ITESO

Universidad Jesuita de Guadalajara

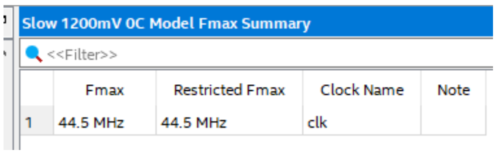
ARQUITECTURA DE COMPUTADORAS

Práctica 3

Raúl Méndez Álvarez

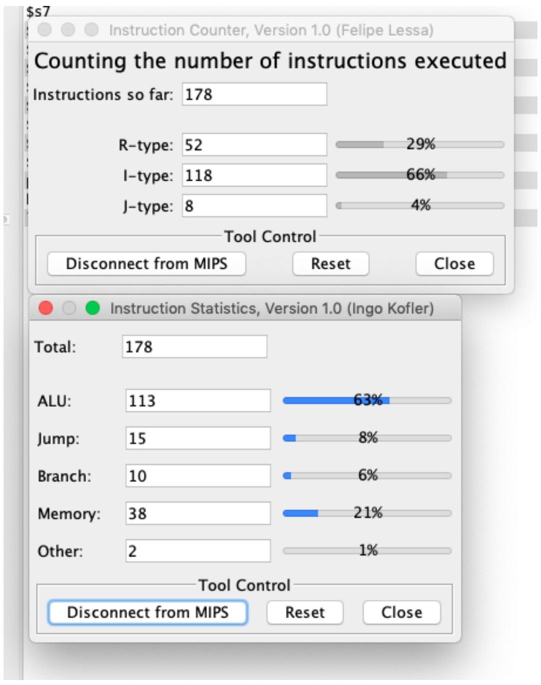
3/junio/2019

IC, CPI, Clock Rate and CPU time for the MIPS Implementation and type R, I and J instructions percentage.



Slow 1200mV 0C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	44.5 MHz	44.5 MHz	clk	



Instruction Counter, Version 1.0 (Felipe Lessa)

Counting the number of instructions executed

Instructions so far: 178

R-type:	52	29%
I-type:	118	66%
J-type:	8	4%

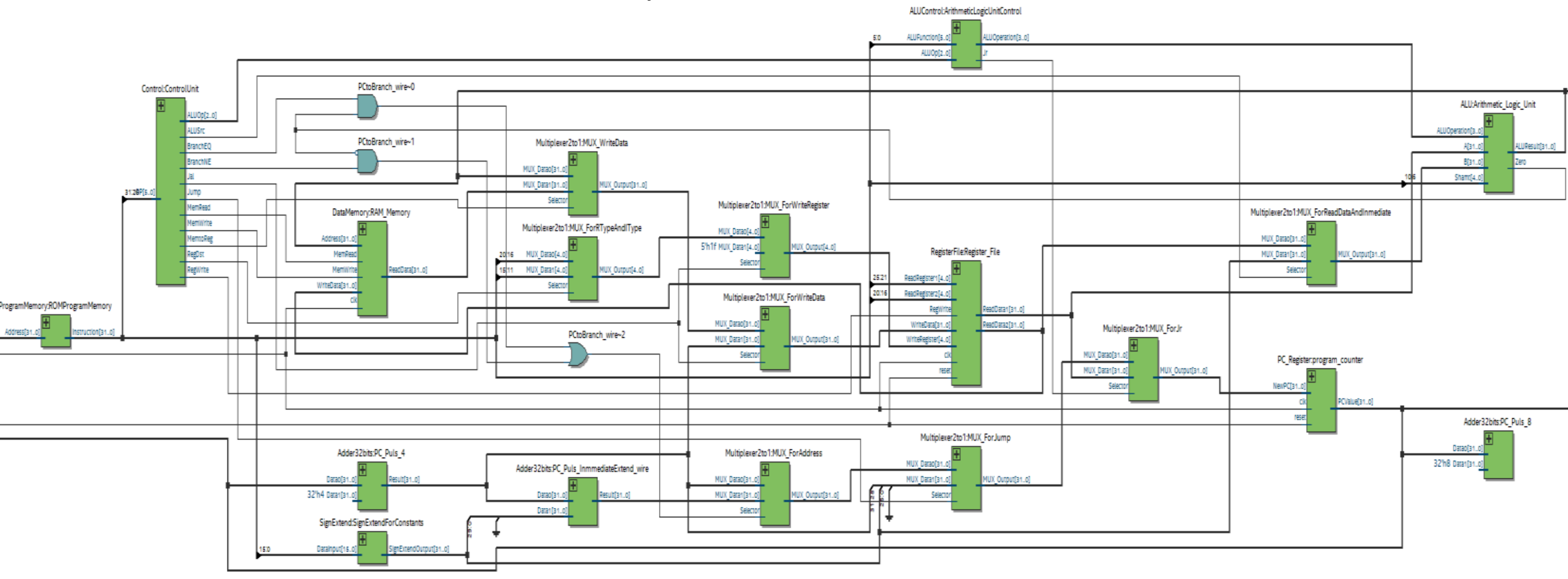
Instruction Statistics, Version 1.0 (Ingo Kofler)

Total: 178

ALU:	113	63%
Jump:	15	8%
Branch:	10	6%
Memory:	38	21%
Other:	2	1%

MIPS Micro-Architecture

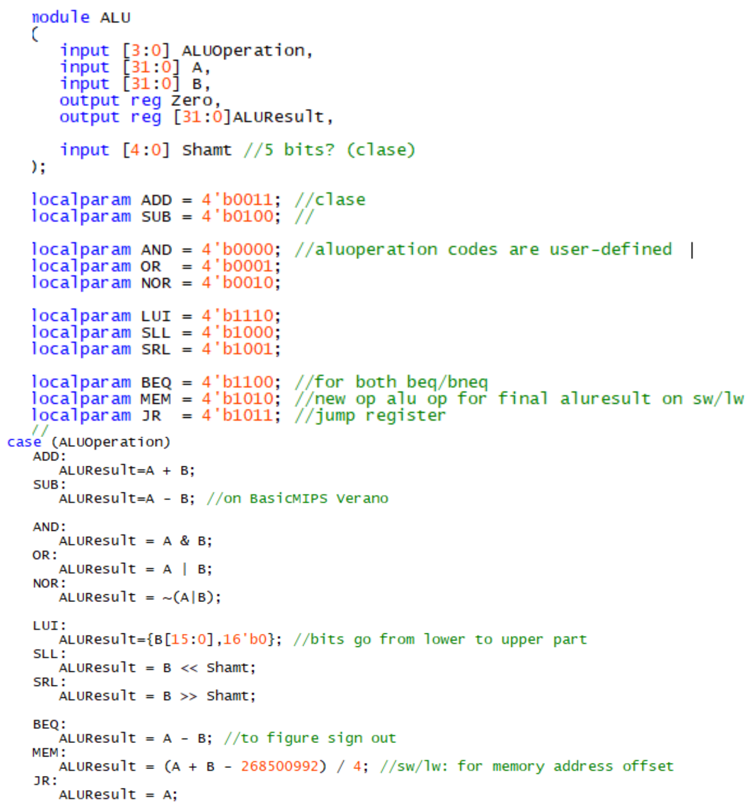
Feel free to look at the full-sized pdf included in the files



Modified Modules

ALU.V

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

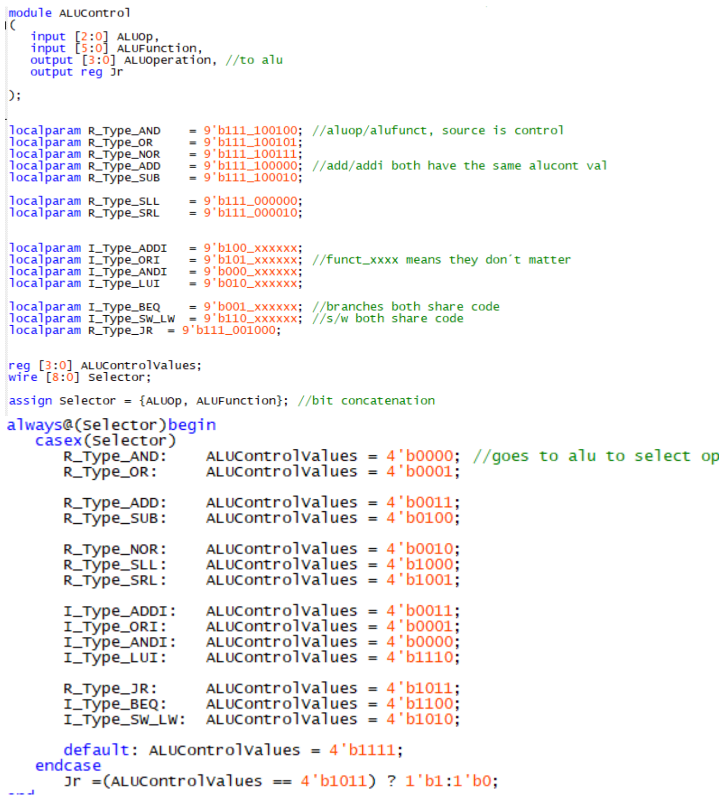


```
module ALU
(
   input [3:0] ALUOperation,
   input [31:0] A,
   input [31:0] B,
   output reg Zero,
   output reg [31:0]ALUResult,

   input [4:0] Shamt //5 bits? (clase)
);

localparam ADD = 4'b0011; //clase
localparam SUB = 4'b0100; //

localparam AND = 4'b0000; //aluoperation codes are user-defined  |
localparam OR  = 4'b0001;
localparam NOR = 4'b0010;

localparam LUI = 4'b1110;
localparam SLL = 4'b1000;
localparam SRL = 4'b1001;

localparam BEQ = 4'b1100; //for both beq/bneq
localparam MEM = 4'b1010; //new op alu op for final aluresult on sw/lw
localparam JR  = 4'b1011; //jump register
//
case (ALUOperation)
   ADD:
      ALUResult=A + B;
   SUB:
      ALUResult=A - B; //on BasicMIPS Verano

   AND:
      ALUResult = A & B;
   OR:
      ALUResult = A | B;
   NOR:
      ALUResult = ~(A|B);

   LUI:
      ALUResult={B[15:0],16'b0}; //bits go from lower to upper part
   SLL:
      ALUResult = B << Shamt;
   SRL:
      ALUResult = B >> Shamt;

   BEQ:
      ALUResult = A - B; //to figure sign out
   MEM:
      ALUResult = (A + B - 268500992) / 4; //sw/lw: for memory address offset
   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 MIPS´ Green Card.

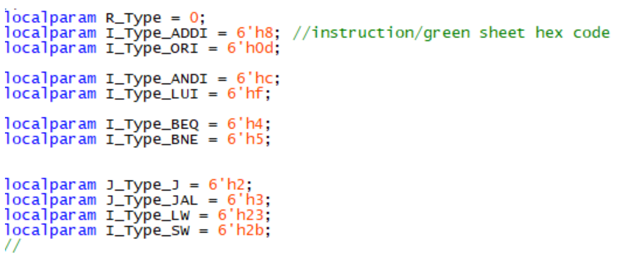
I added new cases in the selector for the implemented instructions and each instruction has its own ALUControlValue that was arbitrarily specified in Alu.v module as a localparam.



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

);

localparam R_Type_AND    = 9'b111_100100; //aluop/alufunct, source is control
localparam R_Type_OR     = 9'b111_100101;
localparam R_Type_NOR    = 9'b111_100111;
localparam R_Type_ADD    = 9'b111_100000; //add/addi both have the same alucont val
localparam R_Type_SUB    = 9'b111_100010;

localparam R_Type_SLL    = 9'b111_000000;
localparam R_Type_SRL    = 9'b111_000010;


localparam I_Type_ADDI   = 9'b100_xxxxxx;
localparam I_Type_ORI    = 9'b101_xxxxxx; //funct_xxxx means they don´t matter
localparam I_Type_ANDI   = 9'b000_xxxxxx;
localparam I_Type_LUI    = 9'b010_xxxxxx;

localparam I_Type_BEQ    = 9'b001_xxxxxx; //branches both share code
localparam I_Type_SW_LW  = 9'b110_xxxxxx; //s/w both share code
localparam R_Type_JR  = 9'b111_001000;


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

assign Selector = {ALUOp, ALUFunction}; //bit concatenation

always@(Selector)begin
   casex(Selector)
      R_Type_AND:    ALUControlValues = 4'b0000; //goes to alu to select op
      R_Type_OR:     ALUControlValues = 4'b0001;

      R_Type_ADD:    ALUControlValues = 4'b0011;
      R_Type_SUB:    ALUControlValues = 4'b0100;

      R_Type_NOR:    ALUControlValues = 4'b0010;
      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_LUI:    ALUControlValues = 4'b1110;

      R_Type_JR:     ALUControlValues = 4'b1011;
      I_Type_BEQ:    ALUControlValues = 4'b1100;
      I_Type_SW_LW:  ALUControlValues = 4'b1010;

      default: ALUControlValues = 4'b1111;
   endcase
      Jr =(ALUControlValues == 4'b1011) ? 1'b1:1'b0;
```

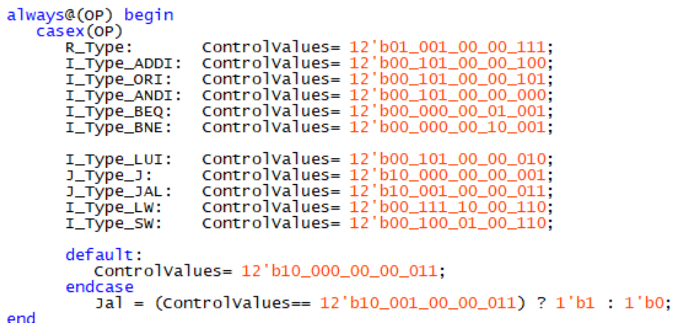
Control.v

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



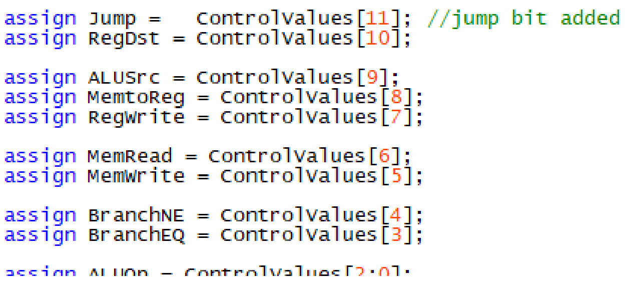
```verilog
localparam R_Type = 0;
localparam I_Type_ADDI = 6'h8; //instruction/green sheet hex code
localparam I_Type_ORI = 6'h0d;

localparam I_Type_ANDI = 6'hc;
localparam I_Type_LUI = 6'hf;

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_LW = 6'h23;
localparam I_Type_SW = 6'h2b;
//
```

I 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.



```verilog
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;
end
```

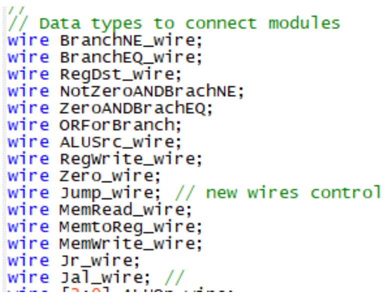
Each bit of the control value has its own meaning and it is specified in this part of the module.



```verilog
assign Jump =   ControlValues[11]; //jump bit added
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

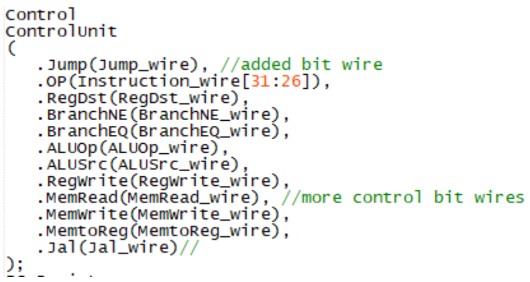
This module contains all the connections in the processor, in the top of this module I added all the new wires that I needed to connect the each module correctly with each other and the signals for each multiplexer.



```
//
// 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; // new wires control
wire MemRead_wire;
wire MemtoReg_wire;
wire MemWrite_wire;
wire Jr_wire;
wire Jal_wire; //
```

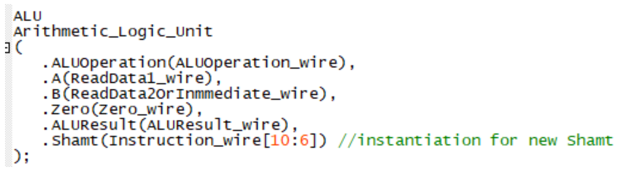
I modified some of the modules that were already instantiated with new connections with the new wires and I also instantiated some more that I needed in order to connect all the modules together.

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



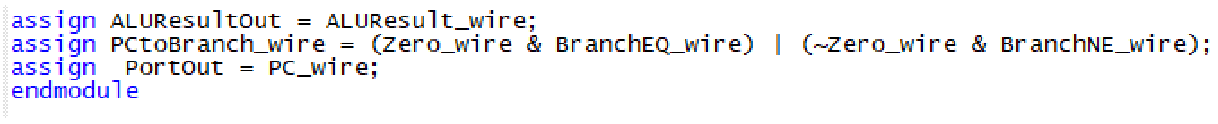
```
Control
ControlUnit
(
    .Jump(Jump_wire), //added bit wire
    .OP(Instruction_wire[31:26]),
    .RegDst(RegDst_wire),
    .BranchNE(BranchNE_wire),
    .BranchEQ(BranchEQ_wire),
    .ALUOp(ALUOp_wire),
    .ALUSrc(ALUSrc_wire),
    .RegWrite(RegWrite_wire),
    .MemRead(MemRead_wire), //more control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

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



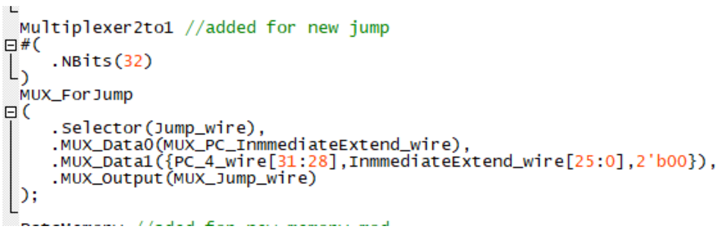
```
ALU
Arithmetic_Logic_Unit
(
    .ALUOperation(ALUOperation_wire),
    .A(ReadData1_wire),
    .B(ReadData2OrInmmediate_wire),
    .Zero(Zero_wire),
    .ALUResult(ALUResult_wire),
    .Shamt(Instruction_wire[10:6]) //instantiation for new Shamt
);
```

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



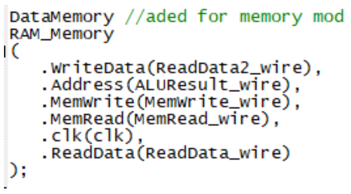
```
assign ALUResultOut = ALUResult_wire;
assign PCtoBranch_wire = (Zero_wire & BranchEQ_wire) | (~Zero_wire & BranchNE_wire);
assign  PortOut = PC_wire;
endmodule
```

I instantiated a new MUX for the jump instruction.



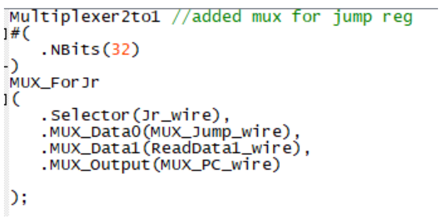
```
Multiplexer2to1 //added for new jump
#(
	.NBits(32)
)
MUX_ForJump
(
	.Selector(Jump_wire),
	.MUX_Data0(MUX_PC_InmmediateExtend_wire),
	.MUX_Data1({PC_4_wire[31:28],InmmediateExtend_wire[25:0],2'b00}),
	.MUX_Output(MUX_Jump_wire)
);
```

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



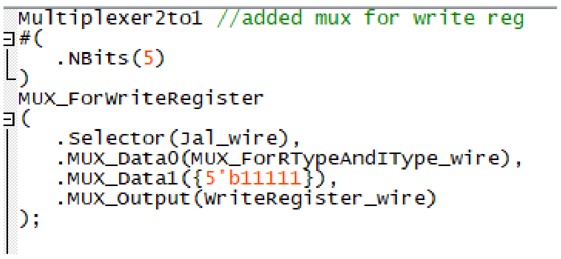
```
DataMemory //aded for memory mod
RAM_Memory
(
	.WriteData(ReadData2_wire),
	.Address(ALUResult_wire),
	.MemWrite(MemWrite_wire),
	.MemRead(MemRead_wire),
	.clk(clk),
	.ReadData(ReadData_wire)
);
```

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



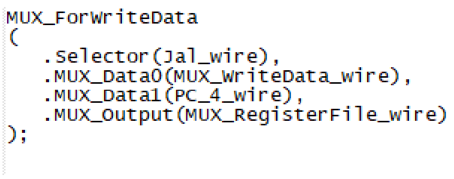
```
Multiplexer2to1 //added mux for jump reg
#(
	.NBits(32)
)
MUX_ForJr
(
	.Selector(Jr_wire),
	.MUX_Data0(MUX_Jump_wire),
	.MUX_Data1(ReadData1_wire),
	.MUX_Output(MUX_PC_wire)

);
```

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



```
Multiplexer2to1 //added mux for write reg
#(
   .NBits(5)
)
MUX_ForWriteRegister
(
   .Selector(Jal_wire),
   .MUX_Data0(MUX_ForRTypeAndIType_wire),
   .MUX_Data1({5'b11111}),
   .MUX_Output(WriteRegister_wire)
);
```

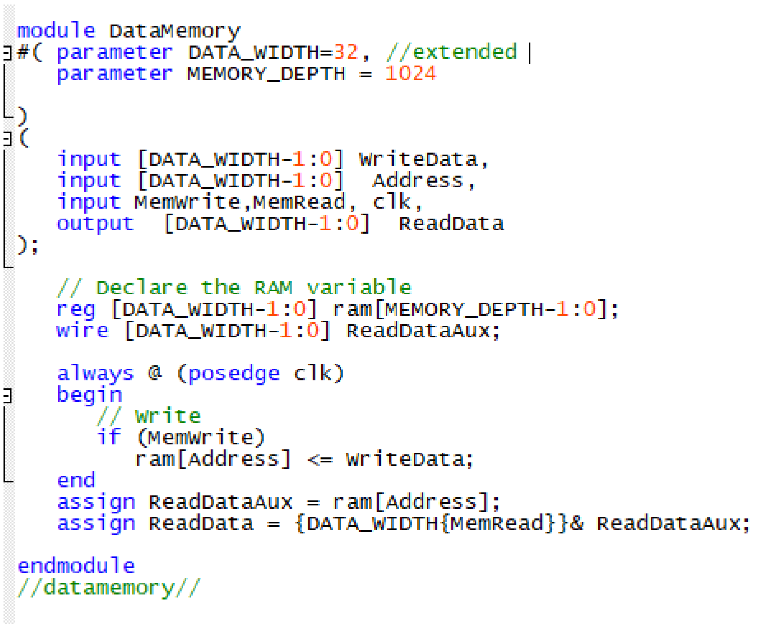
New MUX for the WriteData signal



```
MUX_ForWriteData
(
   .Selector(Jal_wire),
   .MUX_Data0(MUX_WriteData_wire),
   .MUX_Data1(PC_4_wire),
   .MUX_Output(MUX_RegisterFile_wire)
);
```

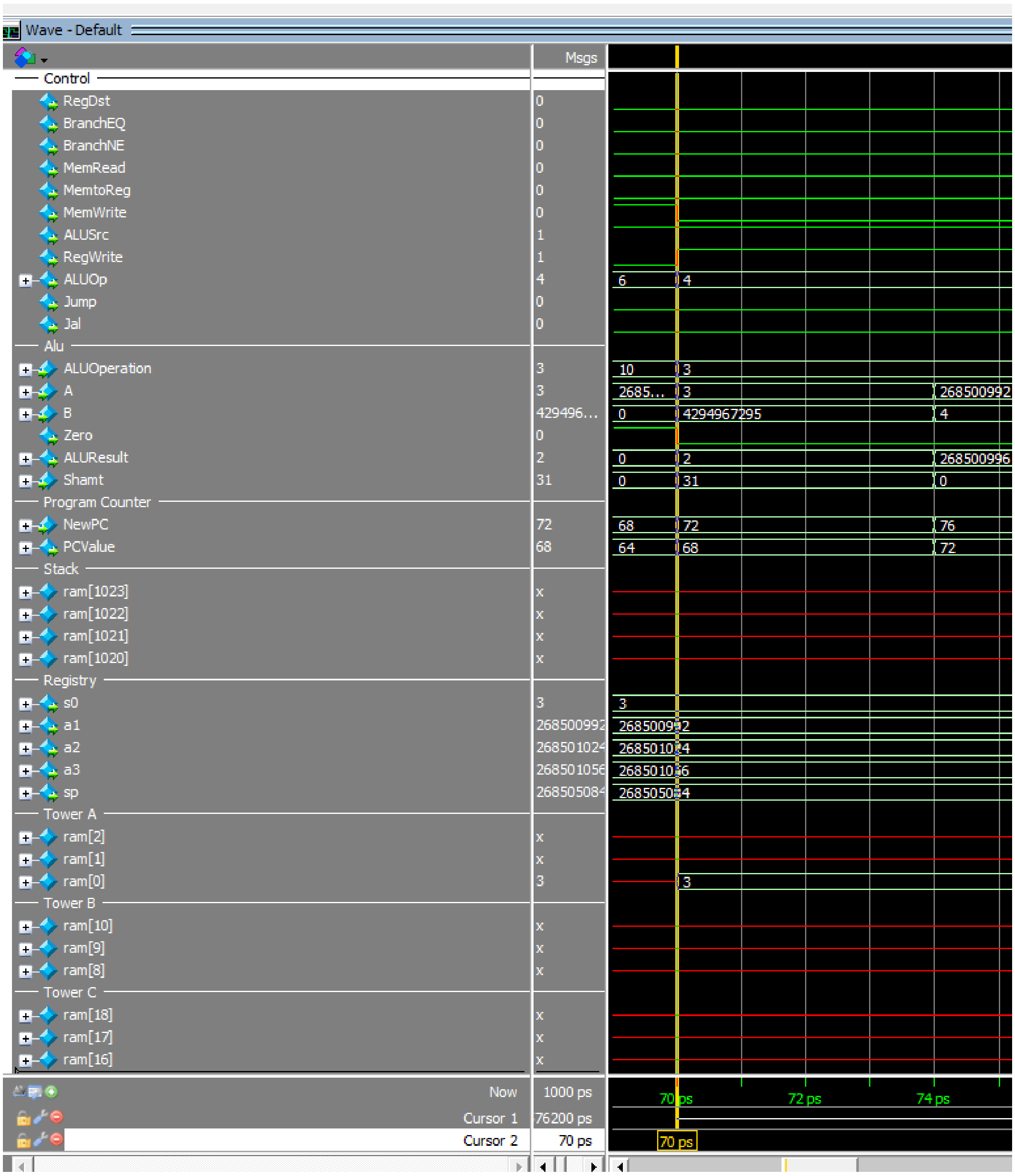
DataMemory.v

We changed the parameter DATA_WIDTH from 8 to 32.



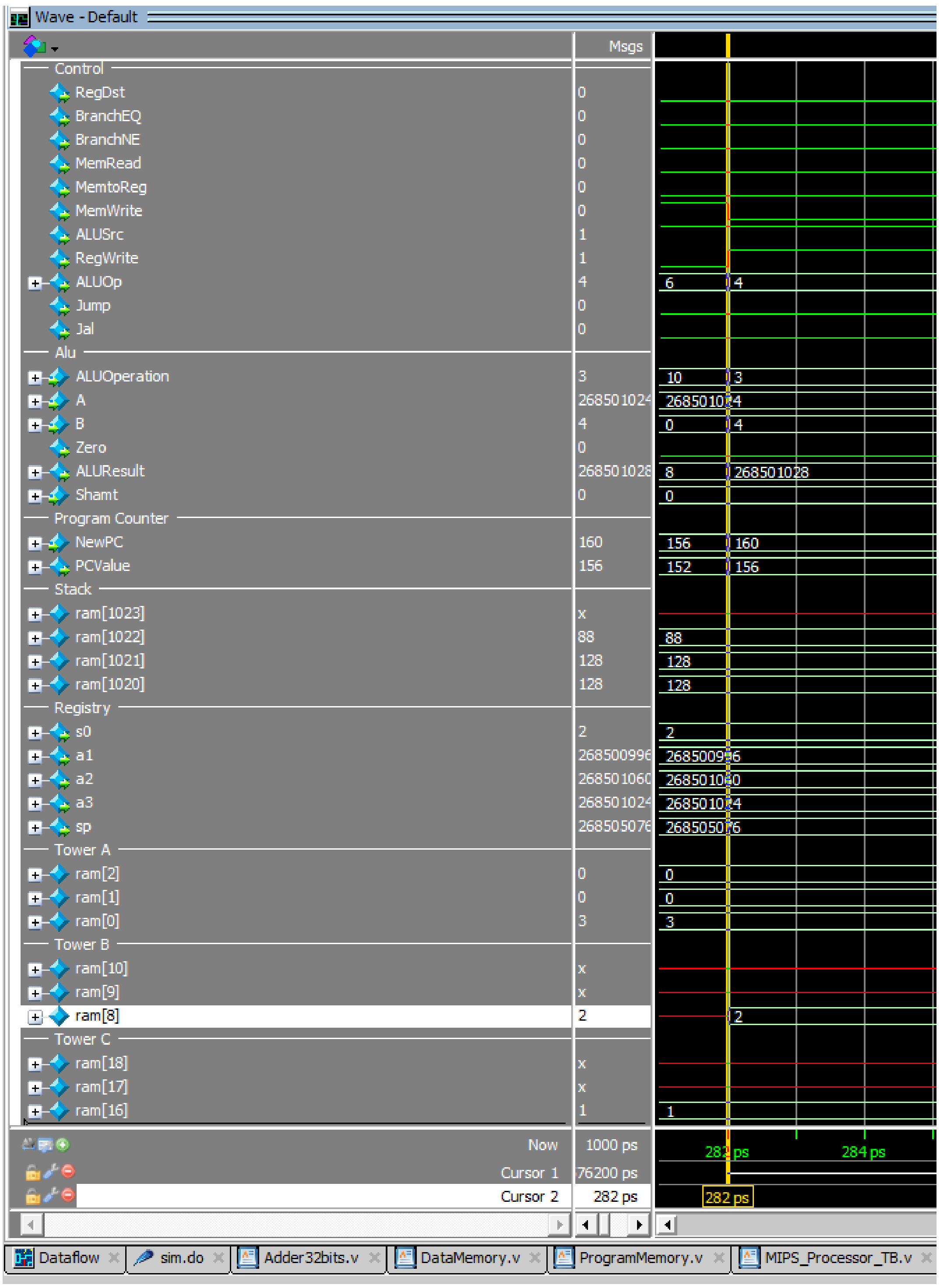
```
module DataMemory
#( parameter DATA_WIDTH=32, //extended
   parameter MEMORY_DEPTH = 1024

)
(
   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
//datamemory//
```

ModelSim MIPS Simulation

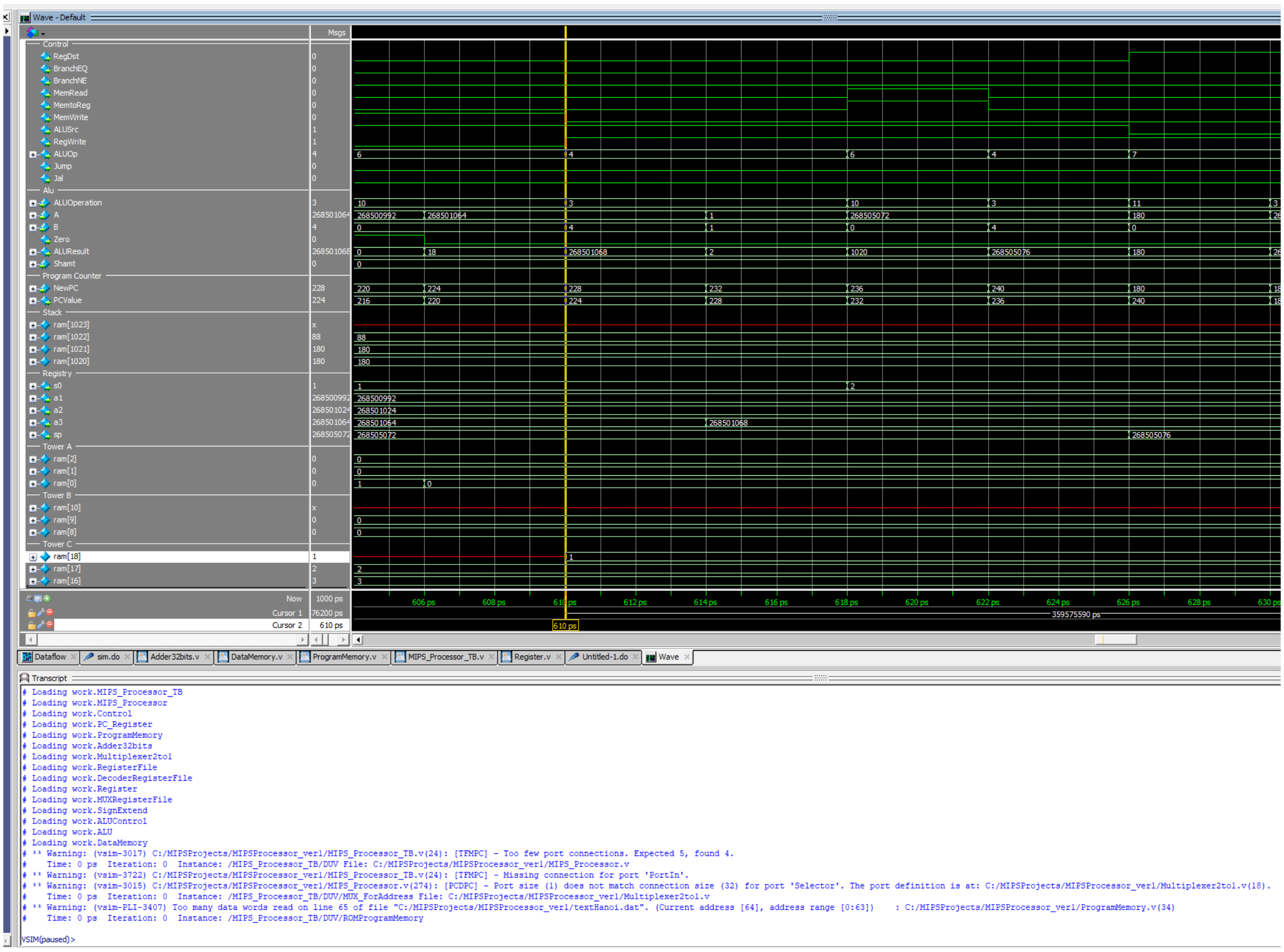


```
Wave - Default
                         Msgs
Control
  RegDst                 0
  BranchEQ               0
  BranchNE               0
  MemRead                0
  MemtoReg               0
  MemWrite               0
  ALUSrc                 1
  RegWrite               1
  ALUOp                  4            6         4
  Jump                   0
  Jal                    0
Alu
  ALUOperation           3            10        3
  A                      3            2685...   3                   268500992
  B                      429496...    0         4294967295          4
  Zero                   0
  ALUResult              2            0         2                   268500996
  Shamt                  31           0         31                  0
Program Counter
  NewPC                  72           68        72                  76
  PCValue                68           64        68                  72
Stack
  ram[1023]              x
  ram[1022]              x
  ram[1021]              x
  ram[1020]              x
Registry
  s0                     3            3
  a1                     268500992    268500992
  a2                     268501024    268501024
  a3                     268501056    268501056
  sp                     268505084    268505084
Tower A
  ram[2]                 x
  ram[1]                 x
  ram[0]                 3                      3
Tower B
  ram[10]                x
  ram[9]                 x
  ram[8]                 x
Tower C
  ram[18]                x
  ram[17]                x
  ram[16]                x

                  Now    1000 ps      70 ps        72 ps        74 ps
             Cursor 1    76200 ps
             Cursor 2    70 ps        70 ps
```

The ps pointer/line is marked at precisely the moment/cycle where the disk with disk #3 would be placed at its initial tower A position.


```
Wave - Default
                              Msgs
Control
  RegDst                      0
  BranchEQ                    0
  BranchNE                    0
  MemRead                     0
  MemtoReg                    0
  MemWrite                    0
  ALUSrc                      1
  RegWrite                    1
+ ALUOp                       4           6 | 4
  Jump                        0
  Jal                         0
Alu
+ ALUOperation                3           10 | 3
+ A                           268501024   268501024
+ B                           4           0 | 4
  Zero                        0
+ ALUResult                   268501028   8 | 268501028
+ Shamt                       0           0
Program Counter
+ NewPC                       160         156 | 160
+ PCValue                     156         152 | 156
Stack
+ ram[1023]                   x
+ ram[1022]                   88          88
+ ram[1021]                   128         128
+ ram[1020]                   128         128
Registry
+ s0                          2           2
+ a1                          268500996   268500996
+ a2                          268501060   268501060
+ a3                          268501024   268501024
+ sp                          268505076   268505076
Tower A
+ ram[2]                      0           0
+ ram[1]                      0           0
+ ram[0]                      3           3
Tower B
+ ram[10]                     x
+ ram[9]                      x
+ ram[8]                      2                 | 2
Tower C
+ ram[18]                     x
+ ram[17]                     x
+ ram[16]                     1           1

                    Now       1000 ps     282 ps       284 ps
                    Cursor 1  76200 ps
                    Cursor 2  282 ps      282 ps

Dataflow | sim.do | Adder32bits.v | DataMemory.v | ProgramMemory.v | MIPS_Processor_TB.v
```

The ps pointer/line is marked at precisely the moment/cycle where the disk with disk #2 would be placed at its mid-process tower B position.

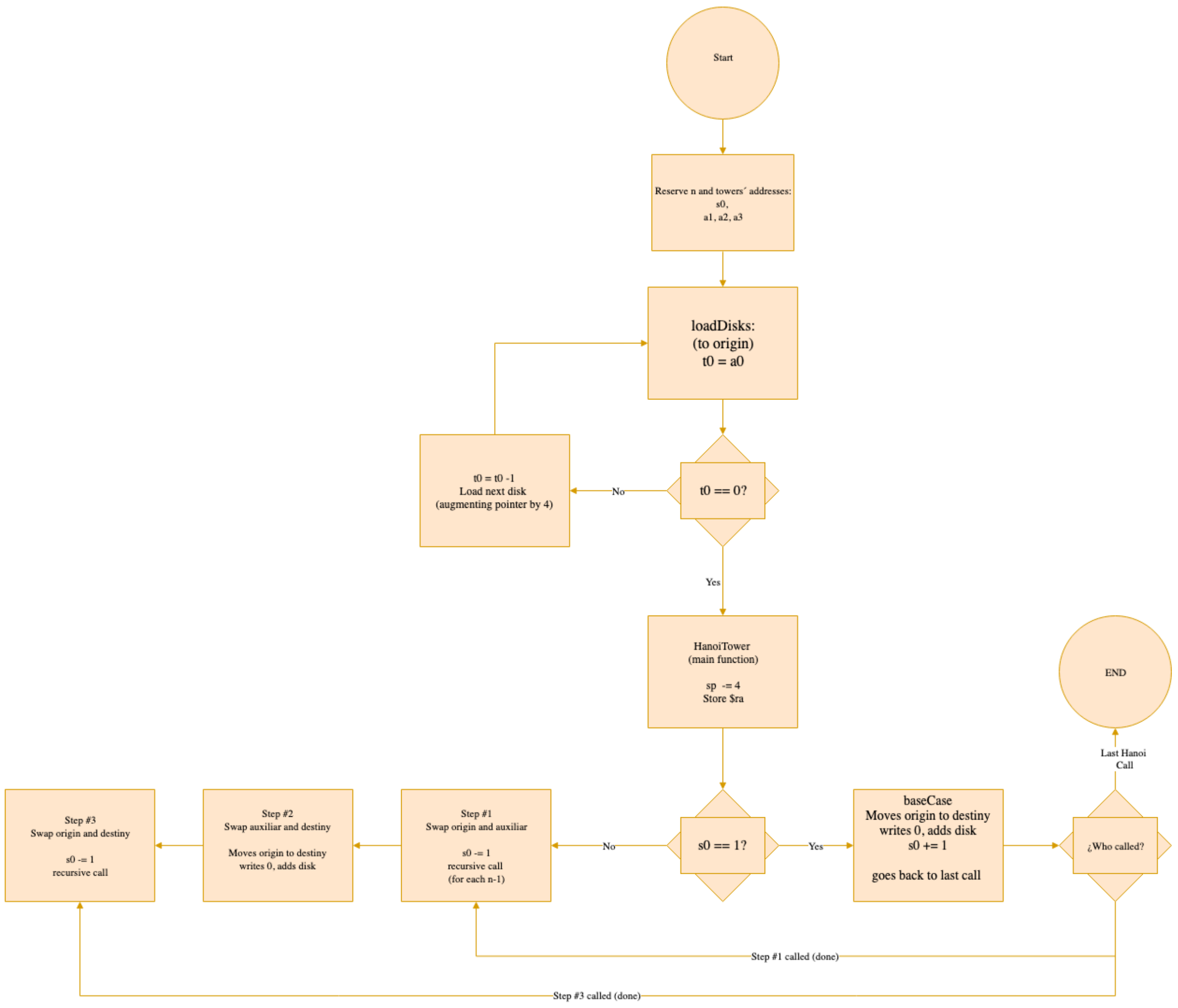


```
# Loading work.MIPS_Processor_TB
# Loading work.MIPS_Processor
# Loading work.Control
# Loading work.PC_Register
# Loading work.ProgramMemory
# Loading work.Adder32bits
# Loading work.Multiplexer2to1
# Loading work.RegisterFile
# Loading work.DecoderRegisterFile
# Loading work.Register
# Loading work.MUXRegisterFile
# Loading work.SignExtend
# Loading work.ALUControl
# Loading work.ALU
# Loading work.DataMemory
# ** Warning: (vsim-3017) C:/MIPSProjects/MIPSProcessor_ver1/MIPS_Processor_TB.v(24): [TFMPC] - Too few port connections. Expected 5, found 4.
#    Time: 0 ps  Iteration: 0  Instance: /MIPS_Processor_TB/DUV File: C:/MIPSProjects/MIPSProcessor_ver1/MIPS_Processor.v
# ** Warning: (vsim-3722) C:/MIPSProjects/MIPSProcessor_ver1/MIPS_Processor_TB.v(24): [TFMPC] - Missing connection for port 'PortIn'.
# ** Warning: (vsim-3015) C:/MIPSProjects/MIPSProcessor_ver1/MIPS_Processor.v(274): [PCDPC] - Port size (1) does not match connection size (32) for port 'Selector'. The port definition is at: C:/MIPSProjects/MIPSProcessor_ver1/Multiplexer2to1.v(18).
#    Time: 0 ps  Iteration: 0  Instance: /MIPS_Processor_TB/DUV/MUX_ForAddress File: C:/MIPSProjects/MIPSProcessor_ver1/Multiplexer2to1.v
# ** Warning: (vsim-PLI-3407) Too many data words read on line 65 of file "C:/MIPSProjects/MIPSProcessor_ver1/textHanoi.dat". (Current address [64], address range [0:63])    : C:/MIPSProjects/MIPSProcessor_ver1/ProgramMemory.v(34)
#    Time: 0 ps  Iteration: 0  Instance: /MIPS_Processor_TB/DUV/ROMProgramMemory

VSIM(paused)>
```

The ps pointer/line is marked at precisely the moment/cycle where the disk with disk #1 would be placed at its final tower C position.

Feel free to enlarge the image to view all of the registers´ finals status (assuming the document format allows you to), as well as to run the simulation yourself on ModelSim (files included).

Flow


Start
Reserve n and towers´ addresses: s0, a1, a2, a3
loadDisks: (to origin) t0 = a0
t0 == 0?
No
t0 = t0 -1 Load next disk (augmenting pointer by 4)
Yes
HanoiTower (main function) sp -= 4 Store $ra
s0 == 1?
No
Step #1 Swap origin and auxiliar s0 -= 1 recursive call (for each n-1)
Step #2 Swap auxiliar and destiny Moves origin to destiny writes 0, adds disk
Step #3 Swap origin and destiny s0 -= 1 recursive call
Yes
baseCase Moves origin to destiny writes 0, adds disk s0 += 1 goes back to last call
¿Who called?
Last Hanoi Call
END
Step #1 called (done)
Step #3 called (done)

Flow description:

My algorithm would begin by associating each of the addresses to each of its respective tower as well as register. We´d also always have n stored in s0, even though such variable is inputed by the user beforehand.

The loadDisks function or subroutine would then begin to load each and every disk onto its respective space or place on tower A/origin tower until the amount of disks are complete. And then it would proceed to call the HanoiTower function for the first time, which would initially allocate the memory necessary to store the ra registry onto the top stack position.

The function would first try to validate whether the amount of disks is or isn´t 1 in order to jump to the base case (which would move the respective tower to its according position) or to the internal function steps (which would swap the pointers or addresses on each of the towers accordingly in order to follow the algorithm described on the assignment).

The process would loop in and out of the steps, unwinding and then rewinding when its ready to swap the disks to their respective next tower, step by step, all by switching the auxiliary tower accordingly. Once the algorithms takes us to the last step or to the top of the rewinding process, it would put the last disk onto its final place and finally it would restore the size of the sp as well as s0 to finally be done with the process.