Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

Implementación del RV32IM monociclo en VHDL

Autor:

Mendoza González, Jorge

Tutor(es):

Cáceres Gómez, Santiago

Departamento de Tecnología Electrónica

Valladolid, Julio 2022.

Resumen

El objetivo primordial de este trabajo es el diseño del circuito en lenguaje VHDL y la implementación sobre la placa Basys 3 Artix-7 de un procesador RISC-V de 32 bits monociclo que incluya la extensión para realizar la multiplicación y la división (RV32IM).

Se realiza el diseño del circuito del Multiplicador de Árbol de Wallace, el cual se genera mediante una aplicación desarrollada en C++.

Los tres circuitos de multiplicación analizados para implementar en el procesador son: Multiplicador Paralelo CSA planteado por Deschamps, el circuito de multiplicación que sintetiza Xilinx y el Multiplicador de Árbol de Wallace desarrollado en este trabajo. Las características que se compararán de los multiplicadores serán los recursos hardware empleados y el tiempo de respuesta.

Abstract

The main objective of this work is the design of the circuit in VHDL language and the implementation on the Basys 3 Artix-7 board of a 32-bit RISC-V single-cycle processor including the extension to perform multiplication and division (RV32IM).

The design of the Wallace Tree Multiplier circuit is carried out, which is generated by means of an application developed in C++.

The three multiplication circuits analysed for implementation in the processor are: CSA Parallel Multiplier proposed by Deschamps, the multiplication circuit synthesised by Xilinx and the Wallace Tree Multiplier developed in this work. The characteristics of the multipliers to be compared will be the hardware resources employed and the response time.

Palabras clave

FPGA, RISC-V, VHDL, Circuitos de multiplicación, Basys 3 Artix-7.

Índice

Índice de Figuras

Índice de tablas

Capítulo 1: Introducción

1.1 Presentación del Trabajo Final de Grado

El presente estudio surge a partir de la propuesta del profesor **Santiago Cáceres Gómez** del departamento de Tecnología Electrónica en la Escuela de Ingenieros Industriales (E.I.I.) de la Universidad de Valladolid (U.V.A.) sobre el Trabajo Final de Grado "**Implementación del RV32IM monociclo en VHDL**". El trabajo ha sido desarrollado por el alumno Jorge Mendoza González estudiante del grado de Ingeniería Electrónica Industrial y Automática (I.E.I.A.).

1.2 Objetivos

Como se presenta en el título del proyecto, el objetivo principal del proyecto es diseñar un procesador RISC-V de 32 bits monociclo que incluya la extensión de multiplicación y se desarrolle en VHDL. Para poder cumplir este objetivo se deben cumplir otros objetivos parciales, que son:

- Diseñar un procesador RISC-V monociclo en lenguaje VHDL e implementarle sobre el hardware aportado.
- Diseñar el algoritmo que genere la descripción del circuito para el Multiplicador de Árbol de Wallace para el procesador RISC-V y compararle con otros dos algoritmos de multiplicación.

Una vez concluidos estos objetivos previos se puede realizar el objetivo principal del proyecto que será implementar el procesador RISC-V monociclo que incluya el módulo de multiplicación y división en la FPGA Basys 3. El algoritmo de multiplicación que se empleará en el procesador RISC-V será el que cumpla con las mejores características de ocupación de elementos en la FPGA y de velocidad de respuesta. Para comprobación de un correcto funcionamiento se realizará la programación de la placa y comprobación de su funcionamiento.

1.3 ¿Por qué el uso de RISC-V y una FPGA?

Dentro de las arquitecturas de procesadores se encuentran dos que trabajan con la arquitectura Von Neuman que son Computador con Conjunto Complejo de Instrucciones (CISC) y Computador con Conjunto Reducido de instrucciones (RISC).

La arquitectura RISC tiene una ventaja sobre la CISC es que las instrucciones de la arquitectura CISC son muchas, no tienen un tamaño fijo y son muy complejas. Por lo que un procesador con arquitectura CISC de alto rendimiento actual para ejecutar una de las instrucciones complejas tiene que dividirla en varias más simples llamadas microinstrucciones. Esto supone que para ejecutar una instrucción puede tardar varios ciclos de reloj.

En cambio la arquitectura RISC tiene menos instrucciones y son de tamaño fijo, lo que hace que una instrucción se pueda ejecutar en un solo ciclo de reloj (monociclo) aunque también se puede tener esta arquitectura segmentada o

pineapple digital lo que supone que cada ciclo hará una parte de la ejecución (obtención de la instrucción, decodificación, procesamiento y almacenamiento) y ejecutando dos instrucciones casi al mismo tiempo. Esto también lo puede realizar la arquitectura CISC, pero de manera menos efectiva. Por lo que la arquitectura RISC consigue mejor paralelismo y simultaneidad de la ejecución de instrucciones. [1, p. 17]

Como las instrucciones de la arquitectura son más simples requieren de menos hardware para ejecutarse lo que supone un coste menor del chip.

Una de las desventajas que tiene RISC es que no puede realizar operaciones entre valores del registro y de la memoria de datos en una instrucción y en cambio la arquitectura CISC sí que puede realizarlo.

Observada esta comparación entre ambas arquitecturas, se darán los motivos por los que este trabajo se va a realizar con la arquitectura RISC-V perteneciente a la arquitectura RISC.

Una de las principales ventajas que da la arquitectura RISC-V sobre muchos otros procesadores es que es de libre uso y no necesita de comparar licencias para su empleo. La simplicidad que tiene la arquitectura hace que se pueda utilizar en contextos académicos. La propia empresa fomenta su empleo académico para dar a conocer la arquitectura a las nuevas generaciones que se adentran dentro de este mundo.

Otra de las ventajas es su modularidad. Esta arquitectura se compone de un módulo base que permite la realización de funciones básicas en tres anchos de palabra (32, 64 y 128 bits) y a su vez tiene módulos que se le pueden añadir a la arquitectura dependiendo del trabajo que se le vaya a dar. Que sea modular hace que se pueda emplear desde proyectos académicos, comerciales, de investigación o supercomputadoras.

Una de las motivaciones para realizar el proyecto con esta arquitectura es el conocer esta arquitectura que está en auge y está llamada a ser una arquitectura importante en los próximos años en el mundo de los procesadores.

Actualmente si se busca en los buscadores web RISC-V y navegamos entre las noticias se puede observar como la arquitectura está empezando a ser usada para el empleo de ordenadores o superordenadores que podrían hacer la competencia a los actuales como se indica en la noticia “Los superordenadores x86 y ARM ya tienen rival RISC-V: Monte Cimone” [2]. Así como uno de los fabricantes de procesadores más potentes del mercado está abandonando la arquitectura x86 para fabricar los procesadores de la nueva generación de computación de alto rendimiento bajo la arquitectura RISC-V.[3], [4]

El dispositivo en el que implementar el proyecto pueden ser tres: un microprocesador, una matriz de puertas lógicas programable en campo (FPGA) o un dispositivo lógico programable complejo (CLPD).

Un microprocesador es un elemento hardware que contiene un circuito integrado fijo. El procesador RISC-V que se va a desarrollar se puede implementar en microprocesador, pero esta fuera del alcance del tema del trabajo. Este dispositivo pertenece a la familia de los circuitos integrados de aplicación especifica (ASIC). Los dispositivos ASIC se diseñan a medida y se utilizan en tareas muy específicas, por lo tanto, esta opción está descartada en este este trabajo.

Las otras dos opciones sí que son reprogramables y pueden ser una de la opción a elegir para la elaboración del proyecto.

Los CLPD son dispositivos que realizan circuitos. Está compuesto por matrices de bloques de arreglos lógicos programables (*PLA*). Los bloques de PLA se disponen de una matriz dedicada a funciones *AND* seguida de otra matriz de *OR* para relacionar las funciones *AND.* Una ventaja que tiene este dispositivo es que cuando se apaga o se deja de suministrar energía mantendrá el circuito, debido a que funciona como una memoria ROM. Están destinados para ejecutar tareas combinacionales y secuenciales.

Las matrices de puertas lógicas programables en campo son dispositivos que se componen de bloques lógicos configurables. Estos bloques pueden incluir memorias RAM, LUTs, procesadores y DSPs. Por lo tanto, estos dispositivos son muy complejos. Las FPGAs se emplean para prototipar circuitos que se implementaran en microprocesadores.

Las FPGAs tienen como ventaja que disponen de memoria ROM que almacena la información del circuito. Algunos dispositivos hardware que no disponen de esta memoria ROM cuando se le quita la energía y después se le devuelve la energía, el circuito que tenía se pierde y para volver a tener hay que introducirle. En cambio, las FPGAs actuales disponen de este bloque de memoria ROM y consigue tener el circuito después de restablecer la energía.

cuando la FPGA recupera la energía se restable el circuito. Otra ventaja es que en una misma FPGA se pueden tener varias funcionalidades trabajando en paralelo.

Una de las ventajas que tiene las FPGAs sobre los CLPD es que contiene miles de bloques lógicos más. Una cosa a favor de los CLPD es que tienen tiempos de ejecución constantes a diferencia de las FPGAs. Pero para desarrollar un procesador es más indicado una FPGA dado que contiene bloques lógicos dedicados para desarrollar operaciones aritméticas y lógicas, como pueden ser los DSP. Por lo tanto, el dispositivo hardware que se empleará para el proyecto será la FPGA. En concreto será una FPGA de la gama Artix 7 de Xilinx.

1.4 ¿Por qué se van a comparar varios algoritmos de multiplicación?

Se van a comparar varios circuitos de multiplicación porque es muy importante conocer cuál de los circuitos es el que mejor encaja en el procesador.

Porque se puede emplear un algoritmo que emplee muchos recursos hardware o uno que genere mucho retardo perjudicando a la ejecución. En la situación que un multiplicador emplee mucho tiempo en realizarse puede que el siguiente ciclo de reloj llegue y no se haya obtenido el resultado de la multiplicación dando error en la ejecución y posiblemente un fallo en la realización de la tarea que estuviera desarrollando este procesador.

En el caso de que emplee muchos recursos hardware puede que no tenga suficientes bloques lógicos para poder crear el circuito que se define en el algoritmo. Consecuentemente se deberá tener en cuenta además del circuito empleado, la cantidad de hardware disponible en la FPGA.

Por lo tanto, en este trabajo se compararán los algoritmos de multiplicación de multiplicador paralelo CSA planteado en [1], el multiplicador de Árbol de Wallace y un multiplicador que emplee las librerías y símbolo de multiplicación (*).

1.5 Estructura del proyecto

El proyecto se estructura en siete apartados que se organizan de la siguiente forma. El segundo comprende las herramientas empleadas en el desarrollo del trabajo. Se explicarán las características y algunos de los usos que se les da. El tercer capítulo comprenderá la arquitectura RISC-V desarrollada y las pruebas realizadas en su elaboración. El cuarto capítulo propondrá los tres algoritmos de multiplicación que se comprarán. Además de comparar los tres algoritmos se expondrá el algoritmo que implementa el circuito de multiplicación de Árbol de Wallace desarrollado en el trabajo. En el quinto capítulo se expondrán las conclusiones sobre los objetivos. Seguido se añadirá la bibliografía empleada en el trabajo, así como los apartados añadidos como anexos.

En los anexos se encuentran los estándares de las instrucciones empleadas para la realización de dos códigos de la memoria de instrucciones y los códigos VHDL empleados en la realización del RISC-V y los multiplicadores, y el código en C++ que genera el circuito de multiplicación de Árbol de Wallace en VHDL.

Capítulo 2: Herramientas

2.1 Basys 3-Artix 7 FPGA Board [5]–[7]

Es una tarjeta de desarrollo basada en la FPGA ARTIX 7 que comercializa Xilinx. La tarjeta Basys 3 además de contar con la FPGA cuenta con una gran variedad de entradas y salidas, con un control de encendido de la placa (interruptor), botón de reset de la FPGA, una entrada micro USB para la programación de la FPGA y alimentación de la placa, cuatro bloques de pines que son empleados para la conexión de módulos adicionales que se quieran conectar a la placa y también dispone de leds de indicación que no pueden ser programables. También dispone de entradas y salidas básicas, y una salida VGA.

La placa Basys 3 se observa en la Ilustración 1.

Ilustración 1: Placa de desarrollo Basys 3.[7, p. 18]

Seguidamente se expondrá el funcionamiento de la placa y algunos de sus componentes.

Como se ha mencionado se tienen entradas y salidas básicas que sus valores únicamente pueden ser encendido o apagado (1/0). Las entradas son: cinco pulsadores y dieciséis interruptores. Las salidas digitales son dieciséis leds y las analógicas son: cuatro displays de siete segmentos.

La alimentación a la placa puede ser mediante micro-USB o mediante alimentación externa de 5v, pudiéndose seleccionar el modo de alimentación mediante un jumper. Los distintos voltajes empleados en la tarjeta son: 5v para la alimentación de reguladores de voltaje y puertos de USB host, 3.3v para la alimentación de las entradas y salidas de la FPGA, puertos USB, relojes, conectores, módulos de entrada y la memoria Flash. El núcleo de la FPGA es alimentado con 1v y con 1.8v se alimenta los bloques auxiliares de la FPGA y la memoria RAM.

La tarjeta contiene una memoria Flash SPI de 32Mb, que se conecta a la FPGA mediante un puerto SPI en modo QUAD. Aunque la memoria tiene un espacio de 32Mb, el usuario solo puede emplear 16Mb.

El reloj/oscilador que tiene la tarjeta Basys 3 es de 100MHz.

La conexión VGA se emplea para controlar monitores RGB con la profundidad de 12bits (4 de cada color base) lo que puede generar hasta 4096 colores, además cuenta con sincronización vertical y horizontal.

La placa tiene comunicación USB-UART o micro USB mediante el protocolo RS-232. La placa tiene el chip "FTDI FT2232HQ" con los drivers necesarios para una correcta comunicación entre la FPGA y el dispositivo que contenga la información.

La placa contiene cuatro conectores PMOD que son empleados para añadir a la placa entradas o salidas adicionales. Los conectores contienen doce pines, de los doce pines solo ocho serán empleados para la comunicación de información. Uno de los conectores PMOD que contiene la placa puede configurarse para transmitir información analógica, este conector se denomina "JXADC". Este conector puede transmitir información analógico o digital dependiendo de la configuración que de la placa. Para convertir las señales analógicas que se transmiten por este módulo la placa contiene un convertidor ADC (analógico a digital) de 12 bits de doble canal, pudiendo llegar a velocidades de muestreo de 1MSPS.

La FPGA, que es el núcleo de la tarjeta de desarrollo, emplea tecnología Artix 7. Esta FPGA contiene 33.280 celdas lógicas divididas en 5.200 partes. Cada parte contiene cuatro LUTs (*Look up table* o tabla de consulta) de seis entradas y ocho flip-flops. La FPGA contiene también un bloque de memoria RAM que tiene una velocidad de 1,8Mb, cinco administradores de señales de reloj con sistema de control PLL (lazo de seguimiento de fase), tiene un total de 20800 LUTs, 90 sectores para DSP, un reloj interno de 450MHz y el convertidor ADC mencionado anteriormente.

Por lo tanto, la placa contiene 2 relojes uno de 450MHz y otro de 100MHz.

2.2 FPGA [8]–[10]

F*ield programmable gate arrays (FPGAs)* es un dispositivo hardware programable que se puede reconfigurar después de la fabricación. Este dispositivo está compuesto por puertas lógicas e interconexiones programables que se agrupan en bloques de memoria configurables (bloques lógicos), bloques de interconexión y DSP (*Digital signal procesing* o Procesador de señales digitales), además contienen bloques de entrada y salida. La estructura general de una FPGA se observa en la Ilustración 2.

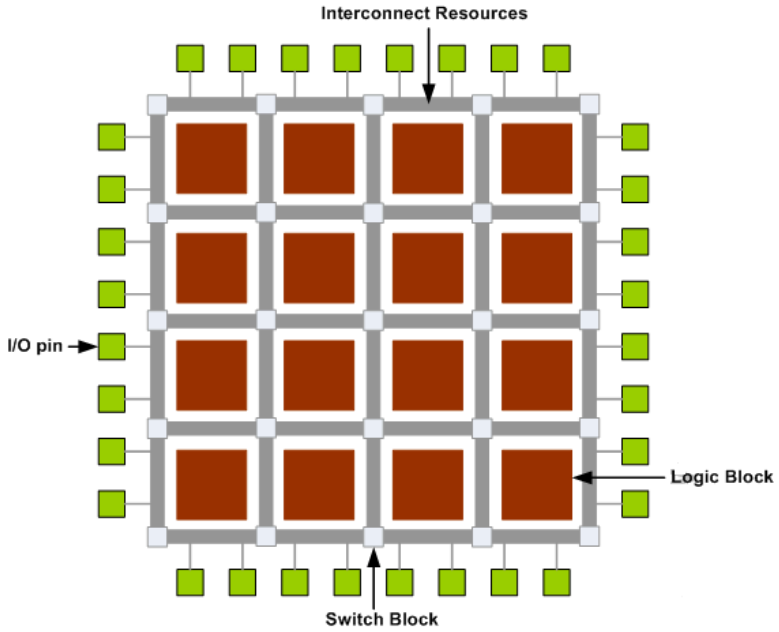

Ilustración 2: Estructura general de una FPGA. [11]

En la Ilustración 2 se identifican cuatro elementos en la arquitectura de las FPGAs entre los que se encuentran Bloques de conexiones de entrada y salida (cuadros rojos), Bloques de interconexiones de las nets (cuadros blancos), las nets (líneas grises) y los bloques de conexión de entrada y salida (cuadrados verdes).

Las primeras FPGAs fueron introducidos por Xilinx en el año 1984 pero no fueron llamados así hasta que Actel popularizo el termino en el año 1988[8]. La primer FPGA contenía 85.000 transistores, 128 celdas lógicas, 64 bloques lógicos programables y un reloj de una frecuencia superior a 50MHz. Comparando los datos de la primera FPGA con una FPGA actual se observa el avance en el sector de los semiconductores debido a que actualmente una FPGA de uso medio contiene 35.000 millones de transistores y 100.000 bloques lógicos. Respecto las primeras FPGAs las características han mejorado como pueden ser la capacidad, la velocidad, reduciendo el coste y la energía por función. Fueron la solución barata y efectiva en comparación a los ASIC (*application specific integrated circuits*).

Las FPGAs pueden ser programados por varios lenguajes de descripción hardware como son: Verilog, o VHDL.

En las FPGAs los bloques lógicos se colocan en un arreglo bidimensional en el que los bloques de interconexión se organizan como canales de enrutamiento verticales y horizontales entre los bloques lógicos. Los canales de enrutamiento contienen “cables” e interruptores programables que permiten realizar las conexiones deseadas.

Los bloques lógicos de las FPGAs tienen varias arquitecturas. La más empleada son las LUTs. Las LUTs contienen celdas de almacenamiento que permiten

implementar pequeñas funciones lógicas a través de unas entradas y una tabla de la verdad.

Las entradas de la tabla de la verdad son los valores que controlan los multiplexores y como mucho se tendrán seis entradas. Las LUTs tienen un almacenamiento de memoria RAM que son los valores que se podrán seleccionar mediante la tabla de la verdad.

Una única LUT puede implementar una función booleana que tenga una cantidad *n* de entradas de control, por lo que tendrá 2^n variables de almacenamiento. El máximo de entradas que puede tener una LUT es seis.

Una LUT que tiene tres entradas requiere de ocho celdas de almacenamiento. En la Ilustración 3 se puede observar una LUT de tres entradas por lo tanto se observa que tiene ocho celdas de almacenamiento.

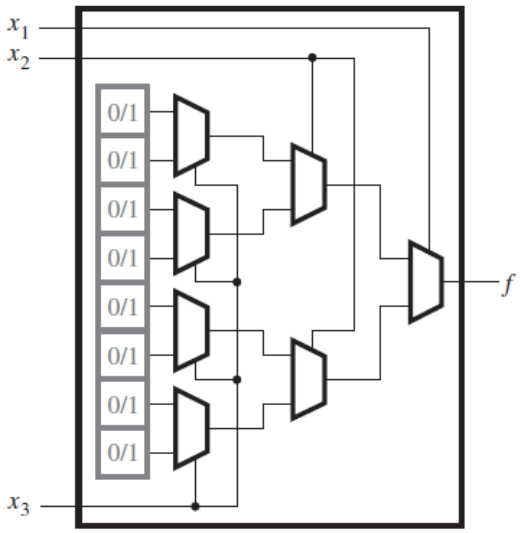

Ilustración 3: LUT de tres entradas.

Las LUTs generalmente tiene un circuito adicional en cada bloque lógico para almacenar el valor con control de ciclos de reloj. Este circuito se consigue con la adición de un flip-flop y un multiplexor quedando como en la Ilustración 4. El funcionamiento de este bloque es como el funcionamiento normal de una LUT, pero el valor de salida dependiendo de la selección estará actualizado por el ciclo de reloj de la FPGA o no.

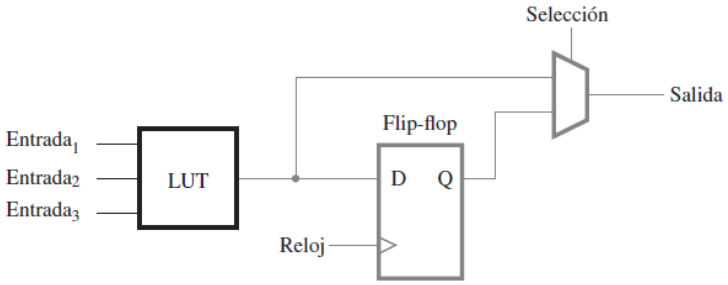

Ilustración 4: Bloque lógico de una FPGA con flip-flop.

En la parte práctica las funciones que se programan en la FPGA se sintetizarán mediante las herramientas CAD (Diseño asistido por ordenador). Las herramientas CAD generarán las conexiones entre las entradas, las salidas y las LUTs empleadas. Además, genera las tablas de la verdad de las LUTs.

Las celdas de almacenamiento de las LUTs son volátiles por lo tanto cada vez que se quite la potencia a la FPGA la información de estas será borrada. Para evitar que los valores almacenados en las LUTs se eliminen cuando se quita la potencia las FPGAs contienen memoria PROM. La función de las memorias PROM es recuperar la información de almacenamiento de las LUTs cuando se devuelva la potencia.

Resumiendo, el esquema o síntesis de como quedaría una FPGA programado con una función de tres entradas y una salida. La función lógica que se implementa en la FPGA es: $f = x_1x_2 + \overline{x_2}x_3$. En la FPGA queda de la forma que se observa en la Ilustración 5.

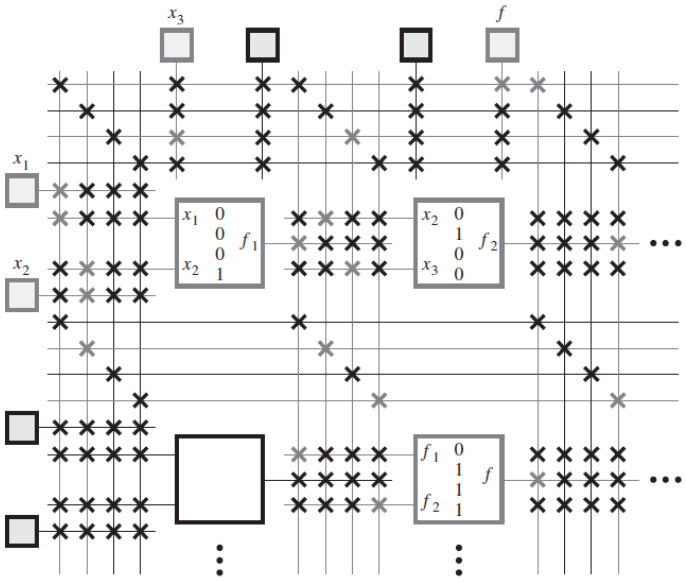

Ilustración 5: Sección de una FPGA programado.

En la Ilustración 5 se tienen varios bloques representativos de la FPGA como son los bloques de entrada y salida, los canales de interconexión y las LUTs. De los bloques que se observan solo se emplean los que se encuentran en color gris. La información que se transmite por los cables se dirige y controla mediante los interruptores que contienen los bloques de interconexión. Los interruptores en negro están apagados por lo que no realizan la interconexión entre las dos líneas. Los que se encuentran en gris sí que realizan la interconexión.

2.3 RISC-V [12]–[15]

La definición que aporta la propia organización es: “RISC-V es una nueva arquitectura de conjunto de instrucciones (ISA) que se diseñó originalmente para respaldar la investigación y la educación en arquitectura informática, pero

que ahora esperamos también se convierta en una arquitectura abierta y gratuita estándar para implementaciones industriales." [13, p. 1]

Esta arquitectura nace en mayo de 2010 en un proyecto de la Universidad de California en Berkeley. Un proyecto dirigido por el profesor Krste Asanović y los estudiantes graduados Yunsup Lee y Adrew Waterman comenzando el conjunto de instrucciones de RISC-V.

Como se menciona en la página web de la organización las principales características que tiene este procesador es que contiene un conjunto de instrucciones libre y abierta, tiene libertad de diseño en todos los dominios e industrias y la posibilidad de tener una amplia gama de niveles de rendimiento.

La arquitectura del RISC-V está teniendo un crecimiento exponencial y ganando mucha importancia en el sector de los procesadores. Las aproximaciones que tiene la organización es que en 2025 haya cerca de 80 billones de procesadores con la arquitectura RISC-V frente a los 10 billones que se estima que hay actualmente.

La arquitectura está siendo empleada en diversas industrias como son: automoción, inteligencia artificial, Internet de las cosas (IoT) y comunicaciones.

La arquitectura RISC-V tiene un diseño modular, lo que quiere decir que tiene un módulo base y a este se le pueden añadir módulos adicionales. Dependiendo de las características que deba cumplir el procesador que se va a desarrollar se deberán añadir uno u otros módulos.

En la siguiente tabla se podrá observar los distintos módulos de los que se compone el RISC-V.

- Módulos Básicos

Denominación	Descripción
RV32I	Conjunto de instrucciones de base entera, 32-bits
RV64I	Conjunto de instrucciones de base entera, 64-bits
RV128I	Conjunto de instrucciones de base entera, 128-bits

Tabla 1: Módulos básicos RISC-V.

Los módulos básicos se diferencian principalmente en la cantidad de bits con la que trabaja el procesador.

- Módulos adicionales

Denominación	Descripción
M	Extensión estándar para la multiplicación de enteros y división
Un	Extensión estándar para las instrucciones atómicas
F	Extensión estándar para coma flotante de precisión simple
D	Extensión estándar para coma flotante de precisión doble
G	Abreviatura para la base y extensiones anteriores
Q	Extensión estándar para coma flotante de precisión cuádruple
L	Extensión estándar para coma flotante decimal
C	Extensión estándar para instrucciones comprimidas
B	Extensión estándar para manipulación de bits
J	Extensión estándar para lenguajes traducidos dinámicamente
T	Extensión estándar para memoria transaccional
P	Extensión estándar para empaquetado-SIMD instrucciones
V	Extensión estándar para operaciones de vector
N	Extensión estándar para interrupciones de nivel de usuario

Tabla 2:Módulos adicionales RISV-V

La denominación G se le daría a un procesador RISC-V que contenga los módulos M, Un, F y D, ya que son los módulos de uso general.

La denominación que obtiene un RISC-V depende del módulo básico que se emplee a lo que se le añade la identificación de los módulos empleados adicionalmente.

La arquitectura básica de un RISC-V que realiza las operaciones de lógicas y aritméticas básicas es la que se muestra en la Ilustración 6.

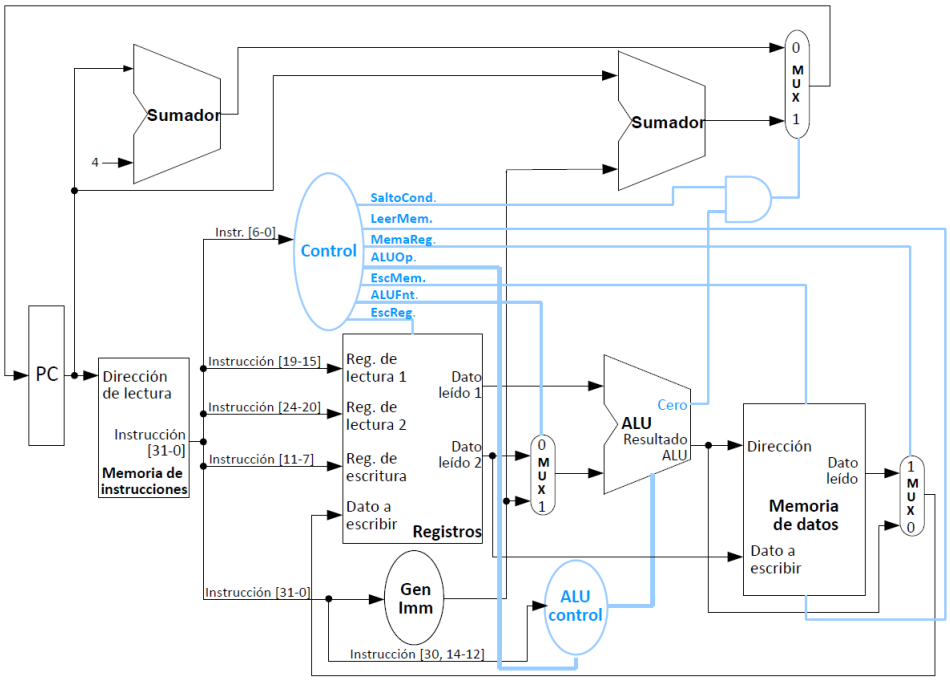

Ilustración 6: Esquema de la arquitectura básica RISC-V.

Los bloques más relevantes que se tienen en la arquitectura son: memoria de instrucciones, bloque de control, registros, memoria de datos, ALU y contador de programa.

La memoria de instrucciones contiene como dice su nombre las instrucciones que se ejecutarán en el procesador. Dependiendo del tamaño de la memoria de instrucciones se tendrá un máximo de instrucciones. Las instrucciones tienen una longitud de 32 bits.

Se tienen dos tipos de bloques que controlan el procesador y la ALU. La función del bloque de control del procesador es generar las señales que se emplean en los bloques del RISC-V. Este bloque tiene como entrada la instrucción a ejecutar que será interpretada por este bloque generando las señales necesarias para controlar el procesador.

El bloque de control de la ALU tiene como función decidir qué función de la ALU se va a ejecutar. Como entradas se tiene la instrucción y una señal de control. El bloque genera una señal de 5 bits.

El Registro es un circuito cuya función es contener los datos sobre los que se realizan operaciones y para guardar los resultados de estas operaciones. Este bloque contiene 32 registros que están destinados a varias funciones preestablecidas pero que no se obliga a utilizarlas para dicho uso. El único registro que tiene un valor constante es el z0 que está destinado al valor cero.

La memoria de datos está destinada al almacenamiento o lectura de los datos. Esta memoria tiene dos direcciones de flujo de los datos. Una será la carga de datos de esta memoria al registro y la segunda será de carga de datos desde el registro a la memoria de datos.

La ALU tiene como función realizar operaciones lógicas o aritméticas. Tiene como entradas los datos y la entrada de control de la ALU que es la que dictamina qué operación se va a realizar. Como salida se tiene el resultado obtenido de la operación realizada.

El contador de programa tiene como función llevar el orden de las instrucciones. Esto se realiza de forma que este bloque tiene dos entradas y una salida. Las entradas son el número de instrucción siguiente y la señal de reloj y la salida es el número de instrucción que se ejecuta. El cambio de numero de instrucción se da en los flancos de subida.

En el Capítulo tres (página 27) se desarrollará más específicamente cada parte del RISC-V, así como el esquema del RISC-V desarrollado con los módulos adicionales.

2.4 VHDL [16]–[18]

La definición de este concepto es: “VHDL es un lenguaje de descripción de hardware. El código describe el comportamiento o la estructura de un circuito electrónico, a partir del cual un compilador puede inferir un circuito físico compatible.” [16, p. 4]

El significado de las iniciales VHDL es lenguaje de descripción de hardware VHSIC (*Very High Speed Integrated Ciruits* o Circuitos integrados de muy alta velocidad).

Las aplicaciones más importantes que tiene VHDL son la síntesis de circuitos digitales en FPGA y CPLD (*Complex Programmable Logic Device* o dispositivo lógico programable complejo), así como la generación de diseño para la fabricación ASIC.

El lenguaje surge en el departamento de defensa estadounidense en los años 80. Está basado en el lenguaje ADA (Lenguaje de programación orientado a objetos que destaca por su gran seguridad y fiabilidad). Su finalidad es emular circuitos eléctricos digitales, simularlos e implementarlos en *hardware.* VHDL fue el primer lenguaje de descripción hardware estandarizado por el IEEE, bajo los estándares 1076 y 1164. Algunos otros lenguajes de descripción hardware similares son: *Verilog* y *ABEL*.

Los circuitos que se diseñan pueden ser circuitos combinacionales, secuenciales o ambos simultáneamente. Como todos los lenguajes de

programación a este se le pueden añadir bibliotecas o paquetes para ampliar las funcionalidades.

La estructura del lenguaje de programación tiene dos partes: Entidad (*Entity*) y Arquitectura (*Architecture*).

La entidad es la parte del código donde se definen las entradas y las salidas que tendrá el circuito y el tipo de estas (un bit, un vector, una booleana, integer, …), además a esta entidad se le deberá dar un nombre para poder ser utilizada en la "arquitectura". La definición de las entradas y salidas se hará con la siguiente estructura: $port\_name : port\_model\ signal\_type;$.

La arquitectura tiene la función de describir cómo se relacionan las entradas definidas en la entidad. La estructura que tiene la arquitectura es la siguiente:

$$ARCHITECTURE\ architecture\_name\ OF\ entity\_name\ IS$$
$$[zona\ de\ declaracion*]$$
$$BEGIN$$
$$[descripcion\ del\ circuito]$$
$$END\ architecture\_name$$

El nombre que se le da a la arquitectura no es muy importante debido a que se le puede poner cualquier nombre, pero hay una convención para ponerle nombre dependiendo del tipo de descripción que se quiera realizar. Los nombres son: *Behavioral* (Comportamental), *Structural* (estructural), *Functional* (Funcional) y *Dataflow* (Flujo de datos).

Como se observa se tiene dos partes dentro de la arquitectura separadas por la sentencia *BEGIN* que son: la zona de declaración y la de descripción del circuito. La zona de declaración es una parte del código que es opcional, pero en ella se podrán declarar varios tipos de datos. Las variables que se definen en este apartado son datos que se emplearan en el código como intermediarios entre las entradas y salidas definidas en la *ENTITY*. Los tipos de datos que se pueden definir son: *SIGNAL, COSNTANT, FILE, etc.*

En la parte de descripción del código como su nombre dice se realizará la descripción del circuito que se quiera realizar. Dentro de la *Architecture* se podrán emplear multitud de estructuras divididas en dos grupos que son: secuenciales y combinacionales. Para las descripciones secuenciales hay las siguientes estructuras: comparativas (*IF*), bucles (*FOR, WHILE, LOOP*) y de selección (*CASE, CASE … SELECT*). Todas las descripciones de código que sean secuenciales se deberán realizar dentro de un *PROCESS* que es una sentencia

que tiene una lista de sensibilidad que solo se ejecutará cuando una o varias de las variables de la lista de sensibilidad cambia de valor.

Dentro de los *PROCCES* se podrán definir variables que se utilizarán únicamente dentro de esta estructura. Este tipo de variable es similar a SIGNAL, con la excepción que no hace falta que se ejecute al completo el código para que cambie su valor.

Las estructuras combinacionales no necesitan estar dentro de ninguna otra estructura con lista de sensibilidad si no que siempre que se cumpla los ciclos el código se ejecutara. Las estructuras que están dentro de este grupo son: selección (*WHEN, SELECT*) y bucle (*GENERATE*).

Además de estas estructuras existen los operadores que realizan funciones lógicas o signos que realizan operaciones aritméticas que se pueden utilizar indistintamente. Algunos de los operandos o funciones no están definidos en las librerías básicas y en algunos casos serán necesarios añadir nuevas libreras.

Algunos de los compiladores que soportan este lenguaje son: Warp, Altera, Xilinx (empleado en este estudio), GHDL, etc.

Un ejemplo de un circuito combinacional desarrollado en VHDL se puede observar en la Ilustración 7.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity F_adder is
  Port (res, carry : out std_logic;
        a,b,c : in std_logic);
end F_adder;

architecture Behavioral of F_adder is
begin
res <= (a xor b) xor c;
carry <= (a and b) or ((a xor b) and c);
end Behavioral;
```

Ilustración 7: Código VHDL de un sumador total extraída de Vivado.

En la imagen superior se muestra el código de un Sumador Total (*Full-Adder)* y servirá como ejemplo para mostrar algunos de los conceptos explicados anteriormente.

Como se muestra en la Ilustración 7 tenemos la zona de la declaración de las librerías básicas dado que este código no necesita ninguna librería adicional. A continuación, se tiene la *ENTITY* que en este caso la definimos como "*F_adder"*, en la cual se definen las entradas y salidas (no hace falta definir primero las

entradas como se ve). Se tienen dos salidas (*res* y *carry*) y tres entradas (*a, b* y *c*).

Una vez acabada la definición de entradas y salidas se cierra la *ENTITY* y se pasa a declarar la *ARCHITECTURE* que la damos el nombre de *Behavioral* y la asociamos a la entidad definida anteriormente. En este caso no definiremos variables intermedias como podrían ser señales o constantes. Tampoco se definirá un proceso debido a que es un diseño combinacional. A continuación, se realiza la descripción del circuito. Este código describe una suma entre tres valores que se conoce como sumador total y se obtiene como consecuencia el resultado y el acarreo.

Para obtener el resultado de la suma (res) se realiza dos operaciones XOR. Esta sentencia se describe en la Ilustración 8.

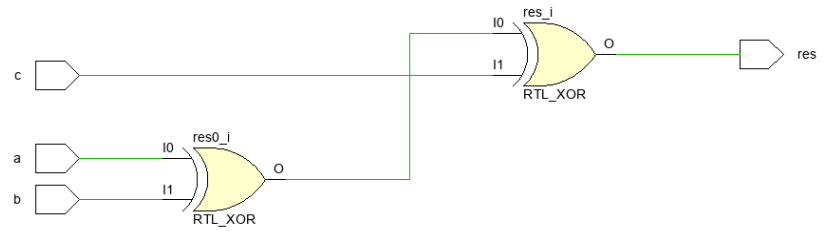

Ilustración 8: Circuito obtención de la salida res

Para el cálculo del acarreo (carry) se emplean varias operaciones lógicas de AND y XOR, la operación lógica se describe en la Ilustración 9.

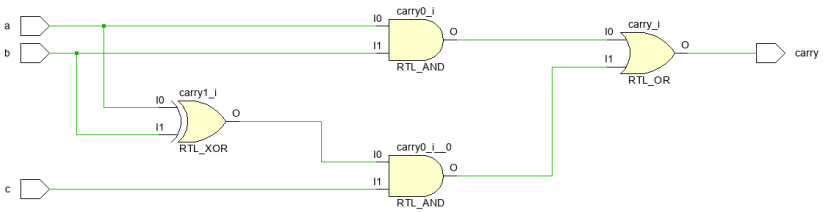

Ilustración 9: Circuito obtención de la salida carry

Por lo que el código definido en VHDL anteriormente describe el conjunto de los dos esquemas anteriores como se muestra en la Ilustración 10.

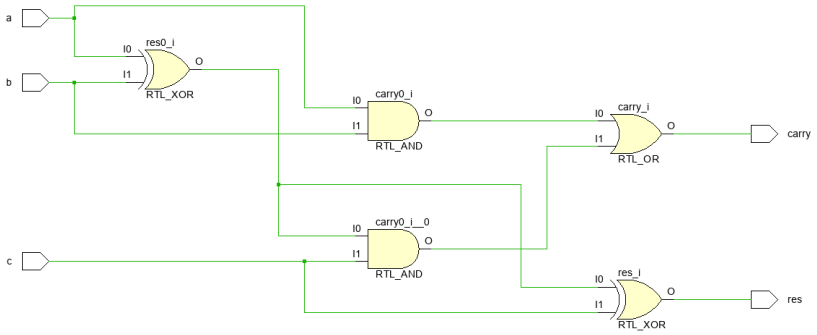

Ilustración 10: Circuito Sumador total

2.5 Vivado 2021.1, Xilinx

Vivado es un programa informático desarrollado por *Xilinx* para la síntesis, implementación y análisis de diseños HDL. Esta versión es una de las más actuales y soporta toda la gama de FPGAs desarrolladas por *Xilinx*. A diferencia de otros programas de diseño HDL, Vivado no permite el diseño de forma esquemática como si lo permitían versiones anteriores y su diseño se realiza mediante programación VHDL o verilog.

La interfaz de Vivado es la siguiente:

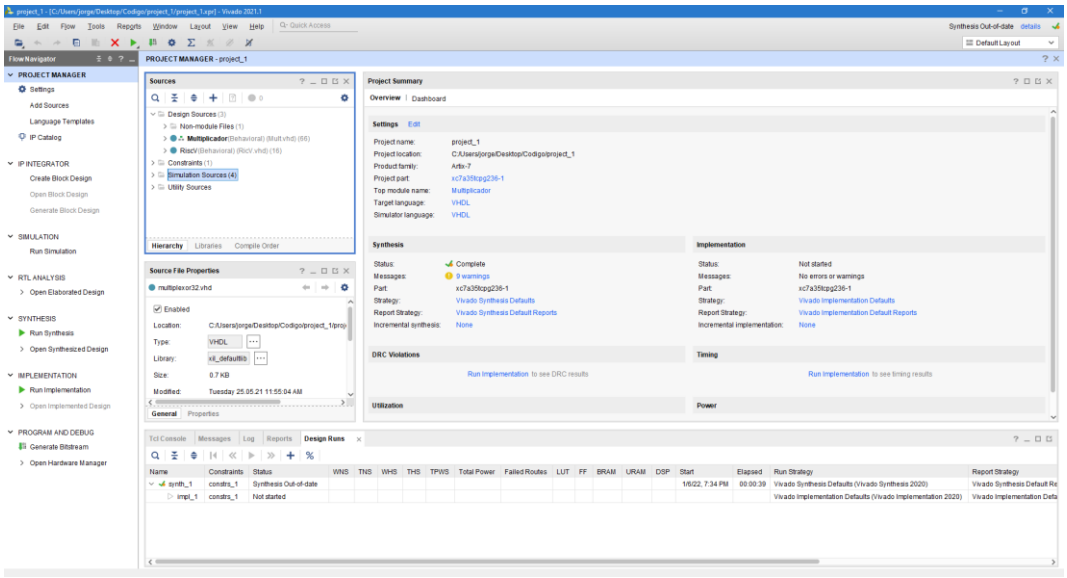

Ilustración 11:Interfaz de Vivado

A continuación, se expondrán las ventanas más importantes de la interfaz de Vivado.

- *Project manager*: En esta ventana se pueden encontrar varias selecciones entre las que destacan:
    - IP *Integrator*: es una funcionalidad que a partir de bloques dados por Xilinx realiza un diseño de la manera más eficiente.

  - *Simulation*: a partir de un diseño realiza una simulación de funcionamiento. Para la simulación aparece otra pantalla en la que se le pueden asignar los valores a las variables de entrada del sistema y otra en la que se observa el resultado de la simulación.
  - *RLT Analysis*: esta función realiza un esquema de los elementos lógicos y bloques creados en el diseño mediante la programación HDL.
  - *Synthesis*: esta función realiza un esquema del hardware empleado en el diseño, un esquema de los elementos empleados y como se conectan entre sí teniendo en cuenta el procesador empleado. En este apartado también se generará la información de elementos empleados en la FPGA, potencia y tiempo en obtener los resultados.
  - *Implementation*: muy similar a *synthesis* con la diferencia que emplea otra forma de compilar para la FPGA seleccionada y si no contiene elementos suficientes dará error.
  - *Program and debug*: es utilizado para generar el código de bits que programa la FPGA física y enviar/pasar el código a la FPGA.
- *Project summary*: Se encuentra la información técnica del procesador, el lenguaje que se emplea (VHDL o Verilog) y el estado de las funciones que se han comentado anteriormente. Proporciona toda la información que se va generando en la sintetización o en la implementación.

Capítulo 3: RISC-V [12], [13], [19]

La arquitectura RISC (*Reduce Instruction Set Computer*) es un tipo de CPU (*Central Processing Unit*) que se emplea en microprocesadores que cumple con las siguientes características fundamentales:

- Instrucciones de tamaño fijo y presentadas en un reducido número de formatos.
- Solo las instrucciones de carga y almacenamiento acceden a la memoria de datos.
- Dispone de registros de propósito general.

El objetivo de las maquinas que contienen esta arquitectura es la de habilitar la segmentación (pipeline) y paralelismo de las instrucciones y reducir el acceso a la memoria de datos. [20]

El origen de esta arquitectura surge en la década de los 70s desarrollada por el IBM (*International Business Machines Corporation*) y las universidades de Standford y Berkeley. David Patterson fue quien acuño el termino RISC. Fue John Cocke investigador del IBM quien demostró que con solo el veinte por ciento de las instrucciones que contenía un ordenador se podía cumplir el ochenta por ciento del trabajo.

Las principales características de esta arquitectura son:

- Ejecución en un único ciclo. Lo que supone que una instrucción se ejecutara en un ciclo de reloj.
- Arquitectura no destructiva de tres direcciones. Esto se debe a que la arquitectura contiene tres registros y por lo tanto no necesitará sobrescribir uno de los registros que contenía operandos. Esto implica que tendrá dos registros de operandos y uno de resultado.
- Las instrucciones son simples, con un formato fijo y con pocos modos de direccionamiento a la memoria de datos. Este formato de instrucciones favorece la decodificación y la ejecución segmentada de las instrucciones. Así como que tenga un tamaño y formato fijo hace que los campos que contienen las instrucciones se encuentren en la misma posición.
- Tiene un diseño de registro a registro. Consecuentemente las operaciones que no sean de carga o almacenamiento no emplearán la memoria de datos. Las demás operaciones utilizaran banco de registro.

Las características expuestas suponen algunas de las ventajas de esta arquitectura sobre otras. Algunas de las ventajas son: mejor rendimiento al limitado número de instrucciones, menor cantidad de hardware físico, menor consumo de energía y un amplio uso en diferentes aplicaciones.

La arquitectura RISC original o RISC-I era una versión de 32 bits que operaba a 0.5 MHz y empleaba 44500 transistores todo esto se consiguió con una densidad de código 30% mayor que VAX. [21]

El diseño RISC-II desarrollado por *Sun Microsystems* en 1983 fue el que demostró que todos los beneficios de estos procesadores eran reales. Esta arquitectura consiguió mejorar la velocidad, obteniendo un periodo seis veces menor, llegando a los 3MHz. En esta versión se añadieron registros alcanzando los veintidós. Esta arquitectura logró reducir los transistores empleados en aproximadamente cuatro mil.[21]

En 1984 se realizó un rediseño y una optimización para la compañía de Smalltalk en la que se consiguió reducir los transistores empleados en cinco mil unidades. Esta versión se denominó SOAR o RISC-III es un simple RISC que tiene arquitectura Von Neumann. Esta arquitectura está diseñada para ejecutar el sistema Smalltalk-80. El sistema Smalltalk es un entorno de programación altamente productivo que plantea escritura dinámica de datos, conjunto de instrucciones de alto nivel y llamadas a procedimientos frecuentes y administra almacenamiento orientado a objetos. [21], [22]

La siguiente versión fue RISC-IV o VLSI-BAM fue un rediseño realizado en 1990 que optimizaba el código desde Prolog (lenguaje de programación lógico e interpretado). Esta arquitectura aumento el uso de los transistores empleando 11200, llegaba a una frecuencia de 20 MHz con un consumo de solo un watio.[21]

Se continuaron las investigaciones sobre la arquitectura RISC y se llegó al desarrollo de RISC-V que fue desarrollado en 2010 en la Universidad de California en Berkeley. En la investigación del RISC-V participó uno de los impulsores de la arquitectura RISC como es David Patterson. Otro académico que participo en la investigación fue Andrew Waterman.

En la actualidad es uno de los procesadores más empleados llegando en el año 2021 a más de dos millones de núcleos implementados. En estandarización también se realizaron avances produciendo una especificación en 2020 y dieciséis en 2021.

Las ventajas que tienen los procesadores RISC-V sobre los demás son [23]:

- Ofrecer un mejor rendimiento, dado el número reducido y limitado de instrucciones que soporta.
- Menos cantidad de soporte físico (menos hardware).
- Permite que el uso libre de del encapsulado para integrar otros circuitos.
- Como consecuencia de menor uso de hardware tendrá un menor consumo y generación de calor.

Son muy polivalentes y se pueden emplear en cualquier tipo de aplicaciones. Puede emplearse desde móviles de gama baja hasta en aplicaciones de seguridad o en supercomputadores.

En este estudio se ha realizado la programación de la arquitectura del RISC-V de 32 bits de computación de números enteros y con el módulo adicional estándar para la multiplicación de enteros y la división (RV32IM) se ha

desarrollado en el lenguaje VHDL mediante la aplicación Vivado para la programación en la FPGA.

El RISC-V que se ha realizado tiene una arquitectura de 32 bits, contiene la extensión estándar para realización de multiplicaciones y es un procesador monociclo. Que sea monociclo supone que la obtención, decodificación, ejecución y almacenamiento de la instrucción se realiza en un ciclo de reloj.

Se comenzará analizando como se ha realizado la escritura de la arquitectura RISCV sin entrar en la programación que se examinará una vez expuestos los bloques que contienen la arquitectura.

3.1 Arquitectura

La arquitectura del RISC-V compuesta por las instrucciones básicas (mayoría de las instrucciones que se observan en la Ilustración 14) que la componen es la que se observa en la Ilustración 12. La arquitectura del RISC-V se diseña a partir del conjunto de instrucciones que contenga el procesador. Para incluir instrucciones que faltan a la arquitectura se le han realizado algunas modificaciones a la arquitectura.

La arquitectura se dividirá en varias partes para explicar el funcionamiento de cada bloque y como se relacionan. Los bloques más importantes que se han desarrollado para esta estructura son: Control del procesador, Control de la ALU, Contador de programa, Memoria de Instrucciones y de datos, Registro y ALU. Se analizarán las características y diseño de estos bloques.

Para entender el funcionamiento de los bloques hay que conocer las instrucciones y señales que relacionan los bloques. Se comenzará analizando las instrucciones.

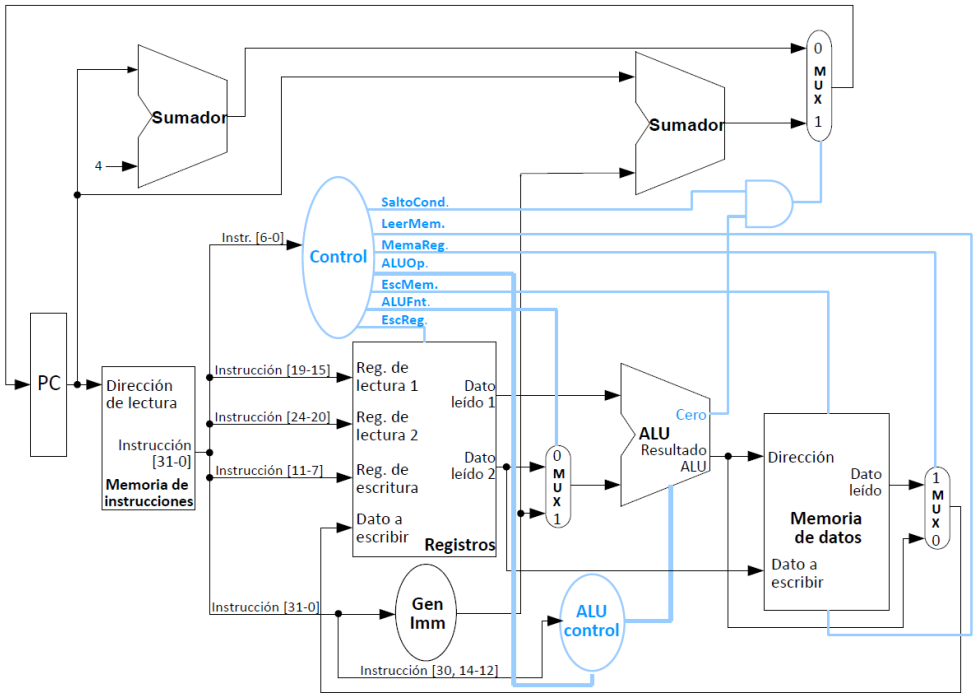

Ilustración 12: Arquitectura básica RISC-V 32 bits

3.1.1 Instrucciones y sus tipos

La principal función de las instrucciones es controlar los bloques y los multiplexores. Hay seis tipos de instrucción en la arquitectura básica. En la arquitectura desarrollada en este trabajo se tendrán seis tipos de instrucción. Los cuales son: Tipo R, Tipo I, Tipo S, Tipo B, Tipo J, Tipo U.

Las instrucciones Tipo R realizan operaciones básicas aritméticas y lógicas entre los registros fuente, dentro de este grupo se contienen las instrucciones del módulo base y las de la extensión de la multiplicación (en este estudio se conocerán como tipo M). Las instrucciones Tipo I tienen varias funcionalidades como son realizar operaciones lógicas y aritméticas entre un registro fuente y un valor inmediato, operaciones de carga de registros en la memoria de datos (en este estudio se conocerán como Tipo L, aunque tiene estructura tipo I) y la instrucción LUI (carga un inmediato en un registro destino). Las instrucciones tipo M realiza operaciones de multiplicación. La instrucción tipo U realiza una operación de suma. Las instrucciones Tipo B y J son para realizar saltos condicionales o incondicionales respectivamente. Las instrucciones S cargan datos en la memoria de datos.

La estructura de los seis tipos de instrucciones básicas se observa en la Ilustración 13.

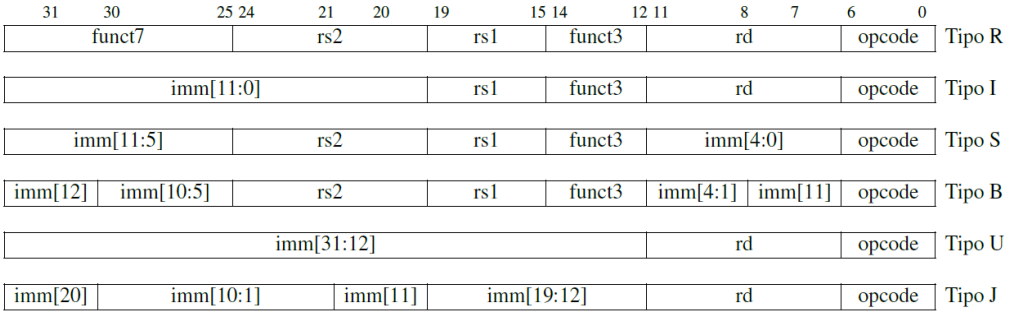

Ilustración 13: Formatos de instrucciones del RISC-V 32 bits [24, Fig. 2.2].

Se identifican varios campos que se repiten en las instrucciones como son *opcode*, *funct7, funct3, rs1, rs2, rd* y *inm[x:y].*

Estos campos contienen la siguiente información:

Opcode Contiene la identificación del tipo de instrucción. El tamaño de esta instrucción es de 7 bits. Este campo se ubica en los 7 de bits de menor peso de la instrucción (del sexto bit al cero)

Funct3 es un identificador de operación. Tiene un tamaño de 3 bits. En las instrucciones que se utiliza este campo emplea desde el decimocuarto bit al decimosegundo.

Funct7 es un identificador de operaciones tipo R. Tiene un tamaño de 7 bits. Se ubica en los siete bits de mayor peso de la instrucción (del bit treinta y uno al veinticinco).

rs2, rs1 y *rd* son los registros empleados para realizar las operaciones. Los dos primeros corresponden a los registros fuente empleados en las operaciones y el tercero corresponde al registro destino. La dirección de registro *rs1* se ubica entre el bit decimonoveno y el decimoquinto, *rs2* entre el vigesimocuarto y el vigésimo bit y el registro *rd* se encuentra entre e bit onceavo y séptimo.

inm[x:y] contiene los bits para generar el inmediato. Los valores *x* e *y* son las posiciones de la instrucción que le corresponden en el número generado. Dependiendo del tipo de instrucción se tendrán una o varias secciones por instrucción. El tamaño de esta sección es variable dependiendo del tipo de instrucción.

En la arquitectura básica de RISC-V los nombres de las funciones que se pueden programar son las que se observan en la Ilustración 14. La parte subrayada en la Ilustración 14 es la parte que se emplea para denominar las funciones.

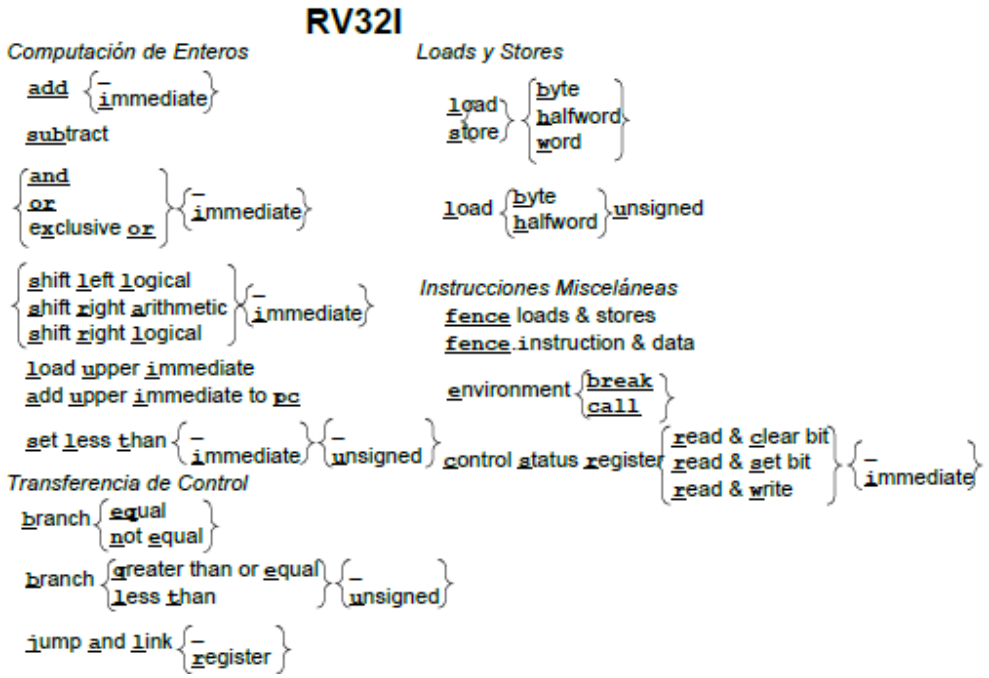

Ilustración 14: Diagrama de Instrucciones en el módulo básico [24, Fig. 2.1].

En el conjunto de instrucciones básicas se tienen las instrucciones de computación de enteros que son suma, resta, *and* lógico*, or* lógico*, xor* lógico, desplazamientos a la derecha o a la izquierda lógicos y desplazamiento a la derecha aritmético. Estas operaciones se pueden realizar entre números enteros que se encuentran almacenados en el banco de registros o entre uno del registro y un inmediato generado. Instrucciones de carga y almacenamiento en la memoria de datos de un resultado completo, de la mitad o de un byte. Instrucciones de transferencia de control que son empleadas para realizar saltos condicionales (instrucciones comienzan por 'b') o incondicionales (para implementar estas operaciones se realiza una modificación en la arquitectura del RISC-V).

Para introducir la funcionalidad de las instrucciones Tipo JAL (J), Tipo B, Tipo S o Tipo L se han tenido que hacer unas modificaciones en la arquitectura del original del RISC-V que se observa en la Ilustración 12. Las modificaciones realizadas para la implementación de todas las instrucciones han sido la adiccion de tres multiplexores, la adiccion de tres señales de contr y la modificacion del comportamiento de algunos de los bloques. La arquitectura RISC-V que se muestra en la Ilustración 15 puede implementar todas las instrucciones explicadas en el estudio.

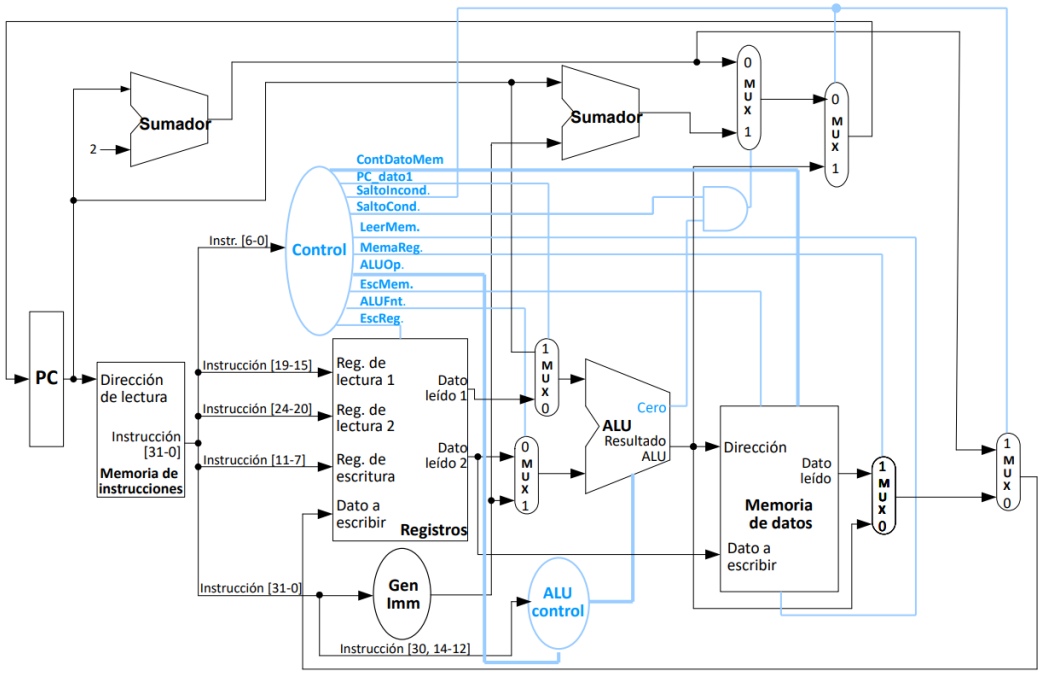

Ilustración 15: Arquitectura RISC-V modificada.

Además, se han introducido algunas señales para controlar los nuevos multiplexores y el comportamiento de los datos en los distintos bloques. Los multiplexores son empleados para controlar el flujo de la información en la arquitectura. Las nuevas señales controlan cómo se comporta la memoria de datos o nuevas operaciones en la ALU, por ejemplo.

Las señales que se han añadido son la señal de salto incondicional (SaltoIncod) empleada en multiplexores, PC_dato1 que se usa para control de un multiplexor y una señal control sobre datos de la memoria (ContDatoMem) tiene 3 bits y se usa para almacenar o leer la parte menos significativa de los datos en la memoria.

2.5.1 Control del procesador

Este bloque genera las señales de control de la arquitectura RISC-V para que se realicen las instrucciones correctamente. El control del procesador contiene una entrada que será la instrucción por ejecutar. Como salidas se tendrán todas las señales que se analizan a continuación:

- AluF, empleada para seleccionar entre el dato2 que aporta el registro o el dato generado por el generador de inmediato.
- EscR, señal empleada para habilitar la escritura en el registro.
- EscM, señal empleada para habilitar la escritura en la memoria.
- MemR, señal empleada para la selección entre el resultado de la ALU y el dato leído de la memoria.
- LeerM, señal empleada para habilitar la lectura de la memoria.
- SaltoC, señal empleada para habilitar el salto condicional.

- SaltoInc, señal empleada para seleccionar entre el valor obtenido de la señal MemR y la dirección de la siguiente instrucción, esta señal también es empleada en la selección entre los 10 bits de menos peso del resultado de la ALU y el número de instrucción que se ejecutaría a continuación. Esta señal controla el salto incondicional.
- Pc_dato1, señal empleada en la selección de la dirección de instrucciones actual o el dato1 aportado por el registro.
- AluO, señal empleada en el bloque de control de la ALU para la selección de la función que debe desempeñar el procesador con esa instrucción.
- ContDatoMem, esta señal se emplea juntamente con las señales EscM (escibe) o LeerM (lee). Es usada para cualquiera de las dos acciones guardar o cargar los cuatro bytes (sw o lw), dos bytes (sh o lh) o un byte (sb o lb), siempre los menos significativos.

  Para las instrucciones Tipo I que pertenecen a las instrucciones de carga o llamadas Tipo L que no almacenan el resultado completo se pueden diferenciar entre dos tipos de almacenamiento con signo (se extiende con respecto el ultimo bit conocido) o sin signo (se extiende con ceros). Cargar con signo o sin signo depende del bit más significativo de la señal.

La codificación para obtener el valor de las señales se obtiene de la identificación de los bits seis al dos de la instrucción que se esté procesando. Y la tabla de la verdad que se logra para la obtención de las señales es la que se observa en la Tabla 3.

Tipo	Codificación	AluF	EscR	EscM	MemR	LeerM	SaltoC	SaltoInc	AluO	PC_ dato1
R/M	01100	0	1	0	0	0	0	0	0110	0
I	00100	1	1	0	0	0	0	0	0010	0
L	00000	1	1	0	1	1	0	0	0000	0
S	01000	1	0	1	X	0	0	0	0100	0
B	11000	0	0	0	X	0	1	0	1100	0
JAL	11011	1	1	0	0	X	X	1	1101	1
JALR	11001	1	1	0	0	X	X	1	1101	0
AUIPC	00101	1	1	0	0	0	0	0	0011	1
LUI	01101	1	1	0	0	0	0	0	0111	0

Tabla 3: Información de control del RISC-V

La tabla se ha generado de forma que se conoce el valor que debe tener cada señal para que el flujo de valores en la arquitectura sea la correcta dependiendo del tipo de instrucción. Las funciones lógicas se han obtenido mediante la utilización de los bits del campo de *opcode* (*op(x)* siendo x posición del bit utilizado). Las funciones lógicas son las siguientes:

- AluF → $OP(2) + \overline{OP(6) + OP(5)} + \overline{OP(6) + OP(4)}$
- EscR → $\overline{OP(6) + OP(5)} + OP(4)$
- EscM → $\overline{OP(6)}OP(5)\overline{OP(4)}$
- MenR → $\overline{OP(6) + OP(5) + OP(4)}$

- LeerM → $\overline{OP(6) + OP(5) + OP(4)}$
- SaltoC → $OP(6)$
- AluO → $OP(6)OP(5)OP(4)OP(2)$
- SaltoInc → $OP(6)OP(2)$
- PC_dato1 → $OP(3) + OP(2)\overline{OP(5)}$
- ContDatoMem → $funct3$

Este bloque en el código realizado se le denomina *control* y genera las señales de control analizadas en este apartado. En el código no se ha distinguido entre los campos que componen la instrucción, si no que se emplea la posición de los bits en la instrucción. Sabiendo que el campo *opcode* corresponde con los siete bits de menor peso. Se puede decir que la ubicación de los bits *OP(x)* corresponden con los de *instrucción(x)*. Y el campo *funct3* corresponde con los bits ubicados entre la posición catorce y la doce de la instrucción. El circuito descrito para control se observa en la Ilustración 16. Los nombres de las señales son similares a los nombres empleados a la explicación desarrollada.

```
AluFnt<=instruccion(2)or not(instruccion(6) or instruccion(5)) or not(instruccion(6) or instruccion(4));
EscReg<=not(instruccion(6) or instruccion(5)) or instruccion(4);
EscMem<=not(instruccion(6) or instruccion(4)) and instruccion(5);
MemReg<=not(instruccion(6) or instruccion(4) or instruccion(5));
LeerMem<=not(instruccion(6) or instruccion(4) or instruccion(5));
SaltoCond<=instruccion(6);
AluOp<=instruccion(6) & instruccion(5) & instruccion(4) & instruccion(2);
SaltoIncond<=instruccion(6) and instruccion(2);
EscDato<=instruccion(6) and instruccion(2);
PC_dato1 <=instruccion(6) and instruccion(3);
ContDatoMem <=instruccion(14 downto 12);
```

Ilustración 16: Descripción del circuito de control.

Como se puede observar se han realizado las funciones mediante los operadores lógicos: *and, or, not.*

3.1.2 Contador de programa

Este bloque tiene tres entradas y una salida. Las entradas son la dirección de instrucción a ejecutar, la señal de reloj y la señal de reset. La salida será la dirección de instrucción que se ejecutará. La señal de la dirección tiene un tamaño de 10 bits (que es el empleado para la dirección de las instrucciones).

La función del contador de programa es mantener el valor que tiene en la salida hasta que se dé un ciclo de reloj que cambiará el valor por el que se tenga en la entrada en dicho instante, además en este caso tiene entrada de “reset”.

Por lo tanto, se mantendrá el valor de la salida hasta un nuevo ciclo de reloj que actualizará el valor de salida con el valor de entrada del bloque. En caso de que la entrada de *reset* se encuentre a nivel alto (‘1’) el valor de la salida será cero independientemente de las demás entradas.

3.1.3 Control del ALU

Su función obtener la operación que se va a realizar en la ALU mediante las entradas y generar una codificación que indique la operación a la ALU. Este

bloque tiene dos entradas que son la instrucción y la codificación de identificación del tipo de instrucción (*AluO*).

La salida generada es de 5 bits y se llama *AluP*. Para obtener la salida se deben tener en cuenta los campos de la instrucción *opcode, funct3* y *funct7*. Para el cálculo de la codificación se tendrán en cuenta algunos de los bits de los campos mencionados.

La codificación que se ha realizado se muestra con las siguientes formulas (*F3(x)* representa a los bits del campo *funct3, F7(x)* representa a los bits del campo *funct7*):

Tipo R → $F3 \ \& \ F7(5) \ \& \ F7(0)$

Tipo I → $F3 \ \& \ \text{“}00\text{”}$

Tipo LUI → $\text{“}11111\text{”}$

Tipo ROL y ROL → $\text{“}100\text{”} \ \& \ F3(2) \ \& \ [F3(2) \ and \ F3(0)]$

Tipo B → $F3(0) \ \& \ [F3(2) \ or \ F3(1)] \ \& \ [F3(2) \ and \ F3(1)] \ \& \ not[F3(2)] \ \& \ '1'$

En las fórmulas codificadas anteriores cuando no se indican el bit del campo es porque es el campo entero y el símbolo “&” indica la concatenación. Por ejemplo en las tipo R se concatena el campo *funct3* con el quinto bit del campo *funct7* y el bit cero del mismo campo.

Para las instrucciones del tipo L y S el valor que toma la salida de este bloque es: “00000”, esto se debe a que estas funciones realizan la operación de suma en la ALU y como resultado se obtiene el valor de la dirección de la memoria de datos.

La codificación y la función empleada se puede observar en la Tabla 4.

El código desarrollado en VHDL para el estudio se puede observar en la Ilustración 17.

```
if(ALUop = TR) then
    if(funct7(0) = '1') then --multiplicacion TM
        ALU <=  (funct3(2) and funct3(1)) &  (funct3(2) and not(funct3(1))) & ((not(funct3(2)) and fun
    elsif(funct7(4) = '1') then -- ROR ROL
        ALU <= "100" & funct3(2)& (funct3(2) and funct3(0));
    else -- originales R
        ALU <= funct3 & funct7(5) & funct7(0);
    end if;
elsif(ALUop = TIn) then -- tipo I
    ALU <= funct3 & "00";
elsif(ALUop = TB) then -- Tipo B
    ALU <= funct3(0) & (funct3(2) or  funct3(1)) & (funct3(2) and funct3(1)) & not(funct3(2)) & '1';
elsif(ALUop = TS or ALUop = TL) then -- Tipo L o S
    ALU <= "00000";
elsif(ALUop = LUI) then --LUI
    ALU <= "11111";
elsif(ALUop = JAL) then --Tipo J
    if(instruccion(3)='1') then  --JALR
          ALU <= "11011";
    elsif(instruccion(3)='0') then --JAL
          ALU <= "11010";
    end if;
end if;
```

Ilustración 17: Circuito desarrollado para el control de la ALU

El circuito descrito que se observa se compone de condiciones. Cada condición representa un tipo de instrucción. Hay dos casos que dentro de cada condición se tiene más condiciones y eso se debe a que los bits que corresponde a la identificación son iguales. El primer caso corresponde con las instrucciones Tipo R, dentro de este estado se encuentran tres más. Las cuales son: instrucciones denominadas Tipo M (se diferencian con el bit cero del campo *funct7,* dado que en este tipo toma el valor uno), las instrucciones Tipo R propiamente dichas y las instrucciones de rotación ROR y ROL (se diferencian porque el bit cuatro del campo *funct7* toma el valor uno).

El segundo caso se corresponde con las instrucciones Tipo J, las cuales tienen la misma identificación y se tiene que comparar el tercer bit de la instrucción para diferenciar entre JALR (instrucción(3)=1) y JAL (instrucción(3)=1).

3.1.4 ALU

Se encarga de realizar las operaciones que se le indican a través de la entrada *AluP*. Las operaciones que realiza pueden ser lógicas (*and, or, slt, etc.*) o aritméticas (*sumas, restas, multiplicaciones, etc.*). Este bloque tiene dos entradas para datos, una entrada de control que es AluP (salida del bloque de control de la ALU) y tiene una salida de datos de 32 bits y una de control "zero" se activa cuando el resultado de una operación es igual a cero (tiene importancia en las instrucciones Tipo B).

La codificación asignada a las operaciones realizadas en la ALU se puede observar en la Tabla 4.

Codificación	Función	Codificación	Función
00000	Suma	10000	Xor
00001	**mul**	10001	ROL
00010	Resta	10010	ROR
00011	Beq	10011	Bne
00100	SLL	10100	SRL
00101	**mulh**	10101	
00110	**mulhsu**	10110	SRA
00111	**mulhu**	10111	
01000	STL	11000	Or
01001	**div**	11001	Bge
01010	**divu**	11010	JAL
01011	BLT	11011	JALR
01100	SLTU	11100	And
01101	bltu	11101	Bgeu
01110	**rem**	11110	
01111	**remu**	11111	LUI

Tabla 4: Codificación de las operaciones de la ALU con su correspondiente función.

Las instrucciones básicas que contiene la ALU son las instrucciones que no se encuentran en negrita en la Tabla 4. Las instrucciones que se encuentran en

negrita son las instrucciones que se añaden con la extensión de multiplicación (extensión M) que se explicará a continuación.

El código desarrollado para la ALU ha sido mediante condiciones que cumplan con la codificación que se observa en la Tabla 4 y se desarrolla la operación que se indica. En la Ilustración 18 se puede observar un pequeño recorte de como se ha programado.

```
if(ALU_control=suma) then
    Result := Dato1 + Dato2;

    if(Result="00000000000000000000000000000000") then
        zero <= '0';
    else
        zero <= '1';
    end if;

elsif (ALU_control=resta) then
    Result:= Dato1 - Dato2;
    if(Result="00000000000000000000000000000000") then
        zero <= '0';
    else
        zero <= '1';
    end if;

elsif (ALU_control=op_and) then
    Result:= Dato1 and Dato2;
    if(Result="00000000000000000000000000000000") then
        zero <= '0';
    else
        zero <= '1';
    end if;

elsif (ALU_control=op_or) then
    Result:= Dato1 or Dato2;
    if(Result="00000000000000000000000000000000") then
        zero <= '0';
    else
        zero <= '1';
    end if;
```

Ilustración 18: Recorte programación de la ALU.

Como se puede examinar la condición no se realiza entre la señal *ALU_control* y la codificación que se muestra en la Tabla 4. Esto se debe a que se ha realizado la asignación de cada codificación a su identificador como se muestra en la Ilustración 19.

```
--Operaciones matematicas
constant Suma : std_logic_vector(4 downto 0):="00000";-- Suma
constant Resta : std_logic_vector(4 downto 0):="00010";-- Resta
--Operaciones logicas
constant op_and : std_logic_vector(4 downto 0):="11100";-- Operacion AND
constant op_or : std_logic_vector(4 downto 0):="11000";-- Operacion OR
constant op_XOR : std_logic_vector(4 downto 0):="10000";-- Operacion XOR
```

Ilustración 19: Recorte definición de los identificadores de la codificación.

Continuando con la explicación del código de la Ilustración 18, se ve que dentro de los bloques de la condición se realiza la operación que se indica y si la variable *result* es cero se le asigna a la señal *zero* el valor cero si no se le asigna el valor uno.

3.1.5 Registro

La arquitectura RISCV contiene 31 registros que están destinados para diversas funciones como: argumentos de función, datos temporales, punteros globales, de cabeceras o de hilos, datos temporales o registros guardados, y un registro que está destinado al valor 0.

Para obtener el valor cero en el registro z0 (x0 en la Tabla 3) se conecta físicamente a una conexión de tierra de la FPGA.

Este bloque tiene cuatro entradas o seis dependiendo si se dividen en la instrucción la dirección del dato 1, 2 y la dirección de escritura, además de estas entradas tiene una señal de control de escritura (EscReg), una señal de reloj para la escritura síncrona y el dato a escribir. El bloque tiene dos salidas que serán los datos leídos.

La longitud de estos registros es de 32 bits. La organización de los treinta y dos registros es la que se observa en la Tabla 5.

Registro	Uso del Registro	Registro	Uso del Registro
x0	Cero	x16	Argumento de función
x1	Dirección Retorno	x17	Argumento de función
x2	Puntero de pila	x18	Registro guardado
x3	Puntero Global	x19	Registro guardado
x4	Puntero de hilo	x20	Registro guardado
x5	Temporal	x21	Registro guardado
x6	Temporal	x22	Registro guardado
x7	Temporal	x23	Registro guardado
x8	Registro guardado, Puntero de frame	x24	Registro guardado
x9	Registro guardado	x25	Registro guardado
x10	Argumento de función	x26	Registro guardado
x11	Argumento de función	x27	Registro guardado
x12	Argumento de función	x28	Temporal
x13	Argumento de función	x29	Temporal
x14	Argumento de función	x30	Temporal
x15	Argumento de función	x31	Temporal

Tabla 5: Organización de los 32 registros del banco de registros. [12, Cap. 2.3]

El código desarrollado para este bloque se corresponde con la Ilustración 20.

Se declara un proceso para detectar la señal de flanco de subida del reloj (*clk'event and clk=1*) y en caso de que esté habilitada la escritura (*hw='1'*, esta señal se vincula con en el RISC-V con la señal *EscR*) se le dará dirección del registro de entrada a la matriz, excepto cuando la dirección de escritura del registro es cero que no se habilita. En la situación de que la dirección no sea cero la salida será la matriz direccionada (dirección de entrada) y se le asignara el *dato_escritura* (lo que se quiere escribir).

```
process(clk, instruccion)
begin
    if (clk'event and clk ='1')then
        if(hw='1') then
                if(Dir_escritura /= "00000") then
                    men_registros(conv_integer(Dir_escritura))<= dato_escritura;
                end if;
        end if;
    end if;
end process;

Dir_lectura1 <= instruccion(19 DOWNTO 15);
Dir_lectura2 <= instruccion(24 DOWNTO 20);
Dir_escritura <= instruccion(11 DOWNTO 7);
Dato_s1 <= men_registros(conv_integer(Dir_lectura1));
Dato_s2 <= men_registros(conv_integer(Dir_lectura2));
```

Ilustración 20: Descripción del circuito del registro.

Para la lectura no se necesita un proceso, si no que directamente se obtienen las direcciones de los registros de lectura (Dir_lectura1 y Dir_lectura2). Una vez se tienen las direcciones se le da a la matriz de registros la dirección de los registros que se quieren usar y se extraen en las señales Dato_s1 y Dato_s2.

3.1.6 Memoria

La memoria tiene una arquitectura tipo Harvard lo que quiere decir que la memoria está dividida en memoria de datos y memoria de instrucciones.

3.1.6.1 Memoria de datos

La memoria de datos se emplea para almacenar o extraer valores de 4 bytes que pueden ser resultados, registros punteros o valores que no se vayan a emplear a corto plazo. Este bloque tiene cinco entradas y una salida. La salida será el valor extraído de una de las direcciones de memoria. De las entradas tres son de control y la restante será la dirección del valor a cargar o descargar. Una característica de la memoria es que es una memoria alineada, lo que significa que no se emplearán direcciones impares para el almacenamiento o lectura de palabras (4 bytes) o medias palabras (2 bytes). Las tres entradas de control son: la señal de habilitación de lectura (*hr*), la señal de habilitación de escritura (*hw*) y la señal de control *ContDatoMem* que indicará si se leen o cargan uno, dos o los cuatro bytes.

La señal de la dirección tiene un tamaño de 12 bits de los cuales diez corresponden a la dirección y dos se refieren a cada uno de los cuatro bytes de la dirección. Por lo que solo se emplearán los dos bits cuando se trabaje con un solo byte. En caso de que se use media palabra se empleara el de mayor peso y en caso de que se trabaje con los cuatro bytes se obviarán los dos bits.

Las posiciones que pueden ocupar los datos dependiendo de la señal de control *ContDatoMem* y dependiendo de los dos bits de menos peso de la dirección son las que se observan en la Ilustración 21.

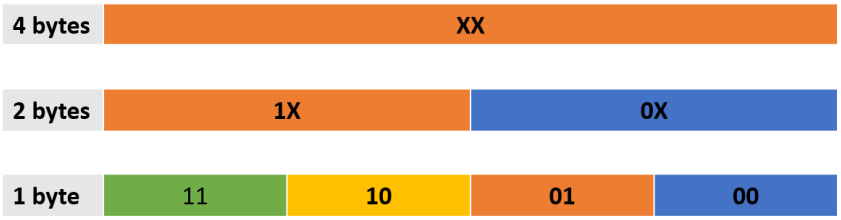

Ilustración 21: Diagrama de las posiciones que ocupan los bytes dependiendo de la dirección y la cantidad de bytes con la que se trabaje.

En la Ilustración 21 se tiene a la izquierda la cantidad de bytes que se emplean y representada en colores y con dos bits se tiene las posiciones que puede ocupar los bytes.

La memoria puede almacenar o sacar datos y se procederá de distinta forma.

En el caso de almacenar solo se habrá tres combinaciones útiles de la señal de ContDatoMem que serán “000”, “001” y “010”. La primera almacenará un byte, la segunda dos bytes y la tercera los tres bytes. En el momento de guardar uno o dos bytes en una dirección las demás posiciones de esa dirección no se verán afectadas. Por lo tanto, solo se modificará la indicada por los dos bits de menor peso de la dirección.

Para almacenar en la memoria de datos se debe tener la señal de escritura a nivel alto y un flanco de subida de la señal de reloj. En ese instante se realizará la escritura.

Por ejemplo, si queremos almacenar dos bytes en la dirección “011100111010” la dirección de la memoria será la “0111001110” y la posición en esa dirección es la “10”. Observando la Ilustración 21 se ve que el dato se almacenará en los dos bytes de la parte superior dejando los dos de la parte inferior con su valor sin modificar. Esto se observa en la Ilustración 22.

Valor a almacenar =x“3458EFDA”, ContDatoMem=“001”, pos=“10”		
Dirección	**1X**	**0X**
	Valor inicial	
0111001110	**8A9E**	**F343**
	Valor después del flanco de subida del Reloj	
0111001110	**EFDA**	**F343**

Ilustración 22: Ejemplo de cómo se almacenarán dos bytes en una dirección de memoria dependiendo de la posición.

El código que se ha empleado para realizar el circuito de esta parte de memoria ha sido el que se observa en la Ilustración 23.

```
if clk'event and clk ='1' then
    if hw='1' then
        if ContDatoMem = "000" then
            if posicion="00" then
                men_datos(conv_integer(Direcion))(7 downto 0)<= Dato_escritura(7 downto 0);
            elsif posicion="01" then
                men_datos(conv_integer(Direcion))(15 downto 8)<= Dato_escritura(7 downto 0);
            elsif posicion="10" then
                men_datos(conv_integer(Direcion))(23 downto 16)<= Dato_escritura(7 downto 0);
            else
                men_datos(conv_integer(Direcion))(31 downto 24)<=  Dato_escritura(7 downto 0);
            end if;

        elsif ContDatoMem = "001" then
            if posicion="00" then
                men_datos(conv_integer(Direcion))(15 downto 0)<= Dato_escritura(15 downto 0);
            elsif posicion="10" then
                men_datos(conv_integer(Direcion))(31 downto 16)<= Dato_escritura(15 downto 0);
            end if;
        elsif ContDatoMem = "010" then
            men_datos(conv_integer(Direcion))<= Dato_escritura;
        end if;
    end if;
end if;
```

Ilustración 23: Recorte código empleado en la escritura de la memoria de datos.

En el caso de extraer datos de la memoria se puede actuar de cinco formas dependiendo de *ContDatoMem* que además de las combinaciones que tiene en el caso de almacenar tendrá también las combinaciones posibles de “100” y “101”. Los dos bits de menos peso tendrán la misma función que en el caso de almacenar, como se explica en la Ilustración 21.

En esta situación se tendrán dos combinaciones de *ContDatoMem* para extraer un byte y dos bytes y solo una para la palabra completa. Las dos combinaciones se diferencian en que en las combinaciones “000” (un byte) y “001” (dos bytes) se extenderá el último bit hasta completar los 32 bits. Y en las combinaciones “100” (un byte) y “101” (dos bytes) se completará con ceros. Cada combinación corresponde con una instrucción diferente. Para un byte o dos bytes se trata el dato con signo o sin signo dependiendo de la combinación empleada.

Por ejemplo, si se quiere extraer un solo byte y el bit de mayor peso tiene el valor ‘1’ dependiendo de la señal de control *ContDatoMem* se tendrá un valor u otro. En el caso que se tenga la señal de control “000” se extenderá con unos y en caso de que se tenga la señal “100” se extenderá con ceros. Esto se observa en la Ilustración 24. Pero en el caso que el byte extraído contenga un cero en la posición de mayor peso ambas codificaciones de *ContDatoMem* funcionaran de la misma manera completando el valor con ceros como se observa en la Ilustración 25.

Se quiere extraer el byte de la posición “10”.
Tiene el valor ‘1’ en la octava posición del byte.

Dirección	11	10	01	00
Dirección a leer	11011001	10111010	01110011	11010011
ContDatoMem				
000	11111111	11111111	11111111	10111010
100	00000000	00000000	00000000	10111010

Ilustración 24: Ejemplo de valor leído de la memoria, 1.

Se quiere extraer el byte de la posición “01”.
Tiene el valor ‘0’ en la octava posición del byte.

Dirección	11	10	01	00
Dirección a leer	11011001	10111010	01110011	11010011
ContDatoMem				
000	00000000	00000000	00000000	10111010
100	00000000	00000000	00000000	10111010

Ilustración 25: Ejemplo de valor leído de la memoria, 2.

El código que se ha desarrollado para realizar el circuito de la parte de lectura de la memoria de datos ha sido el que se puede examinar en la Ilustración 26.

```
if hr='1' then
    Dato_i1 := men_datos(conv_integer(Direcion));
    Dato_i2 := men_datos(conv_integer(Direcion-1));

    if ContDatoMem = "000" or ContDatoMem = "100" then
        if posicion="00" then
            f8 := Dato_i1(7 downto 0);
        elsif posicion="01" then
            f8 := Dato_i1(15 downto 8);
        elsif posicion="10" then
            f8 := Dato_i1(23 downto 16);
        elsif posicion="11" then
            f8 := Dato_i1(31 downto 24);
        end if;

        if ContDatoMem = "100" or (f8(7)='0' and ContDatoMem = "000")  then
          Dato_s <= "00000000000000000000000000" & f8;
        elsif f8(7)='1' and ContDatoMem = "000" then
          Dato_s <= "11111111111111111111111111" & f8;
        end if;

    elsif ContDatoMem = "001" or ContDatoMem = "101" then
        if posicion="00" then
            f16 := Dato_i1(15 downto 0);
        elsif posicion="10" then
            f16 := Dato_i1(31 downto 16);
        end if;
        if ContDatoMem = "101" or (f16(15)='0' and ContDatoMem = "001")  then
          Dato_s <= "0000000000000000" & f16;
        elsif f16(15)='1' and ContDatoMem = "001" then
          Dato_s <= "1111111111111111" & f16;
        end if;

    elsif ContDatoMem = "010" then
          Dato_s <= Dato_i1;

    end if;
```

Ilustración 26: Recorte código empleado en la lectura de la memoria de datos.

En la Ilustración 26 se observa que se usan dos nuevas variables que son *f8* y *f16* en las que se almacenará el valor leído dependiendo de si se lee uno o dos bytes respectivamente. Para controlar cuantos bytes se leen se emplea varias condiciones *IF* comprobando el valor de la señal *ContDatoMem* (en los casos que tiene diferencia entre extender o no, se entra en el mismo bloque de condición). Se extrae el valor de la posición indicada mediante los dos bits de menos peso de la dirección y se tendrán tres situaciones.

- Se extrae la palabra entera, se pasa el valor de la dirección leída a la señal de salida.
- Se extraen dos bytes, el circuito se comporta de igual manera cuando la señal *ContDatoMem* tiene el valor "001" y el valor del bit de mayor peso vale cero que cuando se tiene la codificación "101", en ambos casos se extenderá con ceros. En el caso que se tenga "001" y el bit de mayor peso sea uno se extenderá con unos.
  Se extraen un byte, se comportará de igual manera que cuando se extraen dos bytes, pero con la codificación de *ContDatoMem* propia.

3.1.6.2 Memoria de instrucciones

La memoria de instrucciones contiene las instrucciones que se van a ejecutar en la arquitectura del RISC-V. Este bloque tiene una entrada y una salida. La entrada es la dirección de la instrucción que se va a ejecutar y la salida es la instrucción que se ejecuta. La entrada al bloque contiene 10 bits, pero solo se emplean nueve, se descarta el bit de menos peso. Esto se debe a que las instrucciones la dirección solo tomará direcciones pares.

Esta memoria almacena datos de 32 bits. La lectura de los datos de la memoria de instrucciones se realiza de manera asíncrona. Por lo que se leerá la instrucción cuando se actualice el valor de la dirección (este valor si se actualiza de forma síncrona).

El código que se ha empleado es similar a la operación de lectura del bloque registro. Con la excepción que solo tiene una entrada y una salida. Solo se le un valor.

3.1.7 Generador de números inmediatos

Se emplea para generar los números inmediatos en las instrucciones que lo necesiten. La generación de inmediatos se llevará a cabo dependiendo del tipo de instrucción.

Como se observa en la Ilustración 13 los campos *inm[x:y]* no se encuentran seguidos, ni organizados en algunas instrucciones. Por lo tanto, tendrá que organizar los bits concatenado los valores. Una vez se tienen todos los bits organizados se extenderá el bit de mayor peso hasta los 32 bits que debe contener el numero generado.

El circuito que se desarrolla para generar los 32 bits del inmediato se realiza a partir de la instrucción que se está ejecutando. Primero se identifica el tipo de

instrucción que se está ejecutando y después se organizan los bits que contiene la propia instrucción y se extiende el signo.

3.1.8 Módulo de extensión para Multiplicación y División

Este módulo añade las instrucciones para poder realizar las operaciones de multiplicación, división y obtener el resto de la división. Para ello hay que realizar unas modificaciones en el bloque de control de la ALU y añadir las operaciones a la ALU de la arquitectura RISC-V básico para poder implementar estas operaciones.

La modificación realizada en el bloque de control de la ALU ha sido introducir un nuevo circuito que identifica las instrucciones Tipo R que pertenecen a la multiplicación y división. Las instrucciones que pertenecen a este grupo se han denominado Tipo M en este estudio.

El circuito introducido se describe mediante las siguientes funciones lógicas que tienen como salida la señal ALUp (codificación de las operaciones en la ALU). Las funciones lógicas se han generado a partir de realizar operaciones lógicas con los bits del campo *funct3.* Las funciones se observan a continuación generan el bit de la codificación de mayor a menor peso.

Tipo M→

$$ALU\_Op(4) = F3(2)\ and\ F3(1)$$

$$ALU\_Op(3) = F3(2)\ and\ not[F3(1)]$$

$$ALU\_Op(2) = \{not[F3(2)]\ and\ F3(0)\}\ or\ F3(1)$$

$$ALU\_Op(1) = \{not[F3(2)]\ and\ F3(1)\}\ or\ \{F3(2)\ and\ F3(1)\}\ or\ \{F3(1)\ and\ F3(0)\}$$

$$ALU\_Op(0) = \{F3(1)\ and\ [F3(2)\ or\ F3(0)]\}\ or\ \{not[F3(2)]\ and\ [F3(0)\ or\ not[F3(1)]]\}\ or\ \{not[F3(1)]\ and\ F3(0)\}$$

Para entrar en este circuito y obtener el valor de ALUp correspondiente con la operación que se quiera realizar se debe identificar la instrucción como tipo R y después identificarla como Tipo M. Una vez dentro de ese bloque condicional se ejecutará el circuito y se generará la codificación. Las codificaciones de las operaciones pertenecientes a este módulo son las operaciones en negrita de la Tabla 4.

Las operaciones que se incluyen con la extensión de multiplicación y división son ocho instrucciones. De las ocho cuatro se emplean para multiplicar, dos para dividir y dos para obtener el resto de la división. El nombre las funciones viene dado por la parte subrayada en la Ilustración 27.

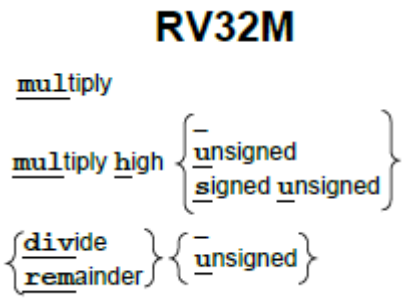

Ilustración 27: Diagrama de Instrucciones de la extensión M [24, Fig. 4.1].

Para la multiplicación se hace hincapié en que una de las cuatro instrucciones realiza el cálculo de la parte baja de la multiplicación y las otras tres realizan la multiplicación de la parte alta.

Esto se debe a que cuando se realiza una multiplicación entre dos valores un tamaño *n* el resultado tendrá el doble del tamaño de los datos (*2*n*). Por lo tanto, si se quiere obtener el resultado en una multiplicación se necesitarán dos ciclos de reloj.

La instrucción que realiza la obtención de la parte baja de la multiplicación no diferencia entre coeficientes con o sin signo. Para obtener el resultado de la parte superior si se diferencia entre coeficientes con signo, sin signo o ambas mezcladas. Por esto solo hay una instrucción para la parte baja y tres para la parte superior.

Para la división se tienen dos instrucciones, una de ellas realizará la división entre dos valores con signo (signed) la otra la realizará sin signo (unsigned). Las instrucciones que calculan el resto de la división son dos y del mismo modo que para la división se diferencian en que una calcula entre valor con signo y la otra sin signo.

3.1.9 Bloques auxiliares.

El funcionamiento de la arquitectura RISC-V necesita otros bloques para el funcionamiento como son los multiplexores, sumadores y variables constantes.

3.1.9.1 Multiplexores

Son bloques que tienen tres entradas y una salida. Dos de las entradas son datos y la otra es una señal de control que se genera en el control del procesador, excepto la señal para el salto condicional que además de depender de una señal generada por el control depende también del resultado de la ALU si su valor es cero.

Los multiplexores actúan en diversas funciones como por ejemplo en la selección del primer dato1 que se introduce a la ALU que realiza la elección entre el dato fuente uno del banco de registros y la dirección de la instrucción.

3.1.9.2 Sumador

Se emplea para realizar la suma entre dos datos.

En la arquitectura se emplea para la obtención de la dirección de la instrucción realizando la suma entre la dirección de instrucción actual y la constante de valor dos.

3.1.9.3 Constante
Se utiliza para aumentar en dos la dirección de la instrucción a ejecutar.

3.2 Programación de la arquitectura RISC-V en la FPGA

La programación de la arquitectura se ha realizado en Vivado de Xilinx realizando los bloques mencionados anteriormente en códigos separados (componentes) y juntándolos mediante en un código que contiene la arquitectura completa.

Este código tendrá como entradas el ciclo de reloj (*reloj*) que se meterá mediante un pulsador y la señal de reset (rst) que se introducirá mediante un interruptor. Además, la señal de reloj que controla la arquitectura se tendrá otra señal de reloj (*reloj2*) que tendrá un periodo de 100.000ns. Para controlar las salidas que son los cuatro display siete segmentos se tiene como entradas cuatro interruptores (*int_display* emplea tres y el restante *num_int*) que controlan el valor mostrado y la parte del valor que se muestra. Las demás salidas serán los leds que muestran las señales de control (de izquierda a derecha: AluF, EscR, EscM, MemR,LeerM, SaltoC, saltoInc, EscData, pc_dato1), el ciclo de la entrada *reloj* y la señal de reset (estos dos leds se encuentran encima de los interruptores que lo controlan). En la arquitectura la señal *reloj2* se emplea para actualizar los display y mostrar el valor en cada uno de los display pero como es una velocidad muy alta de frecuencia se ha diseñado un divisor de frecuencia que

En el display siete segmentos se mostrarán cuatro de los dígitos hexadecimales de la dirección de la instrucción, el valor de resultado de la ALU, valores leídos del registro, la instrucción o el inmediato generado dependiendo del valor de la señal *int_display*. Una aclaración es que se emplea el punto de los display (*dp*) para diferenciar entre los valores 8 y 0 con las representaciones de B y D, se emplea el punto para identificar que son las letras.

Para comprobar su funcionamiento, se han implementado y probado todos los componentes uno a uno y realizado simulaciones e implementaciones en la placa para comprobar el funcionamiento. Según se iban realizando las pruebas e iban siendo correctas se iban ensamblando en el conjunto los bloques y realizando las conexiones.

Así para realizar la comprobación de que la arquitectura funciona correctamente se ha realizado un conjunto de instrucciones que ejecuta todas las operaciones o funcionalidades que debe realizar la arquitectura.

Los bloques más importantes y que más pruebas se han realizado para una buena ejecución de la arquitectura son: la ALU, contador de programa y la memoria de datos.

El código empleado para la descripción del circuito que describe la arquitectura se puede observar en la Ilustración 28.

```
A1: Contador_programa port map(pc_ent, Reloj, rst, pc_sal);

A2: mem_instrucciones port map(pc_sal,instruccion);
A3: control port map(instruccion,AluF, EscR, EscM, MemR,LeerM, SaltoC, saltoInc, EscData, pc_dato1, ALUo, ContDatoMem);
A4: generador_inmediato port map(instruccion, inm);

A5: Registro port map(instruccion,dato1, dato2, dato_esc, reloj,EscR);

A6: ALU_control port map (instruccion, ALUo, ALUp);
M1: Multiplexor32 port map(dato2, inm, AluF, dato_alu_2);
pc_mux <= "00000000000000000000000000" & pc_sal;
M2: Multiplexor32 port map(dato1, pc_mux, PC_dato1, dato_alu_1);
A7: ALU port map(instruccion, ALUp, dato_alu_1, dato_alu_2, cero, res_alu);
A8: Memoria_datos port map( res_alu(11 downto 0), ContDatoMem,dato_leidoM, dato2, LeerM, Reloj,EscM);
M3: Multiplexor32 port map(res_alu, dato_leidoM, MemR, MuxS);
and1<= SaltoC and cero;
S0: sumador_9 port map("0000000010", pc_sal, res_sum1);
S1: sumador_9 port map(inm(9 downto 0), pc_sal, res_inm);
M4: Multiplexor9 port map(res_sum1, res_inm,and1, pc_int);
M5: Multiplexor9 port map(pc_int, res_alu(9 downto 0), saltoInc, pc_ent);
jal <= ("00000000000000000000000000" & res_sum1);
M6: Multiplexor32 port map(MuxS, jal, SaltoInc, dato_esc);
```

Ilustración 28: código de descripción del circuito RISC-V.

Para realizar el ensamblaje de la arquitectura se va realizando la llamada a los componentes anteriormente descritos relacionándolos con las señales de entrada y salida que emplea cada uno.

La primera llamada a un componente es al contador de programa que dará la dirección de la primera instrucción a ejecutar cuando se dé la señal de reloj. El segundo componente que se define es la memoria de instrucciones que generará la instrucción a ejecutar, esto pondrá en marcha los demás bloques mencionados anteriormente. Una vez se tiene la instrucción esta pasara al control de la arquitectura donde se generarán las señales de intervención. A continuación, se generan los inmediatos mediante la instrucción, la señal de control de la ALU y se obtienen los registros. En este punto se realiza la operación de la ALU. La ejecución se puede resumir con el diagrama que se observa en la Ilustración 29.

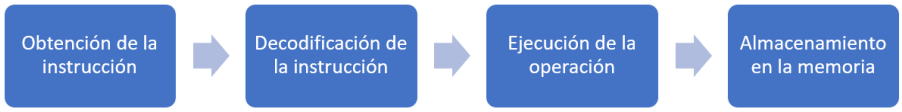

Ilustración 29: Diagrama de ejecución de la arquitectura RISC-V.

En el código de la Ilustración 28 se observan los multiplexores, el sumador, las operaciones de concatenación y la operación lógica *and* que habilita los saltos condicionales.

Teniendo el circuito de la arquitectura descrito se ha realizado un circuito adicional para observar el comportamiento en la placa Basys 3. Este circuito consiste en mostrar el resultado de la ALU o la dirección de instrucción en hexadecimal, así como iluminar un led para la señal de reloj y la reset.

Para mostrar los números hexadecimales se ha diseñado el siguiente circuito.

```
display_pre <=  res_alu when int_display = "000" else
                "00000000000000000000000000000000" & pc_sal when int_display ="001" else
                dato1 when int_display ="010" else
                dato2 when int_display ="011"else
                dato_leidoM when int_display ="100"else
                inm when int_display ="101"else
                instruccion when int_display ="110";


01: sacar_display port map(display_pre, num_int, valor_int1, valor_int2, valor_int3, valor_int4);
maq0: divisor_frecuencia port map(Reloj2, clk2);
maq1: Maquina_estados port map(clk2, aux);


valor_display <= valor_int1 when aux = "00" else
                 valor_int2 when aux = "01" else
                 valor_int3 when aux = "10" else
                 valor_int4 when aux = "11" else
                 "0000";

hab_display <= "1110" when aux = "00" else
               "1101" when aux = "01" else
               "1011" when aux = "10" else
               "0111" when aux = "11" else
               "0000";

02: display port map(display_u, valor_display, dp);
```

Ilustración 30: Código para mostrar en los displays.

Lo que primero se comprueba es el valor de la información que se quiere mostrar en los displays. En el caso que se quiera mostrar la dirección de salida se le concatenan ceros para completar los cuatro bytes. A continuación, se tiene la función que extrae los cuatro dígitos hexadecimales que se van a mostrar. Esta función agrupa de cuatro en cuatro bits pasándolo a las señales "*Valor_intx*". Dado que la velocidad de la mínima frecuencia que puede dar el reloj que contiene la placa no es suficiente para mostrar bien los valores en los displays se realiza un divisor de frecuencia que reduce esta. Posteriormente se define una máquina de estados, la cual tiene cuatro estados que ira recorriendo desde el estado cero a tres. Los valores de los estados corresponden con los valores de "*aux*". Los valores de los siguientes dos circuitos se cambian con *aux* para ir dando a cada uno de los displays el valor. Cuando se le da un valor al display hay que habilitarle (el display funciona en lógica negativa).

Y por último se tiene la función que introduciendo los cuatro bits del valor a mostrar dará como resultado el valor de los siete segmentos que se van a representa. Así como el punto que solo se iluminará cuando la representación de una letra coincida con la de un número.

Las ultimas líneas que describen el circuito desarrollado para mostrar la información del reloj, del reset mediante leds y las señales de control. Se describe mediante el código mostrado en la Ilustración 31

```
senales_control <= AluF & EscR & EscM & MemR & LeerM & SaltoC & saltoInc & EscData & pc_dato1;
clk_out <= Reloj;
rst_out <= rst;
```

Ilustración 31: Código para salidas leds.

Los leds correspondientes a la señal de control serán a los leds comprendidos entre LD8 y LD0, el asignado al reloj es el LD14 y al reset LD15.

3.3 Pruebas realizadas para la comprobación de la arquitectura

Para comprobar el correcto funcionamiento de los componentes y de la arquitectura se han realizado diversas pruebas. A continuación, se mostrarán las más importantes.

3.3.1 ALU

Una de las pruebas realizadas para comprobar el correcto funcionamiento de la ALU y de la realización de la correspondiente operación dependiendo de la codificación. Para ello se ha realizado un código que recorra todas las codificaciones de operación y se han asignado dos valores constantes a los datos entrada. Observando el comportamiento de la ALU se ha comprobado el buen funcionamiento.

De la comprobación anteriormente explicada se ha obtenido la Ilustración 32. De esta ilustración se observa la señal “as” que se corresponde con la codificación de las operaciones, el dato1 y el dato2 que respectivamente valen ocho mil uno y cuatro mil.

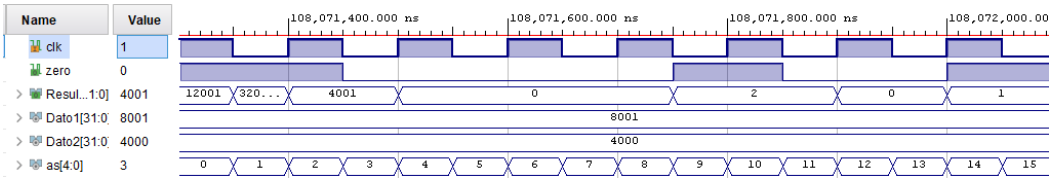

Ilustración 32: Captura resultados simulación ALU.

Examinando la Tabla 4 se ve que la codificación cero corresponde a la operación de suma, la dos a la resta. Comprobando los resultados de estas dos operaciones se ve que es correcto el funcionamiento, obteniendo doce mil uno y cuatro mil uno respectivamente.

Comprobando la Ilustración 32 se observa que hay unas codificaciones que se obtiene un valor que podría no corresponderle como por ejemplo la codificación “00011” (3) que corresponde a la instrucción BEQ (*Branch if equal*, salta sí es igual). En la instrucción BEQ se obtiene el mismo valor que en la codificación anterior como resultado, pero cambia el valor de *zero.* Esto se debe a que la instrucción BEQ compara los valores introducidos y si son iguales pondrá a ‘1’ la variable *zero* y si son distinto lo pondrá a ‘0’ sin modificar el valor del resultado porque no afecta a la ejecución en la arquitectura.

La segunda prueba que se ha realizado en la ALU ha sido comprobar el correcto funcionamiento de las operaciones con datos negativos y positivos indistintamente.

La comprobación de la suma se obtienen los resultados de la Ilustración 33. Como se observa el resultado es 3756884095 que es casi el mismo valor que el del dato2. Esto se debe a que la suma del dato1 más el dato2 supera el valor máximo con 32 bits (FFFFFFFF). Este resultado se obtiene de la suma de ambos valores menos este valor máximo. Si se Comprueba

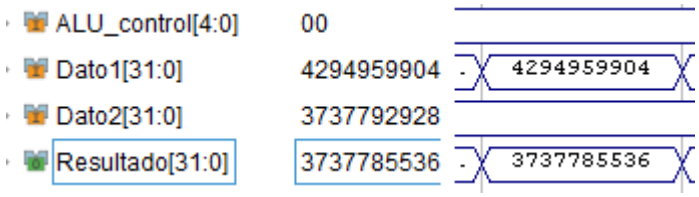

Ilustración 33: Captura resultados simulación ALU, 2.

Del mismo modo se han comprobado las demás operaciones de las ALU.

Otra prueba que se ha realizado ha sido una comprobación general del procesador (3.3.4.1). En el cual se ha comprobado el buen funcionamiento de la ALU.

3.3.2 Contador de programa

Para realizar la comprobación del contador de programa se ha realizado un código de prueba que el valor de entrada del bloque va cambiando y las señales de reset y clk se van modificando con el tiempo. Las señales de reset y clk modifican los valores del contador de programa.

En la simulación se han obtenido los siguientes resultados. Como se observa en la Ilustración 34 el funcionamiento del contador de programa es correcto debido a que, aunque se modifique el valor de *pc_in* (entrada) el valor *pc_out* (salida) no se modifica hasta que se da un flanco de subida.

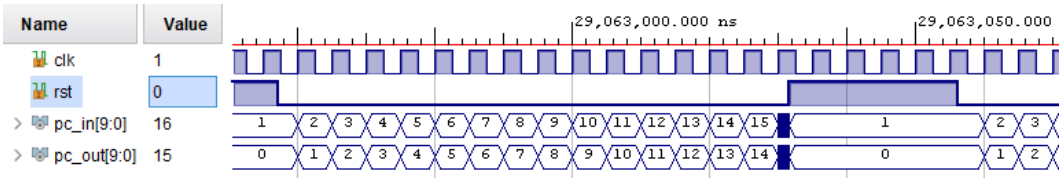

Ilustración 34: Captura resultados simulación del contador de programa.

Otra situación a tener en cuenta es cuando se mantiene activa la señal de reset (asíncrona) y se dan flancos de subida de la señal clk. Como se puede analizar mientras la señal reset está activa la salida se mantiene en cero. Cuando la señal de reset pase a cero en el siguiente flanco de subida de clk se actualiza la salida al valor correspondiente. Mientras tanto el valor de la salida será cero.

3.3.3 Memoria de datos

La memoria de datos ha sido un módulo para el que se han debido de realizar varias pruebas para comprobar que se realiza el correcto almacenamiento y lectura de los datos.

3.3.3.1 Comprobación de lectura

Se comenzará analizando la parte de lectura de la memoria. Para ello se realizará un código de prueba que recorre todas las combinaciones posibles de *ContDatoMem* que serán las codificaciones de cero hasta cinco excluyendo a la tres. Para la dirección se le dará una dirección y las posiciones de lectura del valor irán cambiando. La señal de escritura (*hw*) será constante en el valor cero y la de lectura (*hr*) será un reloj.

De la simulación anteriormente explicada se obtienen los siguientes resultados de la simulación.

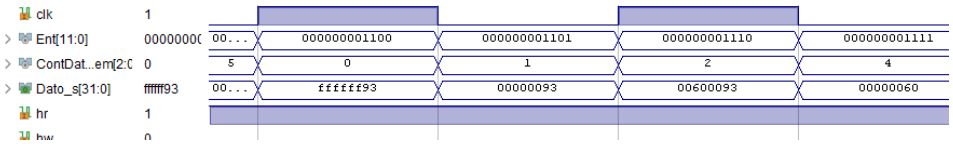

Ilustración 35: Captura resultados simulación memoria de datos, lectura.

El valor que se está leyendo es el valor “00600093” que se encuentra alojado en la dirección tres de la memoria de datos. Como se observa para leer no se necesita flanco de subida del reloj si no el cambio de cualquier señal que intervenga en el proceso de la lectura. En la Ilustración 35 se tiene cuatro casos de lectura (valores de datos leídos en hexadecimal y la entrada en binario).

- Primer caso: dirección tres, posición “00” y *ContDatoMem* “000”. En esta situación se extraerá el byte de menos peso del valor leído extendiendo el signo. El valor extraído es “93” por lo que completará el valor mediante unos. Quedando “FFFFFF93”.
- Segundo caso: dirección tres, posición “01” y *ContDatoMem* “001”. Por lo tanto, se extraerán dos bytes de la parte baja del valor de la dirección tres con extensión del signo. La parte extraída es “0093” por lo que se extenderá con ceros quedando “00000093”
- Tercer caso: dirección tres, posición “10” y *ContDatoMem* “010”. Este caso es simple y se extraen los cuatro bytes del valor de la dirección tres.
- Cuarto caso: dirección tres, posición “11” y *ContDatoMem* “100”. Como en el primer caso se extraerá un solo byte, pero sin extender el signo de la parte extraída. El valor leído de la memoria será el “00000093”.

3.3.3.2 Comprobación de escritura

En la operación de la escritura de memoria se tendrán tres instrucciones posibles que vendrán dadas por las tres posibles codificaciones de *ContDatoMem* que serán de la cero a la dos. La señal de reloj (*clk*) será un reloj de periodo 100ns, la señal de escritura (*hw*) también será un reloj de mayor periodo y la señal de lectura se encontrará a nivel bajo. La dirección de la memoria de datos se obtendrá de manera aleatoria y además se probará las cuatro combinaciones de la posición posible.

Los resultados obtenidos de esta prueba han sido los siguientes.

Se ha comprobado que no se realice la escritura cuando no está habilitada. Obteniendo un correcto funcionamiento.

La simulación se ha dado los valores de entrada dirección de la memoria: 0D y 13 en hexadecimal. Por lo que las direcciones de la memoria de datos son la tres y la cuatro así como las posiciones son "01" y "11" respectivamente.

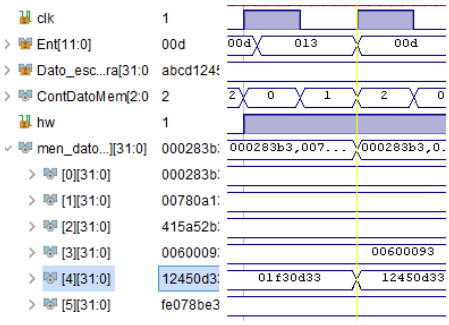

Ilustración 36: Captura resultados simulación memoria de datos, escritura 1.

En la simulación se marca con línea amarilla el instante de la señal de reloj que se analizará. En ese instante se tiene la entrada hexadecimal trece y la codificación de *ContDatoMem* uno. Por lo que se almacenará en la dirección cuatro y posición "11" (dos bytes de mayor peso), como se tiene la codificación uno serán los dos bytes. Como se observa en la Ilustración 36 se modifica los dos bytes de la parte alta del valor que se encuentra en esa dirección con los dos bytes de la parte baja del valor a escribir.

En la Ilustración 37 se muestra un tercer caso extraído de la simulación con las mismas condiciones. Pero distinta entrada y codificación. En este caso es la dirección tres, posición "01" y codificación "000". En este caso con las señales dadas se almacena el byte de menos peso en la posición del segundo byte de menos peso. Consecuentemente el valor anterior en esa dirección ("ABCD1245") se modifica con el flanco de subida del reloj quedando el valor "ABCD4545".

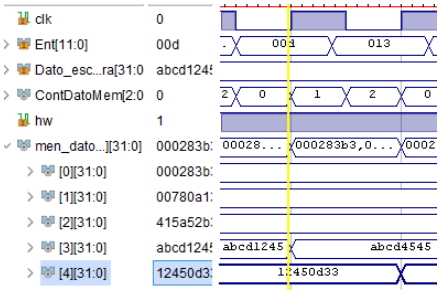

Ilustración 37: Captura resultados simulación memoria de datos, escritura 2.

3.3.4 Pruebas realizadas para el procesador

Para comprobar que el buen funcionamiento de la arquitectura se han realizado dos pruebas. La primera consiste en un programa que ejecuta las todas las instrucciones que puede realizar la arquitectura desarrollada en el estudio. La segunda se trata de un algoritmo que obtiene la serie de Fibonacci.

Para estas pruebas que se han realizado además de incluir las capturas de pantalla obtenidas en vivado se añadirán fotografías de la placa Basys 3 para ratificar el correcto funcionamiento.

3.3.4.1 Primera prueba

En esta prueba como se ha comentado se ha creado un código con todas las instrucciones que puede ejecutar la arquitectura RISC descrita.

Las instrucciones se ejecutan en grupos. Lo que quiere decir que el código creado consiste en grupos de instrucciones que se ubican en la memoria de instrucciones. La memoria de instrucciones contiene el siguiente orden de instrucciones.

Dirección	Instrucción	Dirección	Instrucción	Dirección	Instrucción
0	LUI	42	SLTI	84	LH
2	BQE -> 20	44	XORI	86	LHU
4	BNE -> 40	46	ORI	88	LW
6	BLT -> 60	48	ANDI	90	JALR -> 10
8	BGE -> 80	50	SLLI	92	-
10	BLTU -> 100	52	SRLI	94	-
12	BGEU -> 120	54	SRAI	96	-
14	JALR ->0	56	JALR -> 6	98	-
16	-	58	-	100	MUL
18	-	60	SW	102	MULH
20	ADD	62	SW	104	MULHSU
22	SUB	64	SW3	106	MULHU
24	SLL	66	SW4	108	DIV
26	SLT	68	SB	110	DIVU
28	XOR	70	SH	112	REM
30	SRL	72	JALR ->8	114	REMU
32	SRA	74	-	116	JALR -> 0
34	OR	76	-	118	-
36	AND	78	-	120	AUIPC
38	JALR -> 4	80	LB	122	JAL -> 0
40	ADDI	82	LBU	124	-

Tabla 6: Dirección de la memoria de instrucción y su correspondiente instrucción.

Una vez expuesto el orden de ejecución de las instrucciones se procederá a explicar lo que realiza este código de prueba. Como partida inicial se realiza la instrucción LUI que asignará el valor cero a la dirección de registro uno para que se cumpla la condición de la segunda instrucción *BEQ* que realiza un salto en caso de que ambos datos sean iguales. En este caso se compara las direcciones de registro cero y uno, la cero contiene siempre el valor cero y la uno se le asigna con la instrucción LUI por lo tanto se realiza el salto a la dirección de instrucción veinte. En este punto se realizan las instrucciones básicas tipo R, una vez ejecutadas mediante una instrucción *JALR* se le da la dirección de instrucción cuatro que realiza otra instrucción Tipo B. En este caso todas las instrucciones tipo B se cumplen y realizan el salto a la dirección de

instrucción que se indica en la tabla. Los bloques a los que direccionan las instrucciones de salto condicional son: instrucciones básicas tipo R, instrucciones básicas del tipo I, instrucciones de almacenamiento tipo S, instrucciones de descarga tipo I y como último bloque se tienen las instrucciones tipo R de multiplicación denominadas en el estudio como instrucciones tipo M. El comportamiento de las instrucciones codificadas es el que se observa en la Ilustración 38.

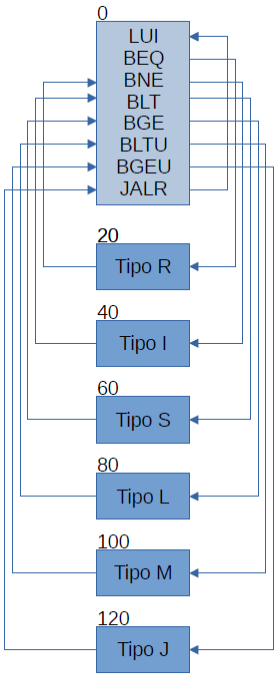

Ilustración 38: Diagrama de ejecución de la simulación.

Realizando la simulación se obtiene el siguiente comportamiento que se acompañará como se ha comentado de su respuesta en la placa.

Para esta simulación se han dado el valor a la señal de reloj que controla el procesador. Para las entradas que controlan la parte de visualización de datos en la placa no se han dado valores en esta simulación dado que se quiere comprobar el funcionamiento de la arquitectura. En cada captura se observarán las señales más importantes en cada situación.

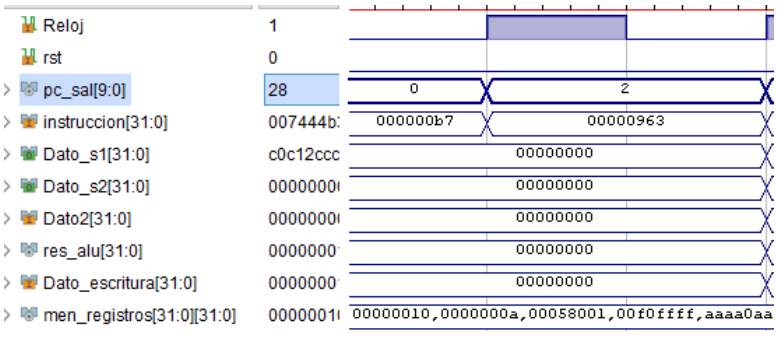

Ilustración 39: Captura simulación de la arquitectura, 1.

En la Ilustración 39 se observa las dos primeras instrucciones que ejecuta la arquitectura que son la instrucción LUI que escribe en la dirección uno del banco de registros el valor cero y la instrucción *BEQ* que salta a la instrucción veinte en caso de que el registro fuente cero y uno valgan lo mismo. Con la instrucción LUI se consigue que esto se cumpla y por lo tanto se dé el salto. La instrucción veinte realiza la operación de suma (*ADD*) como se explicará a continuación.

Después de la instrucción *BEQ* se tiene las operaciones de suma y resta. Como se comprueba en la Ilustración 40 el dato dos es el mismo para ambas operaciones (utilizan el mismo segundo registro fuente). El resultado de la operación de suma será el registro fuente uno para la operación de resta. Por lo tanto el resultado de la resta es el mismo valor que el del registro fuente uno de la operación de suma.

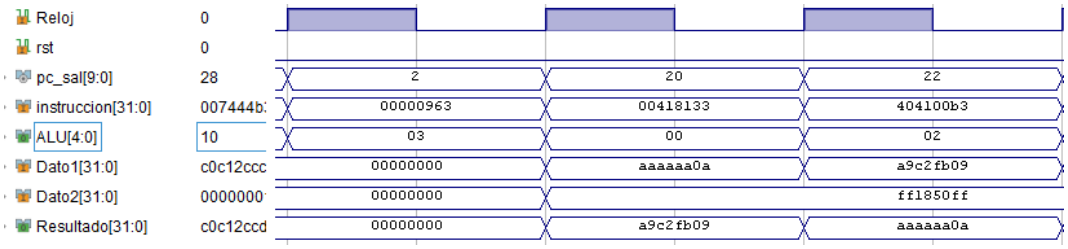

Ilustración 40: Captura simulación de la arquitectura, 2.

Si nos pasamos a la placa y se comprueba las operaciones realizadas se obtiene que la dirección de instrucción veinte es una suma entre el registro fuente tres y cuatro que contienen los valores "AAAAAA0A" y "FF1850FF" respectivamente. El resultado de esta operación es "A9C2FB09" como se puede comprobar en la placa. El valor obtenido se almacenará en la dirección dos del banco de registros.

Ilustración 41: Conjunto de fotografías de los valores obtenidos en la placa para la instrucción 20.

La siguiente instrucción que se tiene empleada será la resta entre el valor obtenido de la dirección dos y cuatro del banco de registros. Luego se empleará el valor obtenido en la anterior instrucción. Dando como resultado el valor "AAAAAA0A". Esto se debe a que se realiza la resta del valor obtenido en la suma de la instrucción anterior y uno de los sumandos de está dando como resultado el valor del otro sumando de la suma. Como se observa en la Ilustración 42.

Ilustración 42: Conjunto de fotografías del resultado de la instrucción 22.

En estas dos instrucciones se tiene las mismas señales de control. En la placa se observan las siguientes señales como se muestra en la Ilustración 43. Solo se encuentra iluminado un LED de control (van desde el led LD8 a LD0). El led iluminado corresponde con la señal de control EscReg. Comprobando la Tabla 3 se observa que en las instrucciones Tipo R solo tiene una señal de control de un bit a nivel alto y es la señal de escribir en el registro.

Ilustración 43: Fotografía leds encendidos en la placa durante la instrucción 22.

Además, ejecutando estas dos instrucciones se comprueba que la escritura en el banco de registros se realiza de manera correcta.

En el siguiente grupo de operaciones definidas se encuentran las que se realizan entre un valor extraído del banco de registros y el inmediato generado. Se ha comprobado que los resultados son correctos. Además, se adjunta una fotografía que muestra las señales de control utilizadas en dichas operaciones. Se muestra en la Ilustración 44. En este caso se ilumina la señal de control EscReg y AluF.

Ilustración 44: Fotografía señales de control de la instrucción 52 (34 hexadecimal).

Continuando con la ejecución de las instrucciones programadas en la memoria encontramos las instrucciones de lectura de la memoria de datos ubicadas a partir de la instrucción ochenta (50 hexadecimal). Seguidamente se expondrán fotografías de la instrucción ochenta.

Ilustración 45: Fotografía de la dirección de instrucción y señales de control.

Como se observa en la Ilustración 45 se tienen 4 señales de control activas, las cuales de derecha a izquierda son: AluF, EscReg, MemR y LeerM. Comprobando la Tabla 3 se ve que son correctas las señales control para las instrucciones tipo I para la lectura de valores de la memoria. La instrucción que corresponde a esta dirección de memoria es “0CD00303”. De la instrucción se

extrae que la señal de control *ContDatoMem* es “000” por lo tanto se leerá solo un byte.

En la Ilustración 46 se observa la dirección de la memoria de datos de donde se leerá el dato. En el caso de las instrucciones de lectura de la memoria de datos el resultado de la ALU será la dirección de lectura. En este caso se tiene el valor hexadecimal “CD” que dará la dirección de memoria cincuenta y uno y la posición “01”. Lo que hace que se lea el segundo byte de la dirección cincuenta y uno y se extenderá el bit de mayor peso.

Ilustración 46: Fotografía de la dirección de memoria de datos a leer.

Comprobando los valores asignados a la memoria de datos se ve que los valores a partir de la dirección ocho, tienen el valor hexadecimal “90780A13”. Por lo que se extrae el valor 0A y como el bit de mayor peso es 0 se extenderá hasta los 32bits con ceros.

Para ver el dato leído de la memoria de datos o el dato que se va a escribir en el banco de registros se tiene la combinación de interruptores siguiente: para mostrar este valor se tendrá a uno el cuarto interruptor por la derecha. Para mostrar la parte baja o la parte alta se pondrá a cero o uno el interruptor de la derecha.

Ilustración 47: Fotografía del dato leído de la memoria de datos.

Observando la a partir de la instrucción 8(JALR hacia las instrucciones tipo L) se mostrará la instrucción anteriormente comentada (operación LB) y la siguiente (operación LBU). Ambas instrucciones tienen el mismo registro destino y como el bit de mayor peso del valor leído es cero en ambos casos extiende con ceros. Por ello el dato a escribir en ambas instrucciones es el mismo.

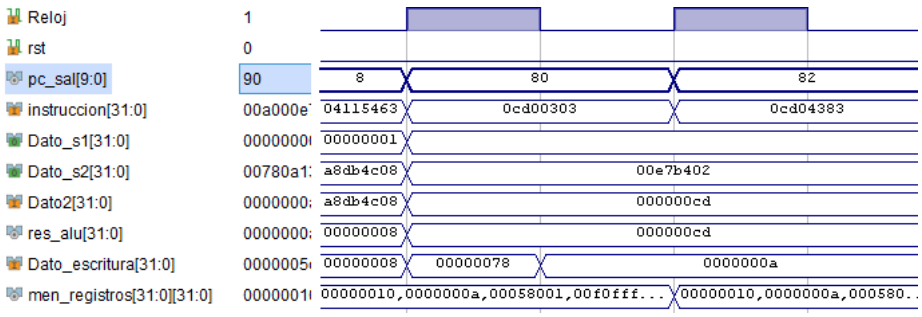

Ilustración 48: Captura simulación de la arquitectura, 2.

3.3.4.2 Segunda prueba

Para esta segunda prueba se han realizado las instrucciones que generan la serie de Fibonacci durante cuarenta y un intervalos, los valores de la serie se almacenarán en la memoria de datos a partir de la dirección uno de la misma. Para generar la serie de Fibonacci se debe tener en cuenta lo siguiente nociones sobre cómo se realiza. Los valores iniciales son el cero ($fb0_0$) y el uno ($fb1_0$). Y los términos se obtienen mediante la suma de los dos anteriores. Para explicar el algoritmo se emplearán dos variables *fb0* y *fb1*, además se indicará cuando se ha generado cada variable mediante el subíndice.

$$fb1_{e+1} = fb1_e + fb0_e$$

$$fb0_{e+1}=fb1_e$$

Seguidamente con estas dos ecuaciones se tendrá las siguientes ejecuciones:

- Paso inicial

$$fb1_1 = fb1_0 + fb0_0=1+0=\mathbf{1}$$

$$fb0_1=fb1_0=1$$

- Paso uno

$$fb1_2 = fb1_1 + fb0_1=1+1=\mathbf{2}$$

$$fb0_2=fb1_1=1$$

Los valores generados en la serie son los valores iniciales y los que se van obteniendo en la variable *fb1*. Como se observa los primeros valores de la serie son: 0, 1, 1, 2…. La serie continuará siguiendo las siguientes ecuaciones infinitamente. Por lo que la serie hasta el onceavo valor será: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. En el caso que se está analizando el valor en la dirección cuarenta y dos de memoria de datos será el 165580141 (9DE8D6D en hexadecimal).

Para ello se han empleado las instrucciones que se observan en la Tabla 7.

Dirección	Instrucción
0	**LUI** X4, 0
2	**LUI** X5, 160
4	**SRLI** X5, X5, 12
6	**LUI** X8, 0
8	**LUI** X7, 1
10	**SRLI** X7, X7, 12
12	**SW** X8, 4(X0)
14	**SW** X7, 8(X0)

Dirección	Instrucción
16	**BEQ** X4, X5, 14
18	**ADD** X9, X0, X7
20	**ADD** X7, X7, X8
22	**ADD** X8, X0, X9
24	**SW** X7, 12(X4)
26	**ADDI** X4, X4, 4
28	**JALR** X4, X5, 14

Tabla 7: Dirección de la memoria de instrucción y la instrucción que realiza.

La ejecución de las instrucciones se resume en el diagrama de ejecución de la Ilustración 49.

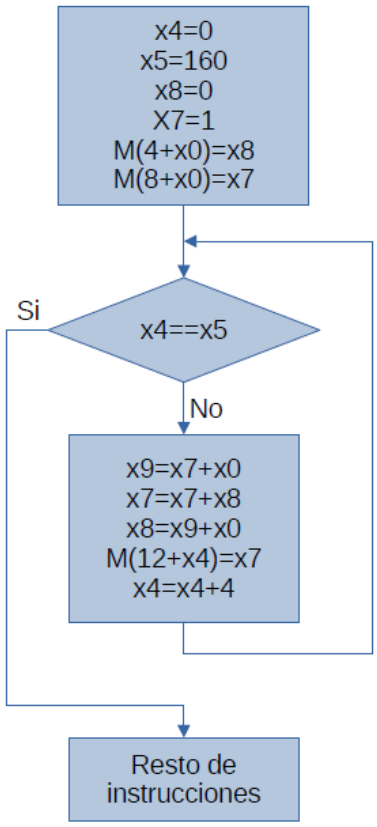

Ilustración 49: Diagrama de ejecución de las instrucciones de la serie de Fibonacci.

Las primeras tres instrucciones asignan los valores para realizar el contador. Primero se asigna al registro *x4* el valor cero que será la variable que cuente y al registro *x5* el valor ciento sesenta, pero se le desplaza doce bits a la derecha para que el valor que contiene el registro *x5* realmente sea el valor que se quiere. El desplazamiento se realiza dado que la instrucción LUI escribe el valor empezando en el decimotercer bit del inmediato. Se le ha dado el valor ciento sesenta para que además de utilizarlo de contador emplearlo a la hora de almacenar valores en la memoria de datos (cada cuatro bytes es un valor). La instrucción que tiene la dirección veintiséis será la que aumente el valor del contador y lo hará en cuatro unidades. Consecuentemente se tendrán cuarenta y un valores generados en la serie de Fibonacci.

Las siguientes cinco instrucciones se emplean para asignar los dos primeros valores de la serie y almacenarlos en la memoria de datos. Las tres primeras de este grupo son similares a las explicadas anteriormente con la excepción que en el registro *x8* se almacena el valor cero y en el *x7* se almacena el valor uno. Para el registro *x7* se realiza de nuevo la operación de desplazar a la derecha doce bits quedando el valor uno en el banco de registro. Las dos instrucciones *SW* se emplean para almacenar los registros *x8* y *x7* en las direcciones cuatro y ocho de la memoria de datos.

Posteriormente se tendrá un bucle *FOR* que será el que realmente genere la serie de Fibonacci. Este bucle se consigue con la instrucción *BEQ* que compara el registro *x4* (contador) con el registro *x5* (valor límite del bucle) y si ambos valores son iguales esta saltará a la dirección de instrucción treinta (*dirección actual más 14*). Una vez ejecuta las instrucciones del bucle se ejecuta la instrucción *JALR* que dará la dirección de la instrucción *BEQ*.

Dentro del bucle primero se le da al registro *x9* el valor del registro *x7* mediante la suma del registro *x7* y *x0* (siempre tiene el valor cero). Después se realiza la suma entre el registro *x7* y *x8* que se almacenará en el registro *x7*. Se pasa el valor del registro *x9* al registro *x8* (misma forma que cuando se trabajaba con *x9=x8*). La siguiente instrucción se emplea para guardar el valor generado de la serie de Fibonacci en la memoria de datos. El valor se almacenará en la dirección doce más el valor del registro *x7*. La última instrucción que se ejecuta dentro del bucle realiza la operación de sumar cuatro al contador que se realiza mediante la suma de un registro y un valor inmediato.

Implementando el RISC-V en la FPGA con las instrucciones para generar la serie de Fibonacci observamos los resultados en la simulación generada por Vivado y en la placa.

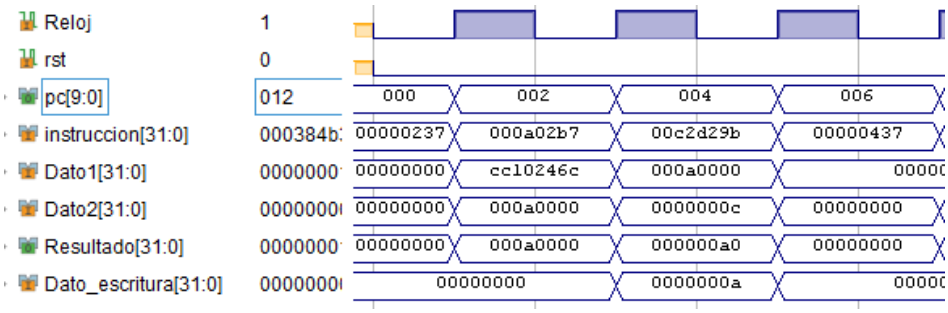

Ilustración 50: Captura de la simulación primeras instrucciones ejecutadas.

La primera instrucción se puede ejecutar cuando se inicia el programa o después de un reset. En el programa es la "00000237" como se muestra en la Ilustración 50 y en la placa se colocan los interruptores para visualizar la parte baja de la instrucción (Ilustración 51).

Ilustración 51: Fotografía instrucción dirección cero.

Las siguientes dos instrucciones como se ha comentado se emplean para almacenar en el registro *x5* el valor ciento sesenta. La segunda corresponde como se indica en la Ilustración 52 a la dirección cuatro de la memoria de instrucciones que corresponde con la operación de desplazar el valor lógicamente a la derecha doce veces. Por lo tanto, el valor inicial en el registro *x5* "000A0000" si se desplaza a la derecha doce veces queda el valor "000000A0" que es el esperado.

Ilustración 52: Conjunto de fotografías de la dirección de instrucción cuatro.

Las instrucciones ubicadas en la octava y décima posición de la memoria de instrucciones realizan la misma operación que las ubicadas en la segunda y tercera posición exceptuando que trabajan para obtener el valor uno.

Observando varios ciclos del bucle de obtención de la serie de Fibonacci se observa que es correcto. Para verificarlo se expondrá la captura en la que el bucle escribe en la dirección catorce de la memoria de datos.

Los resultados obtenidos en la placa son:

- Para la dirección de instrucción dieciocho, se obtiene el valor de resultado (dirección de la memoria de datos donde se va a escribir el dato2 extraído del banco de registros): 38 en hexadecimal que corresponde con la dirección catorce (decimal) y posición "00" en binario (no tiene importancia cuando se escribe la palabra completa).

Ilustración 53: Fotografía resultado de la ALU.

El dato dos extraído de la ALU (dato a escribir en la memoria de datos) es el "000000E9".

Ilustración 54: Fotografía dato a escribir en la memoria de datos.

El valor que en este instante tiene el contador es "002C" como se observa en la Ilustración 55. Se puede comprobar que el programa funciona correctamente dado que el valor del contador más doce tiene que dar el valor de la dirección de escritura. El valor 2C en decimal es cuarenta y cuatro más los doce que son constante para la suma de la dirección da como resultado cincuenta y seis que en hexadecimal es 38.

Ilustración 55: Fotografía del valor del contador.

Comprobando la simulación se observa que los resultados expuestos en las fotografías son correctos, tal y como se muestra en la Ilustración 56 y la Ilustración 57.

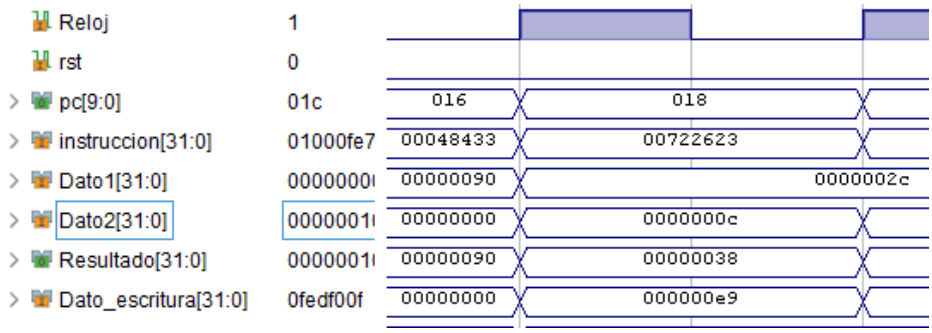

Ilustración 56: Captura simulación, instrucción 18 con valor del contador 2C

Ilustración 57: Captura simulación, valor de memoria de datos en la dirección catorce.

Comparando las fotográficas de la placa expuestas con las capturas de la simulación se comprueba que la ejecución es la misma en ambos medios.

Comprobando los valores de las instrucciones que tienen en la dirección dieciséis de la memoria de instrucciones se observa que los valores obtenidos como resultado corresponden con los valores de la serie de Fibonacci. Consecuentemente se obtiene el correcto funcionamiento.

Capítulo 3: RISC-V [12], [13], [19]

Capítulo 4: Comparación de los multiplicadores

En este capítulo se realizará la comparación de tres modos diferentes de realizar la multiplicación en el procesador. Se compararán los códigos, así como la cantidad de elementos empleados y los retardos máximos en obtener los resultados. Los tres modos de multiplicación que se emplearán son: Multiplicador paralelo CSA (*Parallel Carry Save Adder Multiplier*) planteado por Deschamps [25, Cap. 8.2.2], el realizado por compilador de Xilinx cuando solo se introduce el símbolo de multiplicación y la multiplicación de Árbol de Wallace que se elabora en este trabajo.

Como los autores comentan en [25, p. 189] hay muchos algoritmos que describen circuitos multiplicadores basados en Árboles de contadores como pueden ser el de Wallace o el de Dadda. Pero como dejan entender que los algoritmos CSA implementados en las FPGAs pueden ser más eficientes.

Se analizará el Multiplicador paralelo CSA ya que emplea una arquitectura de ejecución distinta a los otros multiplicadores. El circuito de multiplicación de Árbol de Wallace que se desarrolla en este trabajo y el paralelo CSA son similares dado que realizan suma por columnas pasando acarreos, pero con la diferencia de como realizan estas. El algoritmo propuesto por Deschamps realiza las multiplicaciones entre bits de los coeficientes y las introduce una a una en un sumador y en cambio el Árbol de Wallace realiza todas las multiplicaciones entre bits en el inicio y las agrupa en sumadores. Ambos algoritmos emplean el principio de sumador de ahorro de acarreo (CSA) para obtener el resultado final.

El bloque CSA funciona igual que un sumador total por lo tanto tiene el mismo circuito. La diferencia es el nombre que se le dan a las señales al bloque. El diagrama que tiene este bloque es la que se muestra en la Ilustración 58.

Ilustración 58: Diagrama del sumador carry-save.

Como se ve tiene tres entradas y dos salidas que se aplicara hasta que solo queden dos números que se sumarán mediante otro protocolo de suma. En el caso que se van a analizar se realizara mediante CLA (Carry Look Ahead).

En todas las multiplicaciones que se analizan los datos de entrada tendrán el mismo tamaño. El tamaño de los datos a analizar va desde 4 hasta 128 bits. Y solo se emplearán valores del tamaño de 2^n.

A continuación, se expondrán y analizarán algunas de las características de cada uno de los códigos:

4.1 Multiplicación implementada por Xilinx

Este circuito es el más simple de los tres que se analizarán. Incluye dentro de la arquitectura una única línea de código que permite al compilador de Xilinx realizar la implementación de un multiplicador sobre la FPGA. Para este código se definen los datos de entrada (*x* e *y*), el de salida (*z*) y la descripción del circuito. El código empleado es.

```
ENTITY multiplier IS
  GENERIC(n,m: NATURAL:= 8);
PORT(
  x: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  y: IN STD_LOGIC_VECTOR(m-1 DOWNTO 0);
  z: OUT STD_LOGIC_VECTOR(n+m-1 DOWNTO 0)
);
END multiplier;

ARCHITECTURE circuit OF multiplier IS
BEGIN
    z<=x*y;
END circuit;
```

Ilustración 59: Recorte código multiplicación mediante símbolo.

4.2 Multiplicador paralelo CSA

Este código se extrae del libro "*Guide to FPGA Implementation of Arithmetic Functions"*[25]. Se ha tenido que realizar una pequeña modificación dado que los códigos que se aportan en este libro se plantean para realizar la multiplicación entre dos datos y sumarle dos valores a la multiplicación. Este código consiste en una arquitectura de multiplicaciones de un bit y de sumas totales (*Full-adder*).

Carry Save Adder (CSA) o Sumador de ahorro de acarreo

El código original propone la multiplicación: $z = x * y + v + u$. Y que el valor del multiplicando y el multiplicador pueda tener un tamaño distinto.

La modificación que se ha realizado plantea: $z = x * y$, por lo tanto se han eliminado los valores que se sumaban a mayores, además, se debe cumplir que los coeficientes tienen el mismo número de bits.

El código plantea el siguiente esquema para la multiplicación de valores de 4 bits:

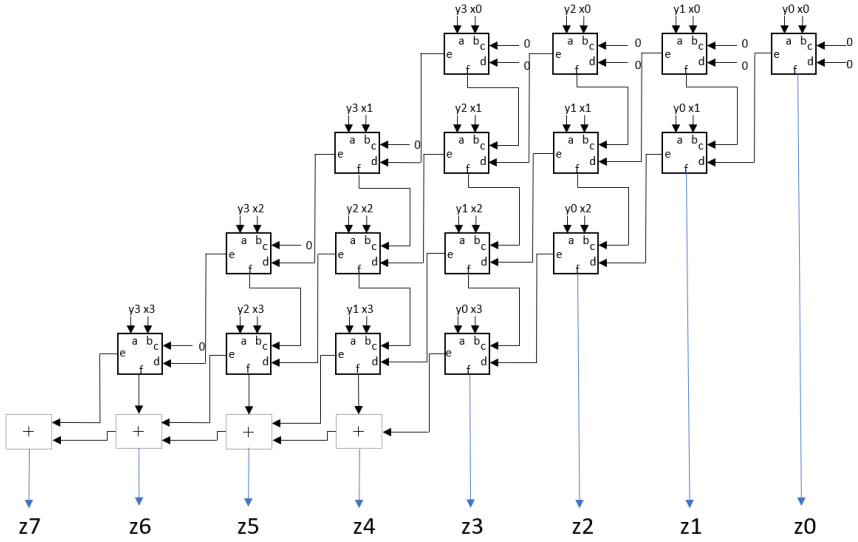

Ilustración 60: Arquitectura esquemática de los sumadores en el algoritmo de multiplicación: Multiplicador paralelo CSA.

En la arquitectura se puede ver que los sumadores se organizan por filas y cada fila inferior se desplaza una columna a la izquierda (forma de paralelogramo). De las entradas de los bloques que ejecutan la operación se tiene como entradas un bit del multiplicando, otro del multiplicador y en algunos casos se tiene un acarreo y resultado obtenidos en los otros bloques. La salida del resultado (*f*) de los bloques se introduce en la puerta *c* del bloque inferior en el caso que haya. En el caso del acarreo (*e*) es diferente al resultado. Lo que se hace con la salida *e* es introducirle en la entrada *d* del bloque inferior una posición desplazada a la izquierda. Por lo tanto la relación entre los bloques y las salidas es: *f* se pasa verticalmente hacia abajo (↓) y *e* se pasa diagonalmente hacia abajo a la izquierda (↙). Con las excepciones que en los bloques de la primera fila no se tienen resultados ni acarreos superiores y en los bloques de la diagonal izquierda no se tienen resultados, en estos bloques se introducen ceros.

Como se observa hay varios tipos de bloques en la arquitectura. Todos los bloques excepto los tres inferiores son sumadores totales modificados. Esto se debe a que en [1, Cap. 8.2.2] se plantea que dentro de estos bloques sumadores se realice una multiplicación lógica (And) además de realizar la suma de los tres valores. La operación lógica *and* realiza la multiplicación entre los bits del multiplicando y el multiplicador. El símbolo y la estructura interna de los bloques es la que se observa en la Ilustración 61.

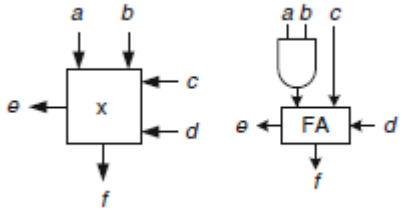

Ilustración 61: Símbolo y estructura interna de los bloques empleados.[25, Fig. 8.1]

La operación lógica que se realiza dentro de estos bloques es la planteada en la Ilustración 62.

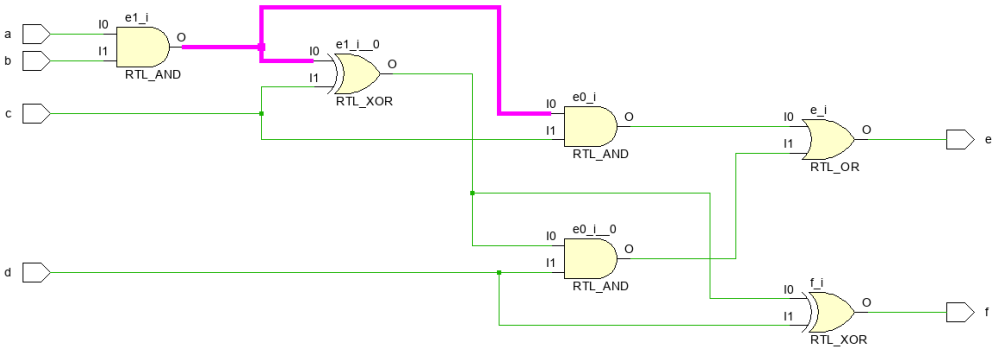

Ilustración 62: Estructura desarrollada de los bloques empleados.

En la Ilustración 62 se tiene implementada mediante bloques lógicos la función que se desarrolla dentro del bloque de la Ilustración 61. La función tiene cuatro entradas y dos salidas de un bit. Las entradas *a* y *b* son los bits del multiplicando y multiplicador respectivamente, las entradas *c* y *d* son bits que se obtienen como resultado y acarreo de los otros bloques.

En la función se realizan seis operaciones lógicas. Los bits *a* y *b* se multiplican mediante un *and* obteniendo la señal representada en rosa en la Ilustración 62. Teniendo los valores *c, d* y *señal rosa* se realiza la operación correspondiente a un sumador total. Las cuatro entradas del bloque darán como resultado dos salidas el resultado (*f*) y el acarreo (*e*) que se relacionan de la siguiente forma para obtener el resultado: $f = ((a\ and\ b)\ xor\ c)\ xor\ d$ y para la elaboración del acarreo: $e = [((a\ and\ b)\ xor\ c)\ and\ d]\ or\ [(a\ and\ b)\ and\ c]$.

El algoritmo se puede observar en la Ilustración 63 que muestra la arquitectura del código VHDL.

```
ARCHITECTURE circuit OF parallel_csa_multiplier IS
  TYPE matrix IS ARRAY (0 TO m-1) OF STD_LOGIC_VECTOR(n-1 DOWNTO 0);
  SIGNAL c, d, e, f: matrix;
  SIGNAL first_operand: STD_LOGIC_VECTOR(n-2 DOWNTO 0);
  SIGNAL second_operand: STD_LOGIC_VECTOR(n-1 DOWNTO 0);
BEGIN                                                              Parte 1
  main_iteration: FOR i IN 0 TO m-1 GENERATE
    internal_iteration: FOR j IN 0 TO n-1 GENERATE
      f(i)(j) <= (x(j) AND y(i)) XOR c(i)(j) XOR d(i)(j);
      e(i)(j) <= (x(j) AND y(i) AND c(i)(j)) OR (x(j) AND y(i) AND d(i)(j)) OR (c(i)(j) AND d(i)(j));
    END GENERATE;
  END GENERATE;

  connections1: FOR j IN 0 TO n-1 GENERATE
    c(0)(j) <= '0';
  END GENERATE;                                                    Parte 2
  connections2: FOR i IN 1 TO m-1 GENERATE
    connections3: FOR j IN 0 TO n-2 GENERATE
      c(i)(j) <= f(i-1)(j+1);
    END GENERATE;
    c(i)(n-1) <= '0';
  END GENERATE;

  connections4: FOR j IN 0 TO m-1 GENERATE
    d(0)(j) <= '0';
  END GENERATE;
  connections5: FOR j IN m TO n-1 GENERATE
    d(0)(j) <= '0';
  END GENERATE;                                                    Parte 3
  connections6: FOR i IN 1 TO m-1 GENERATE
    connections7: FOR j IN 0 TO n-1 GENERATE
      d(i)(j) <= e(i-1)(j);
    END GENERATE;
  END GENERATE;

  outputs: FOR j IN 0 TO m-1 GENERATE                              Parte 4
    z(j) <= f(j)(0);
  END GENERATE;                                                                Parte 5
  first_operand<= f(m-1)(n-1 DOWNTO 1);
  second_operand <= e(m-1);
  z(n+m-1 DOWNTO m) <= first_operand + second_operand;

END circuit;
```

Ilustración 63: Recorte arquitectura del código Multiplicador paralelo CSA.

La arquitectura del código VHDL tiene la parte declaración de señales y la descripción del circuito. Seguidamente se realizará el análisis de ambas partes de la arquitectura (a su vez se divide en cinco partes la descripción del circuito).

En la declaración de señales de la arquitectura se tienen definidas las matrices *c, d, e* y f que tiene un tamaño de (*n-1*)(*m*-1). La organización de los campos de las matrices es: *matriz(fila)(columna)*. Las matrices *f* y *e* almacenarán los valores del resultado y acarreo derivado de las operaciones de suma. Las matrices *c* y *d* son empleadas para almacenar los valores obtenidos de *e* y *f* obtenidos en cada fila y ser empleados en posteriores ejecuciones de la parte uno. Después de las matrices se realiza la declaración de los vectores *firts_operand* y *second_operand* que serán utilizados para el almacenamiento temporal del resultado.

En la Ilustración 63 se ve que en la parte de la descripción del circuito hay tres *connections* que no están dentro de ninguna de las partes. Estas son: *connections1* su función original era introducir los valores de *u* en la primera

fila de la matriz *c*, con la modificación realizada su función es introducir ceros en esa fila. En la Ilustración 60 representa los valores introducidos en los bloques de la primera fila en la entrada *c*. *Connections4* de igual forma que la anterior su función era introducir los valores de *v,* pero en el algoritmo actual es introducir ceros en la matriz *d*. *connections5* originalmente se plantea para los casos en que el valor de *m* y de *n* es distinto (siempre debe ser *n* mayor que *m*). Pero como en el caso desarrollado se plantea que ambos valores sean iguales esta *connections* no se empleará.

A continuación se examinará cual es la función y lo que realiza el algoritmo descrito de las cinco partes en las que se divide la descripción del circuito.

La parte 1 es la iteración principal y su función es la de crear los bloques sumadores totales desarrollados en la Ilustración 61. En esta parte obtiene el valor del resultado (*f*) y el acarreo (e). Los valores se obtienen mediante el conjunto de la operación lógica *and* y el sumador total. El código que se ha descrito es el que se muestra en la Ilustración 64. Se observa que las sentencias que obtienen las salidas (*f* y *e*) se encuentran dentro de un bucle *FOR* anidado. El circuito descrito dentro de los bucles anidados es el desarrollado para la explicación de la Ilustración 62.

```
main_iteration: FOR i IN 0 TO m-1 GENERATE
  internal_iteration: FOR j IN 0 TO n-1 GENERATE
    f(i)(j) <= (x(j) AND y(i)) XOR c(i)(j) XOR d(i)(j);
    e(i)(j) <= (x(j) AND y(i) AND c(i)(j)) OR (x(j) AND y(i) AND d(i)(j)) OR (c(i)(j) AND d(i)(j));
  END GENERATE;
END GENERATE;
```

Ilustración 64: Código generar circuito sumador.

El uso de emplear un bucle *FOR* anidado es que se vayan recorriendo los valores de cada fila y una vez completada la fila pasar a la siguiente de la matriz empleada. La obtención de las matrices de resultados y acarreos se irá realizando fila a fila desde el bit de menos peso al de mayor.

La parte 2 contiene “connections2” y “connections3” y su función es de almacenar los valores obtenidos como resultados (*f)* en la matriz *c*. Esto se realiza para emplear los valores obtenidos del resultado en las sumas de la siguiente fila. Estas connections se componen de dos bucles *FOR* anidados. Dentro de los bucles se diferencian dos sentencias que tienen como almacenamiento la matriz *c.*

La primera sentencia se utiliza para almacenar los resultados (matriz *f*) obtenidos de las sumas de cada etapa en la matriz *c*. Como se puede observar en la Ilustración 60 los resultados obtenidos de los bloques de la diagonal derecha no se emplean en las sumas posteriores. En el código de la Ilustración 65 se representa de forma que el valor de la primera columna de la matriz c no corresponda con la primera columna de la matriz *f*. Si no que los valores de la columna de la matriz *f* irán de uno hasta *n* menos uno (*n-1*), lo que significa que la variable *j* usara el rango desde cero hasta *n* menos dos (*n-2*). Que la matriz

c solo llegue a *n* menos dos se debe al desplazamiento realizado (no añadir el valor de la diagonal izquierda). Generando que la matriz *f* tenga un valor más en que la matriz *c* en la columna. En la sentencia la columna de la matriz *c* es *j* y en la matriz *f* será *j+1.*

Observar que las matrices *f* no pasa los valores en las mismas posiciones de la matriz *c*. Por lo tanto los valores de la matriz *f* se almacenan a partir de la segunda fila de matriz *c*, así como los valores de la columna de menos peso de la matriz *f* no se pasa. Esto se debe que en la primera fila de la matriz *c* se encuentran almacenados ceros y que los resultados (matriz *f*) obtenidos en la fila *i* se emplearan como sumandos (matriz *c*) en la obtención de la fila *i* más uno (*i+1*). Esto se representa en la sentencia relacionando la fila i de la matriz *c* con la fila i menos uno (*i-1*) de la matriz *f*. El valor de i irá desde uno hasta *n* menos uno.

Por lo tanto la relación entre la matriz *c* y la matriz *f* es: *c(i)(j) = f(i-1)(j+1).*

Como se observa las matrices tienen tamaño *(n-1)(m-1)* y con esta primera sentencia no se rellena al completo quedando una vacante la columna de mayor peso en cada fila. Esta vacante se rellena con la segunda sentencia. La segunda sentencia solo emplea el bucle *FOR* más externo y se utiliza para rellenar con cero el bit de mayor peso. Esto se observa en la Ilustración 60 con los ceros que se introducen en los bloques de la diagonal izquierda en la entada *c*. En el código se realiza de forma que la matriz *c* recorrerá los valores de *i* para las filas y para las columnas mantendrá el valor de *n* menos uno (*n-1*) y a esto se le iguala el valor cero, quedando la sentencia *c(i)(n-1)=0.*

```
connections2: FOR i IN 1 TO m-1 GENERATE
  connections3: FOR j IN 0 TO n-2 GENERATE
    c(i)(j) <= f(i-1)(j+1);
  END GENERATE;
  c(i)(n-1) <= '0';
END GENERATE;
```

Ilustración 65: Código de connections2 y connection3

La parte 3 del algoritmo se utiliza para pasar los valores de los acarreos obtenidos en los bloques a la matriz *d*. En este caso como se observa en la Ilustración 60 los valores de los acarreros generado en cada fila se emplearan en la misma columna pero en la siguiente fila. La matriz *d* contiene en la primera fila ceros (en el algoritmo incial se tendría los bits de *v*).

Por lo tanto en la Ilustración 66 se puede ver que en la matriz *d* se introduce la matriz *e* deplazada una fila abajo. Por lo tanto la última fila de la matriz *e* no se introducirá en la matriz *d.* Esto se debe a que si ambas matrices tienen el mismo tamaño y la matriz *d* tiene la primera fila ocupada. El paso de los bits de la matriz *e* a la matriz *d* se ejecuta mediante un bucle *FOR* anidado que recorreran todos los valores de la matriz *d* desde la fila uno hasta la *n* menos uno. La ecuación empleada es: *d(i)(j)=e(i-1)(j)*. La matriz *e* utiliza en el

argumento de las filas *i* menos uno por lo explicado anteriormente. Además la última fila de la matriz *e* se emplea en la Ilustración 68.

El codigo de la tercera parte se observa en la Ilustración 66.

```
connections6: FOR i IN 1 TO m-1 GENERATE
  connections7: FOR j IN 0 TO n-1 GENERATE
    d(i)(j) <= e(i-1)(j);
  END GENERATE;
END GENERATE;
```

Ilustración 66: Código conecctions6 y connections7

La parte 4 del algoritmo es utilizada para extarer los resultados obtenidos en los bloques de la diagonal derecha. Por lo tanto los valores que pertenecen a la columna cero de la matriz *f* se introducen en la mitada inferior del vector *z*. El codigo empleado ha sido un bucle *FOR* que va desde la fila cero hasta la n menos uno mateniendo constante el valor de la columna en cero de la matriz *f*. Al mismo tiempo el bucle recorrerá las columas del vector.

```
outputs: FOR j IN 0 TO m-1 GENERATE
  z(j) <= f(j)(0);
END GENERATE;
```

Ilustración 67: Código salida parte inferior del resultado.

La parte 5 realiza la última operación del algoritmo *CSA* mediante los bloques representados con una suma en la Ilustración 60. Para aplicar *CSA* se ejecutan dos sentencias previas, que tienen como almacenamiento las señales *firts_operand* y *second_operand.* La primera sentencia pasa los valores de la última fila de la matriz *f* al vector *firts_operand* (Valores de resultado). La segunda sentencia almacena la última fila de la matriz *e* (Valores de acarreo). Una vez se tienen estos dos valores se realiza la suma que da como resultado la mitad de mayor peso del resultado (*z*).

En el código el protocolo *CSA* se realiza mediante los sumadores totales que reducen la cantidad de valores en dos. Y para finalizar se realiza la suma entre ambos valores obtenidos.

```
first_operand<= f(m-1)(n-1 DOWNTO 1);
second_operand <= e(m-1);
z(n+m-1 DOWNTO m) <= first_operand + second_operand;
```

Ilustración 68: Sentencias finales.

Una vez obtenidos ambas partes del vector *z* se tendrá el resultado de la multiplicación.

4.3 Wallace Tree Multipilier o Multiplicador Árbol de Wallace

El *Wallace Tree* o Árbol de Wallace es un algoritmo de implementación hardware eficiente de multiplicación de números enteros. Este algoritmo el primero que lo implementó fue el científico informático Chris Wallace en el año 1964 [2].

El circuito de sumadores de Árbol de Wallace mejora el rendimiento de las multiplicaciones[26, Cap. 1]. Una mejora del rendimiento en la realización de las sumas de la multiplicación supone un mejor tiempo de respuesta.

El algoritmo del Árbol de Wallace tiene tres pasos principales que son [27]:

1- Multiplicar parcialmente cada bit de uno de los bits del multiplicando por cada bit del multiplicador.
2- Realizar varias etapas de suma hasta que en cada columna haya un máximo de dos productos parciales o resultados. Esto se realizará mediante sumadores totales o semisumadores.
3- Implementar un CLA para obtener el resultado final.

Los tres pasos que hay que realizar en el multiplicador de Wallace se representa con el siguiente diagrama de flujo.

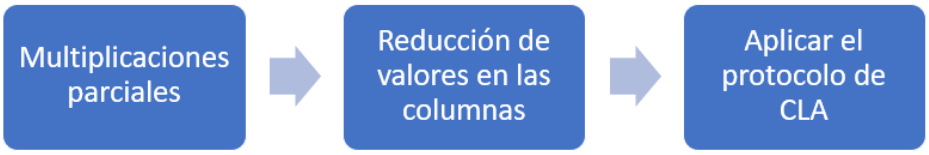

Ilustración 69: Diagrama de flujo de ejecución del Multiplicador de Árbol de Wallace.

Este algoritmo en el paso dos tiene unas reglas que no son estrictas. Por lo tanto, hay unas nociones generales que se deben tener para obtener el resultado.

En la Ilustración 70 se puede contemplar el diagrama del multiplicador de Árbol de Wallace de 8 bits, cada punto de la primera etapa representa a cada una de las multiplicaciones parciales obtenida de la multiplicación de cada bit del multiplicando por los bits del multiplicador.

Desde la primera etapa hasta la cuarta lo que se realiza es el paso dos del algoritmo que es reducir el número de sumandos hasta que hay un máximo de dos, esto se realiza mediante sumadores totales o semisumadores. En la etapa 5 se aplica el algoritmo de *CLA* para obtener el resultado final.

El paso dos se realizará de forma que se ejecutan sumadores totales, semisumadores o se pasan los valores a la siguiente etapa (en la Ilustración 11 son los valores que no están rodeados). Esto reduce la cantidad de valores por columna.

En cada etapa se generan nuevos resultados y acarreos que serán empleados en la siguiente etapa. Además de estos valores también puede haber valores arrastrados de etapas anteriores.

La diferencia entre las sumas realizadas en las etapas del paso 2 y las del paso 3, es que las sumas del paso 3 (CLA) se pasará el acarreo generado en una columna a la siguiente en la misma etapa y en el paso 2 los acarreos se pasarán a la siguiente etapa.

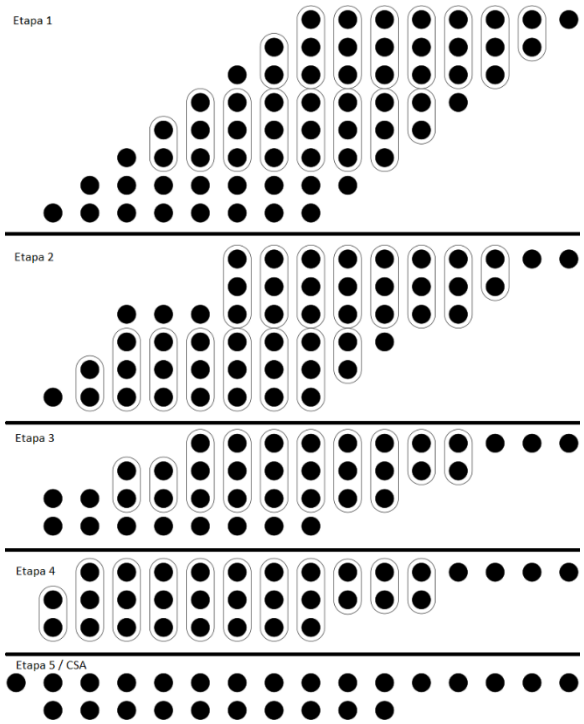

Ilustración 70: Diagrama del multiplicador de Árbol de Wallace

El código del multiplicador Árbol de Wallace que se ha desarrollado para el trabajo no se puede variar de tamaño directamente. La solución que se ha obtenido para poder modificar el tamaño del multiplicador es realizar un código en C++ que al ejecutarse elabore el código en VHDL con el tamaño de los coeficientes deseados.

El código elaborado en C++, cuando se ejecuta, solicita el tamaño del multiplicando y del multiplicador que se quiere para los coeficientes del Árbol de Wallace. Una vez introducido este valor la aplicación genera el código VHDL para el Árbol de Wallace de dicho tamaño.

En la pantalla de la aplicación una vez introducido el tamaño de los coeficientes se generará información relativa con el multiplicador realizado. Una muestra de ello es la Ilustración 71. Esta ilustración muestra la información generada de la etapa cero.

Como se ha mencionado, no hay unas reglas estrictas de cómo se han de realizar las sumas o la cantidad de sumas que se realizan por etapa y por columna. Por ello, si se comprueban la cantidad de sumas que se realizan o el tipo de suma entre la Ilustración 70 y la Ilustración 71 no coincide en todos los casos. Por ejemplo, en la columna siete (siendo la columna más a la derecha la cero) en la Ilustración 70 solo hay dos sumadores totales y en la Ilustración 71 se realizan dos sumas totales y una semisuma.

Las cabeceras de la Ilustración 71 se corresponden con: número de la columna "n", cantidad de bits que se tienen en la columna sin tener en cuenta los acarreos de la etapa anterior "numC", número de bits que se tiene teniendo en cuenta los acarreos "nbt", cantidad de sumas totales o *Full-adder* "fadder", cantidad de sumas parciales/medias o *Half-adder* "hadder" y cantidad de acarreos se tienen para la siguiente etapa "carrys".

```
Tamano de los datos a multiplicar: 8


 Etapa 0
n       numC    nbt     fadder  hadder  carrys
0       1       1       0       0       0
1       2       2       0       1       1
2       3       3       1       0       1
3       4       4       1       0       1
4       5       5       1       1       2
5       6       6       2       0       2
6       7       7       2       0       2
7       8       8       2       1       3
8       7       7       2       0       2
9       6       6       2       0       2
10      5       5       1       1       2
11      4       4       1       0       1
12      3       3       1       0       1
13      2       2       0       1       1
14      1       1       0       0       0
15      0       0       0       0       0
```

Ilustración 71: Recorte aplicación de generación código VHDL.

4.3.1 Código VHDL del multiplicador de Árbol de Wallace.

El código que se ha desarrollado para describir los circuitos del multiplicador de Árbol de Wallace no tiene una estructura regular. Esto depende del tamaño de los datos a multiplicar. A continuación, se desarrollará y comentará el código que se genera para un multiplicador de Árbol de Wallace de tamaño de 8 bits.

El diagrama que resume el circuito que describe el Árbol de Wallace se muestra en la Ilustración 72.

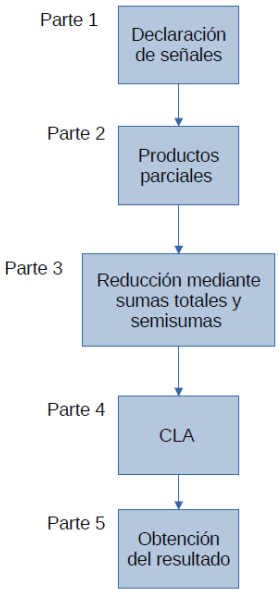

Ilustración 72: Diagrama flujo de la descripción del circuito del multiplicador de Árbol de Wallace, dividido en partes.

Como en los otros dos multiplicadores, en este se definen dos entradas (coeficientes) y una salida (resultado).

La parte 1 del código contiene la definición de todas las señales parciales (cada punto de la Ilustración 70) y de los dos componentes empleados que son el sumador total y el semisumador.

El componente sumador total o *F_adder* en el código tiene cinco argumentos de los cuales los dos primeros corresponden con las salidas (resultado de la suma y su acarreo) y los otros tres corresponden con las entradas que son los valores a sumar.

Todos los componentes del sumador total tienen el siguiente identificador Fn. n es el número de suma total que se realiza.

El componente semisumador o *H_adder* en el código tiene cuatro argumentos dos son de salida y dos de entrada. Los dos primeros argumentos son el resultado y el acarreo generado. Los siguientes dos son los valores que se sumarán.

Todos los componentes del semisumador tienen el siguiente identificador Hn. Siendo n el número de semisuma que se realiza.

```
H1:H_adder port map(a1_4_2, a1_5_1,a0_4_3,a0_4_4);
F3:F_adder port map (a1_5_2, a1_6_0,a0_5_0,a0_5_1,a0_5_2);
```

Ilustración 73: Recorte código multiplicador del Árbol de Wallace, semisumador y sumador total

Seguido de la definición de los componentes empleados se definen las señales que se emplearán en el multiplicador. La cantidad de señales que se definen depende del tamaño de los coeficientes. Se pueden diferenciar entre dos tipos de señales las que tienen una estructura "*ax_x_x*" y las "*RBx*" (siendo las *x* valores numéricos), ambas se definen como bit.

Las señales *RBx* se emplean para almacenar temporalmente los valores obtenidos de la multiplicación y cuando se tienen todos estos valores pasarlos a la variable de salida "*Resultado*". Dependiendo del tamaño del multiplicador se tendrá una cantidad u otra.

Se realiza también la declaración de las señales del protocolo CLA que son: "primer_operando", "segundo_operando" y "resultado_operando".

Una vez acabada la definición de las señales y componentes se comienza la descripción del circuito.

La descripción del circuito se puede dividir en cuatro partes. Las tres primeras partes son en las que se divide el multiplicador de Árbol de Wallace y la última es aplicar el protocolo CLA. Las partes en las que se ha dividido se pueden observar en la Ilustración 72.

La parte 2 realiza las multiplicaciones parciales de los coeficientes. En esta parte se realiza la operación lógica *AND* de los bits del multiplicando por los

bits del multiplicador. El resultado de esa operación lógica se almacenará en las señales definidas con el identificador *a0_x_x*. En el primer dígito se dará el valor cero debido a que es la primera etapa, los otros dos dígitos dependerá de la posición de los bits que se hallen en esa columna.

```
a0_0_0 <= DatoA(0) and DatoB(0);
a0_1_0 <= DatoA(1) and DatoB(0);
a0_2_0 <= DatoA(2) and DatoB(0);
a0_3_0 <= DatoA(3) and DatoB(0);
a0_4_0 <= DatoA(4) and DatoB(0);
```

Ilustración 74: Recorte del código multiplicación Árbol de Wallace. Zona multiplicaciones parciales

En la parte 3 del código se encuentra la ejecución de las sumas para reducir la cantidad de valores en las columnas. Dependiendo del tamaño de los coeficientes se tendrán más etapas o menos en esta parte. En este fragmento de código se emplean cuatro tipos de sentencias: el componente sumador total (*f_adder*), el componente semisumador (h_*adder*), pasar el valor de una señal *ax_x_x* a otra y pasar el valor de una variable *ax_x_x* a una señal *RBx*.

A la hora de aplicar la reducción de cantidad de valores hay dos formas de actuar. Cuando se encuentra en la mitad de menos peso de los bits se realizan las sumas sin ningún tipo de condición. Pero en la parte superior se tendrán tres situaciones para realizar las sumas.

- Cuando se tienen dos bits en una columna no se realizará ninguna suma
- Cuando se tengan tres bits en la columna se realizará una semisuma
- Cuando se tengan más de tres bits en la columna se agruparán los bits de la forma normal.

El sumador total se utiliza cuando en una columna se encuentran tres o más valores. El semisumador se emplea cuando en una columna quedan dos valores por sumar. Estas sumas almacenan los valores obtenidos en señales *ax_x_x*. Las señales *ax_x_x* no empleadas se pasan a la siguiente etapa. El último caso, se usa en la situación que se realiza una única suma, se encuentra en la parte inferior de las columnas y no tiene acarreo de la fila anterior en la siguiente etapa. Al valor obtenido de la suma que será *ax_x_0* se le pasa a una señal de salida *RBx.* Un ejemplo de este último caso se puede observar en la Ilustración 75.

```
 -- Etapa 2
H8:H_adder port map(a3_3_0, a3_4_0,a2_3_0,a2_3_1);
RB3 <= a3_3_0;
```

Ilustración 75: Recorte código multiplicador Árbol de Wallace, etapa 2 obtención suma.

La sentencia de pasar un valor de una señal *ax_x_x* a otra se emplea cuando queda restante un valor en una columna o quedan dos valores en las columnas de la parte superior.

```
H12:H_adder port map(a3_12_1, a3_13_0,a2_12_0,a2_12_1);
a3_12_2 <= a2_12_2;
a3_13_1 <= a2_13_0;
a3_13_2 <= a2_13_1;
```

Ilustración 76: Recorte código multiplicador Árbol de Wallace,

En la Ilustración 76 se pueden observar dos situaciones. En el primer caso la columna doce contiene tres valores, pero como se muestra solo se suman dos de ellos. Esto se debe a que se encuentra en la mitad de mayor peso de los bits.

El segundo caso corresponde a la columna trece que contiene dos valores que se pasan a la siguiente etapa (como se encuentra en las columnas de la parte superior no se realizan sumas). Los valores de la columna trece no pasan con el mismo valor del identificador a la etapa siguiente debido a que en la suma H12 el acarreo pasa a la siguiente columna (columna trece) con el valor cero desplazando los valores que se pasan en la columna trece.

La parte 4 del código realizado en VHDL es la parte final donde se ejecuta el protocolo *CLA*. En esta parte se concatenarán todas las señales *ax_x_0* para obtener la señal vector "primer_operando". Del mismo modo se concatenarán las señales *ax_x1* para obtener la señal vector "segundo_operando".

Una vez obtenidas ambas señales se realiza la suma entre ellas que se almacena en la señal "resultado_operando". El código que describe esta parte del circuito se muestra en la Ilustración 77.

```
primer_operando <= a4_15_0 & a4_14_0 & a4_13_0 & a4_12_0 & a4_11_0 & a4_
segundo_operando <= a4_14_1 & a4_13_1 & a4_12_1 & a4_11_1 & a4_10_1 & a4
resultado_operando <= primer_operando + segundo_operando;
```

Ilustración 77: Recorte código multiplicador Árbol de Wallace, Parte CLA

La parte 5 es la encargada de concatenar la señal "resultado_operando" con las señales RBx obtenidas durante la reducción. La descripción del circuito se observa en la Ilustración 78

```
resultado <= resultado_operando & RB4 & RB3 & RB2 & RB1 & RB0;
```

Ilustración 78: Recorte de código Árbol de Wallace, obtención de resultado final

4.3.2 Código en C++ para la generación del código VHDL

Como se ha comentado para poder realizar las simulaciones, síntesis e implementaciones de este código cambiando el tamaño de los coeficientes se necesita una aplicación que genere el código VHDL. Dicha aplicación se ha desarrollado mediante C++, y su función es generar el código del multiplicador de Árbol de Wallace del tamaño que se requiera.

Este código se compone de cuatro funciones que se agrupan en la función *main* (función base). Estas funciones son: *generar_senales_parciales, generar_senales, multiplicaciones_parciales y generar_ sumas.* Dos de las

funciones contienen un algoritmo que es primordial para la generación del código. A su vez es empleado para generar la información del multiplicador que se muestra por pantalla cuando se ejecuta. La información que se muestra se ha podido observar en la Ilustración 71.

El código utiliza tres variables que se aprovecharán como argumento de las funciones auxiliares. La variable principal "*nbits*" será el tamaño del multiplicador que se le indicará en la ejecución. A partir de esta variable se obtendrán las otras dos variables: *nbitsM* (es el tamaño indicado menos uno, esto se debe a que se comienza desde cero) y *nbitsZ* (tamaño del resultado, dos veces el tamaño de los coeficientes menos uno, igual que en el caso anterior empezando desde cero).

En C++ se debe declarar la librería *fstream,* empleada para poder escribir en archivos externos. Una vez definida la librería se debe declarar el comando que se va a emplear para escribir en el archivo. El comando empleado es *myfile*. En la Ilustración 79 se muestra la función *main()* que es la función principal y que contiene las llamadas a las demás funciones auxiliares.

```
int nbits, nbitsM, nbitsR;
 //Petición de número de bits
cout << "Numeros de bits de los datos a multiplicar: ";
cin >> nbits;
// Código bloc de notas
int calculo, resultados, f_sum, h_sum;
int carrys=0;
int carrys_a=0;

//def bits
nbitsM=nbits-1;
nbitsR=nbits*2-1;

// Código VHDL
myfile.open ("Codigo5_1.txt");
myfile << "library IEEE;\n";
myfile << "use IEEE.STD_LOGIC_1164.ALL;\n";
myfile << "USE IEEE.STD_LOGIC_ARITH.ALL;\n";
myfile << "USE IEEE.STD_LOGIC_UNSIGNED.ALL;\n\n";

myfile << "entity Multiplicador is\n";
myfile << "Port (  DatoA, DatoB : in STD_LOGIC_VECTOR(" << nbitsM << " DOWNTO 0);\n";
myfile << "     Resultado : out STD_LOGIC_VECTOR(" << nbitsR << " DOWNTO 0));\n";
myfile << "end Multiplicador;\n\n";
myfile << "architecture Behavioral of Multiplicador is\n";

myfile << "component F_adder is\n";
myfile << "port(  res : out STD_LOGIC;\n";
myfile << "     carry : out STD_LOGIC;\n";
myfile << "     a : in STD_LOGIC;\n";
myfile << "     b : in STD_LOGIC;\n";
myfile << "     c : in STD_LOGIC);\n";
myfile << "end component;\n";
myfile << "component H_adder is\n";
myfile << "port(  res : out STD_LOGIC;\n";
myfile << "     carry : out STD_LOGIC;\n";
myfile << "     a : in STD_LOGIC;\n";
myfile << "     b : in STD_LOGIC);\n";
myfile << "end component;\n";


generar_senales_parciales (nbits);
generar_senales(nbitsR, nbitsM);


myfile << "\nbegin\n";

multiplicaciones_parciales(nbits);
myfile << "\n\n -- sumatoiros \n\n";
generar_sumas(nbits);
```

Ilustración 79: Recorte de código main.cpp, función main().

4.3.2.1 Algoritmo

Como se ha mencionado este algoritmo es empleado para generar información de la multiplicación de Árbol de Wallace sobre la que se está realizando la descripción del circuito. La principal ocupación de la información generada no es mostrar la información, si no utilizarla en las funciones de generar señales y generar sumas.

La generación de información tiene dos flujos que dependen de la cantidad de valores máximos que se encuentran en las columnas del multiplicador. El diagrama que resume estos flujos se muestra en la Ilustración 80.

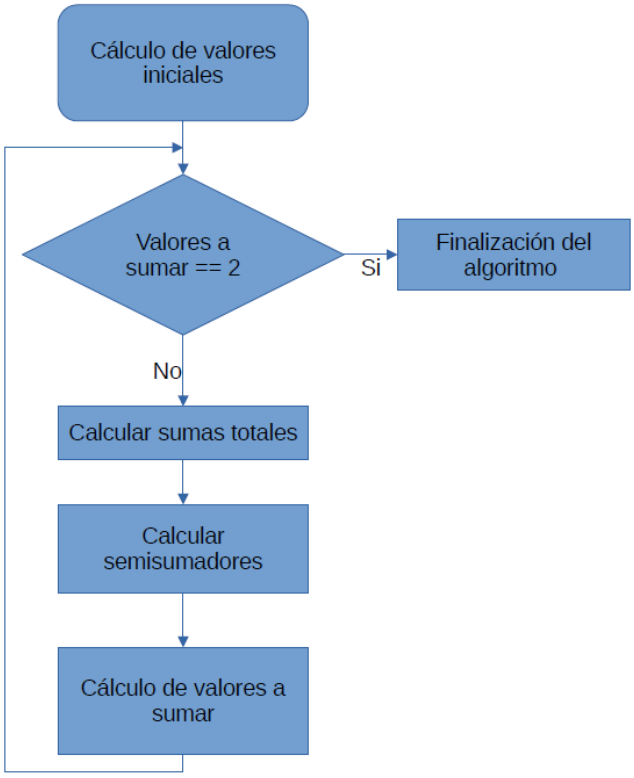

Ilustración 80: Diagrama de flujo del algoritmo.

El algoritmo generará la información inicial del multiplicador, después se comprobará cuál es la cantidad de valores máximos que hay en todas las columnas.

En caso de que la cantidad máxima de valores sea de dos, el algoritmo finalizará, en caso negativo se realizará el cálculo de sumadores totales, semisumadores y el cálculo de los valores a ejecutar.

Como se observa en el diagrama de flujo se tiene un bucle, cada vez que se pase por la comprobación se tendrá una nueva etapa del multiplicador.

Para obtener los sumadores totales será el valor entero de la división de los valores a sumar entre tres. Se tiene una excepción y es que si el valor se

encuentra en la mitad superior y solo se suman tres valores se asigna un semisumador.

El cálculo de los semisumadores será el valor entero de la división de los valores restantes del cálculo de sumadores totales (resto de dicha división) entre dos. Tiene una excepción y es que cuando solo se tienen dos valores a sumar en la mitad superior no se realiza suma.

Después de los cálculos de los sumadores se obtendrá la cantidad de sumandos que se van a tener en la siguiente etapa volviendo a la comprobación.

4.3.2.2 Funciones empleadas en el código C++

4.3.2.2.1 Función main()

Función que sirve como punto inicial para la ejecución del programa. Dentro de este código además de ejecutar las sentencias que se le declaran también se realiza la llamada a otras funciones. El código completo que se emplea en esta se muestra en la Ilustración 79.

La función *main* comienza con la petición del tamaño de los coeficientes y almacenando el valor en la variable *nbits*. Después este valor se trata para obtener los valores de las variables *nbitsM* y *nbitsR*, mediante las sentencias que se observan en la Ilustración 81.

```
nbitsM=nbits-1;
nbitsR=nbits*2-1;
```

Ilustración 81: Recorte de código main.cpp, obtención de variables.

Acto seguido se comienza con la apertura del archivo donde se generará la descripción del multiplicador y la declaración de las librerías empleadas en VHDL.

A continuación, se realiza la declaración de la entidad en el archivo. En estas sentencias se encuentra la definición de los datos de entrada (*DatoA* y *DatoB*) empleando la variable *nbitsM* y el de salida (*Resultado*) que emplea la variable *nbitsR*. La declaración de la entidad se puede comprobar en la Ilustración 82.

```
myfile << "entity Multiplicador is\n";
myfile << "Port (  DatoA, DatoB : in STD_LOGIC_VECTOR(" << nbitsM << " DOWNTO 0);\n";
myfile << "     Resultado : out STD_LOGIC_VECTOR(" << nbitsR << " DOWNTO 0));\n";
myfile << "end Multiplicador;\n\n";
```

Ilustración 82: Recorte de código main.cpp, declaración de la entidad.

Posteriormente, se comienza con la definición de la arquitectura del archivo VHDL. En esta declaración se encuentran la definición de los componentes *F_adder* (sumador total) y *H_addder* (Semisumador) y la declaración de las señales empleadas. En este punto se comienza a emplear las funciones auxiliares de la generación de señales (*generar_senales, generar_seanles_parciales*) que se explicaran en los siguientes apartados.

Una vez ejecutada la declaración de las señales se comienza con la parte que genera la descripción del circuito. En esta, se emplearán funciones para la obtención de los valores parciales y la definición de las sumas. Las funciones que se emplearán son *multiplicaciones_parciales* y *generar_suma.* Las dos funciones representan los tres pasos a seguir en el desarrollo del multiplicador de Árbol de Wallace mencionados en la Ilustración 69. La función *multiplicaciones_parciales* representa la obtención de los productos parciales, así como la función *generar_sumas* contiene los pasos dos y tres que se ejecutan en el multiplicador de Árbol de Wallace.

4.3.2.2.2 Función generar_senales_parciales

Generar_senales_parciales se emplea para generar las señales que almacenarán los valores obtenidos de las multiplicaciones parciales entre el multiplicando y el multiplicador, cuya definición se realiza mediante un bucle *FOR* anidado.

La cantidad de señales que hay por columna en el multiplicador aumenta desde uno en la columna de menor peso hasta el valor del tamaño de los coeficientes, a partir de ese punto disminuirá hasta uno. Como se observa en la primera etapa de la Ilustración 70.

En la definición de las señales se le dará el valor de etapa cero, la columna vendrá definida por la suma de los bucles y la posición del valor depende de la columna.

```
for (int i=0; i<=nbitsM; i++)
{
    for(int j=0; j <= nbitsM; j++)
        {
                if( (i+j) <= nbitsM )
                {
                    myfile << " signal a0_"<< (i+j) << "_" << i <<" : std_logic;\n"; /
                }
                else
                {
                    int id;
                    id=(i+nbitsM)-(i+j);
                    myfile << " signal a0_"<< (i+j) << "_" << id <<" : std_logic;\n";
                }
```

Ilustración 83: Recorte de código main.cpp, declaración de señales parciales.

4.3.2.2.3 Función generar_senales

La función que me ocupa realiza la declaración de las señales que se emplearán en el código VHDL utilizadas en los sumadores, así como las señales del protocolo CLA y de salida intermedias (RBx). Esta maneja información generada por el algoritmo (4.3.2.1).

Para la declaración de las señales empleadas en los sumadores, se ha llevado a cabo utilizando la cantidad de valores a sumar en cada etapa.

La declaración de las señales se consigue añadiendo al flujo principal del algoritmo las sentencias de código que se muestran en la Ilustración 84.

```
for (int r = 0; r < matrix2[t][2]; r++)
{
    myfile << " signal a"<< etapa << "_" << t << "_" << r <<" : std_logic;\n";
}
```

Ilustración 84: Recorte de código main.cpp, generar señales.

Las señales del protocolo CLA y señales RBx se realizan una vez la cantidad de valores a sumar en el algoritmo es igual a dos, por lo tanto, en el diagrama de la Ilustración 80 se ubica en "finalización del algoritmo".

Para la declaración de las señales RBx se ha tenido en cuenta las etapas realizadas del flujo principal del algoritmo. Cada etapa se realiza la declaración de una señal RBx empezando desde cero.

Para la declaración de las señales del protocolo CLA también se ha tenido en cuenta las etapas del flujo principal del algoritmo ejecutadas, dado que al tamaño de los operandos tendrá el tamaño del resultado menos la etapa que se encuentre más uno (*nbitsZ*-etapa+1).

4.3.2.2.4 Función multiplicaciones_parciales

Las multiplicaciones_parciales se emplean para realizar la parte de la descripción del circuito en la que se declaran las multiplicaciones de los bits de los multiplicando por los del multiplicador.

En esta parte realiza la declaración las sentencias del código VHDL, que ejecutan la operación lógica *AND* entre los bits del *datoA* y los del *datoB*. Dicha operación se almacena en las señales definidas como parciales. El código empleado es el mostrado en la Ilustración 85.

```
id=nbitsM-j;
myfile << " a0_"<< (i+j) << "_" << id << " <= DatoA("<< j <<") and DatoB(" << i <<");\n";
```

Ilustración 85: Recorte de código main.cpp, Sentencia declaración multiplicaciones parciales, parte superior.

4.3.2.2.5 Función generar_sumas

Esta función es la encargada de generar los sumadores y aplicar el protocolo CLA. Para ello emplea el algoritmo (4.3.2.1).

Esta se diferencia en dos partes: la primera define los sumadores, correspondiendo con la parte cíclica del algoritmo y una segunda parte, la aplicación del protocolo CLA, que se aplica cuando se finaliza el algoritmo.

Dentro de la ejecución cíclica del algoritmo además de realizar la declaración de los sumadores se realiza la asignación de las señales no empleadas para la siguiente etapa, así como las señales que se obtienen como resultado final se asignan a las señales RBx.

Los sumadores totales se declaran cuando se ha obtenido en el algoritmo su realización, del mismo modo ocurre con los semisumadores. El código que se muestra en la Ilustración 86 corresponde con la declaración de un

semisumador. La declaración se hará tantas veces como se ha calculado en el algoritmo.

```
if(h_sum!=0)
{
    for(h; h < f_sum+h_sum; h++)
    {
        myfile << "H" <<sumatorioHA <<":H_adder port map(a" << etapa+1 << "_" << t ;
        myfile<< "_" << h+matrix1[t-1][5] <<", a" << etapa+1 << "_" << t+1 << "_" << h;
        for(k; k< 2+k1; k++)
                {
                    myfile << ",a" << etapa << "_" << t <<"_" << k;
                }
        k1=k;
        myfile <<");\n";
        sumatorioHA+=1;
    }
```

Ilustración 86: Recorte de código main.cpp, algoritmo de declaración de semisumas.

Las señales que no han sido empleadas en los sumadores se pasarán a la siguiente etapa.

Las señales resultado que se van obteniendo en la reducción de la cantidad de valores se van pasando a las señales RBx (resultados parciales). Se define una señal como resultante cuando en una columna del multiplicador no se realizan más operaciones de suma, quedando solamente un valor. Como, por ejemplo, el semisumador de la etapa cero de la segunda columna por la derecha de la Ilustración 70.

Una vez se ha reducido la cantidad máxima de valores a dos se realizará la preparación y el protocolo CLA y la obtención del resultado.

La preparación del algoritmo consiste en declarar la asignación de las señales generadas como acarreos en los sumadores a la señal vector “primer_operando” y las generadas como resultado a la señal vector “segundo_operando”. Estas dos señales se consiguen generando la concatenación de las señales ax_x_x mediante bucles *FOR*.

Después se realiza la sentencia que declara la suma de ambas, seguido a esto se ejecuta la sentencia que concatena el valor obtenido de la suma y los resultados parciales RBx que se asigna a la señal de salida Resultado.

```
myfile << "resultado <= resultado_operando";
for(int g= etapa; g>=0;g--)
{
    myfile << " & RB"<<g;
}

myfile << ";\n";
```

Ilustración 87: Obtención de resultado.

En el código mostrado en la Ilustración 87 se observa las sentencias de obtención del resultado que se concatenará el resultado del protocolo CLA y los valores RBx. Cada etapa se ha realizado de la parte cíclica se obtiene un resultado RBx, así el bucle de definición que concatena estas tendrá como valor inicial los ciclos realizados (etapas en el código) hasta cero.

4.4 Comparación de los recursos hardware y tiempo de respuesta obtenido a partir de la sintetización realizada por Xilinx

Para comparar los valores obtenidos con los distintos multiplicadores analizados se debe aplicar una restricción a las opciones de síntesis e implementación. Esta restricción es que el compilador de Xilinx no emplee los módulos DSP. Esta se aplica porque en la síntesis e implementación de la multiplicación básica el compilador recurre a estos módulos que son empleados para cálculos matemáticos intensos entre los que se encuentran las multiplicaciones. Para poder comparar entre los tres tipos de multiplicadores se aplica la restricción.

Se ha observado que una vez que la síntesis supera la capacidad máxima de entradas y salidas o de LUTs no se puede realizar la implementación. Por lo que se va a comparar, son los datos obtenidos de la síntesis.

La cantidad máxima de LUTs que se pueden emplear en la FPGA Artix-7 son 20800 y la cantidad máxima de entradas y salidas útiles son 106 (en la placa Basys 3 se dispone de una cantidad muy inferior de entradas).

La cantidad de LUTs empleadas clasificadas por tipo de multiplicador y tamaño de los datos a multiplicar se pueden observar en la Tabla 1.

	Deschamps	Wallace	Xilinx
4 bits	15 (0,072%)	17 (0,086%)	16 (0,077%)
8 bits	72 (0,34%)	86 (0,41%)	70 (0,33%)
16 bits	405 (1,94%)	395 (1,89%)	337 (1,62%)
32 bits	1864 (8,96%)	1575 (7,57%)	1313 (6,31%)
64 bits	9237 (44,40%)	6479(31,14%)	5579 (26,82%)
128 bits	36655 (176,22%)	25882 (124,43%)	22652 (108,90%)

Tabla 8: Datos obtenidos de uso de LUTs (porcentaje de LUTs empleadas)

La cantidad de LUTs que emplean los diferentes multiplicadores solo se excede cuando el tamaño de los datos a multiplicar es de 128 bits.

Con los resultados obtenidos de LUTs empleadas se observa que el compilador de Xilinx emplea menos LUTs cuando se realiza el multiplicador básico (el algoritmo de multiplicación lo realiza el compilador) que cuando se aplican los multiplicadores de Árbol de Wallace o multiplicador paralelo CSA.

Para multiplicadores de tamaño de los coeficientes pequeños (4 y 8 bits) la cantidad de LUTs empleadas es similar para los tres multiplicadores. A partir del tamaño de 16 bits se observa una diferencia de cantidad de LUTs empleadas en los distintitos multiplicadores que aumenta con el tamaño. La diferencia de LUTs empleadas entre el multiplicador de Árbol de Wallace y el sintetizado por Xilinx que la del multiplicador paralelo CSA.

Se observa que el multiplicador planteado por Deschamps emplea más recursos hardware que los otros dos multiplicadores. El multiplicador de Árbol

de Wallace y el sintetizado por Xilinx mantienen la diferencia de LUTs empleadas en torno al 1,15 %.

A continuación, se analizará el tiempo de respuesta más lento que se tiene en cada uno de los multiplicadores para cada tamaño de los coeficientes. El tiempo de respuesta depende de la transmisión de las señales que se transmiten mediante las nets más el retardo máximo de configuración de las LUTs.

Los retardos obtenidos se miden en nanosegundos (*ns*). Los datos obtenidos de los tres métodos de multiplicación se observan en la Tabla 2.

	Deschamps	Wallace	Xilinx
4 bits	7.038	7.123	6.933
8 bits	10.585	9.390	9.786
16 bits	15.693	11.424	12.332
32 bits	27.132	13.836	18.040
64 bits	51.223	16.540	23.528
128 bits	125.155	20.011	31.136

Tabla 9: Datos del tiempo de respuesta en la obtención de los resultados.

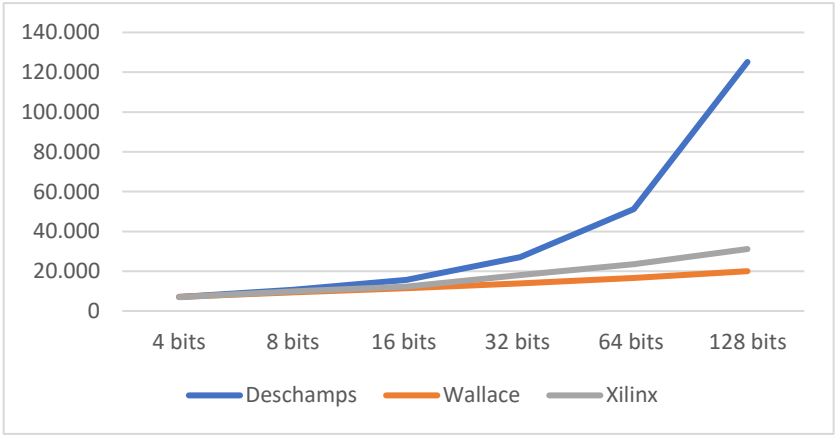

Ilustración 88: Gráfica de tiempos de respuesta.

Tanto en la Tabla 9 como en la Ilustración 88 se puede observar que con el aumento del tamaño los tiempos de respuesta se aumentan.

Analizando los tiempos obtenidos del multiplicador paralelo CSA planteado por Deschamps se observa que inicialmente el tiempo aumenta poco, pero para multiplicadores de mayor tamaño de 32 bits el aumento del tiempo es notable llegando a más del doble entre el multiplicador de 64 bits y 128 bits.

En el multiplicador de Árbol de Wallace desarrollado en este trabajo se observa que los tiempos tienen un aumento constante en torno al 23%. Por lo que el aumento de tiempo es lineal.

Por su parte, en el multiplicador sintetizado por Xilinx se observa un aumento de tiempo del 35% según aumenta el tamaño, aunque el aumento del tiempo no es constante.

Para multiplicadores de 4 y 8 bits se obtienen unos resultados de hardware empleado y tiempo de respuesta similares. Pero para multiplicadores de mayor tamaño las diferencias van siendo más notables.

Comprobando la información obtenida en la Tabla 8 y Tabla 9 se demuestra que el multiplicador más desfavorable con respecto a los datos de hardware empleado y tiempo de respuesta es el multiplicador paralelo CSA planteado por Deschamps. Llegando a una diferencia de casi once mil LUTs empleadas con el multiplicador de Árbol de Wallace y a más de cien nanosegundos en el tiempo de respuesta.

Observando los datos entre el multiplicador de Árbol de Wallace y el sintetizado por Xilinx se ve que son dos circuitos de multiplicación con datos muy parejos para los tamaños analizados. Siendo la diferencia de LUTs empleadas de tres mil, y una diferencia de tiempo de respuesta de diez nanosegundos.

Aunque el multiplicador de Árbol de Wallace emplea más elementos hardware que el sintetizado por Xilinx, el de Wallace realiza la multiplicación más rápido. Con ello se descarta que el mayor uso de LUTs significa más tiempo de respuesta, si no que dependen de cómo están organizadas.

Como no se conoce la arquitectura desarrollada por la sintetización de Xilinx no se podrá comparar con los otros dos a nivel de estructura.

La principal diferencia entre el multiplicador de Árbol de Wallace y el multiplicador paralelo CSA es el tiempo de respuesta. En los multiplicadores de hasta 32 bits, el hardware empleado es similar pero los tiempos son muy dispares llegando a casi el doble de tiempo de respuesta.

Esto se debe a cómo se organizan las sumas en las arquitecturas de los multiplicadores. En el multiplicador paralelo las sumas se van realizando fila a fila como se observa en la Ilustración 60. Por lo que la siguiente fila tiene que esperar a que se calculen los resultados y los acarreos de la anterior. En cambio, el multiplicador de Árbol de Wallace realiza todas las sumas posibles para columna, realizándose en paralelo lo que elimina el retardo de tiempo en esperar el resultado y acarreo de las filas.

Por lo tanto, el mejor multiplicador en empleo hardware como se ha observado es el multiplicador sintetizado por Xilinx y el mejor en tiempo de respuesta es el multiplicador de Árbol de Wallace.

La relación entre ambos multiplicadores es similar y por lo tanto daría igual cuál implementar en la ALU del RISC-V (3.1.4). Se ha decidido implementar el desarrollado por Xilinx por la ocupación del hardware dado que emplea menos recursos y el tiempo es similar cuando se emplean datos de 32 bits de tamaño.

Capítulo 5: Conclusiones

En este Trabajo Final de Grado se ha realizado la implementación de un procesador RISC-V monociclo de 32 bits con la extensión para multiplicaciones y divisiones sobre la FPGA Artix-7.

En primer lugar, se ha conseguido el primero de los objetivos que era el diseño la arquitectura RISC-V IM32 monociclo. La comprobación del procesador se ha verificado mediante las pruebas del apartado 3.3.4 y con ello se ha verificado el correcto diseño realizado de la descripción del circuito del procesador.

El diseño actual del procesador puede ser un punto de partida para trabajos futuros incrementando las capacidades del procesador RISC-V IM32 como pueden ser la ejecución multiciclo o la adicción de otros de sus módulos.

En segundo lugar, también se ha alcanzado el segundo objetivo que era el diseño del Algoritmo de Multiplicación de Árbol de Wallace de tamaño de datos variable. Para ello se ha realizado un algoritmo en C++ que genera el código VHDL de cada uno de los tamaños del multiplicador de Árbol de Wallace. Así como la comprobación con otros dos circuitos de multiplicación.

La conclusión que se ha llegado de este segundo objetivo es que dejando al compilador de Xilinx realizar la sintetización del circuito se consigue el diseño más optimo.

El objetivo principal de este trabajo era la realización la implementación sobre la Basys 3 de la arquitectura del procesador RISC-V IM32 con el módulo de multiplicación y división. Así como en el diseño del módulo emplear el circuito de multiplicación que tuviera la mejor relación tiempo de respuesta y uso de elementos hardware. Este objetivo ha sido alcanzado dado que era la combinación de los dos primeros objetivos combinados en uno. La implementación se puede comprobar en las pruebas mostradas en el apartado 3.3.4.

Bibliografía

[1] «Informacion_de_referencia_2_ISE3_4_2.pdf». Accedido: 2 de julio de 2022. [En línea]. Disponible en: http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion_de_referencia_2_ISE3_4_2.pdf

[2] «Los superordenadores x86 y ARM ya tienen rival RISC-V: Monte Cimone», *El Chapuzas Informático*, 10 de junio de 2022. https://elchapuzasinformatico.com/?p=467544 (accedido 2 de julio de 2022).

[3] «Intel invertirá 200M€ en Barcelona para diseñar chips basados en RISC-V», *El Chapuzas Informático*, 1 de junio de 2022. https://elchapuzasinformatico.com/2022/06/intel-invertira-200me-en-barcelona-para-disenar-chips-basados-en-risc-v/ (accedido 2 de julio de 2022).

[4] «Will Intel abandon x86 for RISC-V for its next-gen supercomputing chips?», *TechRadar*, 4 de junio de 2022. https://www.techradar.com/news/will-intel-abandon-x86-for-risc-v-for-its-next-gen-supercomputing-chips (accedido 2 de julio de 2022).

[5] «Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics», p. 64, 2022.

[6] «Basys 3 Artix-7 FPGA Board - Lógica Programable». https://sites.google.com/site/logicaprogramable/calculadoras/fpga/basys-3-artix-7-fpga-board (accedido 21 de febrero de 2022).

[7] «Basys 3™ FPGA Board Reference Manual». Digilent, 8 de abril de 2016. [En línea]. Disponible en: https://digilent.com/reference/_media/basys3:basys3_rm.pdf

[8] S. M. Trimberger, «Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology», *Proc. IEEE*, vol. 103, n.º 03, pp. 318-331, abr. 2015, doi: 10.1109/JPROC.2015.2392104.

[9] A. Barkalov, L. Titarenko, M. Kolopienczyk, K. Mielcarek, y G. Bazzydlo, *Logic Synthesis for FPGA-Based Finite State Machines*, vol. 38. Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland: Springer. [En línea]. Disponible en: http://www.springer.com/series/13304

[10] H. Muthumala Waidyasooriya, M. Hariyama, y K. Uchiyama, *Design of FPGA-Based Computing Systems with OpenCL*, 1.ª ed. Springer, 2018. [En línea]. Disponible en: https://link.springer.com/book/10.1007/978-3-319-68161-0

[11] digitaltagebuch, «FPGA», *My digital designing diary*. https://digitaltagebuch.wordpress.com/category/fpga/ (accedido 11 de mayo de 2022).

[12] David Patterson y Andrew Waterman, *Guía práctica de RISC-V*, Primera. [En línea]. Disponible en: http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf

[13] A. Waterman y K. Asanovic, Eds., «The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA». 13 de diciembre de 2019.

[14] «History», *RISC-V International*. https://riscv.org/about/history/ (accedido 29 de marzo de 2022).

[15] «RISC-V Introduction». Accedido: 29 de marzo de 2022. [En línea]. Disponible en: https://docs.google.com/presentation/d/1jqmAgj-5iimymT6jZJSViEnIm3Dfgs7B-LWEdW90UjU
[16] V. A. Pedroni, *Circuit Design with VHDL, second edition*. The MIT Press, 2010.
[17] F. Machado Sánchz y S. Borromeo López, *Diseño de circuitos digitales con VHDL*. Móstoles, Madrid, España, 2010.
[18] M. Sanchez Elez, «Introducción a la programación en VHDL». julio de 2014.
[19] «RISC: Qué es y cómo funciona esta arquitectura de procesadores», *Profesional Review*, 17 de julio de 2021. https://www.profesionalreview.com/2021/07/17/que-es-risc/ (accedido 30 de mayo de 2022).
[20] «2.2.2 Arquitectura CISC RISC y X86 - fhwjesus2». https://sites.google.com/site/fhwjesus2/home/procesador-cpu/componentes-cpu/2-2arquitectura-cisc-risc (accedido 30 de mayo de 2022).
[21] «RISC-V: toda la información de la arquitectura», *Profesional Review*, 22 de abril de 2022. https://www.profesionalreview.com/2022/04/22/risc-v/ (accedido 7 de junio de 2022).
[22] D. Ungar, R. Blau, P. Foley, D. Samples, y D. Patterson, «Architecture of SOAR: Smalltalk on a RISC», *ACM SIGARCH Comput. Archit. News*, vol. 12, n.º 3, pp. 188-197, jun. 1984, doi: 10.1145/773453.808182.
[23] «4 things IT leaders should know about RISC-V». https://enterprisersproject.com/article/2022/5/4-things-it-leaders-should-know-about-risc-v (accedido 7 de junio de 2022).
[24] «guia-practica-de-risc-v-1.0.5.pdf». Accedido: 21 de febrero de 2022. [En línea]. Disponible en: http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf
[25] J.-P. Deschamps, G. D. Sutter, y E. Cantó, *Guide to FPGA Implementation of Arithmetic Functions*. Dordrecht, Heidelberg: Springer, 2012.
[26] Y. devi Ykuntam, K. Pavani, y K. Saladi, «Design and analysis of High speed wallace tree multiplier using parallel prefix adders for VLSI circuit designs», en *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, Kharagpur, India, jul. 2020, pp. 1-6. doi: 10.1109/ICCCNT49239.2020.9225404.
[27] U. Kumar y A. Fam, «Enhanced Wallace Tree Multiplier via a Prefix Adder», en *2020 IEEE Student Conference on Research and Development (SCOReD)*, Batu Pahat, Malaysia, sep. 2020, pp. 211-216. doi: 10.1109/SCOReD50371.2020.9251025.