Project 3. MIPS Multicycle Microarchitecture

Pre-silicon Verification - Spring 2022

TEAM:

LUIS ANGEL BUSTAMANTE ROSAS

ABRAHAM JOSUE DELGADO NAVA

OSVALDO DAMIAN CRUZ LOPEZ

RICARDO BAZALDÚA CALDERÓN

June 19rd, 2022

MIPS ARCHITECTURE

CONTENTS

SUMMARY

This document will discuss the development of the MIPS Multicycle Microarchitecture, which has been designed with the necessary modules to follow multiple data paths according to the type of instruction (R,I or J) and execute a large number of instruction just like the Monocycle architecture but dividing the execution of each instruction in several smaller cycles.

This architecture allows each instruction to use the number of cycles it needs.

In the introduction we will discuss a little about the mips architecture and the differences between the monocycle and multicycle architecture.

Later, in the description of the design, the block diagram of the multicycle architecture will be described, highlighting the differences and advantages it has with the previous design (the unicycle architecture) as well as the new blocks implemented in this design.

Consequently, the HDL implementation used in the design is presented, as well as the description of the process and test program.

After that , the flowchart of the simulated program, its assembler and machine code, the results obtained in the simulations and their description are presented.
Finally, the conclusions will be made with the knowledge acquired and the possible future works that this project generated will be proposed.

INTRODUCTION

MIPS (Microprocessor without Interlocked Pipe Stages) is a general purpose processor architecture designed to be implemented on a single VLSI chip. The main goal of the design is high performance in the execution of compiled code. The architecture is experimental since it is a radical break with the trend of modern computer architectures. The basic philosophy of MIPS is to present an instruction set that is a compiler-driven encoding of the microengine.

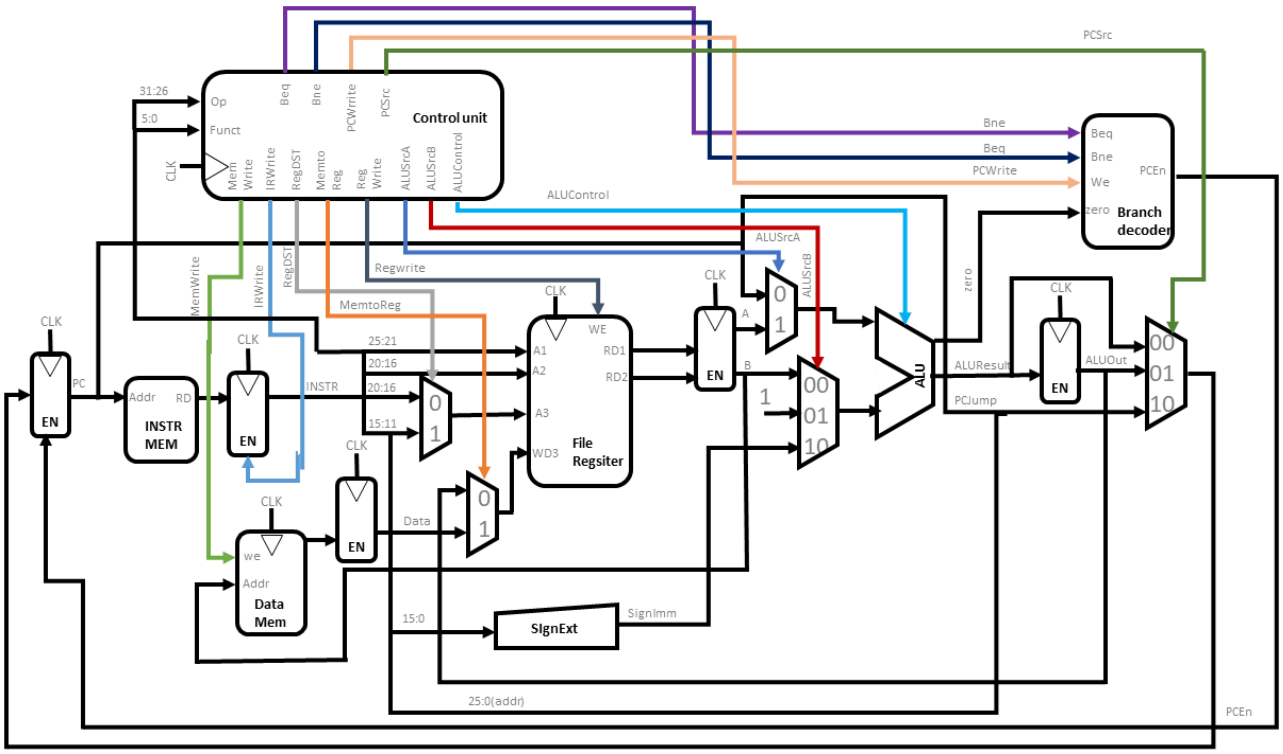
MIPS is designed for high performance. To allow the user to get maximum performance, the complexity of individual instructions is minimized. This allows the execution of these instructions at significantly higher speeds. To take advantage of simpler hardware and an instruction set that easily maps to the microinstruction set. additional compiler-type translation is needed. This compiler technology makes a compact and time-efficient mapping between higher level constructs and the simplified instruction set.

If data operands are used repeatedly in a basic block of code, having them in registers will prevent redundant load/stores and redundant addressing calculations; this allows higher throughput since more operations directly related to the computation can be performed.

A unicycle datapath is characterized in that the execution of each instruction lasts for one clock cycle. Since the clock cycle must be adapted to the needs of the execution of all instructions and needs to adjust its duration to that of whichever instruction is longer. But the time required for execution can vary appreciably from one instruction to another. So a jump or jump unconditionally takes much less time than, say, a load.

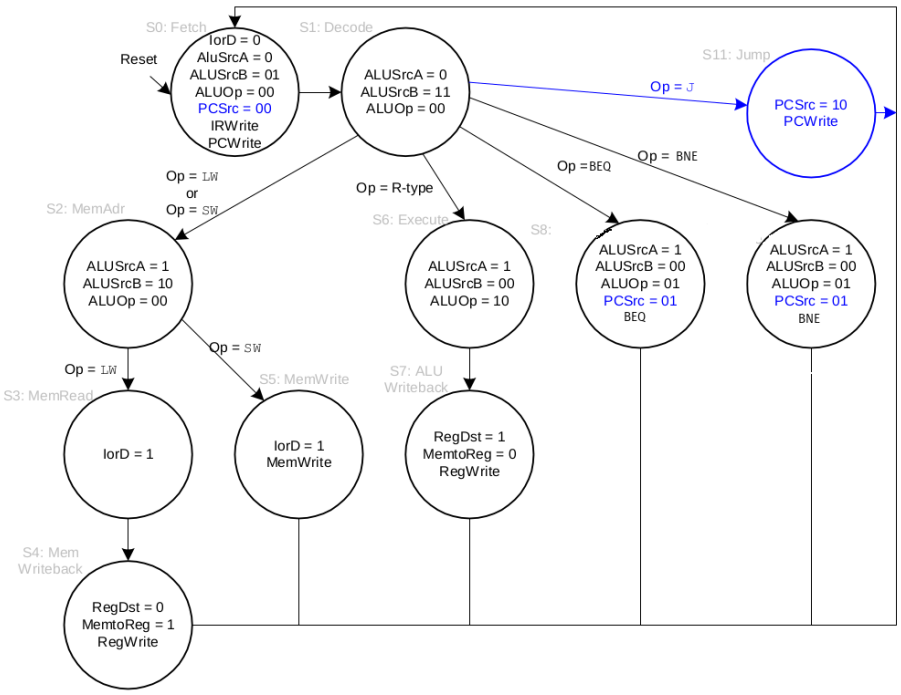
In the first place, one might think that a monocycle processor is faster than a multicycle one since the first would only require, as its name indicates, a single cycle to carry out any instruction, while the second would require more than one. The problem is that the monocycle processor actually requires as much time to execute any instruction as it does to execute the most complex instruction. On the other hand, since all possible operations need to be implemented in a single cycle, it is necessary to have the ability to do calculations at various times. and, therefore, several ALUs are required. The multicycle implementation gets rid of these problems and, by requiring less hardware, it would also imply a reduction in power consumption.

DESIGN DESCRIPTION



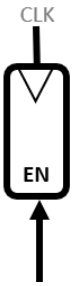
In this design we take the monocycle design previously designed and eliminate the extra modules we will not use it this architecture, also the control unit was modified, using a state machine, and some registers was added in the end of some process to be able to save the data at the end of the shorter cycles.

Due to the nature of the control unit, for its elaboration an FSM (finite state machine) diagram was used, which contains the sequence followed by this state machine and is presented below.



In this diagram we can observe the path followed by the processor in each instruction and it is more notable how some instructions run faster than others.

Although the unicycle approach is easy to understand, it is not practical since all instructions take the same time (one cycle) to complete; the clock cycle will have to adapt to the slowest. A realization would be better than allowing different instructions to take a different number of cycles, the size of the much shorter cycle. Thus, the instructions can be divided into different steps according to the unit's functions that will be used during the execution of the different stages. These allow circuitry to be shared allowing one functional unit to be used on more than one occasion during the execution of the same instruction, as long as it is done for different cycles.



This division of instructions is possible thanks to the registers added in the design, these were added as new modules in the design and allow saving the data of the instruction between cycles. Two different types of registers were added, one with the writing always enabled and the other one with the writing controlled by a variable.

HDL IMPLEMENTATION

Top level Design

```
`include "latch.v"
`include "ALU.v"
`include "control_unit.v"
`include "data_memory.v"
`include "inst_mem.v"
`include "register_file.v"
`include "SignExt.v"
`include "mux2to1.v"
`include "mux3to1.v"
`include "branch_decoder.v"

module MIPS_multicycle (input clk, rst);
/*---------------------Define internal nets---------------------*/
wire    [31:0]  PC_w, PCNext_w, Inst_w, InstLatched_w,
                Reg1_w, Reg1Latched_w, Reg2_w, Reg2Latched_w,
                SignE_w, SrcA_w, SrcB_w, ALUResult_w, ALUOut_w,
                DataM_w, DataMLatched_w, RegWD_w;

wire    [4:0]   RegWA_w;
wire    [3:0]   ALU_Ctl_w;
wire    [1:0]   ALUSrcB_w, PCSrc_w;

wire            MemtoReg_w, RegDst_w, ALUSrcA_w,
                IRWrite_w, MemWrite_w, PCWrite_w, BEQ_w, BNE_w,
                RegWrite_w, PCEn_w, zero_w;
/*---------------------Module instantiatons---------------------*/
////////////// INSTRUCTIONS MEMORY //////////////
inst_mem MIPS_inst_mem (
        .address(PC_w),
        .inst(Inst_w)
);
////////////// FILE REGISTER //////////////
register_file MIPS_register_file (
```

```
        .clk(clk),
        .regWrite(RegWrite_w),
        .readReg1(InstLatched_w[25:21]),
        .readReg2(InstLatched_w[20:16]),
        .writeReg(RegWA_w),
        .writeData(RegWD_w),
        .readData1(Reg1_w),
        .readData2(Reg2_w)
);
////////////// ALU OPERATIONS //////////////
ALU MIPS_ALU (
        .ALU_Ctl(ALU_Ctl_w),
        .A(SrcA_w),
        .B(SrcB_w),
        .ALUOut(ALUResult_w),
        .Zero(zero_w)
);
////////////// DATA MEMORY //////////////
data_memory MIPS_data_memory (
        .clk(clk),
        .address(ALUOut_w),
        .memWrite(MemWrite_w),
        .writeData(Reg2Latched_w),
        .readData(DataM_w)
);
////////////// SIGN EXTENDER //////////////
SignExt MIPS_SignExt (
        .data_in(InstLatched_w[15:0]),
        .data_out(SignE_w)
);
////////////// BRANCH DECODER //////////////
branch_decoder MIPS_branch_decoder (
        .beq(BEQ_w),
        .bne(BNE_w),
        .pcwrite(PCWrite_w),
        .zero(zero_w),
        .ctrl(PCEn_w)
```

```
);
////////////// CONTROL UNIT //////////////
control_unit MIPS_control_unit (
        .clk(clk),
        .rst(rst),
        .Opcode(InstLatched_w[31:26]),
        .Funct(InstLatched_w[5:0]),
        .MemtoReg(MemtoReg_w),
        .RegDst(RegDst_w),
        .ALUSrcA(ALUSrcA_w),
        .ALUSrcB(ALUSrcB_w),
        .PCSrc(PCSrc_w),
        .IRWrite(IRWrite_w),
        .MemWrite(MemWrite_w),
        .PCWrite(PCWrite_w),
        .BEQ(BEQ_w),
        .BNE(BNE_w),
        .RegWrite(RegWrite_w),
        .ALU_Ctl(ALU_Ctl_w)
);

/*-------------------------Multiplexers-------------------------*/

////////////// MUX RegDst //////////////
mux2to1 #(.WORD_LENGTH(5)) MIPS_mux_RegDst (
        .sel(RegDst_w),
        .Data_0(InstLatched_w[20:16]),
        .Data_1(InstLatched_w[15:11]),
        .Mux_Output(RegWA_w)
);
////////////// MUX MemtoReg//////////////
mux2to1 MIPS_mux_MemtoReg(
        .sel(MemtoReg_w),
        .Data_0(ALUOut_w),
        .Data_1(DataMLatched_w),
        .Mux_Output(RegWD_w)
);
```

```
////////////// MUX ALUSrcA //////////////
mux2to1 MIPS_mux_ALUSrcA(
        .sel(ALUSrcA_w),
        .Data_0(PC_w),
        .Data_1(Reg1Latched_w),
        .Mux_Output(SrcA_w)
);
////////////// MUX ALUSrcB //////////////
mux3to1 MIPS_mux_ALUSrcB(
        .sel(ALUSrcB_w),
        .Data_0(Reg2Latched_w),
        .Data_1(32'd1),
        .Data_2(SignE_w),
        .Mux_Output(SrcB_w)
);
////////////// MUX PCSrc //////////////
mux3to1 MIPS_mux_PCSrc(
        .sel(PCSrc_w),
        .Data_0(ALUResult_w),
        .Data_1(ALUOut_w),
        .Data_2({PC_w[31:26], InstLatched_w[25:0]}),
        .Mux_Output(PCNext_w)
);
/*----------------------------Latches-----------------------------*/
////////////// LATCH PC //////////////
latch MIPS_latch_PC(
        .clk(clk),
        .rst(rst),
        .en(PCEn_w),
        .in(PCNext_w),
        .out(PC_w)
);
////////////// LATCH INSTRUCTIONS //////////////
latch MIPS_latch_Inst(
        .clk(clk),
        .rst(rst),
        .en(IRWrite_w),
```

```
        .in(Inst_w),
        .out(InstLatched_w)
);
////////// LATCH DATA //////////
latch MIPS_latch_Data(
        .clk(clk),
        .rst(rst),
        .en(1'b1),
        .in(DataM_w),
        .out(DataMLatched_w)
);
////////// LATCH RD1 //////////
latch MIPS_latch_RD1(
        .clk(clk),
        .rst(rst),
        .en(1'b1),
        .in(Reg1_w),
        .out(Reg1Latched_w)
);
////////// LATCH RD2 //////////
latch MIPS_latch_RD2(
        .clk(clk),
        .rst(rst),
        .en(1'b1),
        .in(Reg2_w),
        .out(Reg2Latched_w)
);
////////// LATCH ALU//////////
latch MIPS_latch_ALU(
        .clk(clk),
        .rst(rst),
        .en(1'b1),
        .in(ALUResult_w),
        .out(ALUOut_w)
);

endmodule
```

Test Bench

```
`timescale 1ns/1ns
module MIPS_multicycle_tb;
reg clk_tb, rst_tb;
integer i;
MIPS_multicycle UUT (.clk(clk_tb), .rst(rst_tb));

initial begin
    clk_tb = 0;//initialize clock
    rst_tb = 1;//activate reset
    #2;
    rst_tb = 0;//deactivate reset
    #1080;
  for (i=0; i <32; i=i+1) begin

$display("-----------------------------------------------------------------
----------------------");
        $display("PC=%d  ::  REG  MEMORY=>%d  ::  DATA  MEMORY=>%d",  i,
UUT.MIPS_register_file.reg_mem[i], UUT.MIPS_data_memory.mem[i]);
  end
    $finish();
end
always forever #1 clk_tb = ~clk_tb;

initial begin
    $dumpvars(3, MIPS_multicycle_tb);
    $dumpfile("dump.vcd");
end
endmodule
```

CONTROL UNIT

```
`include "ALUControl.v"

`include "control_fsm.v"

module control_unit(

    input clk, rst,

    input [5:0] Opcode, Funct,

    /*----------Multiplexer Selects-----------*/

    output MemtoReg, RegDst, ALUSrcA,

    output[1:0] ALUSrcB, PCSrc,

    /*----------Register enables--------------*/

    output IRWrite, MemWrite, PCWrite, BEQ, BNE, RegWrite,

    /*----------ALU control------------------*/

    output [3:0] ALU_Ctl

);

wire [1:0] ALUOp;

////////// CONTROL FSM //////////

control_fsm MIPS_control_fsm (

    .clk(clk),

    .rst(rst),

    .Opcode(Opcode),

    .MemtoReg(MemtoReg),

    .RegDst(RegDst),

    .ALUSrcA(ALUSrcA),

    .ALUSrcB(ALUSrcB),

    .PCSrc(PCSrc),
```

```
    .IRWrite(IRWrite),
    .MemWrite(MemWrite),
    .PCWrite(PCWrite),
    .BEQ(BEQ),
    .BNE(BNE),
    .RegWrite(RegWrite),
    .ALUOp(ALUOp)
);
////////// ALU DECODER //////////
ALUControl MIPS_ALUControl (
    .ALUOp(ALUOp),
    .func_code(Funct),
    .ALU_Ctl(ALU_Ctl)
);
endmodule
```

CONTROL FSM

```
module control_fsm (
    /*----------Input Variables---------------*/
    input       clk, rst,
    input [5:0] Opcode,
    /*----------Multiplexer Selects-----------*/
```

```
  output reg        MemtoReg, RegDst, ALUSrcA,
  output reg  [1:0] ALUSrcB, PCSrc,
  /*----------Register enables--------------*/

  output reg  IRWrite, MemWrite, PCWrite, BEQ, BNE, RegWrite,
   /*----------ALU control------------------*/
   output reg  [1:0] ALUOp);
reg [3:0] state;
reg [3:0] next_state;
// 1. CODIFICATION
//States
parameter Fetch           = 4'd0;
parameter Decode          = 4'd1;
parameter MemAdr          = 4'd2;
parameter Mem_Read        = 4'd3;
parameter Mem_Writeback   = 4'd4;
parameter Mem_Write       = 4'd5;
parameter Execute         = 4'd6;
parameter ALU_Writeback   = 4'd7;
parameter Branch_BEQ      = 4'd8;
parameter Branch_BNE      = 4'd9;
parameter Jump            = 4'd10;
//Opcodes
parameter Op_R            = 6'd0;
```

```
  parameter Op_LW           = 6'd35;
  parameter Op_SW           = 6'd43;
  parameter Op_BEQ          = 6'd4;
  parameter Op_BNE          = 6'd5;
  parameter Op_Jump         = 6'd2;
  // 2. STATE REGISTER
  always @(posedge clk)
    begin
      if ( rst )
        state <= Fetch;
      else
        state <= next_state;
    end
  // 3. NEXT STATE PROCESS
  always @(state, Opcode)
    begin
      case(state)



////////// STATES FOR EACH OPTION FROM THE INPUT //////////
        // RESET MAIN VALUES
        Fetch : next_state = Decode;
        // INPUT STATE
        Decode : begin
```

```
    if ( Opcode == Op_LW | Opcode == Op_SW ) // lw sw
                next_state = MemAdr;

    else if ( Opcode == Op_R ) // TYPE - R
                next_state = Execute;

    else if ( Opcode == Op_BEQ ) // BEQ
                next_state = Branch_BEQ;

    else if ( Opcode == Op_BNE ) // BNE
                next_state = Branch_BNE;

    else if ( Opcode == Op_Jump ) // JUMP
                next_state = Jump;

    else        next_state = Fetch;
  end
  // SW OR LW STATE
  MemAdr : begin

    if ( Opcode == Op_LW )
                next_state = Mem_Read;

    else if ( Opcode == Op_SW )
```

```
                next_state = Mem_Write;


      else        next_state = Fetch;
    end



    // LW READ STATE
    Mem_Read :  next_state = Mem_Writeback;
    // LW FINAL STATE
    Mem_Writeback : next_state = Fetch;
    // SW STATE
    Mem_Write : next_state = Fetch;
    // R - TYPE STATE
    Execute : next_state = ALU_Writeback;
    // ALU WRITEBACK STATE
    ALU_Writeback : next_state = Fetch;
    // BRANCH IF EQUAL STATE
    Branch_BEQ : next_state = Fetch;
    // BRANCH IF NOT EQUAL STATE
    Branch_BNE : next_state = Fetch;
    // JUMP STATE
    Jump : next_state = Fetch;
    default : next_state = Fetch;
  endcase
```

```
  end

// 4. OUTPUT LOGIC PROCESS

always @(state)

    case (state)

      Decode : begin

          PCSrc      = 2'b00;

          ALUOp      = 2'b00;

          ALUSrcB    = 2'b10;

          ALUSrcA    = 1'b0;

          IRWrite    = 1'b0;

          MemWrite   = 1'b0;

          PCWrite    = 1'b0;

          BEQ        = 1'b0;

          BNE        = 1'b0;

          RegWrite   = 1'b0;

          MemtoReg   = 1'b0;

          RegDst     = 1'b0;

      end


      MemAdr : begin

          PCSrc      = 2'b00;

          ALUOp      = 2'b00;

          ALUSrcB    = 2'b10;

          ALUSrcA    = 1'b1;
```

```
            IRWrite    = 1'b0;
            MemWrite   = 1'b0;
            PCWrite    = 1'b0;
            BEQ        = 1'b0;
            BNE        = 1'b0;
            RegWrite   = 1'b0;
            MemtoReg   = 1'b0;
            RegDst     = 1'b0;
        end
        Mem_Read : begin
            PCSrc      = 2'b00;
            ALUOp      = 2'b00;
            ALUSrcB    = 2'b00;
            ALUSrcA    = 1'b0;
            IRWrite    = 1'b0;
            MemWrite   = 1'b0;
            PCWrite    = 1'b0;
            BEQ        = 1'b0;
            BNE        = 1'b0;
            RegWrite   = 1'b0;
            MemtoReg   = 1'b0;
            RegDst     = 1'b0;
        end
        Mem_Writeback : begin
```

```
            PCSrc       = 2'b00;
            ALUOp       = 2'b00;
            ALUSrcB     = 2'b00;
            ALUSrcA     = 1'b0;
            IRWrite     = 1'b0;
            MemWrite    = 1'b0;
            PCWrite     = 1'b0;
            BEQ         = 1'b0;
            BNE         = 1'b0;
            RegWrite    = 1'b1;
            MemtoReg    = 1'b1;
            RegDst      = 1'b0;
        end
        Mem_Write : begin
            PCSrc       = 2'b00;
            ALUOp       = 2'b00;
            ALUSrcB     = 2'b00;
            ALUSrcA     = 1'b0;
            IRWrite     = 1'b0;
            MemWrite    = 1'b1;
            PCWrite     = 1'b0;
            BEQ         = 1'b0;
            BNE         = 1'b0;
            RegWrite    = 1'b0;
```

```
        MemtoReg    = 1'b0;
        RegDst      = 1'b0;
    end
    Execute : begin
        PCSrc       = 2'b00;
        ALUOp       = 2'b10;
        ALUSrcB     = 2'b00;
        ALUSrcA     = 1'b1;
        IRWrite     = 1'b0;
        MemWrite    = 1'b0;
        PCWrite     = 1'b0;
        BEQ         = 1'b0;
        BNE         = 1'b0;
        RegWrite    = 1'b0;
        MemtoReg    = 1'b0;
        RegDst      = 1'b0;
    end

    ALU_Writeback : begin
        PCSrc       = 2'b00;
        ALUOp       = 2'b00;
        ALUSrcB     = 2'b00;
        ALUSrcA     = 1'b0;
        IRWrite     = 1'b0;
```

```
            MemWrite    = 1'b0;
            PCWrite     = 1'b0;
            BEQ         = 1'b0;
            BNE         = 1'b0;
            RegWrite    = 1'b1;
            MemtoReg    = 1'b0;
            RegDst      = 1'b1;
        end
        Branch_BEQ : begin
            PCSrc       = 2'b01;
            ALUOp       = 2'b01;
            ALUSrcB     = 2'b00;
            ALUSrcA     = 1'b1;
            IRWrite     = 1'b0;
            MemWrite    = 1'b0;
            PCWrite     = 1'b0;
            BEQ         = 1'b1;
            BNE         = 1'b0;
            RegWrite    = 1'b0;
            MemtoReg    = 1'b0;
            RegDst      = 1'b0;
        end
        Branch_BNE : begin
            PCSrc       = 2'b01;
```

```
            ALUOp       = 2'b01;
            ALUSrcB     = 2'b00;
            ALUSrcA     = 1'b1;
            IRWrite     = 1'b0;
            MemWrite    = 1'b0;
            PCWrite     = 1'b0;
            BEQ         = 1'b0;
            BNE         = 1'b1;
            RegWrite    = 1'b0;
            MemtoReg    = 1'b0;
            RegDst      = 1'b0;
        end
        Jump : begin
            PCSrc       = 2'b10;
            ALUOp       = 2'b00;
            ALUSrcB     = 2'b00;
            ALUSrcA     = 1'b0;
            IRWrite     = 1'b0;
            MemWrite    = 1'b0;
            PCWrite     = 1'b1;
            BEQ         = 1'b0;
            BNE         = 1'b0;
            RegWrite    = 1'b0;
            MemtoReg    = 1'b0;
```

```
            RegDst      = 1'b0;
        end
        default : begin//Fetch
            PCSrc       = 2'b00;
            ALUOp       = 2'b00;
            ALUSrcB     = 2'b01;
            ALUSrcA     = 1'b0;
            IRWrite     = 1'b1;
            MemWrite    = 1'b0;
            PCWrite     = 1'b1;
            BEQ         = 1'b0;
            BNE         = 1'b0;
            RegWrite    = 1'b0;
            MemtoReg    = 1'b0;
            RegDst      = 1'b0;
        end
      endcase
endmodule
```

ALU

```
module ALU (
    input [3:0] ALU_Ctl, //from ALU //book 317
    input [31:0] A,B,
    output reg [31:0] ALUOut,
    output Zero);
    assign Zero = (ALUOut == 0);
    always @(ALU_Ctl, A, B) begin
        case (ALU_Ctl)
            0: ALUOut = A & B;
            1: ALUOut = A | B;
            2: ALUOut = A + B;
            6: ALUOut = A - B;
            12:ALUOut = A < B ? 1:0;
            default: ALUOut = 0;
        endcase
    end
endmodule
```

ALU CONTROL

```
module ALUControl (
    input [1:0] ALUOp, //from CU
    input [5:0] func_code,  // 5:0
    output reg [3:0] ALU_Ctl // to ALU
);
```

```
    always @(ALUOp, func_code) begin
    if(ALUOp == 0)
        ALU_Ctl = 2;     //LW and SW use add
    else if(ALUOp == 1)
        ALU_Ctl = 6;         // branch if equal
    else
        case(func_code)
            //pag. 317
            32: ALU_Ctl = 2; //add
            34: ALU_Ctl = 6; //subtract
            36: ALU_Ctl = 0; //AND
            37: ALU_Ctl = 1; //OR
            42: ALU_Ctl = 12; //slt
            default: ALU_Ctl = 4'hf;
        endcase
    end
endmodule
```

DATA MEMORY

```
module data_memory (
    input clk,
    input [31:0] address,
```

```
    input memWrite,
    input [31:0] writeData,
    output [31:0] readData
    );
  reg [31:0] mem[0:255]; //32 bits memory with 256 entries
    initial begin
      $readmemb("data_mem.txt", mem);//***ESCRITURA
    end
    always @ (posedge clk) begin
        if (memWrite) begin
          mem[address] = writeData;
            $writememb("data_mem.txt", mem);
        end
    end
  assign readData = mem[address];
endmodule
```

INSTRUCTIONS MEMORY

```
module inst_mem(
    input [31:0] address,
    output [31:0]   inst);
  reg [31:0] Mem [0:255];
```

```
    initial begin
        $readmemb("instructions.txt", Mem);
    end
  assign inst = Mem[address];
endmodule
```

FILE REGISTER

```
module register_file(
    input clk,
    input regWrite,
    input [4:0] readReg1, readReg2, writeReg,
    input [31:0] writeData,
    output [31:0] readData1, readData2);
  reg [31:0] reg_mem [31:0]; //32 registers of 32 bits
/*-------------------- Initialize all registers in 0-----------------*/
  initial begin
    $readmemb("regMem.txt",reg_mem);
  end
/*-------------------- Save one register   --------------------------*/
  always @ (posedge clk) begin
    if(regWrite) begin
      reg_mem[writeReg] = writeData;
```

```
      $writememb("regMem.txt", reg_mem);
    end
  end
/*---------------------- Read the 2 registers-------------------------*/
  assign readData1 = reg_mem[readReg1];
  assign readData2 = reg_mem[readReg2];
endmodule
```

SIGN EXTENDER

```
module SignExt (
    input [15:0] data_in,
    output [31:0] data_out
);
    assign data_out = {{16{data_in[15]}},data_in};
endmodule
```

MULTIPLEXER

```
module mux2to1
#(parameter WORD_LENGTH = 32)
(input sel,
  input [WORD_LENGTH-1 : 0] Data_0, Data_1,
    // Output Ports
  output [WORD_LENGTH-1 : 0] Mux_Output);
  assign Mux_Output = (sel) ? Data_1: Data_0;
endmodule
```

```
module mux3to1
#(parameter WORD_LENGTH = 32)
(
  input [1:0] sel,
  input [WORD_LENGTH-1 : 0] Data_0, Data_1, Data_2,
  output reg [WORD_LENGTH-1 : 0] Mux_Output);
always @(*) begin
  case (sel)
  0: Mux_Output = Data_0;
  1: Mux_Output = Data_1;
  2: Mux_Output = Data_2;
  default: Mux_Output = 0;
  endcase
end
endmodule
```

BRANCH DECODER

```
module branch_decoder(
  input beq, bne, pcwrite, zero,
  output ctrl);
  assign ctrl = (bne&~zero) | (beq&zero) | (pcwrite);
endmodule
```

LATCH

```
module latch(clk, rst, en, in, out);

    input clk, rst, en;

    input [31:0] in;

    output reg [31:0] out;

    always @(posedge clk) begin

        if (rst)

        out <= 0;

        else if (en)

        out <= in;

    end

endmodule
```

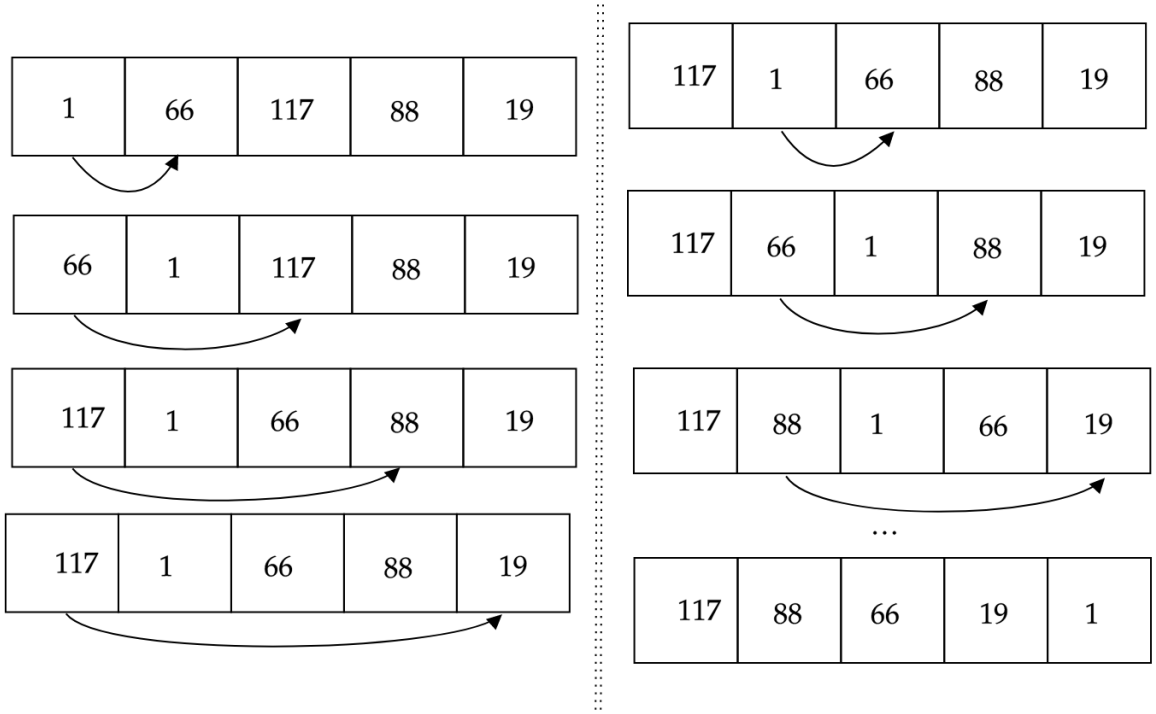
DETAILED DESCRIPTION OF VERIFICATION PROCESS

TEST PROGRAM

Given five numbers, the test program orders them from greatest to least. For example, given the numbers 1, 66, 117, 88, 19, the output of the algorithm orders them in the form 117, 88, 66, 19, 1.

OBJECTIVE

The algorithm implemented is known as “Bubble sort” that consists of comparing each value with the rest and making the changes if the condition “<” between the two numbers is true. The graphical process of this algorithm is shown below.



DESIRED RESULTS

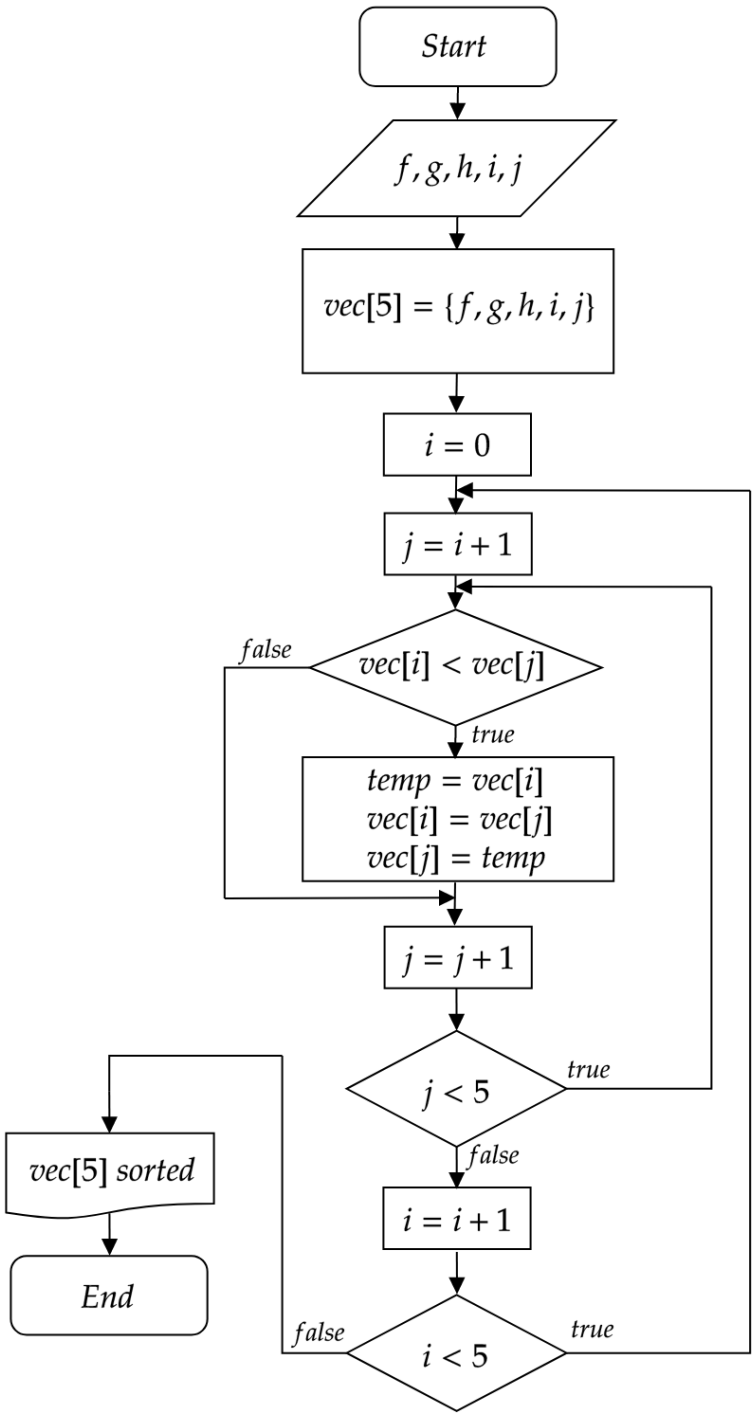
On Data Memory, program would show the next information:

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000075	0x00000058	0x00000042	0x00000013	0x00000001

PROGRAM FLOWCHART

The flow chart for the code implemented to sort the numbers (as described above) is the following:



C CODE

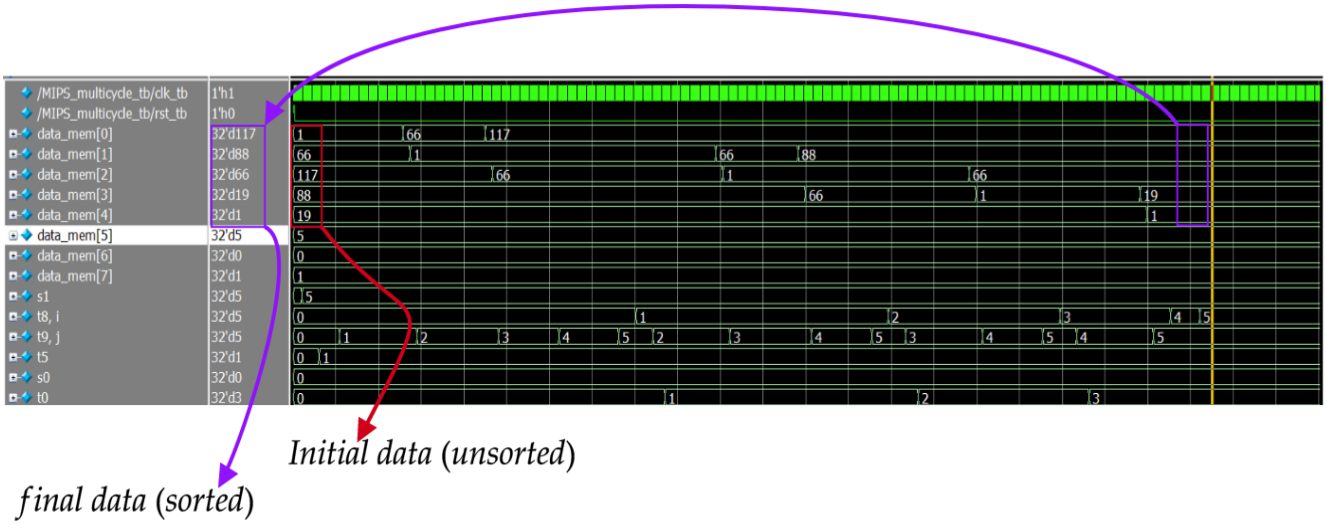
With the flow chart elaborated, the next step is to write a code in C language that works as intended. The result is shown below.

```c
#include <stdio.h>
#include <stdlib.h>

int main ()
{

    int vec [5] = {0};//Array used to store values
    int i=0, j=0; //Control variables used in For loops
    int temp;//Integer used to save previous values of vec[i]
             //to change places with vec[j]


    //Data request from monitor, to be saved in vec array
    for(i=0;i<5;i++){
        printf("Ingresa numero %d:",i+1);
        scanf("%d",&vec[i]);
    }

    // Sorting algorithm, order the values from greatest to lowest
    for(i=0;i<5;i++){
       for(j=i+1;j<5;j++){
        if(vec[i]<vec[j]){
            temp=vec[i];      //Save current value of vec[i]
            vec[i]=vec[j];    //Replace value of vec[i] with vec[j]
            vec[j]=temp;      //Save previous value of vec[i] into vec[j]

        }
       }
    }

    putchar('\n');              // jump one line to display correctly
    for(i=0;i<5;i++){
        printf(" %d\n",vec[i]); //Display sorted values

    }

    return 0;
```

ASSEMBLER AND MACHINE CODE

labels	Assemby instruction	Type	Binary machine code
	lw $s1,5($0)	I	10001100000100000000000000000000
	lw $t8,6($0)	I	10001100000110000000000000000000
	lw $t5,7($0)	I	10001100000011000000000000000000
	lw $s0,8($0)	I	10001100000100000000000000000000
FOR_i:	beq $t8, $s1, DONE_i	I	00010011000100010000000000000000
	add $t9, $t8,$t5	R	00000011000011011100100000000000
FOR_j:	beq $t9, $s1, DONE_j	I	00010011001100010000000000000000
	add $t0, $t8, $s0	R	00000011000100000100000000000000
	lw $t1, 0($t0)	I	10001101000010000000000000000000
	add $t2, $t9, $s0	R	00000011001100000101000000000000
	lw $t3, 0($t2)	I	10001101010010100000000000000000
	slt $t4, $t1, $t3	R	00000001001010110110000000000000
IF:	beq $t4,$0, EXIT_IF	I	00010001100000000000000000000000
	lw $t7,0($t0)	I	10001101000011100000000000000000
	sw $t3,0($t0)	I	10101101000010100000000000000000
	sw $t7,0($t2)	I	10101101010011100000000000000000
EXIT_IF:	add $t9,$t9,$t5	R	00100011001110010000000000000000
	j FOR_j	J	00001000000100000000000000000000
DONE_j:	add $t8, $t8, $t5	R	00100011000110000000000000000000
	j FOR_i	J	00001000000100000000000000000000
DONE_i:			

RESULTS




```
#    Time: 0 ps  Iteration: 0  Process: /MIPS_multicycle_tb/#INITIAL#35 File: C:/quartus/Diplomado/mips_multi_cycle/MIPS_multicycle_tb.v Line: 37
# ------------------------------------------------------------------------------
# PC=          0 :: REG MEMORY=>         0 :: DATA MEMORY=>       117
# ------------------------------------------------------------------------------
# PC=          1 :: REG MEMORY=>         0 :: DATA MEMORY=>        88
# ------------------------------------------------------------------------------
# PC=          2 :: REG MEMORY=>         0 :: DATA MEMORY=>        66
# ------------------------------------------------------------------------------
# PC=          3 :: REG MEMORY=>         0 :: DATA MEMORY=>        19
# ------------------------------------------------------------------------------
# PC=          4 :: REG MEMORY=>         0 :: DATA MEMORY=>         1
```

EDA Playground PROJECT LINK

The link for the Project on EDA Playground is:

https://www.edaplayground.com/x/m3EN

CONCLUSIONS

Making a multicycle Mips processor required a lot of work on every detail and running test by test in order to improve the main code without errors, so it was developed module by module and when each of them was able to work properly, add it to the top program.

On the other hand, there was some issues while creating the program as each module have a different purpose and even if they worked in the testing alone, at the time it was added to the program, some problems were discovered an step by step solved by looking for the malfunctioning area and adapting it to the goal of the processor.

Creating a processor in this way is an excellent opportunity to understand and even improve the functionality of it by researching and analyzing each part that composes the structure. This project leaves the team the knowledge needed to be aware of the processes that involve a processor and let us modify it to an improved framework with less coding and/or better understandability when reviewing the program.

FUTURE WORK

After the implementation of this microprocessor in its multicycle configuration, it is planned to develop verification tests using Systemverilog/UVM as part of the course.

REFERENCES

- **Hennessy, J. and Patterson, D., n.d. *Computer architecture*. 6th ed.**
- **https://link.springer.com/chapter/10.1007/978-3-642-68402-9_37**
- **https://resources.pcb.cadence.com/blog/2020-hardware-description-languages-vhdl-vs-verilog-and-their-functional-uses**
- **https://www.dsi.uclm.es/personal/FcoAlfaro/pfc/TFG%20Pablo%20Abril.pdf**