Entrega Procesador

Estudiantes: | Mateo Giraldo Arboleda

Docente: Jose Alfredo Jaramillo Villegas
Universidad Tecnológica de Pereira

30 de abril de 2024

Introducción

El siguiente informe detalla el diseño y la implementación en SystemVerilog, un lenguaje de descripción de hardware, de un procesador monociclo fundamentado en la arquitectura RISC-V de 32 bits. Este procesador representa un paso significativo en la comprensión y aplicación práctica de los principios subyacentes de la arquitectura RISC-V, destacando su potencial para aplicaciones diversas en el ámbito de los sistemas embebidos y la computación de bajo consumo. A lo largo del documento, se presentará en detalle la descripción, el código de diseño, el testbench y el EPWave de cada módulo, proporcionando una visión completa del proceso de desarrollo y verificación del procesador.

1. Program Counter

El módulo Program Counter (PC) constituye un componente fundamental en el diseño del procesador, encargado de mantener la dirección de la instrucción actual. Este módulo cuenta con una entrada de 32 bits, denominada pcinput, la cual recibe la dirección de la siguiente instrucción a ejecutar, así como una señal de reloj (clock) para sincronizar su funcionamiento. Como salida, proporciona pcoutput, que representa la dirección de la instrucción actualizada. La implementación del Program Counter se realiza de manera sincronizada con el flanco de subida del reloj, lo que garantiza que en cada ciclo de reloj, la salida tome el valor actualizado del pcinput. Esta funcionalidad se logra mediante la coordinación eficiente de varios módulos, asegurando un seguimiento preciso del flujo de ejecución del programa.

1.1. Diseño

```
module ProgramCounter (
    input wire clk,
    input logic [31:0] pc_input,
    output logic [31:0] pc_out
```

```
);

    logic first_time = 1'b1;

    always_ff @(posedge clk) begin
        if (first_time) begin
            pc_out <= 32'h00000000;
            first_time <= 1'b0;
        end else begin
            pc_out <= pc_input;
        end
    end

endmodule
```

1.2. Testbench

```
`timescale 1ns/1ns

module ProgramCounter_tb;

    // Par metros del m dulo
    localparam CLK_PERIOD = 10;  // Per odo del reloj en unidades de
   tiempo (en este caso, 10ns)
    localparam SIM_TIME = 200;   // Tiempo total de simulaci n en
   unidades de tiempo

    // Se ales del test bench
    reg clk = 0;                // Se al de reloj
    reg [31:0] pc_input = 0;    // Se al de entrada pc_input
    wire [31:0] pc_out;         // Se al de salida pc_out

    // Instanciaci n del m dulo bajo prueba
    ProgramCounter dut (
        .clk(clk),
        .pc_input(pc_input),
        .pc_out(pc_out)
    );

    // Generaci n del reloj
    always #((CLK_PERIOD)/2) clk = ~clk;

    // Generaci n del archivo VCD
    initial begin
        $dumpfile("ProgramCounter_tb.vcd");
        $dumpvars(0, ProgramCounter_tb);
    end

    // Cambio de pc_input para observar el cambio en pc_out
    initial begin
        // Inicializaci n
        // Cambios en pc_input
```

```
34          #0   pc_input = 32'h00000000;
35          #10  pc_input = 32'h00000001;
36          #10  pc_input = 32'h00000002;
37          #10  pc_input = 32'h00000003;
38          #10  pc_input = 32'h00000004;
39          #10  pc_input = 32'h00000005;
40          #10  pc_input = 32'h00000006;
41          #10  pc_input = 32'h00000007;
42          #10  pc_input = 32'h00000008;
43          #10  pc_input = 32'h00000009;
44          #10  pc_input = 32'h0000000A;
45          #10  pc_input = 32'h0000000B;
46          #10  pc_input = 32'h0000000C;
47          #10  pc_input = 32'h0000000D;
48          #10  pc_input = 32'h0000000E;
49          #10  pc_input = 32'h0000000F;
50          $finish;
51      end
52
53      // Finalizaci n de la simulaci n
54      always @(posedge clk) begin
55          if ($time == SIM_TIME) begin
56              $display("Fin de la simulaci n.");
57              $finish;
58          end
59      end
60
61  endmodule
```

1.3. EPWave

2. Instruction Memory

El módulo Instruction Memory (IM) es un componente crucial en el procesador, diseñado para almacenar y proporcionar las instrucciones necesarias para la ejecución del programa. Este módulo cuenta con una única entrada, la dirección de memoria (Address), y una única salida, la instrucción a ejecutar (Instruction), ambas representadas por valores de 32 bits.

La operación de la Memoria de Instrucciones se basa en la lectura de un archivo de memoria (.mem) que contiene todas las instrucciones del programa, representadas en formato hexadecimal. Cada instrucción está separada de la siguiente por saltos de línea, y dentro de cada instrucción, cada byte está representado por dos nibbles (cuatro bits cada uno), los cuales están separados por un espacio en blanco.

La lectura del archivo de memoria se realiza utilizando la dirección proporcionada como índice dentro del arreglo de instrucciones, permitiendo así obtener la instrucción correspondiente para su procesamiento posterior en el flujo de ejecución del programa.

2.1. Diseño

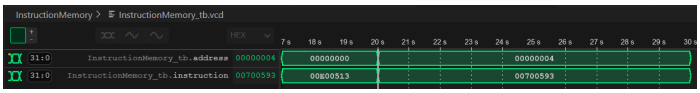
```
module InstructionMemory(
    input logic [31:0] address,
    output logic [31:0] instruction
    );
    logic [7:0] memory [0:256];
    initial begin
        $readmemh("test.mem", memory);
    end
    assign instruction = {memory[address], memory[address + 1],
   memory[address + 2], memory[address + 3]};

endmodule
```

2.2. Testbench

```
module InstructionMemory_tb;

    // Par metros
    parameter ADDRESS_WIDTH = 32;
    parameter DATA_WIDTH = 32;

    // Entradas
    logic [ADDRESS_WIDTH-1:0] address;

    // Salidas
    logic [DATA_WIDTH-1:0] instruction;

    // Clock
    logic clk;

    // Instancia del m dulo bajo prueba
    InstructionMemory dut (
        .address(address),
        .instruction(instruction)
    );

    // Generador de clock
    always #5 clk = ~clk;

    // Testbench
    initial begin
        // Inicializaci n de la simulaci n
        $dumpfile("InstructionMemory_tb.vcd");
        $dumpvars(0, InstructionMemory_tb);

```

```
        // Inicializaci n de la memoria de instrucciones
        $readmemh("test.mem", dut.memory);

        // Reset inicial
        address = 0;
        #10;

        // Test 1: Leer instrucci n en la direcci n 0
        address = 0;
        #10;
        if (instruction !== 32'hXXXXXXXX) $display("Test 1: Failed");
        else $display("Test 1: Passed");

        // Test 2: Leer instrucci n en la direcci n 4
        address = 4;
        #10;
        if (instruction !== 32'hXXXXXXXX) $display("Test 2: Failed");
        else $display("Test 2: Passed");

        // Finalizar simulaci n
        $finish;
    end

endmodule
```

2.3. EPWave



3. Control Unit

La Control Unit (Unidad de Control) es un componente esencial del procesador, encargado de coordinar y controlar las operaciones dentro de cada uno de los módulos en función de las señales de entrada proporcionadas. Este módulo dispone de tres entradas principales: OpCode (código de operación) de 7 bits, Funct3 (campo de función 3) de 3 bits y Funct7 (campo de función 7) de 7 bits.

La función primordial de la Control Unit radica en generar señales de control que dirigen el funcionamiento de los diversos módulos del procesador. Estas señales de control incluyen RUWr (registro de escritura), ImmSrc (fuente de inmediato), ALUASrc (fuente de operando A de la ALU), ALUBSrc (fuente de operando B de la ALU), BrOp (operación de salto condicional), ALUOp (operación de la ALU), DMWr (escritura en memoria de datos), DMCtrl (control de memoria de datos) y RUDataWrSrc (fuente de escritura de datos del registro de usuario).

La selección de estas señales de control se realiza mediante un análisis exhaustivo y la asignación de valores, basados en el estudio de las señales de entrada proporcionadas.

De esta manera, la Control Unit garantiza una correcta sincronización y ejecución de las operaciones dentro del procesador, optimizando su rendimiento y eficiencia en la ejecución de instrucciones

3.1. Diseño

```
module ControlUnit (
    input logic [6:0] opcode,
    input logic [2:0] Funct3,
    input logic [6:0] Funct7,
    output logic RUWr,
    output logic [3:0] ALUOp,
    output logic [2:0] ImmSrc,
    output logic ALUASrc,
    output logic ALUBSrc,
    output logic DMWr,
    output logic [2:0] DMCtrl,
    output logic [4:0] BrOp,
    output logic [1:0] RUDataWrSrc
);

always @* begin
    if (opcode == 7'b0110011 || opcode == 7'b0010011 || opcode == 7'b0000011) begin
        RUWr = 1'b1;
        if (opcode == 7'b0110011) begin
                ALUOp = {Funct7[5], Funct3};
            end
                else if (opcode == 7'b0010011) begin
                    if(Funct3 == 3'b001 || Funct3 == 3'b101) begin
                        ALUOp = {Funct7[5], Funct3};
                    end
                    else begin
                        ALUOp = {1'b0, Funct3};
                    end
                end
                    else begin
                        ALUOp = 4'b0000;
                    end
        ImmSrc = 3'b000;
        ALUASrc = 1'b0;
        ALUBSrc = ~opcode[5];
        DMWr = 1'b0;
        DMCtrl = Funct3;
        BrOp = 5'b00000;
        RUDataWrSrc = {1'b0, ~opcode[4]};
    end
    else if (opcode == 7'b1100011 || opcode == 7'b1101111 || opcode == 7'b0100011) begin
        RUWr = opcode[3];
        ALUOp = 4'b0000;
        ALUASrc = opcode[6];
```

```
        ALUBSrc = 1'b1;
        DMWr = ~opcode[6];
        DMCtrl = Funct3;
        if (opcode == 7'b1100011) begin
            ImmSrc = 3'b101;
            BrOp = {2'b01, Funct3};
        end
            else if (opcode == 7'b1101111) begin
                ImmSrc = 3'b110;
                BrOp = 5'b10000;
            end
                else begin
                    ImmSrc = 3'b001;
                        BrOp = 5'b00000;
                end
        RUDataWrSrc = 2'b10;
    end
    else if (opcode == 7'b1100111 || opcode == 7'b0110111) begin
        RUWr = opcode[6];
        ALUOp = (opcode == 7'b1100111) ? 4'b0000 : 4'b0111;
        ImmSrc = (opcode == 7'b1100111) ? 3'b000 : 3'b010;
        ALUASrc = 1'b0;
        ALUBSrc = 1'b1;
        DMWr = 1'b0;
        DMCtrl = 3'b000;
        BrOp = (opcode == 7'b1100111) ? 5'b10000 : 5'b00000;
        RUDataWrSrc = (opcode == 7'b1100111) ? 2'b10 : 2'b00;
    end
    else begin
        RUWr = 1'b0;
        ALUOp = 1'b0;
        ImmSrc = 1'b0;
        ALUASrc = 1'b0;
        ALUBSrc = 1'b0;
        DMWr = 1'b0;
        DMCtrl = 1'b0;
        BrOp = 1'b0;
        RUDataWrSrc = 1'b0;
    end
end

endmodule
```

3.2. Testbench

```
`timescale 1ns / 1ps

module ControlUnit_tb;

    // Par metros
    parameter PERIOD = 10;

```

```
    // Se ales de entrada
    logic [6:0] opcode;
    logic [2:0] Funct3;
    logic [6:0] Funct7;

    // Se ales de salida
    logic RUWr;
    logic [3:0] ALUOp;
    logic [2:0] ImmSrc;
    logic ALUASrc;
    logic ALUBSrc;
    logic DMWr;
    logic [2:0] DMCtrl;
    logic [4:0] BrOp;
    logic [1:0] RUDataWrSrc;

    // Instancia de ControlUnit
    ControlUnit dut (
        .opcode(opcode),
        .Funct3(Funct3),
        .Funct7(Funct7),
        .RUWr(RUWr),
        .ALUOp(ALUOp),
        .ImmSrc(ImmSrc),
        .ALUASrc(ALUASrc),
        .ALUBSrc(ALUBSrc),
        .DMWr(DMWr),
        .DMCtrl(DMCtrl),
        .BrOp(BrOp),
        .RUDataWrSrc(RUDataWrSrc)
    );

    // Generador de est mulo
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, ControlUnit_tb);
        // Casos de prueba
        // Caso 1: opcode == 7'b0110011
        opcode = 7'b0100011;
        Funct3 = 3'b000; // Ejemplo de Funct3
        Funct7 = 7'b0000000; // Ejemplo de Funct7
        #PERIOD; // Espera un ciclo de reloj
        // Verificar las se ales de salida aqu

        // Caso 2: opcode == 7'b1100011
        opcode = 7'b0110111;
        Funct3 = 3'b001; // Ejemplo de Funct3
        Funct7 = 7'b0000001; // Ejemplo de Funct7
        #PERIOD; // Espera un ciclo de reloj
        // Verificar las se ales de salida aqu

        // Finalizar simulaci n
        $finish;
    end
```

```
62
63 endmodule
```

3.3. EPWave

4. Register Unit

La Register Unit (Unidad de Registro) constituye un elemento crucial en el procesador, encargado de gestionar y almacenar los datos en los registros del procesador. Este módulo cuenta con varias entradas y salidas fundamentales para su funcionamiento eficiente. Las entradas incluyen rs1 (registro fuente 1), rs2 (registro fuente 2), rd (registro destino), DataWr (datos a escribir en el registro), y RUWr (señal de escritura en el registro).

La Register Unit posee 32 registros de usuario (RU) de 32 bits cada uno, inicializados a cero, excepto el registro número 2, que es el puntero a la pila ubicada en el Data Memory, que es inicializado a un valor específico, para poder manejar la pila de forma correcta(en este caso, 1000).

Las salidas de la Register Unit son RURs1 y RURs2, que representan los valores almacenados en los registros especificados por las entradas rs1 y rs2, respectivamente.

El comportamiento de escritura en los registros se activa en el flanco de subida del reloj. Cuando la señal RUWr está activa y el registro de destino (rd) no es igual a cero, los datos de entrada (DataWr) se escriben en el registro especificado por rd.

Este diseño de la Register Unit garantiza una gestión eficiente y fiable de los datos en el procesador, contribuyendo así a un funcionamiento óptimo y preciso del mismo durante la ejecución de instrucciones.

4.1. Diseño

```
1 module RegisterUnit (
2   input wire clk,
3   input [4:0] rs1,
4   input [4:0] rs2,
5   input [4:0] rd,
```

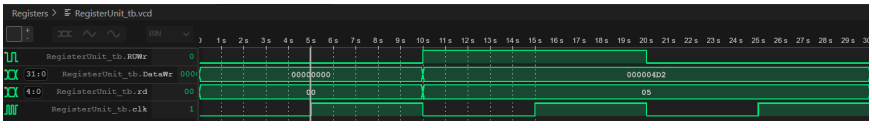
```
  input [31:0] DataWr,
  input RUWr,
  output reg [31:0] RURs1,
  output reg [31:0] RURs2
);

  logic [31:0] RU [0:31];

initial begin
    for (int i = 0; i < 32; i++) begin
        RU[i] = 32'h00000000;
    end
    RU[2] = 32'd1000;
end

  assign RURs1 = RU[rs1];
  assign RURs2 = RU[rs2];

  always @(posedge clk) begin
    if(RUWr && rd != 0)
      RU[rd] <= DataWr;
  end //always


endmodule
```

4.2. Testbench

```
module RegisterUnit_tb;

  // Par metros de simulaci n
  parameter CLK_PERIOD = 10; // Periodo del reloj en unidades de tiempo
   de simulaci n

  // Se ales de entrada
  reg clk = 0;
  reg [4:0] rs1, rs2, rd;
  reg [31:0] DataWr;
  reg RUWr;

  // Se ales de salida
  wire [31:0] RURs1, RURs2;

  // Instancia del m dulo RegisterUnit
  RegisterUnit dut (
    .clk(clk),
    .rs1(rs1),
    .rs2(rs2),
    .rd(rd),
    .DataWr(DataWr),
    .RUWr(RUWr),
    .RURs1(RURs1),
```

```
    .RURs2(RURs2)
  );

  // Inicializaci n de la simulaci n
  initial begin
    // Abrir el archivo VCD
    $dumpfile("RegisterUnit_tb.vcd");
    $dumpvars(0, RegisterUnit_tb);

    // Inicializar las se ales
    clk = 0;
    rs1 = 5'd0;
    rs2 = 5'd0;
    rd = 5'd0;
    DataWr = 32'd0;
    RUWr = 0;

    // Esperar un ciclo de reloj antes de comenzar la prueba
    #CLK_PERIOD;

    // Iniciar la secuencia de prueba
    // Ejemplo: escribir un valor en el registro rd = 5
    RUWr = 1;
    rd = 5;
    DataWr = 32'd1234;
    #CLK_PERIOD;
    RUWr = 0;
    #CLK_PERIOD;


    // Terminar la simulaci n
    $finish;
  end

  // Generador de reloj
  always #((CLK_PERIOD / 2)) clk = ~clk;

endmodule
```

4.3. EPWave



5. Branch Unit

La Branch Unit es un componente esencial del procesador, diseñado para gestionar las operaciones de salto condicional dentro del flujo de ejecución de instrucciones. Este módulo

cuenta con tres entradas principales: RURs1 y RURs2, ambos de 32 bits, que representan los valores almacenados en los registros especificados por las instrucciones, y una tercera entrada que proviene de la señal de control BrOp.

La salida de la Branch Unit es NextPCSrc, un bit que determina, mediante un multiplexor, si se debe realizar un salto durante la lectura de instrucciones. Este valor se utiliza para controlar el comportamiento de la Unidad de Control, permitiendo la ejecución de instrucciones de salto (como las instrucciones de tipo j, b y i-salto), lo que modifica el flujo de ejecución del programa.

En resumen, la Branch Unit desempeña un papel crucial en el procesador al facilitar la toma de decisiones sobre los saltos condicionales durante la ejecución de instrucciones, contribuyendo así a la correcta secuenciación y control del flujo de ejecución del programa

5.1. Diseño

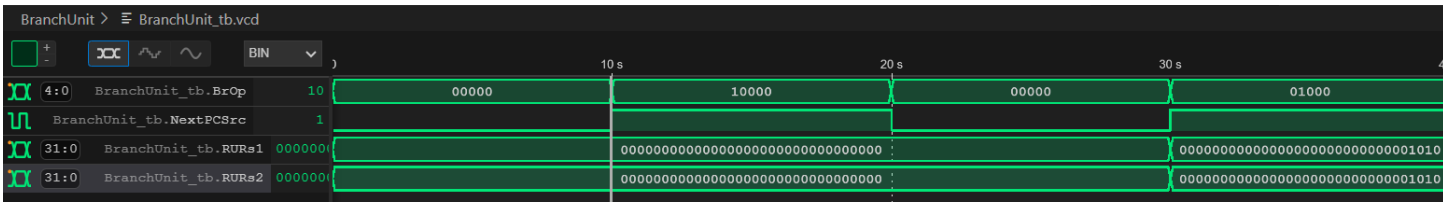
```
module BranchUnit (
    input logic [31:0] RURs1,
    input logic [31:0] RURs2,
    input logic [4:0] BrOp,
    output logic NextPCSrc
);

    always @* begin
        if (BrOp[4] == 1) NextPCSrc <= 1;
        else if (BrOp[4] == 0 && BrOp[3] == 0) NextPCSrc <= 0;
        else if (BrOp[4] == 0 && BrOp[3] == 1) begin
            if (BrOp[2:0] == 3'b000) NextPCSrc <= RURs1 == RURs2;
            else if (BrOp[2:0] == 3'b001) NextPCSrc <= RURs1 != RURs2;
            else if (BrOp[2:0] == 3'b100) NextPCSrc <= $signed(RURs1) <
   $signed(RURs2);
            else if (BrOp[2:0] == 3'b101) NextPCSrc <= $signed(RURs1) >=
   $signed(RURs2);
            else if (BrOp[2:0] == 3'b110) NextPCSrc <= RURs1 < RURs2;
            else if (BrOp[2:0] == 3'b111) NextPCSrc <= RURs1 >= RURs2;
        end

    end

endmodule
```

5.2. Testbench

```
module BranchUnit_tb;

  // Par metros de simulaci n
  parameter CLK_PERIOD = 10; // Periodo del reloj en unidades de tiempo
   de simulaci n

  // Se ales de entrada
  reg [31:0] RURs1, RURs2;
```

```
  reg [4:0] BrOp;

  // Se al de salida
  wire NextPCSrc;

  // Instancia del m dulo BranchUnit
  BranchUnit dut (
    .RURs1(RURs1),
    .RURs2(RURs2),
    .BrOp(BrOp),
    .NextPCSrc(NextPCSrc)
  );

  // Inicializaci n de la simulaci n
  initial begin
    // Abrir el archivo VCD
    $dumpfile("BranchUnit_tb.vcd");
    $dumpvars(0, BranchUnit_tb);

    // Inicializar las se ales
    RURs1 = 32'd0;
    RURs2 = 32'd0;
    BrOp = 5'b00000;

    // Esperar un ciclo de reloj antes de comenzar la prueba
    #CLK_PERIOD;

    // Iniciar la secuencia de prueba
    // Prueba 1: BrOp[4] = 1
    BrOp = 5'b10000;
    #CLK_PERIOD;

    // Prueba 2: BrOp[4:3] = 00
    BrOp = 5'b00000;
    #CLK_PERIOD;

    // Prueba 3: BrOp[4:3] = 01, BrOp[2:0] = 000 (RURs1 == RURs2)
    BrOp = 5'b01000;
    RURs1 = 32'd10;
    RURs2 = 32'd10;
    #CLK_PERIOD;


    // Terminar la simulaci n
    $finish;
  end

  // Generador de reloj
  always #((CLK_PERIOD / 2)) clk = ~clk;

endmodule
```

5.3. EPWave



6. ALU

La Arithmetic Logic Unit (ALU), o Unidad Aritmético Lógica, es un componente central en el procesador, diseñado para realizar operaciones aritméticas y lógicas según las necesidades del procesamiento de datos. Esta unidad cuenta con tres entradas principales: A y B, ambas de 32 bits, que representan los operandos de las operaciones, y ALUOp, un campo de 4 bits que indica el tipo de operación a realizar.

La ALU opera sobre los datos proporcionados en las entradas A y B, utilizando el ALUOp para determinar la operación específica a realizar. Las operaciones típicas incluyen sumas, restas, operaciones lógicas (AND, OR, XOR, etc.) y comparaciones.

La salida de la ALU, denotada como S, es un valor de 32 bits que representa el resultado de la operación realizada.

En resumen, la ALU desempeña un papel fundamental en el procesador al proporcionar capacidades de cálculo y lógica esenciales para la ejecución eficiente de las instrucciones, contribuyendo así al procesamiento adecuado de datos y al funcionamiento óptimo del sistema

6.1. Diseño

```
1
2  module ALU(A, B, AluOp, S);
3    input [31:0] A, B;
4    input [3:0] AluOp;
5    output reg [31:0] S;
6
7    always @* begin
8      case (AluOp)
9        4'b0000: S = A + B;
10       4'b1000: S = A - B;
11       4'b0001: S = A << B[4:0];
12       4'b0010: S = ($signed(A) < $signed(B)) ? 1 : 0;
13       4'b0011: S = (A < B) ? 1 : 0;
14       4'b0100: S = A ^ B;
15       4'b0101: S = A >> B[4:0];
16       4'b1101: S = $signed(A) >>> B[4:0];
17       4'b0110: S = A | B;
18       4'b0111: S = A & B;
19       default: S = 4'b0000;
```

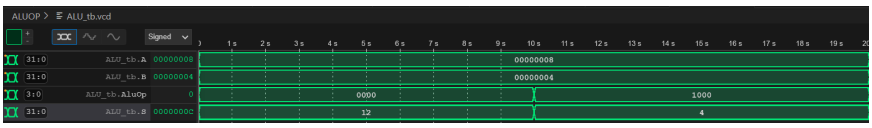
```
    endcase
  end
endmodule
```

6.2. Testbench

```
module ALU_tb;

  // Par metros
  parameter WIDTH = 32; // Ancho de los operandos

  // Se ales
  reg [WIDTH-1:0] A, B;
  reg [3:0] AluOp;
  wire [WIDTH-1:0] S;

  // Instancia del m dulo ALU
  ALU dut (
    .A(A),
    .B(B),
    .AluOp(AluOp),
    .S(S)
  );

  // Generaci n de est mulos
  initial begin
    // Abre el archivo VCD
    $dumpfile("ALU_tb.vcd");
    $dumpvars(0, ALU_tb);

    // Inicializa las se ales
    A = 0;
    B = 0;
    AluOp = 0;

    // Prueba de suma (AluOp = 4'b0000)
    A = 8;
    B = 4;
    AluOp = 4'b0000;
    #10;

    // Prueba de resta (AluOp = 4'b1000)
    A = 8;
    B = 4;
    AluOp = 4'b1000;
    #10;

    // Puedes a adir m s casos de prueba aqu  para las otras
   operaciones
    // Por ejemplo:
    // Prueba de desplazamiento a la izquierda (AluOp = 4'b0001)
    // Prueba de comparaci n (AluOp = 4'b0010)
```

```
46      // Prueba de XOR (AluOp = 4'b0100)
47      // Prueba de desplazamiento a la derecha (AluOp = 4'b0101)
48      // ... y as  sucesivamente
49
50      // Fin de la simulaci n
51      $finish;
52    end
53
54  endmodule
```

6.3. EPWave



7. Immediate Generator

El Generador de Inmediatos es un componente esencial en el procesador, diseñado para extender y organizar los valores inmediatos contenidos dentro de las instrucciones. Este módulo cuenta con dos entradas principales: Inst, un campo de 25 bits que representa la instrucción recibida, e ImmSrc, un campo de 3 bits que indica el tipo de instrucción y, por ende, qué parte de la instrucción contiene el inmediato.

La función primordial del Generador de Inmediatos es identificar y extender correctamente el inmediato de acuerdo con la instrucción recibida y el tipo de operación que se va a realizar. Para lograr esto, el módulo utiliza ImmSrc para determinar qué bits de la instrucción corresponden al inmediato y los organiza en un formato adecuado.

La salida del Generador de Inmediatos, denotada como ImmExt, es un valor de 32 bits que representa el inmediato extendido, listo para ser utilizado en las operaciones aritméticas, lógicas o de control correspondientes.

En resumen, el Generador de Inmediatos desempeña un papel crucial en el procesador al asegurar la correcta interpretación y extensión de los valores inmediatos dentro de las instrucciones, facilitando así la ejecución precisa y eficiente de las operaciones.

7.1. Diseño

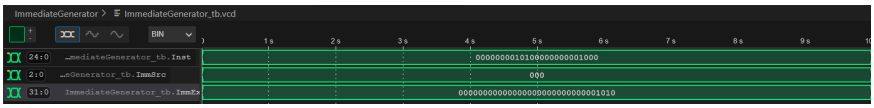
```
1
2   module ImmediateGenerator (
3       input logic [24:0] Inst,
4       input logic [2:0] ImmSrc,
5       output logic [31:0] ImmExt
6   );
7
8       always @* begin
```

```
        case (ImmSrc)
            3'b000: ImmExt = { {20{Inst[24]}}, Inst[24:13]};
            3'b001: ImmExt = { {20{Inst[24]}}, Inst[24:18], Inst[4:0]};
            3'b101: ImmExt = { {19{Inst[24]}}, Inst[24], Inst[0],
    Inst[23:18], Inst[4:1], 1'b0};
            3'b010: ImmExt = { {12{Inst[24]}}, Inst[24:5]};
            3'b110: ImmExt = { {11{Inst[24]}}, Inst[24], Inst[12:5],
    Inst[13], Inst[23:14], 1'b0};
            default: ImmExt = 32'h00000000;
        endcase
     end

endmodule
```

7.2. Testbench

```
module ImmediateGenerator_tb;

  // Par metros
  parameter WIDTH = 25; // Ancho de la instrucci n

  // Se ales
  reg [WIDTH-1:0] Inst;
  reg [2:0] ImmSrc;
  wire [31:0] ImmExt;

  // Instancia del m dulo ImmediateGenerator
  ImmediateGenerator dut (
    .Inst(Inst),
    .ImmSrc(ImmSrc),
    .ImmExt(ImmExt)
  );

  // Generaci n de est mulos
  initial begin
    // Abre el archivo VCD
    $dumpfile("ImmediateGenerator_tb.vcd");
    $dumpvars(0, ImmediateGenerator_tb);

    // Inicializa las se ales
    Inst = 0;
    ImmSrc = 0;

    // Prueba de ImmSrc = 3'b000
    Inst = 25'b0000000010100000000001000; // Ejemplo de instrucci n
    ImmSrc = 3'b000;
    #10;

    // Fin de la simulaci n
    $finish;
  end

endmodule
```

7.3. EPWave



8. Data Memory

El Data Memory (Memoria de Datos) es un componente fundamental en el procesador, diseñado para gestionar las operaciones de lectura y escritura en la memoria de datos. Este módulo cuenta con varias entradas cruciales para su funcionamiento eficiente: Address y DataWr, ambas de 32 bits, que representan la dirección de memoria y los datos a escribir, respectivamente. Además, cuenta con las señales de control DMWr y DMCtrl, de un bit y tres bits respectivamente.

La Memoria de Datos está organizada en 2048 filas, cada una de un byte de tamaño. Durante la inicialización del módulo, se produce una lectura de la memoria. Sin embargo, las operaciones de lectura o escritura se realizan solo cuando es necesario, según las señales de control proporcionadas.

La salida del Data Memory, denominada DataRd, es un valor de 32 bits que representa los datos leídos desde la dirección de memoria especificada.

En resumen, el Data Memory desempeña un papel crucial en el procesador al permitir el acceso eficiente a los datos almacenados en la memoria, facilitando así la ejecución precisa y eficiente de las operaciones de lectura y escritura en el sistema

8.1. Diseño

```
1
2  module DataMemory (
3    input [31:0] Address,
4    input [31:0] DataWr,
5    input DMWr,
6    input [2:0] DMCtrl,
7    output logic [31:0] DataRd
8  );
9
10   logic [7:0] Memory [0:2048];
11
12   always @(Address or DataWr or DMCtrl or DMWr) begin
13     case (DMCtrl)
14       3'b000: begin
15         DataRd = {{24{Memory[Address][7]}}, Memory[Address]}; // Byte
16       end
17       3'b001: begin
18         DataRd = {{16{Memory[Address + 1][7]}}, Memory[Address + 1],
    Memory[Address]}; // HalfWord (signed)
19       end
```

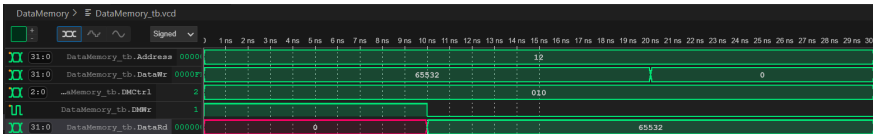
```
      3'b010: begin
        DataRd = {Memory[Address + 3], Memory[Address + 2],
    Memory[Address + 1], Memory[Address]}; // Word
      end
      3'b100: begin
        DataRd = {{24{1'b0}}, Memory[Address]}; // Byte (unsigned)
      end
      3'b101: begin
        DataRd = {{16{1'b0}}, Memory[Address + 1], Memory[Address]}; //
    HalfWord (unsigned)
      end
      default: begin
        DataRd = Memory[Address]; // Por defecto, leer una palabra
      end
    endcase


    if (DMWr) begin
      #1
      // Si DMWr es 1, escribir en la direcci n especificada por Address
      if (DMCtrl == 3'b000) begin
        Memory[Address] <= DataWr[7:0];
      end
      else if (DMCtrl == 3'b001) begin
        Memory[Address] <= DataWr[7:0];
        Memory[Address + 1] <= DataWr[15:8];
      end
      else if (DMCtrl == 3'b010) begin
        Memory[Address] <= DataWr[7:0];
        Memory[Address + 1] <= DataWr[15:8];
        Memory[Address + 2] <= DataWr[23:16];
        Memory[Address + 3] <= DataWr[31:24];
      end
      else if (DMCtrl == 3'b100) begin
        Memory[Address] <= DataWr[7:0];
      end
      else if (DMCtrl == 3'b101) begin
        Memory[Address] <= DataWr[7:0];
        Memory[Address + 1] <= DataWr[15:8];
      end
    end
  end
endmodule
```

8.2. Testbench

```
`timescale 1ns / 1ps

module DataMemory_tb;

  reg [31:0] Address;
  reg [31:0] DataWr;
```

```
  reg DMWr;
  reg [2:0] DMCtrl;
  wire [31:0] DataRd;

  DataMemory dut (
    .Address(Address),
    .DataWr(DataWr),
    .DMWr(DMWr),
    .DMCtrl(DMCtrl),
    .DataRd(DataRd)
  );

  initial begin
    $dumpfile("DataMemory_tb.vcd");
    $dumpvars(0, DataMemory_tb);


    // Test 2: Write to memory
    Address = 12;
    DataWr = 32'b1111111111111100;
    DMCtrl = 3'b010; // Word
    DMWr = 1;
    #10;
    DMWr = 0;
    #10;
    // Test 1: Read from memory
    Address = 12;
    DMCtrl = 3'b010; // Word
    DataWr = 32'h00000000;
    #10;

    // Add more tests as needed
    $finish;
  end

endmodule
```

8.3. EPWave



9. Procesador Monociclo

El módulo del Procesador Monociclo es el núcleo del sistema, responsable de ejecutar las instrucciones del programa. Está compuesto por varios módulos interconectados que

trabajan en conjunto para realizar las diferentes operaciones necesarias para la ejecución de las instrucciones.

El módulo del Procesador incluye:

Program Counter (Contador de Programa): Encargado de mantener la dirección de la próxima instrucción a ejecutar. Instruction Memory (Memoria de Instrucciones): Almacena las instrucciones del programa y proporciona la instrucción actual al procesador. Control Unit (Unidad de Control): Genera señales de control necesarias para coordinar las operaciones dentro del procesador. Register Unit (Unidad de Registros): Administra los registros del procesador y realiza operaciones de lectura y escritura en ellos. Immediate Generator (Generador de Inmediatos): Extiende y organiza los valores inmediatos contenidos en las instrucciones. Branch Unit (Unidad de Ramificación): Gestiona las operaciones de salto condicional dentro del flujo de ejecución del programa. ALU (Unidad Aritmético Lógica): Realiza operaciones aritméticas y lógicas según las necesidades del procesamiento de datos. Data Memory (Memoria de Datos): Almacena datos y realiza operaciones de lectura y escritura en la memoria de datos. Estos módulos se conectan entre sí mediante señales de control y datos para coordinar la ejecución del programa de manera eficiente y precisa. Cada módulo desempeña un papel crucial en el funcionamiento del procesador, contribuyendo al procesamiento adecuado de las instrucciones y al correcto flujo de ejecución del programa.

9.1. Diseño

```
1
2  `include "../InstructionMemory/InstructionMemory.sv"
3  `include "../ProgramCounter/ProgramCounter.sv"
4  `include "../ALUOP/ALU.sv"
5  `include "../BranchUnit/BranchUnit.sv"
6  `include "../ControlUnit/ControlUnit.sv"
7  `include "../DataMemory/DataMemory.sv"
8  `include "../ImmediateGenerator/ImmediateGenerator.sv"
9  `include "../Registers/RegisterUnit.sv"
10
11
12
13 module Processor(
14     input logic clk
15 );
16
17     //pc_plus-4
18     logic [31:0] pc_out_plus_4;
19
20     //Program Counter
21     logic [31:0] pc_out;
22     logic [31:0] instruction;
23     logic [31:0] pc_input;
24
25     //Control Unit
26     logic [6:0] opcode;
27     logic [2:0] Funct3;
28     logic [6:0] Funct7;
```

```
    logic ALUASrc;
    logic ALUBSrc;
    logic [3:0] ALUOp;
    logic DMWr;
    logic [2:0] DMCtrl;
    logic [1:0] RUDataWrSrc;

    //Registers Unit
    logic [4:0] rs1;
    logic [4:0] rs2;
    logic [4:0] rd;
    logic [31:0] DataWr;
    logic RUWr;
    logic [31:0] RURs1;
    logic [31:0] RURs2;

    //Imm Generator
    logic [24:0] Inst;
    logic [2:0] ImmSrc;
    logic [31:0] ImmExt;

    //Branch Unit
    logic [4:0] BrOp;
    logic NextPCSrc;

    //ALU
    logic [31:0] A;
    logic [31:0] B;
    logic [31:0] S;

    //Data Memory
    logic [31:0] Address;
    logic [31:0] DataRd;



    ProgramCounter PC(
        .clk(clk),
        .pc_input(pc_input),  // Inicializar pc_input en 0 solo una vez
        .pc_out(pc_out)
    );

    // Instanciar el m dulo InstructionMemory
    InstructionMemory IM(
        .address(pc_out),
        .instruction(instruction)
    );

    // Extraer los bits 6 al 0 de la instrucci n para el OpCode
    assign opcode = instruction[6:0];
    assign Funct3 = instruction[14:12];
    assign Funct7 = instruction[31:25];

    // Instanciar el m dulo ControlUnit
```

```
    ControlUnit CO(
        .opcode(opcode),
        .Funct3(Funct3),
        .Funct7(Funct7),
        .RUWr(RUWr),
        .ImmSrc(ImmSrc),
        .ALUASrc(ALUASrc),
        .ALUBSrc(ALUBSrc),
        .ALUOp(ALUOp),
        .BrOp(BrOp),
        .DMWr(DMWr),
        .DMCtrl(DMCtrl),
        .RUDataWrSrc(RUDataWrSrc)
        // Pasa otras entradas y salidas seg n sea necesario
    );

    assign rs1 = instruction[19:15];
    assign rs2 = instruction[24:20];
    assign rd = instruction[11:7];



    RegisterUnit RUnit(
        .rs1(rs1),
        .rs2(rs2),
        .rd(rd),
        .DataWr(RUDataWrSrc == 2'b10 ? pc_out_plus_4 : RUDataWrSrc ==
    2'b01 ? DataRd : RUDataWrSrc == 2'b00 ? S : 2'b00),
        .RUWr(RUWr),
        .clk(clk),
        .RURs1(RURs1),
        .RURs2(RURs2)
    );

    assign Inst = instruction[31:7];

    ImmediateGenerator IG(
        .Inst(Inst),
        .ImmSrc(ImmSrc),
        .ImmExt(ImmExt)
    );

    BranchUnit BU(
        .RURs1(RURs1),
        .RURs2(RURs2),
        .BrOp(BrOp),
        .NextPCSrc(NextPCSrc)
    );

    always @* begin
        pc_out_plus_4 = pc_out + 4;

        if (ALUASrc) begin
            A <= pc_out;
```

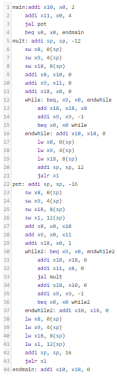
```
        end else begin
            A <= RURs1;
        end

        if (ALUBSrc) begin
            B <= ImmExt;
        end else begin
            B <= RURs2;
        end

        if (NextPCSrc) begin
            pc_input <= S;
        end else begin
            pc_input <= pc_out_plus_4;
        end

        /*case(RUDataWrSrc)
            2'b10: begin
                DataWr = pc_out_plus_4;
            end
            2'b01: begin
                DataWr = DataRd;
            end
            2'b00: begin
                DataWr = S;
            end
        endcase*/
    end


    ALU ALU(
        .A(A),
        .B(B),
        .AluOp(ALUOp),
        .S(S)
    );

    DataMemory DM(
        .Address(S),
        .DataWr(RURs2),
        .DMWr(DMWr),
        .DMCtrl(DMCtrl),
        .DataRd(DataRd)
    );

endmodule
```

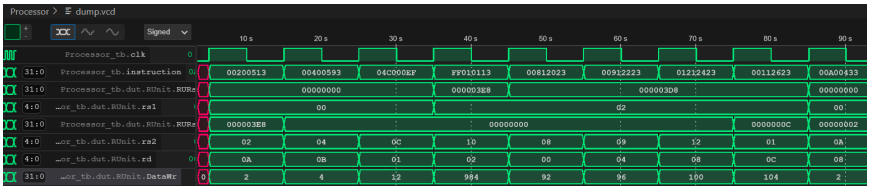
9.2. Testbench

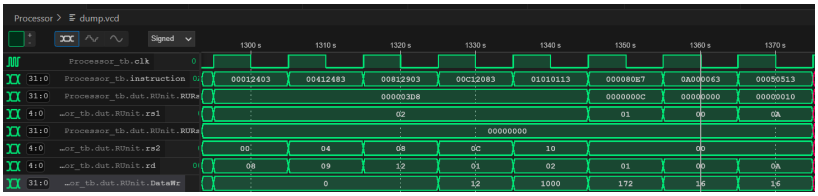
```
module Processor_tb;

  logic clk;
```

```
  logic [31:0] pc_input, pc_output, instruction, rs1, rs2;

  // Instantiate the DUT (Processor)
  Processor dut (
    .clk(clk)
    // Connect other ports here
  );

  // Assign signals from DUT to testbench signals
  assign pc_input = dut.pc_input;
  assign pc_output = dut.pc_out;
  assign instruction = dut.instruction;
  assign rs1 = dut.rs1;
  assign rs2 = dut.rs2;

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Display values of rs1 and rs2 on every positive clock edge
  always @(posedge clk) begin
    $display("rs1 %h rs2 %h", rs1, rs2);
  end

  // Dump VCD file and finish simulation after some time
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, Processor_tb);
    #4000;
    $finish;
  end

endmodule
```

9.3. EPWave




Processor > dump.vcd
Signed
10 s
20 s
30 s
40 s
50 s
60 s
70 s
80 s
90 s
Processor_tb.clk
Processor_tb.instruction
Processor_tb.dut.RUnit.RURs
...or_tb.dut.RUnit.rs1
Processor_tb.dut.RUnit.RURs
...or_tb.dut.RUnit.rs2
...or_tb.dut.RUnit.rd
...or_tb.dut.RUnit.DataWr
00200513 00400593 04C000EF FF010113 00812023 00912223 01212423 00112623 00A00433
00000000 000003E8 000003D8 00000000
00 02 00
000003E8 00000000 0000000C 00000002
02 04 0C 10 08 09 12 01 0A
0A 0B 01 02 00 04 08 0C 08
2 4 12 984 92 96 100 104 2


Processor > dump.vcd
Signed
1300 s
1310 s
1320 s
1330 s
1340 s
1350 s
1360 s
1370 s
Processor_tb.clk
Processor_tb.instruction
Processor_tb.dut.RUnit.RURs
...or_tb.dut.RUnit.rs1
Processor_tb.dut.RUnit.RURs
...or_tb.dut.RUnit.rs2
...or_tb.dut.RUnit.rd
...or_tb.dut.RUnit.DataWr
00012403 00412483 00812903 00C12083 01010113 000080E7 0A000063 00050513
000003D8 0000000C 00000000 00000010
02 01 00 0A
00000000
00 04 08 0C 10 00
08 09 12 01 02 01 00 0A
0 12 1000 172 16 16