ITESO

Universidad Jesuita
de Guadalajara

ARQUITECTURA DE COMPUTADORAS

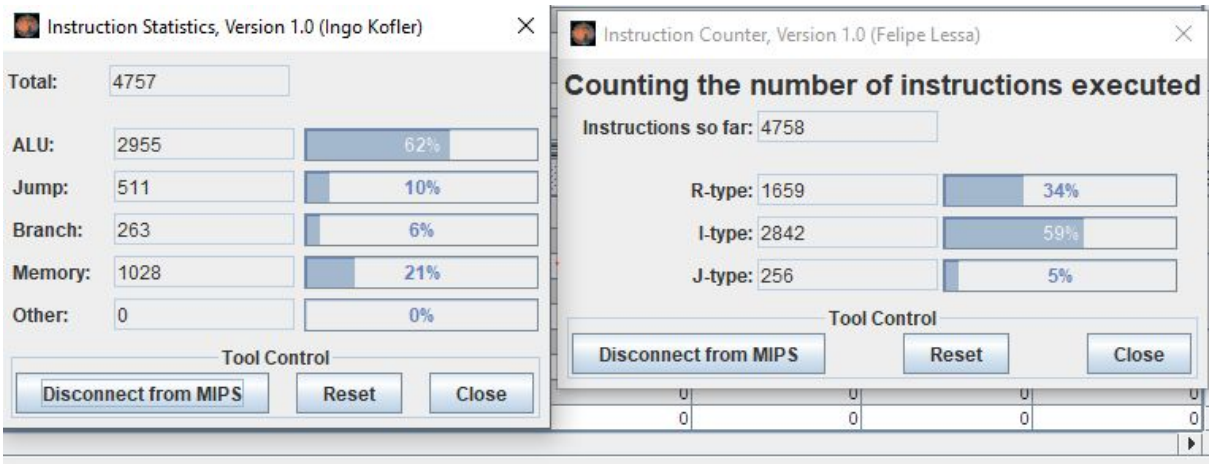
TEAM MEMBERS

Juan Carlos Álvarez Gutiérrez
Luis Fernando Palafox Pucheta

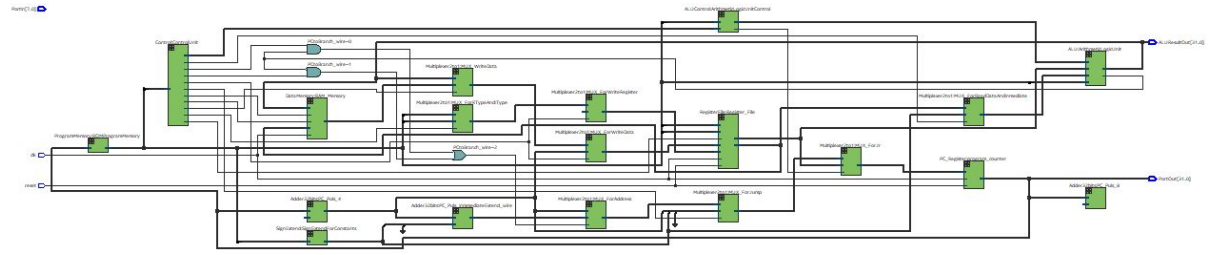
April 17, 2018

PRACTICE # 2

1.- Include IC, CPI, Clock Rate and CPU time for the MIPS Implementation and type R, I and J instructions porcentage.



2.- MIPS Micro-Architecture



3.- Modified Modules

ALU.V

In this module we added an specific localparam for each instruction that was implemented and we set cases for each instruction. In each case the ALUResult obtains a different result from the operation specified by the instruction.
We also added a new input called “Shamt” for the SRL and SLL instructions.

```
module ALU
(
        input [3:0] ALUOperation,
        input [31:0] A,
        input [31:0] B,
        input [4:0] Shamt,
        output reg Zero,
        output reg [31:0]ALUResult
);



localparam AND = 4'b0000;
localparam OR =  4'b0001;
localparam NOR = 4'b0010;
localparam ADD = 4'b0011;
localparam SUB = 4'b0100;
localparam SLL = 4'b1000;
localparam SRL = 4'b1001;
localparam BEQ = 4'b1100;
localparam LUI = 4'b1110;
localparam MEM = 4'b1010;
localparam JR  = 4'b1011;
```

```
case (ALUOperation)
  ADD: // add
        ALUResult=A + B;
  SUB: // sub
        ALUResult=A - B;
  AND:
        ALUResult=A & B;
  OR:
        ALUResult=A | B;
  NOR:
        ALUResult=~(A | B);
  SLL:
        ALUResult= B << Shamt;
  SRL:
        ALUResult= B >> Shamt;
  BEQ:
        ALUResult=A - B;
  LUI:
        ALUResult={B[15:0],16'b0};
  MEM:
        ALUResult=(A + B - 268500992) / 4;
  JR:
        ALUResult = A;
```

ALU Control.v

This module contains an specific localparam for each R-Type and I-Type instruction and each param contains an specific opcode which was taken from the Green Card document that the professor gave us.

We added new cases in the selector for the implemented instructions and each instruction has its own ALUControlValue that was specified in Alu.v module as a localparam.

```
module ALUControl
(
        input [2:0] ALUOp,
        input [5:0] ALUFunction,
        output [3:0] ALUOperation,
        output reg Jr

);

localparam R_Type_AND    = 9'b111_100100;
localparam R_Type_OR     = 9'b111_100101;
localparam R_Type_NOR    = 9'b111_100111;
localparam R_Type_ADD    = 9'b111_100000;
localparam R_Type_SUB    = 9'b111_100010;
localparam R_Type_SLL    = 9'b111_000000;
localparam R_Type_SRL    = 9'b111_000010;
localparam R_Type_JR     = 9'b111_001000;

localparam I_Type_ADDI   = 9'b100_xxxxxx;
localparam I_Type_ORI    = 9'b101_xxxxxx;
localparam I_Type_ANDI   = 9'b000_xxxxxx;
localparam I_Type_LUI    = 9'b010_xxxxxx;
localparam I_Type_BEQ    = 9'b001_xxxxxx;
localparam I_Type_SW_LW  = 9'b110_xxxxxx;


reg [3:0] ALUControlValues;
wire [8:0] Selector;
assign Selector = {ALUOp, ALUFunction};
```

```
always@(Selector)begin
        casex(Selector)
                R_Type_AND:    ALUControlValues = 4'b0000;
                R_Type_OR:             ALUControlValues = 4'b0001;
                R_Type_NOR:     ALUControlValues = 4'b0010;
                R_Type_ADD:     ALUControlValues = 4'b0011;
                R_Type_SUB:    ALUControlValues = 4'b0100;
                R_Type_SLL:    ALUControlValues = 4'b1000;
                R_Type_SRL:    ALUControlValues = 4'b1001;
                I_Type_ADDI:    ALUControlValues = 4'b0011;
                I_Type_ORI:     ALUControlValues = 4'b0001;
                I_Type_ANDI:   ALUControlValues = 4'b0000;
                I_Type_BEQ:             ALUControlValues = 4'b1100;
                I_Type_LUI:     ALUControlValues = 4'b1110;
                I_Type_SW_LW:  ALUControlValues = 4'b1010;
                R_Type_JR:     ALUControlValues = 4'b1011;
                default: ALUControlValues = 4'b1111;
        endcase
                Jr = (ALUControlValues == 4'b1011) ? 1'b1 : 1'b0;
```

Control.v

In this module we added new localparams for each implemented instruction and each localparam is the OPCODE of the instruction.

```
localparam R_Type = 0;
localparam I_Type_ADDI = 6'h8;
localparam I_Type_ORI = 6'h0d;
localparam I_Type_ANDI = 6'h0c;
localparam I_Type_BEQ = 6'h4;
localparam I_Type_BNE = 6'h5;
localparam J_Type_J = 6'h2;
localparam J_Type_JAL = 6'h3;
localparam I_Type_LUI = 6'hf;
localparam I_Type_LW = 6'h23;
localparam I_Type_SW = 6'h2b;
```

We added new ControlValues for each instruction, this control values are the ones that specify the actions that the processor will be doing for each one of the instructions.

```
always@(OP) begin
        casex(OP)
                R_Type:       ControlValues= 12'b01_001_00_00_111;
                I_Type_ADDI:  ControlValues= 12'b00_101_00_00_100;
                I_Type_ORI:   ControlValues= 12'b00_101_00_00_101;
                I_Type_ANDI:  ControlValues= 12'b00_101_00_00_000;
                I_Type_BEQ:       ControlValues= 12'b00_000_00_01_001;
                I_Type_BNE:   ControlValues= 12'b00_000_00_10_001;

                I_Type_LUI:   ControlValues= 12'b00_101_00_00_010;
                J_Type_J:         ControlValues= 12'b10_000_00_00_001;
                J_Type_JAL:       ControlValues= 12'b10_001_00_00_011;
                I_Type_LW:    ControlValues= 12'b00_111_10_00_110;
                I_Type_SW:        ControlValues= 12'b00_100_01_00_110;
                default:
                        ControlValues= 12'b10_000_00_00_011;
                endcase
                Jal = (ControlValues== 12'b10_001_00_00_011) ? 1'b1 : 1'b0;
```

Each bit of the control value has it´s own meaning and it is specified in this part of the module.

```
assign Jump =    ControlValues[11];
assign RegDst = ControlValues[10];

assign ALUSrc = ControlValues[9];
assign MemtoReg = ControlValues[8];
assign RegWrite = ControlValues[7];

assign MemRead = ControlValues[6];
assign MemWrite = ControlValues[5];

assign BranchNE = ControlValues[4];
assign BranchEQ = ControlValues[3];

assign ALUOp = ControlValues[2:0];
```

MIPS_Processor.v

This module contains all the connections in the processor, in the top of this module we added all the new wire´s that we needed to connect the right way each module with each other and the signals for each multiplexer.

```
// Data types to connect modules
wire BranchNE_wire;
wire BranchEQ_wire;
wire RegDst_wire;
wire NotZeroANDBrachNE;
wire ZeroANDBrachEQ;
wire ORForBranch;
wire ALUSrc_wire;
wire RegWrite_wire;
wire Zero_wire;
wire Jump_wire;
wire MemRead_wire;
wire MemtoReg_wire;
wire MemWrite_wire;
wire Jr_wire;
wire Jal_wire;
```

We modified some of the modules that were already instantiated with new connections with the new cables that we declared in this module and we also instantiate some more that we needed in order to connect all modules together.

In the instantiation of the control unit, we added the signal cables for the multiplexers and for the decoding and execution part of the processor.

```
Control
ControlUnit
(
        .Jump(Jump_wire),
        .OP(Instruction_wire[31:26]),
        .RegDst(RegDst_wire),
        .BranchNE(BranchNE_wire),
        .BranchEQ(BranchEQ_wire),
        .ALUOp(ALUOp_wire),
        .ALUSrc(ALUSrc_wire),
        .RegWrite(RegWrite_wire),
        .MemRead(MemRead_wire),
        .MemWrite(MemWrite_wire),
        .MemtoReg(MemtoReg_wire),
        .Jal(Jal_wire)
```

In the AritmethicLogicalUnit we connected shamt to the new input we added in Alu.v module for the SLL and SRL instructions.

```
ALU
ArithmeticLogicUnit
(
        .ALUOperation(ALUOperation_wire),
        .A(ReadData1_wire),
        .B(ReadData2OrInmmediate_wire),
        .Zero(Zero_wire),
        .ALUResult(ALUResult_wire),
        .Shamt(Instruction_wire[10:6])
);
```

In the bottom part of this module we assigned two new cables for the Program Counter in case of a BEQ or a BNE instruction.

```
assign ALUResultOut = ALUResult_wire;
assign PCtoBranch_wire = (Zero_wire & BranchEQ_wire) | (~Zero_wire & BranchNE_wire);
assign  PortOut = PC_wire;
```

We instantiated a new MUX for the jump instruction.

```
Multiplexer2to1
#(
        .NBits(32)
)
MUX_ForJump
(
        .Selector(Jump_wire),
        .MUX_Data0(MUX_PC_InmmediateExtend_wire),
        .MUX_Data1({PC_4_wire[31:28],InmmediateExtend_wire[25:0],2'b00}),
        .MUX_Output(MUX_Jump_wire)
);
```

We also added here an instantiation for the RAM memory and its signals.

```
DataMemory
RAM_Memory
(
        .WriteData(ReadData2_wire),
        .Address(ALUResult_wire),
        .MemWrite(MemWrite_wire),
        .MemRead(MemRead_wire),
        .clk(clk),
        .ReadData(ReadData_wire)
);
```

This new MUX in the module was instantiated for the JR instruction

```
Multiplexer2to1
#(
        .NBits(32)
)
MUX_ForJr
(
        .Selector(Jr_wire),
        .MUX_Data0(MUX_Jump_wire),
        .MUX_Data1(ReadData1_wire),
        .MUX_Output(MUX_PC_wire)

);
```

We also added this MUX called “MUX_ForWriteRegister when the fetching instruction is a JAL.

```
Multiplexer2to1
#(
        .NBits(5)
)
MUX_ForWriteRegister
(
        .Selector(Jal_wire),
        .MUX_Data0(MUX_ForRTypeAndIType_wire),
        .MUX_Data1({5'b11111}),
        .MUX_Output(WriteRegister_wire)
);
```

New MUX for the WriteData signal

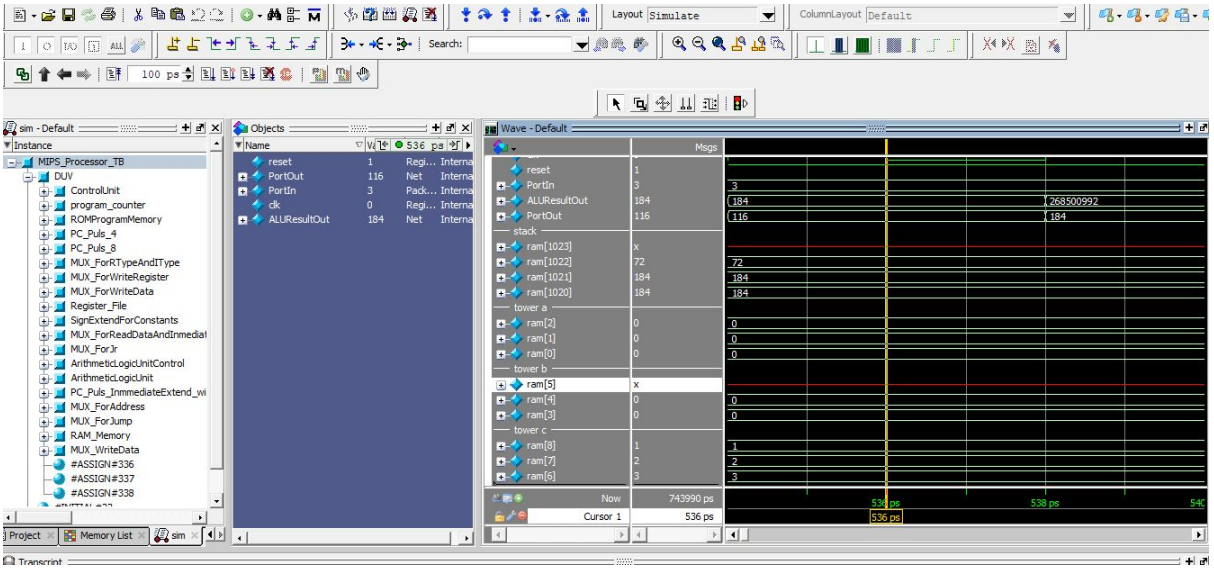
```
Multiplexer2to1
#(
        .NBits(32)
)
MUX_WriteData
(
        .Selector(MemtoReg_wire),
        .MUX_Data0(ALUResult_wire),
        .MUX_Data1(ReadData_wire),
        .MUX_Output(MUX_WriteData_wire)
);
```

DataMemory.v

We changed the parameter DATA_WIDTH from 8 to 32.

```
module DataMemory
#(      parameter DATA_WIDTH=32,
        parameter MEMORY_DEPTH = 1024

)
(
        input [DATA_WIDTH-1:0] WriteData,
        input [DATA_WIDTH-1:0]  Address,
        input MemWrite,MemRead, clk,
        output  [DATA_WIDTH-1:0]  ReadData
);

        // Declare the RAM variable
        reg [DATA_WIDTH-1:0] ram[MEMORY_DEPTH-1:0];
        wire [DATA_WIDTH-1:0] ReadDataAux;

        always @ (posedge clk)
        begin
                // Write
                if (MemWrite)
                        ram[Address] <= WriteData;
        end
        assign ReadDataAux = ram[Address];
        assign ReadData = {DATA_WIDTH{MemRead}}& ReadDataAux;

endmodule
```

4.- ModelSim MIPS Simulation



5.- Flowchart