# Apéndice A: RTL de la unidad de Control

```
`timescale 1ns / 1ps
module control_unit(
  input logic [6:0]  opcode,              //instrucctions [6:0]
  input logic [2:0]  funct_3,             //instructions [14:12]
  input logic [6:0]  funct_7,             //instructions [31:25]
  input logic        zero,              //input from branch(module) if rs1 and rs2 are equal
  output logic       prf_wr_en,           //write enable of prf
  output logic [2:0] cu_imm_sel,          //control slector to immediate generator(imm_gen)
  output logic       prf_pc_mux_ctrl,     //selector to control input in ALU opoerand 1
  output logic       prf_imm_mux_ctrl,    //selector to control input in ALU opereand 2
  output logic [3:0] cu_alu_ctrl,         //selector for ALU operation
  output logic [1:0] cu_mem_out_mux_sel, //selector for mux data_out,alu_result, next_instruction directions to prf data in
  output logic       cu_data_mem_wr_en,   //write enable for data mem
  output logic       cu_pc_add_sel,       //selector to add +4 or +2 to the next instrucion (+2 is for future architecture implementation)
  output logic       branch_taken         //flag in order to evaluate only in  BEQ and JAL instructions
);
//Define the instructions
`include "defines.svh"
  always_comb begin
   prf_wr_en =             1'b1;          //enable prf write
     cu_data_mem_wr_en =     1'b0;          //disable data_mem write
     cu_imm_sel =            IMM_I;         //use IMM_I_TYPE decodification
     prf_pc_mux_ctrl =       1'b0;          //use rs1 as opreand 1 in ALU
     prf_imm_mux_ctrl =      1'b1;          //don't use rs2 instead use imm_gen as operand 2 in ALU
     cu_pc_add_sel =         1'b0;          //Add +4 to pc_in
     cu_mem_out_mux_sel =    ALU_TO_PRF;    //send the result of ALU to prf data_in
     cu_alu_ctrl =           ALU_ADD;       //ADD operation in ALU
     branch_taken =          1'b0;          //Never take a branch in I_TYPE so that mux_pc_in is PC+4
        case (opcode)
            OPCODE_I_TYPE: begin    //Type I instructions
                 case (funct_3)
                     //(addi)
                    ADDI: begin
                        cu_alu_ctrl =           ALU_ADD;       //ADD operation in ALU
                    end
                    //(slti)
                    SLTI: begin
                        cu_alu_ctrl =           ALU_SLT;       //SLT operation in ALU
                    end
                    //(sltiu)
                    SLTIU: begin
                        cu_alu_ctrl =           ALU_SLTU;      //SLTU operation in ALU
                    end
                    //(xori)
                    XORI: begin
                        cu_alu_ctrl =           ALU_XOR;       //XOR operation in ALU
                    end
                    //(slti)
                    ORI: begin
                        cu_alu_ctrl =           ALU_OR;        //OR operation in ALU
                    end
                    //(ANDI)
```

```
                    ANDI: begin
                        cu_alu_ctrl =           ALU_AND;        //AND operation in ALU
                    end
                    default: begin
                        cu_alu_ctrl =           ALU_ADD;        //ADD operation in ALU
                    end
                endcase
            end
            OPCODE_B_TYPE: begin    //Type B instructions
                prf_wr_en =             1'b0;           //disable prf write
                cu_imm_sel =            IMM_B;          //use IMM_B_TYPE
                prf_pc_mux_ctrl =       1'b1;           //use rs1 as opreand 1 in ALU
                    //Type B(BEQ)
                    case (funct_3)
                        BEQ: begin
                            if (zero == 1'b1) begin //This flag is going to be '1' if
                                rs1 = rs2
                                cu_alu_ctrl =           ALU_ADD;        //ADD
                                    operation in ALU
                                branch_taken = 1'b1;                    //Select the
                                    direction of the branch (branch taken)
                            end else
                                branch_taken = 1'b0;
                        end
                        //TODO: Add more types of branches in order to roun any
                            program
                        default: begin
                            cu_alu_ctrl =           ALU_ADD;        //ADD operation in
                                ALU
                            branch_taken =          1'b0;           //branch not taken
                        end
                    endcase
            end
            OPCODE_R_TYPE: begin
                 cu_imm_sel =           IMM_NF;          //use IMM_NO_FUNCTION_TYPE
                 prf_imm_mux_ctrl =     1'b0;           //use rs2 as operand 2 in ALU
                 if (funct_7 == 7'b0000000) begin       //The instructions bellow
                     have this code  funct_7
                     case (funct_3)     //The changes of this types of R operations
                         comes from funct_3
                         //ADDITION OPERATION
                         ADD: begin
                             cu_alu_ctrl =           ALU_ADD;        //ADD operation
                                 in ALU
                         end
                         //LOGICAL LEFT SHIFT
                         SLL: begin
                             cu_alu_ctrl =           ALU_SLL;        //SLL operation
                                 in ALU
                         end
                         //SIGNED COMPARISON
                         SLT: begin
                             cu_alu_ctrl =           ALU_SLT;        //SLT operation
                                 in ALU
                         end
                         //UNSIGNED COMPARISON
                         SLTU: begin
                             cu_alu_ctrl =           ALU_SLTU;        //SLTU operation
                                  in ALU
                         end
                         //XOR
                         XOR_: begin
                             cu_alu_ctrl =           ALU_XOR;        //XOR operation
                                 in ALU
                         end
```

```
106                         //RIGHT LOGICAL SHIFT
107                         SRL: begin
108                             cu_alu_ctrl =           ALU_SRL;        //SRL operation
                                    in ALU
109                         end
110                         //OR
111                         OR_: begin
112                             cu_alu_ctrl =           ALU_OR;         //OR operation in
                                    ALU
113                         end
114                         //AND
115                         AND_: begin
116                             cu_alu_ctrl =           ALU_AND;        //AND operation
                                    in ALU
117                         end
118                         default: begin
119                             cu_alu_ctrl =           ALU_ADD;        //ADD operation
                                    in ALU
120                         end
121                     endcase
122                 end else if (funct_7 == 7'b0100000) begin   //The instructions
                        bellow have this code  funct_7
123                     case(funct_3)
124                         //SUBTRACTION OPERATION
125                         SUB: begin
126                             cu_alu_ctrl =           ALU_SUB;        //SUB operation
                                    in ALU
127                         end
128                         //SIGN-EXTEND SHIFT OPERATION
129                         SRA: begin
130                             cu_alu_ctrl =           ALU_SRA;        //Arithmetic
                                    right shift operation in ALU
131                         end
132                         default: begin
133                             cu_alu_ctrl =           ALU_ADD;        //ADD operation
                                    in ALU
134                         end
135                     endcase
136                 end else begin
137                     cu_alu_ctrl =           ALU_ADD;                //ADD operation
                            in ALU
138                 end
139             end
140             OPCODE_J_TYPE: begin    //In this RISCV architecture it only exists jump
                    and link (JAL) operation
141                 cu_imm_sel =            IMM_J;                  //use
                        IMM_NO_FUNCTION_TYPE
142                 prf_pc_mux_ctrl =       1'b1;                   //use pc_out as
                        opreand 1 in ALU
143                 cu_mem_out_mux_sel =    INSTRUCTION_TO_PRF;     //send the result of
                        ALU to prf data_in
144                 cu_alu_ctrl =           ALU_ADD;                //ALU operation in
                        ALU
145                 branch_taken =          1'b1;                   //always take the
                        branch
146             end
147             //TODO: Add more types of J instructions
148             default: begin
149                 prf_wr_en =             1'b0;           //enable prf write
150                 cu_alu_ctrl =           ALU_ADD;        //ADD operation in ALU
151             end
152         endcase
153   end
154 endmodule
```

Listing 1: Unidad de Conrol

```
module mux_3_to_1 (
    input logic [31:0] data_out_to_pc,
    input logic [31:0] alu_to_mem_addr,
    input logic [31:0] data_out_to_mux,
    input logic [1:0] sel,
    output logic [31:0] data_out
);
    `include "defines.sv"
    always_comb begin
        case (sel)
            ALU_TO_PRF: begin
                data_out = alu_to_mem_addr;
            end
            DATA_OUT_TO_PRF: begin
                data_out = data_out_to_mux;
            end
            INSTRUCTION_TO_PRF: begin
                data_out = data_out_to_pc;
            end
            default: data_out = '0;
        endcase
        end
endmodule
```

Listing 2: Multiplexor ALU, Memoria, PC

```
module plus_4_or_2_mux(
    input logic sel,
    output logic [2:0] instruction_add
    );

always_comb begin
    case (sel)
        1'b0: begin
            instruction_add = 3'b100;
        end
        1'b1: begin
            instruction_add = 3'b010;
        end
        default: instruction_add = 3'b000;
    endcase
end
endmodule
```

Listing 3: Multiplexor auxiliar 4-2 para PC

```
module mux #( parameter WIDTH = 32)(
    input  logic [WIDTH-1:0] in1,
    input  logic [WIDTH-1:0] in2,      // array of N inputs
    input  logic             sel,      // select
    output logic [WIDTH-1:0] out       // mux output
);
    assign out = (sel) ? in1 : in2; //0 = in_2
endmodule
```

Listing 4: Multiplexor auxiliar

```
`timescale 1ns / 1ps
module adder # (parameter DATA_WIDTH = 32)(
```

```
    input logic [2:0] constant_operand,
    input logic [DATA_WIDTH - 1:0] instruction_addr,
    output logic [DATA_WIDTH - 1:0] next_instruction
    );
    assign next_instruction = constant_operand + instruction_addr;
endmodule
```

Listing 5: Sumador interno para PC

```
module branch #(parameter DATA_WIDTH = 32)(
    input logic [DATA_WIDTH - 1: 0]rs_1,
    input logic [DATA_WIDTH - 1: 0]rs_2,
    output logic branch_taken
    );

    assign branch_taken = (rs_1 == rs_2) ? 1'b1 : 1'b0;  //Zero flag

endmodule
```

Listing 6: Multiplexor auxiliar

## Apéndice B: RTL de banco de registro

```
1  `timescale 1ns / 1ps
2  module physical_register_file #(parameter DIR_WIDTH = 10, parameter DATA_WIDTH = 256)
       (
3      input logic clk,                    //Clock signal
4      input logic arst_n,                 //Reset signal
5      input logic write_en,               //Write enable signal
6      input logic [DIR_WIDTH - 1:0] read_dir1,        //Read direction 1
7      input logic [DIR_WIDTH - 1:0] read_dir2,        //Read direction 2
8      input logic [DIR_WIDTH - 1:0] write_dir,        //Write direction
9      input logic [DATA_WIDTH - 1:0] write_data,      //Data to write
10     output logic [DATA_WIDTH - 1:0] read_data1,     //Readed Data 1
11     output logic [DATA_WIDTH - 1:0] read_data2,      //Readed Data 2
12     output logic [DATA_WIDTH -1:0] fibb_out
13     );
14
15     logic [DATA_WIDTH - 1:0] prf [1:(2**DIR_WIDTH) - 1];    //Physical Register File
           32*32bits
16
17     always_ff(posedge clk,negedge arst_n)begin
18         if (arst_n == 1'b0)begin                        //Condition of reset
19             for (int i = 1 ; i < DATA_WIDTH; i++) begin         //For cicle to reset
                   to 0 all values of memory array
20                 prf [i]<= '0;
21             end
22         end else begin
23             if (write_en)begin    //If write enable is set and the direction of write
                    is not 0, then write memory array
24                 prf [write_dir] <= write_data;              //Memory array
                       assignation
25             end
26         end
27     end
28     always_comb begin
29         read_data1 = read_dir1 == '0 ? '0 : prf [read_dir1];      //Assignation of
               readed data 1 depending of read direction 1
30         read_data2 = read_dir2 == '0 ? '0 : prf [read_dir2];      //Assignation of
               readed data 2 depending of read direction 2
31     end
32     assign fibb_out = prf[5];
33
34 endmodule
```

Listing 7: Banco de registros

## Apéndice C: RTL de ALU

```
`timescale 1ns / 1ps
// ALU for basic arithmetic instructions in RV32I
module alu #(parameter N = 32)(
    input logic [N-1:0] operand1,     // First operand, sourced from a register
    input logic [N-1:0] operand2,     // Second operand sourced from a register
    //input logic [N-1:0] imm_in,          // Second operand sourced from immediate generator
    //input logic [N-1:0] pc_in,           // First operand sourced from program counter
    //input logic alusrc_r1,               // ALUsrc selector for the first operand
    //input logic alusrc_r2,               // ALUsrc selector for the second operand
    input logic [3:0]   alucontrol,      // Use 4 bits for the opcode at the moment
    //output logic zero,
    output logic [N-1:0] alu_result     // ALU result, destiny is another register
    );

`include "defines.sv"
    always_comb begin
        unique case(alucontrol)
            ALU_ADD: begin  // At the moment we don't care about the carry
                alu_result = operand1 + operand2;
            end
            ALU_SUB: begin
                alu_result = operand1 - operand2;
            end
            ALU_AND: begin
                alu_result = operand1 & operand2;
            end
            ALU_OR: begin
                alu_result = operand1 | operand2;
            end
            ALU_XOR: begin
                alu_result = operand1 ^ operand2;
            end
            ALU_EQ: begin
                alu_result = operand1 == operand2;
            end
            ALU_SLT: begin  // Compare sign bit
                if (operand1[N-1] != operand2[N-1]) begin
                    // Find which operand has the negative sign
                    alu_result = operand1[N-1] > operand2[N-1];
                end else begin
                    // IF MSB is equal, compare bits directly
                    alu_result = operand1[N-2:0] < operand2[N-2:0];
                end
            end
            ALU_SLTU: begin
                alu_result = operand1 < operand2;
            end
            ALU_SLL: begin
                alu_result = operand1 << operand2;
            end
            ALU_SRL: begin
                alu_result = operand1 >> operand2;
            end
            ALU_SRA: begin
                alu_result = $signed(operand1) >>> operand2; // Implementation could be hard-coded if needed
            end
            default: alu_result = '0;
        endcase
    end
```

```
60 endmodule
```

Listing 8: Módulo de ALU

## Apéndice D: RTL de Módulo Extensión de signo

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/21/2025 10:44:41 PM
// Design Name:
// Module Name: imm_gen
// Project Name:
//////////////////////////////////////////////////////////////////////////////////


module imm_gen(
    input logic [31:0] instr,       // Instructions from program memory
    input logic [2:0] imm_sel,      // 000 = I-type, 001 = S-type ,010 = B-type, 011 = U-type, 100 = J-type of control unit
    output logic [31:0] imm_out     // output for ALU and PC
    );

`include "defines.sv"             //define the imm instructions

    always_comb begin
        unique case (imm_sel)
            IMM_I: begin //000 = I-type
                imm_out = {{20{instr[31]}}, instr[31:20]}; //signed extension (repeat 20 times the MSB)
            end

        IMM_S: begin //001 = S-type
                imm_out = { {20{instr[31]}},instr[31:25], instr[11:7] };//signed extension (repeat 20 times the MSB)
            end

            IMM_B: begin  //010 = B-type
                 imm_out = { {20{instr[31]}}, {instr[7], instr[30:25], instr[11:8], 1'b0 } };
            end

            IMM_U: begin  // 011 = U-type
                imm_out = {  instr[31:12], {12{1'b0}} };
            end

            IMM_J: begin // 100 = J-type
                imm_out = { {12{instr[31]}}, {instr[19:12], instr[20], instr[30:21], 1'b0 } };
            end
            default: imm_out = IMM_NF;
        endcase
    end
endmodule
```

Listing 9: Generador de inmediatos

## Apéndice E: RTL de memoria de instrucciones

```
`timescale 1ns / 1ps
module instruction_memory #(
    parameter BYTE_WIDTH = 8,        // Ancho de caj n de la memoria (1 Byte)
    parameter MEM_DEPTH  = 1024,     // N mero de renglones de memoria
    parameter ADDR_WIDTH = 10,       // con 10 bits podemos direccionar 1024 renglones
    parameter DATA_WIDTH = 32        // Ancho de instrucci n (32 bits)
)
(
    input  logic                     clk,

    // Puerto de Lectura (PC)
    input  logic [ADDR_WIDTH-1:0]  rd_addr,   // Direcci n de lectura (PC)
    output logic [DATA_WIDTH-1:0]  rd_data,   // Instrucci n que sale

    // Puerto de escritura (FIFO)
    input  logic [DATA_WIDTH-1:0]  data_in,   // Dato (instrucci n) que viene de la
        FIFO
    input  logic [ADDR_WIDTH-1:0]  wr_addr,   // Direcci n (10 bits para 1024
        espacios)
    input  logic                     w_en        // Write Enable
);

    localparam BANK_DEPTH = MEM_DEPTH / 4; // 256 renglones por banco

    // bank3 = byte [31:24] (MSB), bank0 = byte [7:0] (LSB)
    logic [BYTE_WIDTH-1:0] bank0 [0:BANK_DEPTH-1];
    logic [BYTE_WIDTH-1:0] bank1 [0:BANK_DEPTH-1];
    logic [BYTE_WIDTH-1:0] bank2 [0:BANK_DEPTH-1];
    logic [BYTE_WIDTH-1:0] bank3 [0:BANK_DEPTH-1];

    // 1. Inicializaci n (Limpieza + programa)
    initial begin : init_memory
        // Limpia toda la memoria
        for (int i = 0; i < BANK_DEPTH; i++) begin
            bank0[i] = {BYTE_WIDTH{1'b0}};
            bank1[i] = {BYTE_WIDTH{1'b0}};
            bank2[i] = {BYTE_WIDTH{1'b0}};
            bank3[i] = {BYTE_WIDTH{1'b0}};
        end
    end

    // 2. Escritura paralela en los 4 bancos
    // [ADDR_WIDTH-1:2] para elegir el rengl n dentro del banco (PC alineado a 4
        bytes)
    always_ff (posedge clk) begin
        if (w_en) begin
            bank3[wr_addr[ADDR_WIDTH-1:2]] <= data_in[31:24]; // MSB
            bank2[wr_addr[ADDR_WIDTH-1:2]] <= data_in[23:16];
            bank1[wr_addr[ADDR_WIDTH-1:2]] <= data_in[15:8];
            bank0[wr_addr[ADDR_WIDTH-1:2]] <= data_in[7:0];   // LSB
        end
    end

    // 3. Lectura (Combinacional)
    assign rd_data = {
        bank3[rd_addr[ADDR_WIDTH-1:2]],
        bank2[rd_addr[ADDR_WIDTH-1:2]],
        bank1[rd_addr[ADDR_WIDTH-1:2]],
        bank0[rd_addr[ADDR_WIDTH-1:2]]
    };
endmodule
```

Listing 10: Memoria de instrucciones

## Apéndice F: RTL de memoria del programa

```
`timescale 1ns / 1ps
module data_memory #(
    parameter DATA_WIDTH = 32,      // Ancho de instruccion / dato
    parameter ADDR_WIDTH = 32,      // Ancho de direcci n del PC
    parameter MEM_DEPTH  = 1024        // Profundidad (4 renglones: 0,1,2,3)
    )(
    input logic clk,
    input logic w_en,             // Write Enable desde CU
    input logic [ADDR_WIDTH-1:0] wr_addr,       // Address
    input logic [DATA_WIDTH-1:0] data_in,      // Write data de memoria de datos
    output logic [DATA_WIDTH-1:0] rd_data      // Read Data (Dato de salida) de
        memoria de datos
);

    // Memoria de 32 bits y 1024 renglones
    logic [DATA_WIDTH-1:0] ram [0:MEM_DEPTH-1];

    // Lectura debe de ser combinacional
    assign rd_data = ram[wr_addr[ADDR_WIDTH-1:2]];

    // 1. Inicializaci n (Limpieza)
    initial begin
        for (int i=0; i<MEM_DEPTH; i++)
            ram[i] = {DATA_WIDTH{1'b0}};
    end

    // Escritura debe de ser secuencial
    always_ff (posedge clk) begin
        if (w_en) begin
            ram[wr_addr[ADDR_WIDTH-1:2]] <= data_in;         //si WE es 1, se escribe
                en la ram el valor
        end
    end
endmodule
```

Listing 11: Memoria de datos

## Apéndice G: RTL de UART

```
module uart_top (
    input logic clk,
    input logic arst_n,
    input logic rx,
   input logic next_program,
    input logic [BAUD_SEL_SIZE - 1 : 0] baud_sel,
    input logic [ADDR_WIDTH - 1 : 0] rdaddr,
    output logic prog_rdy,
    output logic rx_done,
    output logic [DATA_WIDTH - 1 : 0] data_out,
    output logic [((DATA_WIDTH / 4) * 7) - 1 : 0] display,
    output logic [DATA_WIDTH - 1 : 0] read_data,
    output logic tx,
   output logic ready,
   output logic [3:0] state,
   output logic busy,
    output logic tx_done,
    input logic tx_start
);

logic tick;
logic inst_rdy;
logic [BYTE_WIDTH - 1 : 0] uart_byte;
logic [ADDR_WIDTH - 1 : 0] wr_addr;
logic [BYTE_WIDTH - 1 : 0] n_instructions;

shift_register_fsm #(.BYTE_WIDTH(BYTE_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
    shift_register_fsm_i(
    .clk (clk),
    .arst_n (arst_n),
   .next_program(next_program),
    .w_en (rx_done), // input write enable coming from uart rx_done
    .data_in (uart_byte), // data_in coming from uart data_out port
    .data_out (data_out), // data out to be written in the program memory after
        receiving 4 bytes from uart
    .wr_addr (wr_addr), // write addres for the memory
    .inst_rdy (inst_rdy), // flag to indicate that a instruction is ready, this is
        the write enable for the memory
   .n_instructions(n_instructions),
   .busy(busy),
   .state(state),
   .ready(ready),
    .prog_rdy (prog_rdy)  // flag to indicate that the program has been initialized
        into the memory
);


baud_rate_generator #(.CLK_FREQ(CLK_FREQ)) baud_rate_generator_i(
    .clk(clk),
    .arst_n(arst_n),
    .tick(tick),
    .baud_sel(baud_sel)
);

transmitter #(.BYTE_WIDTH(BYTE_WIDTH)) transmitter_i(
   .clk(clk),
   .arst_n(arst_n),
   .data_in(8'd10),
   .tick(tick),
   .tx_start(rx_done),
   .tx(tx),
   .tx_done(tx_done)
```

```
);

display_7_segments #(.DATA_WIDTH(DATA_WIDTH)) display_7_segments_i(
    .data_in (read_data),
    .display (display)
);

receiver #(.BYTE_WIDTH(BYTE_WIDTH)) receiver_i(
.clk(clk),
.arst_n(arst_n),
.tick(tick),
.rx(rx), // RX CONNECTED TO THE TX FROM THE TRANSMITTER MODULE
.rx_done(rx_done),
.data_out(uart_byte)
);

instruction_memory #(.BYTE_WIDTH(BYTE_WIDTH), .ADDR_WIDTH(ADDR_WIDTH), .MEM_DEPTH(
    MEM_DEPTH), .DATA_WIDTH(DATA_WIDTH)) instruction_memory_i(
.clk(clk),
.rd_addr(rdaddr),
.rd_data(read_data),
.data_in(data_out),
.wr_addr(wr_addr),
.w_en(inst_rdy)
);


endmodule
```

Listing 12: Módulo uart_top

```
module baud_rate_generator #(parameter CLK_FREQ =50_000_000)(
    input  logic       clk,
    input  logic       arst_n,
    input  logic [3:0] baud_sel,
    output logic       tick
);

/*
localparam int BAUD_TABLE [0:12] = {
    1200,
    2400,
    4800,
    9600,
    19200,
    28800,
    38400,
    57600,
    76800,
    115200,
    230400,
    460800,
    921600
};

*/


logic [31:0] counter_max;
logic [31:0] counter;

/*
always_comb begin
    if (baud_sel <= 4'd12)
        counter_max = (CLK_FREQ / (BAUD_TABLE[baud_sel] * 16)) - 1;
```

```
    else
        counter_max = (CLK_FREQ / (BAUD_TABLE[0] * 16)) - 1; // default
end
*/

assign counter_max = (CLK_FREQ / (9600 * 16));

always_ff (posedge clk or negedge arst_n) begin
    if(!arst_n) begin
        tick    <= 1'b0;
        counter <= 32'd0;
    end else begin
        if (counter == counter_max) begin
            counter <= 32'd0;
            tick    <= 1'b1;
        end else begin
            counter <= counter + 1;
            tick    <= 1'b0;
        end
    end
end

endmodule
```

Listing 13: Módulo baud_rate_generator

```
module receiver #(parameter BYTE_WIDTH = 8)(
input logic clk,
input logic arst_n,
input logic tick, // TICK COMING FROM THE BAUD RATE GENERATOR
input logic rx, // INPUT SERIAL COMING FROM THE TRANSMITTER TX
output logic rx_done, // DONE SIGNAL TO INDICATE THAT THE BYTE HAS BEEN RECEIVED
output logic [BYTE_WIDTH - 1 : 0] data_out // OUPUT SIGNAL FOR THE BYTE RECEIVED
);

localparam BIT_SAMPLING = 15;
localparam HALFBIT_SAMPLING = 7;

logic [4:0] nbits;
logic [4:0] nbits_next;
logic [4:0] oversampling_count;
logic [4:0] oversampling_count_next;
logic [BYTE_WIDTH - 1 : 0] data_out_reg;
logic [BYTE_WIDTH - 1 : 0] data_out_reg_next;
logic rx_done_next;

typedef enum logic [2:0] {IDLE = 3'b000, START = 3'b001, DATA = 3'b010, STOP = 3'b011
    } state_type;

state_type state_reg, state_next;

always_ff (posedge clk or negedge arst_n) begin
    if(!arst_n) begin
        state_reg <= IDLE;
    end else begin
        oversampling_count <= oversampling_count_next;
        state_reg <= state_next;
        nbits <= nbits_next;
        data_out_reg <= data_out_reg_next;
        rx_done <= rx_done_next;
    end
end

always_comb begin
    oversampling_count_next = oversampling_count;
```

```
    nbits_next              = nbits;
    data_out_reg_next       = data_out_reg;
    state_next              = state_reg;
    rx_done_next = 1'b0;

    case(state_reg)
        IDLE: begin
            rx_done_next = 1'b0;
            oversampling_count_next = '0;
            data_out_reg_next       = data_out_reg;
            nbits_next              = '0;
            if (!rx) begin
                data_out_reg_next = '0;
                state_next = START;
      end
        end

        START: begin
            if (tick) begin
                if (oversampling_count == BIT_SAMPLING) begin
                    oversampling_count_next = '0;
                    nbits_next = '0;
                    data_out_reg_next = '0;
                    state_next = DATA;
                end else begin
                    oversampling_count_next = oversampling_count + 1'b1;
                    if (oversampling_count == HALFBIT_SAMPLING) begin
                        state_next = (!rx) ? START : IDLE;
          end
                end
            end
        end

        DATA: begin
            if (tick) begin
                if (oversampling_count == BIT_SAMPLING) begin
                    oversampling_count_next = '0;
                    if (nbits == (BYTE_WIDTH - 1'b1)) begin
                        state_next = STOP;
                    end else begin
                        nbits_next = nbits + 1'b1;
          end
                end else begin
                    oversampling_count_next = oversampling_count + 1'b1;
                    if (oversampling_count == HALFBIT_SAMPLING) begin
                        data_out_reg_next = {rx, data_out_reg[BYTE_WIDTH-1:1]};
          end
                end
            end
        end

        STOP: begin
            if (tick) begin
                if (rx) begin
                    if (oversampling_count == BIT_SAMPLING) begin
                        state_next = IDLE;
                        oversampling_count_next = '0;
                    end else begin
                        oversampling_count_next = oversampling_count + 1'b1;
            if (oversampling_count == HALFBIT_SAMPLING) begin
              rx_done_next = 1'b1;
            end
                    end
                end else begin
                    state_next = IDLE;
```

```
                    oversampling_count_next = '0;
                end
            end
        end

    endcase
end

assign data_out = data_out_reg;

endmodule
```

Listing 14: Módulo receiver

```
module shift_register_fsm#(parameter DATA_WIDTH = 32, parameter BYTE_WIDTH = 8,
    parameter ADDR_WIDTH = 10)(
   input logic clk,
   input logic arst_n,
   input logic next_program,
   input logic w_en, // input write enable coming from uart rx_done
   input logic [BYTE_WIDTH - 1 : 0] data_in, // data_in coming from uart data_out
       port
   output logic [DATA_WIDTH - 1 : 0] data_out, // data out to be written in the
       program memory after receiving 4 bytes from uart
   output logic [ADDR_WIDTH - 1 : 0] wr_addr, // write addres for the memory
   output logic inst_rdy, // flag to indicate that a instruction is ready, this is
       the write enable for the memory
   output logic [BYTE_WIDTH - 1 : 0] n_instructions,
   output logic busy,
   output logic [3:0] state,
   output logic ready,
   output logic prog_rdy  // flag to indicate that the program has been initialized
       into the memory
);

// internal counters
logic [BYTE_WIDTH - 1 : 0] n_of_instructions;
logic [BYTE_WIDTH - 1 : 0] n_of_instructions_next;

logic [ADDR_WIDTH - 1 : 0] instruction_counter;
logic [ADDR_WIDTH - 1 : 0] instruction_counter_next;

logic [DATA_WIDTH - 1 : 0] temp_data_out;
logic [DATA_WIDTH - 1 : 0] temp_data_out_next;

logic [BYTE_WIDTH - 1 : 0] received_byte;


// states for the fsm
typedef enum logic [3:0]{
    WAIT_PARAMS = 4'b0000,
    WAIT_BYTE0 = 4'b0001,
    WAIT_BYTE1 = 4'b0010,
    WAIT_BYTE2 = 4'b0011,
    WAIT_BYTE3 = 4'b0100,
    WRITE_INSTRUCTION = 4'b0101,
    DONE = 4'b0110,
   CLEAN_MEMORY = 4'b0111
} state_type;

state_type state_reg, state_next;

// reset all the signals and assign the next state sequential logic
always_ff(posedge clk or negedge arst_n) begin
    if(!arst_n) begin
```

```
47          state_reg <= WAIT_PARAMS;
48          instruction_counter <= '0;
49          n_of_instructions <= '0;
50          temp_data_out <= '0;
51      end else begin
52          state_reg <= state_next;
53          instruction_counter <= instruction_counter_next;
54          n_of_instructions <= n_of_instructions_next;
55          temp_data_out <= temp_data_out_next;
56      end
57  end
58
59  always_comb begin
60      //prog_rdy = 1'b0; // flag for program initialized set to 0
61      n_of_instructions_next = n_of_instructions; // default value
62      state_next = state_reg; // default state for state_next
63      instruction_counter_next = instruction_counter; // default value
64      temp_data_out_next = temp_data_out; // default value
65
66      case(state_reg)
67
68          WAIT_PARAMS: begin // receive a byte from the uart to define the number of
                instructions to write into the memory
69              if(w_en) begin
70                  n_of_instructions_next = data_in;
71             instruction_counter_next = '0;
72             temp_data_out_next = '0;
73                  state_next = WAIT_BYTE0;
74              end else begin
75            state_next = state_reg;
76                  n_of_instructions_next = n_of_instructions;
77              end
78          end
79
80          WAIT_BYTE0: begin // receive the 1st byte of the instruction
81              if(w_en) begin
82             temp_data_out_next = '0;
83                  temp_data_out_next[7:0] = data_in;
84                  state_next = WAIT_BYTE1;
85              end else begin
86                  state_next = state_reg;
87              end
88          end
89
90          WAIT_BYTE1: begin // receive the 2nd byte of the instruction
91              if(w_en) begin
92                  temp_data_out_next[15:8] = data_in;
93                  state_next = WAIT_BYTE2;
94              end else begin
95                  state_next = state_reg;
96              end
97          end
98
99          WAIT_BYTE2: begin // receive the 3rd byte of the instruction
100             if(w_en) begin
101                 temp_data_out_next[23:16] = data_in;
102                 state_next = WAIT_BYTE3;
103             end else begin
104                 state_next = state_reg;
105             end
106         end
107
108         WAIT_BYTE3: begin // receive the 4th and last byte of the instruction
109             if(w_en) begin
110                 temp_data_out_next[31:24] = data_in;
```

```
                state_next = WRITE_INSTRUCTION;
            end else begin
                state_next = state_reg;
            end
        end

        WRITE_INSTRUCTION: begin // state to write instruction once the shifter has
            received 4 bytes = 32 bits instruction
            if(instruction_counter == ((n_of_instructions - 1) * 4)) begin
                state_next = DONE;
            end else begin
                instruction_counter_next = instruction_counter + 3'd4;
                state_next = WAIT_BYTE0;
            end
        end

        DONE: begin // done state to set program ready flag to 1
        instruction_counter_next = '0;
        temp_data_out_next = {DATA_WIDTH{1'b0}};
        if(next_program) begin
          state_next = CLEAN_MEMORY;
        end else begin
          state_next = DONE;
        end
        end

      CLEAN_MEMORY: begin
        temp_data_out_next = {DATA_WIDTH{1'b0}};
        if(instruction_counter == ((n_of_instructions - 1 ) * 4)) begin
          state_next = WAIT_PARAMS;
          n_of_instructions_next = '0;
        end else begin
          instruction_counter_next = instruction_counter + 3'd4;
          state_next = state_reg;
        end
      end

    endcase
end

assign prog_rdy = (state_reg == DONE);
assign inst_rdy = (state_reg == WRITE_INSTRUCTION) || (state_reg == CLEAN_MEMORY);
assign data_out = temp_data_out;
assign wr_addr = instruction_counter;
assign state = state_reg;
assign n_instructions = n_of_instructions;
assign ready = (state_reg == WAIT_PARAMS);
assign busy = ~ready;

endmodule
```

Listing 15: Módulo shift_register_fsm

```
import serial
import time

imm = int(input("Enter the # iteration for the fibonacci series: "))

def fibonacci(n):
    serie = [0, 1]
    for _ in range(2, n):
        serie.append(serie[-1] + serie[-2])

    hex_result = format(serie[n-1], 'x')
    print(f"El resultado de la serie Fibonacci para el numero ingresado es: Hex: {
```

```
            hex_result}, int: {serie[-1]}")
        serie = [0, 1]
        return None

fibonacci(imm)

port = 'COM4'
baudrate = 9600
timeout = 1

# Rango del inmediato que quieres enviar
IMM_START = 0x0000
IMM_END   = 0x00FF    # cambialo al rango que necesites

try:
    ser = serial.Serial(port, baudrate, timeout=timeout)
    time.sleep(0.05)

    # Separar el inmediato en dos bytes (little-endian)
    imm_high = (imm & 0x0F) << 4        # LSB
    imm_low  = (imm >> 4) & 0xFF # MSB

    # Construye payload dinamico
    payload = [
        bytes([0x13, 0x05, 0x00, 0x00]),
        bytes([0x93, 0x05, 0x10, 0x00]),
        bytes([0x13, 0x06, imm_high, imm_low]),  # <- SE ACTUALIZA AQUI
        bytes([0x63, 0x0C, 0x06, 0x00]),
        bytes([0xB3, 0x02, 0xB5, 0x00]),
        bytes([0x13, 0x85, 0x05, 0x00]),
        bytes([0x93, 0x85, 0x02, 0x00]),
        bytes([0x13, 0x06, 0xF6, 0xFF]),
        bytes([0x6F, 0xF0, 0xDF, 0xFE]),
        bytes([0x6F, 0x00, 0x00, 0x00])
    ]

    print(f"\n=== Enviando inmediato {hex(imm)} -> HIGH={hex(imm_high)}, LOW={hex(
        imm_low)} ===")

    # Enviar numero de instrucciones (10 -> 0x0A)
    n_instr = b'\x0A'
    ser.write(n_instr)
    time.sleep(0.005)

    # Enviar instrucciones una por una
    for frame in payload:
        ser.write(frame)
        print(f"Sent: {frame}")
        time.sleep(0.005)

    # Pausa opcional entre iteraciones
    time.sleep(0.05)

    ser.close()
    print("Serial port closed.")

except Exception as e:
    print("ERROR:", e)
```

Listing 16: Script en python para UART

## Apéndice H: Testbench del microprocesador

```
1
2  module microprocessor_tb ();
3
4  // `include "defines.svh"
5
6      bit clk;
7      bit arst_n;
8
9      // para el testbench (randomizados)
10     logic [DIR_WIDTH-1:0] rd, rs1, rs2;
11     logic [12:0] imm_bq;
12     logic [20:0] imm_jal;
13
14     logic [DATA_WIDTH-1:0] instruction_tb, instruction_tb_addi, instruction_tb_add,
           instruction_tb_beq, instruction_tb_jal;
15     logic [DIR_WIDTH-1:0] rd_tb, rs1_tb, rs2_tb;
16     logic [11:0] imm_tb;
17
18     // para el BIND
19     logic [DATA_WIDTH-1:0] alu_result;
20     logic [DATA_WIDTH-1:0] pc_out, pc_imm; // Se ales para conectar al bind
21     logic memread, memwrite, memtoreg;
22
23     //CHECKS
24     logic [31:0] pc_before;
25     logic [31:0] imm_check;
26     logic [31:0] rd_expected;
27     logic [31:0] beq_taken;
28     logic [31:0] beq_notaken;
29     logic equal;
30     logic [6:0] count_add, count_addi, count_beq, count_jal, count_add_g,
           count_addi_g, count_beq_g, count_jal_g;
31
32     always #5ns clk = !clk;
33   assign #10ns arst_n = 1'b1;
34
35     microprocessor_if tbprocessor_if(clk, arst_n);
36     //microprocessor_top tbprocessor_top(clk, arst_n);
37
38     microprocessor_top microprocessor_i (
39         .clk(clk),
40         .arst_n(arst_n),
41         .prog_ready(1'b1),
42         .w_en(1'b0),
43         .instruction(tbprocessor_if.instruction)
44       );
45
46
47     `define  MEM_PATH microprocessor_i.instruction_memory_i
48   `define  ALU_PATH microprocessor_i.alu_i
49   `define  BANK_REG_PATH microprocessor_i.prf_i
50   `define  PC_PATH microprocessor_i.pc_i
51   `define  IMM_GEN_PATH microprocessor_i.imm_gen_i
52   `define  MUX_IMM_PATH microprocessor_i.mux_imm_gen_i
53   `define  MUX_PC_PATH microprocessor_i.mux_pc_i
54   `define  CU_PATH microprocessor_i.control_unit_i
55     `define  BRANCH_PATH microprocessor_i.branch_i
56
57     typedef enum bit [6:0] {
58       addi = 7'b0010011,
59       add  = 7'b0110011,
60       beq  = 7'b1100011,
```

```
61        jal  = 7'b1101111
62      } operation;
63
64      operation current_instruction;
65
66      initial begin
67      count_add = 0; count_addi= 0; count_beq = 0; count_jal = 0;
68      count_add_g = 0; count_addi_g = 0; count_beq_g = 0; count_jal_g = 0;
69
70      wait (arst_n);
71        $display("--- Decodificador de Instrucciones (RISC-V Opcode) ---");
72
73        repeat (100) (posedge clk) begin
74        randcase
75          1 :  begin // ADDI
76      instruction_tb_addi = '0; instruction_tb_add= '0; instruction_tb_beq= '0;
          instruction_tb_jal= '0;
77                std::randomize(rd,rs1);
78                tbprocessor_if.write_addi_instr(rd,rs1);
79                current_instruction = tbprocessor_if.instruction[6:0];
80                rd_tb = tbprocessor_if.instruction[11:7];
81                rs1_tb = tbprocessor_if.instruction[19:15];
82                imm_tb = tbprocessor_if.instruction[DATA_WIDTH-1:DATA_WIDTH-12];
83                instruction_tb_addi = tbprocessor_if.instruction;
84                count_addi = count_addi + 1;
85                (posedge clk); // Esperamos un ciclo
86                    //if (`ALU_PATH.alu_result !== (`ALU_PATH.operand1 + `ALU_PATH.
                        operand2)) begin
87                    //  $error("ERROR EN ADDI!! ALU: esperado %0d + %0d = %0d,
                        obtenido = %0d",
88                    //         `ALU_PATH.operand1, `ALU_PATH.operand2,
89                    //         `ALU_PATH.operand1 + `ALU_PATH.operand2, `ALU_PATH.
                        alu_result);
90                    //end
91                    //if (`BANK_REG_PATH.write_dir !== rd) begin
92                    //  $error("ERROR EN ADDI!! Direcci n esperada = %0d, direcci n
                        escrita = %0d",
93                    //         rd, `BANK_REG_PATH.write_dir);
94                    //end else begin
95                    //count_addi_g = count_addi_g + 1;
96                    //end
97              end
98
99          1 : begin // ADD
100     instruction_tb_addi = '0; instruction_tb_add= '0; instruction_tb_beq= '0;
          instruction_tb_jal= '0;
101               std::randomize(rs1,rs2,rd);
102               tbprocessor_if.write_add_instr(rs1,rs2,rd);
103               current_instruction = tbprocessor_if.instruction[6:0];
104               rd_tb = tbprocessor_if.instruction[11:7];
105               rs1_tb = tbprocessor_if.instruction[19:15];
106               rs2_tb = tbprocessor_if.instruction[24:20];
107               instruction_tb_add = tbprocessor_if.instruction;
108               count_add =count_add +1;
109               (posedge clk);
110               //if (`ALU_PATH.alu_result !== (`ALU_PATH.operand1 + `ALU_PATH.
                      operand2)) begin
111               //  $error("ERROR EN ADD!! ALU: esperado %0d + %0d = %0d, obtenido %0
                      d",
112               //         `ALU_PATH.operand1, `ALU_PATH.operand2,
113               //         `ALU_PATH.operand1 + `ALU_PATH.operand2, `ALU_PATH.
                      alu_result);
114               //end
115               //if (`BANK_REG_PATH.write_dir !== rd) begin
```

```
116                 //  $error("ERROR EN ADD!! Direcci n esperada = %0d, direcci n
                        escrita = %0d",
117                 //         rd, `BANK_REG_PATH.write_dir);
118                 //end else begin
119                 //count_add_g = count_add_g + 1;
120                 //end
121               end
122
123
124         1 : begin //BEQ
125     instruction_tb_addi = '0; instruction_tb_add= '0; instruction_tb_beq= '0;
            instruction_tb_jal= '0;
126                 std::randomize(rs1,rs2,imm_bq);
127                 tbprocessor_if.write_beq_instr(rs1,rs2,imm_bq);
128                 current_instruction = tbprocessor_if.instruction[6:0];
129                 rs1_tb = tbprocessor_if.instruction[19:15];
130                 rs2_tb = tbprocessor_if.instruction[24:20];
131                 instruction_tb_beq = tbprocessor_if.instruction;
132                 pc_before = $signed(`PC_PATH.pc_out);
133                 equal = (rs1 == rs2);
134                 imm_check = $signed({imm_bq[12], imm_bq[10:5], imm_bq[4:1], imm_bq
                        [11], 1'b0});
135                 beq_taken = $signed(pc_before) + imm_check;
136                 beq_notaken = $signed(pc_before) + 4;
137                 rd_expected = equal ? $signed(beq_taken) : $signed(beq_notaken);
138                 count_beq = count_beq + 1;
139                 (posedge clk);
140                 //if ($signed(`PC_PATH.pc_out) !== $signed(rd_expected)) begin
141                 //  $error("ERROR BEQ FALL !\n  PC actual      = %d \n   PC anterior
                            = %d \n %d, %d -> condici n: %0s\n   Inmediato B    = %0d (%b)
                        \n  Esperado        = %d  (%0s)",
142                 //         $signed(`PC_PATH.pc_out), $signed(pc_before),
143                 //         rs1, rs2, equal ? "IGUALES -> TOMADO" : "DIFERENTES -> NO
                        TOMADO",
144                 //         $signed(imm_check), $signed(imm_bq),
145                 //         $signed(rd_expected),
146                 //         equal ? "SALTO TOMADO" : "SALTO NO TOMADO");
147                 //end else begin
148                 //    count_beq_g = count_beq_g + 1;
149                 //end
150         end
151
152         5 : begin //JAL
153     instruction_tb_addi = '0; instruction_tb_add= '0; instruction_tb_beq= '0;
            instruction_tb_jal= '0;
154                 std::randomize(rd,imm_jal);
155                 tbprocessor_if.write_jal_instr (rd,imm_jal);
156                 current_instruction = tbprocessor_if.instruction[6:0];
157                 rd_tb = tbprocessor_if.instruction[11:7];
158                 instruction_tb_jal = tbprocessor_if.instruction;
159                 pc_before = $signed(`PC_PATH.pc_out);
160                 imm_check = $signed({{12{imm_jal[20]}}, imm_jal[20:0], 1'b0});
161                 rd_expected = $signed(pc_before) + $signed(imm_check);
162                 count_jal = count_jal + 1;
163                 (posedge clk);
164                 //if ($signed(`PC_PATH.pc_out) !== $signed(rd_expected)) begin
165                 //  $error("ERROR JAL: Salto incorrecto!\n   PC actual = %d\n
                        Esperado    = PC(%d) + imm(%0d) = %d",
166                 //         $signed(`PC_PATH.pc_out), $signed(pc_before), $signed(
                        imm_check), $signed(rd_expected));
167                 //end
168                 //if ($signed(`BANK_REG_PATH.write_dir) !== $signed(rd_expected))
                        begin
169                 //  $error("ERROR JAL: write_dir = %0d, esperado rd=%0d", $signed(`
                        BANK_REG_PATH.write_dir), $signed(rd_expected));
```

```
                //end
                //if ($signed(`BANK_REG_PATH.write_data) !== $signed(pc_before) + 4)
                    begin
                //  $error("ERROR JAL: No guard  PC+4 en rd!\n   Escrito = %d\n
                    Esperado = PC+4 = %d",
                //          $signed(`BANK_REG_PATH.write_data), $signed(pc_before) +
                    4);
                //end else begin
                //    count_jal_g = count_jal_g + 1;
                //end
        end
      endcase
    end
    $display("TOTAL DE OPERACIONES = %d \n ADD  = %d    Sin errores = %d \n ADDI = %d
            Sin errores = %d \n BEQ  = %d    Sin errores = %d \n JAL  = %d    Sin
        errores = %d ",count_add+count_addi+count_beq+count_jal,count_add,count_add_g
        , count_addi,count_addi_g, count_beq,count_beq_g, count_jal, count_jal_g);
end


    initial begin // Initial block to open shared memory and probe signals
      $shm_open("shm_db");
      $shm_probe("ASMTR");
  end

  initial begin // Timeout thread
    #1ms;
    $finish;
  end




    logic probe_signal_beq;
    logic [31:0] probe1, probe2;
    assign probe1 = microprocessor_i.prf_i.prf[rs1];
    assign probe2 = microprocessor_i.prf_i.prf[rs2];
    assign probe_signal_beq = microprocessor_i.prf_i.prf[rs1] == microprocessor_i.
        prf_i.prf[rs2];


    //ADDI
    // Comprueba que si rd = 0, se escribe 0. Si rd != 0, se escribe rs1_val_expected
         + Inmediato.
    `AST(uC, instruction_tb_addi,
        current_instruction == OPCODE_I_TYPE[6:0] |=>,
        $past(rd_tb) == 5'd0 ?
            // Caso: rd es X0 (debe escribir 0)
            $signed(`BANK_REG_PATH.prf[$past(rd_tb)]) == 32'd0 :
            // Caso: rd es un registro v lido (debe escribir la suma)
            $signed(`BANK_REG_PATH.prf[$past(rd_tb)]) == (
                // rs1_val: Si rs1 == 0, se usa 0; si no, se usa el valor le do
                ($past(rs1_tb) == 5'd0 ? 32'd0 : $past($signed(`BANK_REG_PATH.
                    read_data1)))
                + $past($signed(`IMM_GEN_PATH.imm_out))
            )
    )

    // ADD
    // Comprueba que si rd = 0, se escribe 0. Si rd != 0, se escribe rs1_val_expected
         + rs2_val_expected.
    `AST(uC, instruction_tb_add,
       current_instruction  == OPCODE_R_TYPE[6:0] |=>,
        $past(rd_tb) == 5'd0 ?
            // Caso: rd es X0 (debe escribir 0)
```

```
            $signed(`BANK_REG_PATH.prf[$past(rd_tb)]) == 32'd0 :
            // Caso: rd es un registro v lido (debe escribir la suma de los
                operandos corregidos)
            $signed(`BANK_REG_PATH.prf[$past(rd_tb)]) == (
                // rs1_val: Si rs1 == 0, se usa 0; si no, se usa read_data1
                ($past(rs1_tb) == 5'd0 ? 32'd0 : $past($signed(`BANK_REG_PATH.
                    read_data1)))
                +
                // rs2_val: Si rs2 == 0, se usa 0; si no, se usa read_data2
                ($past(rs2_tb) == 5'd0 ? 32'd0 : $past($signed(`BANK_REG_PATH.
                    read_data2)))
            )
    )

//BEQ
  `AST(uC, instruction_tb_beq,
    current_instruction == OPCODE_B_TYPE[6:0] |=>,
    $past(`BRANCH_PATH.rs_1) == $past(`BRANCH_PATH.rs_2) ?
     $signed(microprocessor_i.pc_i.pc_in) == $past(microprocessor_i.pc_i.pc_out) +
         $past(`IMM_GEN_PATH.imm_out) : // branch taken
     $signed(microprocessor_i.pc_i.pc_in) == $past(microprocessor_i.pc_i.pc_out) +
         32'd4 // branch not taken
)
      // JAL
    // Salto del PC
    `AST(uC, instruction_tb_jal,
      current_instruction[6:0]  == OPCODE_J_TYPE[6:0] |=>,
        microprocessor_i.pc_i.pc_in == $past(microprocessor_i.pc_i.pc_out +
            microprocessor_i.imm_gen_i.imm_out)
    )
```

Listing 17: Testbench para Microprocesador