ITESO

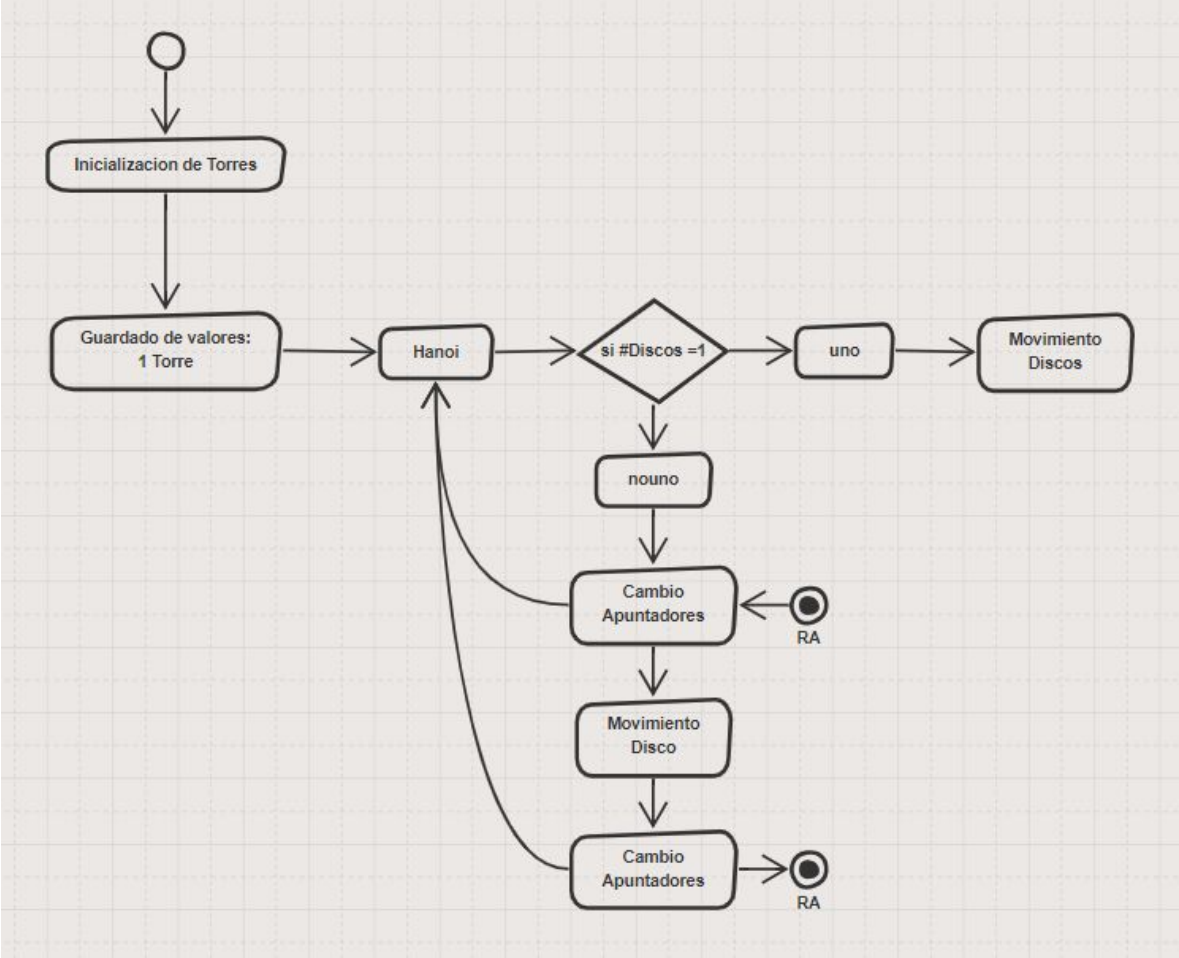
Universidad Jesuita
de Guadalajara

PRÁCTICA 2

Diseño y simulación de un procesador uni-ciclo basado en la arquitectura MIPS

Lilia Arceli Lobato Martínez
Jorge Karim Naciff Maldonnat

Diagrama de Flujo



Modificaciones

En MIPS_PROCESSOR lo que se hizo fue agregar todos los módulos faltantes para completar la práctica además de agregar los wires necesarios para interconectar dichos módulos entre ellos.

Para lograr el funcionamiento debido de la práctica lo que se modificó al datapath y al programa fue lo siguiente:

- ❏ MUX agregados
  - ❏ PCShift_OR_PC
  - ❏ PC_Jump
  - ❏ ForJalandReadData_ALUResult
  - ❏ WriteRegister_Ra
  - ❏ ForALUResultAndReadData
  - ❏ ForRJumpAndJump
- ❏ Modificaciones
  - ❏ Control Unit
  - ❏ Program Counter
  - ❏ Register_File
- ❏ Shifts Agregados

- ❏ ShiftLeft28
- ❏ ShiftLeft2

❏ Adder Agregados
- ❏ PC_Adder_shift2

❏ Unidad RAM
- ❏ Data Memory

En PC_REGISTER se modificó el PCValue para que la ROM comience en la localidad de memoria debida, en este caso 400 000 h.

```
module PC_Register
#(
    parameter N=32
)
(
    input clk,
    input reset,
    input  [N-1:0] NewPC,


    output reg [N-1:0] PCValue
);

always@(negedge reset or posedge clk) begin
    if(reset==0)
        PCValue <= 'h400_000; //empezaremos la ROM 400000
    else
        PCValue<=NewPC;
end

endmodule
```

Se cambió el apuntador de Stack para que se inicialice en la última localidad de la RAM.

```
Register
#(
    .START('h3ff) //1023
)
Register_sp
(
    .clk(clk),
    .reset(reset),
    .enable(SelectRegister_wire[29]&RegWrite),
    .DataInput(WriteData),
    .DataOutput(Intercnection_wire[30*N-1:29*N])
);
```

```
module Register
#(
    parameter N=32,
    parameter START=0
)
(
    input clk,
    input reset,
    input enable,
    input  [N-1:0] DataInput,


    output reg [N-1:0] DataOutput
);

always@(negedge reset or posedge clk) begin
    if(reset==0)
       DataOutput <= START;
    else
       if(enable==1)
          DataOutput<=DataInput;
end

endmodule
```

En ProgramMemory se modificaron los bits del Real Address para que coincidan con las verdaderas localidades de memoria.

```
module ProgramMemory
#
(
    parameter MEMORY_DEPTH=32,
    parameter DATA_WIDTH=32
)
(
    input [(DATA_WIDTH-1):0] Address,
    output reg [(DATA_WIDTH-1):0] Instruction
);
wire [(DATA_WIDTH-1):0] RealAddress;

//movemos para que coincidan con verdaderas localidades de memoria
assign RealAddress = {2'b0,Address[(DATA_WIDTH-24):2]};

    // Declare the ROM variable
    reg [DATA_WIDTH-1:0] rom[MEMORY_DEPTH-1:0];

    initial
    begin
        $readmemh("C:/MIPS/Single_Cycle_Practica2/Sources/text.dat", rom);
    end

    always @ (RealAddress)
    begin
        Instruction = rom[RealAddress];
    end

endmodule
```

En la ALU solo se modificaron los parámetros de las instrucciones para que de esta manera al ser identificada la operación debida se ejecute ésta de manera correcta.

```
module ALU
(
    input [3:0] ALUOperation,
    input [31:0] A,
    input [31:0] B,
    input [4:0] shamt, //se agrego shamt como input a la ALU
    output reg Zero,
    output reg [31:0]ALUResult
);

//Se declaran las instrucciones que podra ejecutar la ALU que corresponden a la salida de ALUControl
localparam AND = 4'b0000;
localparam OR  = 4'b0001;
localparam NOR = 4'b0010;
localparam ADD = 4'b0011;
localparam SUB = 4'b0100;
localparam SRL = 4'b0101;
localparam SLL = 4'b0110;
localparam LUI = 4'b0111;
localparam BEQ = 4'b1000;
localparam BNE = 4'b1001;

   //se agrega la instruccion en el switch case
   always @ (A or B or ALUOperation or shamt)
     begin
      case (ALUOperation)
        ADD: // add
         ALUResult=A + B;
        SUB: // sub
         ALUResult=A - B;
        OR:
         ALUResult=A | B;
        AND:
         ALUResult=A & B;
        NOR:
         ALUResult= ~(A | B);
        SRL:
         ALUResult= B >> shamt;
        SLL:
         ALUResult= B << shamt;
        LUI:
         ALUResult= B << 16;
        BEQ:
         ALUResult= (A == B) ? 1'b0 : 1'b1;
        BNE:
         ALUResult= (A != B) ? 1'b0 : 1'b1;
      default:
         ALUResult= 0;
      endcase // case(control)
      Zero = (ALUResult==0) ? 1'b1 : 1'b0;
     end // always @ (A or B or control)
endmodule // ALU
```

En la ALUControl lo que se hizo fue agregar los indicadores para diferenciar entre los tres tipos de instrucciones (R, I, J), utilizando los primeros 3 bits de los 9 que recibe este módulo. Los 6 bits restantes servirán para después de saber el tipo de la instrucción distinguir qué instrucción es la deseada a ejecutar.

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

);
//se agregan todas las operacion que nuestro procesador sera capaz de ejecutar
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_SRL    = 9'b111_000010;
localparam R_Type_SLL    = 9'b111_000000;
localparam R_Type_JR     = 9'b111_001000;

localparam I_Type_ADDI   = 9'b100_xxxxxx;
localparam I_Type_ORI    = 9'b101_xxxxxx;
localparam I_Type_LUI    = 9'b011_xxxxxx;
localparam I_Type_BEQ    = 9'b001_xxxxxx;
localparam I_Type_BNE    = 9'b010_xxxxxx;
localparam I_Type_SW     = 9'b110_xxxxxx;
localparam I_Type_LW     = 9'b000_xxxxxx;

reg [3:0] ALUControlValues;
wire [8:0] Selector;

assign Selector = {ALUOp, ALUFunction};

//se le agregan las instrucciones con un valor que debe coincidir con los valores en la ALU
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_SRL:    ALUControlValues = 4'b0101;
        R_Type_SLL:    ALUControlValues = 4'b0110;
        R_Type_JR:     ALUControlValues = 4'b1110;

        I_Type_LUI:    ALUControlValues = 4'b0111;
        I_Type_ORI:    ALUControlValues = 4'b0001;
        I_Type_ADDI:   ALUControlValues = 4'b0011;
        I_Type_BEQ:    ALUControlValues = 4'b1000;
        I_Type_BNE:    ALUControlValues = 4'b1001;
        I_Type_SW:     ALUControlValues = 4'b0011;
        I_Type_LW:     ALUControlValues = 4'b0011;

        default: ALUControlValues = 4'b1111;
    endcase
end


assign ALUOperation = ALUControlValues;

endmodule
```

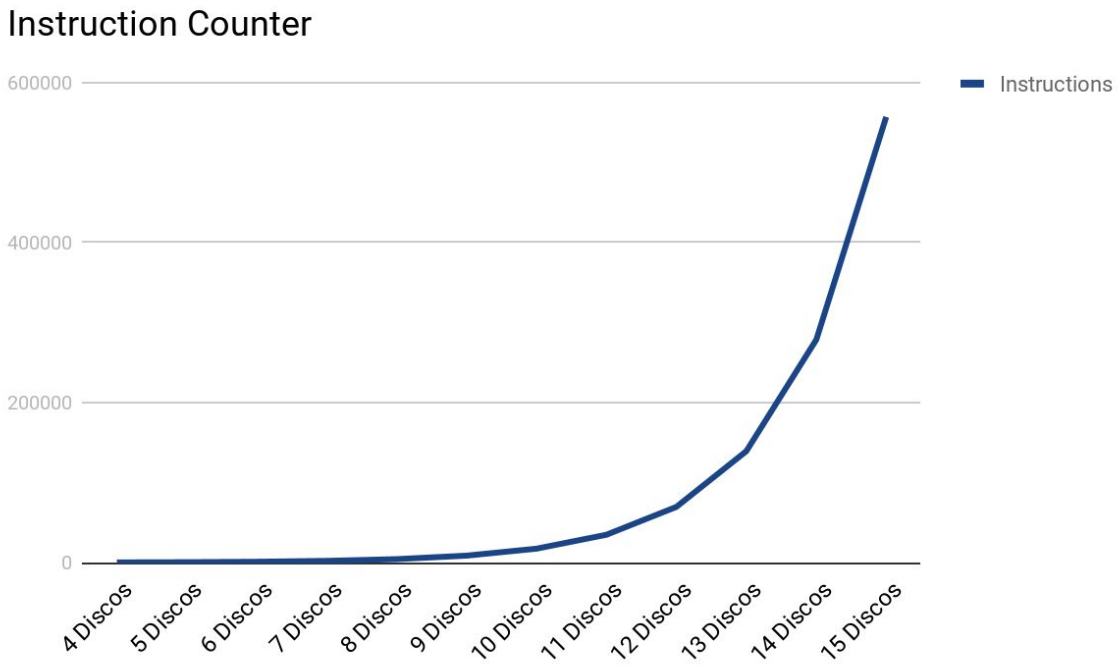
En Control se agregaron los valores de control debidos para que de esta manera se ejecute cada instrucción de manera debida utilizando cada parámetro para guiarse por el camino específico de cada instrucción en el Datapath sugerido.

```
// agregado en pract 2
wire MemRead_wire;
wire MemWrite_wire;
wire Jump_wire;
wire MemtoReg_wire;

wire PCSrc_wire;

wire JumpR_wire;
wire JumpJal_wire;

wire [04:0] MUX_Ra_WriteRegister_wire;
wire [31:0] ReadData_wire;
wire [31:0] MUX_ReadData_ALUResult_wire;
wire [31:0] PC_Shift2_wire;
wire [31:0] ShiftLeft2_SignExt_wire;
wire [31:0] Shifted28_wire;
wire [31:0] MUX_to_PC_wire;
wire [31:0] MUX_to_MUX_wire;
wire [31:0] MUX_ForRetJumpAndJump;
wire [31:0] MUX_Jal_ReadData_ALUResult_wire;
```

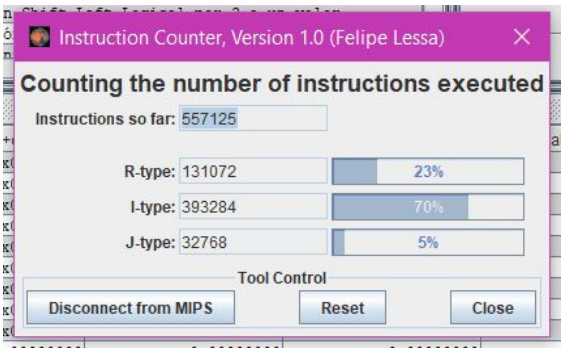
```
//Modificaciones
Control // agregamos las señales faltantes
ControlUnit
(
    .OP(instruction_bus_wire[31:26]),
    .RegDst(reg_dst_wire),
    .BranchEQ_NE(branch_eq_ne_wire), //
    .MemRead (MemRead_wire),
    .MemtoReg (MemtoReg_wire), //
    .MemWrite (MemWrite_wire), //
    .ALUOp(aluop_wire),
    .ALUSrc(alu_src_wire),
    .Jump (Jump_wire), //
    .RegWrite(reg_write_wire)
);

//Agregado en pract 2
DataMemory //conectamos nuestra RAM
#(
     .DATA_WIDTH(32),
     .MEMORY_DEPTH(1024)

)
DataMemory
(
    //In
    .clk(clk),
    .WriteData(read_data_2_wire),
    .Address({20'b0,alu_result_wire[11:0]>>2}),
    .MemRead(MemRead_wire),
    .MemWrite(MemWrite_wire),
    //out
    .ReadData(ReadData_wire)


);

ShiftLeft2 //Mueve la direccion << 2 para poder accedar a memoria (lo haca multiplo de 4)
Left2
( //Inmmediate_extend_wire
    .DataInput(Inmmediate_extend_wire), //Nos da un error así que concatenamos directamente
    .DataOutput(ShiftLeft2_SignExt_wire)
);

ShiftLeft2 //concatenamos la direccion de salto
ShiftLeft28
(
    .DataInput({6'b00000,instruction_bus_wire[25:0]}),

    .DataOutput(Shifted28_wire)
);

assign PCSrc_wire = branch_eq_ne_wire & zero_wire; //Define si es un salto u otra instruccion

Adder32bits //Agrega PC4 al JumpAddress para hacerla de 32 bits
PC_Adder_Shift2
(
    .Data0(pc_plus_4_wire),

    //.Data1(Inmmediate_extend_wire),
    .Data1({Inmmediate_extend_wire[29:0],2'b00}),

    .Result(PC_Shift2_wire) //queda PC4 + JumpAddress[25-0] + 00


);

RegisterFile // Modificado
Register_File
(
    .clk(clk),
    .reset(reset),
    .RegWrite(reg_write_wire),
    .WriteRegister(MUX_Ra_WriteRegister_wire), //elige escribir RA o dato
    .ReadRegister1(instruction_bus_wire[25:21]),
    .ReadRegister2(instruction_bus_wire[20:16]),
    .WriteData(MUX_Jal_ReadData_ALUResult_wire), //JumpAdress
    .ReadData1(read_data_1_wire),
    .ReadData2(read_data_2_wire)
);
```

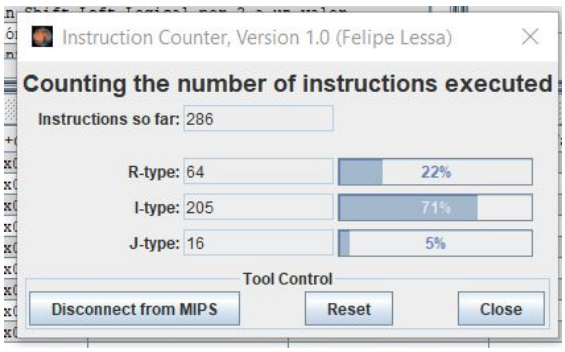
```
//MUX agregado en pract 2

//aiuda
Multiplexer2to1
#(
    .NBits(32)
)
MUX_ForALUResultAndReadData //seleccionamos que resultado debemos enviar para escribir
(
    .Selector(MemtoReg_wire),
    .MUX_Data0(alu_result_wire),
    .MUX_Data1(ReadData_wire),

    .MUX_Output(MUX_ReadData_ALUResult_wire)
);

assign JumpR_wire = (alu_operation_wire == 4'b1110) ? 1'b1 : 1'b0; //vamos a ver si la instruccion fue J
assign JumpJal_wire = ({instruction_bus_wire[31:26],Jump_wire} == 7) ? 1'b1 : 1'b0; // o vemos si es Jal

Multiplexer2to1
MUX_ForRJumpAndJump //seleccionamos la siguente instruccion del PC/jump
(
    .Selector(JumpR_wire),
    .MUX_Data0(MUX_ForRetJumpAndJump),
    .MUX_Data1(read_data_1_wire),

    .MUX_Output(MUX_to_PC_wire)
);

Multiplexer2to1 //vemos si vamos a hacer jal o ejecutaremos la siguiente instruccion
#(
    .NBits(32)
)
MUX_ForJalAndReadData_AlUResult
(
    .Selector(JumpJal_wire),
    .MUX_Data0(MUX_ReadData_ALUResult_wire),
    .MUX_Data1(pc_plus_4_wire),

    .MUX_Output(MUX_Jal_ReadData_ALUResult_wire)
);

Multiplexer2to1 //seleccionamos el registro en el que escribiremos RA/Registro N
#(
    .NBits(5)
)
MUX_WriteRegister_Ra
(
    .Selector(JumpJal_wire),
    .MUX_Data0(write_register_wire),
    .MUX_Data1(5'b11111),

    .MUX_Output(MUX_Ra_WriteRegister_wire)
);

Multiplexer2to1 //seleccionamos cual sera la siguiente instruccion
#(
    .NBits(32)
)
PCShift_OR_PC
(
    .Selector(PCSrc_wire), //decide si la siguiente instruccion es de la direccion a la que saltamos
    .MUX_Data0(pc_plus_4_wire),
    .MUX_Data1(PC_Shift2_wire),

    .MUX_Output(MUX_to_MUX_wire)
);


Multiplexer2to1 //seleccionamos entre pc o jump
#(
    .NBits(32)
)
MUX_PCJump
(
    .Selector(Jump_wire),
    .MUX_Data0(MUX_to_MUX_wire),
    .MUX_Data1({pc_plus_4_wire[31:28],Shifted28_wire[27:0]}),

    .MUX_Output(MUX_ForRetJumpAndJump)
);
```

IC



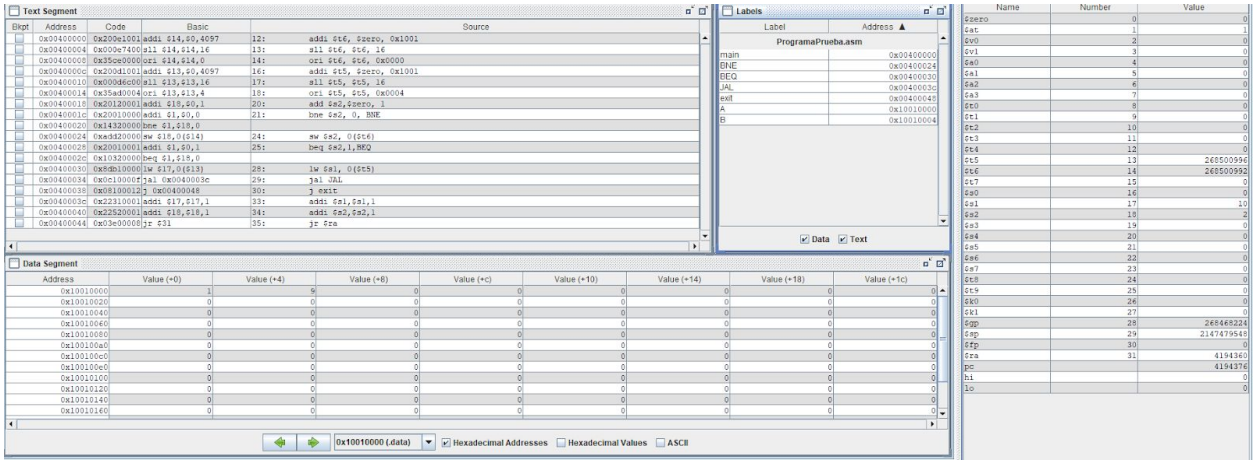
	Instructions
4 Discos	286
5 Discos	563
6 Discos	1112
7 Discos	2205
8 Discos	4386
9 Discos	8743
10 Discos	17452
11 Discos	34865
12 Discos	69686
13 Discos	139323
14 Discos	278592
15 Discos	557125


Instruction Counter, Version 1.0 (Felipe Lessa)
Counting the number of instructions executed
Instructions so far: 557125
R-type: 131072
23%
I-type: 393284
70%
J-type: 32768
5%
Tool Control
Disconnect from MIPS
Reset
Close

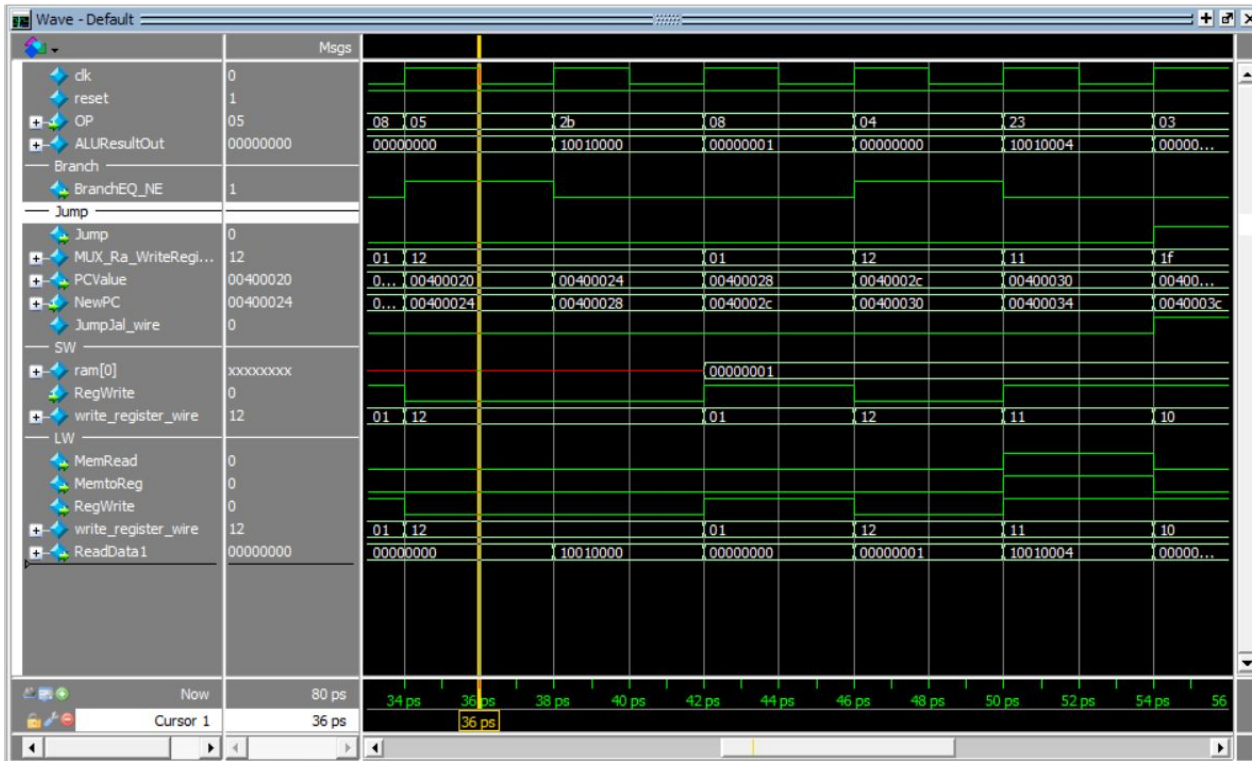

Instruction Counter, Version 1.0 (Felipe Lessa)
Counting the number of instructions executed
Instructions so far: 286
R-type: 64
22%
I-type: 205
71%
J-type: 16
5%
Tool Control
Disconnect from MIPS
Reset
Close

Programa Simulación BNE BEQ LW SW J JR

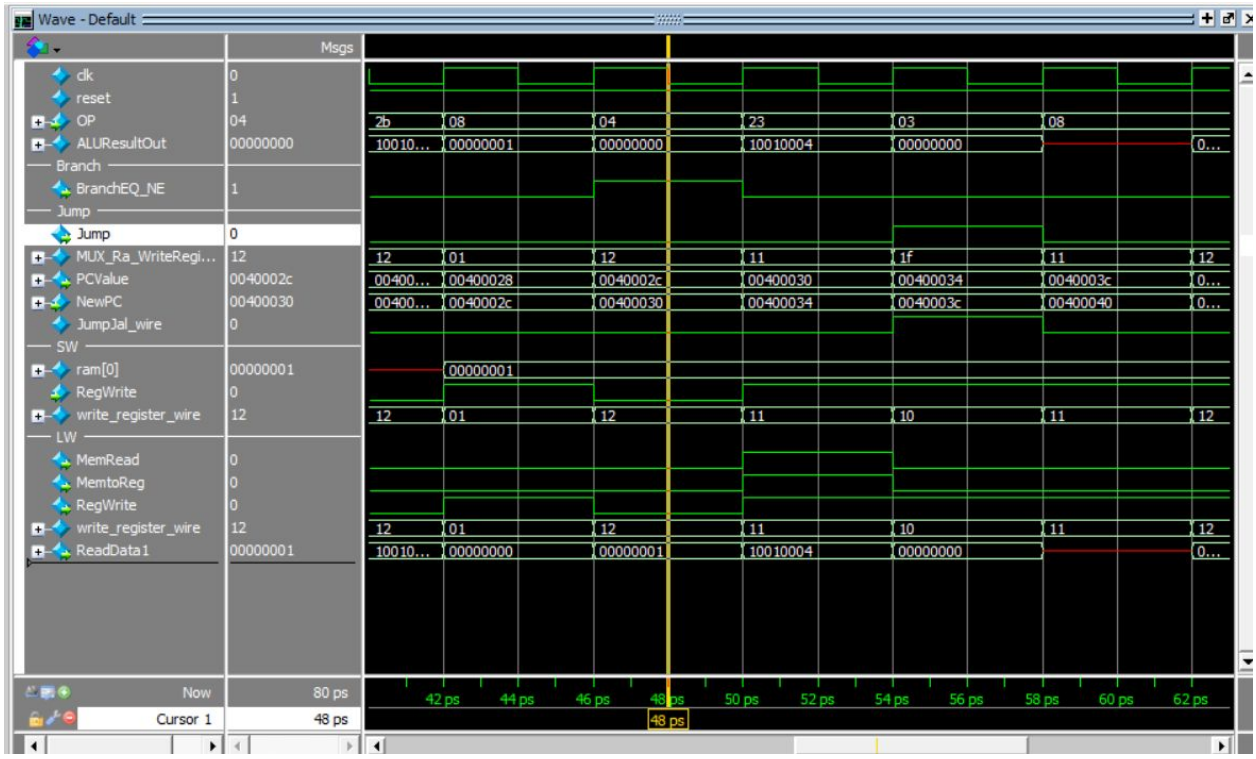
Programa a Simular:



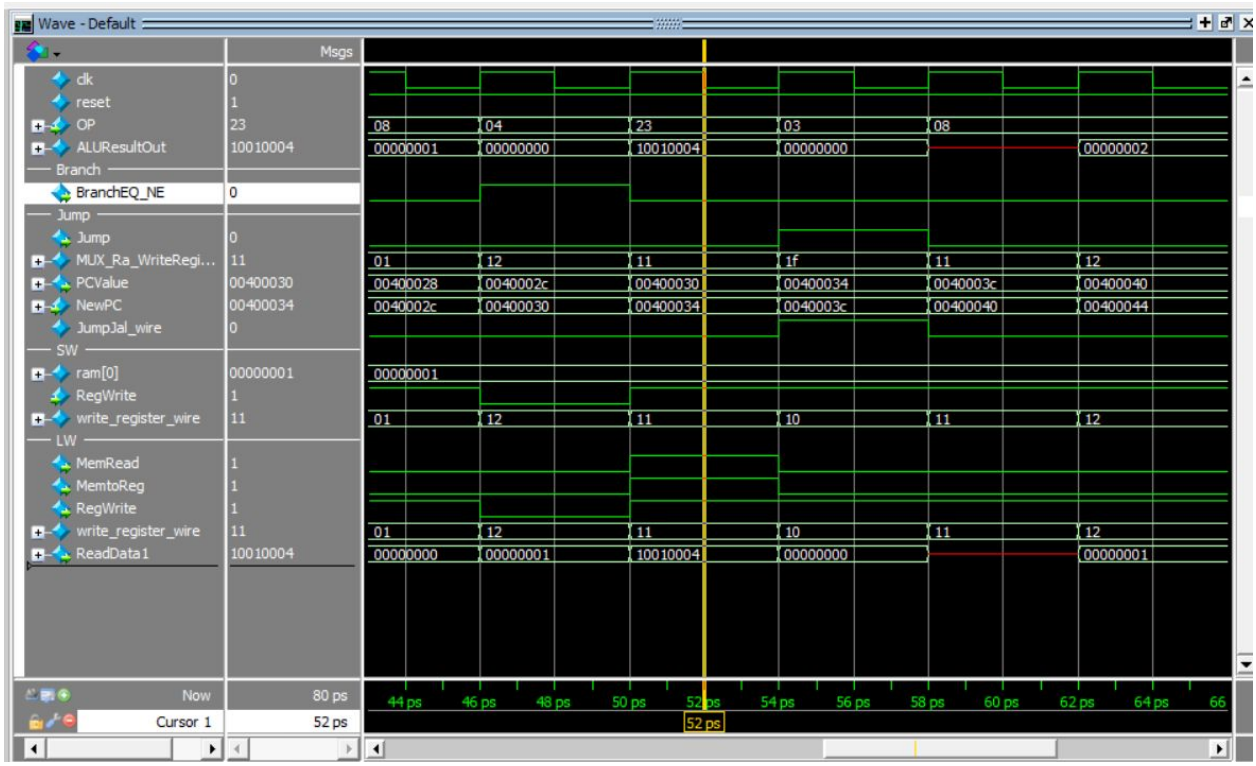
Bne: OP=5, JumpAdress=400024



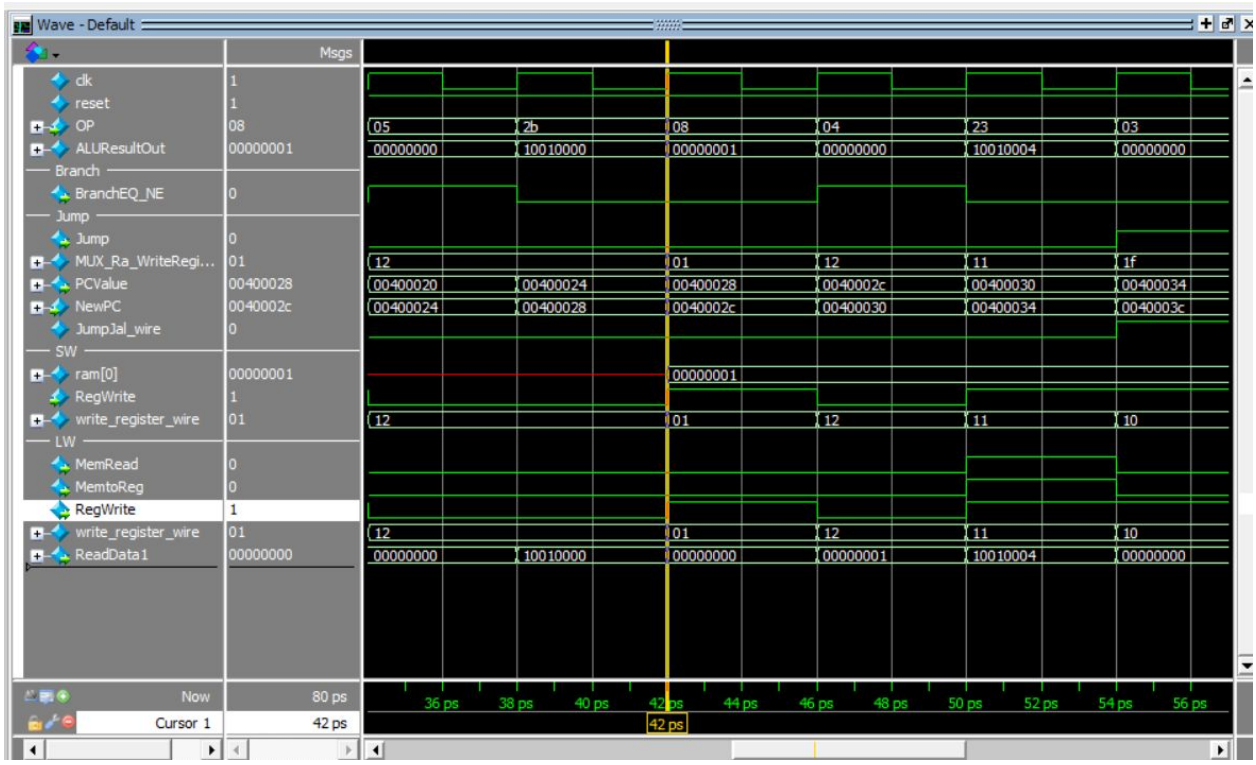
Beq: OP=4, JumpAdress=400030



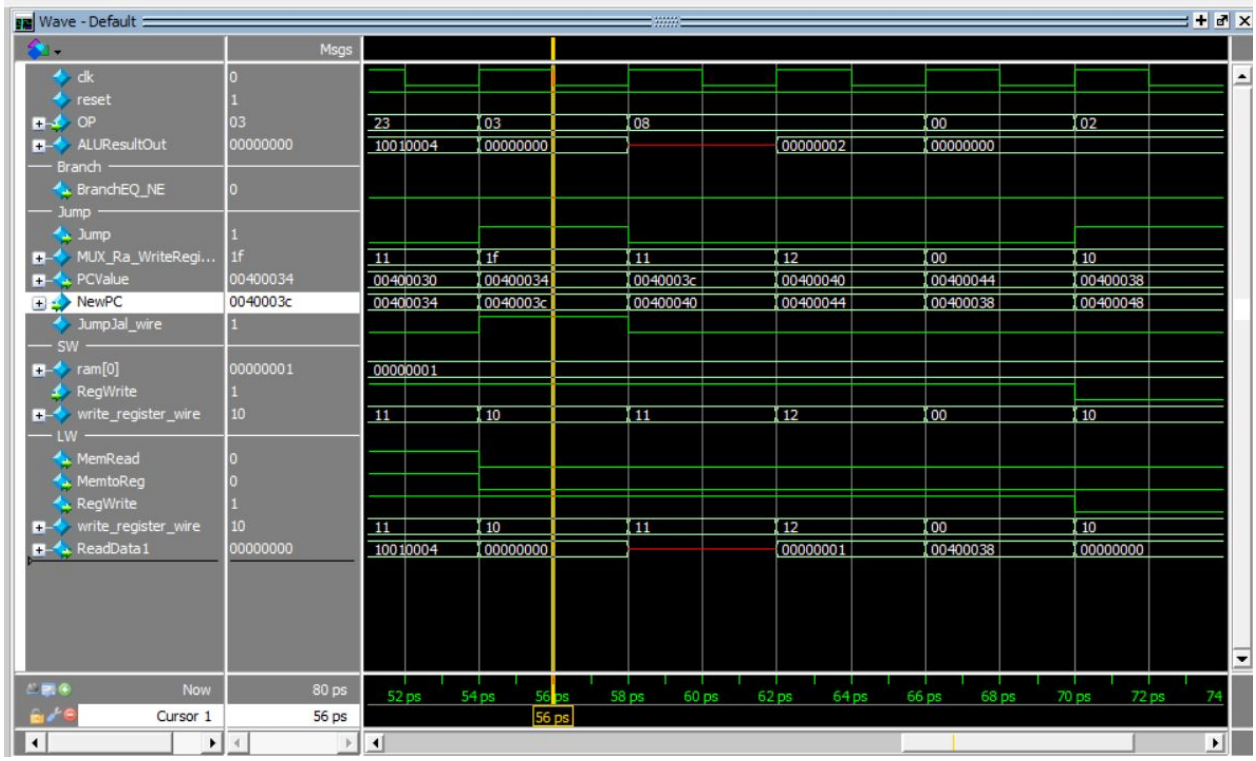
Lw: OP=23, en ReadData1 se ve la memoria en RAM donde está almacenado el valor



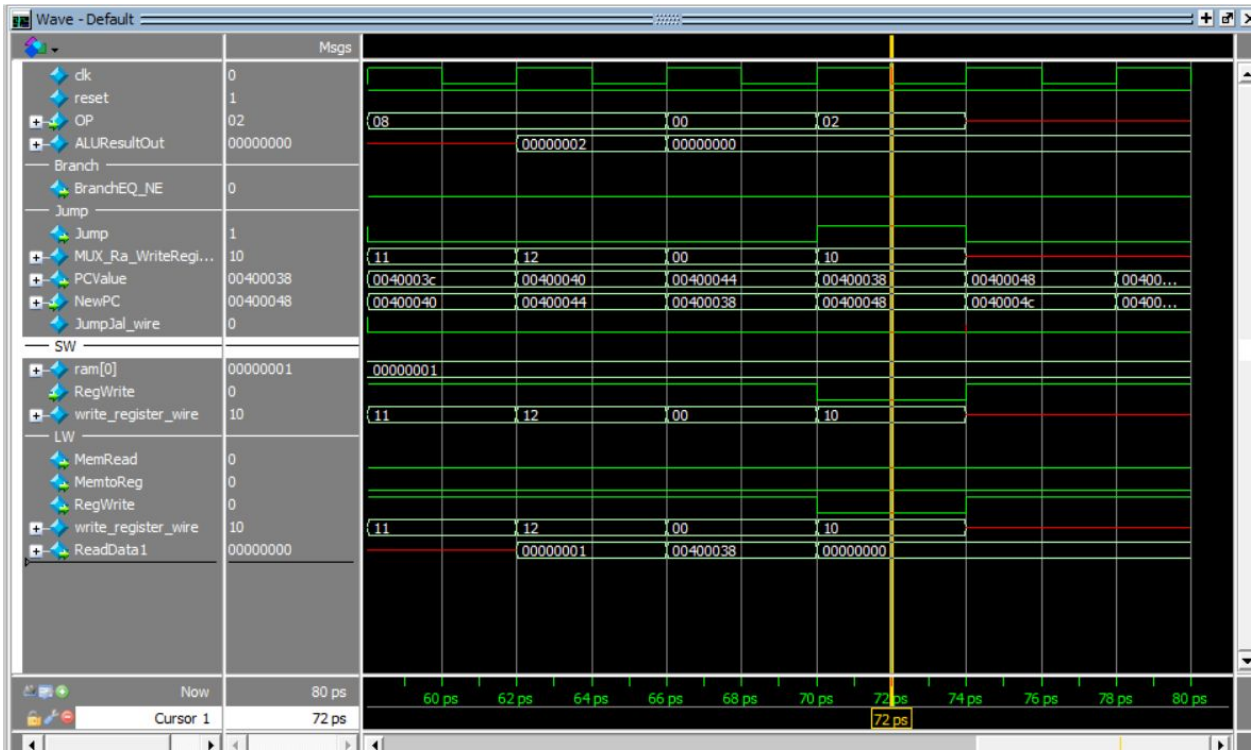
Sw: OP=2b, en RAM se puede ver como se guarda el valor



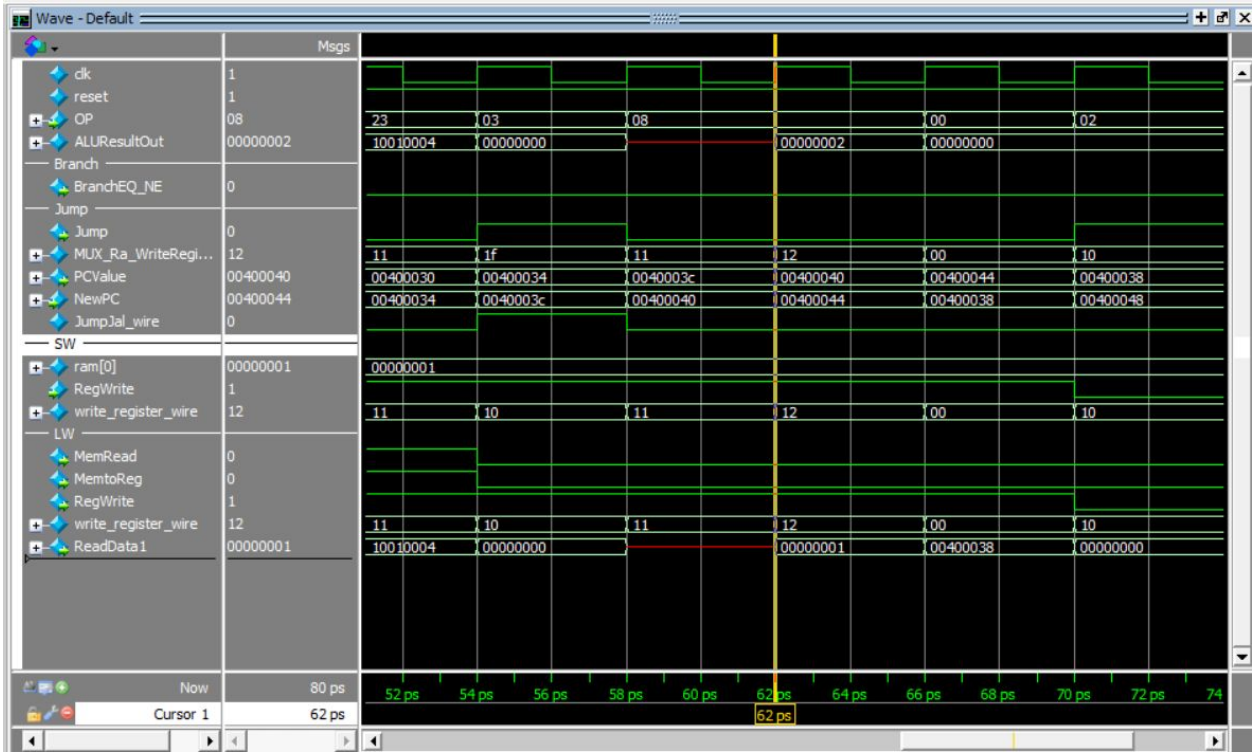
Jal: OP=3, JumpAdress=40003c (En el MUX_RA se ve cómo se elige R[31])



J: OP=2, JumpAdress=400048 (Es un j exit y termina el programa)

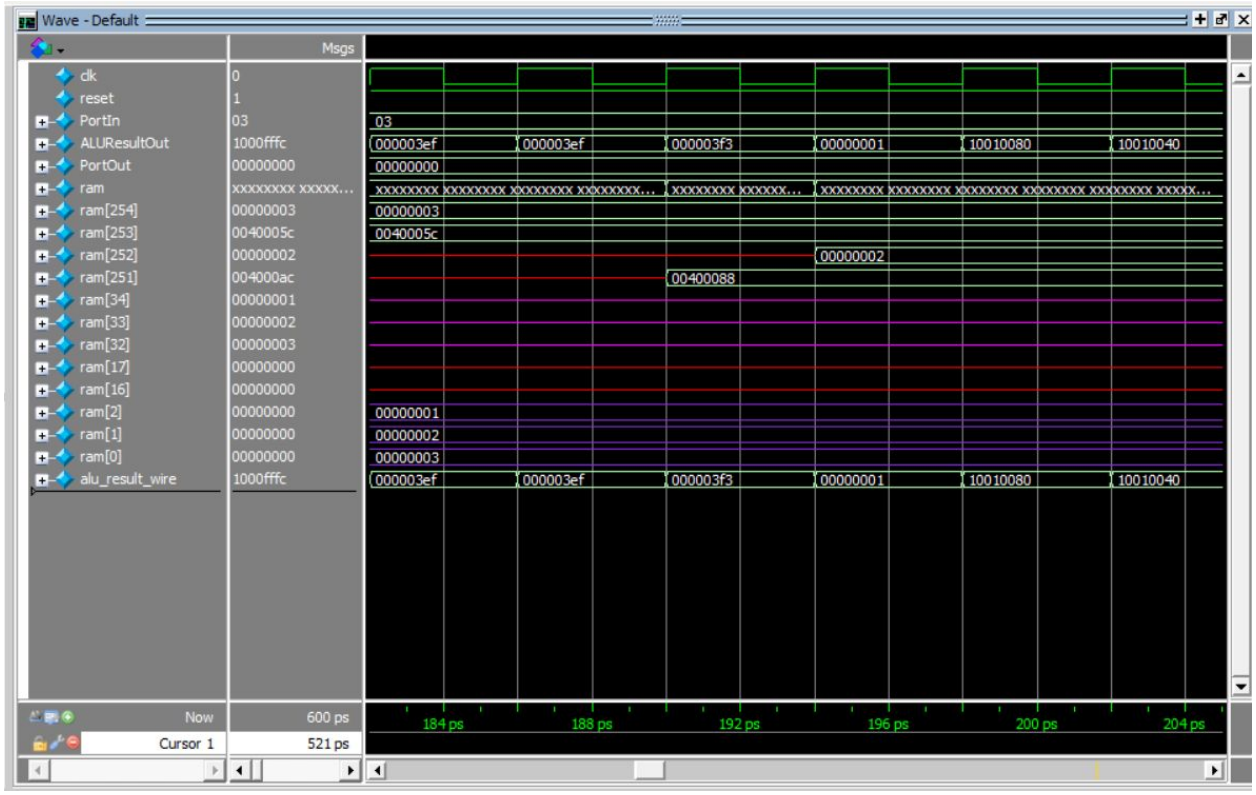


Jr: OP=8, JumpAdress=400038 (Hace un regreso en memoria, de 44 a 38)

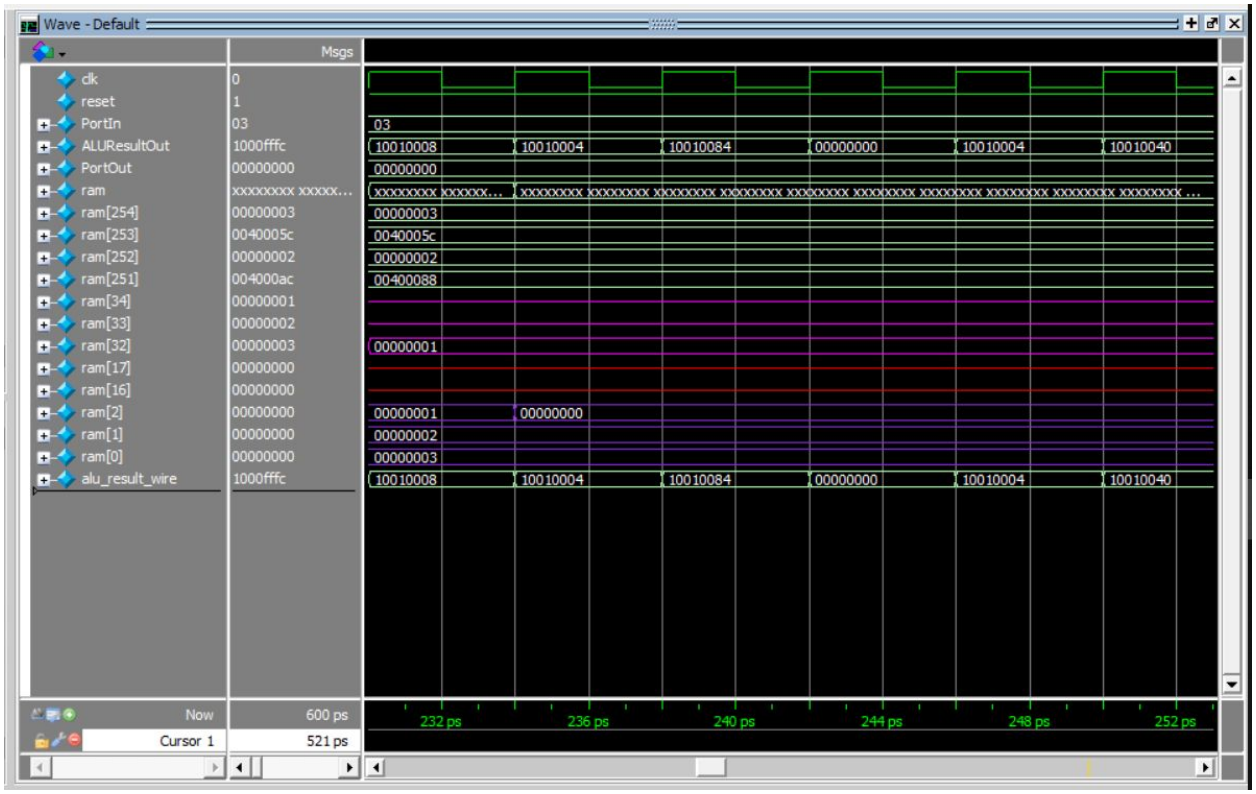


Simulación 3 Discos

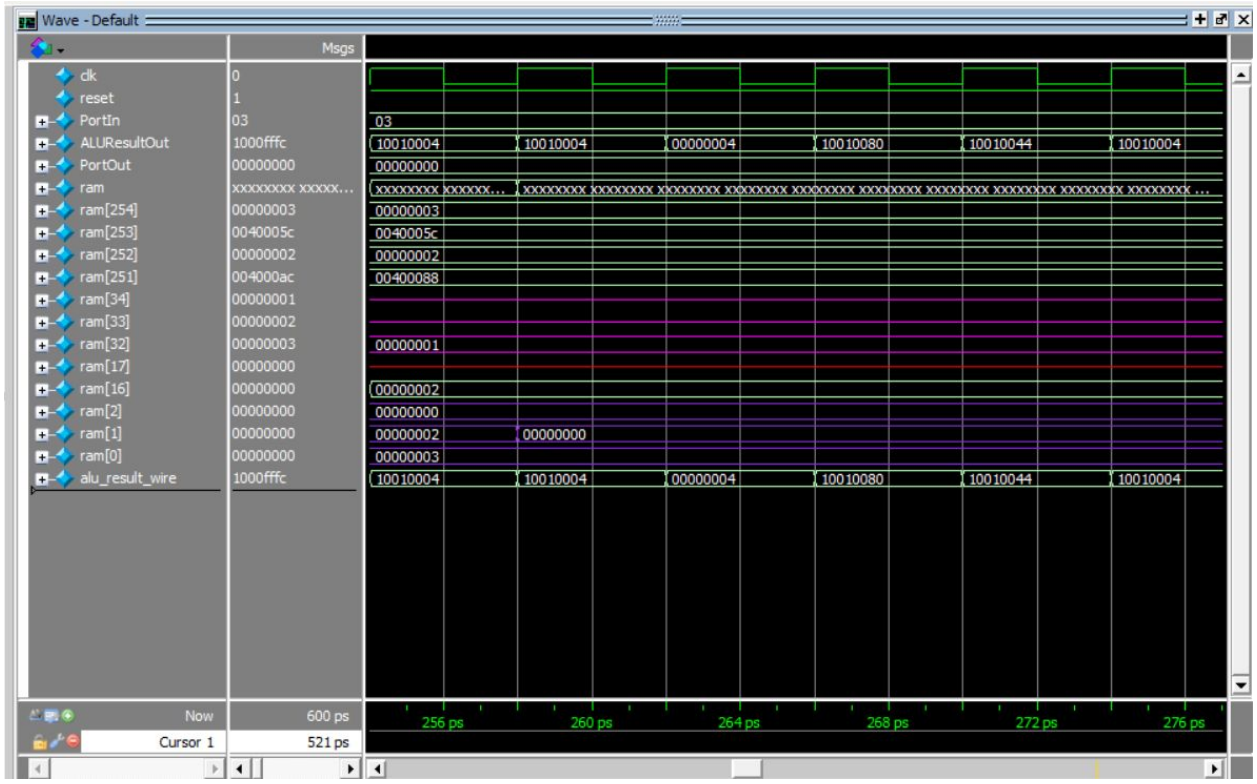
Comenzamos llenando la torre con los 3 discos que usaremos en Torre A



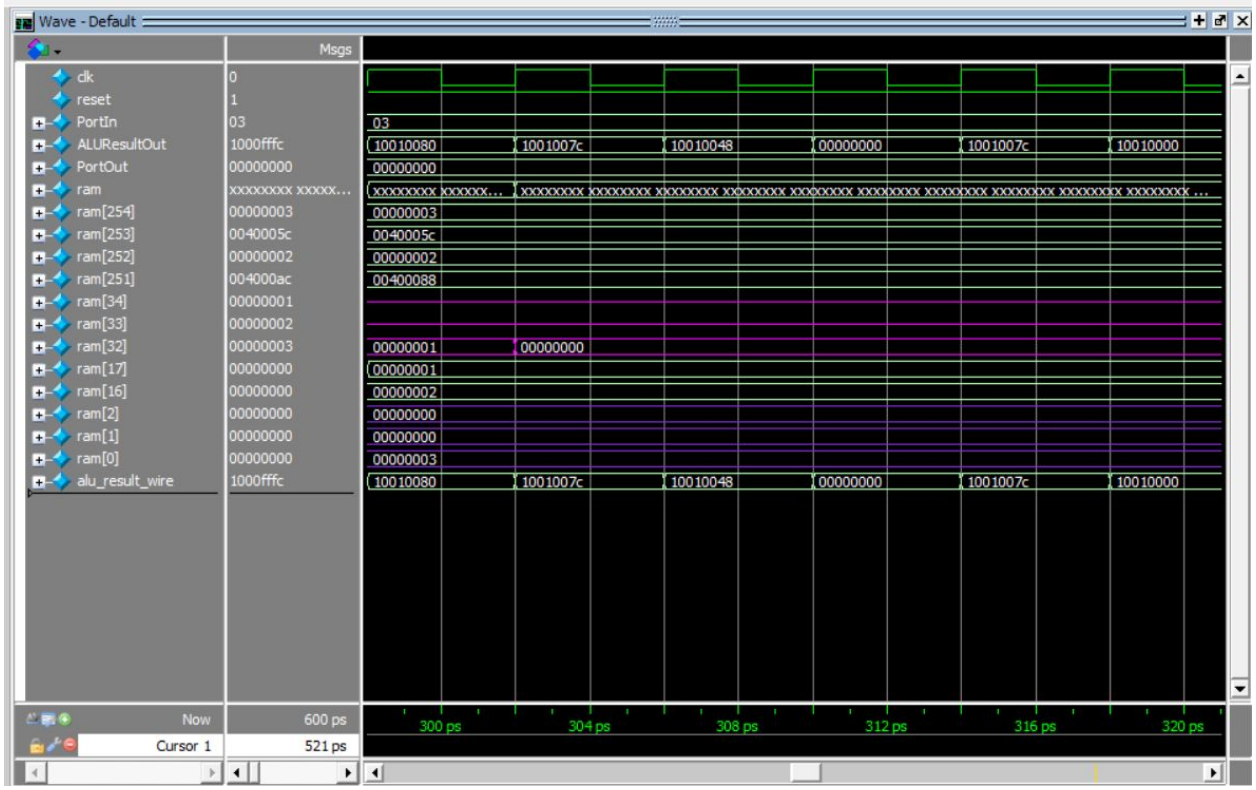
Primero movemos al registro de la torre C[0] el registro en el tope de Torre A



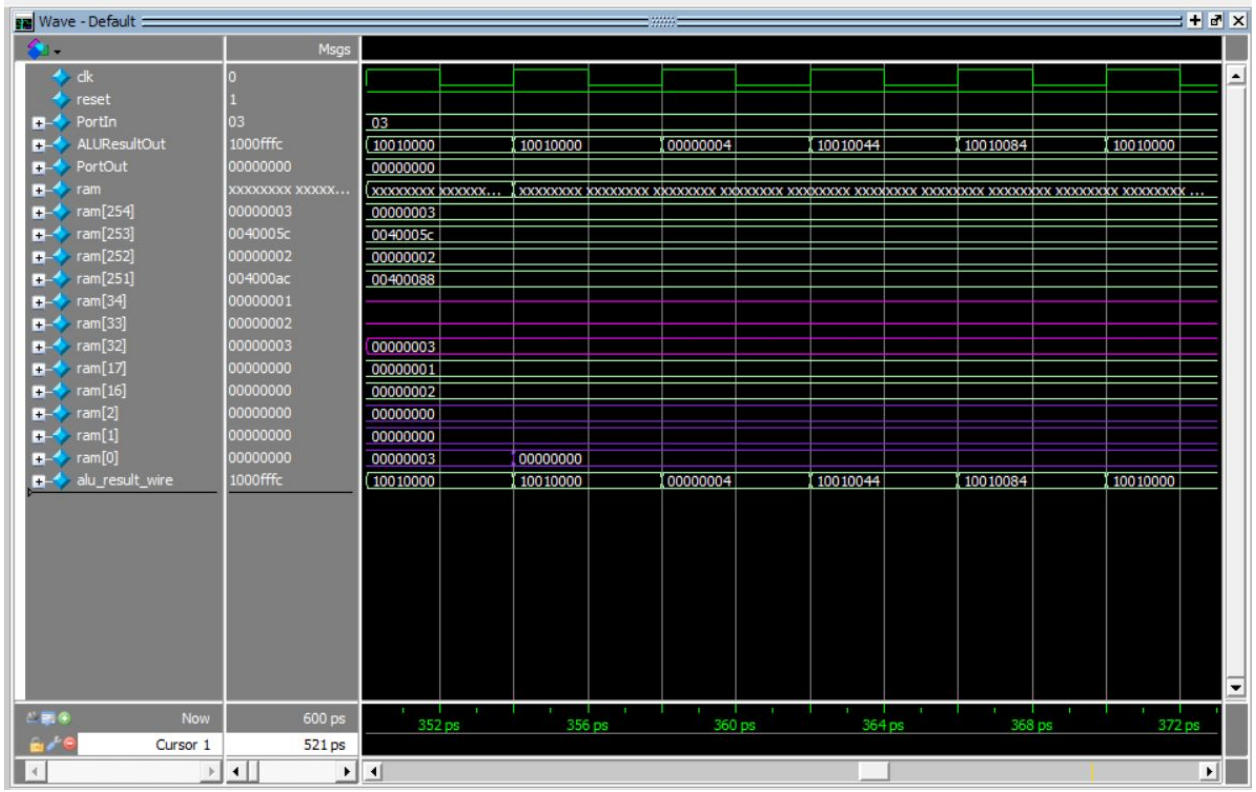
Luego movemos el nuevo tope de Torre A a torre B[0]



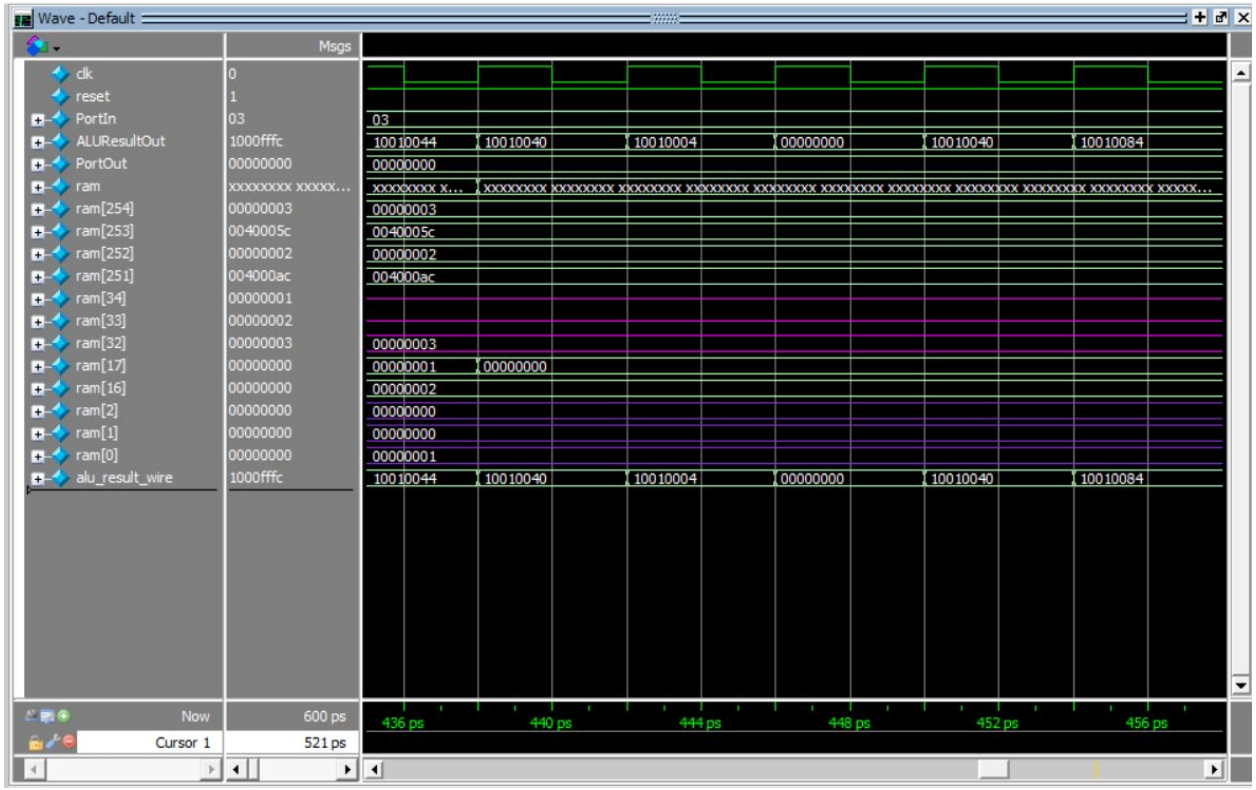
Luego movemos la base de Torre C al tope de torre B



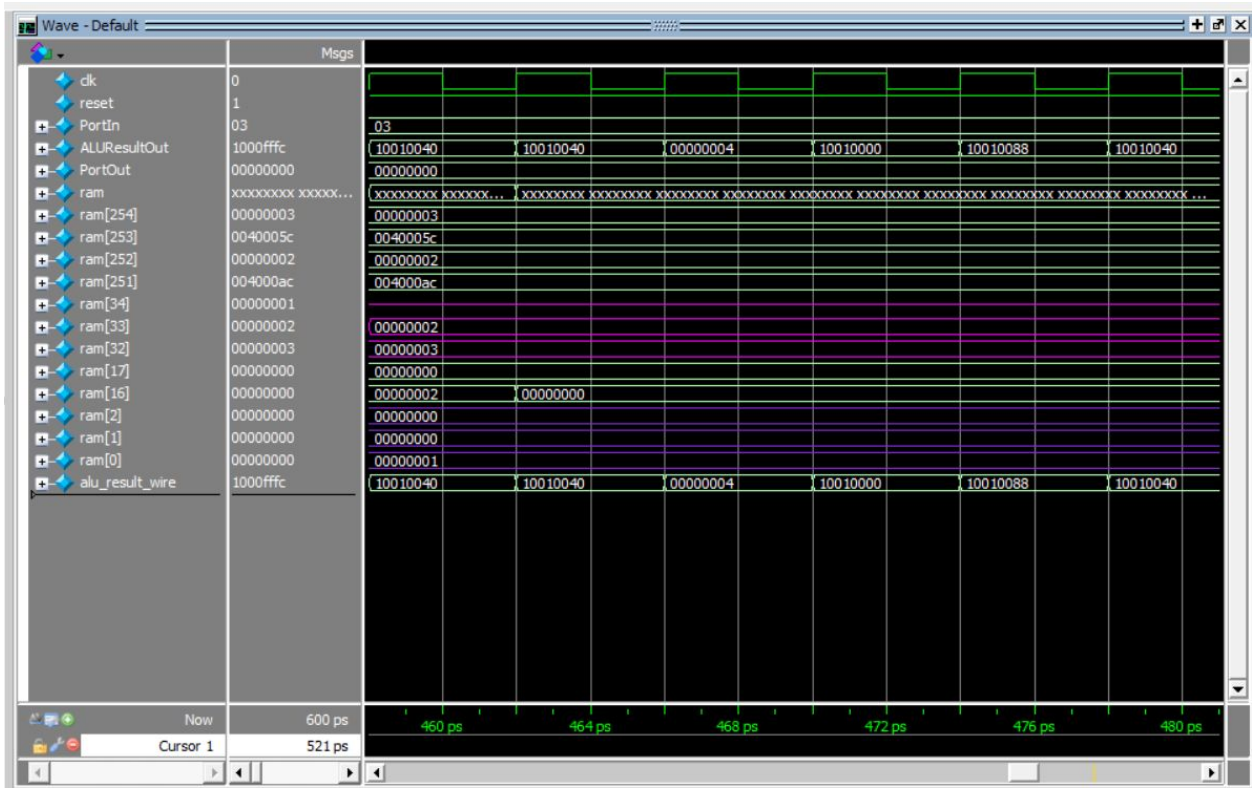
Movemos la base de Torre A a la base de Torre C



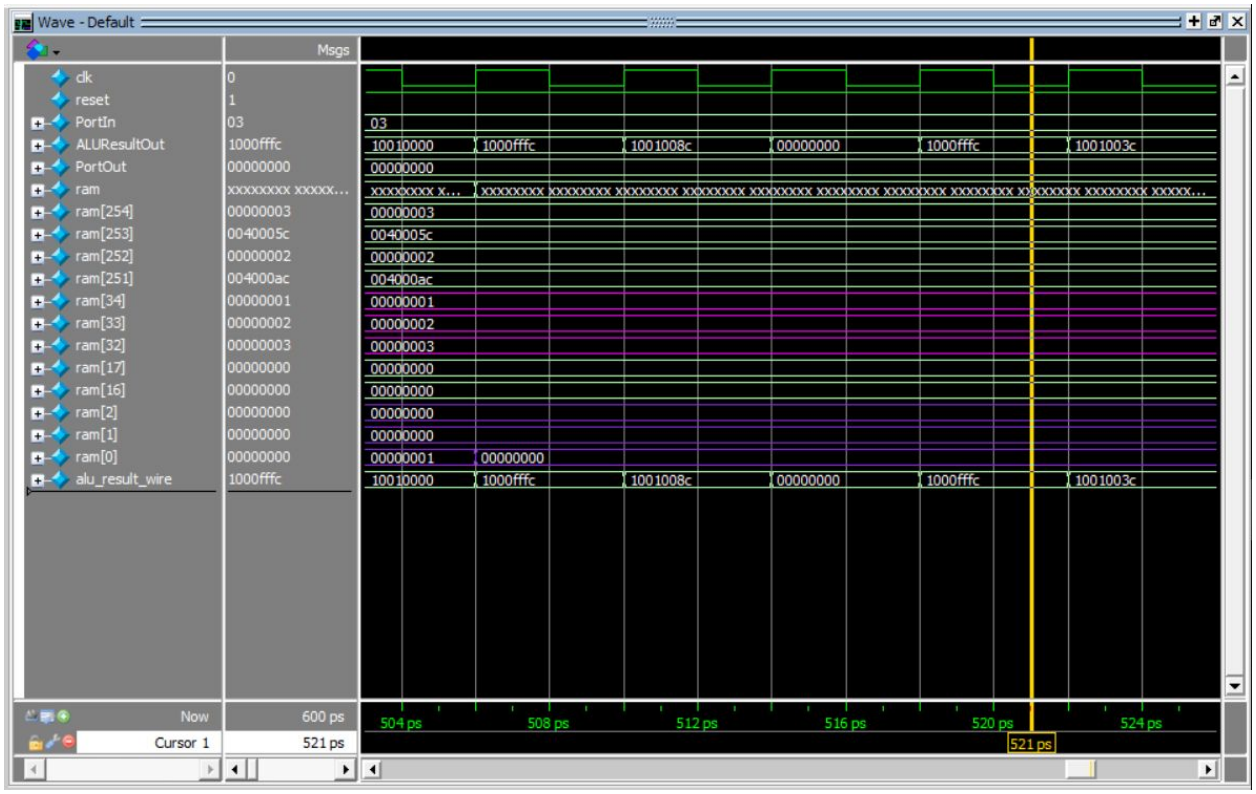
Movemos el tope de la torre B a la base de la Torre A



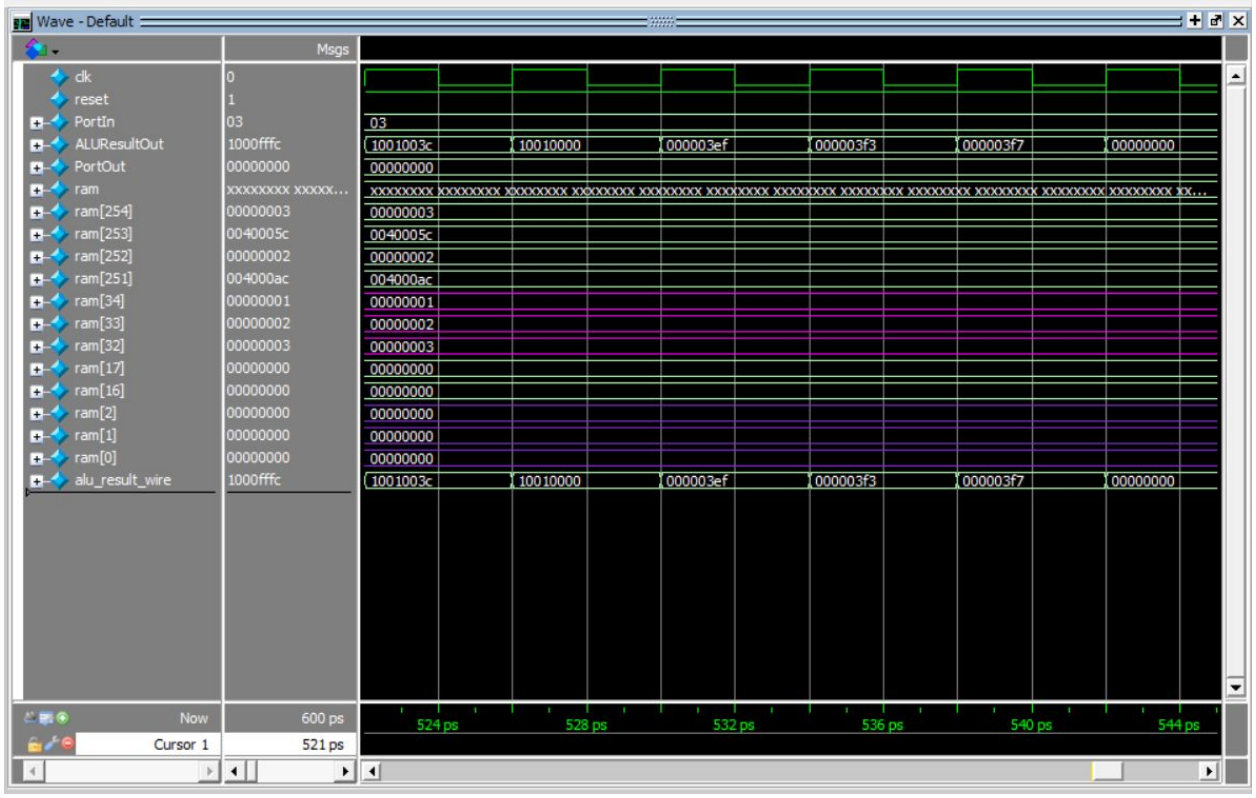
Movemos la base de Torre B al nuevo tope de Torre C



Movemos la Base restante de Torre A al Tope de Torre C



Finalmente tenemos nuestra torre de hanoi movida desde la Torre A en ram hasta la torre C en ram



Conclusiones

Karim Naciff

Con ésta práctica pude aprender a utilizar Quartus de una manera más avanzada y poder simular mejor en ModelSim, además pude practicar el datapath que conforma un MIPS single cycle, agregando, quitando y modificando cosas por parte propia. Me costó trabajo el agarrar el ritmo al principio pero a lo largo del proceso de la práctica se logró entender todo lo necesario.

Lilia Lobato

En la práctica logré entender el data path de las instrucciones así como su simulación dentro de ModelSim. Entendí la importancia de saber utilizar Verilog y entender cuando se necesita instanciar o cuando se necesita crear desde 0 o cuando simplemente se necesita agregar inputs.

Considero que fue una práctica pesada por al cantidad de conocimiento teórico que debe aplicarse y lo tardado que es el debug.