Diseño e Implementación de un Procesador RISC-V Escalar y Segmentado en FPGA

Introducción

El presente trabajo de investigación consiste en al creación de un procesador escalar y segmentado con una microarquitectura RISC-V, dicha creación será propuesta y representada en el documento desde el punto de vista teórico. El principal objetico del presente trabajo de investigación consiste en la evaluación del funcionamiento y comportamiento del procesador creado en tareas representativas en el campo de la automoción y la inteligencia artificial.

Objetivos

- Diseñar la microarquitectura de un procesador RISC-V
- Simular el funcionamiento del procesador con diferentes programas.
- Sintetizar e implementar en el procesador una FPGA para emular RISC-V
- Cargar y validar el funcionamiento del procesador ante diversos eventos.
- Evaluar el rendimiento del procesador en programas representativos de varios sectores.

Diseño de la microarquitectura del procesador RISC-V

Para la creación y desarrollo del diseño del procesador RISC-V se ha utilizado Verilog, compilando un testbench de la CPU mediante iVerilog (Icarus Verilog).

La estructura de los ficheros del proyecto que define la arquitectura del procesador y que se encuentran dentro de la carpeta ubicada en ./RISCV_Architecture/ es la siguiente:

```
>/.RISCV_Architecture
    >/src
        >alu.v
        >control_unit.v
        >cpu.v
        >data_memory.v
        >immediate_generator.v
        >instruction_memory.v
        >reg_file.v
    >/test
        >cpu_testbench.v
        >data.mem
        >program.mem
    >run.sh
```

Fichero alu.v

Los ficheros con formato .v que se encuentran dentro de la carpeta /scr (source) son los encargados del funcionamiento y arquitectura propia del procesador RISC-V. En alu.v (la ALU del procesador) se han definido las entradas por el cana A y B, además del control que determina la operación a realizar y el resultado a devolver, adicionalmente, se ha determinado el funcionamiento de la ALU para las instrucciones: add, sub, and, or, xor, sll y srl (en caso de no ser ninguna de las anteriores la ALU devuelve 0).

alu.v:

```
module alu (
    input [31:0] a,             // Operando A
    input [31:0] b,             // Operando B
    input [3:0] alu_ctrl,       // Control para operacion
    output reg [31:0] result    // Resultado
);

    always @(*) begin
        case (alu_ctrl)
            4'b0000: result = a + b;        // add
            4'b0001: result = a - b;        // sub
            4'b0010: result = a & b;        // and
            4'b0011: result = a | b;        // or
            4'b0100: result = a ^ b;        // xor
            4'b0101: result = a << b[4:0];  // sll
            4'b0110: result = a >> b[4:0];  // srl
            default: result = 0;
        endcase
    end
endmodule
```

Fichero control_unit.v

En la control_unit.v (Unidad de control) se ha definido el código de operación, la función de 3 bits y la de 7 bits (por los diferentes formatos de instrucción), la selección del operando B de la entrada a la ALU y las señales de habilitación de lectura y escritura en memoria, la escritura en registro y la selección del dato a desplazar en la etapa WB (Write Back) además del control de operación de la ALU.

Cómo instrucciones se ha optado por la creación de las que vienen dada por el repertorio RISC-V32I (instrucciones básicas), entre las que se encuentran: addi, lw, sw, beq y jal (el resto de las operaciónes son mandadas a la ALU para su resolución).

control_unit.v:

```
module control_unit (
    input [6:0] opcode,         // Código de operación
    input [2:0] funct3,         // función de 3 bits
    input [6:0] funct7,         // función de 7 bits
    output reg alu_src,         // Selección de entrada B de la ALU
```

```
    output reg mem_read,        // Señal de lectura de memoria
    output reg mem_write,       // Señal de escritura de memoria
    output reg reg_write,       // Señal de escritura en registros
    output reg mem_to_reg,      // Selección escribir en registro (WB)
    output reg [3:0] alu_ctrl   // Control de la operación de la ALU
);
    always @(*) begin
        alu_src = 0;
        mem_read = 0;
        mem_write = 0;
        reg_write = 0;
        mem_to_reg = 0;
        alu_ctrl = 4'b0000;

        case (opcode)
            7'b0010011: begin       // addi
                alu_src = 1;
                reg_write = 1;
                alu_ctrl = 4'b0000; // add
            end
            7'b0000011: begin       // lw
                alu_src = 1;
                mem_read = 1;
                reg_write = 1;
                mem_to_reg = 1;
                alu_ctrl = 4'b0000; // add
            end
            7'b0100011: begin       // sw
                alu_src = 1;
                mem_write = 1;
                alu_ctrl = 4'b0000; // add
            end
            7'b1100011: begin       // beq
                alu_src = 0;
                alu_ctrl = 4'b0001; // sub
            end
            7'b1101111: begin       // jal
                alu_src = 0;
                reg_write = 1;
                mem_to_reg = 0;
                alu_ctrl = 4'b0000; // (irrelevante, salto)
            end
            default: ;
        endcase
    end
endmodule
```

Fichero cpu.v

La cpu.v (Central Unit Process) especifica las etapas que se desarrollan al realizar cada instrucción además del clk (señal de reloj) y del reset (señal de reinicio). El diseño de la arquitectura del procesador creado es la de un procesador segmentado de 5 etapas (IF, ID, EX, MEM, WB) que busca 1 instrucción por ciclo.

cpu.v -> Definición del módulo:

```
module cpu (
    input clk,      // Señal de reloj
    input reset     // Señal de reinicio
);
```

La etapa de IF (Instruction Fetch) "lee" la instrucción basandose en el Contador del programa PC (registro de 32 bits inicializado a 0), así como la dirección de la instrucción, tras esto, se realiza el registro del pipeline entre la etapa de IF y la ID (Instrucion Decode) para pasar a dicha etapa.

cpu.v -> IF STAGE:

```
    // ============================================================
    // IF (Instruction Fetch) stage
    // ============================================================
    reg [31:0] pc;          // Contador de programa
    wire [31:0] instr;      // instrucción obtenida de memoria

    // Instancia de la memoria de instrucciones
    instruction_memory imem (
        .addr(pc),              // Dirección de la instrucción = PC
        .instruction(instr)     // Instrucción leída
    );

    // IF/ID registro de pipeline (entre IF e ID)
    reg [31:0] ifid_pc, ifid_instr;
```

En la etapa de ID (Instruction Decode) se definen los capos para obtener el tipo de instrucción de la que se trata mediante el código de operación, los registros a utilizar y la función de 3 y 7 bits respectivamente además de definir la salida del banco de registros por las dos vías. Adicionalmente, la etapa hace una llamada al fichero reg_file.v cómo rf en donde se le pasan los parámetros necesarios para el correcto funcionamiento del banco de registros (véase en la explicación del fichero reg_file.v).

cpu.v -> ID STAGE (Primera Parte):

```
    // ============================================================
    // ID (Instruction Decode) stage
    // ============================================================
    // Campos de la instrucción (formato R/I/S/B/J)
    wire [6:0] opcode = ifid_instr[6:0];
    wire [4:0] rs1 = ifid_instr[19:15];
    wire [4:0] rs2 = ifid_instr[24:20];
    wire [4:0] rd  = ifid_instr[11:7];
    wire [2:0] funct3 = ifid_instr[14:12];
    wire [6:0] funct7 = ifid_instr[31:25];
```

```
    // Salida del banco de registros
    wire [31:0] reg_data1, reg_data2;

    // Banco de registros: lectura de rs1 y rs2, escritura en rd
    reg_file rf (
        .clk(clk),
        .rs1(rs1),                      // Registro fuente 1
        .rs2(rs2),                      // Registro fuente 2
        .rd(memwb_rd),                  // Registro destino
        .rd_data(memwb_result),         // Dato a escribir en rd (dedse WB)
        .reg_write(memwb_reg_write),    // Habilita la escritura
        .data1(reg_data1),              // Salida de rs1
        .data2(reg_data2)               // Salida de rs2
    );
```

En esta etapa se ha de verificar la lógica del salto dado que el salto dado por una instrución branch, no es el mismo que el dado por la instrucción jal, además, hay que verificar si el salto es o no tomado por el procesador, para ello, se ha implementado un sistema de burbujas para tener un sistema de esperas activas y que el procesador no se quede bloqueado ni se salte instrucciones.

cpu.v -> ID STAGE (Segunda Parte):

```
    // Lógica de salto
    wire is_branch = (opcode == 7'b1100011); // beq
    wire is_jump   = (opcode == 7'b1101111); // jal
    wire branch_taken = (is_branch && reg_data1 == reg_data2);
    wire insert_bubble = (branch_taken || is_jump);

    // Registro de IF/ID con burbuja en saltos
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 0;
            ifid_pc <= 0;
            ifid_instr <= 32'b0;
        end else begin
            // Actualiza el PC según el tipo de instrucción
            if (branch_taken) begin
                pc <= pc + branch_offset;
                ifid_pc <= 0;
                ifid_instr <= 32'b0;    // NOP (burbuja por salto)
            end else if (is_jump) begin
                pc <= pc + jump_offset;
                ifid_pc <= 0;
                ifid_instr <= 32'b0;    // NOP (burbuja por salto)
            end else begin
                pc <= pc + 4;
                ifid_pc <= pc;
                ifid_instr <= instr;
            end
        end
    end
```

En cuanto a las instrucciones que usan inmediatos (p.ej.: addi), se llama al fichero immediate_generator.v cómo imm_gen al que se le pasa el id de la isntrucción y el inmediato en cuestión (para más información véase la explicación del fichero immediate_generator.v) y se crean los offsets para los saltos y ramas con el mismo inmediato (ya dado por imm_gen).

cpu.v -> ID STAGE (Tercera Parte):

```
    // Inmediato decodificado
    wire [31:0] imm;
    immediate_generator imm_gen (
        .instr(ifid_instr),
        .imm(imm)
    );

    // Offsets para salto y ramas (simple: ambos usan el mismo inmediato)
    wire [31:0] branch_offset = imm;
    wire [31:0] jump_offset = imm;
```

Cómo última parte de esta etapa, se llama a control_unit.v cómo ctrl para pasarle la instrucción decodificada y que ésta se encargue de que se ejecute correctamente, tras esto, se realizan los registros del pipeline entre las etapas ID y EX (EXecute) para continuar con la misma.

cpu.v -> ID STAGE (Última Parte):

```
    // Unidad de control: genera señales de control a partir de la instrucción
    wire alu_src, mem_read, mem_write, reg_write, mem_to_reg;
    wire [3:0] alu_ctrl;

    control_unit ctrl (
        .opcode(opcode),            // Código de operación
        .funct3(funct3),            // Función 3bits
        .funct7(funct7),            // Función 7bits
        .alu_src(alu_src),          // Selecctión de entrada B de la ALU
        .mem_read(mem_read),        // Señal de lectura de memoria
        .mem_write(mem_write),      // Señal de escritura de memoria
        .reg_write(reg_write),      // Señal de escritura en registros
        .mem_to_reg(mem_to_reg),    // Selección para WB
        .alu_ctrl(alu_ctrl)         // Operación de la ALU
    );

    // ID/EX registros de pipeline (entre ID y EX)
    reg [31:0] idex_pc, idex_reg_data1, idex_reg_data2, idex_imm;
    reg [3:0]  idex_alu_ctrl;
    reg        idex_alu_src, idex_mem_read, idex_mem_write, idex_reg_write,
idex_mem_to_reg;
    reg [4:0]  idex_rd;

    always @(posedge clk) begin
```

```
        idex_pc         <= ifid_pc;
        idex_reg_data1  <= reg_data1;
        idex_reg_data2  <= reg_data2;
        idex_imm        <= imm;
        idex_alu_ctrl   <= alu_ctrl;
        idex_alu_src    <= alu_src;
        idex_mem_read   <= mem_read;
        idex_mem_write  <= mem_write;
        idex_reg_write  <= reg_write;
        idex_mem_to_reg <= mem_to_reg;
        idex_rd         <= rd;
    end
```

En la etapa de EX (EXecute) se selecciona el segundo opreando de la ALU dependiendo si es un registro o un inmediato y se define el resultado de 32 bits de dicha operación, tras esto, se llama al fichero alu.v (ALU del procesador) para pasarle por parámetro los registros (o inmediato) a utilizar para los cálculos de la unidad, por último se realiza el registro de pipeline entre las etapas EX y MEM (MEMory) para pasar a la misma.

cpu.v -> EX STAGE:

```
    // ============================================================
    // EX (Execute) stage
    // ============================================================
    // Selección del segundo operando de la ALU (reg o inm)
    wire [31:0] alu_in2 = idex_alu_src ? idex_imm : idex_reg_data2;

    // Resultado de la ALU
    wire [31:0] alu_result;

    // Instancia de la ALU
    alu alu_inst (
        .a(idex_reg_data1),
        .b(alu_in2),
        .alu_ctrl(idex_alu_ctrl),
        .result(alu_result)
    );

    // EX/MEM registro de pipeline (entre EX y MEM)
    reg [31:0] exmem_result, exmem_reg_data2;
    reg        exmem_mem_read, exmem_mem_write, exmem_reg_write, exmem_mem_to_reg;
    reg [4:0]  exmem_rd;

    always @(posedge clk) begin
        exmem_result     <= alu_result;
        exmem_reg_data2  <= idex_reg_data2;
        exmem_mem_read   <= idex_mem_read;
        exmem_mem_write  <= idex_mem_write;
        exmem_reg_write  <= idex_reg_write;
        exmem_mem_to_reg <= idex_mem_to_reg;
        exmem_rd         <= idex_rd;
    end
```

La etapa de MEM (MEMory) consiste en definir el dato que se obtiene de memoria a través del fichero data_memory, al que se le pasa por parámetro el reloj, la dirección de memoria calculada por la ALU (en la etapa EX), el dato a escribir en memoria, la habilitación de la señal de lectura y escritura en memorai y la salida de dicho dato de memoria (para más información véase la explicación del fichero data_memory.v). Una vez realizado lo anterior, se realiza el registro del pipeline entre las etapas EX y WB (Write Back) para terminar con esta. En este caso, al hacerse en el post-pipeline el guardado de datos en memoria, se realiza a su vez la etapa de WB.

cpu.v -> EX y WB STAGE:

```
    // ============================================================
    // MEM (Memory Access) stage y WB (Write Back) stage
    // ============================================================
    wire [31:0] mem_data_out;

    // Acceso a la memoria de datos
    data_memory dmem (
        .clk(clk),
        .addr(exmem_result),            // Dirección calculada por la ALU
        .write_data(exmem_reg_data2),   // Dato a escribir en caso de SW
        .mem_read(exmem_mem_read),      // Señal de lectura
        .mem_write(exmem_mem_write),    // Señal de escritura
        .read_data(mem_data_out)        // Salida de memoria
    );

    // MEM/WB registro de pipeline (entre MEM y WB)
    reg [31:0] memwb_result;
    reg        memwb_reg_write;
    reg [4:0]  memwb_rd;

    always @(posedge clk) begin
        // Selección del dato a escribir en el registro destino
        memwb_result    <= exmem_mem_to_reg ? mem_data_out : exmem_result;
        memwb_reg_write <= exmem_reg_write;
        memwb_rd        <= exmem_rd;
    end
endmodule
```