations is shown in table 2.

Iteration 0 (original array)	57	46	35	50	11
Iteration 1	46	35	50	11	57
Iteration 2	35	46	11	50	57
Iteration 3	35	11	46	50	57
Iteration 4	11	35	46	50	57

Table 2: All iterations of bubble sort

6.2 Flowchart

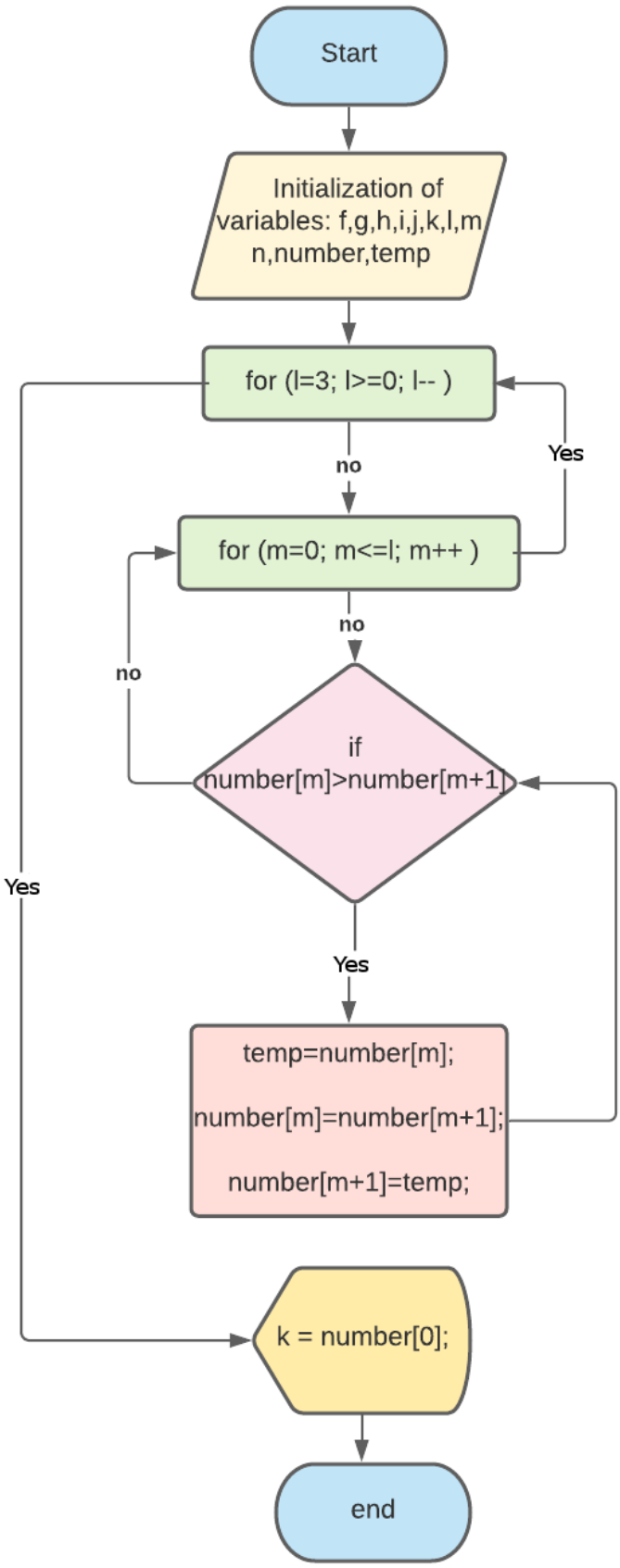


Figure 3: Bubble method flowchart.

6.3 C code

```c
#include <stdio.h>

int main()
{
    int f,g,h,i,j,k;
    printf("Enter 5 numbers: \n");
    scanf("%i", &f);
    scanf("%i", &g);
    scanf("%i", &h);
    scanf("%i", &i);
    scanf("%i", &j);

    int number [5] = {f,g,h,i,j};
    int temp;

    for(int l=3 ;l>=0;l--){
      for(int m =0; m<=l ;m++){
        if(number[m]>number[m+1]){
            temp=number[m];
            number[m]=number[m+1];
            number[m+1]=temp;
        }
      }
   }

   k = number[0];
    printf("k = %i",k);
    return 0;
}
```

6.4 Assembly code

```
#numbers:
addi $s0,$0,57
addi $s1,$0,46
addi $s2,$0,35
addi $s3,$0,50
addi $s4,$0,11
addi $t3,$0,0 # pointer
#Save numbers into memory int number [5] = {f,g,h,i,j}:
sw $s0,0($t3)
addi $t3,$t3,1
sw $s1,0($t3)
addi $t3,$t3,1
sw $s2,0($t3)
addi $t3,$t3,1
sw $s3,0($t3)
addi $t3,$t3,1
sw $s4,0($t3)

addi $t0,$0,0 #temp
addi $t1,$0,5 #i
addi $t2,$0,0 #j
addi $t4,$0,0 #k
addi $t8,$0,1 #constant 1
addi $t9,$0,16

c_for_i:
addi $t2,$0,0 #j = 0
  c_for_j:
  #number[j] = number[0]:
  lw $t5,0($t2) #s0
  #number[j+1] = number[1]
  lw $t6,1($t2) #s1
```

```
  #if(number[j]>number[j+1]):
   slt $t7,$t6,$t5
   beq $t7,$t8, OP1
   j end_if #else
   OP1: addi $t0,$t5,0 # temp = number[j]
   addi $t5,$t6,0 # number[j] = number[j+1]
   sw $t5,0($t2) #Save in memory Number[j]
   addi $t6,$t0,0 # number[j+1] = temp
   sw $t6,1($t2) #Save in memory Number[j+1]

   end_if:
   addi $t2,$t2,5
   beq $t9,$t2, dec_for_i #(j=0;j<=i;j++)
   j c_for_j

   dec_for_i:
   addi $t1,$t1,-1 #(i=3 ;i>=0;i--)
   beq $t1,$0,END_FOR
   j c_for_i

END_FOR:

lw $t4,0($0) # k = number[0] smaller number
```

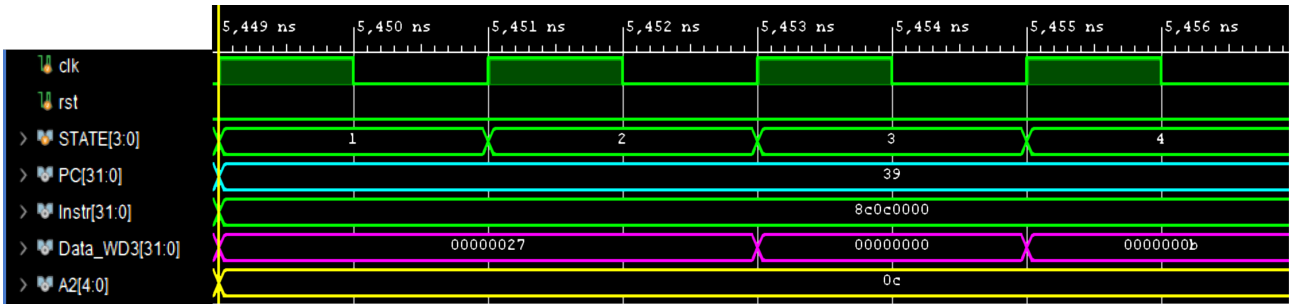
6.5 Machine Code

```
20100039
20110032
2012002e
20130023
2014000b
200b0000
ad700000
216b0001
ad710000
216b0001
ad720000
216b0001
ad730000
216b0001
ad740000
20080000
20090004
200a0000
200c0000
20180001
20190004
200a0000
8d4d0000
8d4e0001
01cd782a
11f80001
08000020
21a80000
21cd0000
ad4d0000
210e0000
ad4e0001
214a0001
132a0001
08000016
2129ffff
11200001
08000015
8c0c0000
```

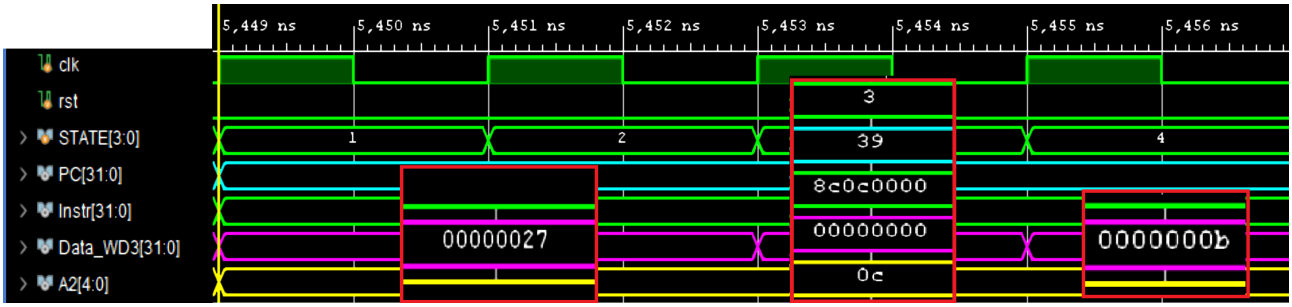
7 Results

7.1 Results obtained description from simulations

To check results, data memory is saved in a txt file and shown in this report in figure 5, then to test register file it is printed in another txt file and presented in figure 6, this final result is presented in waveform of figure 4.

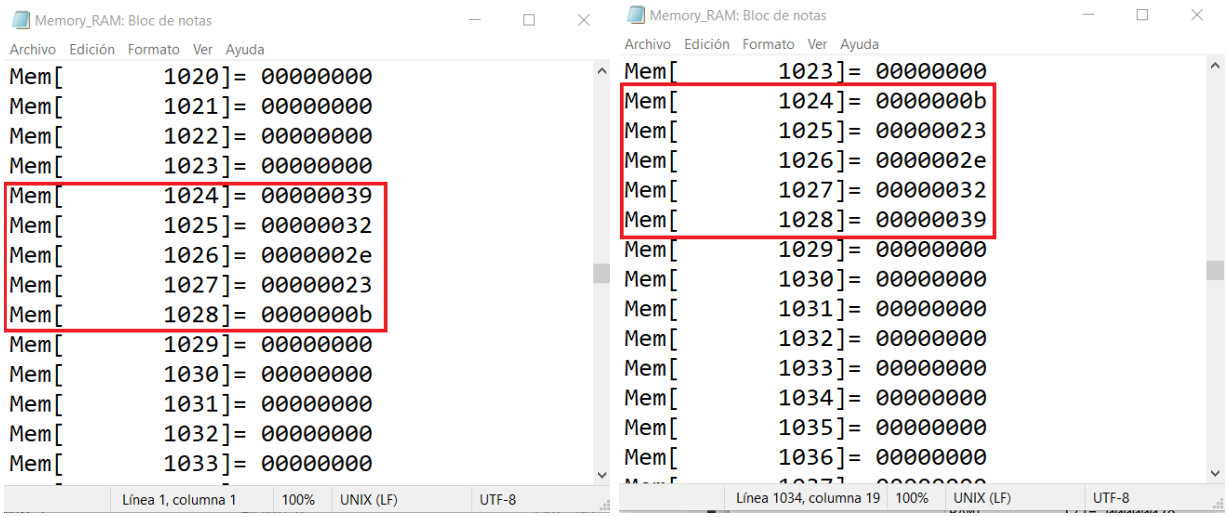


(a) .



(b) zoom.

Figure 4: Part of waveform with final output.



(a) Numbers before sort

(b) Sorted numbers

Figure 5: Data Memory file

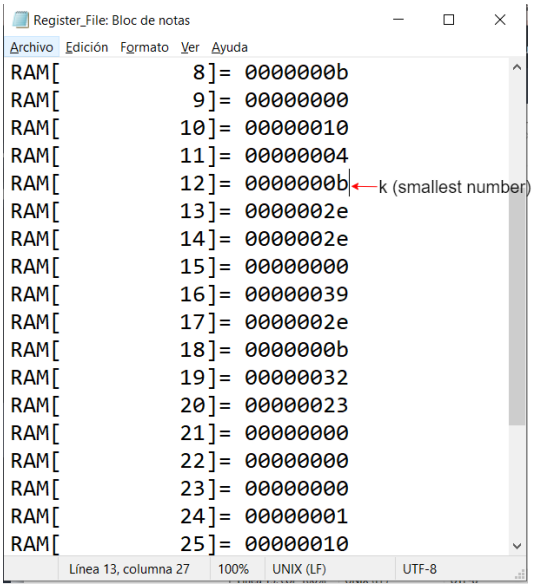


Figure 6: Register file after the execution of the program

8 Conclusions

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. The multicycle processor uses varying numbers of cycles for the various instructions. The multicycle processor requires three cycles for beq and j instructions, four cycles for sw, addi, and R-type instructions, and five cycles for lw instructions. Also, the multicycle processor is designed that each cycle involved one ALU operation, memory access, or register file access, this helps to have simpler cycles, because the multicycle processor does less work in a single cycle and, thus, has shorter cycle time.

Comparing the single cycle and multicycle Control Units, the first one has combinational logic, and the multicycle has sequential logic. The Control Unit is now a FSM, this made the verification process easier, because we used the states changes signal in the EPWave to check the correct operation of each instruction by comparing it with the state diagram.

9 Future work

9.1 Pipelined Processor

Pipelining is a powerful way to improve the throughput of a digital system. We plan to design a pipelined processor by subdividing the single-cycle processor into five pipeline stages.

9.2 Adding the rest of the instructions

The instructions implemented in this project are just a small part of the MIPS instructions, one way to complement this work would be to implement the rest of the instructions.

References

[1] D. A Patterson and J.L. Hennessy, COMPUTER ORGANIZATION AND DESIGN, 4th ed.: Morgan Kaufmann. ISNB: 978-0-12-374750-1

[2] D. M. Harris and S. L. Harris, DIGITAL DESIGN AND COMPUTER ARCHITECTURE, 1st ed.: Morgan Kaufmann. ISNB: 10: 0-12-370497-9

[3] Aaron M. Tenenbaum et al, ESTRUCTURAS DE DATOS EN C, 1st ed.: Prentice Hall. ISBN: 968-880-256-5