

# Diseño, implementación y validación de un microprocesador RISC-V

Javier Presmanes Cardama  
Iván Vivas Pastor  
Samuel García Such  
Pasqual Moya Ruiz

<b>Risc-V</b>	<b>3</b>
<b>Ventajas e Inconvenientes</b>	<b>4</b>
<b>Diseño</b>	<b>5</b>
3.1 General	5
3.2 Autochequeos	8
3.3 Registros procesador segmentado	9
3.4 Detectores de riesgos	11
<b>Verificación</b>	<b>12</b>
4.1 Básica	12
4.2 Single Cycle	12
4.3 Segmented	15
4.4 Segmented full	16
<b>Mejoras de Verificación</b>	<b>18</b>
<b>Comentarios</b>	<b>19</b>

# 1. Risc-V

RISC-V es una arquitectura de conjunto de instrucciones (ISA) basado en un diseño de tipo RISC (Reduced Instruction Set Architecture). La filosofía RISC se basa en que una ISA de tamaño reducido hace la implementación del hardware más sencilla, teniendo un hardware más sencillo y más rápido ("simple is better, smaller is faster").

El proyecto comenzó en 2010 en la universidad de California en Berkeley, y en 2014 se publica la especificación básica estable. A diferencia de la mayoría de ISAs es una arquitectura abierta y libre, lo que permite que la implementación de esta crezca ya que permite que cualquiera diseñe, fabrique y venda chips y software de RISC-V. Está diseñada para funcionar con una amplia gama de dispositivos, lo que no contenta a ARM.

Tiene 4 ISAs Base Integer, RV32I, RV32E, RV64I y RV128I. Su ISA base contiene menos de 50 instrucciones entre las que se encuentran sumas, restas, comparaciones, saltos... además tiene múltiples extensiones disponibles (multiplicación/división, punto flotante,...).

## 2. Ventajas e Inconvenientes de RISC-V

RISC-V tiene varias ventajas, como se ha mencionado previamente su ISA base tiene un número más reducido de instrucciones, lo cual la hace una ISA más simple que la mayoría de las comerciales. Es modular, admite múltiples extensiones estándar y además es estable ya que su ISA base se ha fijado y no cambiará.

Como se ha mencionado previamente es una arquitectura abierta y libre, el conjunto de instrucciones es para computadoras prácticas, lo cual hace que tenga características para aumentar la velocidad de la computadoras y reducir los costes y uso de energía. El conjunto de instrucciones además admite tres anchos de palabra, 32, 64 y 128 bits.

RISC-V también tiene sus inconvenientes, por ejemplo el número de registros está predeterminado y limitado, por lo que se debe hacer un uso eficiente de estos. A diferencia de muchas otras ISAs, la de RISC-V no está terminada por lo que queda mucho que hacer para optimizar y ampliar la usabilidad de los chips basados en ello.

## 3. Diseño

### 3.1 General

El diseño de nuestro core sigue una estructura modular, que nos permite abstraer por capas los diferentes módulos separando la instalación del cableado y de los autochequeos.

Esto lo hemos conseguido gracias a la utilización de interfaces para el cableado de cada módulo, como se puede apreciar en la Figura 3.1

```
interface ALU_interface();
    logic [31:0] input1;
    logic [31:0] input2;

    logic [3:0] operation;

    logic [31:0] result;
    logic zero;
endinterface//ALU_interface

module ALU (
    ALU_interface alu_wiring
);
```

Figura 3.1 - ALU interface

De esta manera, para cablear la Alu solo hará falta pasarle una interfaz en su cableado, simplificando el código y por lo tanto minimizando los errores.

Siguiendo esta metodología, hemos instanciado dos módulos

- Single Cycle : golden\_model\_core
- Segmented : segmented\_core

Cada módulo se encarga de instanciar y encapsular los elementos que lo componen, como se puede apreciar en la Figura 3.2

```

module golden_model_core
#(
    parameter sintetizable = 1'b0,
    parameter data_bits = 32,
    parameter memory_size = 1024,
    parameter memory_address_bits = $clog2(memory_size),
    parameter program_file = ""
)
(
    golden_interface wires
);

    // Configure Adders and PC

    ADDER ADDER_SUM (
        .adder_wiring (wires.adder_sum_wiring)
    );

    ADDER ADDER_PC (
        .adder_wiring (wires.adder_pc_wiring)
    );

    mux_2_input mux_pc (
        .mux_2_input_wiring (wires.mux_pc_wiring)
    );

    PC PC (
        .pc_wiring (wires.pc_wiring)
    );

    jump_controller jump_controller(
        .jump_controller_wiring (wires.jump_controller_wiring)
    );

    ALU ALU (
        .alu_wiring (wires.alu_wiring)
    );

    memory data_memory (
        .memory_wiring (wires.data_memory_wiring)
    );

```

Figura 3.2 - Módulo golden\_model\_core

Como se puede apreciar, a cada módulo se le asigna su interfaz a través de una interfaz maestra que hemos llamado golden\_interface. Esta interfaz nos permite el encapsulamiento de todo el sistema de cableado, así como el acceso global a cualquier cable del sistema (algo muy útil para verificación)

En esta interfaz hemos instanciado cada uno de los cableados, y nos permite acceder a los cables siguiendo un patrón de matrioska como puede ser *interfaz\_madre.interfaz\_alu.cable\_alu*

Para seguir un orden en el cableado, hemos creado unos módulos a los que llamamos “Designators” encargados únicamente de la asignación de valores a los cables de los diferentes módulos.

Estos “designators” nos permiten acotar los errores a la hora de cablear, ya que tendremos un módulo distinto por cada elemento a cablear, haciendo que los fallos de cableado

queden más localizables, reduciendo el tamaño de los archivos y por lo tanto, beneficiando a la escalabilidad del proyecto.

En las siguientes figuras se pueden ver varios ejemplos de módulos cableados de manera clara, para entender mejor el uso de estos “designators”

```
module adder_sum_wiring_designator(  
    golden_interface wires  
);  
  
assign wires.adder_sum_wiring.input1 = wires.pc_wiring.out;  
assign wires.adder_sum_wiring.input2 = wires.imm_gen_wiring.out;  
  
endmodule
```

Figura 3.3 - Adder sum designator

```
module data_memory_wiring_designator#(  
    parameter data_bits = 32  
)(  
    golden_interface wires  
);  
  
assign wires.data_memory_wiring.clk          = wires.clk;  
assign wires.data_memory_wiring.address      = wires.alu_wiring.result[data_bits - 1 : 2];  
assign wires.data_memory_wiring.input_data   = wires.register_bank_wiring.read_data_2;  
assign wires.data_memory_wiring.write_enable = wires.main_controller_wiring.memory_write;  
assign wires.data_memory_wiring.read_enable  = wires.main_controller_wiring.memory_read;  
  
endmodule
```

Figura 3.4 - Data memory designator

```
module instruction_memory_wiring_designator #(  
    parameter memory_size = 1024,  
    parameter memory_address_bits = $clog2(memory_size)  
)(  
    golden_interface wires  
);  
  
assign wires.instruction_memory_wiring.clk          = wires.clk;  
assign wires.instruction_memory_wiring.write_enable = 1'b0;  
assign wires.instruction_memory_wiring.read_enable  = 1'b1;  
assign wires.instruction_memory_wiring.address      = wires.pc_wiring.out[memory_address_bits - 1 : 2];  
assign wires.instruction_memory_wiring.input_data   = 32'd0;  
  
endmodule
```

Figura 3.5 - Instruction memory designator

Como se puede ver, para poder tener acceso a todo el cableado, es necesario pasarle a cada designator la interfaz golden, la misma interfaz que se le pasa al módulo que instancia todos los módulos del procesador.

Este tipo de encapsulamiento permite cambiar el cableado fácilmente o incluso cambiar el cableado si tuviéramos módulos más complejos en el diseño, ya que es el módulo el que define el cableado.

Por último, hemos creado un módulo top al que hemos llamado “core” donde hemos instanciado:

1. Interfaz maestra del golden y el segmented
2. Designators para el golden y el segmented
3. Módulo golden\_model\_core y segmented\_core

```
//Designator instances for golden core
adder_sum_wiring_designator      adder_sum_wiring_designator      (.wires(golden_core_wires));
adder_pc_wiring_designator       adder_pc_wiring_designator       (.wires(golden_core_wires));
pc_wiring_designator             pc_wiring_designator             (.wires(golden_core_wires));
instruction_memory_wiring_designator instruction_memory_wiring_designator (.wires(golden_core_wires));
data_memory_wiring_designator    data_memory_wiring_designator    (.wires(golden_core_wires));
register_bank_wiring_designator   register_bank_wiring_designator   (.wires(golden_core_wires));
immediate_generator_wiring_designator immediate_generator_wiring_designator (.wires(golden_core_wires));
alu_wiring_designator            alu_wiring_designator            (.wires(golden_core_wires));
mux_mem_wiring_designator        mux_mem_wiring_designator        (.wires(golden_core_wires));
mux_alu1_wiring_designator       mux_alu1_wiring_designator       (.wires(golden_core_wires));
mux_alu2_wiring_designator       mux_alu2_wiring_designator       (.wires(golden_core_wires));
mux_pc_wiring_designator         mux_pc_wiring_designator         (.wires(golden_core_wires));
jump_controller_wiring_designator jump_controller_wiring_designator (.wires(golden_core_wires));
main_controller_wiring_designator main_controller_wiring_designator (.wires(golden_core_wires));
alu_controller_wiring_designator  alu_controller_wiring_designator  (.wires(golden_core_wires));
```

Figura 3.6 - Cableado del procesador single

```
// Designator instances for segmented core
adder_sum_wiring_designator_segmented adder_sum_wiring_designator_segmented (.wires(segmented_core_wires));
adder_pc_wiring_designator_segmented  adder_pc_wiring_designator_segmented  (.wires(segmented_core_wires));
pc_wiring_designator_segmented        pc_wiring_designator_segmented        (.wires(segmented_core_wires));
instruction_memory_wiring_designator_segmented instruction_memory_wiring_designator_segmented (.wires(segmented_core_wires));
data_memory_wiring_designator_segmented data_memory_wiring_designator_segmented (.wires(segmented_core_wires));
register_bank_wiring_designator_segmented register_bank_wiring_designator_segmented (.wires(segmented_core_wires));
immediate_generator_wiring_designator_segmented immediate_generator_wiring_designator_segmented (.wires(segmented_core_wires));
alu_encapsulator_wiring_designator_segmented alu_encapsulator_wiring_designator_segmented (.wires(segmented_core_wires));
mux_mem_wiring_designator_segmented      mux_mem_wiring_designator_segmented      (.wires(segmented_core_wires));
mux_pc_wiring_designator_segmented       mux_pc_wiring_designator_segmented       (.wires(segmented_core_wires));
jump_controller_wiring_designator_segmented jump_controller_wiring_designator_segmented (.wires(segmented_core_wires));
main_controller_wiring_designator_segmented main_controller_wiring_designator_segmented (.wires(segmented_core_wires));
alu_controller_wiring_designator_segmented alu_controller_wiring_designator_segmented (.wires(segmented_core_wires));
reg_if_id_wiring_designator_segmented    reg_if_id_wiring_designator_segmented    (.wires(segmented_core_wires));
reg_id_ex_wiring_designator_segmented    reg_id_ex_wiring_designator_segmented    (.wires(segmented_core_wires));
reg_mem_wb_wiring_designator_segmented   reg_mem_wb_wiring_designator_segmented   (.wires(segmented_core_wires));
reg_ex_mem_wiring_designator_segmented   reg_ex_mem_wiring_designator_segmented   (.wires(segmented_core_wires));
```

Figura 3.7 - Cableado del procesador segmentado

Este módulo nos permite instanciar ambos procesadores de golpe, algo que posteriormente hemos utilizado para la verificación.

## 3.2 Autochequeos

Para el procesador single core, hemos realizado autochequeos para cada una de las instrucciones. Como este modelo tenía que servirnos como golden model, era muy importante que cada señal estuviera controlada.

Para la realización de autochequeos hemos utilizado el mismo procedimiento que para el cableado. Hemos encapsulado todas las propiedades en un documento al cual se le pasa el cableado principal como argumento y se tiene acceso a todo el cableado de todos los módulos para su comprobación.



En la Figura 3.8 se puede ver un pequeño ejemplo de los autochequeos para algunas de las instrucciones básicas.

```
property LUTp;
@ (posedge wires.clk)
wires.main_controller_wiring.alu_option==4'b0111 && wires.alu_controller_wiring.func_3_bits== 3'b000 [-> wires.main_controller_wiring.memory_write == 1'b0] -> wires.main_controller_wiring.register_write == 1'b1 [-> wires.main_controller_wiring.alu_option==4'b0000 [-> wires.alu_controller_wiring.func_3_bits== 3'b010 [-> wires.alu_controller_wiring.alu_operation == 4'b0000 [-> LUTp] else $display("LUT NOT CO

ADDI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b000 [-> wires.alu_controller_wiring.alu_operation == 4'b0000 [-> IFORMAT] else $display("ADDI N
SLTI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b010 [-> wires.alu_controller_wiring.alu_operation == 4'b1010 [-> IFORMAT] else $display("SLTI N
SLTIU: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b011 [-> wires.alu_controller_wiring.alu_operation == 4'b1001 [-> IFORMAT] else $display("SLTIU
XORI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b100 [-> wires.alu_controller_wiring.alu_operation == 4'b0100 [-> IFORMAT] else $display("XORI N
ORI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b110 [-> wires.alu_controller_wiring.alu_operation == 4'b0011 [-> IFORMAT] else $display("ORI NO
ANDI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b111 [-> wires.alu_controller_wiring.alu_operation == 4'b0010 [-> IFORMAT] else $display("ANDI
SLLI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b001 [-> wires.alu_controller_wiring.alu_operation == 4'b1101 [-> IFORMAT] else $display("SLLI
SRLI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b101 [-> wires.alu_controller_wiring.func_7_bits[5] == 1'b1 [-> wires.alu_controller_wiring.alu
SRAI: assert property(@(posedge wires.clk) wires.main_controller_wiring.alu_option==4'b0010 [-> wires.alu_controller_wiring.func_3_bits== 3'b101 [-> wires.alu_controller_wiring.func_7_bits[5] == 1'b0 [-> wires.alu_controller_wiring.alu
```

Figura 3.8 - Autochequeos

Los autochequeos comprueban constantemente los siguientes parámetros en el flanco de subida del reloj:

1. main controller alu option
2. alu controller func 3 bits
3. alu controller func 7 bits
4. alu controller alu operation

Ya que con estos valores es posible determinar de qué formato es la operación y por lo tanto que operación debe hacer la alu así como qué señales tiene que activar o desactivar el main controller.

Por último faltará instanciar dicho módulo en el módulo core del procesador single y pasarle el acceso al cableado, como se puede ver en la Figura 3.9

```
golden_model_autochecks golden_model_autochecks(
    .wires (wires)
);
```

Figura 3.9 - Instanciación módulo autochequeo

### 3.3 Registros procesador segmentado

Para la realización del procesador segmentado y la implementación de los diferentes registros, se ha procedido de la misma manera que para los diferentes módulos del procesador.

Se han creado interfaces y módulos para registros pequeños, llamados:

- ex\_register
- m\_register
- wb\_register

Estos registros a su vez se han instanciado en los registros de un nivel superior, los cuales hemos llamado como marcaba la tarea (*if\_id*, *id\_ex*, *ex\_mem*, *mem\_wb*)

De esta manera la interfaz creada para el cableado del registro *id\_ex* (ejemplificamos con este ya que es el registro con más conexiones) quedaría como se muestra en la siguiente figura:

```

interface register_id_ex_interface #(
    parameter data_bits = 32
);

    logic clk;
    logic clear_pipeline;

    register_ex_interface    ex_wiring();
    register_m_interface    m_wiring();
    register_wb_interface    wb_wiring();

    logic [5 : 0] instruction_in;
    logic [5 : 0] instruction_out;

    logic [data_bits - 1 : 0] pc_in;
    logic [data_bits - 1 : 0] pc_out;

    logic [data_bits - 1 : 0] read_data_1_in;
    logic [data_bits - 1 : 0] read_data_1_out;

    logic [data_bits - 1 : 0] read_data_2_in;
    logic [data_bits - 1 : 0] read_data_2_out;

    logic [data_bits - 1 : 0] immediate_gen_in;
    logic [data_bits - 1 : 0] immediate_gen_out;

    logic [4:0] instruction_11_7_in;
    logic [4:0] instruction_11_7_out;

    logic [2:0] instruction_14_12_in;
    logic [2:0] instruction_14_12_out;

    logic instruction_30_in;
    logic instruction_30_out;

endinterface //reg_id_ex_interface

```

Figura 3.10 - Interfaz registro id ex

Como se puede apreciar, dentro de la propia interfaz del registro hay a su vez una instancia de cada uno de los registros. Esta técnica nos permite ahorrarnos nombres de variables y por lo tanto, fallos.

Para el cableado de todo este sistema se han empleado los “designators” explicados en el apartado [3.1 General](#). Por ejemplo, en el caso anterior el cableado es mostrado en la siguiente Figura

```

module reg_id_ex_wiring_designator_segmented (
    segmented_interface wires
);

    assign wires.reg_id_ex_wiring.clear_pipeline      = 1'b0;

    assign wires.reg_id_ex_wiring.clk                = wires.clk;
    assign wires.reg_id_ex_wiring.pc_in              = wires.reg_if_id_wiring.pc_out;
    assign wires.reg_id_ex_wiring.read_data_1_in     = wires.register_bank_wiring.read_data_1;
    assign wires.reg_id_ex_wiring.read_data_2_in     = wires.register_bank_wiring.read_data_2;
    assign wires.reg_id_ex_wiring.immediate_gen_in   = wires.imm_gen_wiring.out;
    assign wires.reg_id_ex_wiring.instruction_11_7_in = wires.reg_if_id_wiring.instruction_out[11:7];
    assign wires.reg_id_ex_wiring.instruction_14_12_in = wires.reg_if_id_wiring.instruction_out[14:12];
    assign wires.reg_id_ex_wiring.instruction_30_in  = wires.reg_if_id_wiring.instruction_out[30];
    assign wires.reg_id_ex_wiring.instruction_in     = wires.reg_if_id_wiring.instruction_out;

    assign wires.reg_id_ex_wiring.m_wiring.jump_pc_in      = 1'bz;
    assign wires.reg_id_ex_wiring.m_wiring.instruction_func_in[4:0] = 5'bz;
    assign wires.reg_id_ex_wiring.m_wiring.force_jump_in   = 1'bz;
    assign wires.reg_id_ex_wiring.m_wiring.branch_in       = wires.main_controller_wiring.branch;
    assign wires.reg_id_ex_wiring.m_wiring.mem_write_in    = wires.main_controller_wiring.memory_write;
    assign wires.reg_id_ex_wiring.m_wiring.mem_read_in     = wires.main_controller_wiring.memory_read;

    assign wires.reg_id_ex_wiring.ex_wiring.alu_op_in[3:0] = wires.main_controller_wiring.alu_option [3:0];
    assign wires.reg_id_ex_wiring.ex_wiring.alu_src_in     = wires.main_controller_wiring.alu_source;
    assign wires.reg_id_ex_wiring.ex_wiring.lui_src_in     = wires.main_controller_wiring.AuipcLui;

    assign wires.reg_id_ex_wiring.wb_wiring.reg_write_in  = wires.main_controller_wiring.register_write;
    assign wires.reg_id_ex_wiring.wb_wiring.jump_rd_in    = 1'bz;
    assign wires.reg_id_ex_wiring.wb_wiring.mem_to_reg_in = wires.main_controller_wiring.memory_to_register;

endmodule

```

Figura 3.11 - Registro ID EX designator

## 3.4 Detectores de riesgos

Para los detectores de riesgo, se han intentado implementar los 3 módulos principales :

- Hazard detection unit
- Clear pipeline
- Data forwarding

Por falta de tiempo, estos módulos se han dejado en su etapa de diseño, sin pasar a la etapa de implementación o de verificación.

Estos módulos pueden encontrarse en Design/Risk Detectors/

## 4. Verificación

### 4.1 Básica

Para la verificación de los módulos básicos como la memoria, la ALU etc se empleó un test bench sencillo para comprobar las acciones y limitaciones básicas de los componentes. Estos testbenches son sencillos y no se adjuntan imágenes, sin embargo se pueden encontrar dentro del proyecto en la carpeta Testing/Submodules.

### 4.2 Single Cycle

Para la verificación del procesador single cycle se han generado 2 códigos en ensamblador, uno para el cálculo del enésimo número de fibonacci y otro para un bubble sort.

Para comprobar que efectivamente el procesador funcionaba correctamente, se ha generado a su vez dos programas, uno para fibonacci y otro para bubble sort, los cuales nos sirven como golden model, de tal manera que podemos comparar resultados.

El testbench se basa en un módulo top encargado de instanciar una clase de test para fibonacci y otra para bubble sort con un método inicial encargado de esperar a que ambos programas, el bubble sort y el fibonacci como se muestra a continuación.

```
initial begin
    @(bubble_sort_verificator.test_finished == 1 && fibonnaci_verificator.test_finished == 1);

    $stop();
end
```

Figura 4.1 - Initial del testbench

Cada uno de los tests, se encarga de la instancia de procesador single y de los chequeos con el modelo golden para comprobar que funciona correctamente el procesador. Esto se hace cómo se puede ver a continuación en la Figura, mediante la comprobación con el contenido de la memoria del procesador.

```

class fibonacci_test;

    fibonacci_golden golden_model;

    int total_numbers_to_verify;    // Total amount of numbers to be verified at memory

    task init(int total_numbers);
        golden_model = new();
        golden_model.init();

        this.total_numbers_to_verify = total_numbers;
    endtask

    task check();

        $display("[FIBONNACI CHECKING START]");

        assert (Phase2_testbench.fibonacci_verificator.core.data_memory.data_pool[0] == 32'd0)
        else    $display("[ERROR] POSITION 0 OF FIBONACCI IS WRONG");

        assert (Phase2_testbench.fibonacci_verificator.core.data_memory.data_pool[1] == 32'd1)
        else    $display("[ERROR] POSITION 0 OF FIBONACCI IS WRONG");

        for(int x = 2 ; x < this.total_numbers_to_verify; x++)begin
            golden_model.compute_new_number();

            assert (Phase2_testbench.fibonacci_verificator.core.data_memory.data_pool[x] == golden_model.current_number)
            else    $display("[ERROR] POSITION %d OF FIBONACCI IS WRONG", x);
        end

        $display("[FIBONNACI CHECKING FINISHED]");

    endtask //automatic
endclass //fibonacci_scoreboard

```

Figura 4.2 - Comprobaciones test fibonacci

Como se puede observar, el test consiste en comprobar que el modelo golden y el contenido real de la memoria es el esperado mediante un conjunto de asserts.

El funcionamiento del test del bubble sort es similar.

Por último, se puede comprobar con questa sim el resultado de los \$display() como se muestra a continuación

```

>>> CHARGING FILE [C:/Users/Javier Presmanes/Documents/Universidad/3ro Teleco/Primer Cuatri/ISDIGI/Practicas/ISDIGI-Projects/Practica 3 - RISC V/Core/Testing/Fase2/Phase2_testbench.mem] IN MEMORY
>>> CHARGING FILE [C:/Users/Javier Presmanes/Documents/Universidad/3ro Teleco/Primer Cuatri/ISDIGI/Practicas/ISDIGI-Projects/Practica 3 - RISC V/Core/Testing/Fase2/Phase2_testbench.mem] IN REGISTER BANK
>>> CHARGING FILE [C:/Users/Javier Presmanes/Documents/Universidad/3ro Teleco/Primer Cuatri/ISDIGI/Practicas/ISDIGI-Projects/Practica 3 - RISC V/Core/Testing/Fase2/Phase2_testbench.mem] IN MEMORY
>>> CHARGING FILE [C:/Users/Javier Presmanes/Documents/Universidad/3ro Teleco/Primer Cuatri/ISDIGI/Practicas/ISDIGI-Projects/Practica 3 - RISC V/Core/Testing/Fase2/Phase2_testbench.mem] IN REGISTER BANK
[FIBONNACI CHECKING START]
[FIBONNACI CHECKING FINISHED]
[BUBBLE SORT CHECKING START]
[BUBBLE SORT CHECKING FINISHED]
** Note: $stop : C:/Users/Javier Presmanes/Documents/Universidad/3ro Teleco/Primer Cuatri/ISDIGI/Practicas/ISDIGI-Projects/Practica 3 - RISC V/Core/Testing/Fase2/Phase2_testbench.v(39)
Time: 4190 ns Iteration: 4 Instance: /Phase2_testbench
Break in Module Phase2_testbench at C:/Users/Javier Presmanes/Documents/Universidad/3ro Teleco/Primer Cuatri/ISDIGI/Practicas/ISDIGI-Projects/Practica 3 - RISC V/Core/Testing/Fase2/Phase2_testbench.v line 39

```

Y a su vez, el resultado del contenido de las memoria en el apartado “memory list”

- Fibonacci:

```

!_testbench/fibonnaci_verificator/core/data_memory/data_pool
00000000 00000000 00000001 00000001 00000002
00000004 00000003 00000005 00000008 0000000d
00000008 00000015 00000022 00000037 00000059
0000000c 00000090 000000e9 00000179 00000262
00000010 000003db 0000063d 00000a18 00001055
00000014 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000018 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
0000001c xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000020 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000024 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000028 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
0000002c xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000030 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000034 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx

```

- Bubble sort:

```

!_testbench/bubble_sort_verificator/core/data_memory/data_pool
00000000 00000003 00000003 00000005 00000006
00000004 00000007 00000008 00000018 00000019
00000008 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
0000000c xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000010 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000014 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000018 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
0000001c xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000020 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000024 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000028 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
0000002c xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000030 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
00000034 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx

```

Los códigos de los programas se pueden encontrar en la carpeta Testing/Programs/Complex

## 4.3 Segmented

Para la verificación del procesador segmentado y una versión mejorada de la verificación de todas las instrucciones del procesador, se ha creado un testbench que genera una instancia de un módulo general llamado "Instruction\_core\_validator" el cual genera una instancia de un procesador segmentado y de un procesador single.

A cada una de las instancias, se le pasa como parámetro los siguientes valores:

- program\_file : programa que queremos cargar en memoria
- expected\_mem\_value : Valor final que tiene que aparecer en memoria para comparar

Con el primer parámetro, podemos pasarle como valor la referencia a un programa creado específicamente para cada una de las instrucciones que queremos verificar con el procesador.

Para que sea compatible con la versión segmentada (sin comprobación de errores) y con la versión single, hemos introducido instrucciones extra que no tienen ningún impacto y con la única finalidad de que el procesador segmentado no necesite de comprobación de errores ya que entre cada instrucción útil hay 5 instrucciones que no sirve para nada.

Esta instrucción simula un NOP, pero en vez de "no hacer nada" simplemente intenta escribir un 1 en el registro x0, lo cual por diseño no está permitido, de esta manera esta instrucción no tendrá ningún impacto en nuestro programa.

Además, para saber cuando han terminado ambos procesadores de hacer sus cálculos, se ha añadido al final una instrucción fijada por nosotros la cual es escribir un 0 en el registro x0, esto siempre da un valor fijo y hacemos que el sistema espere hasta esa instrucción para parar.

En el módulo de testbench, podemos ver la instanciación de los diferentes módulos para las diferentes instrucciones del procesador a continuación:

```

Instruction_core_validator #(.program_file("ADD/add.mem"), .expected_mem_value(2)) add_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("ADDI/addi.mem"), .expected_mem_value(1)) addi_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("AND/and.mem"), .expected_mem_value(1)) and_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("ANDI/andi.mem"), .expected_mem_value(1)) andi_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("AUIPC/auipc.mem"), .expected_mem_value(24)) auipc_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("BEQ/peq.mem"), .expected_mem_value(1)) beq_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("BGE/bgeu.mem"), .expected_mem_value(1)) bge_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("BGEU/bgeu.mem"), .expected_mem_value(1)) bgeu_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("BLT/blt.mem"), .expected_mem_value(1)) blt_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("BLTU/bltu.mem"), .expected_mem_value(1)) bltu_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("BNE/bne.mem"), .expected_mem_value(1)) bne_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("JAL/jal.mem"), .expected_mem_value(28)) jal_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("JALR/jalr.mem"), .expected_mem_value(0)) jalr_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("LUI/lui.mem"), .expected_mem_value(0)) lui_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("LW/lw.mem"), .expected_mem_value(1)) lw_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("OR/or.mem"), .expected_mem_value(1)) or_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("ORI/ori.mem"), .expected_mem_value(1)) ori_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SLL/sll.mem"), .expected_mem_value(2)) sll_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SLLI/slli.mem"), .expected_mem_value(2)) slli_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SLT/slt.mem"), .expected_mem_value(1)) slt_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SLTI/lti.mem"), .expected_mem_value(1)) slti_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SLTIU/ltiu.mem"), .expected_mem_value(1)) sltiu_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SLTU/ltu.mem"), .expected_mem_value(1)) sltu_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SRA/sra.mem"), .expected_mem_value(1)) sra_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SRAI/srai.mem"), .expected_mem_value(1)) srai_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SRL/srl.mem"), .expected_mem_value(1)) srl_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SRLI/srli.mem"), .expected_mem_value(1)) srli_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SUB/sub.mem"), .expected_mem_value(1)) sub_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("SW/sw.mem"), .expected_mem_value(1)) sw_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("XOR/xor.mem"), .expected_mem_value(1)) xor_validator (.clk(clk), .reset(reset));
Instruction_core_validator #(.program_file("XORI/xori.mem"), .expected_mem_value(1)) xori_validator (.clk(clk), .reset(reset));

```

El parámetro de "expected mem value" es un valor que hemos implementado para poder simplificar el test bench.

El programa espera a que termine de ejecutarse todas las instrucciones y cuando esto sucede, cada programa pequeño ha cambiado un valor concreto de memoria (siempre la misma dirección), y como nosotros como programadores sabemos lo que debería de haber en ese espacio de memoria cuando termine el programa, podemos indicarle al testbench el resultado de antemano.

Los códigos están hechos de tal manera que se auto-verifiquen si ese resultado de la memoria es el que toca, como por ejemplo, probar que el BEQ salta a una dirección de memoria cuando se cumple una condición, y escribe un valor determinado en la dirección general para verificación.

Utilizando este método y utilizando como modelo golden el procesador single, podemos verificar que todas las etapas intermedias del procesador segmentado funcionan correctamente.

## 4.4 Segmented full

Por último, si estuvieran implementados correctamente los módulos de detección de riesgo, se debería de chequear el programa de Fibonacci y el de Bubble sort utilizando el procesador single como golden model



Para este testbench se emplea la técnica anterior, esperar a que ambos procesadores terminen de ejecutar los programas con una instrucción final que nosotros como programadores hemos introducido y cuando terminen, comparar la memoria de ambos procesadores y comprobar que su contenido es el correcto.

A continuación se muestra el código para la serie de fibonacci, que es idéntico al bubble sort

```
task fibo_test();
    int Nmax_num = 20;
    int dir_ini_core = 0;
    int dir_ini_DUV = 0;
    int file;

    string relative_path = "../../Proyecto Questasim/Basic Testing/../../";
    string complete_path = {relative_path,"Core/Testing/Fase3/Out/fib_memory_testbench_out.txt"};

    file = $fopen(complete_path,"w");

    $display("[FIBBONACCI] starting test!");

    // Write header for the file
    $fwrite(file,"[GOLDEN], [SEGMENTED]\n");

    fork
    begin
        // Wait till the golden model finishes
        @(testbench.cores.fibo_core.golden_core_wires.instruction_memory_wiring.output_data != 32'h00000013);
        $display("[FIBBONACCI] Golden model finished");
    end
    begin
        // Wait till the segmented model finishes
        @(testbench.cores.fibo_core.segmented_core_wires.instruction_memory_wiring.output_data != 32'h00000013); //instruccion fin real
        $display("[FIBBONACCI] Segmented model finished");
    end
    join

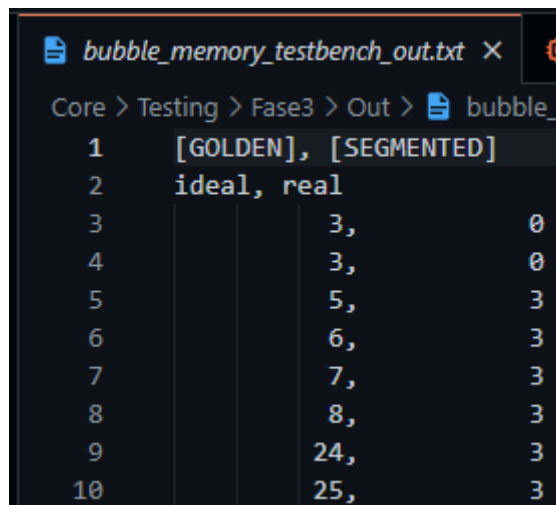
    $display("[FIBBONACCI] Starting with the checking");
    // Write data into file and check if both are the same
    for (int i = 0; i < Nmax_num; i++) begin
        int golden_value = testbench.cores.fibo_core.golden_core.data_memory.data_pool[i];
        int segmented_value = testbench.cores.fibo_core.segmented_core.data_memory.data_pool[i];

        $fwrite(file,"%d, %d\n",golden_value, segmented_value);

        $assert (golden_value == segmented_value) else $info("El elemento numero ",i," de fibonacci NO COINCIDE");
    end

    $display("[FIBBONACCI] finishing test!");
    $fclose(file);
endtask
```

Además el código guarda los datos tanto del single como del segmentado en un archivo de texto como se puede ver a continuación



	[GOLDEN], [SEGMENTED]	ideal, real
1		
2		
3	3, 0	
4	3, 0	
5	5, 3	
6	6, 3	
7	7, 3	
8	8, 3	
9	24, 3	
10	25, 3	

fib_memory_testbench_out.txt				
Core > Testing > Fase3 > Out > fib_memory				
1	[GOLDEN], [SEGMENTED]			
2			0,	20
3			1,	0
4			1,	0
5			2,	0
6			3,	0
7			5,	0
8			8,	0
9			13,	0
10			21,	0
11			34,	0
12			55,	0
13			89,	0
14			144,	0
15			233,	0
16			377,	0
17			610,	0
18			987,	0
19			1597,	0
20			2584,	0
21			4181,	0

## 5. Mejoras de Verificación

Debido a la complejidad de la tarea y los recursos de los que disponíamos, la parte de verificación se ha quedado en una versión muy prematura de lo que debería de ser una verificación correcta de cualquier sistema.

Es por esto que hemos invertido tiempo en investigar métodos de verificación más reales y los queremos mencionar en el siguiente apartado.

La primera mejora que tendría que ser implementada sería la generación de instrucciones aleatorias para poder verificar que efectivamente el microprocesador es capaz de cumplir con todas las especificaciones de diseño

Además una implementación de esta etapa de verificación permitiría una cómoda comprobación de los corner cases de todos los datos así como testear correctamente partes como los detectores de riesgos.

## 6.Comentarios

No se ha podido realizar el estudio de frecuencia y espacio mediante quartus debido a que siempre nos lanzaba el mismo error

```
Problem Details
Error:
Internal Error: Sub-system: VRFX, File: /quartus/synth/vrfx/verific/verilog/verimisc_elab.cpp, Line: 460
type
Stack Trace:
0x51b90: vrfx_altera_assert + 0x20 (synth_vrfx)
0x182ef3: VeriInst::Initialize + 0x83 (synth_vrfx)
0x18b154: VeriModuleInstantiation::Initialize + 0x64 (synth_vrfx)
0xf0588: VeriModule::InitializeInternals + 0xf8 (synth_vrfx)
0xf0442: VeriModule::Initialize + 0xf2 (synth_vrfx)
0xf01d8: VeriInterface::ElaborateInterfaceObject + 0xc8 (synth_vrfx)
0x15f627: VeriInterfaceTypeDef::EvaluateInterfaceConstraint + 0x547 (synth_vrfx)
0x138e63: VeriDataType::EvaluateInterfaceConstraint + 0xb3 (synth_vrfx)
0x13b86d: VeriVariable::InitializeInterfacePort + 0x6d (synth_vrfx)
0xf03fc: VeriModule::Initialize + 0xac (synth_vrfx)
0xf4a0: VeriModule::Elaborate + 0x510 (synth_vrfx)
0x70b1a: VRFX_VERIFIC_VERILOG_ELABORATOR::elaborate + 0x6da (synth_vrfx)
0x69bb7: VRFX_ELABORATOR::elaborate + 0xd7 (synth_vrfx)
0xdabff: SGN_FN_LIB::elaborate + 0x24f (synth_sgn)
0xe382f: SGN_FN_LIB::start_vrf_flow + 0xf (synth_sgn)
0xe1e0b: SGN_FN_LIB::start + 0xa1b (synth_sgn)
0xc29ca: SGN_EXTRACTOR::single_module_extraction + 0x15a (synth_sgn)
0xb75a4: SGN_EXTRACTOR::recursive_extraction + 0x204 (synth_sgn)
0xb0943: SGN_EXTRACTOR::extract + 0xd3 (synth_sgn)
0x1324e: sgn_qic_full + 0x19e (synth_sgn)
0x4351: qsyn_execute_sgn + 0x131 (quartus_map)
0x13f9c: QSYN_FRAMEWORK::execute_core + 0x12c (quartus_map)
0x13aa6: QSYN_FRAMEWORK::execute + 0x496 (quartus_map)
0x112bc: qexe_do_normal + 0x1ec (comp_qexe)
0x16142: qexe_run + 0x432 (comp_qexe)
0x16e51: qexe_standard_main + 0xc1 (comp_qexe)
0x1b08b: qsyn_main + 0x51b (quartus_map)
0x12e98: msg_main_thread + 0x18 (CCL_MSG)
0x1467e: msg_thread_wrapper + 0x6e (CCL_MSG)
0x16660: mem_thread_wrapper + 0x70 (ccl_mem)
0x12761: msg_exe_main + 0xa1 (CCL_MSG)
0x29872: __tmainCRTStartup + 0x10e (quartus_map)
0x17033: BaseThreadInitThunk + 0x13 (KERNEL32)
0x52650: RtlUserThreadStart + 0x20 (ntdll)

End-trace

Executable: quartus_map
Comment:
None

System Information
Platform: windows64
OS name: Windows 10
OS version: 10.0

Quartus Prime Information
Address bits: 64
Version: 17.1.0
Build: 590
Edition: Lite Edition
```

Por el mismo motivo no hemos podido sacar una imagen RTL que podría haber sido utilizado tanto para debugear las conexiones como simplemente para ver como ha sintetizado cada módulo el sintetizador.