UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

CORE IP DE PROCESADOR RISC-V EN SYSTEM VERILOG

Izan Segarra Górriz

Tutor: Dr. Jorge Daniel Martínez Pérez
Cotutor: Dr. José María Monzó Ferrer

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2018-19

Valencia, XX de septiembre de 2019

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Politécnica de València
Edificio 4D. Camino de Vera, S/N, 46022 Valencia
Tel. +34 963 87 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada

Resumen

El objetivo fundamental del presente trabajo es desarrollar un core IP de procesador RISC-V en SystemVerilog que soporte de manera completa el ISA (*Instruction Set Architecture*) RV32IM, tanto en su versión *Single Cycle,* como en su versión *Multi Cycle.*

La aplicación del core está orientada a la implementación de un *SoC* (*System-on-Chip*) en dispositivos programables de bajo coste y consumo de potencia, por lo que su concepción está orientada a la minimización de los recursos necesarios para su implementación.

Además, se detallan los pasos a seguir para instalar y configurar una *toolchain* para arquitecturas de 32 bits que permita la ejecución de *software* compilado en C sobre el procesador diseñado, así como el desarrollo de una aplicación con interfaz gráfica que permita realizar la compilación del *software* y cargar los archivos generados de forma sencilla en el procesador.

También se muestran los *testbenches* que se han ejecutado sobre el procesador para comprobar su correcto funcionamiento y comparar el rendimiento de las versiones *Single Cycle* y *Multi Cycle*.

Resum

L'objectiu fonamental d'aquest treball és desenvolupar un core IP de processador RISC-V en SytemVerilog que suporte de manera completa l'ISA (*Instruction Set Architecture*) RV32IM, tant en la seua versió *Single Cycle,* com en la seua versió *Multi Cycle*.

L'aplicació del core està orientada a la implementació d'un *SoC* (*System-on-Chip*) en dispositius programables de baix cost i consum de potència. D'aquesta manera, la seua concepció està orientada a la minimització dels recursos necesaris per a la seua implementació.

A més, es detallen els passos que s'han de seguir per instal·lar i configurar una *toolchain* per a arquitectures de 32 bits que permeta l'execució de *software* compilat en C sobre el processador disenyat, així com el desenvolupament d'una aplicació amb interfaç gràfica que permeta realitzar la compilació del *software* i carregar els arxius generats de forma senzilla en el processador.

També es mostren els *testbenches* que s'han executat sobre el processador per a comprovar el seu correcte funcionament i comparar el rendiment de les versions *Single Cycle* i *Multi Cycle*.

Abstract

The main objective of this project is to develop a RISC-V processor IP core in SystemVerilog that fully supports the RV32IM ISA (Instruction Set Architecture), both in its Single Cycle version and in the Multi Cycle version.

The application of the core is oriented to the implementation of a SoC (System-on-Chip) in programmable devices of low cost and power consumption. Thus, its conception is oriented to the minimization of the necessary resources for its implementation.

In addition, the steps to be followed to install and configure a toolchain for 32-bit architectures that allow the execution of software compiled in C on the designed processor are detailed, as well as the development of an application with a graphic interface that allows compiling the software and load the generated files in a simple way in the processor.

The testbenches that have been executed on the processor are also shown to check its correct operation and compare the performance of the Single Cycle and Multi Cycle versions.

Índice

Capítulo 1. Preámbulo

El punto de partida de este trabajo es el proyecto desarrollado por Pedro Antequera Cañadas, Jorge Rodríguez Ponce, Jesús Marín Rodríguez e Izan Segarra Górriz en la asignatura de Integración de Sistemas Digitales (ISDIGI) cursada durante el grado, que consiste en un core RISC V *Single Cycle* con soporte parcial para ISA RV32I.

El objetivo es ampliar este desarrollo para que el procesador soporte de forma completa el ISA RV32IM, tanto en la versión *Single Cycle* como en la versión *Multi Cycle,* y pueda realizarse una implementación de un *SoC* en dispositivos programables de bajo coste y consumo de potencia. Por ello, su diseño está orientado a la minimización de los recursos necesarios para su implementación.

Además, se realiza paso a paso la instalación y configuración de la *toolchain* RISC-V GNU Compiler disponible en GitHub para compilar y ejecutar código C en procesador y se muestra cómo diseñar una *app* que permite realizar la compilación de forma rápida y sencilla mediante una interfaz de usurario creada con MATLAB App Designer.

Para ejecutar las pruebas de funcionamiento, hay que lanzar los *testbenches* diseñados y visualizar los resultados en la web diseñada para ello.

Para la realización de este trabajo se ha utilizado un MacBook Pro con el sistema macOS Mojave y se ha hecho uso de VirtualBox para ejecutar una máquina virtual con Windows 10 Pro. El equipo ha sido conectado a dos monitores con la finalidad de trabajar en paralelo con ambos sistemas operativos y ser más productivo. Ambos sistemas están comunicados mediante una carpeta compartida en la que se encuentra el código RTL del diseño.

Figura 1. Entorno de trabajo

A continuación, se detalla qué aplicaciones han sido utilizadas en cada sistema operativo:

- **Entorno Windows:** Este entorno ha sido utilizado para el diseño de los diagramas del procesador con Visio, ejecutar los *testbenches* y comprobar el correcto funcionamiento del diseño mediante Questa Sim, el análisis temporal de las unidades funcionales con Quartus Prime y la obtención de la frecuencia máxima de operación mediante la herramienta TimeQuest Timing Analyzer integrada en Quartus.
- **Entorno macOS:** Este entorno ha sido utilizado para realizar la descripción *hardware* del procesador en SystemVerilog haciendo uso de Visual Studio Code, crear una aplicación que permita compilar el código C de forma sencilla mediante MATLAB y crear una página web que permita visualizar el resultado de las simulaciones mediante Sublime Text.

Cabe destacar que todas las aplicaciones utilizadas son compatibles con Windows, macOS y Linux, a excepción de Questa Sim Quartus Prime y Visio, que únicamente son compatibles con Windows y Linux.

Además, se han utilizado otras herramientas como Trello para la gestión del proyecto o como GitHub para alojar el trabajo utilizando el sistema de control de versiones Git.

Durante el desarrollo de la memoria, se establecen puntos para mejorar los módulos del procesador y obtener mejores frecuencias de operación.

1.1. Objetivos

Los objetivos de este trabajo son los siguientes:

- Búscar bibliografía y recopilar información relativa a implementaciones del ISA RISCV
- Establecer una implementación básica *Single Cycle*.
- Soportar cargas/almacenamientos tipo *byte* y *half word*.
- Soportar las instrucciones de la extensión M.
- Analizar en el dominio temporal las unidades funcionales de la implementación **Single Cycle**.
- Establecer una implementación básica *Multi Cycle*.
- Detectar y controlar los riesgos de segmentación.
- Configurar una *toolchain* en C para el core desarrollado.
- Validar de forma funcional el core desarrollado.

1.2. Plan de trabajo

La planificación temporal de las tareas para la realización del presente trabajo es el mostrado en el diagrama de *Gantt* de la *Figura 2.*

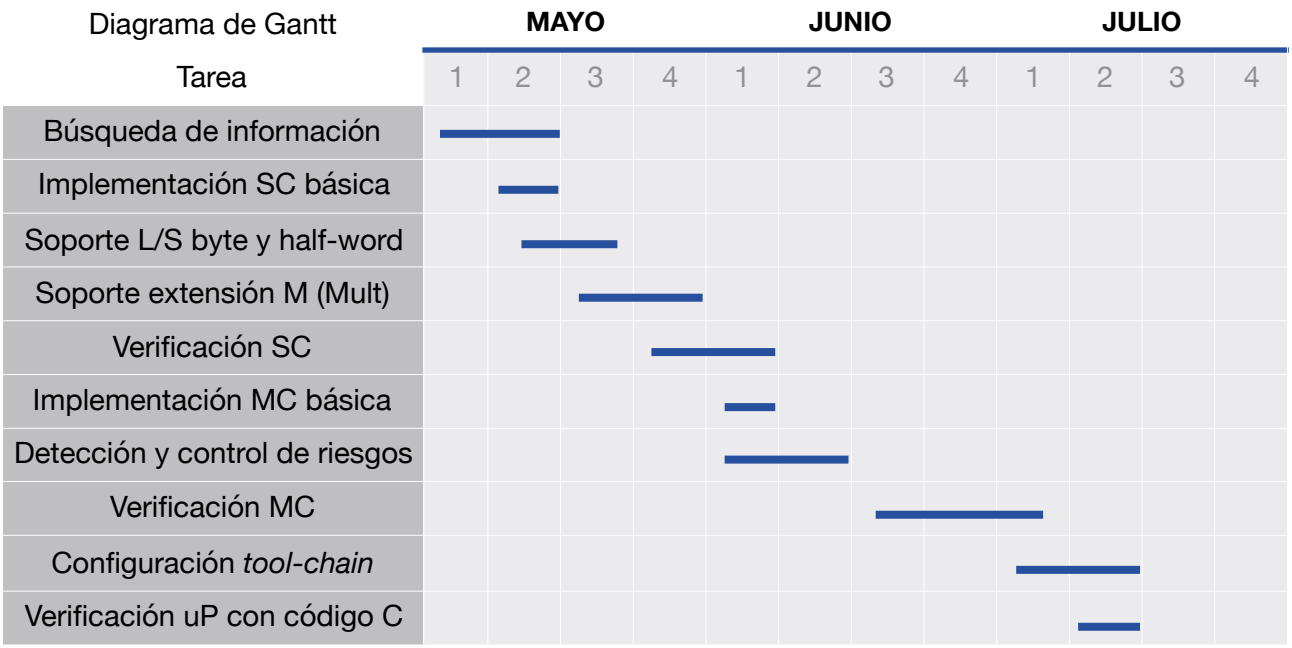

Figura 2. Diagrama de Gantt.

Capítulo 2. Introducción a RISC V

2.1. ¿Qué es RISC V?

RISC V es un conjunto de instrucciones, también conocido por su término en inglés ISA *(Instruction Set Architecture)*, que describe cómo deben funcionar los núcleos de los procesadores, es decir, constituye una interfaz entre el *hardware* y el *software* que fija la forma en la que las instrucciones deben ser codificadas, interpretadas y ejecutadas.

Este trabajo se centra en las arquitecturas RISC *(Reduced Instruction Set Computer)*, cuya finalidad es la utilización de ISAs pequeñas, simples y rápidas de ejecutar. Hay numerosas arquitecturas como x86 (Intel), ARM (ARM Holdings), SPARC (Sun Microsystems) o MIPS (MIPS Technologies) entre otras. Estas arquitecturas tienen ISAs propietarios que están asociadas al pago de *royalties* a sus propietarios, lo que dificulta su utilización en el ámbito académico y de investigación [1].

Es en 2010, cuando surge el proyecto RISC V en la Universidad de California, en Berkeley, con la meta de crear un ISA abierto y moderno que incluyera las mejores ideas en diseño de procesadores y fuese más rápido, eficiente, fácil de implementar y existiesen diversas soluciones en el mercado que compartan un mismo juego de instrucciones con la ventaja que ello conlleva. Todo el mundo puede diseñar su propia solución utilizando el ISA RISC V de forma gratuita, lo que permite adaptarlo a cualquier mercado, diferentes aplicaciones y con distintas especificaciones en el diseño [2].

A partir de 2017, se lanza la versión estable para el conjunto de instrucciones del espacio de usuario y admite anchos de palabra de 32, 64 y 128 bits [3].

2.2. Convención de nombres en RISC V

Tal y como se indica en el apartado anterior, existen diversos anchos de palabra compatibles con RISC V que configuran distintos ISAs asociados a un *hardware* concreto, además es posible añadir extensiones al ISA base.

Para comprender la nomenclatura analizaremos el siguiente ejemplo:

RV32IM

- **RV**: Las dos primeras letras son la abreviación de RISC V y todos los nombres comienzan de esta forma.
- **32**: Este número indica el número de bits que tiene de ancho el banco de registros, en este ejemplo es de 32 bits. Las opciones disponibles son 32, 64 y 128 bits.
- **I**: Indica que se soporta la aritmética de números enteros, forma parte del ISA base.
- **M**: Extensión que indica que el *hardware* soporta la multiplicación y la división.

Las extensiones estándar disponibles son: “M” para multiplicación y división, “A” para instrucciones atómicas, “F” para soporte de punto flotante de precisión simple (32 bits) y “D” para soporte de punto flotante de precisión doble (64 bits) [4].

Se puede configurar un ISA con todas las extensiones de forma que el nombre sería RV32IMAFD, existiendo la posibilidad de sustituir IMAFD por la letra “G” de forma que se sobreentiende que se soportan todas las extensiones estándar. Este documento se centra en la versión de 32 bits y no trata el resto de extensiones ni el conjunto de instrucciones privilegiado, para obtener más información acerca de ello véase *The RISC-V Instruction Set Manual: Unprivileged ISA* y *Privileged ISA* en https://riscv.org/specifications/

2.3. Conjunto de instrucciones base RV32I

RISC V es una arquitectura de carga/almacenamiento. Esto quiere decir que únicamente se transfieren los datos desde y hacia la memoria en las instrucciones de acceso a memoria *load* y *store.* El resto de instrucciones solo realiza operaciones en el banco de registros.

Hay 32 registros con un ancho de 32 bits (Véase *Tabla 1*) de los cuales en el ISA RV32I hay 31 (x1-x31) registros de propósito general que permiten trabajar con números enteros y un registro x0 cuyo valor es 0, es decir, está cableado directamente a GND. El registro x0 también puede ser utilizado para descartar el resultado de una operación. También existe un registro llamado contador de programa (PC) que es el que contiene la dirección de la próxima instrucción que se debe ejecutar.

Registro	Nombre	Uso
x0	zero	Valor constante 0
x1	ra	*Return Address*: Dirección de retorno de la función
x2	sp	*Stack Pointer*: Puntero de pila
x3	gp	*Global Pointer*: Puntero global a los datos
x4	tp	*Thread Pointer*: Puntero de hilo
x5 - x7	t0 - t2	*Temporaries*: Registros de propósito general que no conservan el valor entre funciones.
x8	s0/fp	*Saved Register/Frame Pointer*: Registro seguro que conserva el valor entre funciones. / Contiene la dirección anterior a la llamada de una función.
x9	s1	*Saved Register*: Registro seguro de propósito general que conserva el valor entre funciones.
x10 - x11	a0 - a1	*Function arguments/Return values*: Registros utilizados para pasar argumentos a las funciones o como valor de retorno de las funciones.
x12 - x17	a2 - a7	*Function arguments*: Registros utilizados para pasar argumentos a las funciones.
x18 - x27	s2 - s11	*Saved Registers*: Registros seguros de propósito general que conservan el valor entre funciones.
x28 - x31	t3 - t6	*Temporaries*: Registros de propósito general que no conservan el valor entre funciones.

Tabla 1. Estructura del banco de registros [4 - 5].

Las instrucciones están codificadas con 32 bits y se almacenan por defecto de forma alineada en la memoria (4 bytes) con el formato *little-endian,* existe la posibilidad de adaptar el hardware para trabajar en el modo *big-endian (*Véase la sección *Modelo de memoria)*. Existen 6 formatos de instrucción, de los cuales 4 son los principales (R, I, S y U) y 2 son variaciones (SB y UJ). La *Tabla 2* muestra como se codifican las instrucciones según el formato.

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	Opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	Opcode
U	imm[31:12]					rd	Opcode
UJ	imm[20|10:1|11|19:12]					rd	Opcode

Tabla 2. Formato de instrucciones [4].

El ISA RV32I contiene 37 instrucciones sin incluir las llamadas al sistema *ecall* y *break*, suficientes como para poder configurar una arquitectura a la hora de compilar el código C y cumplir con las necesidades de los dispositivos actuales. Se pueden clasificar en tres grupos: instrucciones aritméticas, instrucciones de acceso a memoria e instrucciones de control de flujo.

Instrucción	Formato	Clasificación	Operación	Descripción
add	R	Aritmética	+	R[rd] = R[rs1] + R[rs2]
sub	R	Aritmética	-	R[rd] = R[rs1] - R[rs2]
sll	R	Aritmética	<<	R[rd] = R[rs1] << R[rs2]
slt	R	Aritmética	(A<B)	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0
sltu	R	Aritmética	(A<B)	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0
xor	R	Aritmética	^	R[rd] = R[rs1] ^ R[rs2]
srl	R	Aritmética	>>	R[rd] = R[rs1] >> R[rs2]
sra	R	Aritmética	>>>	R[rd] = R[rs1] >>> R[rs2]
or	R	Aritmética	|	R[rd] = R[rs1] | R[rs2]
and	R	Aritmética	&	R[rd] = R[rs1] & R[rs2]
lb	I	Acceso a memoria	+	R[rd] = {24'bM[7],M[R[rs1]+imm](7:0)}
lh	I	Acceso a memoria	+	R[rd] = {16'bM[15],M[R[rs1]+imm](15:0)}
lw	I	Acceso a memoria	+	R[rd] = M[R[rs1]+imm](31:0)
lbu	I	Acceso a memoria	+	R[rd] = {24'b0,M[R[rs1]+imm](7:0)}
lhu	I	Acceso a memoria	+	R[rd] = {16'b0,M[R[rs1]+imm](15:0)}
addi	I	Aritmética	+	R[rd] = R[rs1] + imm
slli	I	Aritmética	<<	R[rd] = R[rs1] << imm
slti	I	Aritmética	(A<B)	R[rd] = (R[rs1] < imm) ? 1 : 0
sltiu	I	Aritmética	(A<B)	R[rd] = (R[rs1] < imm) ? 1 : 0
xori	I	Aritmética	^	R[rd] = R[rs1] ^ imm
srli	I	Aritmética	>>	R[rd] = R[rs1] >> imm
srai	I	Aritmética	>>>	R[rd] = R[rs1] >>> imm
ori	I	Aritmética	|	R[rd] = R[rs1] | imm
andi	I	Aritmética	&	R[rd] = R[rs1] & imm
jalr	I	Control de flujo	+	R[rd] = PC+4; PC = R[rs1] + imm
sb	S	Acceso a memoria	+	M[R[rs1] + imm](7:0) = R[rs2](7:0)
sh	S	Acceso a memoria	+	M[R[rs1] + imm](15:0) = R[rs2](15:0)
sw	S	Acceso a memoria	+	M[R[rs1] + imm](31:0) = R[rs2](31:0)
beq	SB	Control de flujo	-	if(R[rs1] == R[rs2]) then PC = PC +{imm,1'b0}
bne	SB	Control de flujo	^	if(R[rs1] != R[rs2]) then PC = PC +{imm,1'b0}
blt	SB	Control de flujo	(A<B)	if(R[rs1] < R[rs2]) then PC = PC +{imm,1'b0}
bge	SB	Control de flujo	(A>=B)	if(R[rs1] >= R[rs2]) then PC = PC +{imm,1'b0}
bltu	SB	Control de flujo	(A<B)	if(R[rs1] < R[rs2]) then PC = PC +{imm,1'b0}
bgeu	SB	Control de flujo	(A>=B)	if(R[rs1] >= R[rs2]) then PC = PC +{imm,1'b0}
auipc	U	Aritmética	+	R[rd] = R[rs1] + {imm, 12'b0}
lui	U	Aritmética	+	R[rd] = {32b'imm[31],imm,12'b0}
jal	UJ	Control de flujo	+	R[rd] = PC + 4; PC = PC + {imm,1'b0}

Tabla 3. Formato de instrucciones [4].

2.4. Modelo de memoria

RISC V es una arquitectura de 32 bits, por lo que las direcciones tienen 4 bytes y permite direccionar hasta 2^{32} = 4.294.967.296 (4 GiB) direcciones de memoria. El rango de valores disponible es de 0x00000000 - 0xFFFFFFFF.

La especificación contempla el acceso de datos no alineados, pero en este caso solo se permite el acceso alineado. Se ha tomado esta decisión porque que de no ser así se incrementaría la complejidad del *hardware* y los programas funcionarían con menor velocidad debido al mayor numero de ciclos de reloj necesarios para obtener el dato.

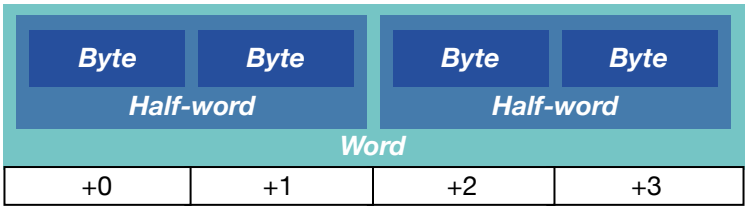

Figura 3. Alineamiento.

En la *Figura 3* se observa claramente las alineaciones permitidas en función del tamaño del dato. Para que el dato esté alineado correctamente:

- **Tipo *byte*:** La dirección tiene que ser múltiplo de 1, en este caso todas las alineaciones posibles (“0”, ”1”, ”2” y “3”) son correctas.
- **Tipo *half-word:*** La dirección tiene que ser múltiplo de 2, en este caso solo son correctas las alineaciones “0” y “2”.
- **Tipo *word:*** La dirección tiene que ser múltiplo de 4, por lo que la única alineación correcta es “0”.

En definitiva, los datos tienen que estar alineados en fronteras de 4 *bytes*, 2 *bytes* y 1 *byte*.

2.4.1. Formato de almacenamiento

El formato de almacenamiento, conocido por el término inglés *endianness,* denota la forma en la que deben almacenarse los datos que contengan más de un *byte* en memoria. Poniendo como ejemplo el valor 0x0A0B0C0D, en el formato *big-endian* el byte más significativo (0A) se almacena en el *byte* con la dirección más baja, mientras que en el formato *little-endian* el *byte* menos significativo (0D) es el que se almacena en el *byte* con la dirección más baja.

Dirección	+0	+1	+2	+3
0x10000008	00	00	00	00
0x1000004	00	00	00	00
0x10000000	0A	0B	0C	0D

Tabla 4. *Big-endian.*

Dirección	+0	+1	+2	+3
0x10000008	00	00	00	00
0x1000004	00	00	00	00
0x10000000	0D	0C	0B	0A

Tabla 5. Little-*endian.*

Si se produce una lectura del byte con la posición más baja obtendremos 0x0A en *big-endian* y 0x0D en *little-endian.* Por ello se debe tener en cuenta el formato de almacenamiento con el que se está trabajando.

El ISA del RISC V ha sido definida para trabajar en *little-endian* aunque deja abierta la posibilidad de trabajar en *big-endian* o *bi-endian* (capacidad para trabajar en ambos formatos) como variantes no estándar [4].

2.4.2. Disposición de memoria

La memoria del RISC V está dividida en varias partes denominadas segmentos, además como es una arquitectura de carga/almacenamiento cada segmento está asociado a la memoria de programa (ROM) o a la memoria de datos (RAM). La *Figura 4* muestra una posible forma de asignar los segmentos.

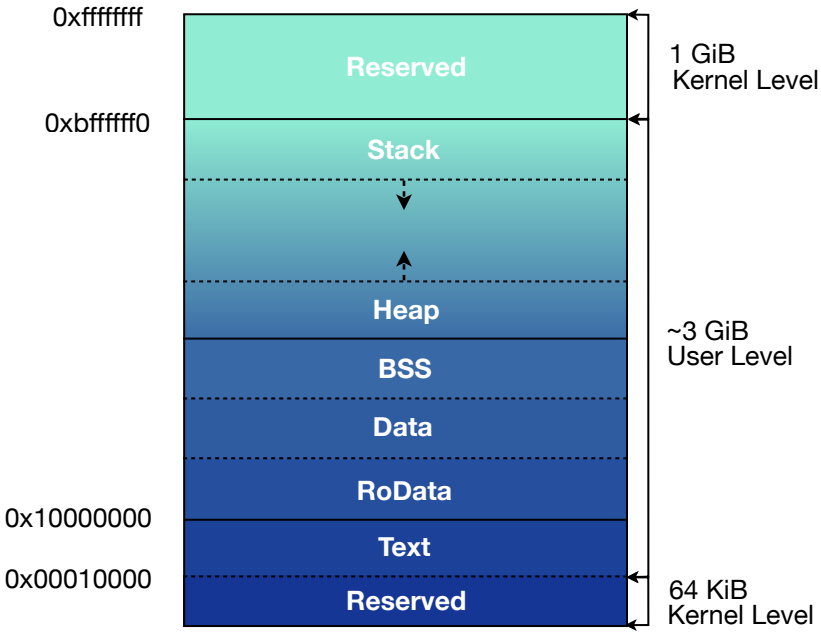

Figura 4. Formato de instrucciones [6 - 8].

A continuación se analiza cada sección de memoria por separado (sólo las secciones *user level*):

- **Text**: El segmento de texto o segmento de código, es la sección de memoria que contiene las instrucciones del programa qué se deben ejecutar. Este segmento es solo de lectura y no puede modificarse.
- **RoData**: El segmento rodata (*read only data*) contiene las variables constantes que no se pueden modificar durante el tiempo de ejecución del programa.
- **Data**: El segmento de datos contiene las variables globales y las variables estáticas cuyo valor de inicialización es distinto de 0. En este segmento puede leer y escribir datos.
- **BSS**: El segmento bss contiene las variables globales y las variables estáticas que no han sido inicializadas, es el sistema quien iniciativa las variables a 0 antes de que el programa se ejecute. En este segmento se puede leer y escribir datos.
- **Heap**: El segmento *heap* contiene las variables cuyo tamaño solo puede conocerse en tiempo de ejecución, es decir, se asigna la memoria de forma dinámica. Se realiza a través de las funciones de asignación de memoria dinámica *malloc, calloc, new, free* o *delete* entre otras y crece de direcciones bajas a direcciones altas.
- **Stack**: Es una estructura LIFO (*Last In First Out*) que se encuentra en las direcciones de memoria más altas y crece hacia las mas bajas. Se utiliza para almacenar de forma temporal los datos necesarios durante una llamada a una función y recuperarlos después. Por ejemplo antes de llamar a una función se almacena el valor de los registros en la pila (stack) y se recupera una vez se ha vuelto de ella.

La *toolchain* coloca las secciones .rodata, .data, .bss, .heap, y el área de stack en la RAM y la sección .text en la ROM *(Véase la sección Configuración de una toolchain)*.

Capítulo 3. Implementación single-cycle

3.1. Unidades funcionales básicas

En esta sección se sigue paso a paso el desarrollo de las unidades funcionales necesarias para la implementación *single-cycle* del RISC V. Una implementación *single-cycle* ejecuta una instrucción en cada ciclo de reloj, por lo que debe tenerse en cuenta a la hora de diseñar las unidades funcionales (memoria de instrucciones, memoria de datos y unidad aritmético-lógica).

3.1.1. Memoria de instrucciones y datos

Se va a realizar la descripción de una memoria ROM de 1024 posiciones con un ancho de palabra de 32 bits, es decir, una memoria de 1Kx32 (32 Kbits). A pesar de que el ISA RV32I soporta más memoria, para esta implementación es más que suficiente ya que los programas que se cargarán en la memoria de instrucciones no serán muy extensos.

El buses de direcciones de la ROM tiene un ancho de 10 bits ya que es el número de bits necesarios para poder direccionar las 1024 posiciones de memoria (2^{10} = 1024).

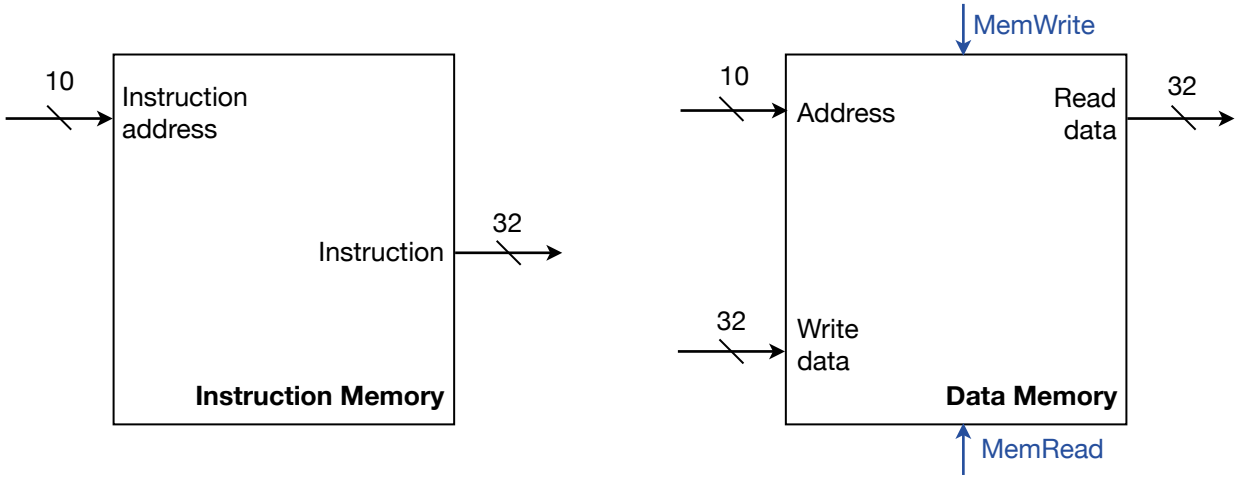

Figura 5. Memoria de instrucciones (ROM) y datos (RAM) [9].

Al igual que con la memoria de instrucciones, la memoria de datos (RAM) tendrá una capacidad de 32 Kbits, suficiente para almacenar los datos requeridos por los programas. Cabe destacar que esta primera versión sólo es compatible con las instrucciones sw y lw por lo que los datos deben estar alineados en una frontera de 4 *bytes* (32 bits).

Se puede definir la memoria con un acto de palabra y una profundidad parametrizables de una forma muy sencilla, la *Tabla 6* contiene un segmento de código SystemVerilog que muestra como realizar la descripción de la memoria ROM.

```
parameter MEM_DEPTH = 1024;             //1024 posiciones de memoria;
parameter SIZE = 32;                    //Tamanyo de palabra 32 bits;
logic [SIZE-1:0] rom [MEM_DEPTH-1:0];   //Definicion de la memoria ROM;
```

Tabla 6. Memoria de instrucciones (ROM) y datos (RAM) [10].

Las memorias se inicializan a partir de un fichero .txt mediante las funciones $readmemh si el contenido del fichero esta en hexadecimal o $readmemb si esta en binario. En esta primera implementación del procesador, ambas memorias implementan lecturas asíncronas y escrituras síncronas con la finalidad de llevar a cabo la ejecución de una instrucción en un ciclo de reloj.

3.1.2. Unidad aritmético-lógica (ALU)

La unidad aritmético-lógica o ALU (*Arithmetic Logic Unit*) es la encargada de realizar las operaciones aritméticas como sumar o restar y operaciones lógicas como evaluar una condición. La ALU tiene dos entradas de 32 bits que producen un resultado con el mismo número de bits y una salida de 1 bit que indica si el resultado de la operación ha sido 1 o 0.

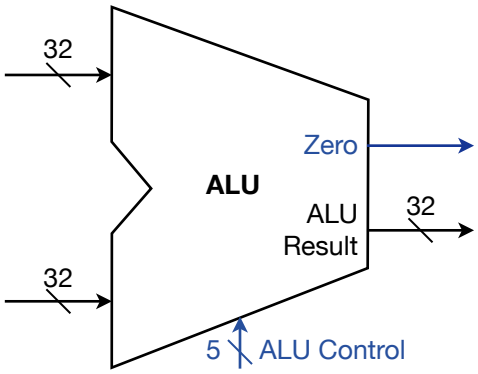

Figura 7. ALU. [9]

Para conseguir el valor de ALU Control se hace uso de los campos OPCODE, FUNCT7 y FUNCT3. El OPCODE sirve para obtener la señal ALU Operation que permite identificar el formato de instrucción y es la concatenación de los bits 6,5,4 y 2 del OPCODE. La *Figura 8* muestra el proceso.

	6	5	4	3	2	1	0
OPCODE	0	1	1	0	0	1	1
				3	2	1	0
ALUOP				0	1	1	0

Figura 8. Compresion OPCODE.

Tras esta compresión del OPCODE se puede diferenciar el tipo de instrucción de una forma mucho más compacta. En función del valor de ALU Operation, se pueden concatenar los campos de FUNCT7 y FUNCT3 de forma que se obtenga la codificación de ALU Control.

A continuación se indica la forma de obtener ALU Control en función del tipo de instrucción:

- **Formato R**: ALU Control = {FUNCT3, FUNCT7[5], FUNCT7[0]}
- **Formato I**: ALU Control = {FUNCT3, 2'b0}
- **Formato B**: ALU Control = {FUNCT3[0], FUNCT3[2], FUNCT3[1], !(FUNCT3[2]^FUNCT3[0]), 1'b0}
- **SRAI**: ALU Control = {FUNCT3, 1'b1, 1'b0}
- **Load**: ALU Control = 5'b00000
- **Store**: ALU Control = 5'b00000
- **LUI**: ALU Control = 5'b00000
- **AUIPC**: ALU Control = 5'b00000
- **JAL y JALR**: ALU Control = 5'b00000

La señal de control de la ALU de 5 bits es la que indica el tipo de operación que se debe realizar, la *Tabla 7* muestra la tabla de verdad con los valores de ALU Operation junto a la señal ALU Control y su operación asociada.

Instrucción	ALU Operation				FUNCT7[5]	FUNCT7[0]	FUNCT3			Operación	ALU Control				
LUI	0	1	1	1	X	X	X	X	X	+	1	1	1	1	1
AUIPC	0	0	1	1	X	X	X	X	X	+	0	0	0	0	0
JAL	1	1	0	1	X	X	X	X	X	+	0	0	0	0	0
JALR	1	1	0	1	X	X	0	0	0	+	0	0	0	0	0
BEQ	1	1	0	0	X	X	0	0	0	-	0	0	0	1	0
BNE	1	1	0	0	X	X	0	0	1	^	1	0	0	0	0
BLT	1	1	0	0	X	X	1	0	0	(A<B)	0	1	0	0	0
BGE	1	1	0	0	X	X	1	0	1	(A>=B)	1	1	0	1	0
BLTU	1	1	0	0	X	X	1	1	0	(A<B)	0	1	1	0	0
BGEU	1	1	0	0	X	X	1	1	1	(A>=B)	1	1	1	1	0
LB	0	0	0	0	X	X	0	0	0	+	0	0	0	0	0
LH	0	0	0	0	X	X	0	0	1	+	0	0	0	0	0
LW	0	0	0	0	X	X	0	1	0	+	0	0	0	0	0
LBU	0	0	0	0	X	X	1	0	0	+	0	0	0	0	0
LHU	0	0	0	0	X	X	1	0	1	+	0	0	0	0	0
SB	0	1	0	0	X	X	0	0	0	+	0	0	0	0	0
SH	0	1	0	0	X	X	0	0	1	+	0	0	0	0	0
SW	0	1	0	0	X	X	0	1	0	+	0	0	0	0	0
ADDI	0	0	1	0	X	X	0	0	0	+	0	0	0	0	0
SLTI	0	0	1	0	X	X	0	1	0	(A<B)	0	1	0	0	0
SLTIU	0	0	1	0	X	X	0	1	1	(A<B)	0	1	1	0	0
XORI	0	0	1	0	X	X	1	0	0	^	1	0	0	0	0
ORI	0	0	1	0	X	X	1	1	0	|	1	1	0	0	0
ANDI	0	0	1	0	X	X	1	1	1	&	1	1	1	0	0
SLLI	0	0	1	0	0	0	0	0	1	<<	0	0	1	0	0
SRLI	0	0	1	0	0	0	1	0	1	>>	1	0	1	0	0
SRAI	0	0	1	0	1	0	1	0	1	>>	1	0	1	1	0
ADD	0	1	1	0	0	0	0	0	0	+	0	0	0	0	0
SUB	0	1	1	0	1	0	0	0	0	-	0	0	0	1	0
SLL	0	1	1	0	0	0	0	0	1	<<	0	0	1	0	0
SLT	0	1	1	0	0	0	0	1	0	(A<B)	0	1	0	0	0
SLTU	0	1	1	0	0	0	0	1	1	(A<B)	0	1	1	0	0
XOR	0	1	1	0	0	0	1	0	0	^	1	0	0	0	0
SRL	0	1	1	0	0	0	1	0	1	>>	1	0	1	0	0
SRA	0	1	1	0	1	0	1	0	1	>>	1	0	1	1	0
OR	0	1	1	0	0	0	1	1	0	|	1	1	0	0	0
AND	0	1	1	0	0	0	1	1	1	&	1	1	1	0	0

Tabla 7. ALU Control.

3.1.3 Banco de registros

En este apartado se va a realizar la implementación del banco de registros del procesador de manera que sea posible realizar 2 lecturas y 1 escritura de forma simultánea, es decir, tendremos 2 puertos de lectura y 1 puerto de escritura.

Siguiendo las especificaciones de la arquitectura *single-cycle*, la lectura será asíncrona y la escritura síncrona, por lo que para realizar una escritura debe estar activada la señal de habilitación RegWrite. Este diseño permite leer y escribir en el mismo ciclo de reloj, obteniendo en la lectura el valor almacenado durante el ciclo anterior. Existe la posibilidad de hacer un *bypass* siempre y cuando la dirección de lectura y escritura coincidan, de este modo no sería necesario esperar un ciclo de reloj para obtener el dato almacenado.

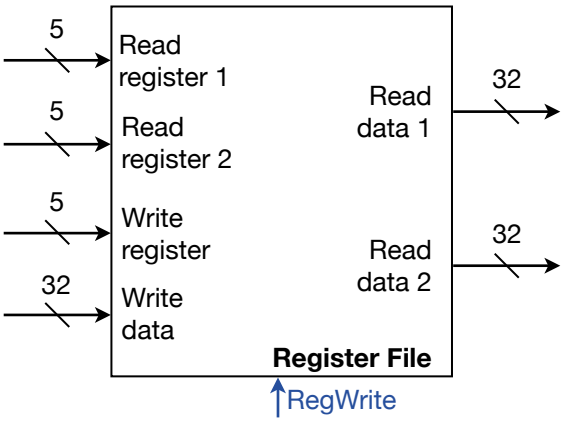

Figura 9. Banco de registros [9].

El banco está formado por 32 registros de 32 bits de propósito general. El registro x0 siempre almacena el valor 0, y cualquier escritura no tiene efecto sobre él. Cada vez que se inicie el sistema, el valor por defecto de los registros será 0 exceptuando los registros x2 y x3 cuyo valor será 0xbffffff0 (sp) y 0x10000000 (gp).

Los buses de direcciones del banco de registros tienen un ancho de 5 bits ya que es el número de bits necesarios para poder direccionar los 32 registros ($2^5 = 32$).

3.2. Establecimiento de una implementación básica

El core está compuesto por el *data path* y el *control path.* El *data path* es el conjunto de unidades funcionales (circuitos combinaciones y secuenciales) que procesa los datos, y el *control path* es la unidad que controla el flujo de datos del procesador y se encarga de descodificar las instrucciones a ejecutar.

Es necesario saber qué elementos son esenciales para ejecutar las instrucciones y combinarlos para componer el *data path*. Todas las unidades funcionales desarrolladas en la sección anterior forman parte del *data path,* pero hay que tener en cuenta que tanto la memoria de instrucciones como la memoria de datos son externas al core del procesador.

Los elementos adicionales que necesitamos son:

- **Contador de programa (PC)**: Registro de 32 bits que almacena la dirección de la instrucción actual.
- **Sumador**: Para esta implementación son necesarios dos sumadores de 32 bits con 2 entradas. El primero de ellos es utilizado para incrementar el PC y obtener la dirección de la próxima instrucción, mientras que el segundo es utilizado para sumar un offset al PC para obtener la dirección de salto en caso de que se produzca un salto condicional o incondicional.

- **Generador de inmediatos**: También conocido como extensor de signo, tiene como entrada los 32 bits de la instrucción y selecciona los bits necesarios (*Véase la Tabla 2*) según el formato de la instrucción. Extiende el signo hasta completar los 32 bits.
- **Multiplexor**: Un multiplexor es un circuito combinacional con varias entradas y una única salida, esto permite seleccionar una entrada mediante una señal de selección. En esta implementación se usan 3 multiplexores.
- **Conmutador**: Dispositivo que permite tener a la salida las entradas o su valor cruzado en función de la señal de control. Permite implementar AUIPC, LUI, JAL y JALR.

Con los componentes mencionados se puede realizar una primera implementación del *data path*. Cabe destacar que la implementación del banco de registros con 2 puertos de salida y 1 puerto de escritura simultáneos permite ahorrar hardware y duplicar recursos.

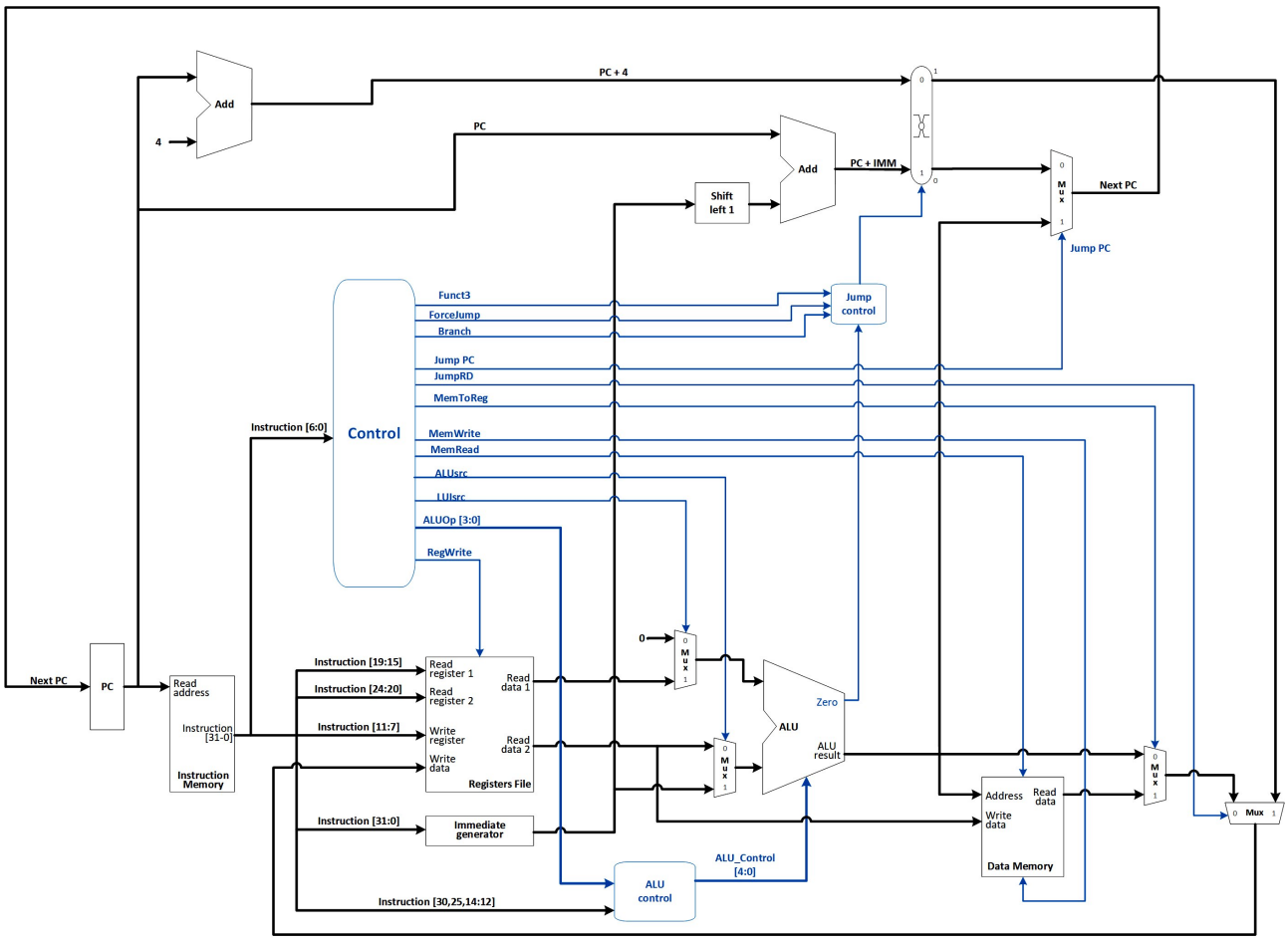

Figura 10. Implementación básica del RISC V [9].

Tal y como se ha avanzado, el *control path* es el encargado de gestionar el flujo de datos mediante las señales de control. La unidad de control actúa como un gran decodificador en el que a partir del OPCODE se obtiene el valor de cada señal. Además, consta de dos sub-unidades de control: ALU Control y Jump Control.

Son 8 las señales que controlan los elementos combinaciones y secuenciales como pueden ser los multiplexores o las memorias, son las siguientes:

- **JumpPC**: Controla el multiplexor que decide entre la salida del conmutador o la salida de la ALU y decide cuál es el próximo PC.

- **JumpRD**: Controla el multiplexor que decide entre el dato de la salida del conmutador y el dato proveniente del anterior multiplexor (resultado de la ALU o dato leído de la RAM) que será almacenado en el banco de registros.
- **MemToReg**: Controla el multiplexor que está en la entrada del operando B de la ALU y decide entre el resultado de la ALU o el dato leído de la RAM.
- **MemWrite**: Señal que habilita la escritura en la memoria de datos.
- **MemRead**: Señal que habilita la lectura en la memoria de datos.
- **ALUsrc**: Controla el multiplexor que decide entre el dato proveniente del banco de registros o el inmediato.
- **RegWrite**: Señal que habilita la escritura en el banco de registros.

La subunidad Jump Control está diseñada para decidir si se produce un salto condicional siendo sus señales:

- **Branch**: Señal de 1 bit que indica que la instrucción es un salto condicional.
- **ForceJump**: Señal de 1 bit que indica que la instrucción es un salto incondicional.
- **FUNCT3**: Campo FUNCT3 codificado en la instrucción.
- **ALU Result**: Bit menos significativo del resultado obtenido por la ALU.
- **Zero**: Señal de 1 bit que indica si el resultado obtenido por la ALU ha sido 0.
- **BranchMux**: Señal de salida de 1 bit que indica que se produce el salto.

Para entender correctamente su funcionamiento, la *Tabla 9* muestra bajo qué condiciones se activa la señal de salida BranchMux en función de sus entradas.

Branch	ForceJump	FUNCT3			ALU Result	Zero	BranchMux
1	1	X	X	X	X	X	1
1	0	0	0	0	X	1	1
1	0	0	0	1	X	0	1
1	0	0	1	0	X	X	0
1	0	0	1	1	X	X	0
1	0	1	0	0	1	X	1
1	0	1	0	1	0	X	1
1	0	1	1	0	1	X	1
1	0	1	1	1	0	X	1
0	1	X	X	X	X	X	0
0	0	X	X	X	X	X	0

Tabla 7. JUMP Control.

El funcionamiento de la subunidad ALU Control ha sido descrito en la sección *Unidades funcionales básicas* en el apartado de la unidad aritmético-lógica.

El resultado de la unión del data path y el control path deriva en la implementación básica del core del procesador y puede verse en la *Figura 10*.

3.3. Soporte de cargas/almacenamiento byte/half-word

La implementación básica sólo soporta las instrucciones *sw* y *lw* y en este apartado se realizan las modificaciones necesarias para soportar el resto de instrucciones de almacenamiento en memoria. Para ello es necesario realizar dos unidades, una que contenga la lógica de lectura y otra que contenga la lógica de escritura, de este modo es posible trabajar con datos de 4 *bytes*, 2 *bytes* y 1 *byte*.

La *Figura 11* describe el hardware adicional que se necesita para añadir la compatibilidad con el resto de instrucciones de carga/almacenamiento.

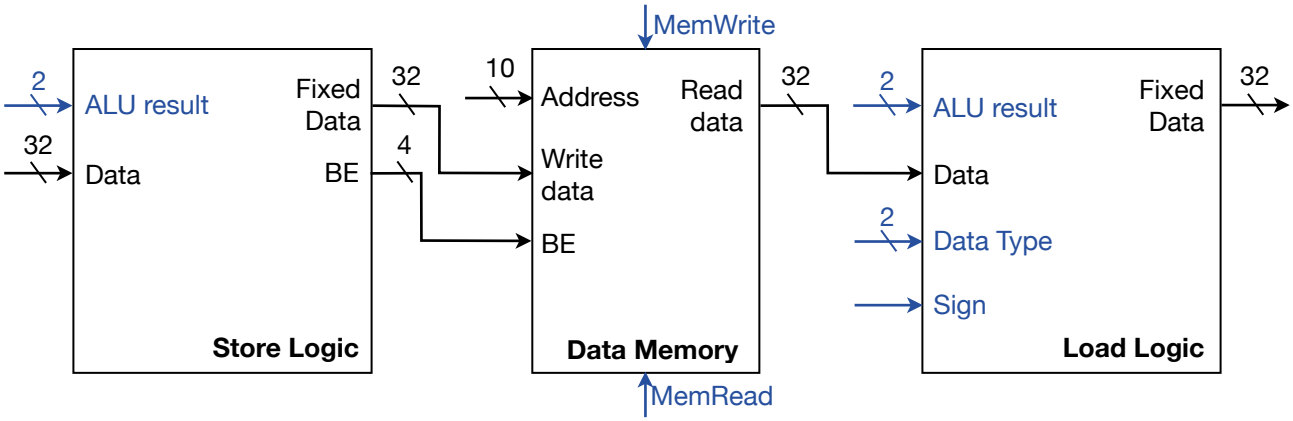

Figura 11. Lógica de carga/almacenamiento.

El primer cambio que se debe realizar es en la memoria de datos (RAM) para hacerla compatible con el direccionamiento tipo *byte*, sin el cual no sería posible realizar la implementación. Para ello la memoria RAM debe pasar de un bloque de 1Kx32 a 4 bloques de 1Kx8 que se disponen en paralelo para crear un bloque equivalente con un ancho de palabra de 32 bits.

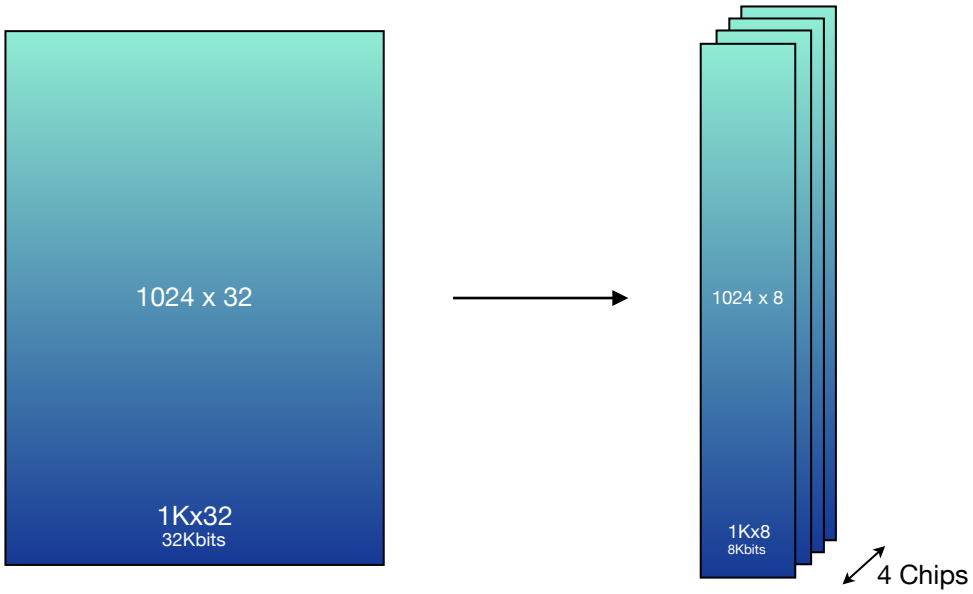

Figura 12. Bloques de memoria RAM.

También es necesaria una señal de habilitación para cada grupo de *bytes,* esta señal se llama byte *enable* y tiene 4 bits (uno para cada bloque de un *byte*). Así a la hora de realizar almacenar un dato, solo se sobreescriben los datos necesarios.

En la *Tabla 8* se indica la forma correcta de describir la memoria en SystemVerilog, solo debe indicarse el número de bloques (4) con su ancho de palabra (8) y la profundidad de la memoria, al igual que en la implementación anterior es de 1024 posiciones.

```
parameter MEM_DEPTH = 1024;              //1024 posiciones de memoria;
logic [3:0][7:0] ram [0:MEM_DEPTH-1];    //Definicion de la memoria RAM;
```

Tabla 8. Memoria RAM en bloques de 8 bits [10].

3.3.1. Lógica de lectura

Esta sub sección describe el modo de implementar la unidad que contiene la lógica de lectura necesaria para conseguir que el procesador sea compatible con las instrucciones *lb*, *lh*, *lbu* y *lhu.* Su funcionamiento consiste en cargar un valor de 32, 16 u 8 bits desde la memoria de datos y luego se extiende el signo a 32 bits antes de ser almacenado en el registro en el caso de *lb* y *lh*, en el caso de *lbu* y *lhu* simplemente se extiende a 32 bits antes de ser almacenado en el registro. [4]

La implementación es sencilla y para llevarla acabo se puede plantear la unidad en tres niveles lógicos: selección de *byte* y *half-word*, extensión de signo y selección del tipo de dato.

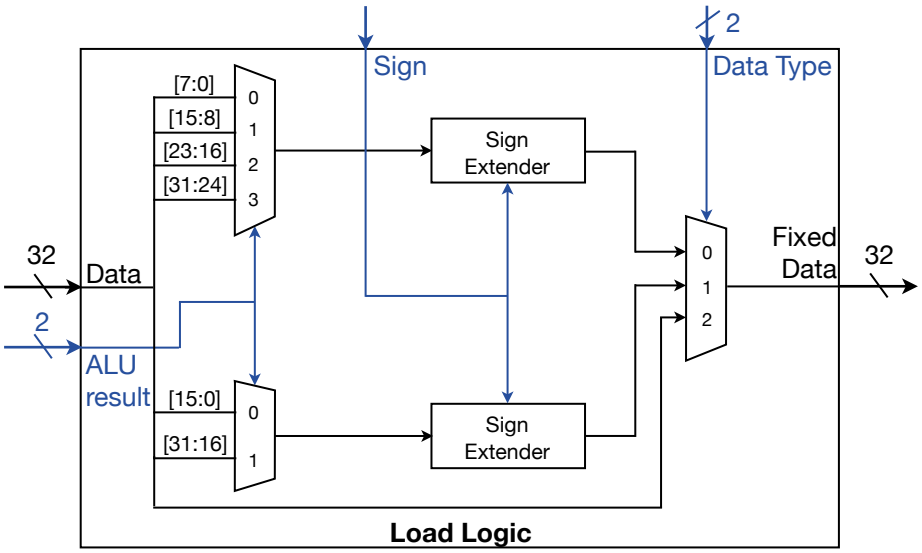

Figura 13. Lógica de lectura en detalle.

Para el correcto funcionamiento de la unidad son necesarias las señales de control que se describen a continuación:

- **ALU *result***: contiene los dos bits menos significativos del resultado obtenido por la ALU, permite identificar el *byte* o la *half-word.*
- ***Sign***: contiene el valor del bit más significativo de FUNCT3 que diferencia entre las instrucciones de almacenamiento con signo y sin signo. Indica si se debe extender el signo del dato o completar con 0 hasta rellenar los 32 bits.
- ***Data type***: identifica el tipo de dato que se debe seleccionar y viene determinado por los bits 1 y 0 de FUNCT3 que diferencia entre *byte, half-word* y *word*.

La Figura 13 muestra el esquema de la unidad y se puede ver la función que ejecuta cada una de las partes descritas.

Para entender mejor su funcionamiento, a continuación en la *Tabla 9* se muestra un ejemplo con cada una de las instrucciones de lectura del ISA base RV32I. En el ejemplo, en las instrucciones de lectura *lb y lbu* se escoge el *byte* menos significativo y en el caso de las instrucciones *lh y lhu* se escoge la media palabra menos significativa.

Instrucción	Dato de entrada	Dato intermedio	Extensor de signo	Tipo de dato	Dato de salida
lb	0x1000CDAB	0xAB	0xFFFFFFAB	*Byte*	0xFFFFFFAB
lh	0X1000CDAB	0xCDAB	0xFFFFCDAB	*Half-Word*	0xFFFFCDAB
lw	0X1000CDAB	-	-	*Word*	0X1000CDAB
lbu	0X1000CDAB	0xAB	0x000000AB	*Byte*	0x000000AB
lhu	0X1000CDAB	0xCDAB	0x0000CDAB	*Half-Word*	0x0000CDAB

Tabla 9. Ejemplo de funcionamiento de la lógica de lectura.

Viendo el ejemplo, se aprecia que mientras las instrucciones *lb, lh, lbu y lhu* utilizan todas la lógica combinacional que compone la unidad, la instrucción *lw* simplemente hace un *bypass* hasta el multiplexor de salida.

3.3.2. Lógica de escritura

Esta sub sección implementa la unidad que contiene la lógica de escritura necesaria para conseguir que el procesador sea compatible con las instrucciones *sb y sh.* Su funcionamiento consiste en almacenar en la memoria de instrucciones los 32, 16 u 8 bits más bajos del registro de origen (rs2). [4]

La implementación es sencilla y para llevarla acabo se puede plantear la unidad en dos niveles lógicos: selección de *byte* y *half-word*, y selección del tipo de dato. La *Figura 14* muestra el esquema de la unidad.

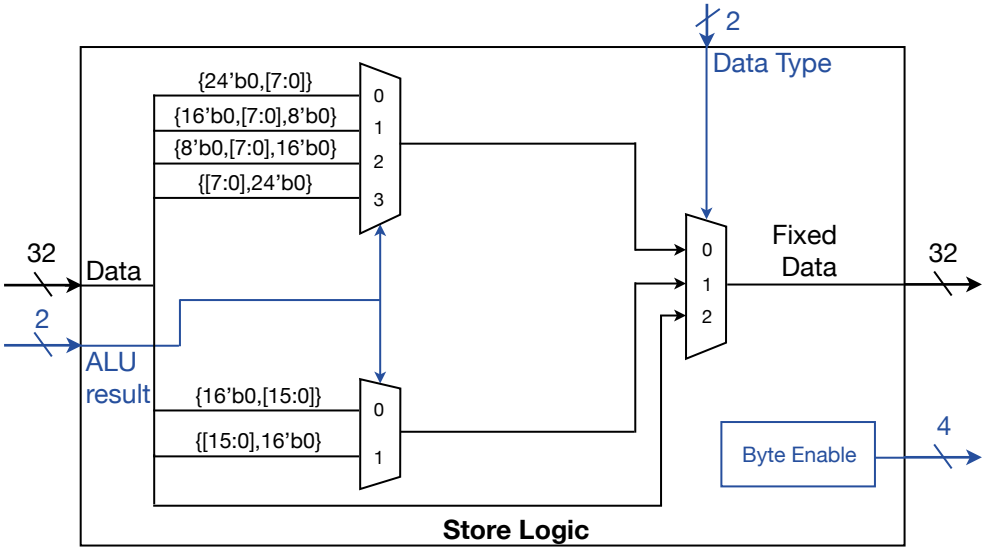

Figura 14. Lógica de lectura en detalle.

Para el correcto funcionamiento de la unidad son necesarias las señales de control que se describen a continuación:

- **ALU *result***: contiene los dos bits menos significativos del resultado obtenido por la ALU, permite identificar el *byte* o la *half-word.*
- ***Byte Enable***: señal que permite almacenar en memoria los bytes especificados. Se genera a partir de ALU *result* y *Data Type*.

- ***Data type***: identifica el tipo de dato que se debe seleccionar y viene determinado por los bits 1 y 0 de FUNCT3 que diferencia entre *byte, half-word* y *word*.

Para entender mejor su funcionamiento, a continuación en la *Tabla 10 s*e muestra un ejemplo con cada una de las instrucciones de escritura del ISA base RV32I. En el ejemplo, en la instrucción de escritura s*b* se escoge el *byte* más significativo y en el caso de la instrucción *sh* se escoge la media palabra más significativa.

Instrucción	Dato de entrada	Dato intermedio	Tipo de dato	Byte Enable	Dato de salida
sb	0xABCDEF11	0xAB000000	*Byte*	1000	0xAB000000
sh	0xABCDEF11	0xABCD0000	*Half-Word*	1100	0xABCD0000
sw	0xABCDEF11	-	*Word*	1111	0xABCDEF11

Tabla 10. Ejemplo de funcionamiento de la lógica de escritura.

Viendo el ejemplo, se aprecia que mientras las instrucciones *sb y sh* utilizan toda la lógica combinacional que compone la unidad, mientras que la instrucción s*w* simplemente hace un *bypass* hasta el multiplexor de salida.

3.4. Implementación de las instrucciones la extensión M

La extensión “M” permite añadir al procesador las operaciones de multiplicación y división entre dos registros. La extensión añade 8 instrucciones adicionales para la versión de 32 bits del ISA, puede verse con más detalle en la *Tabla 11*.

Instrucción	Formato	Clasificación	Operación	Descripción
MUL	R	Aritmética	*	R[rd] = R[rs1] * R[rs2](31:0)
MULH	R	Aritmética	*	R[rd] = R[rs1] * R[rs2](63:32)
MULHSU	R	Aritmética	*	R[rd] = R[rs1] * R[rs2](63:32)
MULHU	R	Aritmética	*	R[rd] = R[rs1] * R[rs2](63:32)
DIV	R	Aritmética	/	R[rd] = R[rs1] / R[rs2]
DIVU	R	Aritmética	/	R[rd] = R[rs1] / R[rs2]
REM	R	Aritmética	%	R[rd] = R[rs1] % R[rs2]
REMU	R	Aritmética	%	R[rd] = R[rs1] % R[rs2]

Tabla 11. Instrucciones de la extensión “M”.

Lo primero que se debe hacer es añadir la codificación para las instrucciones a implementar y ser capaces de codificar la señal ALU Control tal y como se muestra en la *Tabla 12*.

Instrucción	ALU Operation				FUNCT7[5]	FUNCT7[0]	FUNCT3			Operación	ALU Control				
MUL	0	1	1	0	0	1	0	0	0	*	0	0	0	0	1
MULH	0	1	1	0	0	1	0	0	1	*	0	0	1	0	1
MULHSU	0	1	1	0	0	1	0	1	0	*	0	1	0	0	1
MULHU	0	1	1	0	0	1	0	1	1	*	0	1	1	0	1
DIV	0	1	1	0	0	1	1	0	0	/	1	0	0	0	1
DIVU	0	1	1	0	0	1	1	0	1	/	1	0	1	0	1
REM	0	1	1	0	0	1	1	1	0	%	1	1	0	0	1
REMU	0	1	1	0	0	1	1	1	1	%	1	1	1	0	1

Tabla 12. ALU Control para instrucciones de la extensión “M”.

De esta forma se pueden incorporar a la unidad aritmético-lógica unidades de multiplicación y división, aunque en este caso se implementará en paralelo a esta. Para incorporar dichas unidades, utilizaremos IPs (*Intellectual Properties*) desarrollados por Intel siguiendo el procedimiento descrito a continuación:

Primero se debe acceder al *IP Catalog* desde el menú Tools y posteriormente seleccionar ALTMULT_ADD de forma que aparezca el asistente que muestra en la *Figura 15.*

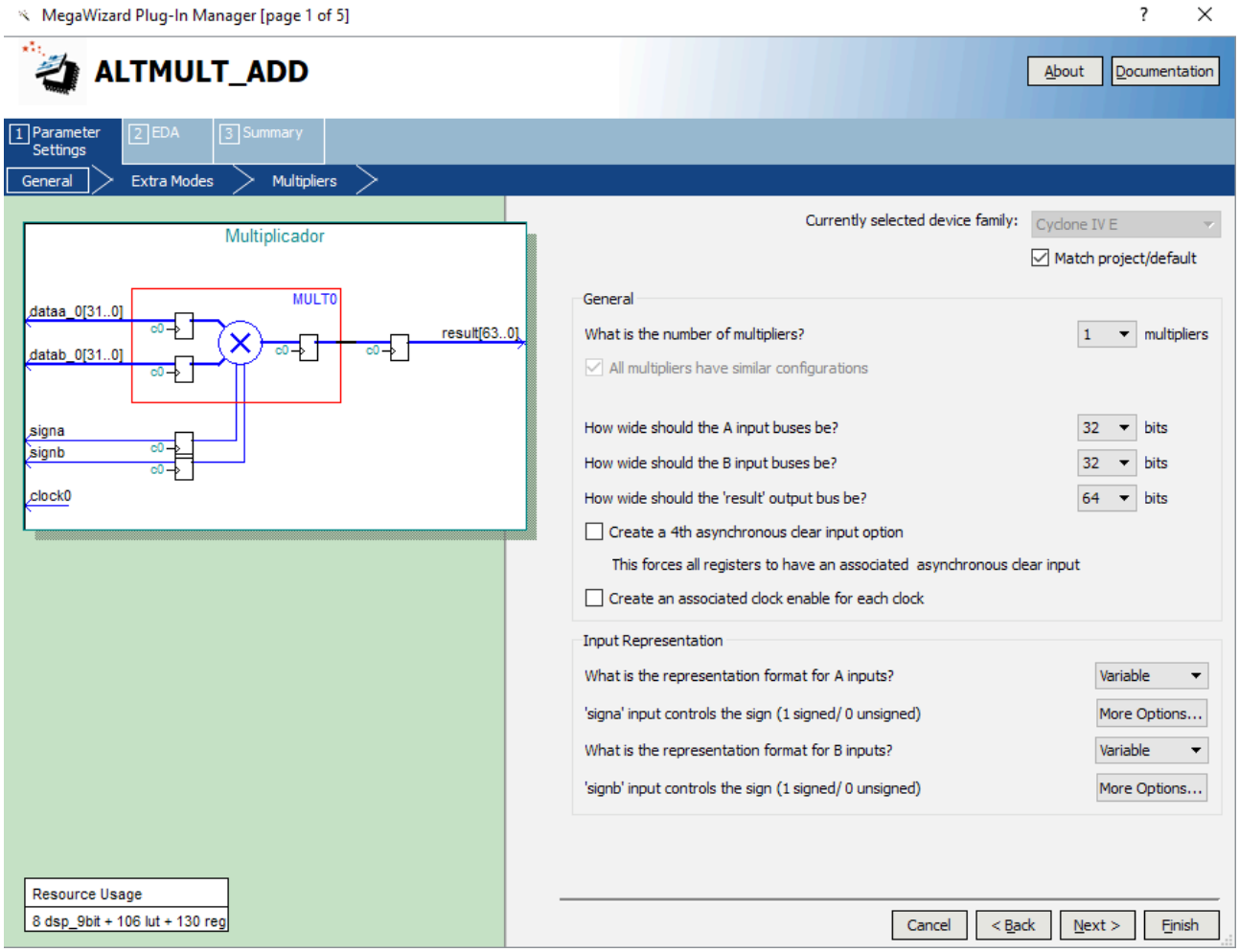

Figura 15. Asistente de configuración de la unidad de multiplicación.

Se debe especificar el ancho de los datos de entrada (32 bits) y el ancho del bus de salida, que en este caso es de 64 bits. También se debe seleccionar que los datos de entrada tienen representación variable para que el multiplicador sea compatible con valores con signo y sin signo.

En el apartado *Extra Modes*, se debe desmarcar las opciones de *Outputs Configurations* y seleccionar *Use dedicated multiplicar circuitry* en el apartado *Implementation* con el fin de utilizar los recursos dedicados del dispositivo programable.

Posteriormente en el apartado *Multipliers* se deben desmarcar todas las casillas en *Input Configuration* y *Output Configuration*. Una vez realizados estos pasos, simplemente hay que clicar *Next* hasta que aparezca *Finish* para cerrar el asistente. Como resultado se obtiene un fichero verilog que puede ser instancia para ser usado en el core desarrollado.

De forma análoga se puede obtener un fichero verilog con el divisor pulsando en la opción LPM_DIVIDE y siguiendo el asistente correspondiente.

En este caso, el divisor obtenido por el asistente tiene una frecuencia de operación de 8 MHz aproximadamente y no ha sido implementado en el procesador debido a que reduce drásticamente la frecuencia del diseño. Esto implica que no se puede hacer uso de todas las instrucciones que incluye la extensión “M”, solo son compatibles las instrucciones de multiplicación mul, mulh, mulhsu y mulhu. Esta situación deja abierta la posibilidad de, en un futuro, desarrollar un divisor con una frecuencia de operación mucho más elevada que la obtenida con la IP de Intel.

3.5. Análisis temporal de las unidades funcionales

Antes de analizar el core del procesador al completo, es importante conocer que unidad de las desarrolladas tendrá menor frecuencia de operación y limita la máxima frecuencia de trabajo del procesador. Las unidades funcionales a analizar son el banco de registros, la unidad aritmético-lógica, la lógica de lectura y la lógica de escritura.

Para obtener la frecuencia de trabajo de cada unidad, se debe poner un registro a la entrada y a la salida del circuito combinacional del cual se quiere obtener la frecuencia. La *Figura 16* muestra este proceso.

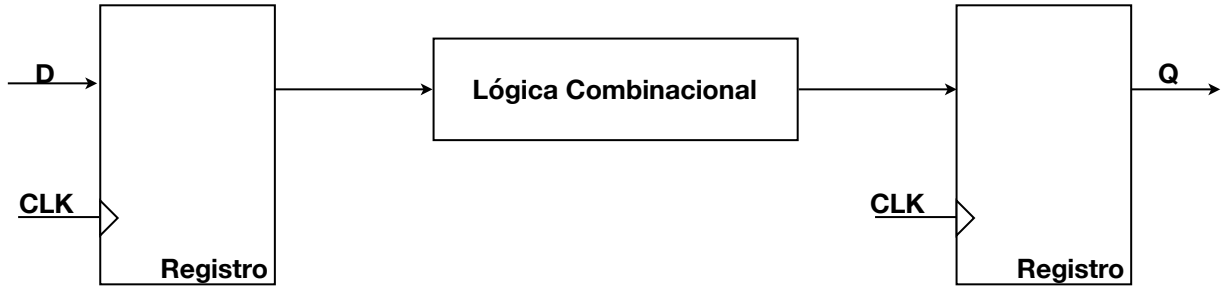

Figura 16. Obtención de la frecuencia de operación.

Mediante este procedimiento y el uso de Quartus Prime, se obtienen las siguientes frecuencias:

- **Banco de registros**: La máxima frecuencia de operación de la unidad es 191,64 MHz.
- **Unidad aritmético-lógica**: La máxima frecuencia de operación de la unidad es 108,33 MHz.
- **Lógica de lectura**: La máxima frecuencia de operación de la unidad es 383 MHz.
- **Lógica de escritura**: La máxima frecuencia de operación de la unidad es 431,41 MHz.
- **Multiplicador**: La máxima frecuencia de operación de la unidad es 98,43 MHz.

En este caso la unidad funcional que va a limitar la máxima frecuencia del procesador es el multiplicador, nótese que ni la memoria ROM ni la memoria RAM han sido analizadas ya que son externas y por lo tanto no forman parte del core.

3.6. Implementación RV32IM

Una vez desarrolladas las unidades necesarias para realizar la implementación del ISA RV32IM, se debe realizar la integración de las mismas en el *data path* e implementar las señales de control necesarias para su correcto funcionamiento.

Dado que el multiplicador se ha dispuesto en paralelo a la unidad aritmético-lógica, es necesario añadir un multiplexor que elija entre el resultado del multiplicador o el de la ALU. La señal de control necesaria viene dada por la sub unidad de control ALU Control y se genera con los bits 4, 1 y 0 de la señal ALU_Control.

En el caso de las unidades de lectura y escritura, sus señales de control ya han sido explicadas en las sub secciones 3.3.1. (Lógica de lectura) y 3.3.2. (Lógica de escritura).

El resultado de la integración de todas las unidades resulta en el esquema de la *Figura 17* y muestra el core completo con la compatibilidad del ISA RV32IM de forma parcial ya que no es compatible con la operación de división.

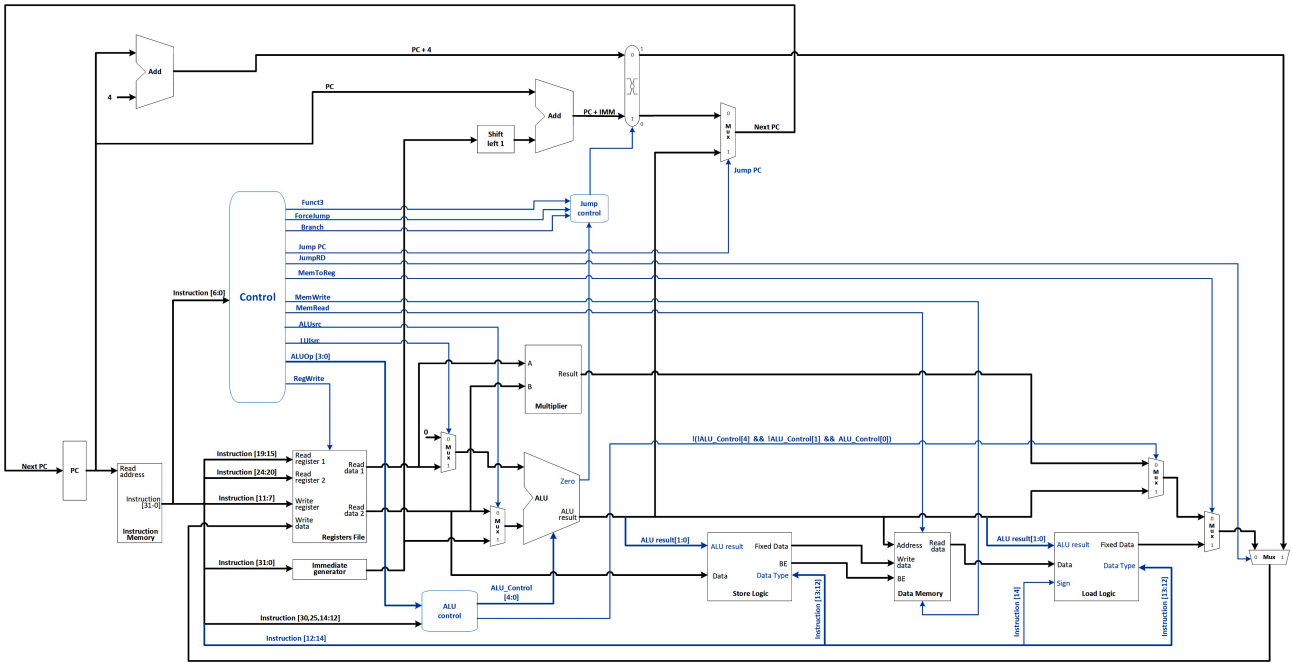

Figura 17. Implementación RV32IM.

Al realizar la compilación del proyecto en Quartus Prime, se obtiene un resumen con los recursos utilizados por la implementación *Single Cycle.* En la *Figura 18* se ve que tan solo se utiliza un 3% de los elementos lógicos, un 32% del total de los pines y un 2 % de los multiplicados embebidos. Por tanto se cumple el objetivo de minimizar el uso de los recursos disponibles en el dispositivo programable .

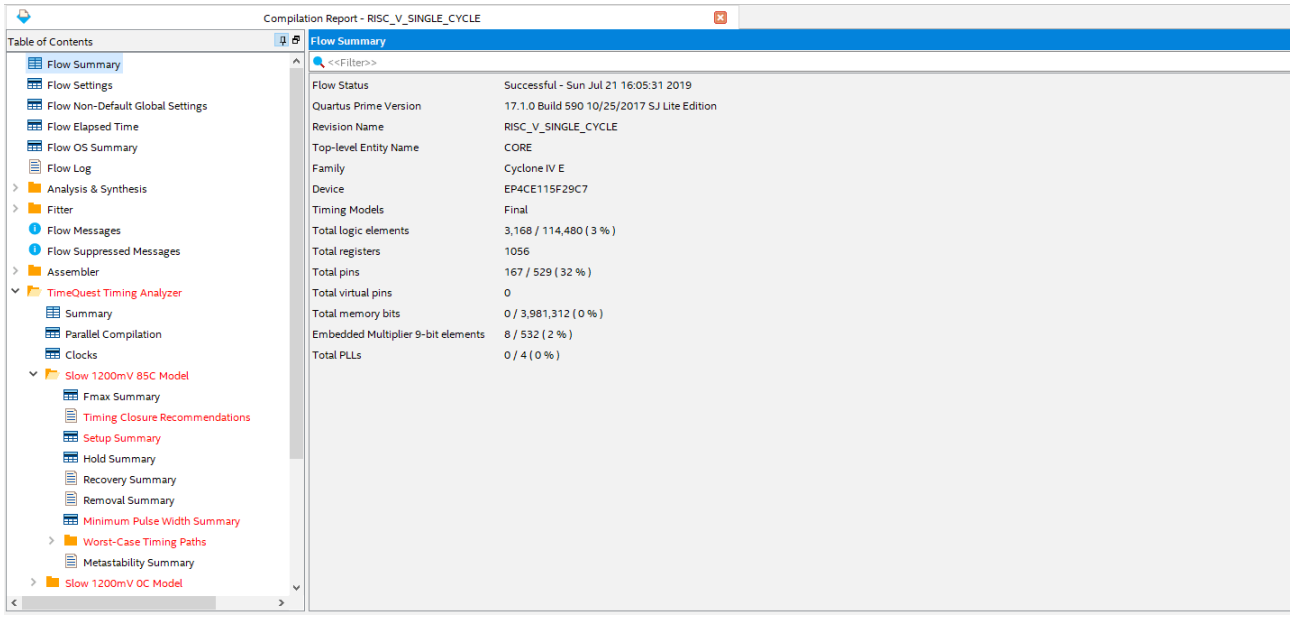

Figura 18. Recursos utilizados por la implementación.

Además, la compilación del proyecto también arroja como resultado el análisis frecuencial realizado por *TimeQuest Time Analyzer*, siendo la frecuencia máxima de operación del procesador 44,27 MHz.

Aunque la implementación Single Cycle funcione correctamente, hoy en día no se utiliza debido a su baja eficiencia. Este rendimiento tan bajo es debido a que el ciclo de reloj debe tener el mismo periodo para cada una de las instrucciones implementadas, dicho periodo viene determinado por la ruta crítica del procesador (la más larga), que en este caso corresponde a la ruta generada por los saltos condicionales. Por este motivo la frecuencia obtenida en este tipo de implementaciones no es demasiado buena.

La solución a este problema se desarrolla en el Capitulo 4, donde se explica el proceso por el cual se puede incrementar la frecuencia de operación del procesador. Este proceso es conocido como segmentación.

3.7. Verificación

Para verificar la implementación Single Cycle se ha realizado un testbench en SystemVerilog y se han desarrollado una serie de programas en ensamblador para comprobar el correcto funcionamiento del procesador. La estructura del directorio de trabajo utilizada es la que se puede ver en la *Figura 19*.

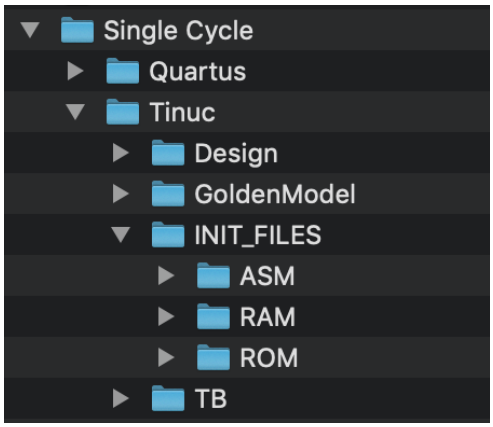

Figura 19. Estructura del directorio de trabajo.

Se ha desarrollado un *golden model* del procesador en alto nivel con el fin detectar fallos en la implementación desarrollada y utilizarlo a modo de unidad de depuración en caso de que falle la ejecución del programa cargado, así como modelo de referencia. También se ha desarrollado un script capaz de ejecutar la verificación sin necesidad de abrir Quartus Prime.

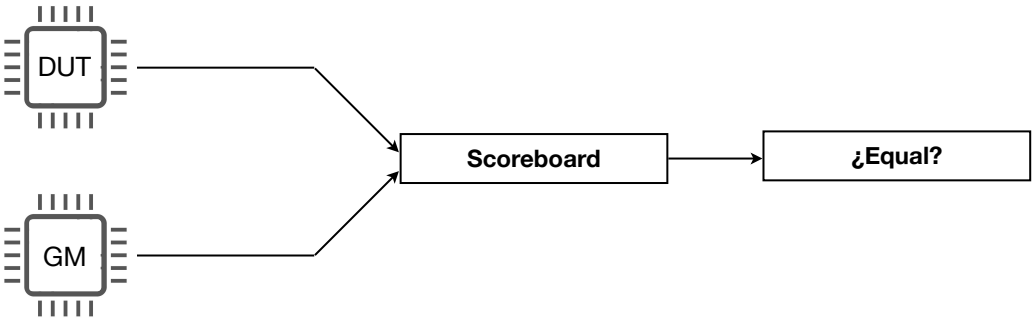

Figura 20. Diagrama simplificado del *testbench*.

Durante la ejecución de los programas de test, se compara ciclo a ciclo el resultado obtenido por el dispositivo bajo test (DUT) y el modelo de referencia en el *scoreboard*. El *scoreboard* se encarga de notificar si ambos modelos tienen la misma respuesta o no.

Se han ejecutado varios programas con la finalidad de probar todas las instrucciones soportadas por el procesador de forma exitosa. Además, se han desarrollado dos programas más complejos para asegurar el

correcto funcionamiento de la implementación. Estos programas se describen en las dos sub-secciones que hay a continuación y se comentan los resultados obtenidos.

3.7.1. *Bubble Sort*

Se va a realizar un programa en ensamblador del RSIC-V que ordene una serie de números almacenados en la memoria de datos utilizando el algoritmo Bubble Sort para comprobar el funcionamiento del procesador. El algoritmo implementado sigue la siguiente estructura:

- Empieza por el primer elemento (índice = 0), comparamos el elemento actual con el siguiente elemento del vector.
- Si el elemento actual es más grande que el siguiente, entonces los intercambiamos.
- Si el elemento actual es más pequeño que el siguiente, entonces nos movemos al siguiente elemento y repetimos el paso inicial.

Vemos un ejemplo gráfico de la implementación del algoritmo en la siguiente tabla:

5 > 1 (Intercambio)	5	1	6	2	4	3	1º Iteración
5 < 6 (No Intercambio)	5	1	6	2	4	3	
6 > 2 (Intercambio)	1	5	6	2	4	3	
6 > 4 (Intercambio)	1	5	2	6	4	3	
6 > 3 (Intercambio)	1	5	2	4	6	3	
-	1	5	2	4	3	6	

Tabla 13. Iteración del algoritmo.

En este caso se ha realizado la explicación para una ordenación de menor a mayor, aunque también se puede aplicar el algoritmo para ordenar de mayor a menor efectuando una pequeña modificación.

El programa de test que se ha ejecutado en el procesador contiene 9 números enteros desordenados desde la posición 0 hasta la posición 8 de la memoria de datos y en la posición 9 se almacena el número de dígitos a ordenar, en este caso el número de dígitos es 9.

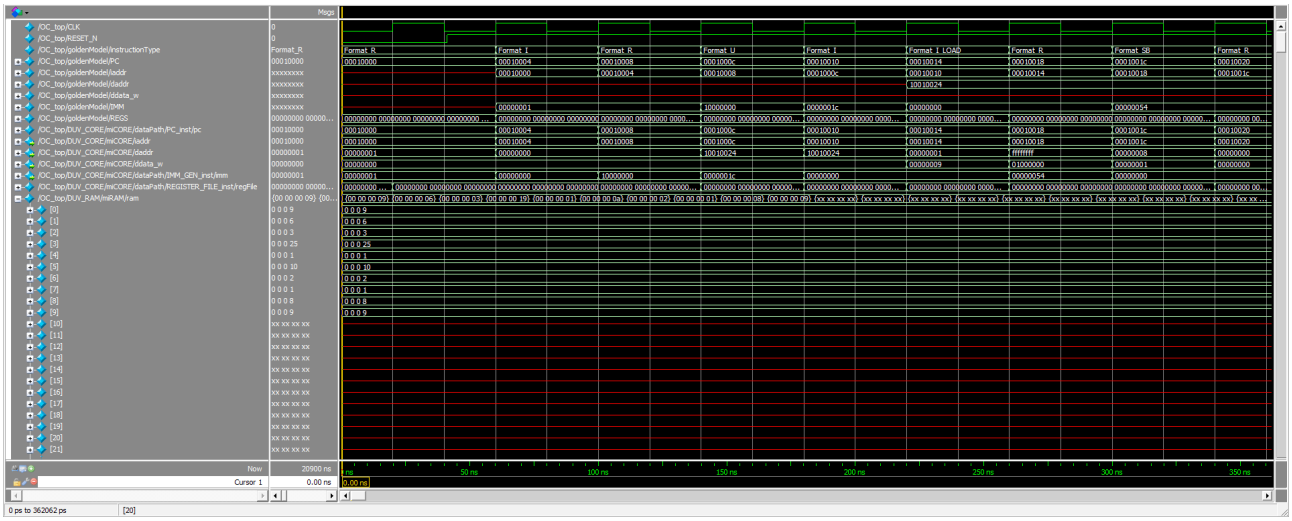

Figura 21. Estado inicial del programa *Bubble Sort.*

La *Figura 22* muestra el estado de la memoria de datos tras la ejecución del *Bubble Sort* y se comprueba que ha sido ejecutado con éxito.

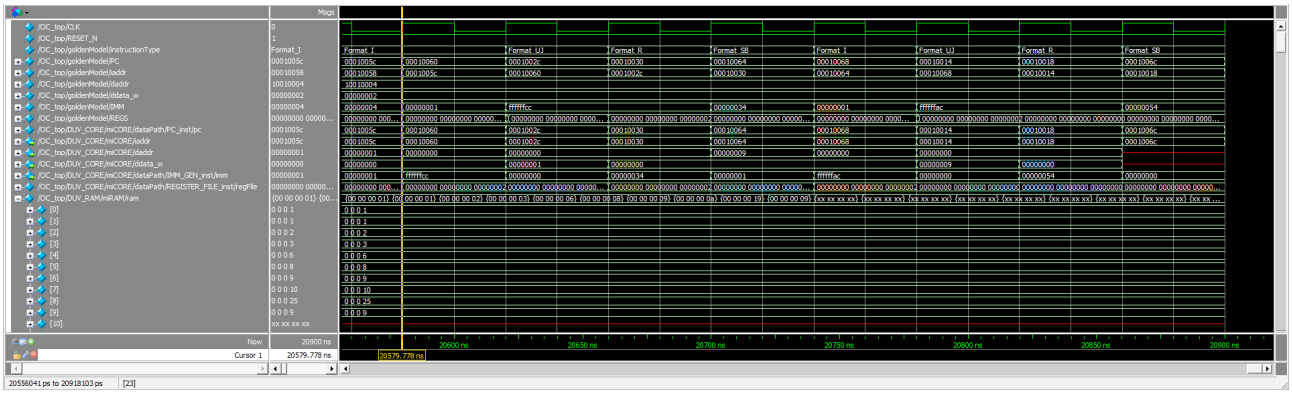

Figura 22. Estado final del programa *Bubble Sort*.

3.7.2. Serie *de Fibonacci*

Se va a realizar un programa en ensamblador del RISC-V que genere los N primeros términos de la sucesión de *Fibonacci* para comprobar el funcionamiento del procesador. Cada termino de la sucesión es la suma de los dos termino anteriores. Los dos primeros términos son 0 y 1.

En este ejemplo se van a generar los 20 primeros números de la sucesión y son los siguientes: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765.

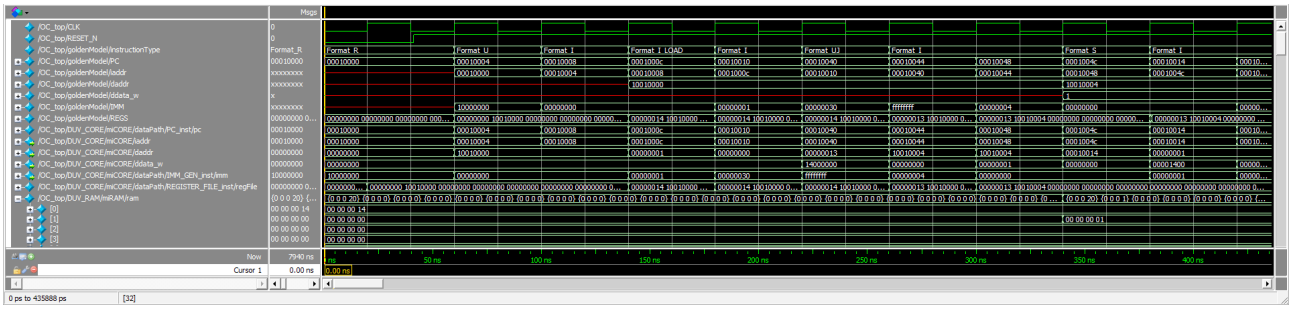

Figura 23. Estado inicial del programa que genera la sucesión de *Fibonacci*.

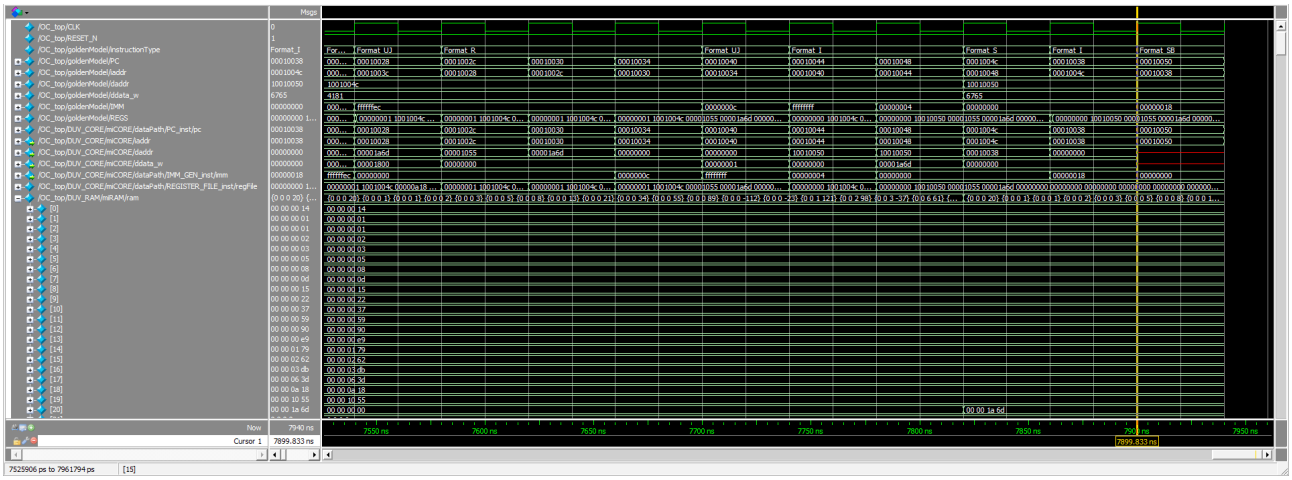

Figura 23. Estado final del programa que genera la sucesión de *Fibonacci*.

En la *Figura 23* se comprueba que desde la posición 1 de la memoria de datos hasta la posición 20, se ha generado de forma exitosa la sucesión de los primeros 20 términos (en hexadecimal).

Capítulo 4. Implementación pipelined

4.1. Introducción a la segmentación

La segmentación es una técnica de implementación por la que se ejecutan varias instrucciones de forma simultánea, eso posibilita el desarrollo de CPUs más rápidas. El objetivo fundamental de esta técnica es paralelizar la ejecución de instrucciones y explotar todos los recursos hardware disponibles, así se consigue aumentar el número de instrucciones por unidad de tiempo, con la mejora de rendimiento que conlleva [11 - 14].

En una segmentación ideal, la velocidad de ejecución se multiplica por el numero de etapas, en cambio, una segmentación real no tiene todas sus etapas balanceadas (mismo retardo) y existen dependencias entre ellas por lo que el incremento de velocidad es menor. Además la velocidad también está limitada por la etapa más lenta [11 - 14].

Para entender este concepto, se muestra la analogía de la lavandería en la *Figura 24* en la que las tareas a realizar son: lavar, tender, planchar y doblar.

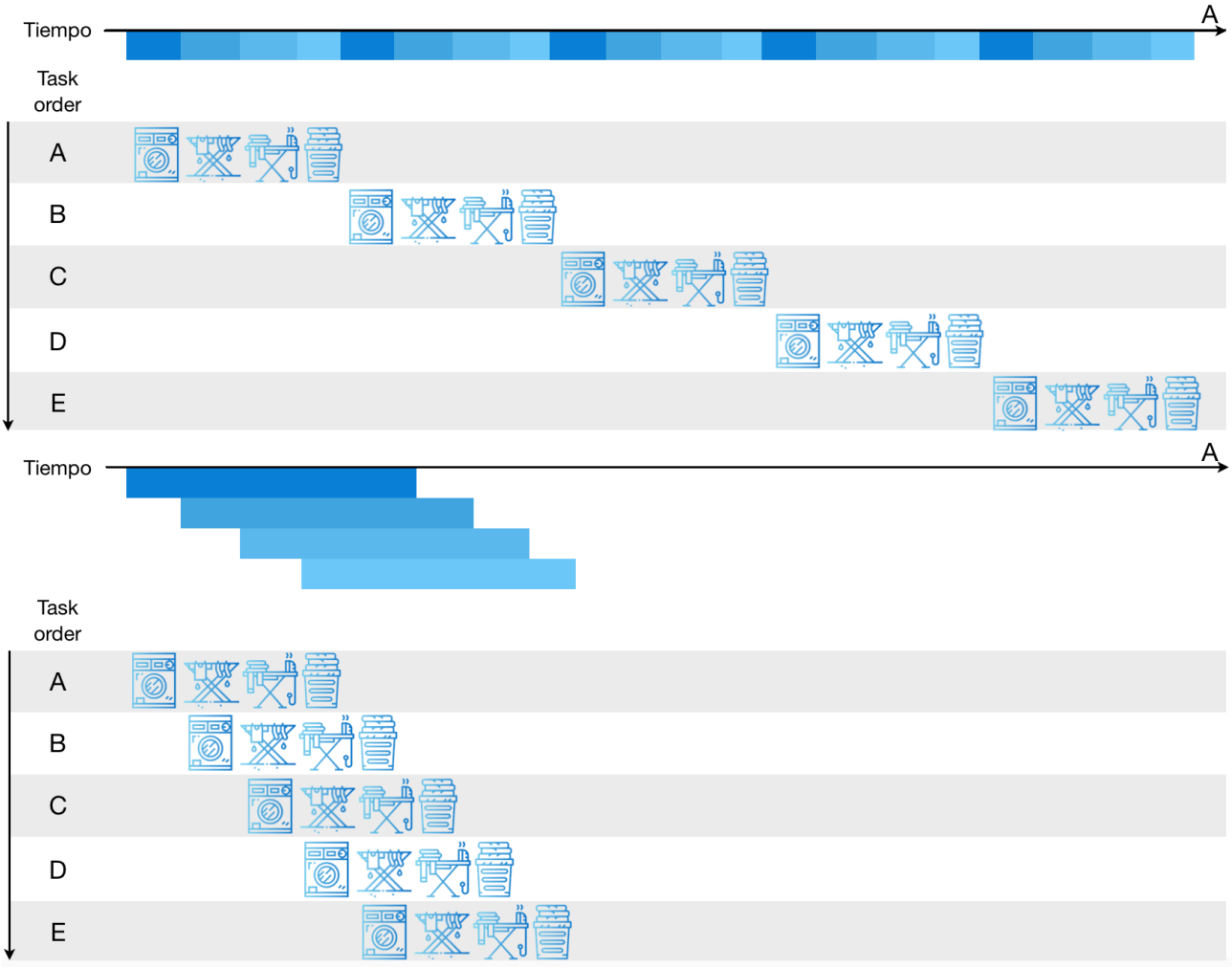

Figura 24. Analogía de la lavandería para la segmentación [9].

La realización de las cuatro tareas de forma no segmentada implica tener que terminarlas antes de volver a empezar el proceso, en cambio, realizarlas de forma segmentada supone un incremento en el rendimiento ya que una vez ser termina la primera tarea (lavar) y se pasa a la segunda (tender), se puede volver a iniciar la primera tarea sin necesidad de esperar a que se realicen las tres restantes (tender, planchar y doblar).

Cabe destacar que la primera vez que se inicia el proceso hay un retardo de cuatro ciclos, posteriormente hay un retardo de un solo ciclo (de forma ideal).

Este concepto podemos aplicarlo al RISC V dividiendo la ejecución de las instrucciones en las tareas siguientes:

- Búsqueda de la instrucción en la memoria de programa (ROM).
- Leer los registros y decodificar la instrucción.
- Ejecución
- Búsqueda de operandos en la memoria de datos (si es necesario).
- Almacenamiento de resultados (si es necesario)

Esta división de tareas puede aplicarse directamente al procesador en forma de etapas, dichas etapas se conocen como IF, ID, EX, MEM y WB. Es natural pensar que la segmentación debe realizarse en 5 etapas debido a que este es el número de tareas en las que se divide la ejecución de una instrucción, pero se puede realizar una segmentación de menos o más etapas.

Durante el establecimiento de la arquitectura segmentada se ha considerado la segmentación en cuatro etapas, pero como se ve en la *Figura 25*, la etapa EX es la que contiene el mayor retardo como se ha explicado en la sección 3.5 (Análisis temporal de las unidades funcionales) debido al multiplicador y la unidad aritmético-lógica. En caso de reducir una etapa aumentaría el retado empeorando la frecuencia de operación del procesador, por esta razón en este caso la segmentación se realiza en 5 etapas.

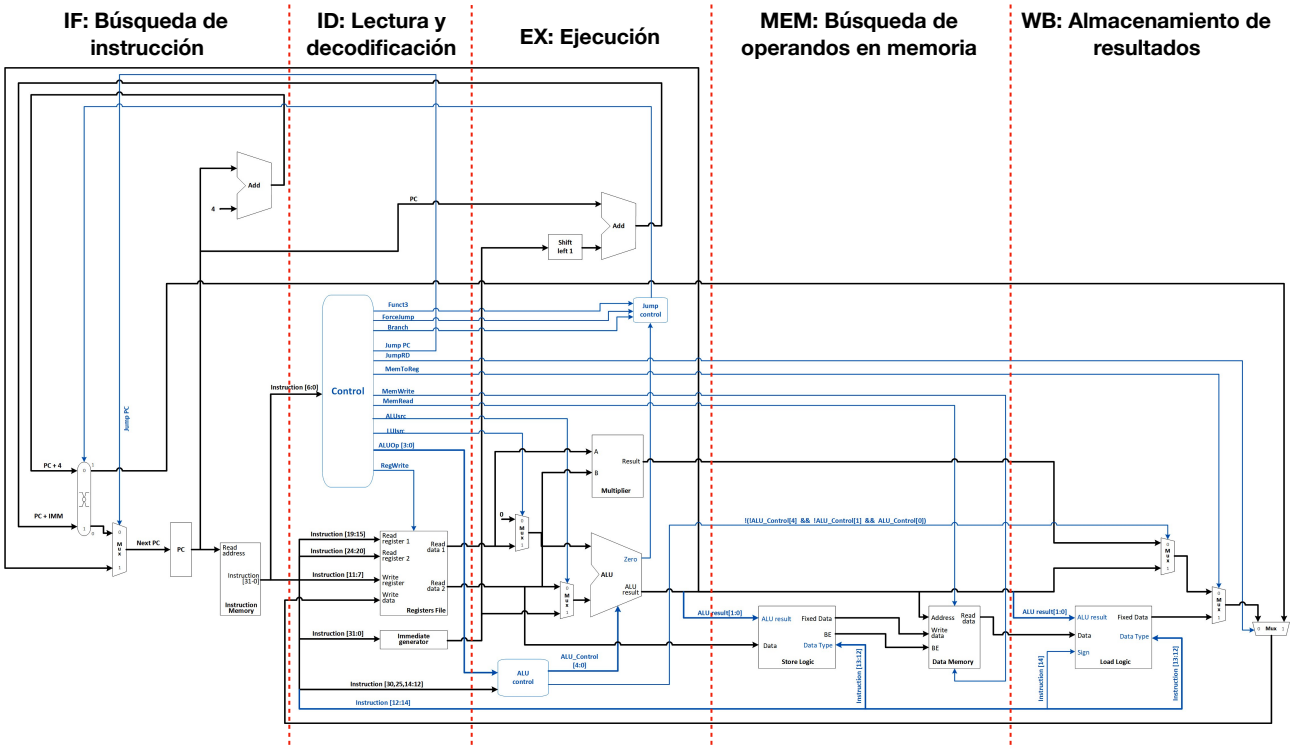

Figura 24. Core *Single Cycle* con propuesta de segmentación.

Como se ha comentado anteriormente, la segmentación mejora el rendimiento y es posible obtener el factor de mejora con respecto a la implementación no segmentada.

$$\text{Tiempo Medio Ejecución} = \frac{T_{\text{Ejecución}}}{\text{Nº instrucciones}} \qquad (4.1)$$

La ecuación 4.1 permite calcular el tiempo medio de ejecución de los programas en el procesador, es necesario obtener dicho valor para la implementación segmentada y para la implementación *single cycle*.

$$\textbf{Aceleración} = \frac{\textbf{T}_{\textbf{Medio sin segmentación}}}{\textbf{T}_{\textbf{Medio con segmentación}}} \qquad \textbf{(4.2)}$$

Con la ecuación 4.2 y el tiempo medio de ejecución de ambas implementaciones se obtiene el factor de aceleración conseguido por el procesador segmentado. En el Capitulo 7 se habla acerca de como medir el rendimiento del procesador y se realiza una comparación entre ambas implementaciones de forma más extensa.

4.2. Establecimiento de una implementación básica

Para llevar a cabo su implementación, se deben colocar registros (N_{ETAPAS} - 1) en el *data path* para conformar las etapas previstas en la *Figura 24.* En este caso se colocan cuatro registros. Además es recomendable que la memoria de instrucciones, la memoria de datos y el banco de registros incorporen lectura síncrona de modo que no es necesario que el valor leído pase por el registro de la etapa.

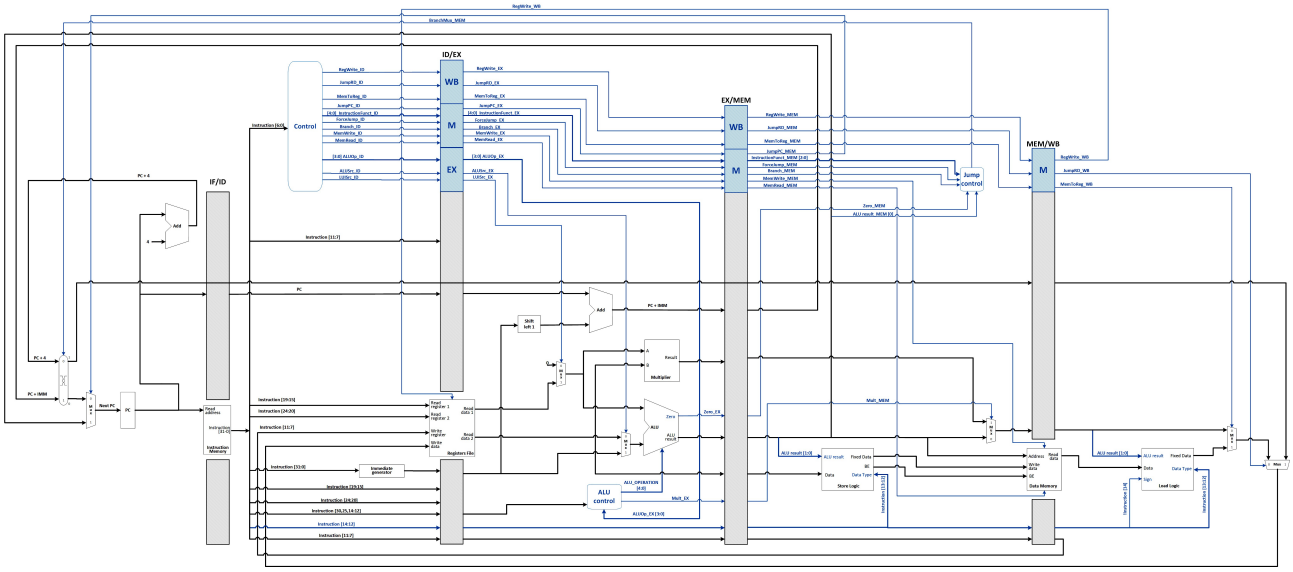

Figura 25. Implementación básica del core segmentado.

El *control path t*iene cuatro etapas de segmentación ya que hasta la etapa ID no se ha descodificado la instrucción, su segmentación consiste en actualizar las señales de control y pasarlas entre las etapas haciendo uso de los registros.

La *Figura 25* muestra la implementación básica del procesador segmentado y es importante destacar que no es del todo funcional debido a que no se han tenido en cuenta los riesgos de segmentación como la falta de recursos o que los datos no estén disponibles en el instante que se requieren. En la sección 4.3 se explican los riesgos de forma más completa y como solucionarlos.

4.3. Detección y control de riesgos

Los riesgos de segmentación impiden que se ejecute la instrucción en el ciclo de reloj que le corresponde de modo que la frecuencia de operación del procesador se ve afectada. Existen varios tipos de riesgos de segmentación y son los siguientes: riesgos estructurales, riesgos de datos y riesgos de control. En las siguientes sub-secciones se explica cada uno de ellos.

4.3.1. Riesgos estructurales

Los riesgos estructurales se producen por el paralelismo en la ejecución de las instrucciones que intentan utilizar de forma simultánea un mismo recurso hardware. Pueden producirse en:

- **Memoria**: Presenta un riesgo estructural cuando se intenta acceder a la memoria desde la etapa IF (Búsqueda de instrucción) y desde la etapa MEM (Búsqueda de operandos en memoria lectura/escritura). Se soluciona haciendo uso de dos memorias separadas, una para datos y otra para instrucciones. En esta implementación no de produce este riesgo estructural ya que RISC V utiliza memorias separadas para el programa y los datos [12][15].
- **Banco de registros**: Presenta un riesgo estructural cuando se intenta acceder al banco de registros desde la etapa de decodificación (ID) y la etapa de almacenamiento de datos (WB) ya que se utiliza la lectura/escritura de forma simultánea. Se soluciona implementando varios puertos en el banco de registros, en este caso dos puertos de lectura y un puerto de escritura. En esta implementación no se produce este riesgo estructural ya que tiene los puertos necesarios para realizar la lectura y escritura de forma simultánea [12][15].

4.3.2. Riesgos de datos

Los riesgos de datos se producen cuando la instrucción actual necesita los datos de la instrucción anterior. Existen varios tipos de dependencia de datos:

- ***Read After Write* (RAW)**: Este tipo de dependencia se presenta cuando la instrucción actual intenta usar un dato que aun no ha sido almacenado en el banco de registros y todavía se encuentra en la segmentación.

  ADD x3, x0, x7
  SUB x4, x3, x9

  Figura 26. Dependencia RAW [15].

  En este caso se produce una “dependencia" (terminología del compilador) debido a que la instrucción actual SUB requerirá el operando de x3 antes de que sea almacenado por la instrucción anterior en el banco de registros [12][15].

- ***Write After Read* (WAR)**: Este tipo de dependencia se presenta cuando la instrucción actual almacena el dato antes de ser leído por la instrucción anterior.

  ADD x4, x1, x5
  SUB x1, x2, x3

  Figura 27. Dependencia WAR [15].

  En este caso se produce una “anti-dependencia” (terminología del compilador) debido a que la instrucción actual SUB almacenará el operando de x1 antes de que sea leído por la instrucción anterior del banco de registros [12][15].

  Esta dependencia no se produce en la arquitectura del RISC V debido a que todas las instrucciones tienen el mismo número de etapas, las lecturas se producen siempre en la etapa de decodificación (ID) y las escrituras en la etapa de almacenamiento de resultados (WB).

- ***Write After Write* (WAW)**: Este tipo de dependencia se presenta cuando la instrucción actual almacena el dato antes de ser almacenado por la instrucción anterior.

  ADD x1, x2, x3
  SUB x1, x4, x5

  Figura 28. Dependencia RAW [15].

En este caso se produce una “dependencia de salida" (terminología del compilador) debido a que la instrucción actual SUB almacena el operando x1 antes antes de que ADD escriba el resultado en el banco de registros [12][15].

Esta dependencia no se produce en la arquitectura del RISC V debido a que todas las instrucciones tienen el mismo número de etapas y las escrituras siempre se realizan en la etapa de almacenamiento de resultados (WB).

Vistos los distintos tipos de dependencia de datos, el único que afecta a la arquitectura desarrollada es la de *Read After Write* (RAW) por lo que a continuación se explican las diferentes soluciones que se pueden adoptar.

Una posible solución puede alcanzarse mediante software, el compilador puede introducir las instrucciones NOP que sean necesarias justo después de la instrucción que cause o pueda causar el riego, también existe la posibilidad de reordenar el código de forma que no se produzca la dependencia (Enfoque de los primeros RISC) [12][15].

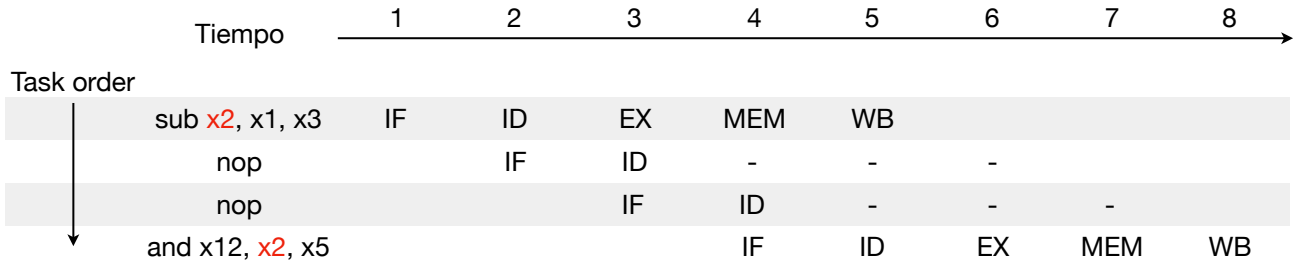

Figura 29. Solución por software NOP [15].

Otra forma de solucionar el problema es mediante hardware utilizando una técnica conocida como interbloqueo que consiste en detener la segmentación (*Stall*) introduciendo una burbuja hasta que ya no se produzca el riesgo. La detención de la segmentación hace que la instrucción permanezca en una misma etapa sin avanzar. Existe una solución más sofisticada mediante hardware conocida como adelantamiento (*Forwarding*) que consiste en utilizar el dato en cuanto esté listo en vez de esperar a que sea almacenado en el banco de registros, esta solución requiere hardware adicional. El adelantamiento de datos no permite solucionar todos los riesgos, por ello se combina con el interbloqueo [9] [12] [15].

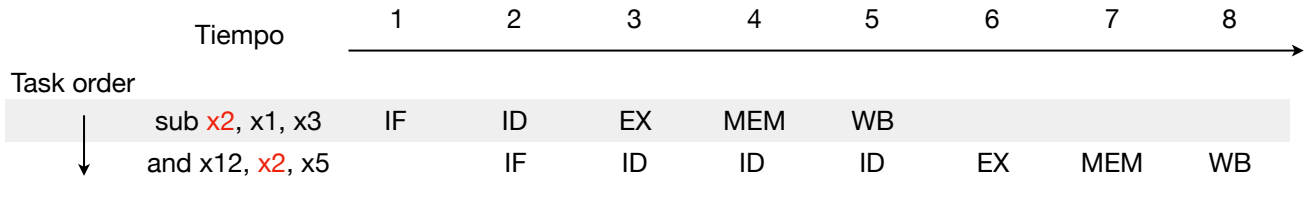

Figura 30. Solución por *hardware STALL* (Burbuja) [15].

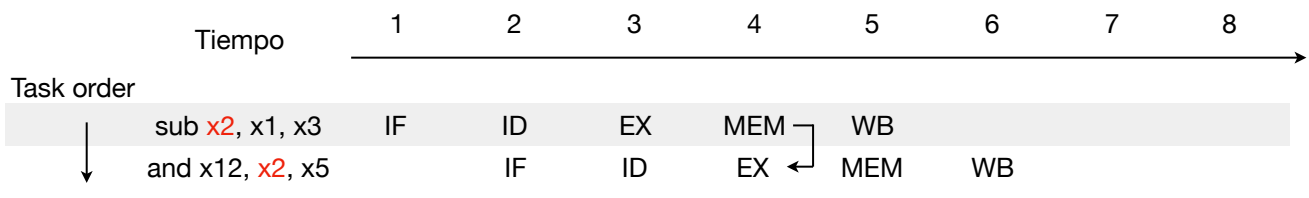

Figura 31. Solución por hardware FORWARDING (Adelantamiento).

Para esta implementación se soluciona el problema mediante hardware ya que la solución hardware ralentiza mucho el funcionamiento. Se realiza el adelantamiento de datos desde la etapa MEM y WB a la unidad aritmético-lógica para no tener que esperar a que el resultado sea almacenado en el banco de registros y pueda ser utilizado.

4.3.3. Riesgos de control

Los riesgos de control se producen cuando la instrucción es un salto condicional o incondicional. Si se trata de un salto incondicional, las instrucciones posteriores tienen que ser anuladas y ejecutar la instrucción que se encuentra en la dirección de salto. En caso de que se trate de un salto condicional, las instrucciones posteriores deben ejecutarse de forma especulativa y en caso de que se produzca el salto debe vaciarse la segmentación para que dichas instrucciones no tengan efecto alguno.

Task order \ Tiempo	1	2	3	4	5	6	7	8	9
jump dir	IF	ID	EX	MEM	WB				
kill		-	-	-	-	-			
kill			-	-	-	-	-		
kill				-	-	-	-	-	
and x12, x2, x5		⧗	⧗	⧗	IF	ID	EX	MEM	WB

Figura 32. Anulación de las instrucciones posteriores para saltos incondicionales.

La ventana de retado producida por este riesgo de segmentación es de 3 ciclos de reloj ya que el contador de programa se actualiza al final de la fase MEM y se realiza la búsqueda de la instrucción en el ciclo siguiente. La *Figura 32* y la *Figura 33* muestran el proceso de ejecución explicado para los saltos incondicionales y los saltos condicionales.

Task order \ Tiempo	1	2	3	4	5	6	7	8	9
beq x0,x0, dir	IF	ID	EX	MEM	WB				
Instrucción 2		IF	ID	EX	-	-			
Instrucción 3			IF	ID	-	-	-		
Instrucción 4				IF	-	-	-	-	
and x12, x2, x5		⧗	⧗	⧗	IF	ID	EX	MEM	WB

Figura 33. Ejecución especulativa para saltos condicionales [15].

El impacto de los saltos puede reducir las prestaciones de forma significativa, para solucionarlo es necesario determinar cuanto antes la dirección de salto y si debe producirse. Una posible solución es mover la lógica combinacional necesaria a la etapa ID, de este modo solo se obtendría un ciclo de reloj de penalización. Además, existe una solución conocida como predicción de saltos que trata de predecir la instrucción que debe ejecutarse según el histórico de saltos realizado [9][15].

Esta implementación utiliza la ejecución especulativa cuyo retardo es de tres ciclos de reloj aunque queda abierta la posibilidad de mejorar el diseño e implementar la predicción de saltos en un futuro.

4.4. Unidades para el control de riesgos

En esta sección se va a explicar la forma de diseñar las unidades para controlar los riesgos de segmentación anteriormente mencionados.

Las unidades a diseñar son la unidad de adelantamiento de datos, la unidad de limpieza de la segmentación y la unidad de detección de riesgos. Esta última unidad es muy importante para realizar la correcta implementación de los saltos y las detenciones (*stalls*).

4.4.1. Unidad de adelantamiento (*Forwarding Unit*)

La unidad de adelantamiento de datos es la encargada de que la unidad aritmético-lógica disponga de los datos necesarios en el instante preciso generando las señales de control, necesarias para controlar los multiplexores de entrada a la ALU.

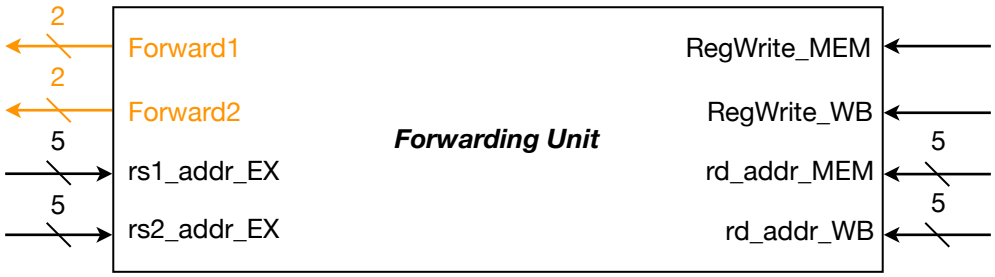

Figura 34. Unidad de adelantamiento de datos.

Su funcionamiento consiste en detectar si alguno de los registros fuente rs1 o rs2 de la etapa EX coincide con el registro de destino en las etapas MEM o WB. También es necesario comprobar que la señal de habilitación de escritura este activada.

Señal de control	Fuente	Explicación
Forward1 = 00	ID/EX	El operando 1 de la ALU proviene del banco de registros.
Forward1 = 10	EX/MEM	El operando 1 de la ALU proviene de un calculo anterior desde la etapa MEM
Forward1 = 01	MEM/WB	El operando 1 de la ALU proviene de un calculo anterior o la memoria de datos desde la etapa WB
Forward2 = 00	ID/EX	El operando 2 de la ALU proviene del banco de registros.
Forward2 = 01	EX/MEM	El operando 2 de la ALU proviene de un calculo anterior desde la etapa MEM
Forward2 = 10	MEM/WB	El operando 2 de la ALU proviene de un calculo anterior o la memoria de datos desde la etapa WB

Tabla 14. Señales de control para los multiplexores de adelantamiento [9].

La *Tabla 14* muestra el valor de las señales de control en función del tipo de adelantamiento que sea necesario realizar en cada uno de los multiplexores que hay a la entrada de la ALU, los posibles adelantamientos son MEM-ALU y WB-ALU.

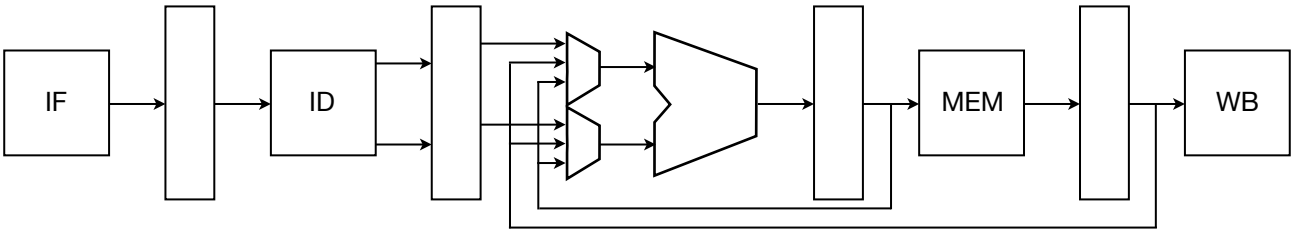

Figura 35. Diagrama de implementación del adelantamiento de datos.

En la *Figura 35* se aprecia el diagrama con los multiplexores para el adelantamiento de datos, dicho diagrama no muestra la unidad de adelantamiento que controla las señales de control de los multiplexores. La unidad de adelantamiento se muestra integrada en la implementación del procesador en la sección 4.5.

La realización de la unidad de adelantamiento de datos en SystemVerilog es sencilla, se basa en el uso de bloques *if* anidados formando un bloque combinacional. La *Tabla 15* muestra parte del código necesario para su implementación.

```
always_comb
    begin
        if((RegWrite_MEM==1) && (rd_addr_MEM!='0) && (rs1_addr_EX==rd_addr_MEM))
        Forward1=2'b10;
        else if ((RegWrite_WB==1) && (rd_addr_WB!='0) && (rs1_addr_EX==rd_addr_WB))
        Forward1=2'b01;
        else
        Forward1=2'b00;

        if((RegWrite_MEM==1)&&(rd_addr_MEM!='0) && (rs2_addr_EX==rd_addr_MEM))
        Forward2=2'b10;
        else if ((RegWrite_WB==1)&&(rd_addr_WB!='0) &&(rs2_addr_EX==rd_addr_WB))
        Forward2=2'b01;
        else
        Forward2=2'b00;
    end
```

Tabla 15. Implementación de la unidad de adelantamiento en SystemVerilog [9].

4.4.2. Unidad de limpieza de la segmentación (*Clear Pipeline*)

Esta unidad es la encargada de generar la señal de control que vacía el cauce de la segmentación en caso de que se realice un salto.

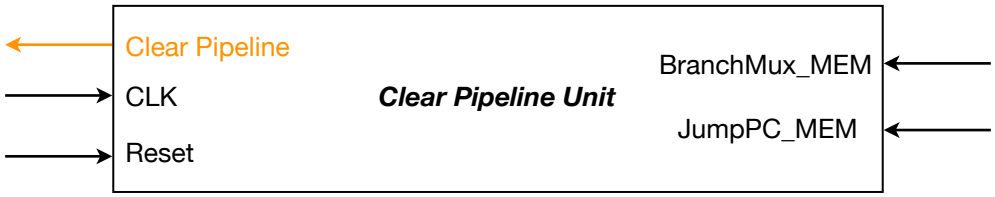

Figura 34. Unidad de limpieza de la segmentación.

Su funcionamiento consiste en detectar si finalmente se produce el salto mediante las señales BranchMux_MEM y JumpPC_MEM. En caso de detectarse, se activa la señal *Clear Pipeline* que anula las señales de control de los registros de segmentación ID/EX y EX/MEM de forma que las instrucciones ejecutadas de forma especulativa no surten efecto.

4.4.3. Unidad de detección de riesgos (*Hazard Detection Unit*)

Esta unidad es la encargada de generar las señales de control que posibilitan la ejecución de los saltos de y las detenciones de forma correcta. Dichas señales de control hacen que las señales de control del registro ID/EX sean cero, introduciendo así instrucciones nulas en las etapas EX, MEM y WB. Además, impiden la actualización del contador de programa (PC) de modo que la instrucción se decodifica constantemente mientras sea necesario.

La *Figura 35* muestra la unidad de detección de riesgos y a continuación se explica cada una de las señales de salida para entender mejor su funcionamiento.

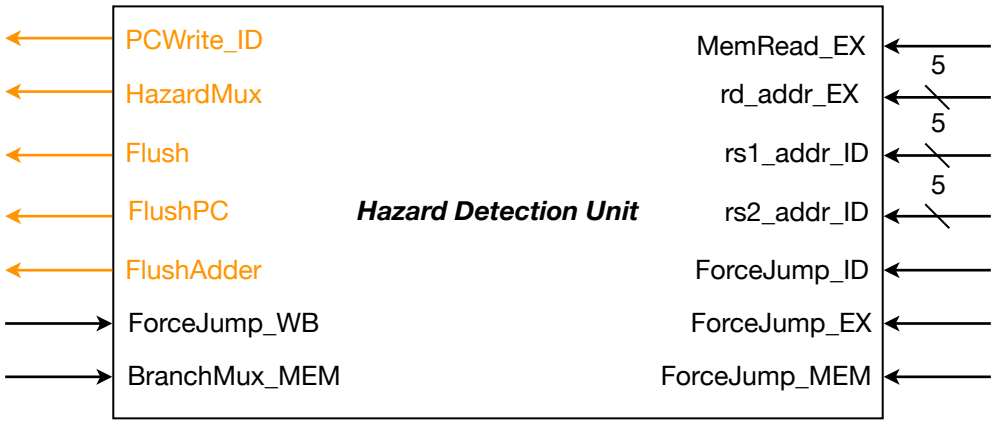

Figura 35. Unidad de detección de riesgos.

El funcionamiento de cada una de las señales de salida es el siguiente:

- **PCWrite_ID**: Señal de control que posibilita la actualización del PC. Se habilita cuando la señal Memread_EX esta activa, no se trata de un salto y la dirección de destino de la etapa EX coincide con la dirección de lectura del puerto rs1 o rs2.
- **HazardMux**: Señal de control que posibilita la anulación de las instrucciones a partir de la etapa ID en caso de producirse una detención. Se habilita cuando la señal Memread_EX esta activa, no se trata de un salto y la dirección de destino de la etapa EX coincide con la dirección de lectura del puerto rs1 o rs2.
- **Flush**: Señal de control que mantiene el PC y descarta las actualizaciones del mismo. Se habilita en caso de que la señal ForceJump se encuentre activa en la etapa EX, MEM o WB.
- **FlushPC**: Señal de control que mantiene el PC y descarta las actualizaciones del mismo en el registro de segmentación IF/ID. Se habilita en caso de que la señal ForceJump se encuentre activa en la etapa ID o EX.
- **FlushAdder**: Señal de control que descarta las actualizaciones del PC en la entrada del sumador de la etapa IF/ID. Se habilita en caso de que la señal ForceJump se encuentre activa en la etapa ID,EX o MEM.

La forma de implementar la unidad de detección de riesgos en SystemVerilog se muestra en la *Tabla 16.*

```
assign Flush = (ForceJump_EX || ForceJump_MEM || ForceJump_WB);
assign FlushPC = (ForceJump_ID || ForceJump_EX);
assign FlushAdder = (ForceJump_ID || ForceJump_EX || ForceJump_MEM);

always_comb
    if((MemRead_EX==1'b1) && ( (rd_addr_EX == rs1_addr_ID) || (rd_addr_EX ==
        rs2_addr_ID) ) && !BranchMux_MEM ) begin
        PCWrite_ID=1'b0;
        HazardMux=1'b1;
    end else begin
        PCWrite_ID=1'b1;
        HazardMux=1'b0;
    end
```

Tabla 16. Implementación de la unidad de detección de riesgos en SystemVerilog [9].

4.5. Implementación segmentada con control de riesgos

La combinación de la implementación básica de la sección 4.2 junto y las unidades desarrolladas en la sección 4.4 resulta en la implementación del ISA RV32IM segmentado funcional.

A diferencia de la implementación básica segmentada, en este caso tanto la unidad aritmético-lógica como la unidad de multiplicación son síncronas de modo que no es necesario pasar los datos por el registro EX/MEM.

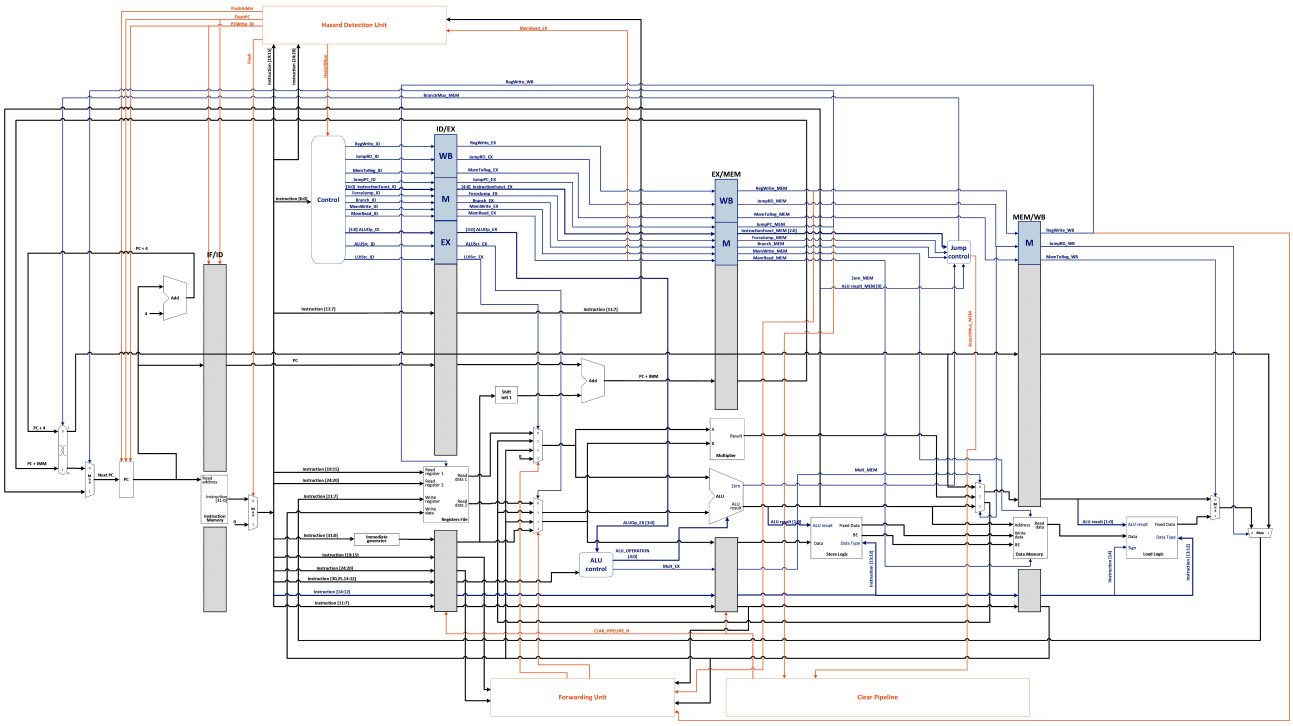

Figura 36. Core del procesador segmentado con control de riesgos.

En la *Figura 36* puede verse el core segmentado con las distintas unidades. Se aprecian tres colores distintos: azul para el *control path*, naranja para las unidades de control de riesgos (*control path*) y en negro las del *data path*.

Para la obtención de la máxima frecuencia de operación de la implementación segmentada se debe compilar desde Quartus Prime excluyendo la unidad de multiplicación. Esto se debe a que existe conexión física con diversos elementos del procesador y el compilador establece caminos falsos.

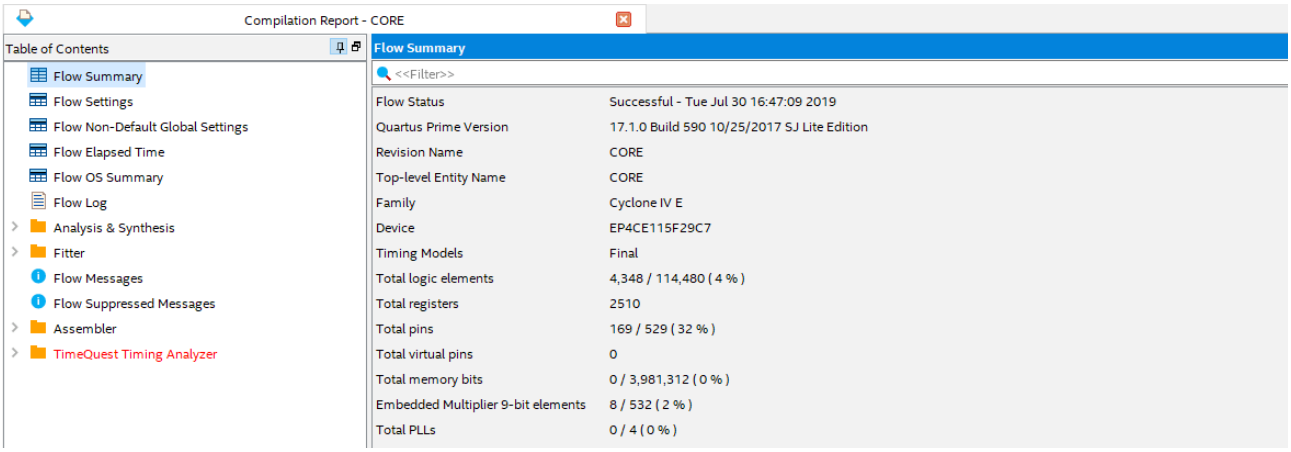

Figura 37. Recursos utilizados por la implementación.

Por ejemplo el compilador entiende que la dirección de salto calculada podría entrar como operando en el multiplicador, pero esto nunca va a suceder, ya que el multiplicador sólo se utiliza por las cuatro instrucciones que tiene asociadas. Esto provoca que la frecuencia de operación obtenida no sea real.

El análisis frecuencial realizado por *TimeQuest Time Analyzer* sin tener en cuenta la unidad de multiplicación (se recuerda que la frecuencia de operación del multiplicador es de 98,43 MHz), muestra una frecuencia máxima de operación del procesador de 72,59 MHz. Esto conlleva un incremento del 61% en la frecuencia de operación con respecto a la implementación *single cycle.*

Una vez obtenida la frecuencia de operación, se realiza la compilación con la unidad de multiplicación incluida, de forma que se comprueba los recursos utilizados para esta implementación. Observado la *Figura 37* se aprecia que solo se ha incrementado el uso de recursos (elemetos lógicos) en un 1%.

4.6. Verificación

Para verificar la implementación *Multi Cycle* se ha reutilizado el testbench en de la implementación *Single Cycle* realizando las modificaciones necesarias. La estructura del directorio de trabajo también es la misma que la mostrada en la *Figura 19.*

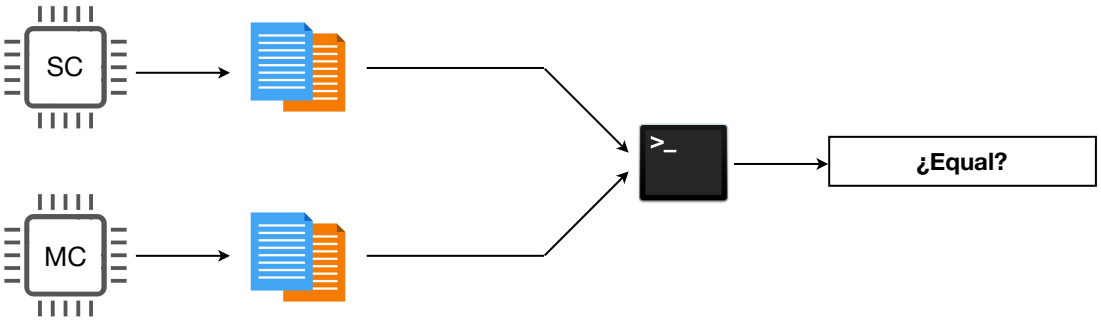

Figura 38. Diagrama simplificado del *testbench* para el procesador segmentado.

En este caso, el modelo de referencia utilizado para comprobar que el procesador segmentado funciona correctamente es el procesador monociclo implementado en el capítulo 3. Dado que comparar ciclo a ciclo el funcionamiento de ambos procesadores es complejo debido a que se deben sincronizar para que coincidan las salidas, se ha optado por utilizar otro método de comparación.

El método consiste en ejecutar el programa y una vez finalizado volcar el contenido de la memoria de datos y el banco de registros en dos ficheros .txt independientes (registerFile.txt y dataMemory.txt) para cada uno de los procesadores. Después se realiza una llamada al sistema operativo mediante el uso de $system(). La *Figura 38* muestra este proceso de forma simplificada.

La comparación de los archivos de texto se realiza mediante el comando fc (*file compare*), simplemente se tiene que indicar la ruta y el nombre de los ficheros para ejecutar el comando. En caso de que los ficheros no coincidan, se especificará cual ha sido la diferencia encontrada.

En la *Tabla 17* se muestra como utilizar el comando fc, es necesario recalcar que en Windows es necesario poner la doble barra invertida (*backslash*) “\\” para que no sea interpretada como una secuencia de escape. Los sistemas basados en UNIX como macOS y Linux no necesitan hacer uso de la doble contra invertida ya que la ruta en estos sistemas esta delimitada por “/“.

```
$system("fc path1\\registerFile_SC.txt path2\\registerFile_MC.txt”);
```

Tabla 17. Comando necesario para comparar dos ficheros en Windows desde el TB.

En caso desde ejecutar el *testbench* desde Linux el comando sería ligeramente distinto, tal y como se muestra en la *Tabla 18.*

```
$system("diff path1/registerFile_SC.txt path2/registerFile_MC.txt");
```

Tabla 18. Comando necesario para comparar dos ficheros en sistemas UNIX desde el TB.

A continuación se puede observar en la *Figura 39* la respuesta generada por el comando fc en el caso de que la ejecución de ambos procesadores coincida, es decir, el estado final de la memoria de datos y el banco de registros sea el mismo.

```
================================================================================================================================

                                        COMPARACION DE RESULTADOS

================================================================================================================================

Comparando archivos G:\TAREA2\PIPELINED5\OUTPUT_FILES\registerFile_SC.txt y G:\TAREA2\PIPELINED5\OUTPUT_FILES\REGISTERFILE_MC.TXT
FC: no se han encontrado diferencias

Comparando archivos G:\TAREA2\PIPELINED5\OUTPUT_FILES\dataMemory_SC.txt y G:\TAREA2\PIPELINED5\OUTPUT_FILES\DATAMEMORY_MC.TXT
FC: no se han encontrado diferencias

================================================================================================================================
```

Figura 39. Mensaje de salida proporcionado por el sistema cuando la ejecución coincide.

El sistema indica que no se han encontrado diferencias en ninguno de los ficheros, este mensaje asegura que la ejecución de la implementación segmentada ha sido correcta.

Durante la ejecución del comando pueden recibirse distintos códigos de respuesta y son los mostrados a continuación:

- **-1**: La sintaxis del comando es incorrecta.
- **0**: Ambos archivos son idénticos.
- **1**: Se han encontrado discrepancias entre los archivos.
- **2**: Alguno de los archivos indicados no se encuentra.

La *Figura 40* muestra la respuesta obtenida en caso de encontrarse discrepancias entre los archivos, se observa que el código de error obtenido es 1 y la linea REG_FILE_[3] es distinta.

```
================================================================================================================================

                                        COMPARACION DE RESULTADOS

================================================================================================================================
(vsim-50) A call to system(fc G:\Tarea2\pipelined5\output_files\registerFile_SC.txt G:\Tarea2\pipelined5\output_files\registerFile_
_MC.txt) returned error code '1'.
The logfile contains the following messages:

Comparando archivos G:\TAREA2\PIPELINED5\OUTPUT_FILES\registerFile_SC.txt y G:\TAREA2\PIPELINED5\OUTPUT_FILES\REGISTERFILE_MC.TXT
***** G:\TAREA2\PIPELINED5\OUTPUT_FILES\registerFile_SC.txt
REGISTER_FILE_[2] = 0x00000002
REGISTER_FILE_[3] = 0x00000032
REGISTER_FILE_[4] = 0x00000000
***** G:\TAREA2\PIPELINED5\OUTPUT_FILES\REGISTERFILE_MC.TXT
REGISTER_FILE_[2] = 0x00000002
REGISTER_FILE_[3] = 0x00000078
REGISTER_FILE_[4] = 0x00000000
*****

Comparando archivos G:\TAREA2\PIPELINED5\OUTPUT_FILES\dataMemory_SC.txt y G:\TAREA2\PIPELINED5\OUTPUT_FILES\DATAMEMORY_MC.TXT
FC: no se han encontrado diferencias

================================================================================================================================
```

Figura 40. Mensaje de salida proporcionado por el sistema cuando la ejecución no coincide.

Este método permite encontrar discrepancias entre los procesadores y localizar el fallo de ejecución gracias a la comparación de los ficheros de texto.

4.6.1. *Bubble Sort*

La verificación de la implementación segmentada con el programa desarrollado en ensamblador *Bubble Sort* es perfecto para comprobar que han sido resueltos los riesgos de segmentación, ya que dicho programa incluye saltos condicionales e incondicionales, cargas, almacenamientos y requiere adelantamiento de datos.

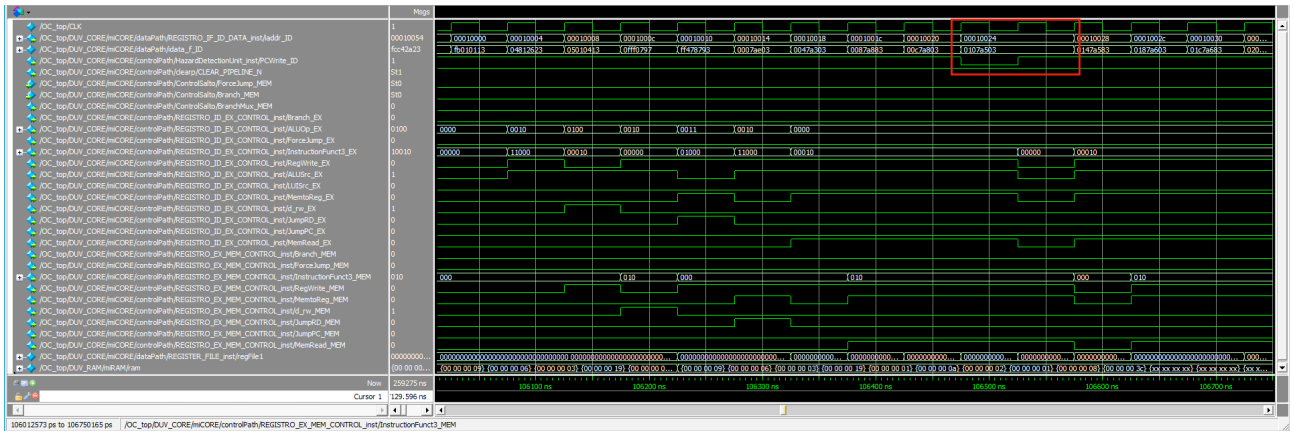

Figura 41. Detención de la instrucción y adelantamiento de datos.

El primer riesgo de segmentación, se observa en la *Figura 41* y se produce porque la instrucción posterior trata de leer un registro después de una instrucción load que va a escribir en ese mismo registro, por tanto es necesario introducir una burbuja ya que no se puede realizar el adelantamiento de datos.

El segundo riesgo de segmentación que se observa en la *Figura 42* viene dado por un salto incondicional (0c00006f -> jal x0, 192), por tanto las instrucciones posteriores se anulan y se producen tres ciclos de retardo en la ejecución.

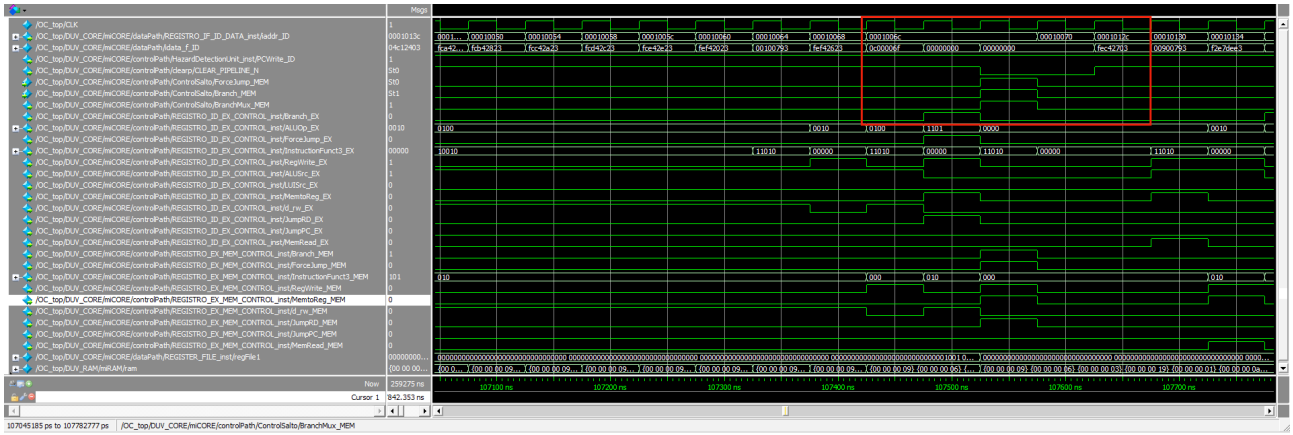

Figura 42. Riesgo de control provocado por salto incondicional.

Posteriormente se puede encontrar en la ejecución del programa un salto condicional (f2e7dee3 -> bge x15, x14, -196), se aprecia en la *Figura 42* que se ejecutan las instrucciones posteriores de forma especulativa y una vez se determina que se va a efectuar el salto, la señal *clear pipeline se activa y* hace que se descarten las instrucciones para que no surtan efecto.

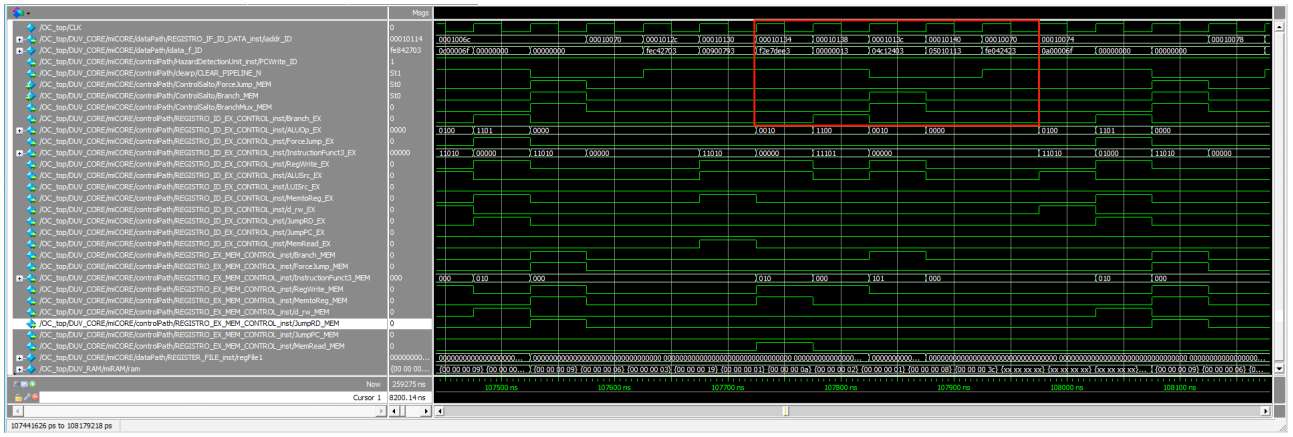

Figura 43. Riesgo de control provocado por salto condicional.

Los riesgos de datos y su corrección con la unidad de adelantamiento de datos se muestra en la sub-sección posterior ya que se aprecia la función de esta unidad con mayor claridad.

Finalmente, la ejecución del programa concluye con éxito y ordena los numero en memoria tal y como se comenta en la verificación de la implementación *single cycle.*

4.6.2. Multiplicación

Para esta prueba se ha desarrollado un programa en ensamblador (*Figura 44*), cuya función es multiplicar el valor del registro x1 por el valor del registro x2 y comparar si el resultado es mayor que el valor almacenado en el registro x20. En caso de ser menor, el contenido de x1 se vuelve a multiplicar por x2, esto se repite en bucle hasta superar el umbral del registro x20. De este modo se puede comprobar el correcto funcionamiento de la unidad de adelantamiento de datos.

```
INICIO:
MUL x1, x2, x1
BLT x1, x20, INICIO
```

Figura 44. Programa de multiplicación en ensamblador.

En la *Figura 45* se comprueba el correcto funcionamiento de la unidad de adelantamiento de datos, en el primer ciclo se introducen los valores 0x2 y 0x14 en el multiplicador, en el segundo ciclo se obtiene el resultado 0x28 y se adelanta el dato a la salida del multiplexor, para finalizar, en el tercer ciclo el dato esta disponible en el instante preciso que se necesita.

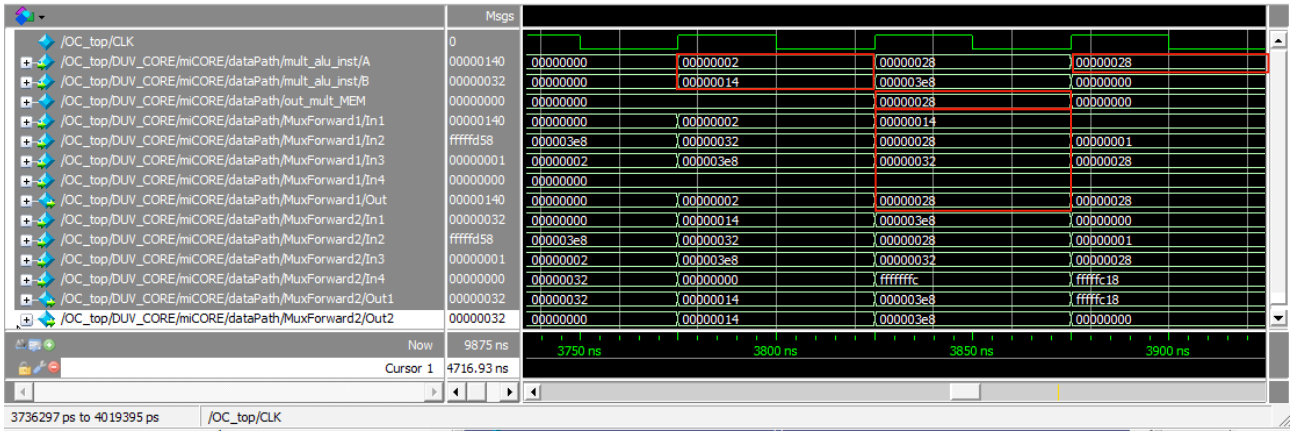

Figura 45. Riesgo de datos provocado por instrucciones dependientes.

Capítulo 5. Soporte para programación y verificación

Desarrollar programas en lenguaje ensamblador es complicado y costoso, por ello esta sección muestra el proceso necesario para convertir un programa escrito en un lenguaje de alto nivel como C en código máquina para que el procesador desarrollado sea capaz de ejecutarlo.

5.1. Introducción al proceso de compilación

Lo primero que hay que tener en cuenta es que el programa desarrollado en código C no será ejecutado en la plataforma de desarrollo, si no que será ejecutado en el procesador desarrollado. Por este motivo el compilador recibe el nombre de *cross compiler* ya que los binarios generados para la ejecución han sido creados para una plataforma diferente a la que los ha generado.

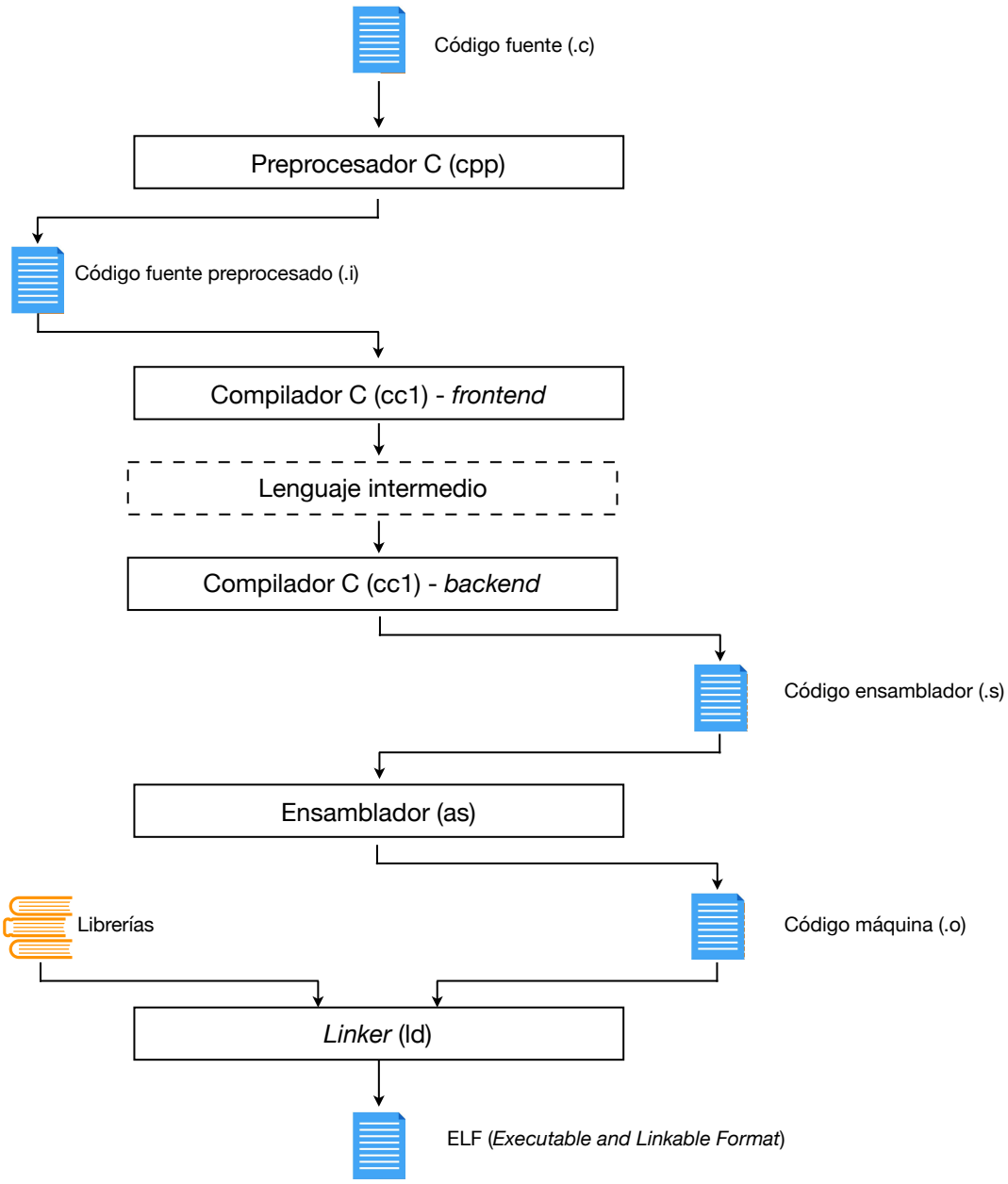

Figura 46. Cadena de herramientas (*toolchain*) [16].

La *Figura 46* detalla la cadena de herramientas (*toolchain*) necesaria para realizar la conversión del código de alto nivel a código máquina y a continuación se hace una breve introducción sobre ellas:

- **Preprocesador (cpp)**: se encarga de eliminar todos los comentarios del código y reemplazar todos los comandos que comienzan por # con las cabeceras correspondientes. Se obtiene un fichero .i una vez finalizado el preprocesado [16].
- **Compilador (cc1)**: es el encargado de transformar el código C a código ensamblador que pueda ser tratado más adelante. El compilador se puede dividir en dos partes (*front-end* y *back-end*), aunque normalmente estas partes van integradas en una única herramienta [16].

  La parte *front-end* traduce el lenguaje C a un lenguaje propietario intermedio que es independiente tanto de la plataforma de origen como de la plataforma de destino. En cambio, la parte *back-end* transforma el lenguaje intermedio a código ensamblador, el cual es dependiente de la plataforma de destino. El fichero generado tras la ejecución de la herramienta tiene extension .s [16].
- **Ensamblador (as)**: es el encargado de transformar el código ensamblador a código objeto. El fichero generado no contiene direcciones absolutas sino que contiene marcadores que serán reemplazados por el *linker* por direcciones reales. Esta herramienta también depende de la plataforma de destino y el fichero generado tiene el formato .o [16].
- ***Linker* (ld)**: es la última herramienta de la cadena, unifica en un único archivo todos los ficheros (algunos de esos ficheros pueden ser bibliotecas externas), reemplazando los marcadores por direcciones absolutas y vinculando llamadas a funciones entre otras cosas [16].

  A la hora de utilizar el *linker* es necesario proporcionarle un *script* que establezca la dirección de inicio de las secciones para coindicir la disposición de memoria descrita en la sub-sección 2.4.2. En este caso el script es el que se muestra en la *Tabla 19*.

```
SECTIONS {
              . = 0x00010000;
              .text : { * (.text); }

              . = 0x10000000;
              .rodata : { * (.rodata); }
              .data : { * (.data); }
              .bss : { * (.bss); }
}
```

Tabla 19. *Script* con las secciones para el *linker*.

  Una vez ejecutada la herramienta, se obtiene un fichero .elf, el cual es un formato de archivo para ejecutables. Este tipo de archivo contiene varias cabeceras, la primera de ellas es la cabecera ELF, que contiene la información general sobre el fichero. Contienen las posiciones de las cabeceras de las distintas secciones vistas anteriormente y cabeceras de programa. Cada una de las cabeceras especifica de forma unívoca la posición de los datos de cada una de las secciones dentro del archivo [17].

5.2. Instalación y configuración de una *tool-chain* para RISC V

Una vez comprendido el proceso de compilación, se puede realizar la instalación y configuran de la cadena de herramientas que permite ejecutar el código desarrollado en C a código máquina. La herramienta que se

va a utilizar se llama GCC (GUN *Compiller Collection*), es una colección de herramientas muy utilizada en los sistemas UNIX como Linux y macOS [18].

Mediante el siguiente enlace https://github.com/riscv/riscv-gnu-toolchain se accede al repositorio que contiene la cadena de herramientas GCC con compatibilidad para RISC V. En este documento se van a seguir los pasos necesarios para instalar la *Toolchain* en macOS, para instalarla en Windows o Linux simplemente se deben seguir los pasos indicados en el enlace especificado.

En primera instancia, es necesario clonar el repositorio que contiene los archivos necesarios para la instalación.

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

Tabla 20. Comando del terminal para clonar el repositorio.

Existen varios paquetes estándar necesarios para configurar la *toolchain*, en macOS se usa el gestor de paquetes Homebrew para instalar las dependencias necesarias mediante el comando mostrado en la *Tabla 21*.

```
$ brew install gawk gnu-sed gmp mpfr libmpc isl zlib expat
```

Tabla 21. Comando del terminal para instalar las dependencias con Homebrew.

Para finalizar el proceso, es necesario elegir una ruta de instalación y en este caso se ha elegido /opt/riscv por lo que será necesario añadir la ruta /opr/riscv/bin al PATH del sistema con el fin de ejecutar las herramientas de forma cómoda. Una vez elegida la ruta, se ejecutan los comandos de la *Tabla 22* para realizar la instalación.

```
$ ./configure –prefix=/opt/riscv
$ make
```

Tabla 22. Instalación de la *toolchain*.

Ahora ya podemos compilar el código C para ser ejecutado en el procesador, pero cabe destacar que por defecto la herramienta realiza la compilación para arquitecturas de 64 bits y por eso es necesario especificar que nuestra arquitectura es de 32 bits a la hora de ejecutar los comandos.

5.3. Uso de la *tool-chain* para RISC V mediante el terminal

El directorio de compilación contiene un fichero con código C y una carpeta llamada linkerScript que contiene el *script* de la *Tabla 19.*

Debido a que RISC V tiene diversas configuraciones, es necesario indicar mediante la opción -march la arquitectura para la cual queremos realizar la compilación que en este caso se realiza para la rv32im. Además, es necesario especificar las reglas de llamada de la función ABI la arquitectura de 32 bits mediante la opción -mabi, indicando como argumento ilp32. Esto indica que las variables int y long son de 32 bits. Es necesario incluir la opción -static para que el código necesario para ejecutar las funciones estén en un único fichero y pueda ser ejecutado sin problemas [18].

```
$ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -mtune=rocket -mcmodel=medany -
msmall-data-limit=8 -mstrict-align -save-temps=obj -static -c bubbleSort.c -o
bubbleSort.o
```

Tabla 23. Obtención del fichero .o.

Obtenido el código maquina, debe ejecutarse el comando que llama al *linker* con el fin de conseguir el archivo ejecutable.

```
$ riscv64-unknown-elf-ld -m elf32lriscv -T linkerScript/sections.ld  bubbleSort.o -o
bubbleSort.elf
```

Tabla 24. Obtención del fichero .elf.

El archivo obtenido no puede ser cargado directamente en el procesador ya que necesitamos separarlo en un fichero para la memoria de instrucciones y un fichero para la memoria de datos. Además cada linea del archivo contiene 128 bits, por lo que antes de cagar el fichero es necesario realizar algunos cambios.

```
7f45 4c46 0101 0100 0000 0000 0000 0000
0200 f300 0100 0000 0000 0100 3400 0000
2021 0000 0000 0000 3400 2000 0200 2800
0800 0700 0100 0000 0010 0000 0000 0100
0000 0100 4801 0000 4801 0000 0500 0000
0010 0000 0100 0000 0020 0000 0000 0010
0000 0010 2800 0000 2800 0000 0400 0000
0010 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
   .    .    .    .    .    .    .    .
   .    .    .    .    .    .    .    .
   .    .    .    .    .    .    .    .
0900 0000 0300 0000 0000 0000 0000 0000
c820 0000 1300 0000 0000 0000 0000 0000
0100 0000 0000 0000 1100 0000 0300 0000
0000 0000 0000 0000 db20 0000 4400 0000
0000 0000 0000 0000 0100 0000 0000 0000
```

Tabla 25. Contenido del fichero elf.

Partiendo del archivo ELF se debe realizar un volcado de la sección de texto que contiene el programa a ejecutar en un fichero .txt separado. De forma análoga debe realizarse para la sección *rodata, data* y *bss* . Para ello se hace uso de la herramienta objcopy tal y como se aprecia en la *Tabla 26* y la *Tabla 27*.

```
$ izansegarra$ riscv64-unknown-elf-objcopy --dump-section .text=text.txt
bubbleSort.elf
```

Tabla 26. Contenido del fichero elf.

```
$ riscv64-unknown-elf-objcopy --dump-section .rodata=rodata.txt --dump-
section .data=data.txt --dump-section .bss=sbss.data  bubbleSort.elf
```

Tabla 27. Contenido del fichero elf.

Los ficheros obtenidos deben ser tratados para obtener únicamente dos ficheros, uno para la memoria de instrucciones y otro para la memoria de datos y que contenga valores de 32 bits en cada una de sus lineas,

para ello se ha optado por desarrollar una pequeña aplicación en MATLAB. Su desarrollo se explica en la siguiente sección (5.2.).

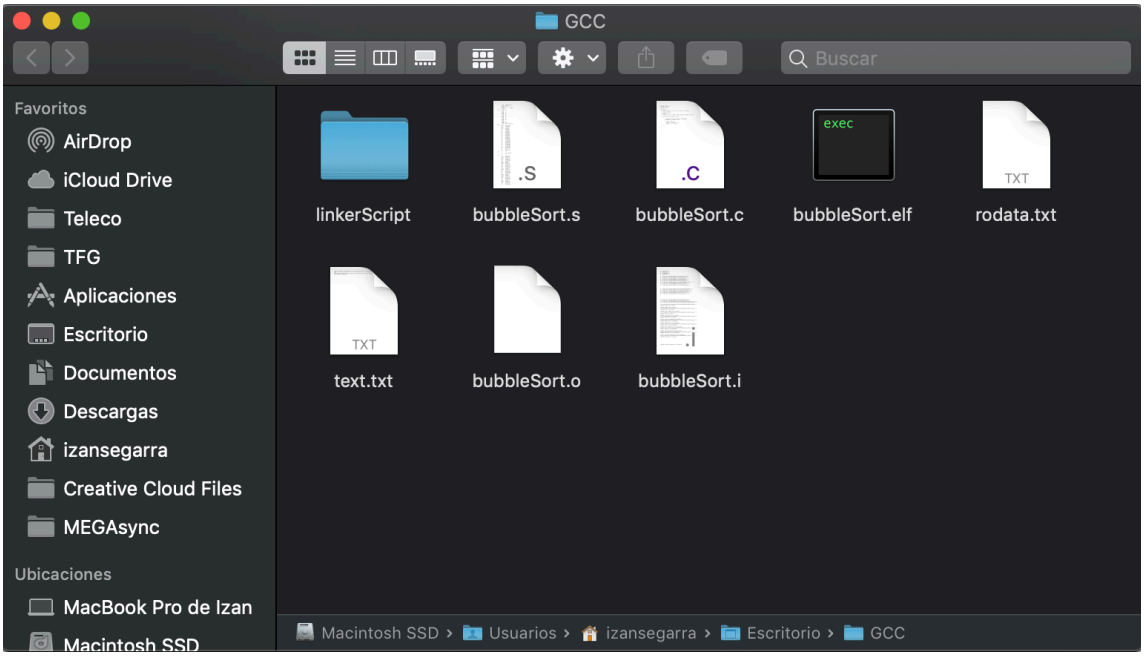

Figura 47. Archivos generados en el directorio de trabajo.

5.2. Diseño de la herramienta de compilación con MATLAB

El desarrollo de la aplicación se ha realizado mediante la utilización de la herramienta MATLAB App Designer que proporciona una plataforma rápida y sencilla con la que hacer aplicaciones multiplataforma con interfaz gráfica.

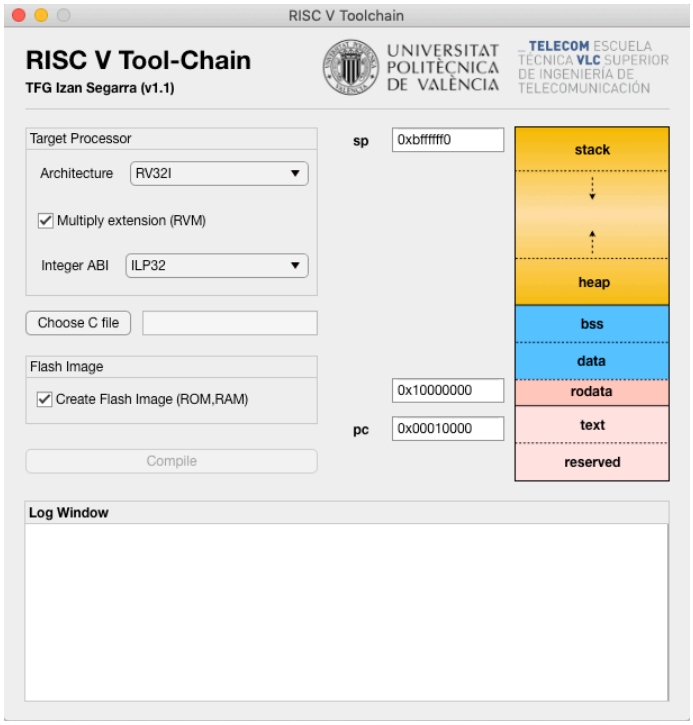

Figura 48. Interfaz de la aplicación de compilación.

El diseño de la aplicación ha sido sencillo, asociar la ejecución de los comandos vistos en la sección anterior con la pulsación del botón *Compile.* Posteriormente se leen los archivos generados y se procesan para que haya 32 bits por linea. Finalmente la sección *text* se copia en un fichero llamado program.txt y las secciones *rodata, data* y *bss* se combinan en un único fichero llamado data.txt. Ambos ficheros están preparados para cargarlos en el procesador.

La aplicación permite seleccionar arquitecturas de 32 bits y 64 bits, indicar las direcciones de inicio de las secciones *text* y *data*. Además, permite elegir si se quieren crear o no los ficheros para cargarlos en el procesador.

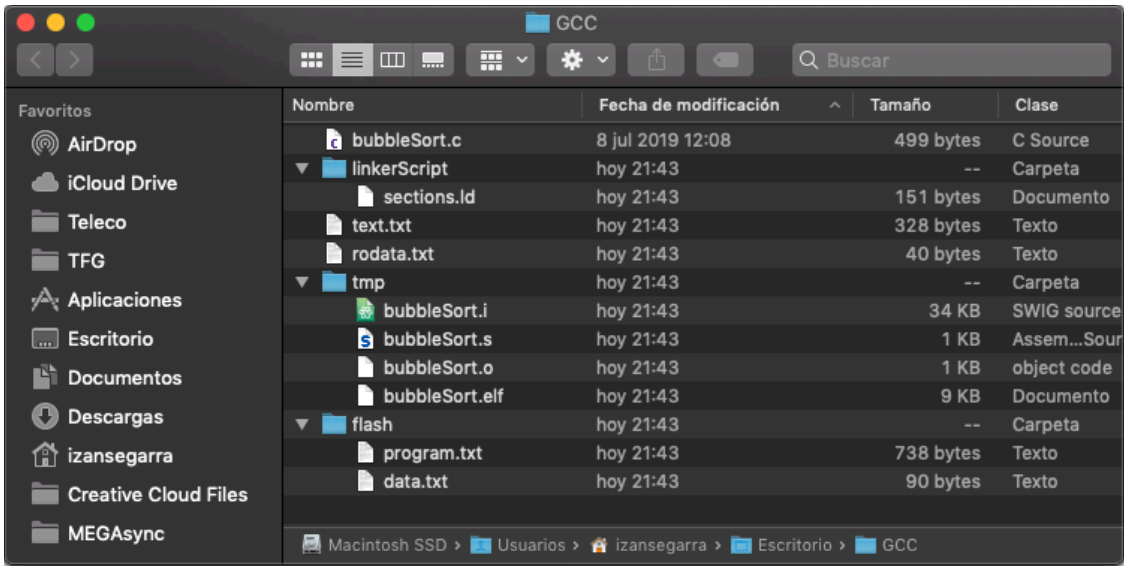

Figura 49. Interfaz de la aplicación de compilación.

Para utilizar la aplicación simplemente es necesario seleccionar el archivo con el código en C y apretar el botón de compilación, el resultado de la ejecución es la generación de todos los archivos que se muestran en la *Figura 49.* Los ficheros a cargar en el procesador se encuentran dentro de la carpeta *flash.*

```
FB010113
04812623
05010413
0FFF0797
FF478793
0007AE03
0047A303
    .
    .
    .
04C12403
05010113
00008067
```

Tabla 28. Contenido del fichero program.txt.

Como se muestra en la *Tabla 28* el contenido del archivo program.txt ya es compatible con el procesador desarrollado. La compilación se ha realizado con el programa *Bubble Sort* desarrollado en C.

5.3. Diseño de la página web para visualizar los resultados

Con la posibilidad de compilar código en C de forma sencilla para el procesador, es interesante poder ver el resultado de la ejecución de los programas de forma más sencilla e intuitiva. Para ello se ha desarrollado la página web que se muestra en la *Figura 50.*

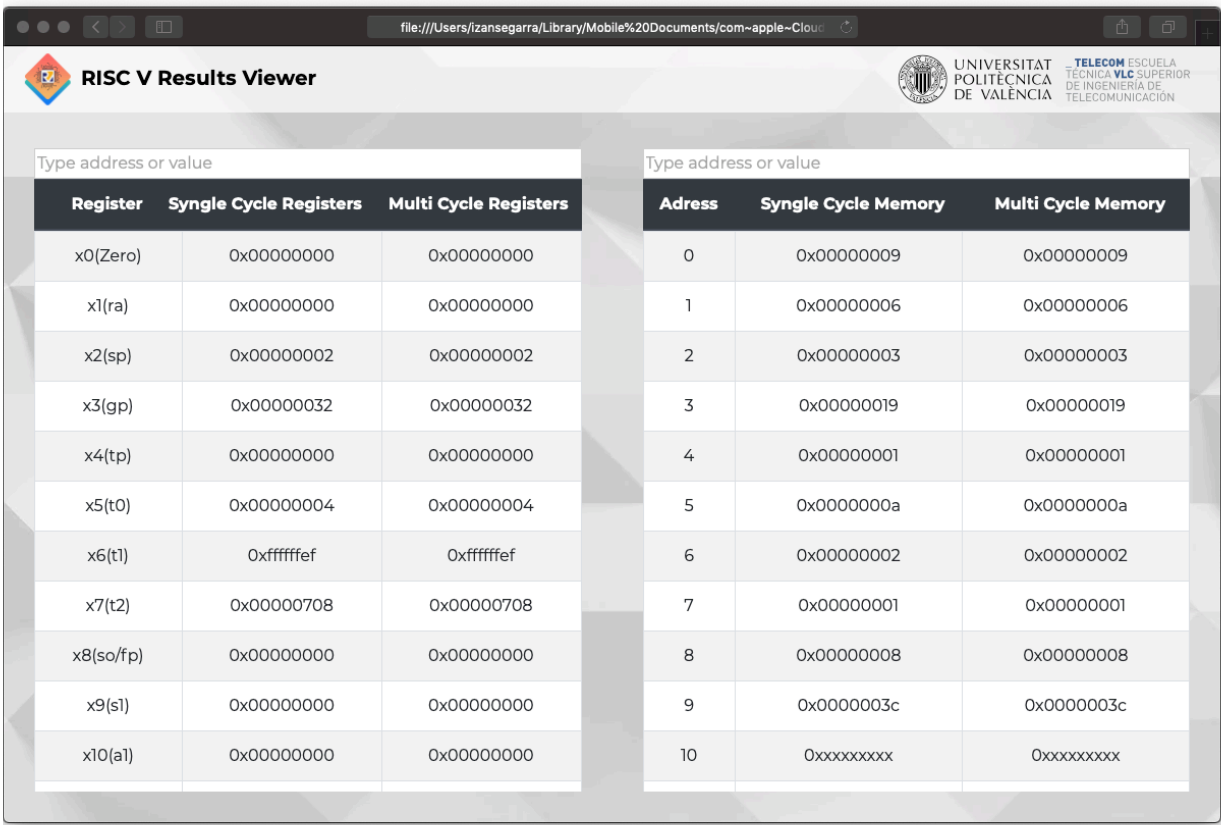

Figura 50. Página web para visualizar el resultado de la ejecución.

La web muestra el contenido del banco de registros y de la memoria de datos, además permite realizar la búsqueda de valores tanto en los registros como en la memoria. De esta forma se pueden visualizar los resultados y localizar las posiciones en las que se produce fallos de forma cómoda.

La web se ha desarrollado usando HTML, CSS, JavaScript y el *framework* Bootstrap. Para acceder a los datos almacenados en la memoria y el registro es necesario modificar el *testbench y* se escriben en un fichero JavaScript los valores almacenados de la forma correspondiente al tipo de fichero tal y como muestra la *Tabla 28*.

```
var regDataSC = [ "00000000","00000000","00000002","00000032","00000000","00000004"];
```

Tabla 28. Declaración de una variable en JavaScript.

Posteriormente se fusiona el archivo generado con el archivo base que conforma la página web mediante una llamada al sistema como se ha visto en la sección 4.6. En este caso se hace uso del comando *copy* con las opciones /b e /y.

```
$system("copy /b/y RF_SC.js+RF_MC.js+scroll_search.js  scroll_search.js");
```

Tabla 29. Comando para fusionar los archivos.

Para automatizar el lanzamiento de la web una vez finalizado el *testbench* se ha realizado una llamada al sistema con el comando *start* la ruta donde se encuentra la página web.

Capítulo 6. Rendimiento del procesador y validación funcional

En esta sección se van a obtener los parámetros típicos que determinan el rendimiento de un procesador y será verificado mediante la ejecución de programas realizados en C, a su vez comparará el rendimiento de las versiones *single cycle* y *multi cycle.*

Para comparar ambas implementaciones se utiliza fundamentalmente el tiempo de ejecución de los programas desarrollados. El tiempo de CPU de los programas puede expresarse tal y como se muestra en la ecuación 6.1, siendo T_C el periodo del ciclo de reloj. [19]

$$\mathbf{T_{cpu} = Número\ de\ ciclos \cdot T_C} \tag{6.1}$$

Además, conociendo el número de instrucciones a ejecutar, es posible obtener el número medio de ciclos por instrucción.

$$\mathbf{CPI} = \frac{\mathbf{Número\ de\ ciclos\ de\ reloj}}{\mathbf{NI}} \tag{6.2}$$

Cabe destacar que NI (Número de Instrucciones) depende tanto del compilador utilizado como de la arquitectura del procesador. [19]

Otro de los parámetros interesantes para medir el rendimiento del las implementaciones son los MIPS (Millones de Instrucciones Por Segundo) que son capaces de ejecutar. Habitualmente es difícil utilizar es parámetro para comparar procesadores ya que depende del juego de instrucciones y varía en función del programa ejecutado. En este caso, como ambas implementaciones tienen el mismo juego de instrucciones, puede usarse como medida de comparación válida. [19]

$$\mathbf{MIPS} = \frac{\mathbf{NI}}{\mathbf{T_{Ejecución} \cdot 10^6}} = \frac{1}{\mathbf{CPI \cdot T_C \cdot 10^6}} = \frac{\mathbf{F_C}}{\mathbf{CPI \cdot 10^6}} \tag{6.3}$$

Una vez vistos los parámetros necesarios para medir el rendimiento de un procesador, se procede a realizar la verificación funcional y con ello la obtención de los parámetros con el fin de compararlos. La implementación *single cycle* ejecuta los programas con un periodo de 23 ns (~43,48 MHz) y la implementación *multi cycle* ejecuta los programas con un periodo de 14 ns (~71,43 MHz).

Las pruebas se han realizado con programas que contienen instrucciones de saltos condicionales e incondicionales en gran cantidad como *Multiplication and Branch o Fibonacci*, programas con gran cantidad de accesos a memoria (lectura/escritura) como *Bubble Sort* y programas con una gran cantidad de operaciones aritméticas como *Multiplication* e *Integer Arithmetic*.

Programa	NI	Tiempo CPU	$T_{Medio\ Ejecución}$	T_C	F_C	Ciclos de reloj	CPI	MIPS
Bubble Sort	106	22264 ns	210,038 ns	23 ns	43,48 MHz	968	9,1	4,76
Mult & Branch	22	1472 ns	66,909 ns	23 ns	43,48 MHz	64	2,9	14,95
Multiplication	1024	22494 ns	21,967 ns	23 ns	43,48 MHz	978	1,0	45,52
Fibonacci	35	374144 ns	10689,829 ns	23 ns	43,48 MHz	16267	464,8	0,094
Int Arithmetic	1024	22484 ns	21,957 ns	23 ns	43,48 MHz	978	1,0	45,54

Tabla 30. Parámetros de rendimiento del procesador mono ciclo.

Programa	NI	Tiempo CPU	$T_{Medio\ Ejecución}$	T_C	F_C	Ciclos de reloj	CPI	MIPS
Bubble Sort	106	21770 ns	205,377 ns	14 ns	71,43 MHz	1555	14,7	4,87
Mult & Branch	22	1792 ns	81,455 ns	14 ns	71,43 MHz	128	5,8	12,28
Multiplication	1024	14364 ns	14,027 ns	14 ns	71,43 MHz	1026	1,0	71,29
Fibonacci	35	406112 ns	11603,2 ns	14 ns	71,43 MHz	29008	828,8	0,086
Int Arithmetic	1024	14364 ns	14,027 ns	14 ns	71,43 MHz	1026	1,0	71,29

Tabla 31. Parámetros de rendimiento del procesador segmentado.

Una vez obtenido los parámetros, es posible calcular la aceleración conseguida por el procesador segmentado con respecto al *single cycle.*

Programa	Aceleración
Bubble Sort	1,02
Mult & Branch	0,82
Multiplication	1,57
Fibonacci	0,92
Int Arithmetic	1,57

Tabla 32. Aceleración.

Viendo los datos obtenidos, es evidente que la implementación segmentada se ve seriamente penalizada por aquellos programas que utilizan saltos de forma repetitiva debido al retardo de tres ciclos que provoca. Por este motivo, se concluye que el adelantamiento de la lógica combinacional que controla los saltos a la etapa ID es imprescindible para obtener un mayor rendimiento del procesador segmentado. A su vez, la solución más acertada sería la implementación de la predicción de saltos en una futura ampliación del proyecto.

Cabe destacar que aquellos programas que no utilizan saltos de forma recurrente consiguen una aceleración de 1,57 veces con respecto a la implementación no segmentada, justificando así la utilización de la implementación segmentada. Cuanto mayor sea el número de instrucciones a ejecutar, mayor será el rendimiento del procesador segmentado.

Capítulo 7. Bibliografía

7.1.Referencias bibliográficas

[1] Waterman, A. S. (2016). Design of the RISC-V Instruction Set Architecture. UC Berkeley. ProQuest ID: Waterman_berkeley_0028E_15908. Merritt ID: ark:/13030/m50c9hkd. Recuperado 12 de Julio de 2019 desde https://escholarship.org/uc/item/7zj0b3m7

[2] Porter III, Harry H. (2018). RISC-V: An Overview of the Instruction Set Architecture. Portland State University. Recuperado 12 de Julio de 2019 desde http://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf

[3] RISC-V. (2019, 9 julio). Recuperado 12 julio, 2019, de https://es.wikipedia.org/wiki/RISC-V

[4] Waterman, A., & Asanović, K. (2017, 7 mayo). The RISC-V Instruction Set Manual Volume I: User-Level ISA [PDF]. Recuperado 12 julio, 2019, de https://riscv.org/specifications/

[5] CC-3. (s.f.). RISC-V - Estructura de Máquinas. Recuperado 13 julio, 2019, de https://cc-3.github.io/notes/02_Intro-RISCV/

[6] Martínez Pérez, J. D. (2018, 15 noviembre). Arquitecturas para procesado de datos. Parte I – Introducción a RISC-V. Recuperado 15 julio, 2019, de https://poliformat.upv.es/

[7] Latra, E. (2019, 3 julio). Compiler [PDF]. Recuperado 15 julio, 2019, de http://www.ele.uva.es/~jesus/hardware_empotrado/Compiler.pdf

[8] Mips memory layout :: Operating systems 2018. (s.f.). Recuperado 17 julio, 2019, de http://www.it.uu.se/education/course/homepage/os/vt18/module-0/mips-and-mars/mips-memory-layout/

[9] Patterson, D. A., & Hennessy, J. L. (2017). Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Ámsterdam, Países Bajos: Elsevier Science.

[10] Intel. (2018, 21 diciembre). Intel Quartus Prime Standard Edition User Guide: Design Recommendations. Recuperado 20 julio, 2019, de https://www.intel.com/content/www/us/en/programmable/documentation/ntt1529445293791.html

[11] Martínez Pérez, J. D. (2018, 15 noviembre). Arquitecturas para procesado de datos. Parte IV – Implementación pipelined. Recuperado 15 julio, 2019, de https://poliformat.upv.es/

[12] Mico Tormos, P. (s.f.). Arquitecturas de Computadores - Tema 4 - Procesadores Segmenrados [PDF]. Recuperado 26 julio, 2019, de http://personales.upv.es/pabmitor/acso/FILES/ArqComp/CST/ArqComp%20t4.pdf

[13] Silva Bijit, L. (2008, 13 noviembre). Cap.14 Segmentación [PDF]. Recuperado 26 julio, 2019, de http://www2.elo.utfsm.cl/~lsb/elo311/clases/c14.pdf

[14] Marín, E. (2003, 4 marzo). Apuntes Segmentados [PDF]. Recuperado 26 julio, 2019, de http://studies.ac.upc.edu/EUPVG/ARCO_I/tema2.pdf

[15] Pérula-Martinez, R., Lorente, L. M., Brunete, A., Alegre, D. G., Gutiérrez, C. A., Montoya, J. C. C. (22/05/2014). Tema 6. Introducción a la segmentación avanzada: Riesgos. Recuperado el 28 julio, 2019, de http://ocw.uc3m.es/ingenieria-informatica/organizacion-de-computadores/material-teorico-1/OC_T06.pdf.

[16] Schiele, R. (2004, 13 agosto). Building and Using a Cross Development Tool Chain [PDF]. Recuperado 31 julio, 2019, de ftp://gcc.gnu.org/pub/gcc/summit/2003/Building%20and%20Using%20a%20Cross%20Development%20Tool%20Chain.pdf

[17] Colaboradores de Wikipedia. (2019, 12 julio). Executable and Linkable Format - Wikipedia, la enciclopedia libre. Recuperado 31 julio, 2019, de https://es.wikipedia.org/wiki/Executable_and_Linkable_Format

[18] Introduction to RISC-V Embedded Development 1: Introduction to the RISC-V GCC Toolchain - Programmer Sought. (s.f.). Recuperado 31 julio, 2019, de http://www.programmersought.com/article/6138108806/

[19] Ruz Ortiz, J. J. (2011, 2 octubre). Tema 4: Rendimiento del procesador [PDF]. Recuperado 28 julio, 2019, de http://studies.ac.upc.edu/EUPVG/ARCO_I/tema2.pdf