

Grado en Ingeniería Electrónica Industrial y Automática
2022-2023

Trabajo Fin de Grado

“Diseño de un núcleo de
microprocesador RISC-V para FPGA”

Alberto Caravantes Arranz

Tutor

Mario García Valderas

Leganés, 2023



Esta obra se encuentra sujeta a la licencia Creative Commons **Reconocimiento - No Comercial - Sin Obra Derivada**

RESUMEN

El presente Trabajo de Fin de Grado expone el diseño de un núcleo de microprocesador basado en la arquitectura RISC-V (*Reduced Instruction Set Computing - V*, computador con conjunto de instrucciones reducido). Junto al núcleo se han diseñado una serie de periféricos, tales como un temporizador y un GPIO (*General Purpose Input/Output*, entrada/salida de propósito general) para comprobar y expandir las funcionalidades de dicho núcleo.

Este diseño se ha implementado en una placa FPGA (*Field Programmable Gate Array* matriz de puertas lógicas programable en campo) mediante el lenguaje VHDL (*Very high speed integrated circuits Hardware Description Language*, lenguaje de descripción de hardware para circuitos integrados de muy alta velocidad).

El objetivo del proyecto radica en mostrar la sencillez de diseñar un núcleo de microprocesador, al basarse éste en una arquitectura reciente tan sencilla como es RISC-V, así como mostrar su gran potencial para sustituir a arquitecturas actuales tales como x86 y ARM.

Previo al propio diseño, se analiza la especificación RISC-V, haciendo hincapié en la versión a implementar, RV32I. Se procede a explicar el proceso de diseño del núcleo, así como el funcionamiento interno del mismo mediante diagramas de bloques. Se muestran los detalles de cada uno de sus componentes, y se comprueba su funcionamiento mediante sencillos programas escritos en lenguaje ensamblador, y convertidos a lenguaje máquina a través de un ensamblador programado en lenguaje C++.

Tras el diseño del núcleo, se muestra su funcionalidad en un caso práctico que consiste en comparar, mediante el *Timer*, el tiempo de computación que requiere ordenar diez números con tres algoritmos distintos de complejidad $O(n^2)$. Finalmente, se configura el GPIO como salida, de modo que se visualicen los números ordenados a través del encendido de los leds en la placa FPGA.

Se realiza además un análisis del uso de recursos físicos de la placa, así como de la temporización, demostrando el correcto diseño del mismo.

Palabras clave: VHDL; Ensamblador; Instrucción; Codificación; Registros.

DEDICATORIA

A mi familia,

Gracias por ayudarme toda mi vida a alcanzar cada una de mis metas. Apoyándome, sufriendo conmigo cada proyecto, cada examen, cada problema. Apoyo incondicional, eso es lo que me regaláis día a día.

A Tamara,

Gracias por ser la persona que me ha lanzado a estudiar fuera y ayudarme a descubrir mi camino. Tu compañía estos años me ha dado la vida.

A Mario García Valderas, mi tutor,

Gracias por enseñarme, ayudarme y orientarme en este trabajo siempre que lo he necesitado, incluso a pesar de la distancia.

ÍNDICE GENERAL

1. INTRODUCCIÓN	1
1.1. Motivación	1
1.2. Objetivo	1
1.3. Factores a destacar	2
2. ARQUITECTURA RISC-V	3
2.1. Tipos de instrucciones RISC-V	4
3. DISEÑO DEL NÚCLEO DE MICROPROCESADOR Y PERIFÉRICOS	9
3.1. Proceso de diseño	10
3.2. Componentes del núcleo.	13
3.2.1. Contador de programa	13
3.2.2. Memoria de instrucciones	14
3.2.3. Control de excepciones	15
3.2.4. Registro de instrucciones	15
3.2.5. Control de la Unidad Aritmético-Lógica.	16
3.2.6. Selección de valor inmediato	17
3.2.7. Comparación de registros	20
3.2.8. Pila de registros de propósito general	20
3.2.9. Registros de Control y Estado.	22
3.2.10. Unidad de Control	22
3.2.11. Unidad Aritmético-Lógica	28
3.2.12. Control de instrucciones de guardado.	29
3.2.13. Control de instrucciones de salto	29
3.2.14. Control de instrucciones de carga	30
3.2.15. Memoria de datos.	31
3.3. Periféricos.	31
3.3.1. GPIO	33
3.3.2. Timer	34
3.4. Diseño completo	36

4. DISEÑO DEL ENSAMBLADOR	38
4.1. Programa ensamblador y lenguaje ensamblador.	38
4.2. Funcionamiento del ensamblador.	39
4.3. Compilar y enlazar. Diferencias.	39
4.4. Makefile y compilación del ensamblador	40
4.5. Utilizando el ensamblador.	41
5. PRUEBAS BÁSICAS DEL NÚCLEO	42
5.1. Ejemplo_1.s: Operaciones lógicas y shifts.	42
5.2. Ejemplo_2.s: sb, lw y comparación slt	44
5.3. Ejemplo_3.s: Bucle con iterador. Carga a GPIOs.	45
5.4. Ejemplo_4.s: CSRs	46
5.5. Ejemplo_5.s: Stores y loads	47
5.6. Ejemplo_6.s: Probando el Timer	48
6. EJEMPLOS PRÁCTICOS	49
6.1. Ejemplo práctico 1: Ordenación de números enteros	49
6.1.1. Comparación de tres algoritmos de ordenación en simulación.	49
6.1.2. Implementación del núcleo en la FPGA y uso de GPIOs.	55
6.2. Ejemplo práctico 2: Correcto funcionamiento de GPIOs	60
7. RESULTADOS	63
7.1. Análisis post-implementación del uso de recursos	64
7.2. Análisis temporal.	66
8. CONCLUSIONES	68
9. MARCO REGULADOR	71
10. PRESUPUESTO	72
11. IMPACTO SOCIO-ECONÓMICO	74
BIBLIOGRAFÍA	76
ANEXO A. RV32I BASE INSTRUCTION SET ENCODING [24]	
ANEXO B. CÓDIGO DEL DISEÑO VHDL	
ANEXO C. CÓDIGO DEL ENSAMBLADOR	

ÍNDICE DE FIGURAS

2.1	Codificación de un solo bit del valor inmediato con multiplexores 32 a 1	4
2.2	Codificación de instrucciones de tipo R.	5
2.3	Codificación de instrucciones de tipo I.	5
2.4	Codificación de instrucciones de desplazamientos de <i>bits</i>	5
2.5	Codificación de instrucciones instrucciones de carga.	6
2.6	Codificación de la instrucción “jalr”.	6
2.7	Codificación de instrucciones de memoria atómica.	6
2.8	Codificación de instrucciones de entorno	7
2.9	Codificación de instrucciones de tipo B.	7
2.10	Codificación de instrucciones de tipo J (jal).	8
2.11	Codificación de instrucciones de tipo S.	8
2.12	Codificación de instrucciones de tipo U.	8
3.1	Diagrama de bloques del núcleo y periféricos.	9
3.2	Diagrama de bloques del primer diseño funcional.	10
3.3	Esquemático del primer diseño funcional.	11
3.4	Ejemplo 1. add.	11
3.5	Ejemplo 2. “jalr”.	12
3.6	Ejemplo 3. “sb”	12
3.7	Diagrama de bloques del Contador de Programa.	13
3.8	Diagrama de bloques de la Memoria de Instrucciones.	14
3.9	Comparativa little/big endian guardando la instrucción 0x01234567 en los primeros 4 <i>bytes</i> de memoria.	14
3.10	Diagrama de bloques del Control de Excepciones.	15
3.11	Diagrama de bloques del Registro de Instrucciones.	15
3.12	Diagrama de bloques del Control de la ALU.	16
3.13	Diagrama de bloques de la Selección de valor inmediato.	17
3.14	Codificación del valor inmediato para instrucciones de tipo I. Superior: [instrucción]. Inferior: [valor inmediato].	18

3.15 Codificación del valor inmediato para instrucciones de tipo B. Superior: [instrucción]. Inferior: [valor inmediato].	18
3.16 Codificación del valor inmediato para instrucciones de tipo S. Superior: [instrucción]. Inferior: [valor inmediato].	18
3.17 Codificación del valor inmediato para instrucciones de tipo U. Superior: [instrucción]. Inferior: [valor inmediato].	19
3.18 Codificación del valor inmediato para instrucciones de tipo J. Superior: [instrucción]. Inferior: [valor inmediato].	19
3.19 Codificación del valor inmediato para instrucciones de memoria atómica. Superior: [instrucción]. Inferior: [valor inmediato].	19
3.20 Diagrama de bloques de la comparación de registros.	20
3.21 Diagrama de bloques de la pila de registros.	21
3.22 Diagrama de bloques de los Registros de Control y Estado.	22
3.23 Diagrama de bloques de la Unidad de Control.	22
3.24 Diagrama de la máquina de estados finitos.	23
3.25 Microcódigo de la FSM.	25
3.26 Diagrama de bloques de la Unidad de Control.	26
3.27 Distribución de la memoria para el control de excepciones.	27
3.28 Diagrama de bloques de la Unidad Aritmetico-Lógica.	28
3.29 Diagrama de bloques del control de instrucciones de guardado.	29
3.30 Diagrama de bloques del control de instrucciones de salto.	30
3.31 Diagrama de bloques del control de instrucciones de carga.	31
3.32 Diagrama de bloques de la memoria de datos.	31
3.33 Direccionamiento de periféricos	32
3.34 Diagrama de bloques de los GPIOs.	33
3.35 Funcionamiento del GPIO para un solo pin.	33
3.36 Simbolo y tabla de verdad de un <i>buffer</i> triestado.	34
3.37 Diagrama de bloques del <i>Timer</i>	35
3.38 Funcionamiento interno del <i>Timer</i>	35
3.39 Esquemático del diseño final (1). ProgramCounter, InstructionMemory, ExceptionControl y InstructionRegister.	36
3.40 Esquemático del diseño final (2). RegisterFile, CSRs, Comparison, ALU-Control, ALU, StoreControl y ImmSelect.	36

3.41	Esquemático del diseño final (3). JumpControl, DataMemory, GPIO, Timer, ControlUnit y LoadControl.	37
4.1	Propósito del ensamblador.	38
4.2	Proceso de compilación en C y C++.	40
4.3	Compilando el ensamblador con make.	40
4.4	Haciendo uso del ensamblador.	41
5.1	Simulación Ejemplo 1.	43
5.2	Simulación Ejemplo 2.	44
5.3	Simulación Ejemplo 3.	45
5.4	Simulación Ejemplo 4.	46
5.5	Simulación Ejemplo 5.	47
5.6	Simulación Ejemplo 6.	48
6.1	Resultados de la simulación para BubbleSort.mem.	54
6.2	Resultados de la simulación para InsertionSort.mem.	54
6.3	Resultados de la simulación para SelectionSort.mem.	55
6.4	Diligent Basys 3 Artix-7 FPGA Trainer Board	56
6.5	Clocking Wizard 6.0 en Vivado.	57
6.6	Representación gráfica de la Basys 3 de Diligent.	61
6.7	Resultados de la simulación para FPGA_0.mem	62
6.8	Probando el programa FPGA_0.mem en la Basys 3 (1).	62
6.9	Probando el programa FPGA_0.mem en la Basys 3 (2).	62
7.1	Números correctamente ordenados en los leds de la FPGA.	63
7.2	Tabla del análisis post-implementación del uso de recursos en Vivado.	64
7.3	Gráfica del análisis post-implementación del uso de recursos en Vivado.	64
7.4	Diseño de una LUT para la expresión booleana (7.1).	65
7.5	Ánálisis temporal en Vivado.	66
7.6	WNS y el retardo en la propagación de señales entre componentes síncronos.	66
7.7	Representación gráfica del <i>Negative Slack</i>	66

8.1 Ejecución en un solo ciclo. Superior: [Sin segmentar]. Inferior: [Segmentada]. [8]	69
--	----

ÍNDICE DE TABLAS

2.1	Tamaño del campo de valor inmediato	3
3.1	Generación de señal “output” en AluControl	16
3.2	Generación de señal “output” en ALUControl para instrucción “lui” . . .	17
3.3	Generación de señal “immsel” para cada tipo de instrucción	17
3.4	Generación de la señal “comparison”	20
3.5	Nombres simbólicos y funcionalidad de los registros de propósito general.	21
3.6	Operación a ejecutar por la ALU para cada señal de “control”	28
6.1	Comparación del tiempo requerido para los tres algoritmos.	55

LISTA DE ABREVIATURAS

ALU	<i>Arithmetic Logic Unit</i> , unidad aritmético-lógica
BUFG	<i>global clock buffer</i> , buffer global de reloj
CPU	<i>Central Processing Unit</i> , unidad central de procesamiento
CISC	<i>Complex Instruction Set Computing</i> , ordenador con conjunto de instrucciones complejas
CSR	<i>Control and State Register</i> , registro de control y estado
FF	<i>Flip-Flop</i> , biestable
FPGA	<i>Field Programmable Gate Array</i> , matriz de puertas lógicas programable en campo
FSM	<i>Finite State Machine</i> , máquina de estados finitos
GPIO	<i>General Purpose Input/Output</i> , entrada-salida de propósito general
HPSC	<i>High-Performance Spaceflight Computing</i> , computación de alto rendimiento para vuelos espaciales
I^2C	<i>Inter-Integrated Circuit</i> , circuito inter-integrado
IEEE	<i>Institute of Electrical and Electronics Engineers</i> , instituto de ingenieros eléctricos y electrónicos
INV	Instrucción No Válida
IO	<i>Input/Output</i> , entrada/salida
IP	<i>Intellectual Property</i> , propiedad intelectual
IRQ	<i>Interrupt Request</i> , petición de interrupción
ISA	<i>Instruction Set Architecture</i> , arquitectura del conjunto de instrucciones
LSB	<i>Least Significant Bit</i> , bit menos significativo
LUT	<i>Look-Up Table</i> , tabla de consulta
LUTRAM	<i>LUT Random Access Memory</i> , LUT de memoria de acceso aleatorio
MMCM	<i>Mixed-Mode Clock Manager</i> , administrador de relojes de modo mixto
MSB	<i>Most Significant Bit</i> , bit más significativo
opcode	<i>Operation Code</i> , código de operación
RISC	<i>Reduced Instruction Set Computing</i> , ordenador con conjunto de instrucciones reducidas
SPI	<i>Serial Peripheral Interface</i> , interfaz periférica serial
SSD	<i>Solid State Drive</i> , disco de estado sólido
UART	<i>Universal Asynchronous Receiver/Transmitter</i> , transmisor-receptor asíncrono universal
USB	<i>Universal Serial Bus</i> , bus universal en serie

VHDL	<i>Very high speed integrated circuit Hardware Description Language</i> , lenguaje de descripción de <i>hardware</i> para circuitos integrados de muy alta velocidad
WNS	<i>Worst Negative Slack</i> , margen de retraso negativo máximo

1. INTRODUCCIÓN

1.1. Motivación

El primer microprocesador de la historia data de 1971, cuando Intel desarrolló el Intel 4004. Éste era capaz de realizar hasta 60 mil operaciones por segundo, con una frecuencia de reloj de 740 khz y con tan solo 4 *bits* por instrucción [1]. Hoy día se pueden adquirir microprocesadores, como el Intel Core i9-13900K, de hasta 5,8 Ghz [2].

La base de todo procesador es su ISA (*Instruction Set Architecture*, arquitectura del conjunto de instrucciones). Ésta define qué instrucciones puede ejecutar, y de qué forma están codificadas. Existen principalmente dos tipos de ISA, la CISC (*Complex Instruction Set Computing*, ordenador con conjunto de instrucciones complejas) y la RISC (*Reduced Instruction Set Computing*, ordenador con conjunto de instrucciones reducidas).

El tipo de ISA CISC incluye instrucciones complejas y lentas, ya que suelen requerir varios ciclos de reloj. Al ser más complejas, pueden realizar varias operaciones en una misma instrucción. Esto permite ejecutar menos instrucciones y por lo tanto, crear programas de espacio más reducido. Sirvan como ejemplo el Intel 4004 mencionado anteriormente, el Motorola 68000 y el Intel 8006.

En contraposición, el tipo de ISA RISC incluye instrucciones sencillas y rápidas pero crean programas que ocupan más espacio [3]. Pese a que este tipo de arquitectura se suele utilizar en dispositivos móviles, Apple sacó al mercado sus propios ordenadores con un microprocesador (M1) basado en una arquitectura RISC (ARM), en el año 2020.

RISC surgió en la década de los 70 del siglo XX, con las investigaciones realizadas por IBM en colaboración con las Universidades de Standford y Berkeley. En 1980 se llevó a cabo el proyecto de la minicomputadora IBM 801, dirigido por John Cocke, marcando el inicio de las arquitecturas RISC. Cocke demostró que el procesador de un ordenador requería tan solo el 20 % de las instrucciones para realizar el 80 % del trabajo [4]. Esto supuso un mejor rendimiento, un coste reducido (menor número de transistores por procesador) y por lo tanto, un menor consumo energético que lo convierte en el procesador idóneo para teléfonos móviles y tabletas.

1.2. Objetivo

En este trabajo se va a implementar sobre una FPGA (*Field Programmable Gate Array*, matriz de puertas lógicas programable en campo) el diseño del núcleo (unidad de procesamiento independiente) de un microprocesador basado en la arquitectura RISC-V, en

concreto, en la RV32I. El objetivo de este trabajo es demostrar la sencillez del desarrollo de un microprocesador con esta arquitectura de *hardware* libre (sin derechos), respaldado por una comunidad global que colabora en la expansión de esta tecnología emergente.

1.3. Factores a destacar

El **marco regulador** del trabajo viene definido en la sección 9 (Marco Regulador), con una descripción más detallada de la especificación RISC-V en la sección 2 (Arquitectura RISC-V).

El **entorno socio-económico** de RISC-V se define en las secciones 10 (Presupuesto) y 11 (Impacto socio-económico), así como las posibles mejoras, o plan de explotación, en la sección 8 (Conclusiones).

2. ARQUITECTURA RISC-V

RISC-V es una arquitectura de conjunto de instrucciones, basado en el tipo de arquitectura RISC, desarrollado en el Parallel Computing Laboratory de la Universidad de California, Berkeley, en 2010. Su objetivo es asentar esta arquitectura simple, modular, y de código abierto, para que pueda llegar a sustituir a arquitecturas actuales como x86 (CISC), o ARM (RISC), gracias a su buen rendimiento y bajo coste [5].

RISC-V dispone de cuatro versiones base (RV32I, RV32E, RV64I y RV128I), las cuales operan sobre valores enteros, y se componen de instrucciones de 32, 64 o 128 *bits*. A excepción de la versión RV32E, disponen de 32 registros (nombrados x0 a x31), y almacenan las instrucciones por *bytes*, generalmente en formato *little endian* (el *byte* menos significativo se almacena en la dirección de memoria más baja, seguido por los *bytes* más significativos en orden creciente).

La versión base puede expandirse mediante extensiones. Por ejemplo: La extensión M incorpora multiplicación y división; F, operaciones de punto flotante de precisión simple; D, operaciones de punto flotante de precisión doble. Para este proyecto, la versión de RISC-V que se va a utilizar es la RV32I. Ésta versión opera sobre valores enteros, dispone de 32 registros y de 45 instrucciones distintas.

La arquitectura RISC-V optimiza la decodificación del procesador mediante una codificación de los principales campos sencilla y similar entre instrucciones. El *opcode* (código de operación) se encuentra siempre entre los *bits* 6 y 0. Y en caso de estar presentes, el registro de destino (rd) entre los *bits* 11 y 7, el primer registro fuente (rs1) entre los *bits* 19 y 15, y el segundo registro fuente (rs2) entre los *bits* 24 y 20.

Los demás campos (a excepción del valor inmediato), funct7 y funct3 residen en los *bits* 31-25 y 14-12 respectivamente.

El tamaño del campo de valor inmediato varía dependiendo del número de registros presentes en la instrucción [6].

TABLA 2.1. TAMAÑO DEL CAMPO DE VALOR INMEDIATO

Número de registros de destino	Número de registros fuente	Tamaño del campo para el valor inmediato	Tipo de instrucción
0	2	12 bits	S, B
1	0	20 bits	U
1	1	12 bits	I
1	2	0 bits	R

Como se va a ver en los siguientes apartados, puede resultar extraña la manera en la que están distribuidos los *bits* para algunos de los campos de valores inmediatos (*bits* y conjuntos de *bits* intercalados). Esto es una consecuencia directa de reducir el tamaño del procesador en el momento de diseñarlo y fabricarlo [6].

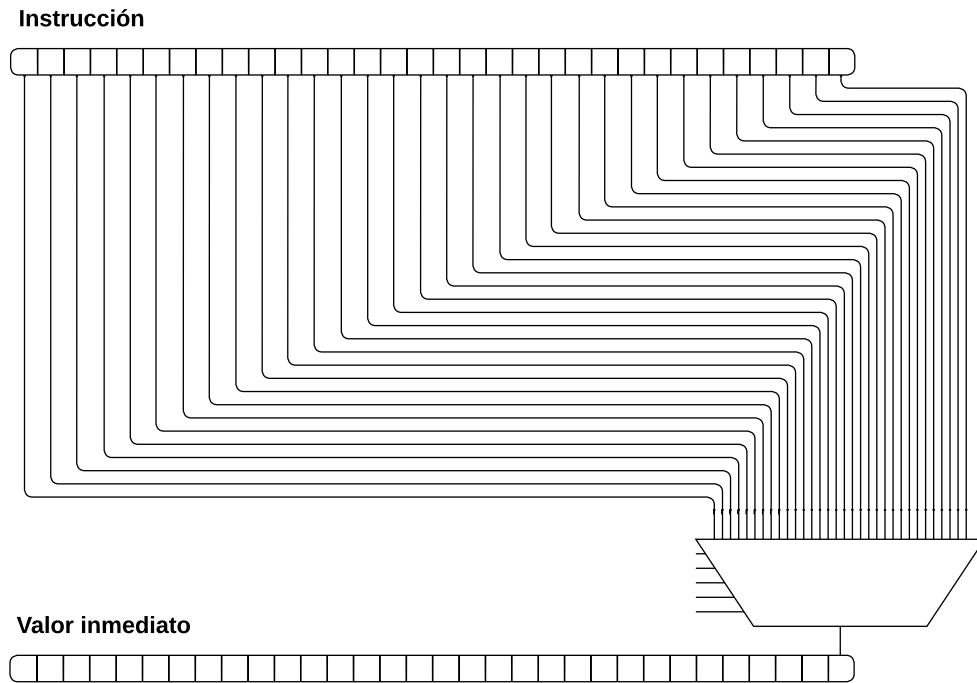


Fig. 2.1. Codificación de un solo bit del valor inmediato con multiplexores 32 a 1.

Para codificar el valor inmediato a nivel de hardware, se necesitan tantos multiplexores como *bits* tenga el valor inmediato a generar (32 en este caso). Si cada *bit* estuviera conectado a un multiplexor con los 32 posibles *bits* de entrada provenientes de la instrucción, se requerirían demasiados multiplexores de gran tamaño. Para evitarlo, RISC-V optimiza este proceso creando instrucciones con una codificación que requiere el uso de multiplexores con un número mínimo de entradas (en este caso solo las imprescindibles para generar ese *bit* del valor inmediato) [6].

2.1. Tipos de instrucciones RISC-V

Las instrucciones se dividen en seis tipos distintos, con una codificación diferente para cada uno: R, I, B, J, S y U.

- **Tipo R: Operaciones aritméticas y lógicas entre dos registros.**

Hacen operaciones y cargan el resultado en el registro rd. Entre sus instrucciones se encuentran:

ADD suma los valores de rs1 y rs2 (con signo). SUB resta el valor de rs2 al de rs1 (con signo). SLL y SRL desplazan lógicamente (desplaza y completa con ceros) tantos *bits* de rs1 como indiquen los cinco LSBs (*Least Significant Bits*, *bits* menos significativos) de rs2 (hacia la izquierda y hacia la derecha respectivamente). SRA desplaza aritméticamente (desplaza y completa con signo) hacia la derecha tantos *bits* de rs1 como indiquen los cinco LSBs de rs2. OR, XOR y AND efectúan operaciones lógicas entre rs1 y rs2. SLTU y SLT comparan el valor de rs1 y rs2 (con y sin signo respectivamente). Si se cumple que rs1 es menor que rs2 se carga un 1 en rd ó un 0 en caso contrario [7].

31	25	24	20	19	15	14	12	11	7	6	0
funct7		rs2		rs1		funct3		rd			opcode
0000000		src2		src1		ADD/SLT/SLTU		dest			OP
0000000		src2		src1		AND/OR/XOR		dest			OP
0000000		src2		src1		SLL/SRL		dest			OP
0100000		src2		src1		SUB/SRA		dest			OP
	7		5		5		3		5		7

Fig. 2.2. Codificación de instrucciones de tipo R.

■ **Tipo I: Operaciones aritméticas y lógicas entre un registro y un valor inmediato.**

ADDI, ORI, XORI, ANDI, SLTIU y SLTI actúan como las instrucciones equivalentes de tipo R, pero en lugar de operar entre rs1 y rs2, operan entre rs1 y un valor inmediato (consultar sección 3.6) [7].

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd			opcode
I-immediate[11:0]		src		ADDI/SLTI[U]		dest			OP-IMM
I-immediate[11:0]		src		ANDI/ORI/XORI		dest			OP-IMM
	12		5		3		5		7

Fig. 2.3. Codificación de instrucciones de tipo I.

SLLI, SR LI y SRAI también actúan como las instrucciones equivalentes de tipo R, pero en lugar de operar entre rs1 y rs2, operan entre rs1 y un valor inmediato (consultar sección 3.6) [7].

31	25	24	20	19	15	14	12	11	7	6	0
funct7		imm[4:0]		rs1		funct3		rd			opcode
0000000		shamt[4:0]		src		SLLI		dest			OP-IMM
0000000		shamt[4:0]		src		SR LI		dest			OP-IMM
0100000		shamt[4:0]		src		SRAI		dest			OP-IMM
	7		5		5		3		5		7

Fig. 2.4. Codificación de instrucciones de desplazamientos de *bits*.

LB Load Byte y *LH Load Halfword* cargan los 8 y 16 LSBs del valor guardado, en la dirección de memoria de la suma del valor en rs1 y un valor inmediato. Los *bits* restantes se llenan con el MSB (*Most Significant Bit*, *bit* más significativo), por lo que extienden el signo. *LW Load Word* igual que LB y LH, pero para los 32 *bits* (no hace falta extender el signo). *LBU Load Byte Unsigned* y *LHU Load Halfword Unsigned* actúan igual que LB y LH, solo que extienden los bits restantes con '0' (sin signo) [7].

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
offset[11:0]		base		LB[U]/LH[U]/LW		dest		LOAD	
12		5		3		5		7	

Fig. 2.5. Codificación de instrucciones instrucciones de carga.

JALR Jump and Link Register carga en rd la dirección de memoria de la siguiente instrucción a ejecutar, PC (*Program Counter*, contador de programa) + 4. A continuación, salta (modifica el valor del PC) a la dirección de memoria de la suma entre rs1 y un valor inmediato [7].

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
offset[11:0]		base		0		dest		JALR	
12		5		3		5		7	

Fig. 2.6. Codificación de la instrucción “jalr”.

Las instrucciones de memoria atómica actúan a nivel de *bits* sobre los CSR (*Control and State Register*, registros de control y estado). Todas cargan el valor actual del csr en rd. CSRRW *Atomic Read and Write* guarda el valor de rs1 en el csr. CSRRS *Atomic Read and Set* realiza una operación lógica OR entre los valores en csr y rs1 (para poner a '1' los *bits* que se consideren). CSRRC *Atomic Read and Clear* realiza una operación lógica AND entre los valores en csr y rs1 (para poner a '0' los *bits* que se precisen). CSRRWI, CSRRSI y CSRRCI actúan igual que los tres anteriores, usando un valor inmediato, en lugar del valor en rs1 [7].

31	20	19	15	14	12	11	7	6	0
csr		rs1		funct3		rd		opcode	
source/dest		src		CSRRW		dest		SYSTEM	
source/dest		src		CSRRS		dest		SYSTEM	
source/dest		src		CSRRC		dest		SYSTEM	
source/dest		uimm[4:0]		CSRRWI		dest		SYSTEM	
source/dest		uimm[4:0]		CSRRSI		dest		SYSTEM	
source/dest		uimm[4:0]		CSRRCI		dest		SYSTEM	
12		5		3		5		7	

Fig. 2.7. Codificación de instrucciones de memoria atómica.

Tanto ECALL *Environment Call* como EBREAK *Environment Break* son instrucciones relevantes para el control y la comunicación entre el *hardware* y el *software* del microprocesador. Pese a que las excepciones son manejadas por este microprocesador, y ambas instrucciones generan una al ser ejecutadas, van a ser las dos únicas instrucciones que no se van a implementar en el microprocesador, puesto que requieren de más de un modo de usuario ó seguridad (consultar sección 8.1.) y de un sistema operativo [7].

31	20	19	15	14	12	11	7	6	0
func12		rs1		funct3		rd		opcode	
ECALL		0		PRIV		0		SYSTEM	
EBREAK		0		PRIV		0		SYSTEM	

Fig. 2.8. Codificación de instrucciones de entorno

■ **Tipo B: Instrucciones de salto condicionado.**

Todas las instrucciones de tipo B actualizan el valor del PC. Llevan a cabo una comparación entre los valores en rs1 y rs2, y en el caso de que se cumpla la condición a evaluar, el PC salta tantas direcciones de memoria como indique el valor inmediato. Este *offset* del PC se codifica para que sea par, y así evitar saltos a direcciones de memoria desalineadas (consultar Sección 3.6.). En el caso de que no se cumpla la condición a evaluar, el PC simplemente pasará a la siguiente instrucción en memoria (PC + 4). BEQ *Branch if Equal* comprueba la igualdad. BNE *Branch if Not Equal*, la desigualdad. BGE *Branch if Greater or Equal* verifica que el valor en rs1 sea mayor o igual que rs2 y BLT *Branch if Less Than* revisa que sea menor o igual. BLTU y BGEU actúan como BLT y BGE pero comparando los valores sin signo [7].

Es importante destacar que estas instrucciones resultan muy útiles para realizar bucles que se repiten mientras se cumpla una condición (como el bucle *while* o *for* en lenguaje C).

31	25	24	20	19	15	14	12	11	7	6	0
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode	
offset[12 10:5]		src2		src1		BEQ/BNE		offset[11 4:1]		BRANCH	
offset[12 10:5]		src2		src1		BLT[U]		offset[11 4:1]		BRANCH	
offset[12 10:5]		src2		src1		BGE[U]		offset[11 4:1]		BRANCH	

Fig. 2.9. Codificación de instrucciones de tipo B.

■ **Tipo J: Instrucciones de salto incondicionado.**

Existe una sola instrucción de tipo J: JAL. Ésta toma un valor inmediato múltiplo de dos (como en las instrucciones de tipo B) y lo usa como *offset* (con signo) en el PC. Al igual que con JALR, JAL también carga la dirección de memoria de la siguiente instrucción en memoria en rd (PC + 4) [7].

31		12	11	7	6	0
imm[20 10:1 11 19:12]			rd	opcode		
offset[20:1]			dest			JAL
20			5	7		

Fig. 2.10. Codificación de instrucciones de tipo J (jal).

■ **Tipo S: Instrucciones de guardado.**

Lo mismo que con las instrucciones de carga (de memoria a registros) LB, LH y LW, existen otras tres instrucciones equivalentes de guardado (de registros a memoria). Estas son SB *Store Byte*, SH *Store Halfword* y SW *Store Word*. Al contrario que con las de carga, dichas instrucciones no extienden el signo. Simplemente guardan los 8, 16 o 32 bits en la dirección de memoria definida por la suma entre el valor en rs1 y un valor inmediato [7].

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	
offset[11:5]		src		base		SB/SH/SW		offset[4:0]		STORE	
7		5		5		3		5		7	

Fig. 2.11. Codificación de instrucciones de tipo S.

■ **Tipo U: Instrucciones de formato de palabra larga.**

Existen dos únicas instrucciones de formato de palabra larga: LUI *Load Upper-Immediate* carga en rd un valor inmediato (formado por los veinte MSBs de la instrucción) y AUIPC *Add Upper-Immediate to Program Counter* que carga en rd la suma entre el valor del PC actual y un valor inmediato (también de palabra larga) [7]. LUI y AUIPC son útiles para guardar datos en direcciones de memoria donde se encuentran periféricos, situados en memoria detrás de la memoria de datos (consultar Sección 3.14.).

31		12	11	7	6	0
imm[31:12]			rd	opcode		
U-immediate[31:12]			dest			LUI
U-immediate[31:12]			dest			AUIPC
20			5	7		

Fig. 2.12. Codificación de instrucciones de tipo U.

3. DISEÑO DEL NÚCLEO DE MICROPROCESADOR Y PERIFÉRICOS

Basándose en la especificación RISC-V, en concreto en la arquitectura RV32I, se ha diseñado un núcleo de microprocesador con dos periféricos.

Un núcleo es una unidad de procesamiento central de un microprocesador. Esta unidad es capaz de ejecutar instrucciones y realizar cálculos de forma independiente. Los microprocesadores actuales suelen contener más de un núcleo, permitiéndoles ejecutar varias tareas o procesos a la vez, de forma paralela.

El siguiente diagrama de bloques muestra la idea general del diseño del núcleo con periféricos:

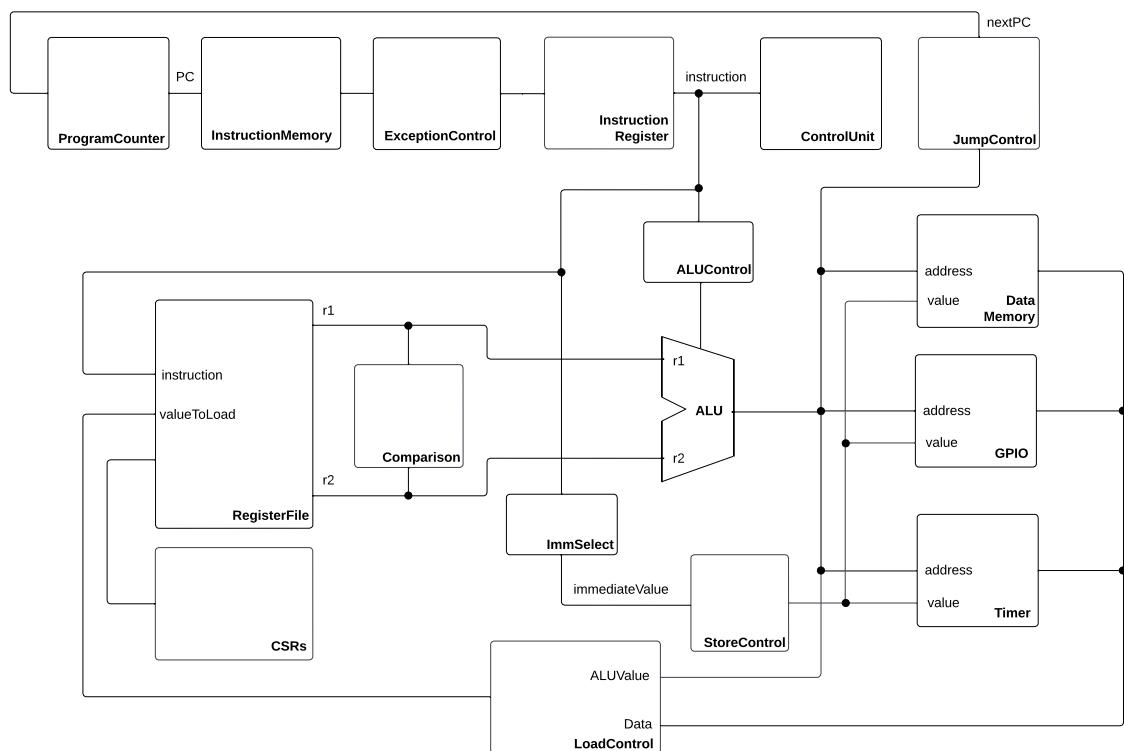


Fig. 3.1. Diagrama de bloques del núcleo y periféricos.

Las principales rutas del diseño son las siguientes; el ProgramCounter actualiza el contador del programa. La instrucción a ejecutar sale del InstructionRegister y entra en la ControlUnit, la cual controla el núcleo mediante una máquina de estados finitos (consultar sección 3.2.10). La pila de registros RegisterFile otorga a la ALU los valores de dos registros. La ALU opera sobre estos valores, y si la instrucción actual quiere cargar el

resultado sobre uno de los registros, éste regresa a RegisterFile a través de LoadControl. Si la instrucción indica guardar un valor a la memoria de datos Datamemory o a uno de los dos periféricos, StoreControl otorga el valor a guardar y la ALU la dirección en la que hacerlo.

A continuación se muestra el proceso de diseño del trabajo, junto con algún ejemplo. Posteriormente, se describe cada uno de los componentes del núcleo y los periféricos, explicando su funcionamiento interno y su relación con otros componentes. Finalmente, se muestra el diseño completo del núcleo con periféricos.

3.1. Proceso de diseño

El diseño del núcleo ha sido un proceso continuo de mejora. Desde el primer diseño funcional con tres componentes hasta la implementación de periféricos y excepciones, se han ido probando las funcionalidades de cada componente de forma individual y en grupos de bloques. Se ha tratado de buscar la eficiencia del núcleo, sin sacrificar el correcto funcionamiento del mismo.

Para su diseño se ha utilizado el lenguaje VHDL, acrónimo formado por VHSIC (*Very High Speed Integrated Circuit*, circuito integrado de velocidad muy alta) y HDL (*Hardware Description Language*, lenguaje de descripción de *hardware*). El programa que se ha utilizado para el diseño, simulación e implementación del microprocesador ha sido Xilinx Vivado ML 2022.2 (software desarrollado por Xilinx (AMD), útil para la simulación y síntesis de diseños HDL).

El proceso de diseño del microprocesador se ha realizado en los siguientes pasos:

1. Diseño de la pila de registros, ALU (*Arithmetic Logic Unit*, unidad aritmético-lógica) y memoria de datos, consiguiendo que funcionen correctamente de forma individual.
2. Conexión y testeо de los tres componentes, añadiéndoles un multiplexor a la salida para seleccionar entre resultados de la ALU y datos de la memoria.

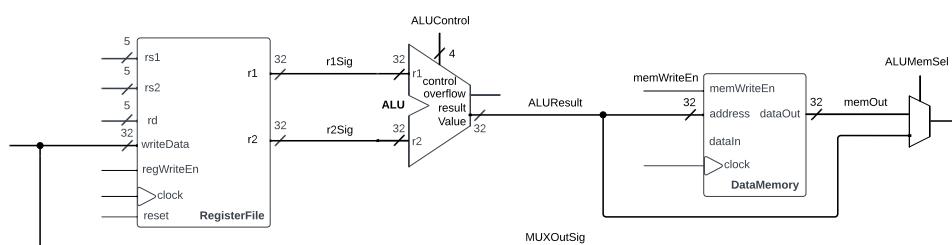


Fig. 3.2. Diagrama de bloques del primer diseño funcional.

3. Diseño del resto de componentes necesarios para ejecutar programas: contador de programa, memoria de instrucciones, registro de instrucciones, control de la ALU, selección del valor inmediato, comparación de registros y control de carga. Se inicia el desarrollo de la unidad de control, identificando los posibles estados de la FSM (*Finite State Machine*, máquina de estados finitos). Consultar sección 3.2.10.

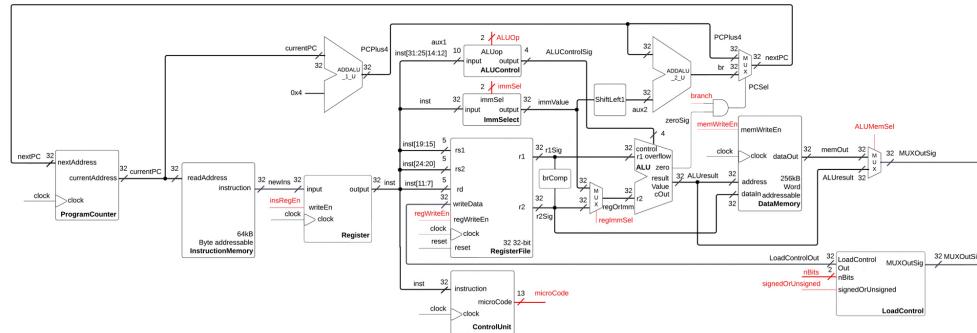
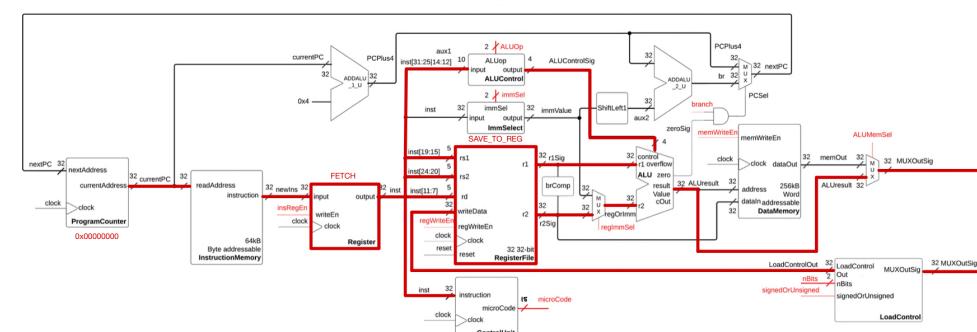


Fig. 3.3. Esquemático del primer diseño funcional.

4. Se comprueban instrucciones de la ISA RV32I y se traza la ruta de datos para completar el ciclo de “FETCH - DECODE - SAVE_TO_REG - SAVE_TO_MEM - SAVE_TO_REG_AND_CSR”. Se diseña durante dicho proceso la unidad de control, codificando manualmente el microcódigo que genera dicha unidad para cada estado. También se codifican las instrucciones de forma manual, introduciéndolas en la memoria de instrucciones para comprobar su comportamiento en simulación.

En este **primer ejemplo**, se obtiene en primer lugar la nueva instrucción a ejecutar haciendo FETCH de “add”. Después en DECODE, se obtienen los dos valores de rs1 y rs2, y se suman en la ALU (como ordena “ALUControl”). El resultado retorna a la pila de registros, donde se carga el valor con “regWriteEn” en SAVE_TO_REG.

add rd,rs1,rs2 rd ← rs1+rs2 , pc ← pc+4



reset → START → FETCH → DECODE → SAVE_TO_REG
0000000000000000 110000100001100 010000100001100 010001100001100

Fig. 3.4. Ejemplo 1. add.

En este **segundo ejemplo**, por un lado se suma “currentPC” y el número 4, y se carga sobre la pila de registros desde “writeData” activando “regWriteEn” (SAVE_TO_REG). Por otro lado, se obtiene la instrucción “jalr” en FETCH, se consigue el valor en rs1 de la pila de registros, y un valor inmediato. La ALU los suma, y el resultado pasa por JumpControl (el cual actualiza “nextPC”).

jalr rd,imm(rs1) $rd \leftarrow PC+4, PC \leftarrow (rs1 + imm_i) \& \sim 1$

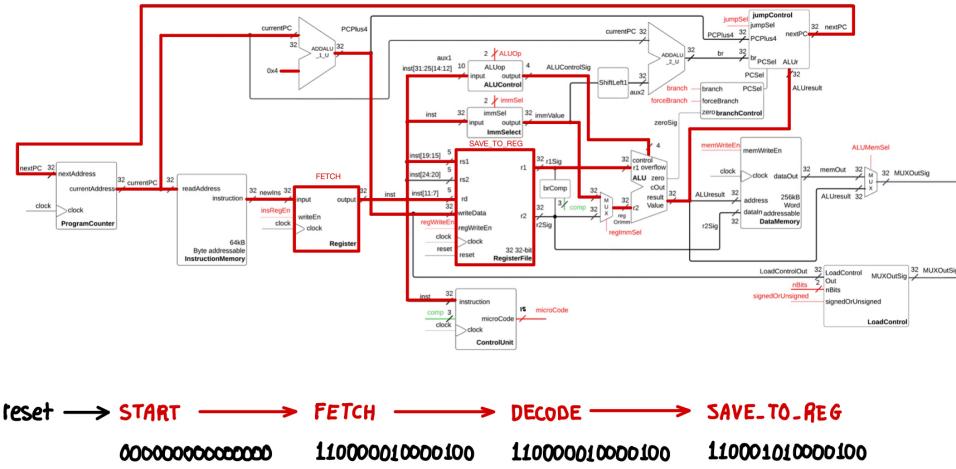


Fig. 3.5. Ejemplo 2. “jalr”.

En este **tercer ejemplo**, se obtiene la instrucción “sb” con el estado FETCH, se suma en la ALU el valor de rs1 y un valor inmediato generado por ImmSelect. El resultado de la ALU actúa como la dirección de memoria de DataMemory (address), mientras que el valor en el registro rs2 es el valor a guardar, entrando a DataMemory por el puerto dataIn. Se guarda el valor en la memoria de datos en el estado SAVE_TO_MEM, activando la señal “memWriteEn”.

sb rs2,imm(rs1) m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4

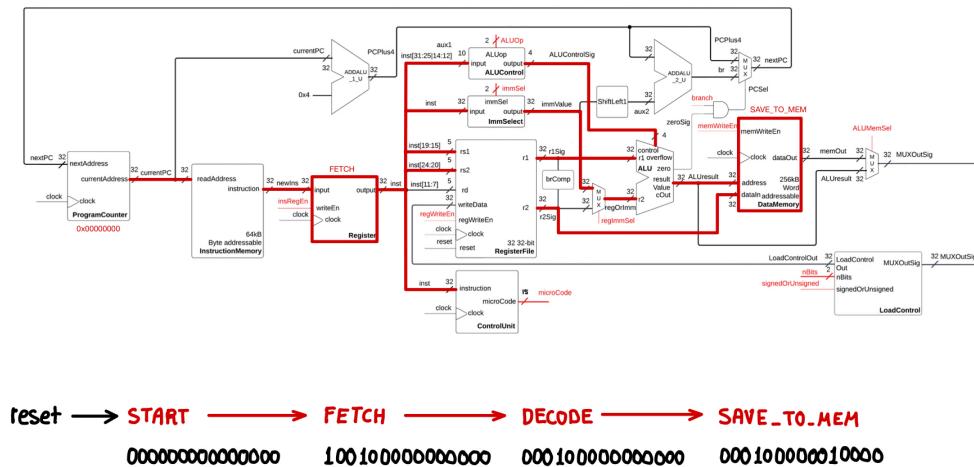


Fig. 3.6. Ejemplo 3. “sb”.

Como se puede apreciar, se guardan los 32 *bits* de rs2 en DataMemory en lugar del *byte* menos significativo, tal y como debería actuar la instrucción “sb”. Para solucionarlo, se añade posteriormente un bloque llamado StoreControl, el cual regula cuántos *bits* se desean guardar en la memoria de datos.

5. Una vez que funcionan las instrucciones que se han puesto a prueba (todas menos las que operan a nivel de *bits* sobre los CSR, así como ecall y ebreak que no se van a implementar) se crean los CSRs y los periféricos ajenos a la memoria de datos. Seguidamente se prueba el sistema completo y se modifica hasta lograr su funcionamiento correcto. Se modifica el diagrama de bloques tantas veces como sea necesario.
6. Por último, se modifica la unidad de control y se añade el bloque ExceptionControl para manejar excepciones (consultar sección 3.2.10.).

3.2. Componentes del núcleo

En esta sección se va a explicar el funcionamiento de cada uno de los componentes (consultar Anexo B, donde se encuentra el código en VHDL para cada uno de ellos).

3.2.1. Contador de programa

Componente síncrono que guarda la dirección de memoria actual de la memoria de instrucciones.

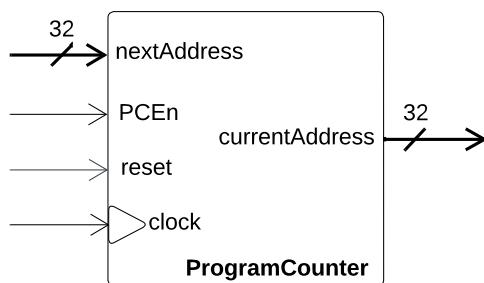


Fig. 3.7. Diagrama de bloques del Contador de Programa.

Actúa como un registro. Si se produce el flanco de subida de reloj y la señal de entrada “PCEn” se encuentra activa, la dirección actual “currentAddress” cambia al valor de la dirección de memoria de la nueva instrucción, “nextAddress”.

3.2.2. Memoria de instrucciones

Componente asíncrono ajeno al núcleo que almacena el programa a ejecutar.

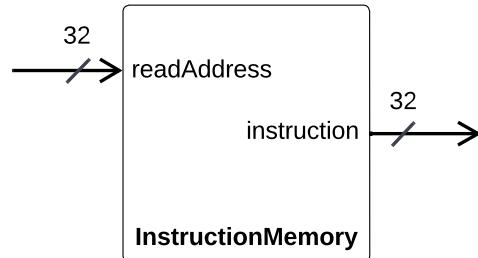


Fig. 3.8. Diagrama de bloques de la Memoria de Instrucciones.

Tras recibir una dirección de memoria del contador de programa a través del puerto *readAddress*, carga la instrucción almacenada en dicha dirección al puerto *instruction*. La memoria implementada es direccionable por *bytes*, almacenando cada instrucción (de 32 bits) en 4 bytes. El formato de la memoria es *big endian* y tiene un tamaño de 1 kB (contiene 2^{10} direcciones).

En las memorias *big endian*, el byte más significativo se almacena en la dirección de memoria más baja, seguido por los bytes menos significativos en orden creciente.



Fig. 3.9. Comparativa little/big endian guardando la instrucción 0x01234567 en los primeros 4 bytes de memoria.

El tamaño de la memoria es reducido, puesto que no se van a ejecutar grandes programas en las pruebas realizadas de esta memoria (del orden de hasta unos 0,4 kB). Si se quisiera ejecutar mayores programas en este núcleo, bastaría con modificar el valor de la constante `INS_MEM_SIZE` en `InstructionMemory`.

3.2.3. Control de excepciones

Componente asíncrono que maneja la entrada de la nueva instrucción a ejecutar por el procesador.

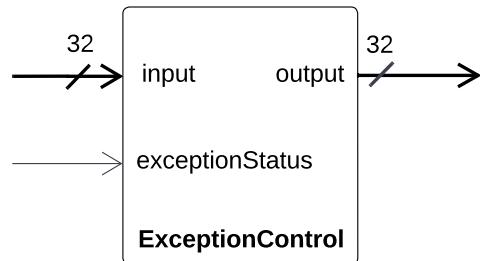


Fig. 3.10. Diagrama de bloques del Control de Excepciones.

En el momento en que la unidad de control detecte una excepción, el bloque control de excepciones detectará que la señal “exceptionStatus” se ha activado, por lo que modificará la nueva instrucción a ejecutar por una instrucción de salto a la dirección de memoria 0x4, donde se encuentra el *Exception Handler* (controlador de excepciones). Consultar la sección 3.2.10 para más información acerca del manejo de excepciones en la FSM.

3.2.4. Registro de instrucciones

Registro síncrono que almacena la próxima instrucción a ejecutar.

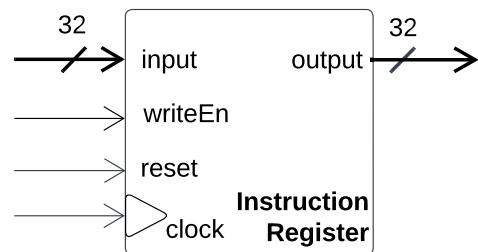


Fig. 3.11. Diagrama de bloques del Registro de Instrucciones.

3.2.5. Control de la Unidad Aritmético-Lógica

Componente asíncrono que genera una señal de control para la ALU.

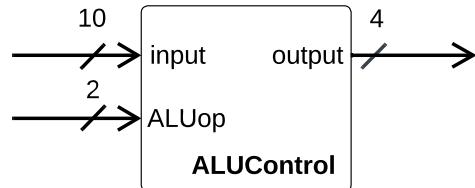


Fig. 3.12. Diagrama de bloques del Control de la ALU.

TABLA 3.1. GENERACIÓN DE SEÑAL “OUTPUT” EN
ALUCONTROL

ALUop	input (inst[31:25 14:12])	output (ALUControlSig)	Instrucción
"00"	"-----"	"0000"	lw / sw
"01"	"-----"	"1000"	beq
"10"	"0000000000"	"0000"	add
"10"	"0100000000"	"1000"	sub
"10"	"0000000001"	"0001"	sll
"10"	"0000000010"	"0010"	slt
"10"	"0000000011"	"0011"	sltu
"10"	"0000000100"	"0100"	xor
"10"	"0000000101"	"0101"	srl
"10"	"0100000101"	"1101"	sra
"10"	"0000000110"	"0110"	or
"10"	"0000000111"	"0111"	and
"11"	"----000"	"0000"	addi
"11"	"----010"	"0010"	slti
"11"	"----011"	"0011"	sltiu
"11"	"----100"	"0100"	xori
"11"	"----110"	"0110"	ori
"11"	"----111"	"0111"	andi
"11"	"0000000001"	"0001"	slli
"11"	"0000000101"	"0101"	srl
"11"	"0100000101"	"1101"	srai
others	"-----"	"0000"	lw / sw

**TABLA 3.2. GENERACIÓN DE SEÑAL “OUTPUT” EN
ALUCONTROL PARA INSTRUCCIÓN “LUI”**

ALUop	input (inst[6:0])	output (ALUControlSig)	Instrucción
"11"	"0110111"	"1111"	lui

Dada la señal “ALUOp” que indica una instrucción load, store, beq, de tipo R o I, junto con una señal de entrada obtenida de la propia instrucción, ALUControl genera una señal de 4 *bits* necesaria para que la ALU sepa qué operación realizar.

3.2.6. Selección de valor inmediato

Componente asíncrono que construye el valor inmediato de 32 *bits*, adecuado para cada tipo de instrucción que contenga valores inmediatos.

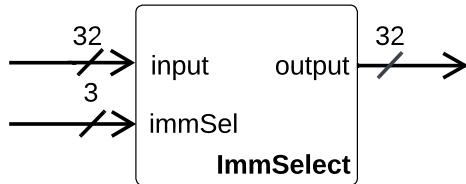


Fig. 3.13. Diagrama de bloques de la Selección de valor inmediato.

Como se puede apreciar en la siguiente tabla, la señal de tres *bits* “immSel” indica al componente qué tipo de valor inmediato se desea generar.

**TABLA 3.3. GENERACIÓN DE SEÑAL “IMMSEL” PARA CADA
TIPO DE INSTRUCCIÓN**

immSel	Tipo de instrucción
"000"	I
"001"	B
"010"	S
"011"	U
"100"	J
"101"	Atómica
<i>others</i>	-

■ Valores inmediatos en instrucciones de tipo I

31	20	19	15	14	12	11	7	6	0		
imm[11:0]					rs1	funct3	rd	opcode			
a	b	c	d	e	f	g	h	i	j	k	l
					12		5	3	5		7
31					12	11	0				
a	a	a	a	a	a	a	a	a	a	a	a
					20			12			

Fig. 3.14. Codificación del valor inmediato para instrucciones de tipo I. Superior: [instrucción]. Inferior: [valor inmediato].

En las instrucciones de tipo I, el valor inmediato se codifica a partir de un valor de 12 bits, al cual se le extiende el signo de modo que complete los 32 bits. El rango del valor inmediato es de [-2048, 2047] [7].

■ Valores inmediatos en instrucciones de tipo B

Fig. 3.15. Codificación del valor inmediato para instrucciones de tipo B. Superior: [instrucción]. Inferior: [valor inmediato].

En las instrucciones de tipo B, el valor inmediato se codifica a partir de un valor de 12 bits, el cual se desplaza lógicamente 1 bit a la izquierda, y se extiende su signo hasta los 32 bits. El rango del valor inmediato es de [-4096, 4094] [7].

■ Valores inmediatos en instrucciones de tipo S

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	
a	b	c	d	e	f	g		u	v	w	x
7			5		5		3	5		7	

31	12	11	5	4	0
a	b	c	d	e	f
a	a	a	a	a	a

Fig. 3.16. Codificación del valor inmediato para instrucciones de tipo S. Superior: [instrucción]. Inferior: [valor inmediato].

En las instrucciones de tipo S, el valor inmediato se codifica a partir de un valor de 12 bits, al cual se le extiende el signo para completar los 32 bits (similar al tipo I, pero con distinta codificación en la propia instrucción). El rango del valor inmediato es de [-2048, 2047] [7].

■ Valores inmediatos en instrucciones de tipo U

31	imm[31:12]	12	11	7	6	0
a b c d e f g h i j k l m n o p q r s t			rd		opcode	
	20			5		7
31		12	11			0
a b c d e f g h i j k l m n o p q r s t		0 0 0 0 0 0 0 0 0 0				0
	20			12		

Fig. 3.17. Codificación del valor inmediato para instrucciones de tipo U. Superior: [instrucción]. Inferior: [valor inmediato].

En las instrucciones de tipo U, se codifica un valor inmediato a partir de los 20 MSBs de la instrucción con 12 bits a su derecha [7].

■ Valores inmediatos en instrucciones de tipo J

31	imm[20:10:1][11:19:12]	12	11	7	6	0
a b c d e f g h i j k l m n o p q r s t			rd		opcode	
	20			5		7
31	20	19	12	11	10	1 0
a a a a a a a a a a	m n o p q r s t	l	b c d e f g h i j k			
12	8	1	10			1

Fig. 3.18. Codificación del valor inmediato para instrucciones de tipo J. Superior: [instrucción]. Inferior: [valor inmediato].

En las instrucciones de tipo J, se codifica un valor inmediato a partir de 20 *bits* en la instrucción, desplazando lógicamente un *bit* a la izquierda (para que el valor sea siempre par), y se extiende su signo para alcanzar los 32 *bits*. El valor inmediato tiene un rango de [-1048576, 1048574] [7].

■ Valores inmediatos en instrucciones de memoria atómica

Fig. 3.19. Codificación del valor inmediato para instrucciones de memoria atómica. Superior: [instrucción]. Inferior: [valor inmediato].

En las instrucciones de memoria atómica, se extiende con ceros 5 *bits* en la instrucción. Rango final de [0, 31] [7].

3.2.7. Comparación de registros

Componente asíncrono que compara los valores r1 y r2.

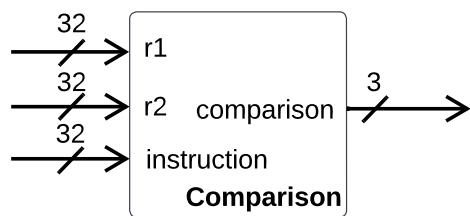


Fig. 3.20. Diagrama de bloques de la comparación de registros.

Como se puede ver en la siguiente tabla, se genera una señal de 3 *bits* (comparison) en base al resultado de la comparación.

TABLA 3.4. GENERACIÓN DE LA SEÑAL “COMPARISON”

	comparison
signed(r1) == signed(r2)	"000"
signed(r1) < signed(r2)	"001"
signed(r1) > signed(r2)	"010"
unsigned(r1) < unsigned(r2)	"011"
unsigned(r1) > unsigned(r2)	"100"

“signed” denota una comparación de valores con signo (valores enteros). ”unsigned” por otro lado, denota una comparación de valores sin signo (valores naturales).

3.2.8. Pila de registros de propósito general

Componente de escritura síncrona y lectura asíncrona, formado por un conjunto de 32 registros de 32 *bits* de longitud.

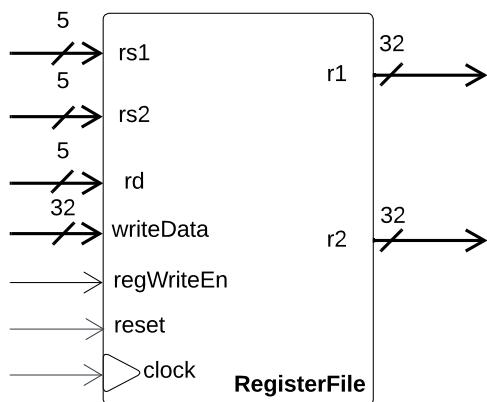


Fig. 3.21. Diagrama de bloques de la pila de registros.

Dadas dos direcciones de 5 *bits* (entradas rs1 y rs2), los puertos r1 y r2 indican de manera asíncrona los valores almacenados en dichas direcciones.

A su vez, dado un flanco de subida de reloj y el puerto de entrada regWriteEn, se escribe el valor en el puerto de entrada writeData sobre el registro que indique el puerto rd.

TABLA 3.5. NOMBRES SIMBÓLICOS Y FUNCIONALIDAD DE LOS REGISTROS DE PROPÓSITO GENERAL.

Registro	Nombre simbólico	Descripción
x0	zero	Registro nulo (valor 0x00000000)
x1	ra	Dirección de retorno
x2	sp	Puntero al <i>stack</i> (pila)
x3	gp	Puntero global
x4	tp	Puntero a un hilo
x5	t0	Registro de valor temporal / <i>alternate link</i>
x6 - x7	t1 - t2	Registros de valores temporales
x8	s0 / fp	Registro guardado / <i>frame pointer</i>
x9	s1	Registro guardado
x10 - x11	a0 - a1	Argumentos de función / Valores de retorno
x12 - x17	a2 - a7	Argumentos de función
x18 - x27	s2 - s11	Registros guardados
x28 - x31	t3 - t6	Registros de valores temporales

El uso de los registros es decisión del desarrollador. El núcleo no hace distinción entre registros, a excepción del x0 (registro nulo). Si se trata de modificar el valor almacenado en el registro nulo, dicho valor no cambia. Debido a su gran utilidad, el registro nulo debe estar siempre disponible.

3.2.9. Registros de Control y Estado

Componente formado por dos registros de 32 bits que ayudan a manejar excepciones y conocer el motivo de las mismas.

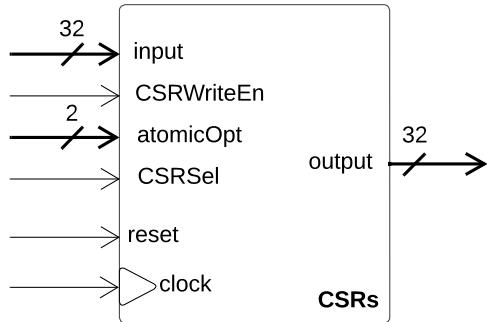


Fig. 3.22. Diagrama de bloques de los Registros de Control y Estado.

- CSR 0 (mtvec): Indica la dirección de memoria del *exception handler* (0x4 en el caso de este procesador). La Unidad de Control lo consulta para saber a dónde saltar en caso de una excepción.
- CSR 1 (mcause): Indica el motivo de la excepción. El bit 0 indica una INV (Instrucción No Válida), y el bit 1 una señal externa de *interrupt* (interrupción).

3.2.10. Unidad de Control

Máquina de estados finitos que controla el funcionamiento del procesador.

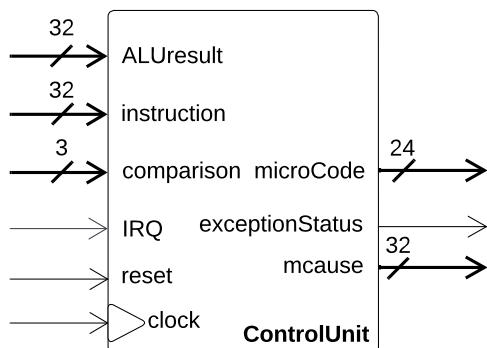


Fig. 3.23. Diagrama de bloques de la Unidad de Control.

■ Máquina de Estados Finitos (FSM)

Una Máquina de Estados Finitos es un modelo que representa los estados en los que se puede encontrar un sistema, y cómo transiciona éste entre ellos dada una serie de condiciones.

La Unidad de Control de este procesador es una FSM en sí misma. Cambia de estado tras cada flanco de subida del reloj, por lo que va modificando el contador de programa según le convenga, y ejecutando cada instrucción siguiendo un ciclo FETCH-DECODE-EXECUTE.

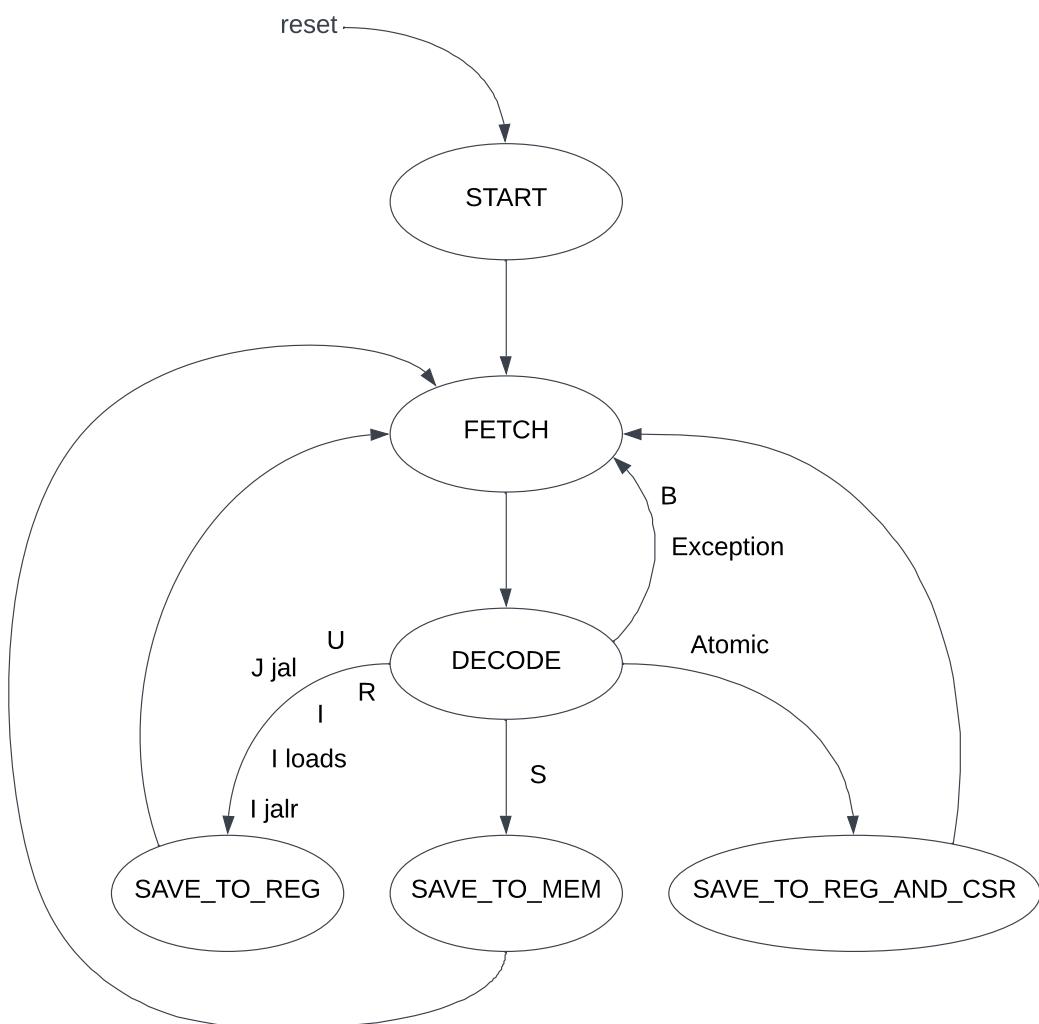


Fig. 3.24. Diagrama de la máquina de estados finitos.

El procesador nunca debe pararse y debe estar siempre ejecutando una instrucción. Si se desea que un programa “acabe”, se debería añadir una instrucción de salto a sí misma al final del programa. De esta forma, se evita que el procesador acceda a direcciones de la memoria de instrucciones vacías (instrucciones de todo ceros), con sus correspondientes consecuencias (bucle de excepciones por instrucciones no válidas, avanzando en memoria hasta alcanzar direcciones de memoria no existentes, produciendo un comportamiento no deseado).

La FSM se compone de los siguientes estados:

- START: Estado inicial del micro. Siempre que se activa el “reset” se retorna a este estado.
- FETCH: En este estado, el registro de instrucciones se encarga de obtener la nueva instrucción a ejecutar (la señal “insRegEn” se activa).
- DECODE: Este estado actúa como DECODE y como EXECUTE realmente. La FSM activa las señales necesarias para realizar operaciones lógicas y aritméticas, extender valores inmediatos, establecer la dirección del próximo PC, y muchas más funcionalidades. Este estado solo deja pendiente la escritura sobre registros, o periféricos.
- SAVE_TO_REG: En el caso de instrucciones que requieran cargar valores en los registros de propósito general, la FSM activa la señal “regWriteEn”, permitiendo su escritura.
- SAVE_TO_MEM: Al igual que con SAVE_TO_REG, este estado actuará para instrucciones que guarden valores en periféricos, activando la señal “memWriteEn”.
- SAVE_TO_REG_AND_CSR: Para instrucciones de memoria atómica, las cuales activan o eliminan *bits* de los CSR, la FSM usa este estado. Éste carga valores en tanto los registros de propósito general, como en los CSR (activa “regWriteEn” y “CSRWriteEn”).

Para controlar el microprocesador en cada estado, la Unidad de Control genera una señal de 24 *bits* llamada “*microcode*” (microcódigo).

El microcódigo de un microprocesador contiene todas las señales de control del mismo. En este caso, los 24 *bits* de microcódigo contienen las 17 señales de control del micro.

23	22	21	20	19	18
CSRWriteEn	atomicOpt		r1orZimm	auipc	PCEn
17	16	15	14	13	12
insRegEn	ALUOp			immSel	
11	10	9	8	7	6
regWriteEn	wdSel		regImmSel	jumpSel	PCSel
5	4	3	2	1	0
memWriteEn	ALUMemSel		nBits		signedOrUnsigned

Fig. 3.25. Microcódigo de la FSM.

- CSRWriteEn: 1 bit que permite la escritura sobre los CSRs.
- atomicOpt: 2 bits que indican a los CSR qué deben cargar en ellos dependiendo de la instrucción de memoria atómica a ejecutar. “00” para csrrw[i], “01” para csrrs[i], y “10” para csrrc[i].
- r1orZimm: 1 bit que selecciona qué cargar a los CSR. ’0’ para el valor del registro r1, y ’1’ para el valor inmediato que genera ImmSelect.
- auipc: 1 bit que actualiza el nuevo PC al valor actual del PC + un valor inmediato, como indica la instrucción auipc.
- PCEn: 1 bit que actualiza el PC.
- insRegEn: 1 bit que actualiza la instrucción actual en el registro de instrucciones.
- ALUOp: 2 bits que seleccionan en ALUControl qué operación debe realizar la ALU dependiendo del tipo de instrucción. “00” para las instrucciones “lw” ó “sw”, “01” para instrucciones de tipo B, “10” para tipo R, y “11” para tipo I o la instrucción “lui”.
- immSel: 3 bits que seleccionan en ImmSelect qué valor inmediato generar. “000” para tipo I, “001” para tipo B, “010” para tipo S, “011” para tipo U, “100” para tipo J, y “101” para operaciones de memoria atómica.
- regWriteEn: 1 bit que permite la escritura sobre la pila de registros.
- wdSel: 2 bits que seleccionan qué cargar sobre la pila de registros (para cualquier instrucción que vaya a hacerlo). “00” para el PC + 4, “01” para la carga de valores en instrucciones de tipo R y loads, y “10” para valores en CSRs.
- regImmSel: 1 bit que selecciona qué segundo operando utiliza la ALU. ’0’ para el valor inmediato que genera ImmSelect, y ’1’ para el valor del registro r2.
- jumpSel: 1 bit que indica a JumpControl que el próximo PC debe ser el resultado de la ALU (en jalr).

- PCSel: 1 bit que indica a JumpControl que el próximo PC debe ser o bien con un '0' el PC actual + 4, o con un '1' el valor de la branch (PC actual + valor inmediato).
- memWriteEn: 1 bit que permite la escritura sobre periféricos.
- ALUMemSel: 2 bits que seleccionan qué cargar sobre la pila de registros en instrucciones de tipo R y loads. "00" para la memoria, "01" para el resultado de la ALU, "10" para el GPIO, y "11" para el Timer.
- nBits: 2 bits que seleccionan cuántos bits cargar sobre la pila de registros (para diferenciar lb, lh, lw, entre otros). "00" para 8, "01" para 16, y "10" para 32.
- signedOrUnsigned: 1 bit que indica la extensión de bits en loads. '0' para extensión de ceros, y '1' para extensión de signo.

■ Funcionamiento de la FSM

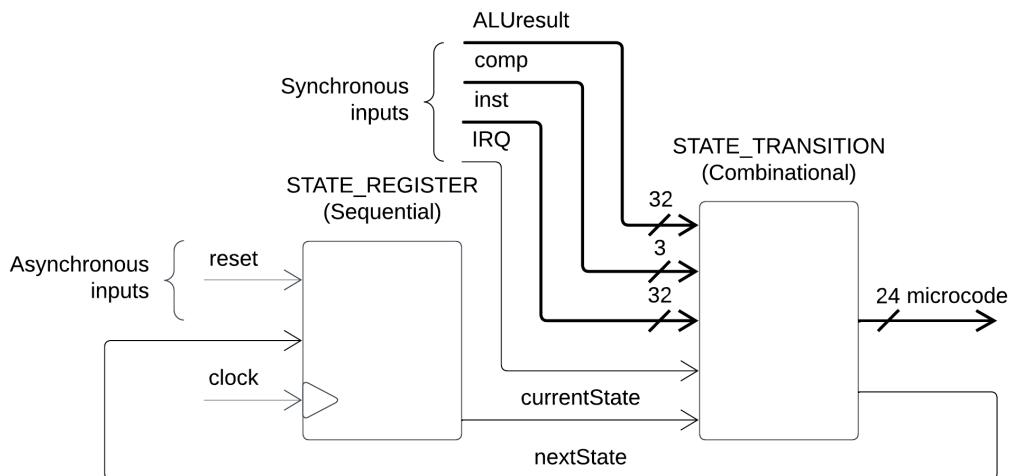


Fig. 3.26. Diagrama de bloques de la Unidad de Control.

El funcionamiento de la máquina de estados viene determinado por dos procesos distintos.

- **STATE_REGISTER:** El registro de estado es responsable de almacenar el estado actual de la máquina de estado finita. Funciona como una memoria que guarda el estado en el que se encuentra el sistema en un momento dado.
Toma como entradas todas aquellas que actúen de forma asíncrona ("reset" en este caso).
- **STATE_TRANSITION:** La transición de estado se encarga de gestionar la transición de un estado a otro en la máquina de estado finita. Define las reglas y condiciones bajo las cuales se produce el cambio de estado. El proceso de transición se

activa cuando se cumple una condición de activación, y luego se realiza la transición al estado correspondiente según las reglas definidas.

Toma como entradas todas aquellas que actúen de forma síncrona (“ALUresult”, “comp”, “inst” y “IRQ” *Interrupt Request* (petición de interrupción) en este caso).

■ Manejo de excepciones

La Unidad de Control debe de ser también capaz de manejar cuándo el procesador se encuentra con una excepción. Ésta puede tratarse de una INV o de una señal externa de interrupción.

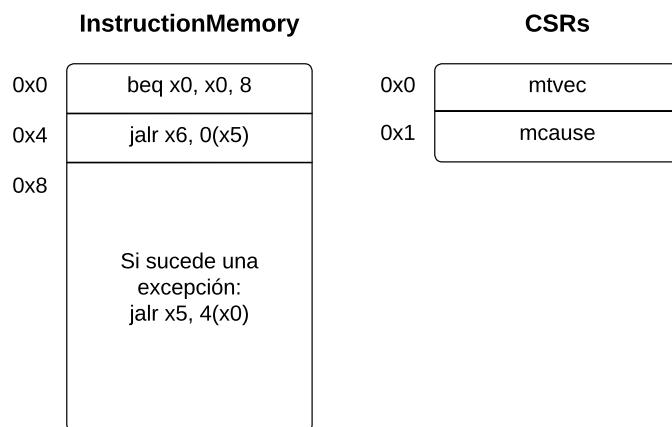


Fig. 3.27. Distribución de la memoria para el control de excepciones.

Como se puede ver en la imagen anterior, la memoria de instrucciones se organiza de una cierta forma para permitir el manejo de excepciones. El procesador ejecuta instrucciones desde la dirección 0x0. Ésta trata de un salto a la dirección 0x8, donde reside el programa a ejecutar como tal. En el caso que durante el programa suceda una excepción, la unidad de control activará la señal “exceptionStatus” y la causa de la excepción en el registro de control y estado 0x1 o “mcause” (consultar sección 3.8 para más detalles), indicándole a la unidad de control de excepciones que debe sustituir la instrucción actual por la instrucción “jalr x5, 4(x0)”. Esta instrucción hace un salto a la dirección 0x4, donde se encuentra el *Exception Handler*. Éste retornará al programa donde se encontraba la excepción, pero en la siguiente instrucción a ejecutar.

La comprobación de excepciones se hace durante el estado DECODE, donde se comprueba que la instrucción actual pertenezca a la ISA RISC-V, y que no se haya recibido una señal externa de interrupción (del *Timer* (temporizador) en este caso).

Si se da una excepción, la FSM retorna al estado FETCH, donde se obtendrá la nueva instrucción otorgada por la unidad de control de excepciones (jalr x5, 4(x0)).

- INV: La Unidad de Control detecta que los campos funct7, funct3 y opcode (*Operation Code*, código de operación) no corresponden con ninguna instrucción existente.

El resto de posibles campos (rs1, rs2, rd, valores inmediatos, CSRs) no pueden ser inválidos por el propio tamaño de *bits* que disponen.

El *bit* 0 de mcause corresponde a una INV.

- IRQ: Dada una señal externa de interrupción (IRQ), el procesador actúa como con una INV.

El *bit* 1 de mcause corresponde con una IRQ.

3.2.11. Unidad Aritmético-Lógica

Componente asíncrono fundamental de cualquier procesador. La labor de la ALU consiste en realizar operaciones aritméticas y lógicas.

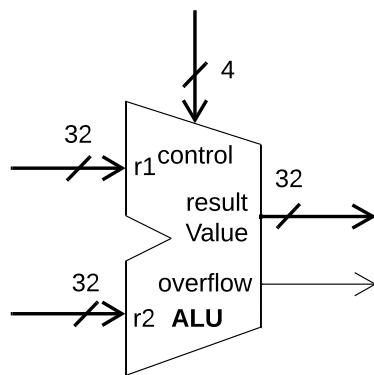


Fig. 3.28. Diagrama de bloques de la Unidad Aritmetico-Lógica.

TABLA 3.6. OPERACIÓN A EJECUTAR POR LA ALU PARA CADA SEÑAL DE “CONTROL”

control	Instrucción	Significado
"0000"	add	Add (suma)
"1000"	sub	Subtract (resta)
"0001"	sll	Shift Left Logic (desplazamiento lógico a la izquierda)
"0010"	slt	Set if Less Than (establecer si es menor que)
"0011"	sltu	Set if Less Than Unsigned (establecer si es menor que sin signo)
"0100"	xor	Exclusive Logic Or (Operación lógica exclusiva ó)
"0101"	srl	Shift Right Logic (desplazamiento lógico a la derecha)
"1101"	sra	Shift Right Arithmetic (desplazamiento aritmético a la derecha)
"0110"	or	Logic Or (“ó” lógico)
"0111"	and	Logic And (“y” lógico)
"1111"	lui	Load Upper Immediate (cargar inmediato superior)
others	-	-

La ALU activa un *flag* (indicador) de *overflow* (desbordamiento) en caso de desbordamiento de *bits*. Esto puede suceder al realizar operaciones aritméticas, o al desplazar *bits* aritméticamente (add, addi, auipc, lb, lbu, lh, lhu, lw, sb, sh, sra, srai, sub ó sw).

3.2.12. Control de instrucciones de guardado

Componente asíncrono que regula el número de *bits* a guardar.

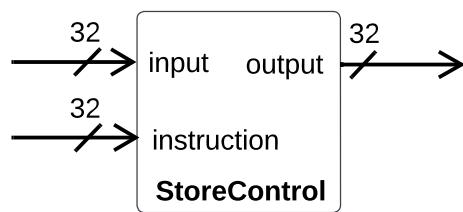


Fig. 3.29. Diagrama de bloques del control de instrucciones de guardado.

Dado un opcode (*bits* 6 a 0 de la instrucción) de "0100011", que indica una instrucción de guardado, el componente añade a la salida tantos *bits* como indique el campo funct3 (*bits* 14 a 12).

- "000": La instrucción es “sb” (*store byte*, guardar byte), por lo que solo se cargan los 8 LSBs.
- "001": La instrucción es “sh” (*store halfword*, guardar media palabra), por lo que solo se cargan los 16 LSBs.
- "010": La instrucción es “sw” (*store word*, guardar palabra), por lo que carga el valor entero de input a output.

3.2.13. Control de instrucciones de salto

Componente asíncrono que establece la dirección de memoria de la próxima instrucción a ejecutar.

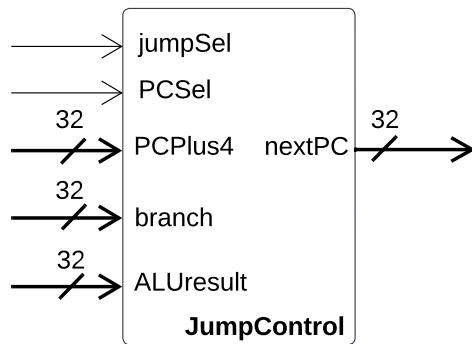


Fig. 3.30. Diagrama de bloques del control de instrucciones de salto.

Si la instrucción a ejecutar es “jalr”, jumpSel se activa y el nextPC es ALUresult (la suma del valor en rs1 y un valor inmediato). En cambio, si la instrucción es de tipo B ó es “jal”, el nextPC es branch. Finalmente, para cualquier otra instrucción, el nextPC es PCPlus4.

La convención estándar de llamada de los registros de propósito general establece x1 como el registro de dirección de retorno. Cuando se llama a una función, la dirección de retorno, es decir, la dirección de la próxima instrucción que se ejecutará cuando la función termine, se almacena en x1. Esto permite a la función saber a dónde retornar una vez termine su ejecución.

Por otro lado, x5 se establece como un registro de enlace alternativo. Si una función necesita almacenar temporalmente un valor que va a modificar, puede recuperarlo al final de su ejecución en el registro x5.

Importante destacar la preferencia de utilizar instrucciones de salto no condicionado como “jal” en lugar de utilizar instrucciones de salto condicionado con comparaciones que siempre sean ciertas. La instrucción “jal” tiene mayor rango de *offset* sobre el PC, y evita realizar comparaciones innecesarias que ralentizan el núcleo [7].

3.2.14. Control de instrucciones de carga

Componente asíncrono que regula los bits a cargar sobre la pila de registros.

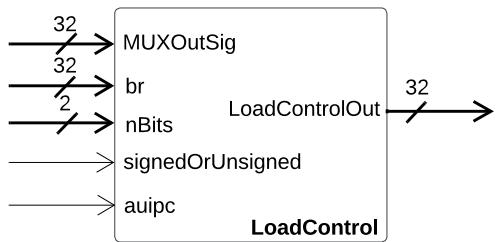


Fig. 3.31. Diagrama de bloques del control de instrucciones de carga.

Tal y como actúa la instrucción “auipc”, si el puerto correspondiente se encuentra activo, LoadControlOut carga sobre rd la suma del PC y un valor inmediato de tipo U. En caso contrario, LoadControl evaluará el número de *bits* (con nBits) provenientes de MUXOutSig a cargar, y extenderá su signo en caso necesario (loads que no sean “lbu” ó “lhu”, en cuyo caso se extenderán con ceros).

3.2.15. Memoria de datos

Memoria de 16 kB compuesta por 4096 palabras de 32 *bits*. Al igual que con la pila de registros, su escritura es síncrona, y su lectura asíncrona.

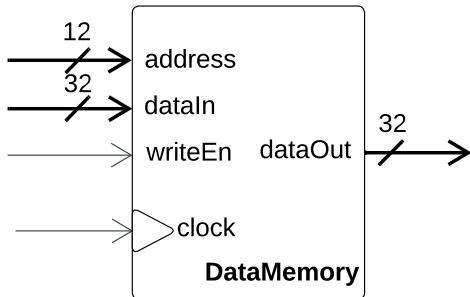


Fig. 3.32. Diagrama de bloques de la memoria de datos.

3.3. Periféricos

En este proyecto se han implementado dos periféricos distintos; un GPIO (*General Purpose Input/Output*, entrada-salida de propósito general), y un Timer. Los dos disponen de memoria, y dan cierta funcionalidad adicional a la CPU (*Central Processing Unit*, unidad central de procesamiento).

Para que éstos puedan actuar como periféricos (interactuar con la CPU, mediante una serie de entradas/salidas), deben tener definida una misma serie de puertos comunes:

- address: Dirección de memoria sobre la que se quiere escribir.
- dataIn: Valor que se quiere escribir en la dirección de memoria “address”.
- writeEn: Puerto enable para permitir la escritura sobre el periférico.
- clock: Escritura síncrona.

A parte de estos, el GPIO y el *Timer* disponen de un puerto de entrada reset (a diferencia de la memoria de datos, que al no estar compuesta de registros, no debe resetearse).

Dejando la interfaz CPU-periféricos aparte, los periféricos pueden tener las salidas que quieran, siempre y cuando sean externas al microprocesador.

Para poder seleccionar si se quiere escribir sobre la memoria de datos, o a uno de los periféricos, se utiliza la dirección de memoria, y el puerto de Enable.

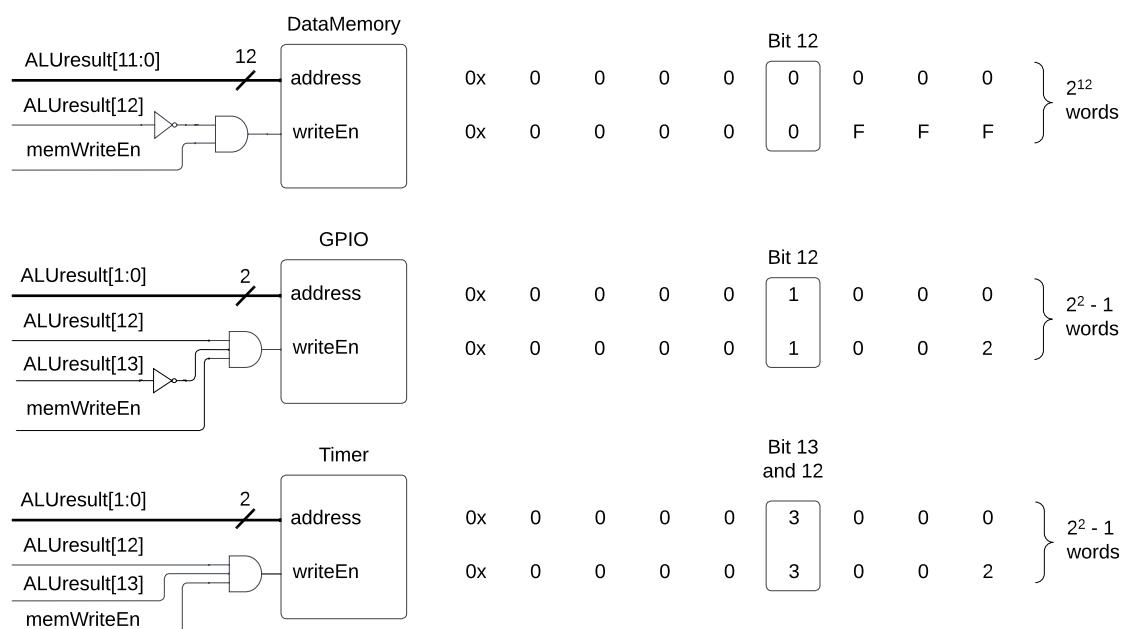


Fig. 3.33. Direccionamiento de periféricos

Como se puede observar en la anterior figura, el direccionamiento de los periféricos se realiza con la propia dirección de memoria. Si la memoria de datos ocupa el rango de direcciones 0x00000000 a 0x00000FFF, el GPIO de 0x00001000 a 0x00001002 y el *Timer* de 0x00003000 a 0x00003002, se pueden usar los *bits* 12 y 13 para diferenciar si se quiere escribir sobre la memoria de datos (cuando el bit 12 vale ‘0’), sobre el GPIO (cuando valen “01”), y sobre el *Timer* (cuando valen “11”). Para implementarlo, se puede invertir

el *bit* 12 para la memoria de datos, y el 13 para el GPIO. Se añade además una puerta lógica AND a la entrada de writeEn para cada uno de los tres componentes, de modo que cuando la unidad de control ordene escribir sobre uno de ellos, la señal memWriteEn lo permita, y la propia dirección de memoria indique sobre qué componente escribir.

3.3.1. GPIO

Componente que actúa como la interfaz entre el núcleo y los pines de la FPGA.

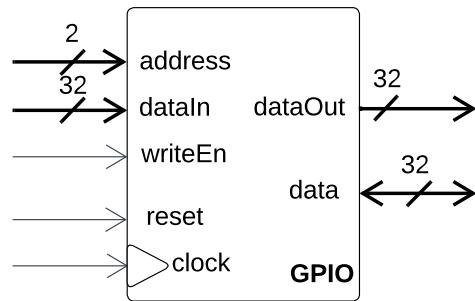


Fig. 3.34. Diagrama de bloques de los GPIOs.

Un GPIO trata de una serie de pines (32 en este caso), los cuales se pueden utilizar individualmente como entradas o salidas en tiempo de ejecución del programa.

En la siguiente figura se puede ver el funcionamiento interno del GPIO para un solo pin (ídem para los 32 pines):

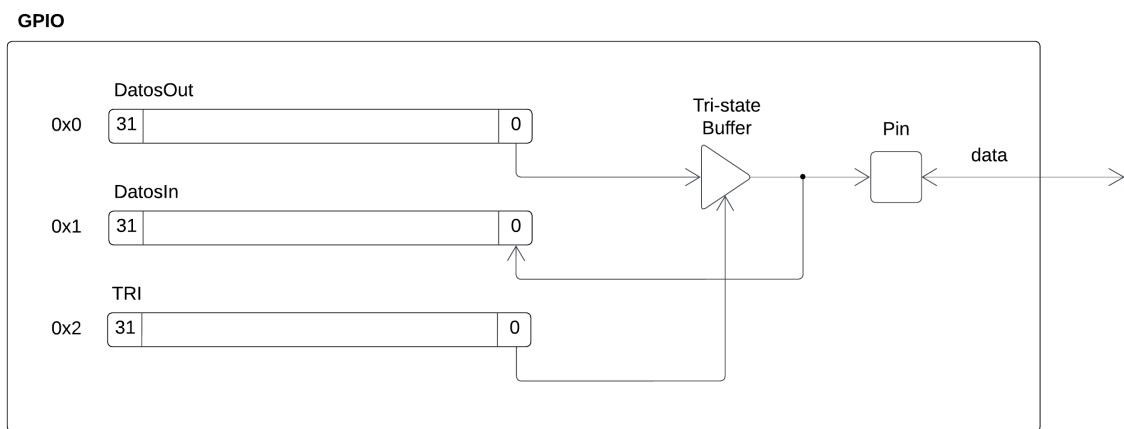


Fig. 3.35. Funcionamiento del GPIO para un solo pin.

- DatosOut (registro 0): Registro de datos de salida.
- DatosIn (registro 1): Registro de datos de entrada.

- TRI (registro 2): Registro que controla 32 diferentes *buffers* triestado.

Un *buffer* (memoria intermedia) triestado es un tipo de *buffer* digital. Dependiendo de la señal de control (*bit* del registro TRI), actúa como un interruptor, o como una alta impedancia.

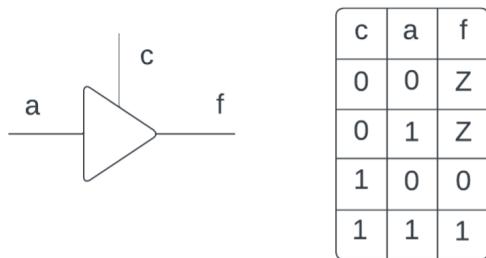


Fig. 3.36. Simbolo y tabla de verdad de un *buffer* triestado.

Cuando el *bit* de control es '1', la salida del *buffer* tiene el mismo valor binario que el *bit* correspondiente de DatosOut. El pin correspondiente actúa como salida.

Cuando el *bit* de control es '0', la salida del *buffer* actúa como una alta impedancia. El pin correspondiente actúa como una entrada y éste se guarda en DatosIn.

De esta forma, si los pines de la FPGA que se quieran usar, se conectan al puerto data del GPIO, éstos pueden actuar como entrada o como salida de forma individual.

Si el micro quiere que uno o más pines actúen como salida, éste guardará los *bits* que desee en el primer registro del GPIO (DatosOut), y activará los *bits* correspondientes en el registro TRI.

Si el micro quiere en cambio leer datos de los pines, podrá cargar el registro DatosIn del registro a los registros de propósito general a través del puerto dataOut del GPIO.

El registro TRI del GPIO es configurado para hacer actuar los *buffer* triestado como entradas por defecto (TRI = 0x00000000). Esto evita generar salidas no deseadas que puedan generar cortocircuitos y daños en la FPGA.

3.3.2. Timer

Componente síncrono que contiene un contador interno, y permite medir intervalos de tiempo.

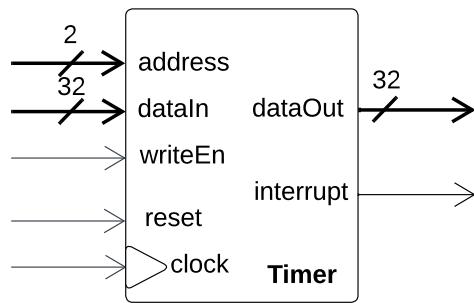


Fig. 3.37. Diagrama de bloques del *Timer*.

Un *timer* o temporizador, es un componente que incrementa o decrementa un contador en cada ciclo de reloj hasta alcanzar un valor preestablecido. Al hacerlo, se desencadena una señal de interrupción que recibe la unidad de control. El *Timer* implementado realiza una cuenta ascendente.

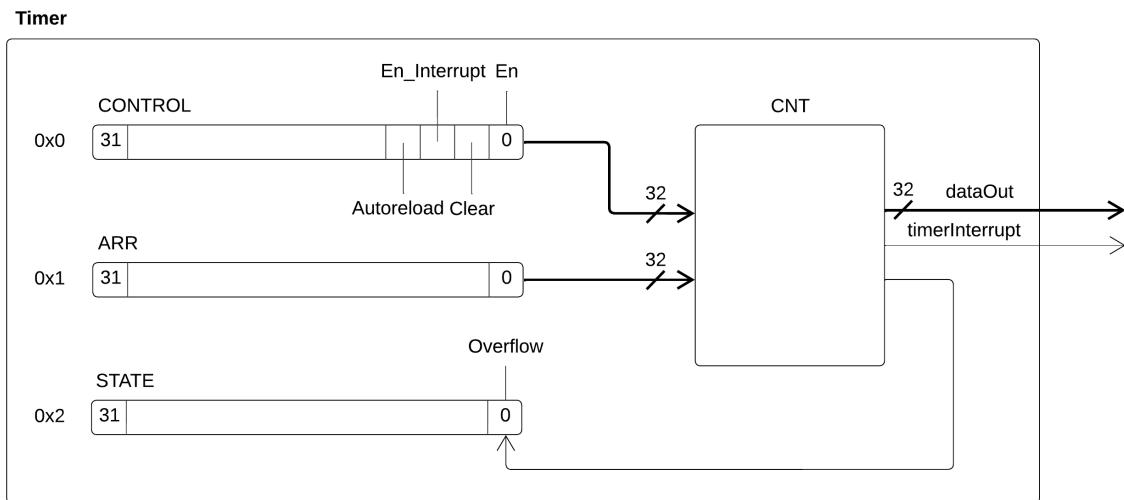


Fig. 3.38. Funcionamiento interno del *Timer*.

- CONTROL: Registro que controla el proceso CNT. El bit 0 indica el Enable de la cuenta, el bit 1 reinicia el conteo, el bit 2 activa la señal de *interrupt* al llegar al valor deseado y el bit 3 permite al *Timer* reiniciar la cuenta de forma automática una vez alcance el valor deseado.
- ARR: Registro que indica el valor a alcanzar.
- STATE: Registro que muestra el estado del *Timer*. El bit 0 indica el final de la cuenta.

Al disponer de un puerto de salida DatosOut, se puede obtener el valor actual de la cuenta

cargándolo a los registros (con una instrucción de carga indicando cualquier dirección de memoria entre 0x3000 y 0x3002).

3.4. Diseño completo

Los siguientes tres esquemáticos muestran el diseño final del núcleo con dos periféricos.

Importante destacar las señales de color rojo. Representan los 24 bits de la señal “microcode” generada por la unidad de control.

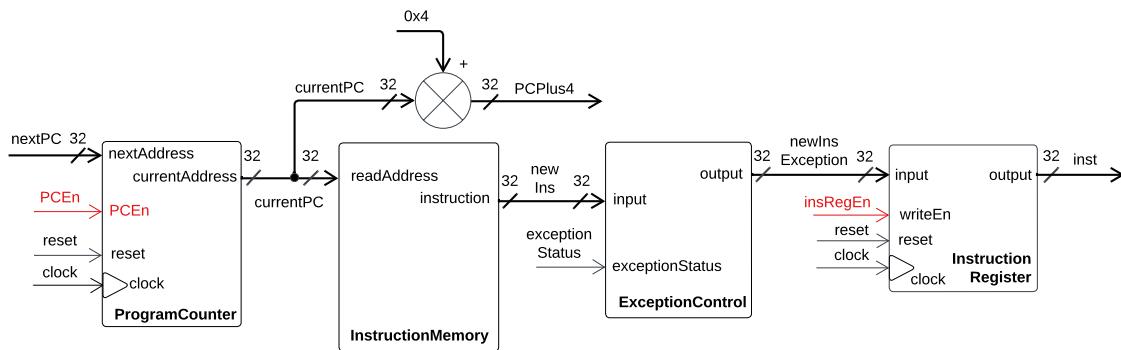


Fig. 3.39. Esquemático del diseño final (1). ProgramCounter, InstructionMemory, ExceptionControl y InstructionRegister.

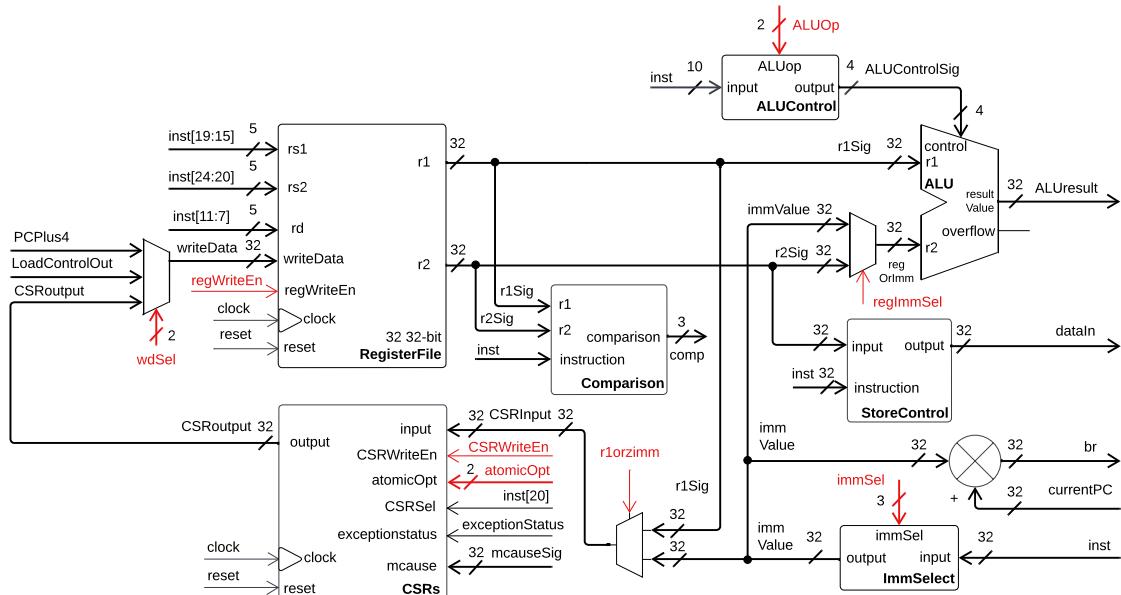


Fig. 3.40. Esquemático del diseño final (2). RegisterFile, CSRs, Comparison, ALUControl, ALU, StoreControl y ImmSelect.

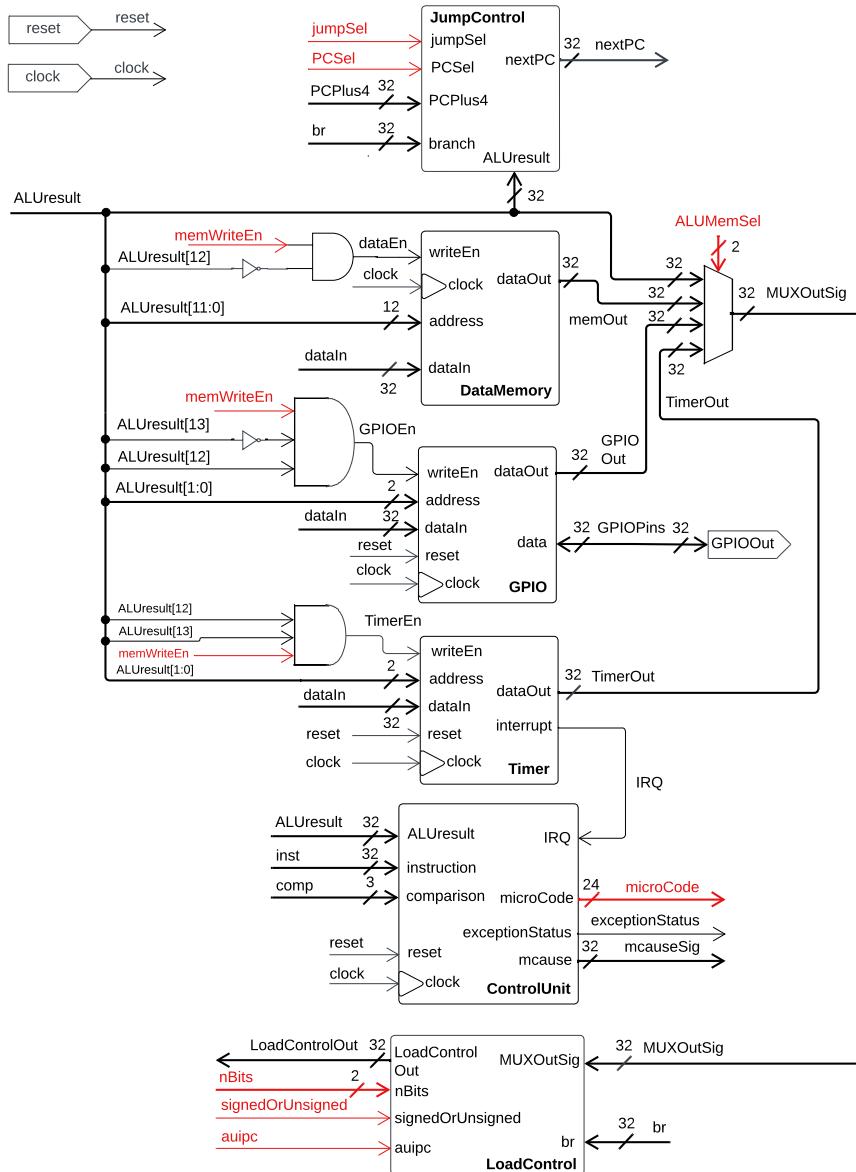


Fig. 3.41. Esquemático del diseño final (3). JumpControl, DataMemory, GPIO, Timer, ControlUnit y LoadControl.

4. DISEÑO DEL ENSAMBLADOR

Puesto que codificar a mano el código máquina que va a ejecutar el núcleo es un proceso lento y tedioso, se ha programado un ensamblador sencillo que permita al desarrollador escribir los programas a ejecutar en un lenguaje formal, no en código máquina.

Dicho ensamblador ha permitido elaborar programas más complejos, y así realizar las simulaciones necesarias para probar el núcleo.

4.1. Programa ensamblador y lenguaje ensamblador

Antes de explicar el funcionamiento del ensamblador creado, es importante hacer una distinción entre un programa ensamblador, y un lenguaje ensamblador.

- Programa ensamblador: Ejecutable capaz de leer un fichero fuente en lenguaje ensamblador y convertirlo a código máquina (conjunto de instrucciones escritas en binario), para que pueda ser leído por el núcleo.
- Lenguaje ensamblador: Lenguaje de programación específico para cada arquitectura de procesador, de muy bajo nivel (control directo sobre el hardware, generalmente poco legible).

Como se puede observar en la siguiente figura, el archivo en lenguaje ensamblador “archivo.s” se introduce al programa ensamblador creado “ensamblador_riscv”. Este programa genera el archivo en código máquina “archivo.mem”. Este archivo ya es legible por el procesador, el cual leerá dicho archivo línea por línea en el componente InstructionMemory.

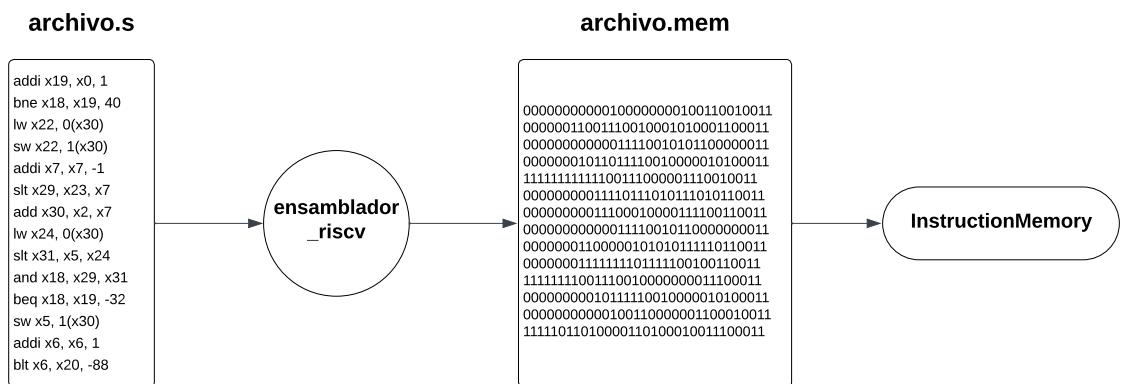


Fig. 4.1. Propósito del ensamblador.

4.2. Funcionamiento del ensamblador

El ensamblador implementado ha sido programado en el lenguaje C++. Dicho programa sigue el siguiente proceso:

1. Abre el archivo que se indica como único argumento del programa (archivo con instrucciones en lenguaje ensamblador), así como el archivo sobre el que se va a guardar el lenguaje máquina.
2. Lee línea a línea el archivo de entrada, y va guardando sus parámetros en un vector de una estructura.
3. Itera sobre el vector, encuentra la instrucción a generar entre todas las posibles (43 instrucciones en el vector de estructuras global “isa”), y genera la instrucción de 32 bits en base a sus argumentos (convirtiendo valores enteros decimales a binario mediante la función `signed_to_binary`).
4. Cierra ambos archivos y cierra el programa devolviendo 0.

En las siguientes secciones se va a tratar la compilación de los archivos “.cpp”, de modo que se obtenga el ensamblador como un ejecutable.

En el Anexo C se puede consultar el código del ensamblador (Makefile, archivos hpp y cpp).

4.3. Compilar y enlazar. Diferencias.

Pese a que comúnmente se dice que se compila un archivo C o C++ para obtener un ejecutable, no es del todo cierto. Cuando gcc o clang “compilan”, realmente están preprocesando, compilando, y posteriormente enlazando. Es importante distinguir dichos procesos.

1. Preprocesamiento: Prepara el código fuente para empezar la compilación. Incluye archivos de cabecera (aquellos que contienen declaraciones de funciones, variables e incluyen otros archivos cabecera), sustituye macros (valores asociados a texto) y elimina comentarios.
2. Compilación: Traduce el código fuente escrito en un lenguaje de alto nivel (C++ en este caso) a código objeto. El código objeto es un nivel de código intermedio no ejecutable, específico de la arquitectura en la que se genera.
3. Enlazado: Asegura que las llamadas a funciones y referencias de variables, se vinculan correctamente. Además, agrupa los archivos objeto con las librerías necesarias para formar el ejecutable.

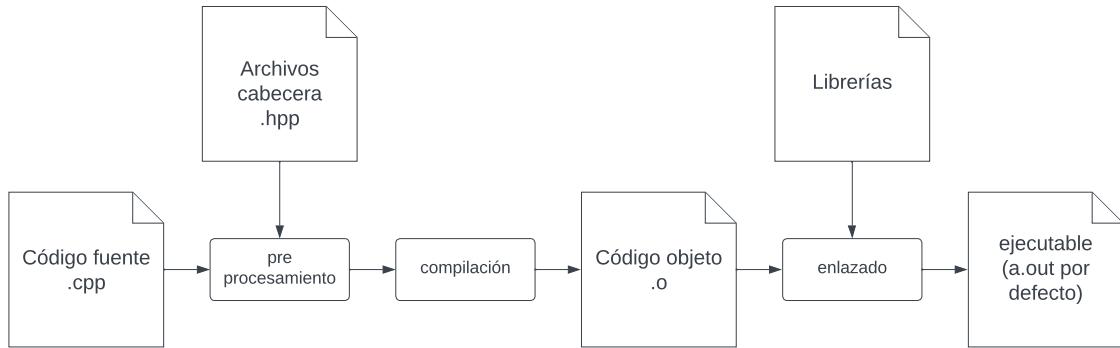


Fig. 4.2. Proceso de compilación en C y C++.

4.4. Makefile y compilación del ensamblador

Como se puede apreciar en la siguiente figura, se compila (y enlaza) el ensamblador mediante un sencillo Makefile, que hace uso del compilador clang++ para C++11.

```

+ ensamblador_riscv git:(main) ✘ make
clang++ -Wall -Wextra -Werror -g -std=c++11 -pedantic -c -o main.o main.cpp
clang++ -Wall -Wextra -Werror -g -std=c++11 -pedantic -c -o srcs/generate_mc.o srcs/generate_mc.cpp
clang++ -Wall -Wextra -Werror -g -std=c++11 -pedantic -c -o srcs/open_files.o srcs/open_files.cpp
clang++ -Wall -Wextra -Werror -g -std=c++11 -pedantic -c -o srcs/signed_to_binary.o srcs/signed_to_binary.cpp
clang++ -Wall -Wextra -Werror -g -std=c++11 -pedantic -g -o ensamblador_riscv main.o ./srcs/generate_mc.o ./srcs/open_files.o ./srcs/signed_to_binary.o

```

Fig. 4.3. Compilando el ensamblador con make.

Se han utilizado los siguientes *flags* de compilación (opciones para personalizar el proceso de compilación):

- -Wall, -Werror, -Wextra y -pedantic mejoran la seguridad y calidad del código fuente mostrando más *warnings* (advertencias), y convirtiéndolas en errores para no finalizar la compilación en caso de haberlos.
- -g añade información al ejecutable, útil para su depuración con GDB, LLDB u otros programas compatibles.
- -c únicamente compila (no enlaza) los archivos “.cpp”, generando archivos objeto “.o”.
- -o especifica el nombre del archivo de salida al compilar o enlazar. Al usar gcc o clang++, el nombre por defecto del ejecutable es “a.out”.

El Makefile en cuestión compila los archivos “.cpp” generando los archivos objeto “.o”, para posteriormente enlazarlos creando el ejecutable “ensamblador_riscv”.

4.5. Utilizando el ensamblador

Así mismo, en la figura 4.4. se puede ver como ejecutando el programa “ensamblador_riscv” con “Ejemplo_3.s” como argumento, se genera el archivo en lenguaje máquina “Ejemplo_3.mem”. Éste ultimo es el que lee el microprocesador línea por línea en el componente InstructionMemory.

```
→ ensamblador_riscv git:(main) ✘ ./ensamblador_riscv tests/asm_files/Ejemplo_3.s
→ ensamblador_riscv git:(main) ✘ cat tests/asm_files/Ejemplo_3.s
addi x18, x0, 80
addi x19, x0, 100
addi x18, x18, 1
bne x18, x19, -4
addi x7, x0, 255
lui x20, 1
sb x7, 2(x20)
sb x18, 0(x20)
lb x8, 0(x20)%
→ ensamblador_riscv git:(main) ✘ cat tests/mc_files/Ejemplo_3.mem
00000101000000000000100100010011
00000110010000000000100110010011
000000000001100100001000100010011
11111111001110010001111011100011
00001111111000000000001110010011
000000000000000000001101000110111
000000000111101000000000100100011
00000001000101000000000000100011
000000000000000000001010000000011
```

Fig. 4.4. Haciendo uso del ensamblador.

En los próximos apartados se utilizará el ensamblador para generar los archivos “.mem” necesarios para las pruebas del núcleo.

5. PRUEBAS BÁSICAS DEL NÚCLEO

Este apartado va a demostrar el correcto funcionamiento del núcleo en simulación, desde el uso básico de las instrucciones y el guardado de datos a periféricos, hasta el manejo de excepciones.

En un principio no es necesario hacerle muchas pruebas al ensamblador, puesto que si el microprocesador actúa de forma correcta durante las pruebas del mismo, lo más probable es que esté recibiendo las instrucciones codificadas correctamente en código máquina.

5.1. Ejemplo_1.s: Operaciones lógicas y shifts

En este ejemplo se muestra la funcionalidad de “addi” para guardar valores en registros, así como varias instrucciones de tipo R para hacer operaciones.

```
1 addi x2, x0, 256      # Guarda el valor 256 en el stack pointer (x2)
2 addi x6, x0, -50       # Guarda el valor -50 en el registro temporal x6
3 addi x7, x0, 102       # Guarda el valor 102 en el registro temporal x7
4 add x18, x6, x7        # -50 + 102 = 52
5 sub x19, x6, x7        # -50 - 102 = -152
6 sub x20, x7, x6        # 102 - (-50) = 152
7 and x21, x6, x7        # -50 and 102 = 70
8 or x22, x6, x7         # -50 or 102 = -18
9 srl x23, x6, x7        # -50 >> 6 = 67108863
10 slli x24, x6, 12        # -50 << 12 = -204800
11 xor x25, x6, x7        # -50 xor 102 = -88
12 sw x18, 0(x2)          # Guardar en memoria los 8 valores desde el sp
13 sw x19, 1(x2)
14 sw x20, 2(x2)
15 sw x21, 3(x2)
16 sw x22, 4(x2)
17 sw x23, 5(x2)
18 sw x24, 6(x2)
19 sw x25, 7(x2)
```

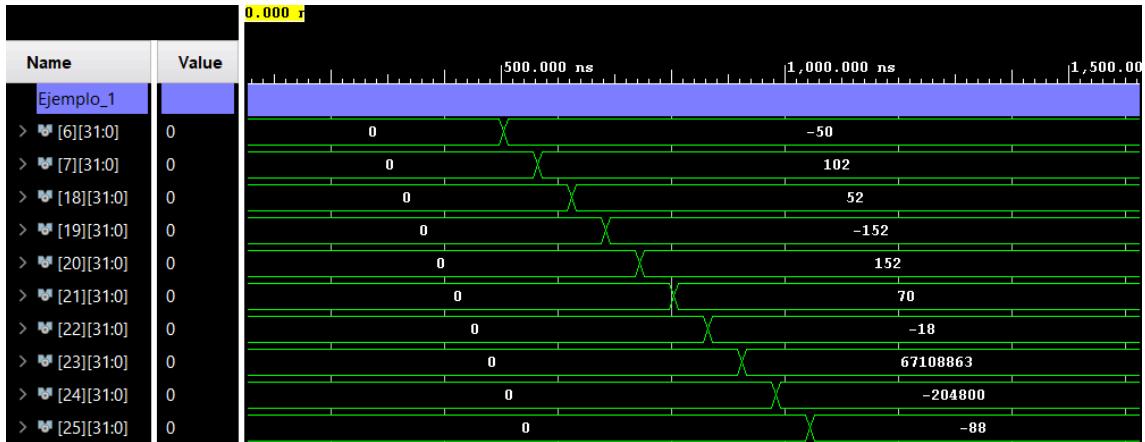


Fig. 5.1. Simulación Ejemplo 1.

1. Se guarda el valor -50 en el registro x6, y 102 en el registro x7.
2. Se suman los valores en dichos registros, y se guarda el resultado en x18 ($-50 + 102 = 52$).
3. Se resta x6 sobre x7, y se guarda el resultado en x19 ($-50 - 102 = -152$).
4. Se resta x7 sobre x6, y se guarda el resultado en x20 ($102 - (-50) = 152$).
5. Se realiza una operación lógica “and” entre los bits de x6 y x7, y se guarda el resultado en x21 ($-50 \text{ and } 102 = 70$).
6. Se realiza una operación lógica “or” entre los bits de x6 y x7, y se guarda el resultado en x22 ($-50 \text{ or } 102 = -18$).
7. Se realiza un desplazamiento lógico a la derecha del valor en x6 de tantos *bits* como indiquen los 5 LSBs del valor en x7, y se guarda el resultado en x23. El valor decimal 102 es “1100110” en binario, por lo que sus 5 LSBs son “00110” (6 en decimal). Se desplaza por lo tanto 6 *bits* a la derecha ($-50 >> 6 = 67108863$).
8. Se realiza un desplazamiento lógico a la izquierda del valor en x6 de 12 *bits*, y se guarda el resultado en x24 ($-50 << 12 = -204800$).
9. Se realiza una operación lógica “xor” entre los bits de x6 y x7, y se guarda el resultado en x25 ($-50 \text{ xor } 102 = -88$).

En la figura anterior se pueden apreciar los registros x6, x7 y x18-x25. Se puede observar cómo los registros x18 a x25 van adquiriendo los valores de las operaciones lógicas y aritméticas descritas.

5.2. Ejemplo_2.s: sb, lw y comparación slt

En este ejemplo se comprueba la carga y guardado de valores entre la pila de registros y la memoria de datos. También se hace uso de una simple comparación mediante la instrucción “slt”.

```

1 addi x2, x0, 256      # Guarda el valor 256 en el stack pointer
2 addi x6, x0, 324      # Guarda el valor 324 en el registro x6
3 addi x7, x6, -400     # 324 - 400 = -76
4 sb x6, 0(x2)          # Guarda los 8 LSBs de x6 en la dir. sp
5 sb x7, 1(x2)          # Guarda los 8 LSBs de x7 en la dir. sp + 1
6 lw x6, 0(x2)           # Carga el valor en la dirección sp sobre x6
7 lw x7, 1(x2)           # Carga el valor en la dir. sp + 1 sobre x7
8 slt x18, x6, x7       # x18 = (68 < 180) ? 1 : 0

```

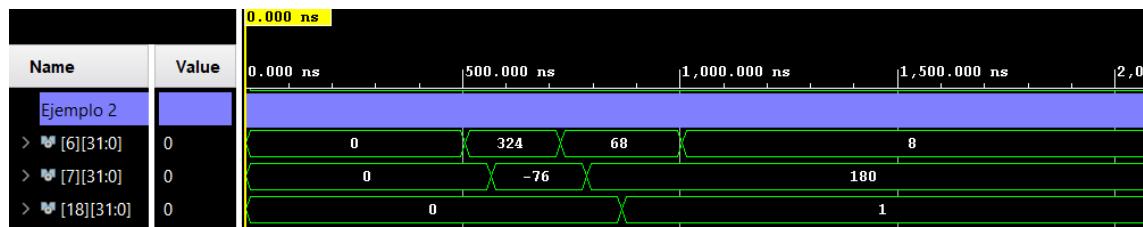


Fig. 5.2. Simulación Ejemplo 2.

1. Se guarda el valor 324 en el registro x6.
2. Se suma el valor -400 al valor en x6 y se guarda el resultado en x7 ($324 + (-400) = -76$).
3. Se guardan los 8 LSBs de x6 en el puntero a la pila (dirección 256_{10} guardada en el registro x2). El valor en x6 es 324_{10} , “101000100” en binario. Se guarda en memoria “1000100”, 68_{10} .
4. Se guardan los 8 LSBs de x7 en la siguiente dirección de memoria. El valor en x7 es -76_{10} , FFFFFFFB4 en hexadecimal. Se guarda en memoria “10110100”, 180_{10} .
5. Se cargan en los registros x6 y x7 los valores guardados en la pila (68 y 180 respectivamente).
6. Se comprueba si el valor en x6 (68) es menor que el valor en x7 (180), al ser cierto, se carga un 1 en x18.

En la figura anterior se muestran los registros x6, x7 y x18. Se comprueba la carga de los 8 LSBs y la comparación final entre x6 y x7.

5.3. Ejemplo_3.s: Bucle con iterador. Carga a GPIOs.

El objetivo de este ejemplo es mostrar la utilidad de las instrucciones de tipo B para hacer bucles. Se incrementará un valor tras cada iteración del bucle, hasta que se cumpla una cierta condición.

En este ejemplo se muestra también la funcionalidad del GPIO, cargando un valor en sus 8 LSBs.

```

1 addi x18, x0, 80      # Guarda el valor 80 en el registro x18
2 addi x19, x0, 100     # Guarda el valor 100 en el registro x19
3 addi x18, x18, 1       # x18 = x18 + 1
4 bne x18, x19, -4      # x18 == x19 ?
5 addi x7, x0, 255      # Guarda el valor 255 en el registro x7
6 lui x20, 1              # Dirección del GPIO
7 sb x7, 2(x20)          # Los 8 LSBs de TRI del GPIO a '1'
8 sb x18, 0(x20)          # Guarda el valor final de x18 en los 8 LSBs del
                           # GPIO
9 lb x8, 0(x20)          # Carga en x8 los 8 LSBs de datosOut del GPIO

```

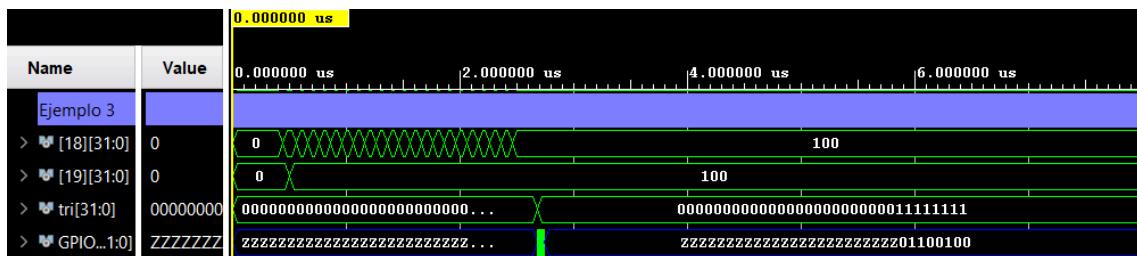


Fig. 5.3. Simulación Ejemplo 3.

1. Se guardan los valores 80 y 100 en los registros x18 y x19 respectivamente.
2. Se incrementa en 1 el valor en x18.
3. Se comprueba si el valor en x18 es distinto al valor en x19. Si lo es, se retrocede una instrucción, incrementando de nuevo el valor en x18. En caso contrario, se avanza a la siguiente instrucción en memoria.
4. Se guarda el valor 255 (“0b1111111”) en el registro x7.
5. Se guarda el valor 0x1000 en el registro x20. Este valor indica la dirección en memoria del GPIO.
6. Utilizando la dirección del GPIO, se guarda el valor del registro x7 en el registro TRI del GPIO (los 8 LSBs a ‘1’).
7. Utilizando la dirección del GPIO, se guarda el valor final de x18 (tras incrementar su valor hasta 100) en el registro datosOut del GPIO.

8. Se muestra también la carga de datos desde el registro datosOut del GPIO, demostrando que el valor guardado en dicho registro es de 100_{10} .

En la figura anterior se muestran los registros x18 y x19, así como TRI y datosOut del GPIO. Se puede observar como x18 cambia de valor hasta alcanzar 100_{10} . Así mismo, se observa como se han configurado los 8 LSBs del GPIO como salida (*bits* a '1') y como se ha guardado el valor 100_{10} en binario "01100100", al GPIO.

5.4. Ejemplo_4.s: CSRs

En este ejemplo se muestra el correcto funcionamiento de las instrucciones de memoria atómica para modificar *bits* en los CSRs.

```

1 addi x6, x0, 1      # Guarda el valor 1 en el registro x6
2 csrrw x18, 0, x6    # Carga al csr 0 el valor en x6, y viceversa
3 csrrsi x19, 0, 4     # Activa el bit 2 del csr 0
4 csrrsi x20, 0, 16    # Activa el bit 4 del csr 0
5 csrrci x21, 0, 16    # Limpia el bit 4 del csr 0
6 csrrci x22, 0, 4     # Limpia el bit 2 del csr 0
7 csrrci x23, 0, 1     # Limpia el bit 0 del csr 0

```

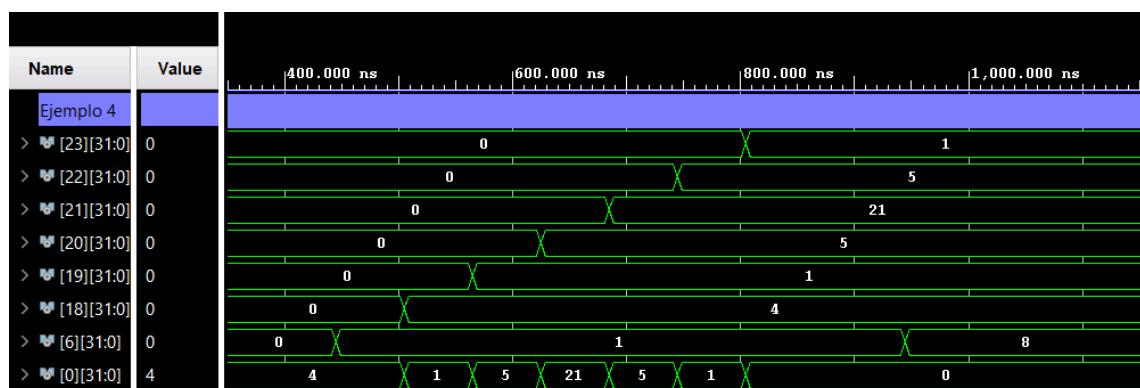


Fig. 5.4. Simulación Ejemplo 4.

1. Se guarda el valor 1 en el registro x6.
2. Se carga al csr 0 el valor en x6, y el valor del csr 0 en el registro x6.
3. Se activa el *bit* 2 del csr 0.
4. Se activa el *bit* 4 del csr 0.
5. Se pone a '0' el *bit* 4, después el 2, y finalmente el 0.

En la anterior figura se puede observar como la última señal (correspondiente al csr 0) empieza en 1 ("0b1"), cambia a 5 ("0b101"), y después a 21 ("0b10101"). Después de

esto retorna a los valores anteriores a medida que se limpian los *bits* 4, 2 y 0.

NOTA: Importante recalcar el valor inicial del csr 0 en 4 al actuar este como registro mtvec, el cual indica la dirección de memoria del *exception handler*.

5.5. Ejemplo_5.s: Stores y loads

Se muestran las diversas instrucciones de carga y guardado dependiendo del número de *bits* a manejar.

```

1 addi x2, x0, 256      # Guarda el valor 256 en el stack pointer
2 addi x18, x0, -1000 # Guarda el valor -1000 en el registro x18
3 sw x18, 0(x2)        # Guarda el valor del registro x18 en el stack
4 lw x19, 0(x2)        # Carga los 32 bits del stack al registro x19
5 sh x19, 1(x2)        # Guarda los 16 LSBs de x19 en el stack
6 lh x20, 1(x2)        # Carga los 16 LSBs del stack al registro x20
7 sb x20, 2(x2)        # Guarda los 8 LSBs de x20 en el stack
8 lb x21, 2(x2)        # Carga los 8 LSBs del stack al registro x21

```

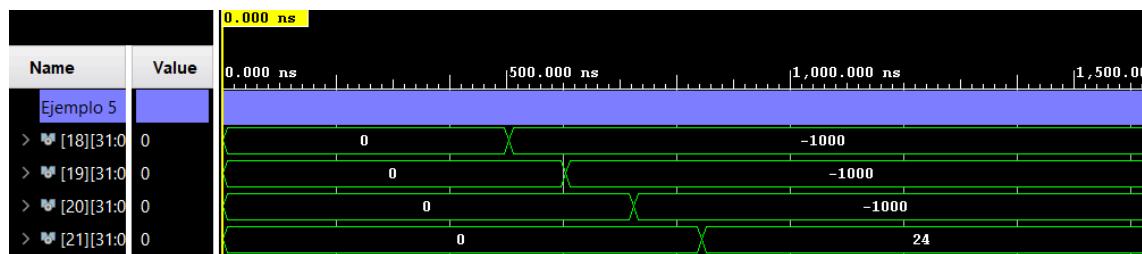


Fig. 5.5. Simulación Ejemplo 5.

1. Se guarda el valor -1000 en el registro x18.
2. Se guarda el valor de dicho registro en el puntero a la pila.
3. Se carga dicho valor de la pila al registro x19.
4. Se guardan los 16 LSBs de x19 en la siguiente dirección de memoria de la pila.
5. Se cargan los 16 LSBs del valor guardado en la pila sobre el registro x20.
6. Se guardan los 8 LSBs de x20 en la siguiente dirección de memoria de la pila.
7. Se cargan los 8 LSBs del valor guardado en la pila sobre el registro x21.

Se observa en los registros x18 a x21 de la anterior figura como el valor -1000 se conserva con los 16 LSBs de -1000, pero no con los 8 LSBs (valor 24_{10}).

5.6. Ejemplo_6.s: Probando el Timer

En este ejemplo se hace un uso básico del *Timer*, indicándole un valor a alcanzar, y comprobando que se produce una interrupción. Esta interrupción debe ser manejada por el procesador, indicando su motivo en el CSR mcause, y avanzando a la siguiente instrucción tras hacer uso del *exception handler*.

```
1 lui x18, 3      # Direccion del Timer
2 lui x19, 24     # Valor a alcanzar = 0x18000
3 sw x19, 1(x18) # Guarda el valor en registro ARR del Timer
4 addi x6, x0, 5  # En y En_interrupt
5 sb x6, 0(x18)  # Inicio de la cuenta
6 beq x0, x0, 0   # Bucle para detectar el final de la cuenta
```

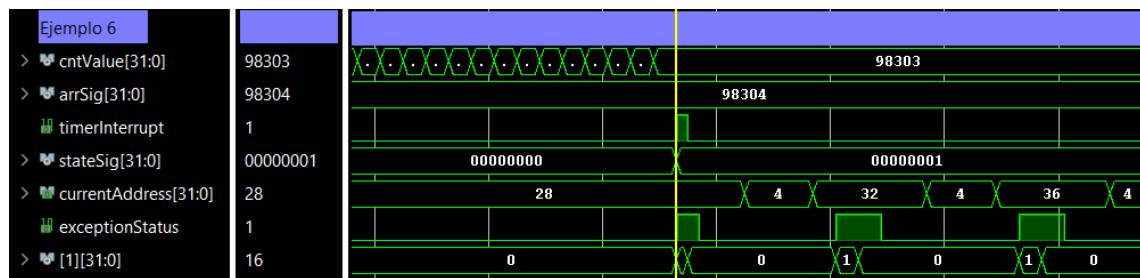


Fig. 5.6. Simulación Ejemplo 6.

En la imagen anterior se puede apreciar cómo aumenta el valor de la cuenta del *Timer* cnt-Value hasta ARR - 1 (comprueba si cntValue es igual a arrSig para la próxima iteración, y si se cumple y En_interrupt se encuentra activo, timerInterrupt se activa).

También se puede observar que al producirse la interrupción, currentAddress (PC) salta al *exception handler* situado en la dirección 0x4, y de ahí salta a la próxima instrucción a ejecutar. A continuación, el núcleo entra en un bucle de excepciones al acceder a direcciones de memoria con instrucciones no válidas. “exceptionStatus” se activa de forma periódica.

6. EJEMPLOS PRÁCTICOS

Con el objetivo de mostrar la aplicación práctica del microprocesador completo para ejecutar una tarea, se van a mostrar dos ejemplos:

- Ordenación de diez números enteros de forma ascendente utilizando tres algoritmos distintos: *Bubble sort* (ordenamiento de burbuja), *Insertion sort* (ordenamiento por inserción) y *Selection sort* (ordenamiento por selección).
- Demostración del funcionamiento de entrada y salida de los GPIOs con un sencillo programa, “FPGA_0.mem”.

6.1. Ejemplo práctico 1: Ordenación de números enteros

En este primer ejemplo, se ejecutarán primero los tres algoritmos de ordenación (*Bubble sort*, *Insertion sort* y *Selection sort*) y se comprobará en simulación la correcta ordenación de los mismos. Se calculará el tiempo requerido para cada uno gracias al *Timer*. Los valores se compararán y de esta forma, se podrá concluir cuál es el más eficiente en el núcleo, para un número reducido de elementos.

Posteriormente, se extenderá el programa de uno de los algoritmos (aquel que haya resultado más rápido) con el objetivo de ejecutarlo en el diseño implementado en la FPGA. Se utilizarán los GPIOs para mostrar los números ordenados en los leds de la placa.

6.1.1. Comparación de tres algoritmos de ordenación en simulación.

El arreglo de los números a ordenar es el siguiente:

$$\{6, -2000, -2, 42, 3, -221, 7, 12, 9, 33\}$$

Se guardan en la memoria de datos de forma contigua a partir del *stack pointer* (dirección de memoria de la pila, valor cargado en x2).

```
1 addi x2, x0, 256    # Set stack pointer
2
3 addi x5, x0, 6
4 sw x5, 0(x2)
5 addi x5, x0, -2000
6 sw x5, 1(x2)
7 addi x5, x0, -2
8 sw x5, 2(x2)
9 addi x5, x0, 42
```

```

10 sw x5, 3(x2)
11 addi x5, x0, 3
12 sw x5, 4(x2)
13 addi x5, x0, -221
14 sw x5, 5(x2)
15 addi x5, x0, 7
16 sw x5, 6(x2)
17 addi x5, x0, 12
18 sw x5, 7(x2)
19 addi x5, x0, 9
20 sw x5, 8(x2)
21 addi x5, x0, 33
22 sw x5, 9(x2)

```

Posteriormente, se configura el *Timer* indicando el valor a alcanzar en la cuenta (ARR = 0x1000). En este caso, puesto que no se desea alcanzar un valor concreto, es necesario introducir un valor elevado. También se debe activar el *bit* de *Enable* en el registro de control del *Timer*, así como el *Enable* de *Interrupt* si se desea producir una interrupción cuando se alcance el valor.

```

1 lui x18, 3           # Dirección del Timer
2 lui x19, 1           # Valor a alcanzar en la cuenta
3 sw x19, 1(x18)      # Guardar el valor en el Timer
4 addi x6, x0, 5       # Ency En_Interrupt
5 sb x6, 0(x18)        # Inicio de la cuenta al cargar Enables

```

Una vez guardado el arreglo en memoria y habiendo iniciado la cuenta el *Timer*, se procede a implementar cada uno de los algoritmos. Se muestra cada uno en lenguaje C para su mejor lectura y, posteriormente, en lenguaje ensamblador:

- **Bubble sort:** Compara repetidamente pares de elementos adyacentes en el arreglo y los intercambia si están en el orden incorrecto.

BubbleSort.c:

```

1 int main()
2 {
3     int arr[10] = {6, -2000, -2, 42, 3, -221, 7, 12, 9, 33};
4
5     for (int i = 0; i < 9; i++)
6     {
7         for(int j = 0; j < 9 - i; j++)
8         {
9             if (arr[j + 1] < arr[j])
10            {
11                 int temp = arr[j];
12                 arr[j] = arr[j + 1];
13                 arr[j + 1] = temp;

```

```

14 }
15 }
16 }
17 }

```

BubbleSort.s:

```

1 addi x29, x0, 9      # n - 1 = 9
2
3 addi x5, x0, 0        # i = 0
4 bge x5, x29, 68      # i < 9 ?
5 addi x6, x0, 0        # j = 0
6 sub x31, x29, x5      # 9 - i
7 bge x6, x31, 48      # j >= 9 - i ?
8 add x7, x2, x6        # x7 = sp + j
9 lw x18, 0(x7)         # arr[j]
10 lw x19, 1(x7)        # arr[j + 1]
11 bge x19, x18, 20      # arr[j + 1] >= arr[j] ?
12 add x28, x0, x18      # temp = arr[j]
13 add x18, x0, x19      # arr[j] = arr[j + 1]
14 sw x18, 0(x7)         # store arr[j]
15 sw x28, 1(x7)         # store temp in arr[j + 1]
16 addi x6, x6, 1        # j++
17 sub x30, x29, x5      # 9 - i
18 blt x6, x30, -40      # j < 9 - i ?
19 addi x5, x5, 1        # i++
20 blt x5, x29, -60      # i < 9 ?

```

- **Insertion sort:** Recorre el arreglo insertando en cada iteración un elemento en la posición correcta, dentro de la parte ya ordenada de la lista.

InsertionSort.c:

```

1 int main()
2 {
3     int arr[10] = {6, -2000, -2, 42, 3, -221, 7, 12, 9, 33};
4
5     int i, key, j;
6
7     for (int i = 1; i < 10; i++)
8     {
9         key = arr[i];
10        j = i - 1;
11        while (j > -1 && arr[j] > key)
12        {
13            arr[j + 1] = arr[j];
14            j = j - 1;
15        }
16        arr[j + 1] = key;

```

```

17 }
18 }
```

InsertionSort.s:

```

1 addi x20, x0, 10      # n = 10
2
3 addi x6, x0, 1        # i = 1
4 bge x6, x20, 96       # i < 10 ?
5 add x28, x2, x6       # dir(arr[i]) = sp + i
6 lw x5, 0(x28)         # key = arr[i]
7 addi x7, x6, -1       # j = i - 1
8 addi x23, x0, -1      #
9 slt x29, x23, x7     # -1 < j ?
10 add x30, x2, x7      # dir(arr[j]) = sp + j
11 lw x24, 0(x30)       # arr[j]
12 slt x31, x5, x24     # key < arr[j] ?
13 and x18, x29, x31    # j > -1 && arr[j] > key
14 addi x19, x0, 1       #
15 bne x18, x19, 40      # j > -1 && arr[j] > key ?
16 lw x22, 0(x30)       # arr[j]
17 sw x22, 1(x30)       # arr[j + 1] = arr[j]
18 addi x7, x7, -1       # j = j - 1
19 slt x29, x23, x7     # -1 < j ?
20 add x30, x2, x7      # dir(arr[j]) = sp + j
21 lw x24, 0(x30)       # arr[j]
22 slt x31, x5, x24     # key < arr[j] ?
23 and x18, x29, x31    # j > -1 && arr[j] > key
24 beq x18, x19, -32     # j > -1 && arr[j] > key ?
25 sw x5, 1(x30)        # arr[j + 1] = key
26 addi x6, x6, 1        # i++
27 blt x6, x20, -88      # i < 10 ?
```

- **Selection sort:** Busca iterativamente el menor elemento en el arreglo no ordenado y lo coloca en la posición correcta, reduciendo la parte no ordenada tras cada iteración.

SelectionSort.c:

```

1 int main()
2 {
3     int arr[10] = {6, -2000, -2, 42, 3, -221, 7, 12, 9, 33};
4
5     for (int i = 0; i < 9; i++)
6     {
7         int minIndex = i;
8
9         for (int j = i + 1; j < 10; j++)
10            if (arr[j] < arr[minIndex])
11                minIndex = j;
```

```

12         if (minIndex != i)
13     {
14         int temp = arr[i];
15         arr[i] = arr[minIndex];
16         arr[minIndex] = temp;
17     }
18 }
19 }
```

SelectionSort.s:

```

1 addi x18, x0, 10      # n = 10
2 addi x19, x0, 9       # n - 1 = 9
3
4 addi x5, x0, 0        # i = 0
5 bge x5, x19, 88       # i >= 9 ?
6 addi x6, x5, 0        # minIndex = i
7 addi x7, x5, 1        # j = i + 1
8 bge x7, x18, 36       # j >= 10 ?
9 add x20, x2, x7       # dir(arr[j]) = sp + j
10 lw x21, 0(x20)        # arr[j]
11 add x22, x2, x6       # dir(arr[minIndex]) = sp + minIndex
12 lw x23, 0(x22)        # arr[minIndex]
13 bge x21, x23, 8       # arr[j] >= arr[minIndex] ?
14 addi x6, x0, x7       # minIndex = j
15 addi x7, x7, 1        # j++
16 blt x7, x18, -28      # j < 10 ?
17 beq x6, x5, 32        # minIndex == i ?
18 add x28, x2, x5       # dir(arr[i]) = sp + i
19 lw x24, 0(x28)        # arr[i]
20 addi x29, x24, 0       # temp = arr[i]
21 add x30, x2, x6       # dir(arr[minIndex]) = sp + minIndex
22 lw x25, 0(x30)        # arr[minIndex]
23 sw x25, 0(x28)        # arr[i] = arr[minIndex]
24 sw x29, 0(x30)        # arr[minIndex] = temp
25 addi x5, x5, 1        # i++
26 blt x5, x19, -80      # i < 9 ?
```

Es importante destacar que estos algoritmos no buscan la eficiencia del ordenamiento, puesto que disponen de un orden de complejidad algorítmica cuadrático (el tiempo de ejecución aumenta proporcionalmente al cuadrado del número de elementos a la entrada, $O(n^2)$) por lo que se han utilizado en este proyecto por su sencillez a la hora de programarlos en lenguaje ensamblador.

Una vez ejecutados los tres algoritmos por separado, se cargan los números ordenados en los registros x18 a x27 y el valor final de la cuenta del *Timer* en x17.

```
| 1 lui x18, 3           # Dirección del Timer
```

```

2 lw x17, 0(x18)      # Carga del valor cuenta
3
4 lw x18, 0(x2)
5 lw x19, 1(x2)
6 lw x20, 2(x2)
7 lw x21, 3(x2)
8 lw x22, 4(x2)
9 lw x23, 5(x2)
10 lw x24, 6(x2)
11 lw x25, 7(x2)
12 lw x26, 8(x2)
13 lw x27, 9(x2)

```

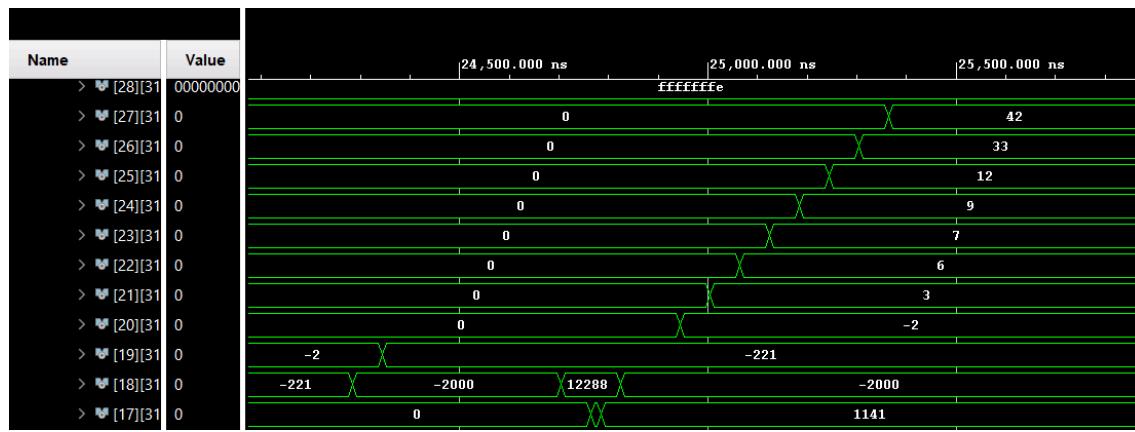


Fig. 6.1. Resultados de la simulación para BubbleSort.mem.

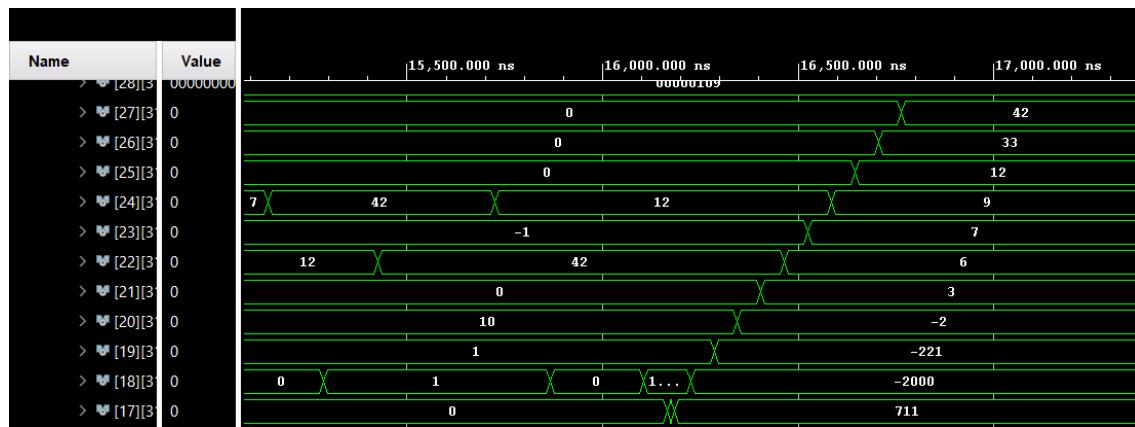


Fig. 6.2. Resultados de la simulación para InsertionSort.mem.

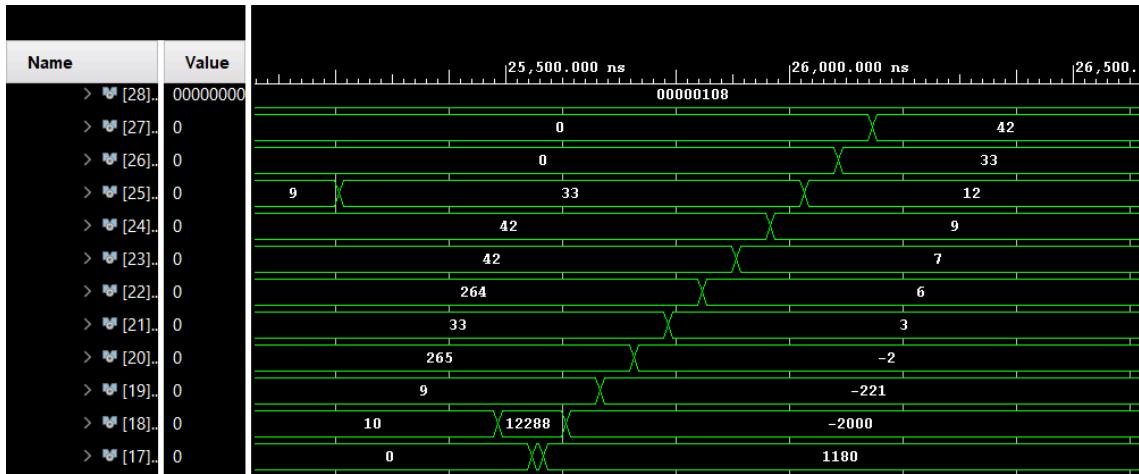


Fig. 6.3. Resultados de la simulación para SelectionSort.mem.

Se confirma la correcta ordenación de los números, y como se puede observar en el registro x17, el *Timer* indica una cuenta de 1.141 para *Bubble Sort*, 711 para *Insertion Sort* y 1.180 para *Selection Sort*.

Para obtener el tiempo real que ha tardado cada uno de ellos, se aplica la siguiente fórmula:

$$tiempo(s) = \frac{\text{valor final de la cuenta del } Timer}{\text{frecuencia de trabajo (Hz)}} \quad (6.1)$$

La frecuencia del reloj que se ha utilizado en simulación es de 45 MHz (la misma a la que trabaja el núcleo). Se obtienen los siguientes tiempos:

TABLA 6.1. COMPARACIÓN DEL TIEMPO REQUERIDO PARA LOS TRES ALGORITMOS.

	Bubble sort	Insertion sort	Selection sort
Valor del Timer	1.141	711	1.180
Tiempo real (us)	25,36	15,80	26,22

Por lo tanto el algoritmo **insertion sort** ha sido el más eficiente, con un tiempo de ejecución de 15,8 us.

6.1.2. Implementación del núcleo en la FPGA y uso de GPIOs.

En este apartado se va a implementar el diseño del núcleo y periféricos en la propia FPGA, con el objetivo de ejecutar el programa “InsertionSort.mem” (puesto que el algoritmo de ordenación por inserción ha sido el más rápido a la hora de ordenar los diez números).

Este programa, explicado en el anterior apartado, se va a extender de modo que tras la ordenación de los números haga uso de los GPIOs, para mostrar dichos números correctamente ordenados en los 16 leds de la FPGA en binario, a intervalos de dos segundos cada uno.

El objetivo es por tanto configurar el *Timer* de modo que tarde dos segundos en contar hasta cierto número. Este proceso se repetirá cada vez que se visualice un número en la placa. Mediante la expresión anterior (6.1), se puede despejar el valor final de la cuenta del *Timer*. Este valor depende de la frecuencia de trabajo.

Para obtener esta frecuencia, se parte de la placa de desarrollo FPGA empleada, la Digilent Basys 3, que dispone de una frecuencia de reloj de 100 MHz.

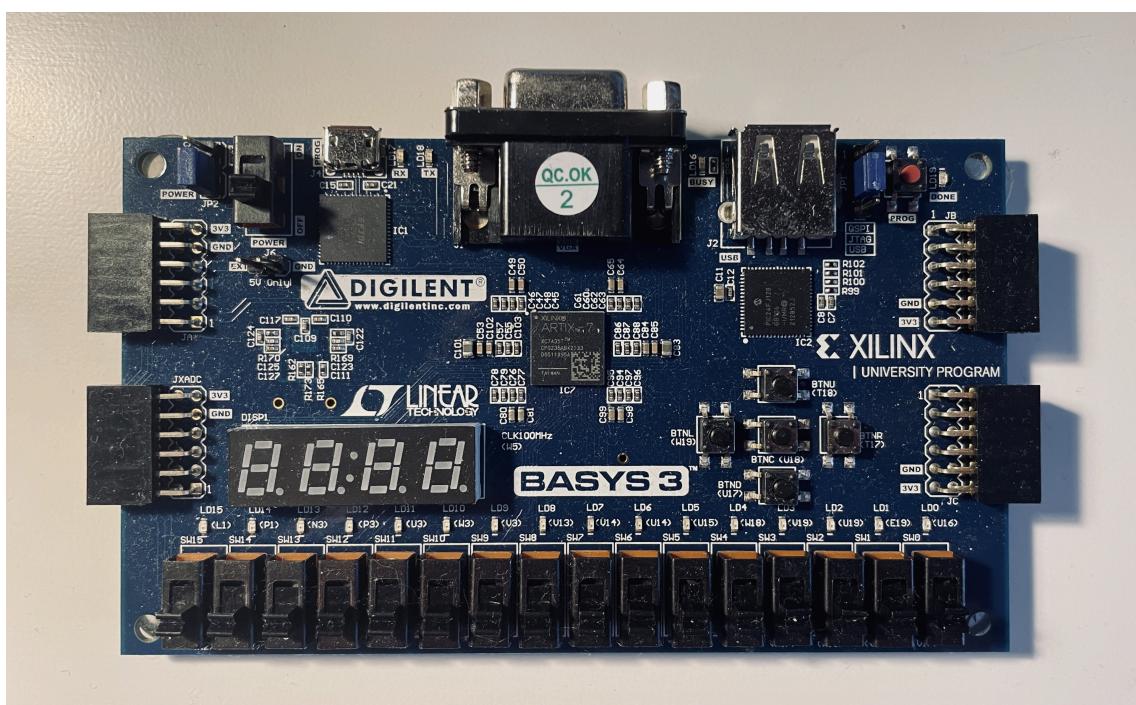


Fig. 6.4. Digilent Basys 3 Artix-7 FPGA Trainer Board

Con el objetivo de reducir el *negative slack* (holgura negativa) en la implementación en la FPGA, se ha tenido que reducir la frecuencia del reloj a 45 MHz mediante un bloque IP (*Intellectual Property*, propiedad intelectual) de la herramienta *Clocking Wizard* 6.0 de Vivado.

Un bloque IP es un componente reutilizable y predefinido por Vivado utilizado para realizar una tarea específica de forma más sencilla. En este caso, la herramienta *Clocking Wizard* ha permitido modificar la frecuencia del reloj que genera el bloque IP, para ser ésta la que se utilice en el proyecto.

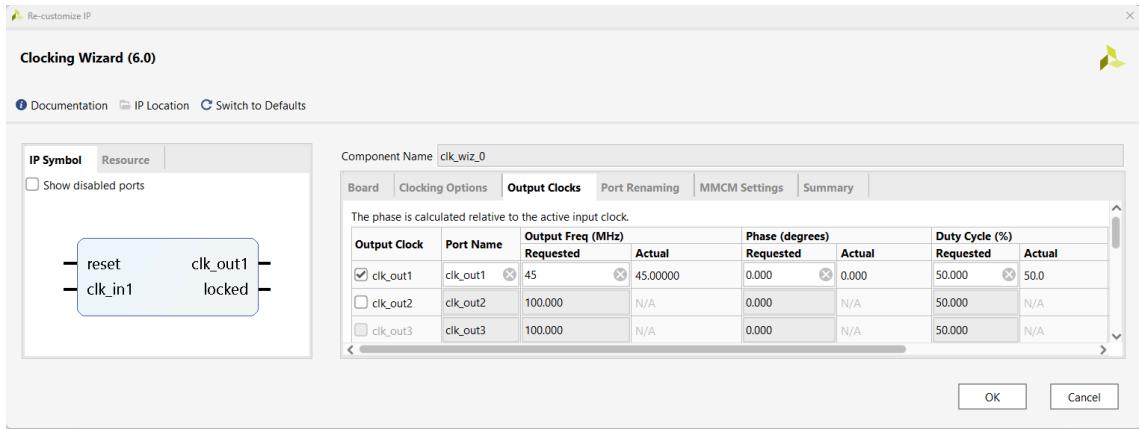


Fig. 6.5. Clocking Wizard 6.0 en Vivado.

Por lo tanto, dada una frecuencia de trabajo de 45 MHz y empleando la expresión (6.1), se obtiene el valor a cargar en el registro ARR del *Timer* para alcanzar dos segundos:

$$ARR = t(s) * f(Hz) = 2 * 45 * 10^6 = 90 * 10^6$$

Para cargar este valor, puesto que es un número elevado, se va a hacer uso de la instrucción “lui”.

El valor a cargar ($90 * 10^6$) es 0x055D4A80 en hexadecimal, y puesto que “lui” carga valores con los 12 LSBs en cero, podemos definir el valor 0x055D4A80 a alcanzar como la siguiente suma:

$$0x055D4000 + 0xA80$$

Esto mismo equivale a una instrucción “lui” de 0x55D4 (21972_{10}), y una instrucción “addi” de 2688_{10} sobre el mismo registro. El valor final de la suma es el valor a cargar en el registro ARR del *Timer*.

Una vez que se conoce el valor a alcanzar por el *Timer*, se extiende el programa InsertionSort.s con la configuración de los GPIO y del *Timer*.

```

1 lui x8, 1          # Direccion de los GPIO
2 lui x9, 3          # Direccion del Timer
3 lw x17, 0(x9)      # Carga del valor cuenta
4 lui x6, 21972       # 89997312
5 addi x6, x6, 2688   # 89997312 + 2688 = 90 * 10^6
6 sw x6, 1(x9)        # Registro ARR del Timer = 90 * 10^6
7 addi x18, x0, 5      # En y En_interrupt
8 addi x19, x0, 2      # Clear
9 addi x28, x0, -1     # 32 bits a '1'
10 sh x28, 2(x8)       # 16 LSBs de TRI a '1'
11
12 addi x27, x0, 10    # 10

```

Posteriormente, el proceso a seguir para mostrar cada número es el siguiente:

1. Guardar en un registro la dirección de memoria del número a mostrar en cada iteración.

```
| 1 addi x20, x0, 0      # i = 0  
| 2 add x21, x2, x20    # sp + i
```

2. Cargar en otro registro el número almacenado en dicha dirección de memoria.

```
| 1 lw x7, 0(x21)       # Cargar numero
```

3. Guardar dicho número en el registro datosOut del GPIO.

```
| 1 sw x7, 0(x8)        # Guardar numero a los GPIO
```

4. Iniciar la cuenta del *Timer* cargando los bits correspondientes a En y En_interrupt.

```
| 1 sb x18, 0(x9)       # Inicio cuenta del Timer
```

5. Realizar un bucle infinito para “detener” el microprocesador mientras el *Timer* cuenta hasta ARR.

```
| 1 beq x0, x0, 0        # Bucle para esperar 2s
```

6. Limpiar la cuenta del Timer para contar hasta ARR en la próxima iteración.

```
| 1 sb x19, 0(x9)       # Cuenta del Timer a cero
```

7. Incrementar en 1 el iterador de la dirección de memoria del número a mostrar.

```
| 1 addi x20, x20, 1     # i++
```

8. Comprobar si se debe repetir el proceso para el siguiente número en el *stack*.

```
| 1 blt x20, x27, -28   # i < 10 ?
```

9. Comprobar si se ha mostrado el último elemento. Retornar al *stack pointer* en caso afirmativo, y repetir de nuevo todo el proceso.

```
| 1 beq x20, x27, -36   # i == 10 ?
```

El archivo “InsertionSort.s” ya está listo para introducirlo como argumento del ensamblador “ensamblador_riscv” mostrado en la sección 4. Éste genera el archivo “InsertionSort.mem”, el cual se introduce en la memoria de instrucciones.

Para programar la placa FPGA con el diseño completo del núcleo, se deben seguir los siguientes pasos básicos en el flujo de diseño de Xilinx Vivado:

1. Análisis RTL (Register Transfer Level):

Vivado importa el diseño VHDL y lo analiza en busca de errores sintácticos o de errores básicos en el código.

2. Síntesis:

Vivado convierte el diseño RTL en una representación de nivel de puerta lógica (un nivel de abstracción más bajo). Esta representación (*netlist*) describe las puer-
tas y biestables del circuito, así como sus conexiones. La síntesis debe optimizar el
diseño para cumplir con los requisitos de área (cantidad de recursos utilizados) y
temporización (frecuencia de reloj).

3. Implementación:

Vivado utiliza la *netlist* generada en síntesis para ser implementada por las celdas lógicas y recursos físicos de la FPGA. Vivado debe optimizar esta implementación, de modo que cumpla con las restricciones de temporización y de área.

Para poder llevar a cabo esta etapa, es necesario incluir un archivo “.xdc” en nuestro diseño. XDC (*Xilinx Design Constraints*) define restricciones y configuraciones es-
pecíficas del diseño de la FPGA. Éstas incluyen, entre otras, la definición del reloj y asignaciones de entradas y salidas del circuito a pines del dispositivo.

En el siguiente fragmento del archivo “Basys-3-Master.xdc” se puede observar la asignación de botones de la FPGA a señales en el diseño. Se ha eliminado el co-
mentario en la línea correspondiente a la asignación del botón central de la Basys 3 a la señal “reset” (consultar archivo “.xdc” completo en Anexo C).

```
1 ## This file is a general .xdc for the Basys3 rev B board
2 ## To use it in a project:
3 ## - uncomment the lines corresponding to used pins
4 ## - rename the used ports (in each line, after get_ports)
   according to the top level signal names in the project
```

```
1 ##Buttons
2 set_property -dict { PACKAGE_PIN U18    IO_STANDARD LVCMOS33 } [
   get_ports reset]
3 #set_property -dict { PACKAGE_PIN T18    IO_STANDARD LVCMOS33 } [
   get_ports btnU]
```

```

4 #set_property -dict { PACKAGE_PIN W19      IO_STANDARD LVCMOS33 } [
    get_ports btnL]
5 #set_property -dict { PACKAGE_PIN T17      IO_STANDARD LVCMOS33 } [
    get_ports btnR]
6 #set_property -dict { PACKAGE_PIN U17      IO_STANDARD LVCMOS33 } [
    get_ports btnD]

```

4. Generación del *bitstream*:

Previo a la propia programación de la Basys 3, Vivado debe generar un archivo binario (*bitstream*) que contiene la información necesaria para configurar e implementar el diseño VHDL en la FPGA.

5. Programación del dispositivo:

Finalmente, habiendo conectado la FPGA al ordenador por USB (*Universal Serial Bus*, bus universal en serie), y habiendo conectado la placa en Vivado a través del *Hardware Manager*, se programa el dispositivo, es decir, se descarga el *bitstream* generado a su memoria de configuración. El diseño ya debería de estar ejecutándose en la FPGA.

Cabe recalcar que, pese a que las etapas de síntesis e implementación de Vivado comprobarán muchos requerimientos para asegurar el correcto funcionamiento de la FPGA, es crucial haber verificado previamente en simulación el correcto comportamiento del diseño. Esto evitará posibles fallos de la FPGA y daños en el *hardware* de la placa.

En el siguiente capítulo (7. Resultados), se comprobará la implementación del núcleo en la placa, así como su correcta ejecución del programa “*InsertionSort.mem*”. Se analizarán los pasos de diseño de Vivado en cuanto a síntesis, uso de recursos de la placa y de tiempo.

6.2. Ejemplo práctico 2: Correcto funcionamiento de GPIOs

En este ejemplo se va a mostrar el correcto funcionamiento de los GPIOs, pudiendo funcionar cada uno de ellos como entrada o como salida. Para ello, se va a implementar en la FPGA un sencillo programa, el cual dados los *bits* que marquen los interruptores de la Basys-3, enciende los leds correspondientes de la placa (aquellos que se encuentran justo encima de cada interruptor activado).

La siguiente representación gráfica de la Basys 3 muestra el comportamiento de los leds en dicho ejemplo:

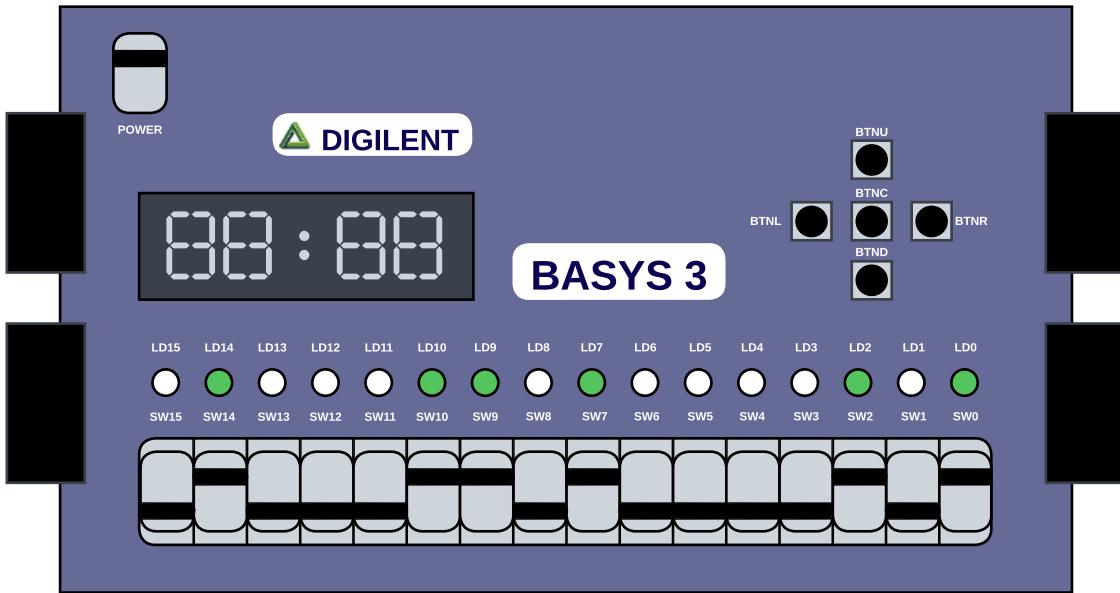


Fig. 6.6. Representación gráfica de la Basys 3 de Digilent.

Como se puede observar en la anterior figura, los leds encendidos corresponden a los interruptores activados.

El objetivo es por lo tanto asociar los 16 LSBs del GPIO a los leds, y los 16 MSBs a los interruptores. Al leer los bits de entrada de los interruptores, se carga dicho valor a un registro, se desplaza 16 bits a la derecha, y se guarda dicho valor en datosOut. De este modo, con un simple desplazamiento lógico se transfieren los bits de los interruptores a los leds.

El código en lenguaje ensamblador RISC-V del programa a ejecutar es el siguiente:

```

1 lui x20, 1           # Dirección del GPIO
2 lui x21, 16          # 0x100000
3 addi x21, x21, -1   # 0xFFFF
4 sh x21, 2(x20)       # 16 LSBs de TRI en '1' (salidas)
5 lw x22, 1(x20)        # Carga de datosIn en x22
6 srl x22, x22, 16     # x22 >> 16
7 sw x22, 0(x20)        # Guardar el desplazamiento en datosOut
8 jal x23, -16         # Repetir proceso

```

Introduciendo “0xFFFF” en los 16 MSBs del GPIO en el banco de pruebas, se observa en simulación como dicho valor se desplaza 16 bits a la derecha, y figura en los 16 LSBs del GPIO.

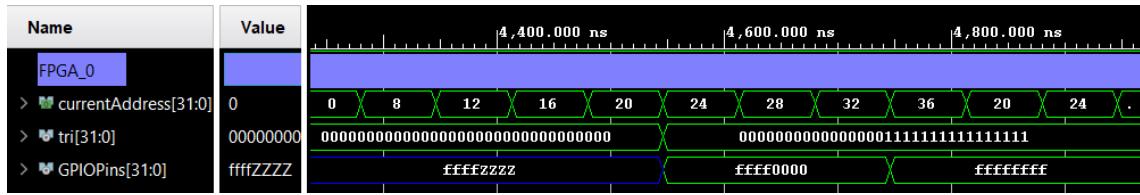


Fig. 6.7. Resultados de la simulación para FPGA_0.mem

En la anterior figura, currentAddress representa el PC, tri el registro TRI del GPIO, y GPIO Pins los pines del GPIO a la placa.

Una vez que se ha comprobado el correcto funcionamiento del programa en simulación, se procede a implementar dicho diseño en el dispositivo FPGA. Tras programarlo, se puede probar a encender y apagar los leds de la placa con los interruptores.

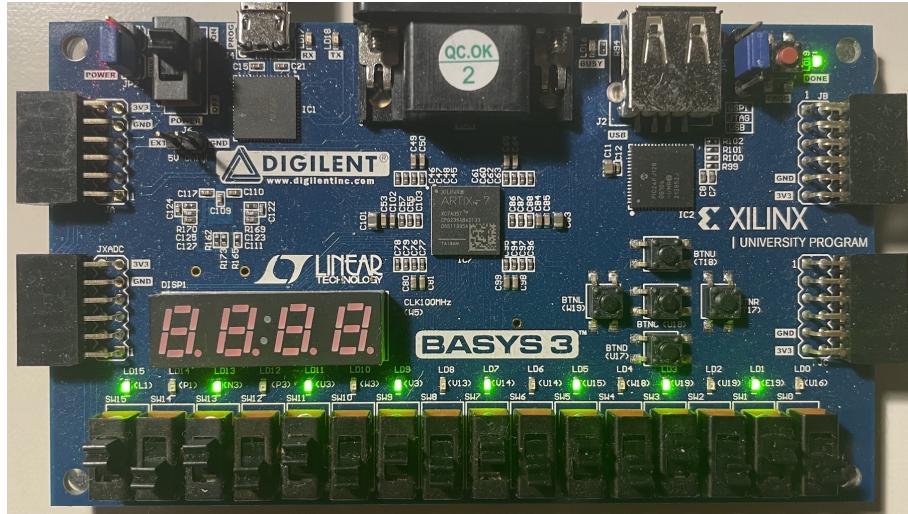


Fig. 6.8. Probando el programa FPGA_0.mem en la Basys 3 (1).

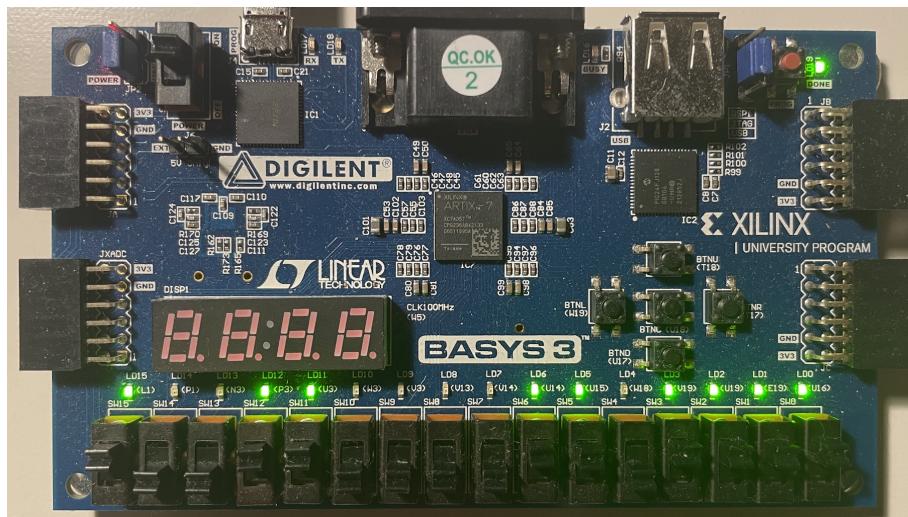


Fig. 6.9. Probando el programa FPGA_0.mem en la Basys 3 (2).

7. RESULTADOS

Tras programar la FPGA con el archivo bitstream generado, el programa “EXInsertion-Sort.mem” del ejemplo práctico 1 empieza a ejecutarse en el núcleo diseñado. Los 16 leds de la placa comienzan a mostrar los números ordenados correctamente, a intervalos de dos segundos cada uno.

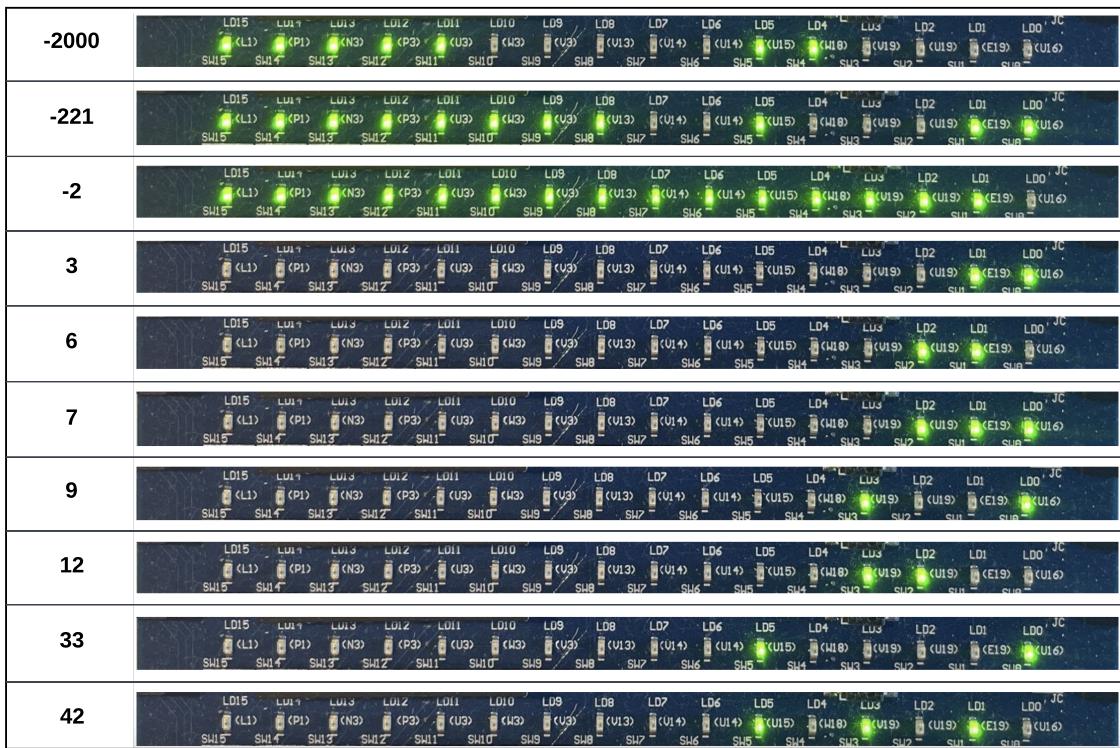


Fig. 7.1. Números correctamente ordenados en los leds de la FPGA.

Una vez que se ha comprobado el correcto funcionamiento del núcleo en la FPGA, se procede a analizar el diseño basándose en los informes de Vivado sobre la Síntesis e Implementación del proyecto. Estos incluyen un análisis del área o uso de recursos y un análisis temporal.

7.1. Análisis post-implementación del uso de recursos

Resource	Utilization	Available	Utilization %
LUT	3951	20800	19.00
LUTRAM	2048	9600	21.33
FF	1328	41600	3.19
IO	34	106	32.08
BUFG	2	32	6.25
MMC M	1	5	20.00

Fig. 7.2. Tabla del análisis post-implementación del uso de recursos en Vivado.

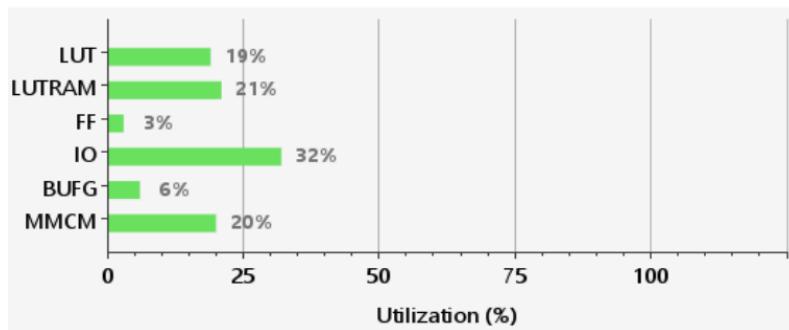


Fig. 7.3. Gráfica del análisis post-implementación del uso de recursos en Vivado.

El análisis del uso de recursos de Vivado, muestra el número de recursos físicos de la FPGA que han sido utilizados para la implementación del diseño del proyecto. Estos son:

- **LUT (*Look-Up Table*, tabla de consulta):**

Componente fundamental en la implementación de lógica combinacional, actuando como una tabla de verdad. Se ha utilizado el 19 % de las LUTs disponibles.

Este ejemplo muestra el funcionamiento de una LUT implementando la siguiente expresión booleana:

$$X = \bar{A}B + AB\bar{C} + \bar{B}C \quad (7.1)$$

Una LUT puede ser diseñada para cumplir dicha expresión, siendo A, B y C las entradas y X la salida. Usando un único multiplexor 8 a 1, se selecciona la salida X deseada con las entradas del mismo.

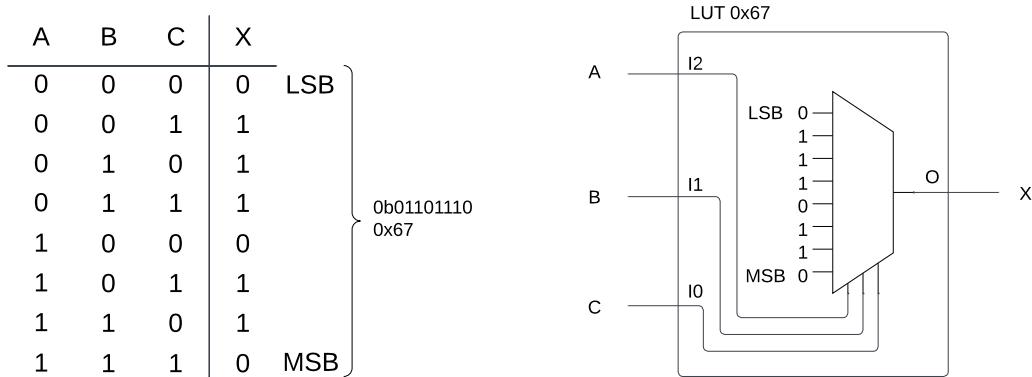


Fig. 7.4. Diseño de una LUT para la expresión booleana (7.1).

- **LUTRAM (LUT Random Access Memory, LUT de memoria de acceso aleatorio):**
Conjunto de LUTs organizadas como una estructura de memoria y utilizadas para almacenar datos. Se ha utilizado el 21,33 % de las LUTs disponibles.
- **FF (Flip-Flop, biestable):**
Componente de almacenamiento síncrono de un solo bit, que se suele utilizar para la formación de registros. Los registros son componentes de memoria de acceso muy rápido que almacenan datos de forma temporal para su uso por la CPU. Los registros del núcleo han utilizado el 3,19 % de los FFs disponibles.
- **IO (Input/Output, entrada/salida):**
Pines de entrada y salida de la FPGA. Habiendo utilizado el reloj interno, los 16 interruptores, los 16 leds y el botón central de la placa, suman un total de 34 pines IO. Constituyen el 32,08 % de los IO disponibles.
- **BUFG (global clock buffer, buffer global de reloj):**
Componente utilizado para distribuir señales de reloj de manera fiable. En este diseño, un BUFG corresponde a la conexión entre el reloj interno de 100 MHz de la FPGA y el bloque IP del *Clocking Wizard*. El otro BUFG corresponde a la salida de reloj de 45 MHz del bloque IP hacia el resto de componentes del diseño. Dos componentes BUFG que constituyen el 6,25 % de los BUFGs disponibles.
- **MMCM (Mixed-Mode Clock Manager, administrador de relojes de modo mixto):**
Componente utilizado para generar y controlar señales de reloj a distintas frecuencias y fases, a partir de una única señal de reloj. El único MMCM mostrado en las figuras 7.2 y 7.3 corresponde al bloque IP utilizado. Hay un total de cinco MMCM disponibles (20 %).

7.2. Análisis temporal

Name	Constraints	Status	WNS	TNS
✓ synth_1 (active)	constrs_1	synth_design Complete!		
✓ impl_1	constrs_1	route_design Complete!	1.158	0.000

Fig. 7.5. Análisis temporal en Vivado.

- **WNS (Worst Negative Slack, margen de retraso negativo máximo):**

Dados dos biestables tipo D (FF1 y FF2) con la misma señal de reloj, se produce un primer flanco de subida del reloj, por lo que $Q \leq D$ en FF1. Si hay mucha lógica entre ambos biestables, puede ocurrir que la señal que debe llegar a la entrada de FF2 lo haga tras al nuevo flanco de reloj. Este concepto se conoce como *Negative Slack*, y representa el tiempo entre el nuevo flanco de reloj y el instante en que la señal llega a dicho biestable.

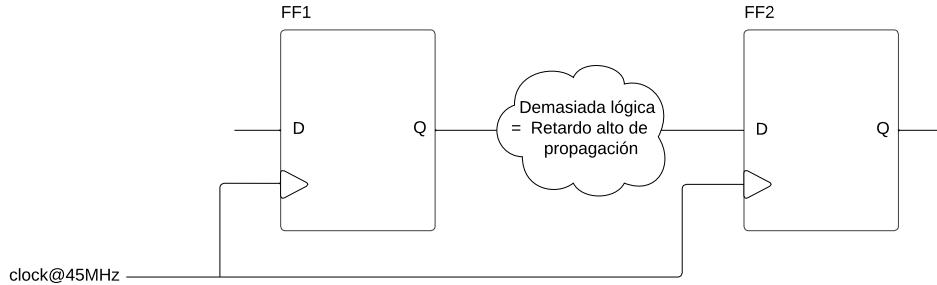


Fig. 7.6. WNS y el retardo en la propagación de señales entre componentes síncronos.

Si el *Negative Slack* es negativo, significa que la señal ha llegado con retraso. El *Worst Negative Slack* representa por lo tanto el peor *Negative Slack* del circuito.

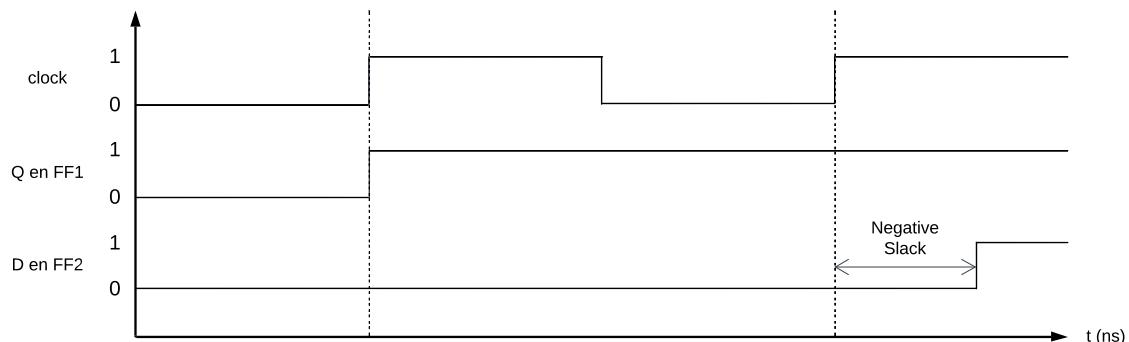


Fig. 7.7. Representación gráfica del *Negative Slack*.

Como se puede observar en la figura 7.5, el WNS del núcleo tiene un valor de 1,158 ns. Es decir, la última señal de todas que ha llegado a un biestable, lo ha hecho a 1,158 ns de que se produjera el siguiente ciclo de reloj.

Tal y como se ha explicado en el apartado anterior (6. Ejemplo práctico), el reloj se ha restringido a 45 MHz. El periodo del reloj (tiempo que tarda en realizar un ciclo) es inverso a su frecuencia:

$$T(s) = \frac{1}{f(Hz)} \quad (7.2)$$

Aplicando dicha fórmula se obtiene un periodo de 22,22 ns para la señal de reloj.

Puesto que el WNS ha sido de 1,158 ns, se puede calcular la frecuencia límite en la que no existe WNS negativo:

$$f = \frac{1}{(22,22 - 1,158) * 10^{-9}} = 47,48MHz$$

- **TNS** (*Total Negative Slack*, margen de retraso negativo total):

Representa la suma de todos los *Negative Slack* en el circuito. Como sucede en este diseño, si el WNS es positivo (no existe *Negative Slack*), el TNS debe ser nulo.

8. CONCLUSIONES

Se ha logrado cumplir con los objetivos previstos de este proyecto, desarrollando un núcleo de microprocesador basado en la arquitectura RV32I de RISC-V. Dicho núcleo es capaz de ejecutar un conjunto de hasta 43 instrucciones distintas, comunicarse con periféricos y manejar excepciones.

El núcleo se ha implementado en una FPGA Artix-7 contenida en la placa de desarrollo Basys-3 de Digilent. Al añadir 32 GPIOs, el núcleo es capaz de leer y escribir información sobre los pines de la Basys-3 (leds, interruptores, displays led, puertos Pmod, etc.). Así mismo, al implementar un temporizador, el núcleo es capaz de medir tiempos.

Durante el proceso de diseño de dicho núcleo, se ha observado la gran cantidad de recursos *online* disponibles sobre la arquitectura RISC-V. La documentación, tutoriales y herramientas compatibles muestran la existencia de una comunidad global que prima el desarrollo libre de tecnologías RISC-V frente a intereses comerciales.

Se demuestran también una serie de beneficios al diseñar e implementar el núcleo en una FPGA. Frente a la forma tradicional de diseñar circuitos digitales, las FPGAs permiten al diseñador crear circuitos mediante lenguajes de descripción de *hardware* como VHDL o Verilog, aportando una gran flexibilidad a los diseños.

Además, tanto el entorno de desarrollo (Vivado ML 2022.2) como la propia FPGA utilizada (Artix-7) han facilitado enormemente el desarrollo del núcleo.

Vivado ha permitido probar el diseño en cada etapa mediante simulaciones. Por otro lado, los recursos de la FPGA han resultado ser más que suficientes para la extensión de este proyecto, al haber utilizado en torno al 10 % de los mismos.

Se concluye también que escoger una frecuencia de trabajo de 45 MHz ha sido la decisión correcta cuando se desea una alta velocidad de procesamiento del núcleo, sin generar *Negative Slacks* negativos (frecuencia límite de 47,48 MHz).

Finalmente, el desarrollo de un ensamblador ha agilizado el proceso de probar el núcleo. Gracias a él se han podido verificar las instrucciones, los periféricos y se ha logrado crear un ejemplo que compara el tiempo de ejecución de tres algoritmos distintos.

Líneas futuras

El diseño de este microprocesador puede adaptarse a distintos mercados mediante múltiples mejoras, potenciando su eficiencia y posible comercialización. Dichas optimizaciones incluyen rediseños para aumentar prestaciones, expansiones de la arquitectura,

ampliaciones de funcionalidad e integración de compiladores.

- Rediseños para aumentar prestaciones:

El diseño se puede modificar para que sea segmentado (*pipelined*). Consiste en dividir la ejecución de las instrucciones en varias etapas de modo que éstas puedan solaparse, haciendo que el procesador sea capaz de ejecutar varias instrucciones a la vez, aumentando su velocidad de procesamiento [8].

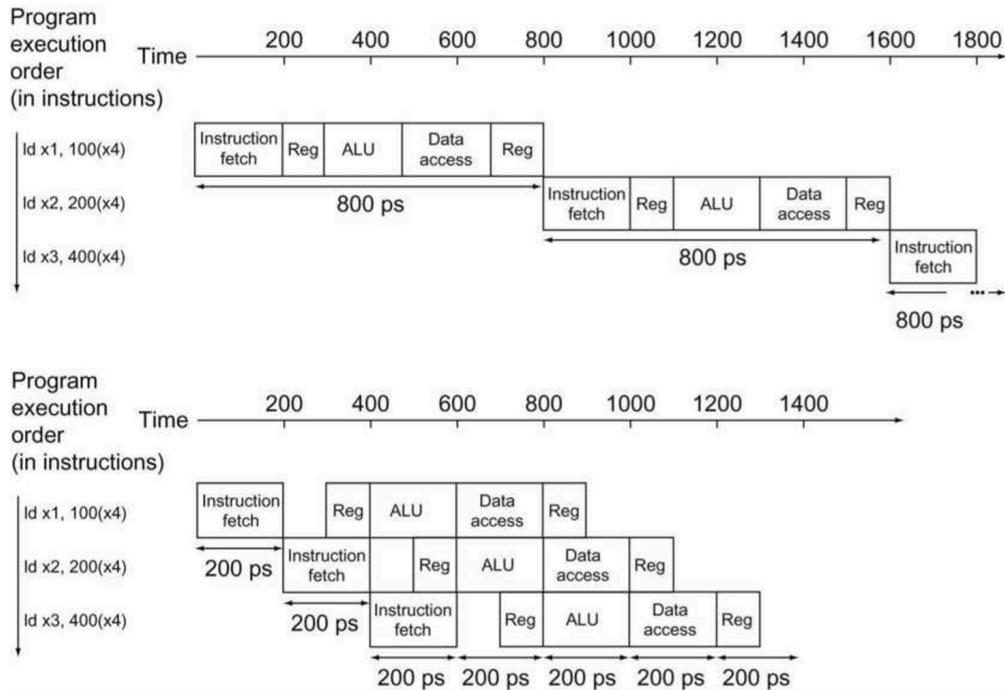


Fig. 8.1. Ejecución en un solo ciclo. Superior: [Sin segmentar]. Inferior: [Segmentada]. [8]

En el ejemplo de la figura 8.1, se puede observar como el diseño segmentado podría reducir la ejecución de tres instrucciones load de 2000 ps a 1300 ps.

Otra opción es el diseño de más de un núcleo. Permitiría al procesador ejecutar varios procesos a la vez (en paralelo) e incrementar el rendimiento del mismo.

- Expansiones de la arquitectura:

Con el objetivo de permitir al microprocesador realizar tareas más específicas, se le pueden implementar extensiones y varios modos de usuario o privilegio.

En cuanto a las extensiones, permitirían al procesador realizar nuevas tareas, como aquellas que requieran cierto nivel de precisión (realizar operaciones con decimales, incluyendo números de punto flotante).

En este trabajo no se han implementado las instrucciones `ecall` y `ebreak` por ser

necesario un sistema operativo. El microprocesador, por su parte, no dispone de un sistema operativo por requerir más de un modo de usuario/privilegio. Por lo tanto, se puede incluir como mejora hasta tres modos distintos (usuario, supervisor y máquina) permitiendo utilizar un sistema operativo como Linux y hasta ocho CSRs (ya se han implementado dos en el diseño actual).

- Ampliaciones de funcionalidad:

Otra posible mejora del núcleo consiste en añadir más periféricos, como más de un *Timer*, permitiéndole contar varios tiempos a la vez, una pantalla para mostrar información, puertos USB o puertos de comunicación serie SPI (*Serial Peripheral Interface*, interfaz periférica serial), UART (*Universal Asynchronous Receiver/Transmitter*, transmisor-receptor asíncrono universal) o I^2C (*Inter-Integrated Circuit*, circuito inter-integrado). Estos últimos otorgan mucha funcionalidad al núcleo, puesto que le permiten comunicarse con cualquier otro dispositivo que soporte estos protocolos (teclados, ratones, displays, etc.).

- Integración de compiladores:

Se pueden integrar los compiladores y ensambladores estándar de RISC-V en el núcleo, permitiéndole ejecutar programas más completos.

Por ejemplo, se ha publicado el “RISC-V GNU Toolchain” para compilar en C y C++, el cual ofrece compiladores para variantes de la ISA, como RV32 y RV64 IMAFD, I, IM, IA, IMA y IMFD [10].

9. MARCO REGULADOR

El principal marco regulador de este trabajo es la especificación RISC-V.

RISC-V es un conjunto de instrucciones con una serie de normas en cuanto al tamaño y codificación de las mismas. Estas especificaciones han sido publicadas por la organización “RISC-V International”.

Las especificaciones de RISC-V utilizadas en este trabajo se encuentran descritas en el documento “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213”.

Es importante recalcar que al ser RISC-V una arquitectura de conjunto de instrucciones libre, cualquiera puede diseñar, desarrollar y vender microprocesadores RISC-V sin tener que pagar licencia alguna. Los diseñadores de núcleos y chips de RISC-V pueden decidir si hacerlos *open source* (hacer público su diseño, de modo que cualquiera lo examine, modifique o distribuya) o no, por lo que no toda la industria RISC-V es libre.

El lenguaje de descripción de *hardware* que se ha utilizado para el diseño del microprocesador es VHDL.

VHDL fue definido en 1993 por el IEEE (*Institute of Electrical and Electronics Engineers*, instituto de ingenieros eléctricos y electrónicos), bajo la especificación ANSI/IEEE 1076-1993 [9]. Ésta define la sintaxis, semántica, tipos de datos y objetos, estructuras de control, operadores, funciones, descripción de componentes y circuitos, simulación y finalmente, la temporización.

Por último, cabe mencionar que todo el diseño del procesador se ha realizado en Vivado ML 2022.2 standard, un paquete de software para el diseño de circuitos digitales en FPGAs y SoCs (*System on Chip*, sistema en un *chip*) de Xilinx. Pese a que el diseño de proyectos en Vivado no requiere de ningún tipo de licencia, el uso de algunos bloques IP proporcionados por Xilinx o terceros sí pueden tener restricciones en su uso, o requerir de una licencia para comercializar diseños que los contengan.

10. PRESUPUESTO

En este apartado se va a abordar el coste que supone llevar a cabo este proyecto, desde el coste por hardware o software, hasta el coste por contratación de un ingeniero.

Los recursos de **software** que se han empleado para el desarrollo de este trabajo son los siguientes:

- Xilinx Vivado ML 2022.2: Desarrollo en VHDL del microprocesador. Incluye diseño, simulación, síntesis e implementación.
- Visual Studio Code 1.81.1: Desarrollo de un ensamblador en C++ para generar el lenguaje máquina a ejecutar por el microprocesador.
- Compilador Clang++ 14.0.3: Compilador del ensamblador.
- LucidChart: Diseño de diagramas de bloques, máquina de estados, y otros diagramas.
- Microsoft Excel para Mac Versión 16.76: Diseño de tablas y figuras.
- Overleaf: Escritura con LaTeX de la memoria siguiendo el formato requerido.

Todas estas herramientas son gratuitas con licencia de estudiante, por lo que el apartado de software no ha supuesto costo alguno.

En cuanto al **hardware**, se ha adquirido:

- Digilent Basys 3 with AMD Artix 7 FPGA Board.
Precio: 165\$ = 151,41€ (1\$ = 0.92€)
- Cable USB 2.0 de tipo A macho a micro B.
Precio: 4,90€

Coste del hardware: $151,41\text{€} + 4,90\text{€} = 156,31\text{€}$

El principal gasto de este proyecto es el coste de contratar a un ingeniero para diseñar el núcleo y llevar a cabo todas las pruebas necesarias.

Suponiendo un sueldo de 15€/hora, y habiendo tardado diez meses desde la idea original al diseño final, a una media de dos horas diarias, el coste total de contratar a un ingeniero para llevar a cabo este proyecto es de:

Coste por **sueldo de ingeniería**: 2 horas/día * 30 días/mes * 10 meses * 15€/hora = 9000€

Coste final del proyecto = 156,31€ + 9000€ = **9156,31€**

11. IMPACTO SOCIO-ECONÓMICO

Desde la publicación de la primera especificación de RISC-V, el 13 de Mayo de 2011, por la Universidad de California en Berkeley, el desarrollo de núcleos RISC-V y su expansión a diversos sectores de la industria se ha mostrado en constante evolución.

En 2015 se fundó la “RISC-V Foundation” en Estados Unidos. En la actualidad, dicha asociación está reubicada en Suiza bajo el nombre de “RISC-V International”, con el objetivo de imprimir un carácter neutral a la misma [10].

Hoy día RISC-V cuenta con más de 3.000 miembros, entre los que destacan Google, Huawei, IBM, Intel, NVIDIA, Alibaba, Qualcomm, Samsung, Seagate y Western Digital.

Así mismo en 2015, los pioneros de RISC-V introdujeron dicha especificación al resto del mundo fundando SiFive, siendo ésta una de las principales organizaciones de desarrollo de núcleos RISC-V hoy en día [11]. SiFive ofrece cuatro familias distintas de procesadores: *Performance*, *Automotive*, *Intelligence* y *Essential* [12]. Samsung anunció en 2019 que incluiría núcleos SiFive en módulos de visión por computador y aplicaciones 5G [13].

Otro de los principales competidores del desarrollo de núcleos RISC-V es T-Head, perteneciente a Alibaba Group [14]. En 2021, T-Head hizo *open source* cuatro de sus núcleos RISC-V incluidos en la serie XuanTie [15].

Como noticias recientes relacionadas con RISC-V, cabe destacar el anuncio de la NASA en Septiembre de 2022, sobre la inclusión de núcleos RISC-V en la próxima generación de su procesador HPSC (*High-Performance Spaceflight Computing*, computación de alto rendimiento para vuelos espaciales) [16]. Así mismo, Google expresó en Enero de 2023 la posibilidad de que RISC-V fuera la principal arquitectura para sistemas Android [17].

Pese a que RISC-V aún no se ha establecido en el mercado de ordenadores comerciales y de servidores, sí se ha asentado en el mercado de sistemas embebidos (especialmente aquellos que tienen aplicaciones específicas). Por ejemplo, la compañía Renesas Technology ha desarrollado el microprocesador AndesCore AX45MP [18], y la empresa Espressif Systems la serie de microcontroladores SoC ESP32-C, ambos basados en la arquitectura RISC-V [19]. El procesador AndesCore AX45MP ha sido incluido en el nuevo SBC (*Single-Board Computer*, ordenador de placa única) de Asus llamado Tinker V, en competencia directa con Raspberry Pi [20]. Además, la empresa Western Digital Technologies, Inc. ha sacado al mercado una serie de núcleos RISC-V, llamados SweRV, dedicados al control de SSDs (*Solid State Drive*, disco de estado sólido) [21].

En Julio de 2022, “RISC-V International” publicó que se habían distribuido más de 10.000

millones de núcleos RISC-V a nivel global [22]. De hecho, la empresa Qualcomm anunció en Diciembre del mismo año haber entregado chips con hasta 650 millones de núcleos RISC-V [23].

Queda demostrado su gran potencial, al estar asentándose en los dispositivos de uso diario, entrando en competencia directa con las arquitecturas x86 y ARM.

RISC-V está influyendo en el mundo de la tecnología en varios **aspectos**:

- **Sociales:**

Al ser una arquitectura abierta y libre, incrementa la democratización de la tecnología, permitiendo a individuos y empresas acceder a una arquitectura de procesador muy eficiente.

A nivel educativo, RISC-V ayuda a estudiantes a aprender con una arquitectura sin restricciones de propiedad intelectual.

- **Económicos:**

Al tener una ISA base simple, con extensiones disponibles, permite a empresas y desarrolladores crear procesadores adaptados a sus necesidades.

La consecuencia inmediata de ser una arquitectura libre, facilita a individuos o empresas sin recursos económicos (como *startups*) desarrollar sus propios procesadores, eliminando la barrera inicial de tener que pagar licencias como las de ARM.

- **Éticos:**

RISC-V es una iniciativa que fomenta la colaboración y contribución de la comunidad internacional. Esto puede tener implicaciones éticas positivas al promover la cooperación en la investigación y el desarrollo de la tecnología. Empresas e individuos que decidan hacer *open source* sus propios diseños, servirán como catalizadores de este desarrollo a nivel global.

BIBLIOGRAFÍA

- [1] J. Gomar, “Cuál fue el primer microprocesador de la historia y quien lo inventó,” *Profesionalreview*, 2018.
- [2] *Procesador Intel® Core™ i9-13900K*. [En línea]. Disponible en: <https://www.intel.com/content/www/xl/es/products/sku/230496/intel-core-i913900k-processor-36m-cache-up-to-5-80-ghz/specifications.html>.
- [3] R. Solé, “Diferencias RISC y CISC: Comparando las principales arquitecturas de procesadores,” *Profesionalreview*, 2021. [En línea]. Disponible en: <https://www.profesionalreview.com/2021/07/18/risc-vs-cisc/>.
- [4] R. Solé, “RISC: La arquitectura de procesadores usada por ARM para cambiar el mercado,” *Profesionalreview*, 2021. [En línea]. Disponible en: <https://www.profesionalreview.com/2021/07/17/que-es-risc/>.
- [5] Wikipedia, *RISC-V*, 2023. [En línea]. Disponible en: <https://es.wikipedia.org/wiki/RISC-V>.
- [6] *RISC-V: Immediate Encoding Variants*, 2016. [En línea]. Disponible en: <https://stackoverflow.com/questions/39427092/risc-v-immediate-encoding-variants>.
- [7] S. I. Andrew Waterman Krste Asanovic, “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA,” CS Division, EECS Department, University of California, Berkeley, inf. téc., 2019.
- [8] J. L. H. David A. Patterson, *Computer Organization and Design RISC-V Edition*. Morgan Kaufmann, 2017.
- [9] “VHDL,” *Wikipedia*, 2023. [En línea]. Disponible en: <https://es.wikipedia.org/wiki/VHDL>.
- [10] *About RISC-V*, 2021. [En línea]. Disponible en: <https://riscv.org/about/>.
- [11] *SiFive Web Page*. [En línea]. Disponible en: <https://www.sifive.com/>.
- [12] *SiFive Core IP Web Page*. [En línea]. Disponible en: <https://www.sifive.com/risc-v-core-ip>.
- [13] A. Anton Shilov, *Samsung To Use SiFive RISC-V Cores For SoCs, Automotive, 5G Applications*, 2019. [En línea]. Disponible en: <https://riscv.org/news/2019/12/samsung-to-use-sifive-risc-v-cores-for-socts-automotive-5g-applications-anton-shilov-anandtech/>.
- [14] *T-Head XuanTie Product Overview*. [En línea]. Disponible en: <https://www.t-head.cn/product/overview>.

- [15] Pandaily, *Alibaba Announces Open Source RISC-V-Based Xuantie Series Processors*, 2021. [En línea]. Disponible en: <https://pandaily.com/alibaba-announces-open-source-risc-v-based-xuantie-series-processors/>.
- [16] SiFive, *NASA Makes RISC-V the Go-to Ecosystem for Future Space Missions*, 2022. [En línea]. Disponible en: <https://www.sifive.com/press/nasa-selects-sifive-and-makes-risc-v-the-go-to-ecosystem>.
- [17] R. Amadeo, *Google wants RISC-V to be a “tier-1” Android architecture*, 2023. [En línea]. Disponible en: <https://arstechnica.com/gadgets/2023/01/google-announces-official-android-support-for-risc-v/>.
- [18] Renesas, *General-purpose Microprocessors with RISC-V CPU Core (Andes AX45MP Single) (1.0 GHz) with 2ch Gigabit Ethernet*. [En línea]. Disponible en: <https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rz-mpus/rzfive-general-purpose-microprocessors-risc-v-cpu-core-andes-ax45mp-single-10-ghz-2ch-gigabit-ethernet>.
- [19] Espressif. [En línea]. Disponible en: <https://www.espressif.com/>.
- [20] D. Robinson, “Enter Tinker: Asus pulls out RISC-V board it hopes trumps Raspberry PI,” *The Register*, 2023. [En línea]. Disponible en: https://www.theregister.com/2023/03/15/asus_announces_riscv_tinker_board/.
- [21] “RISC-V and Open Source Hardware Address New Compute Requirements,” Western Digital, inf. téc.
- [22] R.-V. C. News, *Europe steps up as RISC-V ships 10bn cores | Nick Flaherty, EE News Europe*, 2022. [En línea]. Disponible en: <https://riscv.org/news/2022/07/europe-steps-up-as-risc-v-ships-10bn-cores-nick-flaherty-ee-news-europe/>.
- [23] M. Varma, *Keynote: Accelerating Innovation with RISC-V: Past, Present and Future*, 2022. [En línea]. Disponible en: https://www.youtube.com/watch?v=t6_9pbgg1LI&t=0s.
- [24] J. Winans, “RISC-V Assembly Language Programming,” inf. téc., 2022.

ANEXO A. RV32I BASE INSTRUCTION SET ENCODING [24]

Usage Template	Type	Description	Detailed Description
add rd, rs1, rs2	R	Add	$rd \leftarrow rs1 + rs2, pc \leftarrow pc+4$
addi rd, rs1, imm	I	Add Immediate	$rd \leftarrow rs1 + imm_i, pc \leftarrow pc+4$
and rd, rs1, rs2	R	And	$rd \leftarrow rs1 \wedge rs2, pc \leftarrow pc+4$
andi rd, rs1, imm	I	And Immediate	$rd \leftarrow rs1 \wedge imm_i, pc \leftarrow pc+4$
auipc rd, imm	U	Add Upper Immediate to PC	$rd \leftarrow pc + imm_u, pc \leftarrow pc+4$
beq rs1, rs2, pcrel_13	B	Branch Equal	$pc \leftarrow pc + ((rs1==rs2) ? imm_b : 4)$
bge rs1, rs2, pcrel_13	B	Branch Greater or Equal	$pc \leftarrow pc + ((rs1>=rs2) ? imm_b : 4)$
bgeu rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	$pc \leftarrow pc + ((rs1>rs2) ? imm_b : 4)$
blt rs1, rs2, pcrel_13	B	Branch Less Than	$pc \leftarrow pc + ((rs1<rs2) ? imm_b : 4)$
bltu rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	$pc \leftarrow pc + ((rs1<rs2) ? imm_b : 4)$
bne rs1, rs2, pcrel_13	B	Branch Not Equal	$pc \leftarrow pc + ((rs1!=rs2) ? imm_b : 4)$
csrrw rd, csr, rs1	I	Atomic Read/Write	$rd \leftarrow csr, csr \leftarrow rs1, pc \leftarrow pc+4$
csrrs rd, csr, rs1	I	Atomic Read and Set	$rd \leftarrow csr, csr \leftarrow csr \vee rs1, pc \leftarrow pc+4$
csrrc rd, csr, rs1	I	Atomic Read and Clear	$rd \leftarrow csr, csr \leftarrow csr \wedge \sim rs1, pc \leftarrow pc+4$
csrrwi rd, csr, zimm	I	Atomic Read/Write Immediate	$rd \leftarrow csr, csr \leftarrow zimm, pc \leftarrow pc+4$
csrsi rd, csr, zimm	I	Atomic Read and Set Immediate	$rd \leftarrow csr, csr \leftarrow csr \vee zimm, pc \leftarrow pc+4$
csrrci rd, csr, zimm	I	Atomic Read and Clear Immediate	$rd \leftarrow csr, csr \leftarrow csr \wedge \sim zimm, pc \leftarrow pc+4$
ecall	I	Environment Call	Transfer Control to Debugger
ebreak	I	Environment Break	Transfer Control to Operating System
jal rd, pcrel_21	J	Jump And Link	$rd \leftarrow pc+4, pc \leftarrow pc+imm_j$
jalr rd, imm(rs1)	I	Jump And Link Register	$rd \leftarrow pc+4, pc \leftarrow (rs1+imm_i) \& \sim 1$
lb rd, imm(rs1)	I	Load Byte	$rd \leftarrow sx(m8(rs1+imm_i)), pc \leftarrow pc+4$
lbu rd, imm(rs1)	I	Load Byte Unsigned	$rd \leftarrow zx(m8(rs1+imm_i)), pc \leftarrow pc+4$
lh rd, imm(rs1)	I	Load Halfword	$rd \leftarrow sx(m16(rs1+imm_i)), pc \leftarrow pc+4$
lhu rd, imm(rs1)	I	Load Halfword Unsigned	$rd \leftarrow zx(m16(rs1+imm_i)), pc \leftarrow pc+4$
lui rd, imm	U	Load Upper Immediate	$rd \leftarrow imm_u, pc \leftarrow pc+4$
lw rd, imm(rs1)	I	Load Word	$rd \leftarrow sx(m32(rs1+imm_i)), pc \leftarrow pc+4$
or rd, rs1, rs2	R	Or	$rd \leftarrow rs1 \vee rs2, pc \leftarrow pc+4$
ori rd, rs1, imm	I	Or Immediate	$rd \leftarrow rs1 \vee imm_i, pc \leftarrow pc+4$
sb rs2, imm(rs1)	S	Store Byte	$m8(rs1+imm_s) \leftarrow rs2[7:0], pc \leftarrow pc+4$
sh rs2, imm(rs1)	S	Store Halfword	$m16(rs1+imm_s) \leftarrow rs2[15:0], pc \leftarrow pc+4$
sll rd, rs1, rs2	R	Shift Left Logical	$rd \leftarrow rs1 << (rs2\%XLEN), pc \leftarrow pc+4$
slli rd, rs1, shamt	I	Shift Left Logical Immediate	$rd \leftarrow rs1 << shamt_i, pc \leftarrow pc+4$
slt rd, rs1, rs2	R	Set Less Than	$rd \leftarrow (rs1 < rs2) ? 1 : 0, pc \leftarrow pc+4$
slti rd, rs1, imm	I	Set Less Than Immediate	$rd \leftarrow (rs1 < imm_i) ? 1 : 0, pc \leftarrow pc+4$
sltiu rd, rs1, imm	I	Set Less Than Immediate Unsigned	$rd \leftarrow (rs1 < imm_i) ? 1 : 0, pc \leftarrow pc+4$
sltu rd, rs1, rs2	R	Set Less Than Unsigned	$rd \leftarrow (rs1 < rs2) ? 1 : 0, pc \leftarrow pc+4$
sra rd, rs1, rs2	R	Shift Right Arithmetic	$rd \leftarrow rs1 >> (rs2\%XLEN), pc \leftarrow pc+4$
srai rd, rs1, shamt	I	Shift Right Arithmetic Immediate	$rd \leftarrow rs1 >> shamt_i, pc \leftarrow pc+4$
srl rd, rs1, rs2	R	Shift Right Logical	$rd \leftarrow rs1 >> (rs2\%XLEN), pc \leftarrow pc+4$
srlt rd, rs1, shamt	I	Shift Right Logical Immediate	$rd \leftarrow rs1 >> shamt_i, pc \leftarrow pc+4$
sub rd, rs1, rs2	R	Subtract	$rd \leftarrow rs1 - rs2, pc \leftarrow pc+4$
sw rs2, imm(rs1)	S	Store Word	$m32(rs1+imm_s) \leftarrow rs2[31:0], pc \leftarrow pc+4$
xor rd, rs1, rs2	R	Exclusive Or	$rd \leftarrow rs1 \oplus rs2, pc \leftarrow pc+4$
xori rd, rs1, imm	I	Exclusive Or Immediate	$rd \leftarrow rs1 \oplus imm_i, pc \leftarrow pc+4$

ANEXO B. CÓDIGO DEL DISEÑO VHDL

RISCV_CPU.vhd:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.ComponentsPkg.all;
5
6 entity RISCV_CPU is
7     port(
8         clk:          in    std_logic;
9         reset:        in    std_logic;
10        GPIOBins:    inout std_logic_vector(31 downto 0)
11    );
12 end RISCV_CPU;
13
14 architecture RISCV_CPU_ARCH of RISCV_CPU is
15
16    signal nextPC:           std_logic_vector(31 downto 0);
17    signal currentPC:        std_logic_vector(31 downto 0);
18    signal newIns:           std_logic_vector(31 downto 0);
19    signal newInsexception: std_logic_vector(31 downto 0);
20    signal PCPlus4:          std_logic_vector(31 downto 0);
21    signal inst:             std_logic_vector(31 downto 0);
22    signal MUXOutSig:       std_logic_vector(31 downto 0);
23    signal writeData:        std_logic_vector(31 downto 0);
24    signal ALUControlSig:   std_logic_vector(3 downto 0);
25    signal immValue:         std_logic_vector(31 downto 0);
26    signal r1Sig:            std_logic_vector(31 downto 0);
27    signal r2Sig:            std_logic_vector(31 downto 0);
28    signal dataIn:           std_logic_vector(31 downto 0);
29    signal regOrImm:         std_logic_vector(31 downto 0);
30    signal br:               std_logic_vector(31 downto 0);
31    signal ALUResult:        std_logic_vector(31 downto 0);
32    signal memOut:           std_logic_vector(31 downto 0);
33    signal loadControlOut:   std_logic_vector(31 downto 0);
34    signal comp:              std_logic_vector(2 downto 0);
35    signal dataEn:            std_logic;
36    signal GPIOEn:            std_logic;
37    signal GPIOOut:           std_logic_vector(31 downto 0);
38    signal TimerEn:           std_logic;
39    signal TimerOut:          std_logic_vector(31 downto 0);
40    signal clock:              std_logic;
41    signal exceptionStatus:  std_logic;
42    signal mcauseSig:         std_logic_vector(31 downto 0) := (others
43                                => '0');
```

```

44 signal CSRIinput:           std_logic_vector(31 downto 0);
45 signal CSROutput:          std_logic_vector(31 downto 0);
46 --signal index:            natural range 0 to 1; -- (Mario) Ya no
47      hace falta
48
49 signal microcode:          std_logic_vector(23 downto 0);
50
51      ----microcode-signals
52      -----
53      -----SIGNALS
54
55 signal CSRWriteEn:         std_logic;
56 signal atomicOpt:          std_logic_vector(1 downto 0);
57 signal r1orzMimm:          std_logic;
58 signal auipc:              std_logic;
59 signal PCEn:               std_logic;
60 signal insRegEn:           std_logic;
61 signal ALUop:              std_logic_vector(1 downto 0);
62 signal immSel:             std_logic_vector(2 downto 0);
63 signal regWriteEn:         std_logic;
64 signal wdSel:              std_logic_vector(1 downto 0);
65 signal regImmSel:          std_logic;
66 signal jumpSel:            std_logic;
67 signal PCSel:              std_logic;
68 signal memWriteEn:         std_logic;
69 signal ALUMemSel:          std_logic_vector(1 downto 0);
70 signal nBits:              std_logic_vector(1 downto 0);
71 signal signedOrUnsigned:   std_logic;
72 signal IRQ:                std_logic;
73
74
75
76
77 component clk_wiz_0
78     port
79     (
80         clk_out1      : out    std_logic;
81         reset        : in     std_logic;
82         locked       : out    std_logic;
83         clk_in1      : in     std_logic
84     );
85 end component;
86
87
88 begin
89
90     CLK50: clk_wiz_0
91         port map
92         (
93             clk_out1 => clock,
94             reset => reset,
95             locked => open,
96             clk_in1 => clk
97         );
98
99     CSRWriteEn      <= microcode(23);
100    atomicOpt       <= microcode(22 downto 21);
101    r1orzMimm      <= microcode(20);

```

```

93  auipc          <= microcode(19);
94  PCEn           <= microcode(18);
95  insRegEn       <= microcode(17);
96  ALUOp          <= microcode(16 downto 15);
97  immSel          <= microcode(14 downto 12);
98  regWriteEn     <= microcode(11);
99  wdSel           <= microcode(10 downto 9);
100 regImmSel      <= microcode(8);
101 jumpSel         <= microcode(7);
102 PCSel           <= microcode(6);
103 memWriteEn     <= microcode(5);
104 ALUMemSel       <= microcode(4 downto 3);
105 nBits           <= microcode(2 downto 1);
106 signedOrUnsigned <= microcode(0);

107
108 PC_U: ProgramCounter
109 port map(
110     nextAddress => nextPC,
111     PCEn => PCEn,
112     reset => reset,
113     clock => clock,
114     currentAddress => currentPC
115 );
116
117 INSMEM_U: InstructionMemory
118 port map(
119     readAddress => currentPC,
120     instruction => newIns
121 );
122
123 PCPlus4 <= std_logic_vector(unsigned(currentPC) + 4);

124
125 EXCEPTIONCNTRL_U: ExceptionControl
126 port map(
127     input => newIns,
128     exceptionstatus => exceptionStatus,
129     output => newInsException
130 );
131
132 INSREG_U: singleRegister
133 generic map(
134     REGSIZE => 32
135 )
136 port map(
137     input => newInsException,
138     writeEn => insRegEn,
139     reset => reset,
140     clock => clock,
141     output => inst
142 );
143

```

```

144 ALUCONTR_U: ALUControl
145   port map(
146     input => inst,
147     ALUop => ALUop,
148     output => ALUControlSig
149   );
150
151 IMMSEL_U: ImmSelect
152   port map(
153     input => inst,
154     immSel => immSel,
155     output => immValue
156   );
157
158 with wdSel
159   select writeData <= PCPlus4      when "00",
160           loadControlOut when "01",
161           CSROutput       when "10",
162           (others => '0') when others;
163
164 REGFILE_U: RegisterFile
165   port map(
166     rs1 => unsigned(inst(19 downto 15)),
167     rs2 => unsigned(inst(24 downto 20)),
168     rd => unsigned(inst(11 downto 7)),
169     writeData => writeData,
170     regWriteEn => regWriteEn,
171     clock => clock,
172     reset => reset,
173     r1 => r1Sig,
174     r2 => r2Sig
175   );
176
177 CSRS_U: CSRs
178   port map(
179     input => CSRInput,
180     CSRWriteEn => CSRWriteEn,
181     atomicOpt => atomicOpt,
182     CSRSel => inst(20),
183     exceptionStatus => exceptionStatus,
184     mcause => mcauseSig,
185     clock => clock,
186     reset => reset,
187     output => CSROutput
188   );
189
190 CU_U: ControlUnit
191   port map(
192     instruction => inst,
193     comparison => comp,
194     ALUresult => ALUresult,

```

```

195     IRQ => IRQ,
196     reset => reset,
197     clock => clock,
198     microcode => microcode,
199     exceptionStatus => exceptionStatus,
200     mcause => mcauseSig
201   );
202
203 COMP_U: Comparison
204   port map(
205     instruction => inst,
206     r1 => r1Sig,
207     r2 => r2Sig,
208     comparison => comp
209   );
210
211   with regImmSel
212   select regOrImm <= immValue      when '0',
213           r2Sig          when others;
214
215   br <= std_logic_vector(signed(currentPC) + signed(immValue));
216
217 ALU_U: ALU
218   port map(
219     r1 => r1Sig,
220     r2 => regOrImm,
221     control => ALUControlSig,
222     overflow => open, --overflowSig,
223     resultValue => ALUResult
224   );
225
226 STOREC_U: StoreControl
227   port map(
228     input => r2Sig,
229     instruction => inst,
230     output => dataIn
231   );
232
233   with r1orzimm
234   select CSRIinput <= r1Sig when '0',
235           immValue when others;
236
237 JUMPC_U: JumpControl
238   port map(
239     jumpSel => jumpSel,
240     PCPlus4 => PCPlus4,
241     branch => br,
242     PCSel => PCSel,
243     ALUresult => ALUresult,
244     nextPC => nextPC
245   );

```

```

246
247     dataEn <= memWriteEn and (not ALUresult(12));
248
249     MEM_U: DataMemory
250         port map(
251             writeEn => dataEn,
252             address => ALUResult(11 downto 0),
253             dataIn => dataIn,
254             clock => clock,
255             dataOut => memOut
256         );
257
258
259     GPIOEn <= memWriteEn and ALUresult(12) and (not ALUresult(13));
260
261     GPIO_U: GPIO
262         port map(
263             writeEn => GPIOEn,
264             address => ALUresult(1 downto 0),
265             dataIn => dataIn,
266             reset => reset,
267             clock => clock,
268             dataOut => GPIO0out,
269             data => GPIOpins
270         );
271
272
273     TimerEn <= memWriteEn and ALUresult(12) and ALUresult(13);
274
275     TIMER_U: Timer
276         port map(
277             address => ALUresult(1 downto 0),
278             dataIn => dataIn,
279             writeEn => TimerEn,
280             reset => reset,
281             clock => clock,
282             dataOut => TimerOut,
283             timerInterrupt => IRQ
284         );
285
286
287     with ALUMemSel
288         select MUXOutSig <= memOut      when "00",
289                                         ALUResult      when "01",
290                                         GPIO0out      when "10",
291                                         TimerOut      when others;
292
293
294     LOADC_U: LoadControl
295         port map(
296             MUXOutSig => MUXOutSig,
297             br => br,
298             nBits => nBits,
299             signedOrUnsigned => signedOrUnsigned,
300             auipc => auipc,
301             LoadControl => LoadControlOut
302         );
303
304

```

```
397 end RISCV_CPU_ARCn;
```

ComponentsPkg.vhd:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 package ComponentsPkg is
6
7     component ProgramCounter is
8         port(
9             nextAddress:    in  std_logic_vector(31 downto 0);
10            PCEn:          in  std_logic;
11            reset:          in  std_logic;
12            clock:          in  std_logic;
13            currentAddress: out std_logic_vector(31 downto 0)
14        );
15    end component;
16
17    component InstructionMemory is
18        port(
19            readAddress:    in  std_logic_vector(31 downto 0);
20            instruction:   out std_logic_vector(31 downto 0)
21        );
22    end component;
23
24    component ExceptionControl is
25        port(
26            input:          in  std_logic_vector(31 downto 0);
27            exceptionStatus: in  std_logic;
28            output:         out std_logic_vector(31 downto 0)
29        );
30    end component;
31
32    component singleRegister is
33        generic(
34            REGSIZE: natural
35        );
36        port(
37            input:    in  std_logic_vector(REGSIZE - 1 downto 0);
38            writeEn:  in  std_logic;
39            reset:    in  std_logic;
40            clock:    in  std_logic;
41            output:   out std_logic_vector(REGSIZE - 1 downto 0)
42        );
43    end component;
44
45    component RegisterFile is
46        port(
```

```

47      rs1:      in  unsigned(4 downto 0);
48      rs2:      in  unsigned(4 downto 0);
49      rd:       in  unsigned(4 downto 0);
50      writeData: in  std_logic_vector(31 downto 0);
51      regWriteEn: in  std_logic;
52      clock:     in  std_logic;
53      reset:     in  std_logic;
54      r1:        out std_logic_vector(31 downto 0);
55      r2:        out std_logic_vector(31 downto 0)
56  );
57 end component;
58
59 component CSRs is
60   port(
61     input:      in  std_logic_vector(31 downto 0);
62     CSRWriteEn:    in  std_logic;
63     atomicOpt:    in  std_logic_vector(1 downto 0);
64     CSRSel:      in  std_logic; -- natural (Mario) mejor pasar un
65     bit
66     exceptionStatus: in  std_logic;
67     mcause:      in  std_logic_vector(31 downto 0);
68     clock:       in  std_logic;
69     reset:       in  std_logic;
70     output:      out std_logic_vector(31 downto 0)
71  );
72 end component;
73
74 component ControlUnit is
75   port(
76     instruction:    in  std_logic_vector(31 downto 0);
77     comparison:     in  std_logic_vector(2 downto 0);
78     ALUresult:      in  std_logic_vector(31 downto 0);
79     IRQ:           in  std_logic;
80     reset:          in  std_logic;
81     clock:          in  std_logic;
82     microcode:     out std_logic_vector(23 downto 0);
83     exceptionStatus: out std_logic;
84     mcause:         out std_logic_vector(31 downto 0)
85  );
86 end component;
87
88 component Comparison is
89   port(
90     instruction:    in  std_logic_vector(31 downto 0);
91     r1:            in  std_logic_vector(31 downto 0);
92     r2:            in  std_logic_vector(31 downto 0);
93     comparison:    out std_logic_vector(2 downto 0)
94  );
95 end component;
96
97 component ALU is

```

```

97      port (
98          r1:           in  std_logic_vector(31 downto 0);
99          r2:           in  std_logic_vector(31 downto 0);
100         control:       in  std_logic_vector(3 downto 0);
101         overflow:      out std_logic;
102         resultValue:   out std_logic_vector(31 downto 0)
103     );
104 end component;

105
106 component StoreControl is
107     port(
108         input:        in  std_logic_vector(31 downto 0);
109         instruction: in  std_logic_vector(31 downto 0);
110         output:       out std_logic_vector(31 downto 0)
111     );
112 end component;

113
114 component JumpControl is
115     port(
116         jumpSel:      in  std_logic;
117         PCPlus4:      in  std_logic_vector(31 downto 0);
118         branch:       in  std_logic_vector(31 downto 0);
119         PCSel:        in  std_logic;
120         ALUresult:    in  std_logic_vector(31 downto 0);
121         nextPC:       out std_logic_vector(31 downto 0)
122     );
123 end component;

124
125 component DataMemory is
126     port(
127         writeEn:      in  std_logic;
128         address:     in  std_logic_vector(11 downto 0);
129         dataIn:       in  std_logic_vector(31 downto 0);
130         clock:        in  std_logic;
131         dataOut:      out std_logic_vector(31 downto 0)
132     );
133 end component;

134
135 component GPIO is
136     port(
137         writeEn:      in  std_logic;
138         address:     in  std_logic_vector(1 downto 0);
139         dataIn:       in  std_logic_vector(31 downto 0);
140         reset:        in  std_logic;
141         clock:        in  std_logic;
142         dataOut:      out std_logic_vector(31 downto 0);
143         data:         inout std_logic_vector(31 downto 0)
144     );
145 end component;

146
147 component Timer is

```

```

148     port(
149         address:      in  std_logic_vector(1 downto 0);
150         dataIn:       in  std_logic_vector(31 downto 0);
151         writeEn:      in  std_logic;
152         reset:        in  std_logic;
153         clock:        in  std_logic;
154         dataOut:      out std_logic_vector(31 downto 0);
155         timerInterrupt: out std_logic
156     );
157 end component;
158
159 component ImmSelect is
160     port (
161         input:  in  std_logic_vector(31 downto 0);
162         immSel: in  std_logic_vector(2 downto 0);
163         output: out std_logic_vector(31 downto 0)
164     );
165 end component;
166
167 component ALUControl is
168     port(
169         input:  in  std_logic_vector(31 downto 0);
170         ALUop:   in  std_logic_vector(1 downto 0);
171         output: out std_logic_vector(3 downto 0)
172     );
173 end component;
174
175 component LoadControl is
176     port(
177         MUXOutSig:      in  std_logic_vector(31 downto 0);
178         br:             in  std_logic_vector(31 downto 0);
179         nBits:          in  std_logic_vector(1 downto 0);
180         signedOrUnsigned: in  std_logic;
181         auipc:          in  std_logic;
182         LoadControl:    out std_logic_vector(31 downto 0)
183     );
184 end component;
185
186 end package;

```

ALU.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.ALUPkg.all;
5
6 entity ALU is
7     port (
8         r1:          in  std_logic_vector(31 downto 0);

```

```

9      r2:          in  std_logic_vector(31 downto 0);
10     control:       in  std_logic_vector(3 downto 0);
11     overflow:      out std_logic;
12     resultValue:   out std_logic_vector(31 downto 0)
13   );
14 end ALU;
15
16 architecture ALU_ARCH of ALU is
17
18   signal result: std_logic_vector(32 downto 0);
19
20 begin
21
22   with control
23   select result <= std_logic_vector(add_ins(signed(r1), signed(r2
24   )))      when "0000", -- add
25           std_logic_vector(sub_ins(signed(r1), signed(r2
26   )))      when "1000", -- sub
27           std_logic_vector(sll_ins(unsigned(r1),
28           unsigned(r2)))  when "0001", -- sll
29           std_logic_vector(slt_ins(signed(r1), signed(r2
30   )))      when "0010", -- slt
31           std_logic_vector(sltau_ins(unsigned(r1),
32           unsigned(r2))) when "0011", -- sltu
33           ('0' & (r1 xor r2))
34           when "0100", -- xor
35           std_logic_vector(srli_ins(unsigned(r1),
36           unsigned(r2))) when "0101", -- srl
37           std_logic_vector(sraa_ins(unsigned(r1),
38           unsigned(r2))) when "1101", -- sra
39           ('0' & (r1 or r2))
40           when "0110", -- or
41           ('0' & (r1 and r2))
42           when "0111", -- and
43           ('0' & r2)
44           when "1111", -- lui
45           (others => '1')
46           when others;
47
48   resultValue <= result(31 downto 0);
49   OVERFLOW_DRIVER: process(result)
50   begin
51     if (result(32) = '1') then
52       overflow <= '1';
53     else
54       overflow <= '0';
55     end if;
56   end process OVERFLOW_DRIVER;
57
58 end ALU_ARCH;

```

ALUControl.vhd:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity ALUControl is
5     port(
6         input:  in  std_logic_vector(31 downto 0);
7         ALUop:  in  std_logic_vector(1 downto 0);
8         output: out std_logic_vector(3 downto 0)
9     );
10 end ALUControl;
11
12 architecture ALUControl_ARCH of ALUControl is
13
14     function R_type(  input: std_logic_vector(9 downto 0))
15                     return std_logic_vector is
16         variable output: std_logic_vector(3 downto 0);
17     begin
18
19         case input is
20             when "0000000000" => output := "0000"; -- add
21             when "0100000000" => output := "1000"; -- sub
22             when "0000000001" => output := "0001"; -- sll
23             when "0000000010" => output := "0010"; -- slt
24             when "0000000011" => output := "0011"; -- sltu
25             when "0000000100" => output := "0100"; -- xor
26             when "0000000101" => output := "0101"; -- srl
27             when "0100000101" => output := "1101"; -- sra
28             when "0000000110" => output := "0110"; -- or
29             when "0000000111" => output := "0111"; -- and
30             when others          => output := "0000";
31         end case;
32
33         return (output);
34     end function R_type;
35
36     function I_type(  input: std_logic_vector(31 downto 0))
37                     return std_logic_vector is
38         variable output: std_logic_vector(3 downto 0);
39         variable input_join: std_logic_vector(9 downto 0); -- (Mario)
40         Para juntar bits
41     begin
42         input_join := input(31 downto 25) & input(14 downto 12);
43         if (input(6 downto 0) = "0110111") then -- lui
44             output := "1111";
45             return (output);
46         else
47             case (input(14 downto 12)) is
48                 when "000" => output := "0000"; -- addi
49                 when "010" => output := "0010"; -- slti
```

```

49      when "011" => output := "0011"; -- sltiu
50      when "100" => output := "0100"; -- xor
51      when "110" => output := "0110"; -- ori
52      when "111" => output := "0111"; -- andi
53      when others =>
54          case (input_join) is
55              when "0000000001" => output := "0001"; -- slli
56              when "0000000101" => output := "0101"; -- srli
57              when "0100000101" => output := "1101"; -- srai
58              when others         => output := "0000";
59          end case;
60      end case;
61  end if;

62
63      return (output);
64  end function I_type;

65
66 begin
67
68     with ALUop
69     select output <= "0000"
70         when "00", -- lw or sw
71             "1000"
72         when "01", -- B-type
73             R_type((input(31 downto 25)) & (input(14
74             downto 12))) when "10", -- R-type
75                 I_type(input)
76             when "11", -- I-type or lui
77                 (others => '0')
78             when others;
79
80 end ALUControl_ARCH;

```

ALUPkg.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 package ALUPkg is
6
7     function add_ins( r1: signed(31 downto 0);
8                         r2: signed(31 downto 0))
9                     return signed;
10
11    function sub_ins( r1: signed(31 downto 0);
12                         r2: signed(31 downto 0))
13                     return signed;
14
15    function sll_ins( r1: unsigned(31 downto 0);

```

```

16          r2: unsigned(31 downto 0))
17          return unsigned;
18
19      function slt_ins( r1: signed(31 downto 0);
20                         r2: signed(31 downto 0))
21                         return signed;
22
23      function sltu_ins( r1: unsigned(31 downto 0);
24                         r2: unsigned(31 downto 0))
25                         return unsigned;
26
27      function srl_ins( r1: unsigned(31 downto 0);
28                         r2: unsigned(31 downto 0))
29                         return unsigned;
30
31      function sra_ins( r1: unsigned(31 downto 0);
32                         r2: unsigned(31 downto 0))
33                         return unsigned;
34
35  end package;
36
37 package body ALUPkg is
38
39      function add_ins( r1: signed(31 downto 0);
40                         r2: signed(31 downto 0))
41                         return signed is
42          variable temp: signed(32 downto 0);
43  begin
44      temp := ('0' & r1) + ('0' & r2);
45      return (temp);
46  end function;
47
48      function sub_ins( r1: signed(31 downto 0);
49                         r2: signed(31 downto 0))
50                         return signed is
51          variable temp: signed(32 downto 0);
52  begin
53      temp := ('0' & r1) - ('0' & r2);
54      return (temp);
55  end function;
56
57      function sll_ins( r1: unsigned(31 downto 0);
58                         r2: unsigned(31 downto 0))
59                         return unsigned is
60          variable sizeToShift: integer;
61          variable temp:           unsigned(32 downto 0);
62  begin
63      sizeToShift := to_integer(r2(4 downto 0));
64      temp := '0' & shift_left(r1, sizeToShift);
65      return (temp);
66  end function;

```

```

67
68     function slt_ins( r1: signed(31 downto 0);
69                     r2: signed(31 downto 0))
70                     return signed is
71     variable temp: signed(32 downto 0);
72     begin
73         if (r1 < r2) then
74             temp := (0 => '1', others => '0');
75         else
76             temp := (others => '0');
77         end if;
78         return (temp);
79     end function;
80
81
82     function sltu_ins( r1: unsigned(31 downto 0);
83                         r2: unsigned(31 downto 0))
84                         return unsigned is
85     variable temp: unsigned(32 downto 0);
86     begin
87         if (r1 < r2) then
88             temp := (0 => '1', others => '0');
89         else
90             temp := (others => '0');
91         end if;
92         return (temp);
93     end function;
94
95     function srl_ins( r1: unsigned(31 downto 0);
96                         r2: unsigned(31 downto 0))
97                         return unsigned is
98     variable sizeToShift: integer;
99     variable temp:          unsigned(32 downto 0);
100    begin
101        sizeToShift := to_integer(r2(4 downto 0));
102        temp := '0' & shift_right(r1, sizeToShift);
103        return (temp);
104    end function;
105
106    function sra_ins( r1: unsigned(31 downto 0);
107                      r2: unsigned(31 downto 0))
108                      return unsigned is
109    variable sizeToShift: integer;
110    variable temp:          unsigned(32 downto 0);
111    variable MSBr1:         std_logic;
112    begin
113        sizeToShift := to_integer(r2(4 downto 0));
114        MSBr1 := r1(31);
115        temp := '0' & shift_right(r1, sizeToShift);
116        temp(31 downto (31 - sizeToShift + 1)) := (others => MSBr1);
117        return (temp);
118    end function;

```

```
18
19 end package body;
```

Comparison.vhd:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity Comparison is
6   port(
7     instruction:  in std_logic_vector(31 downto 0);
8     r1:          in std_logic_vector(31 downto 0);
9     r2:          in std_logic_vector(31 downto 0);
10    comparison:  out std_logic_vector(2 downto 0)
11  );
12 end Comparison;
13
14 architecture Comparison_ARCH of Comparison is
15
16  function comparisonft( instruction:  std_logic_vector(31 downto
17    0);
18                      r1:          std_logic_vector(31 downto
19    0);
20                      r2:          std_logic_vector(31 downto
21    0))
22                      return std_logic_vector is
23  variable comp: std_logic_vector(2 downto 0);
24 begin
25
26  if (signed(r1) = signed(r2)) then
27    comp := "000";
28  elsif (signed(r1) < signed(r2)) then
29    comp := "001";
30  elsif (signed(r1) > signed(r2)) then
31    comp := "010";
32  end if;
33  if ((instruction(14 downto 12) = "110") or (instruction(14
34  downto 12) = "111")) then
35    if (unsigned(r1) < unsigned(r2)) then
36      comp := "011";
37    elsif ((unsigned(r1) > unsigned(r2)) or (unsigned(r1) =
38    unsigned(r2))) then
39      comp := "100";
40    end if;
41  end if;
42  return (comp);
43
44 end function comparisonft;
```

```
41 begin  
42  
43     comparison <= comparisonft(instruction, r1, r2);  
44  
45 end Comparison_ARCH;
```

ControlUnit.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.ControlUnitPkg.all;
5
6 entity ControlUnit is
7 port(
8     instruction:      in  std_logic_vector(31 downto 0);
9     comparison:       in  std_logic_vector(2 downto 0);
10    ALUresult:        in  std_logic_vector(31 downto 0);
11    IRQ:              in  std_logic;
12    reset:            in  std_logic;
13    clock:            in  std_logic;
14    microcode:        out std_logic_vector(23 downto 0);
15    exceptionStatus: out std_logic;
16    mcause:           out std_logic_vector(31 downto 0)
17 );
18 end ControlUnit;
19
20 architecture ControlUnit_ARCH of ControlUnit is
21
22    -----state-machine-declarations
23    -----SIGNALS
24    type States_t is (START, FETCH, DECODE, SAVE_TO_REG, SAVE_TO_MEM,
25                      SAVE_TO_REG_AND_CSR);
26    signal currentState: States_t;
27    signal nextState:     States_t;
28
29 begin
30
31    EXCEPTION_STATUS: process(reset, clock)
32    begin
33        if (reset = '1') then
34            exceptionStatus <= '0';
35        elsif (rising_edge(clock)) then
36            if (currentState = START) then
37                exceptionStatus <= '0';
38            elsif (raiseException = '1') then
39                exceptionStatus <= '1';
40            elsif (raiseException = '0') then

```

```

41         exceptionStatus <= '0';
42     end if;
43 end if;
44 end process EXCEPTION_STATUS;
45
46 -----
47
48 PROCESS
49 --
50 -- State Register
51 --
52 -- The State Register is in charge of the synchronous part of the
53 -- FSM.
54 -- It changes the currentState into the nextState at each clock
55 -- pulse.
56 -- If the reset input is active, it will just assign the
57 -- currentState to
58 -- the START state.
59 --
60
61 -----
62
63 STATE_REGISTER: process(reset, clock)
64 begin
65     if (reset = '1') then
66         currentState <= START;
67     elsif (rising_edge(clock)) then
68         currentState <= nextState;
69     end if;
70 end process STATE_REGISTER;
71
72 -----
73
74 PROCESS
75 --
76 -- State Transition
77 --
78 -- The State Transition is in charge of the combinational part of
79 -- the FSM.
80 -- It sets the outputs for each currentState, and assigns the
81 -- nextState
82 -- according to the conditions set on the FSM diagram.
83 --
84
85 -----
86
87 STATE_TRANSITION: process(currentState, instruction, comparison,
88     ALUresult, IRQ)
89     variable tempMicrocode: std_logic_vector(23 downto 0);
90 begin
91
92

```

```

78     microcode <= (others => '0');
79     raiseException <= '0';
80     mcause <= (others => '0');

81
82     if (IRQ = '1') then -- Except. (IRQ)
83         microcode <= (others => '0');
84         raiseException <= '1';
85         mcause <= (4 => '1', others => '0');
86         nextState <= FETCH;
87     else
88         case currentState is
89
-----START
90             when START =>
91                 nextState <= FETCH;
92
-----FETCH
93             when FETCH =>
94                 microcode(17) <= '1'; -- insRegEn
95                 nextState <= DECODE;
96
-----DECODE
97             when DECODE =>
98                 tempMicrocode := decode(instruction, comparison);
99                 if (tempMicrocode = "00000000000000000000000000") then --
100             Except. (Inv. Ins.)
101                 microcode <= (others => '0');
102                 raiseException <= '1';
103                 mcause <= (0 => '1', others => '0');
104                 nextState <= FETCH;
105             else
106                 microcode <= tempMicrocode;
107                 case (instruction(6 downto 0)) is
108                     when "0110011" | "0010011" | "0000011" | "0110111" |
109                         "1100111" | "1101111" | "0010111" =>
110                         --      R      I      I (loads)      U (lui)      I
111                         (jalr)   J (jal)   U (auipc)
112                         microcode(18) <= '0'; -- PCEn
113                         if instruction(6 downto 0) = "1100111" or      -- I (
114                             jalr)
115                             instruction(6 downto 0) = "1101111" or      -- J (
116                             jal)
117                             instruction(6 downto 0) = "0010111" then      -- U (
118                             auipc)
119                             microcode(11) <= '1'; -- regWriteEn
120                         end if;
121                         nextState <= SAVE_TO_REG;
122                         when "0100011" => -- S

```

```

17         microcode(18) <= '0'; -- PCEn
18         nextState <= SAVE_TO_MEM;
19         when "1110011" => -- Atomic
20             microcode(18) <= '0'; -- PCEn
21             nextState <= SAVE_TO_REG_AND_CSR;
22             when "1100011" => -- B
23                 microcode(18) <= '1'; -- PCEn
24                 nextState <= FETCH;
25             when others => -- Except.
26                 microcode <= (others => '0');
27                 raiseException <= '1';
28                 mcause <= (0 => '1', others => '0');
29                 nextState <= FETCH;
30             end case;
31         end if;

32
-----  

33     SAVE_TO_REG
34         when SAVE_TO_REG =>
35             microcode <= decode(instruction, comparison);
36             if ((instruction(6 downto 0) = "1100111") or      -- I-
37 type jalr
38                 (instruction(6 downto 0) = "1101111") or      -- J-
39 type (jal)
40                 (instruction(6 downto 0) = "0010111")) then    -- U-
41 type auipc
42             microcode(11) <= '0'; -- regWriteEn
43             else
44                 microcode(11) <= '1'; -- regWriteEn
45             end if;
46             microcode(18) <= '1'; -- PCEn
47             nextState <= FETCH;
48
-----  

49     SAVE_TO_MEM
50         when SAVE_TO_MEM =>
51             microcode <= decode(instruction, comparison);
52             microcode(5) <= '1'; -- memWriteEn
53             microcode(18) <= '1'; -- PCEn
54             nextState <= FETCH;
55
-----  

56     SAVE_TO_REG_AND_CSR
57         when SAVE_TO_REG_AND_CSR =>
58             microcode <= decode(instruction, comparison);
59             microcode(11) <= '1';
60             microcode(23) <= '1';
61             microcode(18) <= '1'; -- PCEn
62             nextState <= FETCH;
63         when others => -- Except.
64             microcode <= (others => '0');
65             raiseException <= '1';

```

```

160         mcause <= (0 => '1', others => '0');
161         nextState <= FETCH;
162     end case;
163 end if;

164

165 if ((instruction(6 downto 0) /= "0110011") and      -- R-type
166     (instruction(6 downto 0) /= "0110111") and      -- U-type lui
167     (instruction(6 downto 0) /= "0010011")) then    -- I-type
168     if ((unsigned(ALUresult) >= 0) and (unsigned(ALUresult) <= x"-
169 FFF")) then           -- Data memory
170         microcode(4 downto 3) <= "00";
171     elsif ((unsigned(ALUresult) >= x"1000") and (unsigned(
172 ALUresult) <= x"1002")) then    -- GPIO
173         microcode(4 downto 3) <= "10";
174     elsif ((unsigned(ALUresult) >= x"3000") and (unsigned(
175 ALUresult) <= x"3002")) then    -- Timer
176         microcode(4 downto 3) <= "11";
177     end if;
178 end if;

179
180 end process STATE_TRANSITION;
181
182 end ControlUnit_ARCH;

```

ControlUnitPkg.vhd:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 package ControlUnitPkg is
5
6     function decode( instruction: std_logic_vector(31 downto 0);
7                         comparison: std_logic_vector(2 downto 0))
8                         return std_logic_vector;
9
10 end package;
11
12 package body ControlUnitPkg is
13
14     function decode( instruction: std_logic_vector(31 downto 0);
15                         comparison: std_logic_vector(2 downto 0))
16                         return std_logic_vector is
17
18         variable microcode: std_logic_vector(23 downto 0);
19
20     begin
21         case instruction(6 downto 0) is
22             when "0110011" => microcode := "0000001100000001100001100";
-- R-type
23             when "0110111" => microcode := "000001011011101000001100";
--
```

```

1      U-type lui
2      when "0010111" => microcode := "000010000011001000000100"; --
3      U-type auipc
4      when "1101111" => microcode := "000000000100000001000000";
5      J-type (jal)
6      when "0000011" => -- I-type Loads
7      case instruction(14 downto 12) is
8          when "000" => microcode := "00000100000010100000001"; --
9          lb
10         when "001" => microcode := "000001000000101000000011"; --
11         lh
12         when "010" => microcode := "0000010000001010000000101"; --
13         lw
14         when "100" => microcode := "0000010000001010000000000"; --
15         lbu
16         when "101" => microcode := "0000010000001010000000010"; --
17         lhu
18         when others => microcode := (others => '0');
19     end case;
20     when "1100111" => microcode := "000000011000000010000100"; --
21     I-type jalr
22     when "0010011" => microcode := "000000011000001000001100"; --
23     I-type
24     when "0100011" => -- S-type
25     case instruction(14 downto 12) is
26         when "000" => microcode := "000000000001000000000000100"; --
27         sb
28         when "001" => microcode := "000000000001000000000000100"; --
29         sh
30         when "010" => microcode := "000000000001000000000000100"; --
31         sw
32         when others => microcode := (others => '0');
33     end case;
34     when "1100011" => -- B-type
35     case instruction(14 downto 12) is
36         when "000" => -- beq
37             if (comparison = "000") then -- rs1 == rs2
38                 microcode := "000001001001000101000100";
39             else -- rs1 != rs2
40                 microcode := "000001001001000100000100";
41             end if;
42         when "001" => -- bne
43             if (comparison = "000") then -- rs1 == rs2
44                 microcode := "000001001001000100000100";
45             else -- rs1 != rs2
46                 microcode := "000001001001000101000100";
47             end if;
48         when "100" => -- blt
49             if (comparison = "001") then -- rs1 < rs2
50                 microcode := "00000100100100001000100";
51             else -- rs1 >= rs2
52                 microcode := "00000100100100001000100";
53             end if;
54         when "101" => -- bge
55             if (comparison = "000") then -- rs1 >= rs2
56                 microcode := "00000100100100001000100";
57             else -- rs1 < rs2
58                 microcode := "00000100100100001000100";
59             end if;
60         when "110" => -- bne
61             if (comparison = "000") then -- rs1 != rs2

```

```

62         microcode := "000001001001000000000100";
63     end if;
64     when "101" => -- bge
65         if ((comparison = "010") or (comparison = "000")) then
66             -- rs1 >= rs2
67                 microcode := "000001001001000001000100";
68             else -- rs1 < rs2
69                 microcode := "000001001001000000000100";
70             end if;
71     when "110" => -- bltu
72         if (comparison = "011") then -- u(rs1) < u(rs2)
73             microcode := "000001001001000001000100";
74         else -- u(rs1) >= u(rs2)
75             microcode := "000001001001000000000100";
76         end if;
77     when "111" => -- bgeu
78         if (comparison = "100") then -- u(rs1) >= u(rs2)
79             microcode := "000001001001000001000100";
80         else -- u(rs1) < u(rs2)
81             microcode := "000001001001000000000100";
82         end if;
83     when others =>
84         microcode := (others => '0');
85     end case;
86 when "1110011" => -- ATOMIC INS.
87     case instruction(14 downto 12) is
88         when "001" => -- csrrw
89             microcode := "0000000000000001000000000000";
90         when "010" => -- csrrs
91             microcode := "0010000000000001000000000000";
92         when "011" => -- csrrc
93             microcode := "0100000000000001000000000000";
94         when "101" => -- csrrwi
95             microcode := "00010000010101000000000000";
96         when "110" => -- csrrsi
97             microcode := "00110000010101000000000000";
98         when "111" => -- csrrci
99             microcode := "01010000010101000000000000";
100        when others =>
101            microcode := (others => '0');
102        end case;
103    when others => microcode := (others => '0');
104    end case;
105    return (microcode);
106    end function decode;
107 end package body;

```

CSRs.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity CSRs is
6   port(
7     input:      in std_logic_vector(31 downto 0);
8     CSRWriteEn:    in std_logic;
9     atomicOpt:    in std_logic_vector(1 downto 0);
10    CSRSel:      in std_logic; -- natural (Mario) mejor pasar un bit
11    exceptionStatus: in std_logic;
12    mcause:      in std_logic_vector(31 downto 0);
13    clock:       in std_logic;
14    reset:       in std_logic;
15    output:      out std_logic_vector(31 downto 0)
16  );
17 end CSRs;
18
19 architecture CSRs_ARCH of CSRs is
20
21  -- Register 0: mtvec
22  -- Register 1: mcause
23  type ram_type is array(0 to 1) of std_logic_vector(31 downto 0);
24  signal ram: ram_type := (others => (others => '0'));
25
26 begin
27
28  -- Write to a CSR
29  WRITE_TO_CSR: process(reset, clock)
30  begin
31    if (reset = '1') then
32      ram(0) <= (2 => '1', others => '0'); -- mtvec
33      ram(1) <= (others => '0');           -- mcause
34    elsif (rising_edge(clock)) then
35      if (CSRWriteEn = '1') then
36        if CSRSel='0' then
37          case atomicOpt is
38            when "00" => ram(0) <= input;           -- Read
39            and Write
40            when "01" => ram(0) <= ram(0) or input; -- Read and
41            Set
42            when "10" => ram(0) <= ram(0) and (not input); -- Read and
43            Clear
44            when others =>
45            end case;
46          else
47            case atomicOpt is
48              when "00" => ram(1) <= input;           -- Read
49              and Write
50              when "01" => ram(1) <= ram(1) or input; -- Read and
51              Set

```

```

47         when "10" => ram(1) <= ram(1) and (not input); -- Read and
48             Clear
49             when others =>
50                 end case;
51             end if;
52 --                                         (Mario) He deshecho los dos casos
53 --     para no tene que usar un indice
54 --         case atomicOpt is
55 --             when "00" => ram(CSRSel) <= input;
56 --             -- Read and Write
57 --             when "01" => ram(CSRSel) <= ram(CSRSel) or input;
58 --             -- Read and Set
59 --             when "10" => ram(CSRSel) <= ram(CSRSel) and (not input);
60 --             -- Read and Clear
61 --             when others =>
62 --                 end case;
63             end if;
64             if (exceptionstatus = '1') then
65                 ram(1) <= mcause;
66             end if;
67         end if;
68     end process WRITE_TO_CSR;
69
70     -- Read from a CSR
71     output <= ram(0) when CSRSel='0' else ram(1); -- (Mario)
72
73 end architecture CSRs_ARCH;

```

dataMemory.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity DataMemory is
6     generic(
7         RAM_SIZE: integer := 2 ** 12 -- 4096
8     );
9     port(
10         writeEn: in std_logic;
11         address: in std_logic_vector(11 downto 0);
12         dataIn: in std_logic_vector(31 downto 0);
13         clock: in std_logic;
14         dataOut: out std_logic_vector(31 downto 0)
15     );
16 end DataMemory;
17
18 architecture DataMemory_ARCH of DataMemory is
19
20     type ram_type is array(0 to (RAM_SIZE) - 1) of std_logic_vector(31

```

```

      downto 0);
21 signal ram: ram_type := (others => (others => '0'));
22
23 begin
24
25 -- Store: Write register value to memory.
26 STORE: process(clock)
27 begin
28   if (rising_edge(clock)) then
29     if (writeEn = '1') then
30       ram(to_integer(unsigned(address))) <= dataIn;
31     end if;
32   end if;
33 end process STORE;
34
35
36 -- Load: Read memory and update register.
37 LOAD: process(address,ram)
38 begin
39   if ((to_integer(unsigned(address)) >= 0) and
40     (to_integer(unsigned(address)) <= (RAM_SIZE - 1))) then
41     dataOut <= ram(to_integer(unsigned(address)));
42   else
43     dataOut <= (others => '0');
44   end if;
45 end process LOAD;
46
47 end architecture DataMemory_ARCH;

```

ExceptionControl.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity ExceptionControl is
5   port(
6     input:          in  std_logic_vector(31 downto 0);
7     exceptionStatus: in  std_logic;
8     output:         out std_logic_vector(31 downto 0)
9   );
10 end ExceptionControl;
11
12 architecture ExceptionControl_ARCH of ExceptionControl is
13 begin
14
15   EXCEPTIONCNTRL: process(input, exceptionStatus)
16   begin
17     if (exceptionStatus = '1') then
18       output <= "0000000001000000000001011100111"; -- jalr x5, 4(x0
19   )

```

```

19     else
20         output <= input;
21     end if;
22
23 end process EXCEPTIONCNTRL;
24
25 end ExceptionControl_ARCH;

```

GPIO.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity GPIO is
6     port(
7         writeEn:      in      std_logic;
8         address:      in      std_logic_vector(1 downto 0);
9         dataIn:       in      std_logic_vector(31 downto 0);
10        reset:       in      std_logic;
11        clock:       in      std_logic;
12        dataOut:      out     std_logic_vector(31 downto 0);
13        data:        inout   std_logic_vector(31 downto 0)
14    );
15 end GPIO;
16
17 architecture GPIO_ARCH of GPIO is
18
19     component singleRegister is
20         generic(
21             REGSIZE: natural
22         );
23         port(
24             input:       in      std_logic_vector(REGSIZE - 1 downto 0);
25             writeEn:     in      std_logic;
26             clock:      in      std_logic;
27             reset:      in      std_logic;
28             output:     out     std_logic_vector(REGSIZE - 1 downto 0)
29         );
30     end component;
31
32     signal r_WriteEn: std_logic_vector(1 downto 0);
33     signal dOut:      std_logic_vector(31 downto 0);
34     signal tri:       std_logic_vector(31 downto 0);
35
36
37 begin
38
39     DATOSOUT: singleRegister
40         generic map(32)

```

```

41  port map(
42      input => dataIn,
43      writeEn => r_WriteEn(0),
44      reset => reset,
45      clock => clock,
46      output => dOut
47  );
48
49 DATOSIN: singleRegister
50     generic map(32)
51     port map(
52         input => data,
53         writeEn => '1',
54         reset => reset,
55         clock => clock,
56         output => dataOut
57     );
58
59 TRISTATE: singleRegister
60     generic map(32)
61     port map(
62         input => dataIn,
63         writeEn => r_WriteEn(1),
64         reset => reset, -- Todos a 0, entrada por defecto. Evita
65         cortos.
66         clock => clock,
67         output => tri
68     );
69
70 r_WriteEn(0) <= '1' when address="00" and writeEn = '1' else '0';
71     -- writeEn for DatosOut reg.
72 r_WriteEn(1) <= '1' when address="10" and writeEn = '1' else '0';
73     -- writeEn for TRI reg.
74
75 PINS: process(tri, dOut)
76 begin
77     for i in 0 to 31 loop
78         if (tri(i) = '1') then
79             data(i) <= dOut(i);
80         else
81             data(i) <= 'Z';
82         end if;
83     end loop;
84 end process PINS;
85
86 end architecture GPIO_ARCH;

```

ImmSelect.vhd:

```

1 library ieee;
```

```

2 use ieee.std_logic_1164.all;
3 use work.ImmSelectPkg.all;
4
5 entity ImmSelect is
6   port (
7     input:  in  std_logic_vector(31 downto 0);
8     immSel: in  std_logic_vector(2 downto 0);
9     output: out std_logic_vector(31 downto 0)
10    );
11 end ImmSelect;
12
13 architecture ImmSelect_ARCH of ImmSelect is
14
15 begin
16
17   with immSel
18   select output <=  I_type(input) when "000", -- I-type
19                           B_type(input) when "001", -- B-type
20                           S_type(input) when "010", -- S-type
21                           U_type(input) when "011", -- U-type
22                           J_type(input) when "100", -- J-type
23                           Atomic(input) when "101", -- Atomic ins.
24                           (others => '0') when others;
25
26 end ImmSelect_ARCH;

```

ImmSelectPkg.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 package ImmSelectPkg is
5
6   function I_type(  input: std_logic_vector(31 downto 0))
7                     return std_logic_vector;
8
9   function B_type(  input: std_logic_vector(31 downto 0))
10                    return std_logic_vector;
11
12  function S_type(  input: std_logic_vector(31 downto 0))
13                    return std_logic_vector;
14
15  function U_type(  input: std_logic_vector(31 downto 0))
16                    return std_logic_vector;
17
18  function J_type(  input: std_logic_vector(31 downto 0))
19                    return std_logic_vector;
20
21  function Atomic(  input: std_logic_vector(31 downto 0))
22                    return std_logic_vector;

```

```

23
24 end package;
25
26 package body ImmSelectPkg is
27
28     function I_type( input: std_logic_vector(31 downto 0))
29             return std_logic_vector is
30         variable ImmValue: std_logic_vector(31 downto 0) := (others =>
31             input(31));
32         begin
33             ImmValue(11 downto 0) := input(31 downto 20);
34             return ImmValue;
35         end function;
36
37     function B_type( input: std_logic_vector(31 downto 0))
38             return std_logic_vector is
39         variable ImmValue: std_logic_vector(31 downto 0) := (others =>
40             input(31));
41         begin
42             ImmValue(11) := input(7);
43             ImmValue(10 downto 5) := input(30 downto 25);
44             ImmValue(4 downto 1) := input(11 downto 8);
45             ImmValue(0) := '0';
46             return (ImmValue);
47         end function;
48
49     function S_type( input:std_logic_vector(31 downto 0))
50             return std_logic_vector is
51         variable ImmValue: std_logic_vector(31 downto 0) := (others =>
52             input(31));
53         begin
54             ImmValue(11 downto 5) := input(31 downto 25);
55             ImmValue(4 downto 0) := input(11 downto 7);
56             return (ImmValue);
57         end function;
58
59     function U_type( input: std_logic_vector(31 downto 0))
60             return std_logic_vector is
61         variable ImmValue: std_logic_vector(31 downto 0) := (others =>
62             '0');
63         begin
64             ImmValue(31 downto 12) := input(31 downto 12);
65             return ImmValue;
66         end function;
67
68     function J_type( input: std_logic_vector(31 downto 0))
69             return std_logic_vector is
70         variable ImmValue: std_logic_vector(31 downto 0) := (others =>
71             '0');
72         begin
73             ImmValue(31 downto 20) := (others => input(31));

```

```

69     ImmValue(19 downto 12) := input(19 downto 12);
70     ImmValue(11) := input(20);
71     ImmValue(10 downto 1) := input(30 downto 21);
72     return ImmValue;
73 end function;
74
75 function Atomic( input: std_logic_vector(31 downto 0))
76     return std_logic_vector is
77     variable ImmValue: std_logic_vector(31 downto 0) := (others =>
78         '0');
79 begin
80     ImmValue(4 downto 0) := input(19 downto 15);
81     return ImmValue;
82 end function;
83
84 end package body;

```

InstructionMemory.vhd:

```

1 library ieee, std;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use ieee.std_logic_textio.all;
5 use std.textio.all;
6
7 entity InstructionMemory is
8     generic(
9         INS_MEM_SIZE:      integer := 2 ** 10; -- 1024
10        TEXT_FILE:        string := "EXInsertionSort.mem"
11    );
12    port(
13        readAddress:      in  std_logic_vector(31 downto 0);
14        instruction:     out std_logic_vector(31 downto 0)
15    );
16 end InstructionMemory;
17
18 architecture InstructionMemory_ARCH of InstructionMemory is
19
20     -- Big endian
21     type ram_type is array(0 to (INS_MEM_SIZE / 4) - 1) of
22         std_logic_vector(31 downto 0);
23     signal ram: ram_type :=(
24         0 => "00000000"&"00000000"&"00000100"&"01100011", -- beq x0, x0,
25         8
26         1 => "00000000"&"00000010"&"10000011"&"01100111", -- jalr x6, 0(
27         x5)
28         others => (others => '0')
29    );
30
31 begin

```

```

29
30     BUILD_MEM: process
31
32         file      input_file: text;
33         variable  input_line: line;
34         variable  value:      std_logic_vector(31 downto 0);
35         variable  i:          integer;
36
37     begin
38
39         file_open(input_file, TEXT_FILE, read_mode);
40
41         i := 2;
42         while not endfile(input_file) loop
43             readline(input_file, input_line);
44             read(input_line, value);
45             ram(i) <= value;
46             i := i + 1;
47         end loop;
48         wait;
49
50     end process BUILD_MEM;
51
52     instruction <= ram(to_integer(unsigned(readAddress(31 downto 2)));
53
54 end InstructionMemory_ARCH;

```

InstructionRegister.vhd:

```

1 -----
2 -- Company:
3 -- Engineer:
4 --
5 -- Create Date: 08.02.2023 22:25:31
6 -- Design Name:
7 -- Module Name: InstructionRegister - Behavioral
8 -- Project Name:
9 -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 -----

```

```

20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 --use IEEE.NUMERIC_STD.ALL;
28
29 -- Uncomment the following library declaration if instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
32 --use UNISIM.VComponents.all;
33
34 entity InstructionRegister is
35 -- Port ();
36 end InstructionRegister;
37
38 architecture Behavioral of InstructionRegister is
39
40 begin
41
42
43 end Behavioral;

```

JumpControl.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity JumpControl is
6   port(
7     jumpSel:      in  std_logic;
8     PCPlus4:      in  std_logic_vector(31 downto 0);
9     branch:       in  std_logic_vector(31 downto 0);
10    PCSel:        in  std_logic;
11    ALUresult:   in  std_logic_vector(31 downto 0);
12    nextPC:      out std_logic_vector(31 downto 0)
13  );
14 end JumpControl;
15
16 architecture JumpControl_ARCH of JumpControl is
17
18   function sel( jumpSel:      std_logic;
19                 PCPlus4:      std_logic_vector(31 downto 0);
20                 branch:       std_logic_vector(31 downto 0);
21                 PCSel:        std_logic;
22                 ALUresult:   std_logic_vector(31 downto 0))

```

```

23         return std_logic_vector is
24     variable vector: std_logic_vector(31 downto 0) := (others =>
25         '0');
26     begin
27         -- (Mario) Mejor no comprobar el limite de memoria
28         if (jumpSel = '1') then
29             vector := ALUresult;
30         elsif (PCSel = '0') then
31             --if (unsigned(PCPlus4) > ((2 ** 10) - 4)) then -- Ins. Mem.
32             Limit
33                 -- vector := (others => '0');
34             --else
35                 vector := PCPlus4;
36             --end if;
37         else
38             vector := branch;
39         end if;
40
41         return (vector);
42     end function;
43
44 begin
45
46     nextPC <= sel(jumpSel, PCPlus4, branch, PCSel, ALUresult);
47
48 end JumpControl_ARCH;

```

LoadControl.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity LoadControl is
6     port(
7         MUXOutSig:      in  std_logic_vector(31 downto 0);
8         br:            in  std_logic_vector(31 downto 0);
9         nBits:          in  std_logic_vector(1 downto 0);
10        signedOrUnsigned: in  std_logic;
11        auipc:          in  std_logic;
12        LoadControl:    out std_logic_vector(31 downto 0)
13    );
14 end LoadControl;
15
16 architecture LoadControl_ARCH of LoadControl is
17
18     signal nOfBits: natural;
19

```

```

20 --function loadft( MUXOutSig:          std_logic_vector(31 downto
0);
21 --                                br:          std_logic_vector(31 downto
0);
22 --                                nOfBits:      natural;
23 --                                signedOrUnsigned: std_logic;
24 --                                auipc:        std_logic)
25 --                                return std_logic_vector is
26 -- variable vector: std_logic_vector(31 downto 0);
27 --begin
28 --
29 --  if (auipc = '1') then
30 --    vector := br;
31 --  elsif (signedOrUnsigned = '0') then -- zx
32 --    vector := std_logic_vector(resize(unsigned(MUXOutSig(nOfBits -
33 -- 1 downto 0)), 32));
34 --  else
35 --    vector := std_logic_vector(resize(signed(MUXOutSig(nOfBits -
36 -- 1 downto 0)), 32));
37 --  end if;
38 --
39 --  return (vector);
40 --
41 begin
42 
43 --  with nBits
44 --    select nOfBits <= 8 when "00",
45 --                      16 when "01",
46 --                      32 when "10",
47 --                      32 when others;
48 
49 --  LoadControl <= loadft(MUXOutSig, br, nOfBits, signedOrUnsigned,
50 --  auipc);
51 
52 -- (Mario) He reh echo la func i n , poniendo valores fijos en los
53 -- rangos
54 process(nBits,auipc,MUXOutSig,br,signedOrUnsigned)
55 begin
56   if (auipc = '1') then
57     LoadControl <= br;
58   else
59     case nBits is
60       when "00" =>
61         if (signedOrUnsigned = '0') then
62           LoadControl <= std_logic_vector(resize(unsigned(MUXOutSig(
63 -- 7 downto 0)), 32));
64         else
65           LoadControl <= std_logic_vector(resize( signed(MUXOutSig(
66 -- 7 downto 0)), 32));

```

```

63         end if;
64     when "01" =>
65         if (signedOrUnsigned = '0') then
66             LoadControl <= std_logic_vector(resize(unsigned(MUXOutSig
67             (15 downto 0)), 32));
68         else
69             LoadControl <= std_logic_vector(resize( signed(MUXOutSig
70             (15 downto 0)), 32));
71         end if;
72     when others =>
73         LoadControl <= MUXOutSig;
74     end case;
75 end if;
76 end process;
77
78 end LoadControl_ARCH;

```

ProgramCounter.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity ProgramCounter is
6     port(
7         nextAddress:    in std_logic_vector(31 downto 0);
8         PCEn:          in std_logic;
9         reset:          in std_logic;
10        clock:          in std_logic;
11        currentAddress: out std_logic_vector(31 downto 0)
12    );
13 end ProgramCounter;
14
15 architecture ProgramCounter_ARCH of ProgramCounter is
16
17     signal counter: std_logic_vector(31 downto 0);
18
19 begin
20
21     PC_DRIVER: process(clock, reset)
22     begin
23         if (reset = '1') then
24             counter <= (others => '0');
25         elsif (rising_edge(clock)) then
26             if (PCEn = '1') then
27                 counter <= nextAddress;
28             end if;
29         end if;
30     end process;
31

```

```

32   currentAddress <= counter;
33
34 end ProgramCounter_ARCH;

```

RegisterFile.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity RegisterFile is
6   port(
7     rs1:      in  unsigned(4 downto 0);
8     rs2:      in  unsigned(4 downto 0);
9     rd:       in  unsigned(4 downto 0);
10    writeData: in  std_logic_vector(31 downto 0);
11    regWriteEn: in  std_logic;
12    clock:     in  std_logic;
13    reset:     in  std_logic;
14    r1:        out std_logic_vector(31 downto 0);
15    r2:        out std_logic_vector(31 downto 0)
16  );
17 end RegisterFile;
18
19 architecture RegisterFile_ARCH of RegisterFile is
20
21  type t_Input is array (31 downto 0) of std_logic_vector(31 downto
22    0);
22  signal r_Input: t_Input;
23
24  type t_WriteEn is array (31 downto 0) of std_logic;
25  signal r_WriteEn: t_WriteEn;
26
27  type t_Output is array (31 downto 0) of std_logic_vector(31 downto
28    0);
28  signal r_Output: t_Output;
29
30  component singleRegister is
31    generic(
32      REGSIZE: natural
33    );
34    port(
35      input:   in  std_logic_vector(REGSIZE - 1 downto 0);
36      writeEn: in  std_logic;
37      reset:   in  std_logic;
38      clock:   in  std_logic;
39      output:  out std_logic_vector(REGSIZE - 1 downto 0)
40    );
41  end component;
42

```

```

43 begin
44
45 GENERATE_32_REGISTERS: for i in 0 to 31 generate
46     REGX: singleRegister
47         generic map(
48             REGSIZE => 32
49         )
50         port map(
51             input => r_Input(i),
52             writeEn => r_WriteEn(i),
53             reset => reset,
54             clock => clock,
55             output => r_Output(i)
56         );
57 end generate GENERATE_32_REGISTERS;
58
59 DEMUX: process(rd, writeData, regWriteEn)
60 begin
61     for i in 1 to 31 loop
62         r_Input(i) <= writeData;
63         if (to_integer(rd) = i) then
64             r_WriteEn(i) <= regWriteEn;
65         else
66             r_WriteEn(i) <= '0';
67         end if;
68     end loop;
69 end process DEMUX;
70
71 r1 <= r_Output(to_integer(rs1));
72 r2 <= r_Output(to_integer(rs2));
73
74 end RegisterFile_ARCH;

```

singleRegister.vhd:

```

1 -- ****
2 --
3 --* Name: singleRegister
4 --* Designer: Alberto Caravantes
5 --
6 --
7 --
8 -- ****
9
10 library ieee;
11 use ieee.std_logic_1164.all;
12 use ieee.numeric_std.all;
13 use work.BasicPkg.all;

```

```

14
15 entity singleRegister is
16   port(
17     input:      in std_logic_vector(31 downto 0);
18     writeEn:    in std_logic;
19     clock:      in std_logic;
20     reset:      in std_logic;
21     output:     out std_logic_vector(31 downto 0)
22   );
23 end singleRegister;
24
25 architecture singleRegister_ARCH of singleRegister is
26
27 begin
28
29   SINGLEREGISTER_DRIVER: process(reset, clock)
30   begin
31     if (reset = '1') then
32       output <= (others => '0');
33     elsif (rising_edge(clock)) then
34       if (writeEn = '1') then
35         output <= input;
36       end if;
37     end if;
38   end process SINGLEREGISTER_DRIVER;
39
40 end singleRegister_ARCH;

```

StoreControl.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity StoreControl is
5   port(
6     input:      in std_logic_vector(31 downto 0);
7     instruction: in std_logic_vector(31 downto 0);
8     output:     out std_logic_vector(31 downto 0)
9   );
10 end StoreControl;
11
12 architecture StoreControl_ARCH of StoreControl is
13
14   function storeft( input:      std_logic_vector(31 downto 0);
15                      instruction: std_logic_vector(31 downto 0))
16                      return std_logic_vector is
17     variable vector: std_logic_vector(31 downto 0) := (others =>
18               '0');
19   begin
20     if (instruction(6 downto 0) = "0100011") then      -- Store

```

```

Instruction
20      if (instruction(14 downto 12) = "000") then      -- sb
21          vector(7 downto 0) := input(7 downto 0);
22      elsif (instruction(14 downto 12) = "001") then   -- sh
23          vector(15 downto 0) := input(15 downto 0);
24      else
25          vector := input;
26      end if;
27      else -- (Mario) falta el else
28          vector := input;
29      end if;
30      return (vector);
31  end function;

32
33 begin
34
35     output <= storeft(input, instruction);
36
37 end StoreControl_ARCH;

```

Timer.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity Timer is
6     port(
7         address:      in std_logic_vector(1 downto 0);
8         dataIn:        in std_logic_vector(31 downto 0);
9         writeEn:       in std_logic;
10        reset:        in std_logic;
11        clock:        in std_logic;
12        dataOut:       out std_logic_vector(31 downto 0);
13        timerInterrupt: out std_logic
14    );
15 end Timer;
16
17 architecture Timer_ARCH of Timer is
18
19     component singleRegister is
20         generic(  

21             REGSIZE: natural  

22         );  

23         port(  

24             input:      in std_logic_vector(REGSIZE - 1 downto 0);  

25             writeEn:    in std_logic;  

26             clock:     in std_logic;  

27             reset:     in std_logic;  

28             output:    out std_logic_vector(REGSIZE - 1 downto 0)

```

```

29 );
30 end component singleRegister;
31
32 signal r_WriteEn: std_logic_vector(1 downto 0);
33 signal controlSig: std_logic_vector(31 downto 0);
34 signal arrSig: std_logic_vector(31 downto 0);
35 signal stateSig: std_logic_vector(31 downto 0) := (others =>
36      '0');
37 signal cntValue: unsigned(31 downto 0) := (others => '0');
38
39 begin
40   CONTROL: singleRegister
41     generic map(32)
42     port map(
43       input => dataIn,
44       writeEn => r_WriteEn(0),
45       reset => reset,
46       clock => clock,
47       output => controlSig
48 );
49
50   ARR: singleRegister
51     generic map(32)
52     port map(
53       input => dataIn,
54       writeEn => r_WriteEn(1),
55       reset => reset,
56       clock => clock,
57       output => arrSig
58 );
59
60
61   STATE: singleRegister
62     generic map(32)
63     port map(
64       input => stateSig,
65       writeEn => '1',
66       reset => reset,
67       clock => clock
68 );
69
70   r_WriteEn(0) <= '1' when ((to_integer(unsigned(address)) = 0) and
71     (writeEn = '1')) else '0'; -- writeEn for Control reg.
72   r_WriteEn(1) <= '1' when ((to_integer(unsigned(address)) = 1) and
73     (writeEn = '1')) else '0'; -- writeEn for ARR reg.
74
75   CNT: process(clock, reset)
76 begin
77   if (reset = '1') then
78     cntValue <= (others => '0');

```

```

77      stateSig <= (others => '0');
78      timerInterrupt <= '0';
79      elsif (rising_edge(clock)) then
80          timerInterrupt <= '0';
81          if (controlSig(1) = '1') then -- Clear
82              cntValue <= (others => '0');
83              stateSig(0) <= '0';
84          elsif (controlSig(0) = '1') then -- Enable
85              if (((cntValue + 1) = unsigned(arrSig)) and (stateSig(0) =
86                  '0')) then -- Counter reaches arrSig
87                  stateSig <= (@=>'1', others=>'0');
88                  if (controlSig(2) = '1') then -- En_Interrupt
89                      timerInterrupt <= '1';
90                  end if;
91                  if (controlSig(3) = '1') then -- Autoreload
92                      cntValue <= (others => '0');
93                  end if;
94                  elsif ((cntValue + 1) /= unsigned(arrSig)) then
95                      cntValue <= cntValue + 1;
96                  end if;
97              end if;
98          end process CNT;
99
100     dataOut <= std_logic_vector(cntValue);
101
102 end Timer_ARCH;

```

RISCV_CPU_TB.vhd:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity RISCV_CPU_TB is
5 end RISCV_CPU_TB;
6
7 architecture RISCV_CPU_TB_ARC of RISCV_CPU_TB is
8
9     component RISCV_CPU is
10         port(
11             clock:      in std_logic;
12             reset:      in std_logic
13         );
14     end component;
15
16     signal clock: std_logic;
17     signal reset: std_logic;
18
19 begin
20

```

```
21  UUT: RISCV_CPU
22    port map(
23      clock => clock,
24      reset => reset
25    );
26
27  SYS_CLOCK: process
28  begin
29    clock <= '0';
30    wait for 5ns;
31    clock <= '1';
32    wait for 5 ns;
33  end process SYS_CLOCK;
34
35  SYS_RESET: process
36  begin
37    reset <= '1';
38    wait for 100 ns;
39    reset <= '0';
40    wait;
41  end process SYS_RESET;
42
43  RISCV_CPU_DRIVER: process
44  begin
45    reset <= '1';
46    wait;
47  end process;
48
49 end RISCV_CPU_TB_ARCH;
```

ANEXO C. CÓDIGO DEL ENSAMBLADOR

Makefile

```
NAME      = ensamblador_riscv

SRCS      = main.cpp \
            ./srcs/generate_mc.cpp \
            ./srcs/open_files.cpp \
            ./srcs/signed_to_binary.cpp

OBJS      = $(SRCS:.cpp=.o)

CXX       = clang++
RM        = rm -f
CXXFLAGS  = -Wall -Wextra -Werror -g -std=c++11 -pedantic

all:      $(NAME)

$(NAME):   $(OBJS)
           $(CXX) $(CXXFLAGS) -g -o $(NAME) $(OBJS)

clean:    $(RM) $(OBJS)

fclean:   clean
           $(RM) $(NAME)

re:       fclean $(NAME)

test:     $(NAME)

.PHONY:    all clean fclean re test
```

ASM_to_MC.hpp

```
1 #ifndef ASM_TO_MC_HPP
2 # define ASM_TO_MC_HPP
3
4 # include <iostream>
5 # include <fstream>
6 # include <string>
7 # include <vector>
8 # include <array>
9 # include <algorithm>
10 # include <stdbool.h>
11 # include <bitset>
12
```

```

13 enum instruction_type { U, J, I_Jalr, B, I_Loads, S, I, I_Shifts, R,
14     I_Atomic };
15
16 typedef struct s_isa
17 {
18     std::string ins_name;
19 } t_isa;
20
21 static std::vector<t_isa> isa = \
22 { //func7func3opcode
23     {"lui", "0110111"}, // 0
24     {"auipc", "0010111"}, // 1
25     {"jal", "1101111"}, // 2
26     {"jalr", "0001100111"}, // 3
27     {"beq", "0001100011"}, // 4
28     {"bne", "0011100011"}, // 5
29     {"blt", "1001100011"}, // 6
30     {"bge", "1011100011"}, // 7
31     {"bltu", "1101100011"}, // 8
32     {"bgeu", "1111100011"}, // 9
33     {"lb", "0000000011"}, // 10
34     {"lh", "0010000011"}, // 11
35     {"lw", "0100000011"}, // 12
36     {"lbu", "1000000011"}, // 13
37     {"lhu", "1010000011"}, // 14
38     {"sb", "0000100011"}, // 15
39     {"sh", "0010100011"}, // 16
40     {"sw", "0100100011"}, // 17
41     {"addi", "0000010011"}, // 18
42     {"slti", "0100010011"}, // 19
43     {"sltiu", "0110010011"}, // 20
44     {"xori", "1000010011"}, // 21
45     {"ori", "1100010011"}, // 22
46     {"andi", "1110010011"}, // 23
47     {"slli", "00000000010010011"}, // 24
48     {"srli", "00000001010010011"}, // 25
49     {"srai", "01000001010010011"}, // 26
50     {"add", "00000000000110011"}, // 27
51     {"sub", "01000000000110011"}, // 28
52     {"sll", "000000000010110011"}, // 29
53     {"slt", "00000000100110011"}, // 30
54     {"sltu", "00000000110110011"}, // 31
55     {"xor", "00000001000110011"}, // 32
56     {"srl", "00000001010110011"}, // 33
57     {"sra", "01000001010110011"}, // 34
58     {"or", "00000001100110011"}, // 35
59     {"and", "00000001110110011"}, // 36
60     {"csrrw", "0011110011"}, // 37
61     {"csrrs", "0101110011"}, // 38
62     {"csrrc", "0111110011"}, // 39

```

```

63     {"csrrwi",      "1011110011"}, // 40
64     {"csrrsi",      "1101110011"}, // 41
65     {"csrrci",      "1111110011"} // 42
66 };
67
68 typedef struct s_instruction
69 {
70     instruction_type    type;
71     std::string          name;
72     std::string          args;
73     std::string          machine_code;
74
75     s_instruction(const std::string &name, const std::string &args): \
76         name(name), args(args), machine_code(std::string(32, '0'))
77     {
78         size_t i;
79
80         for (i = 0; i < isa.size(); i++)
81             if (this->name == isa[i].ins_name)
82                 break;
83
84         if ((i == 0) || (i == 1))
85             this->type = U;
86         else if (i == 2)
87             this->type = J;
88         else if (i == 3)
89             this->type = I_Jalr;
90         else if ((i >= 4) && (i <= 9))
91             this->type = B;
92         else if ((i >= 10) && (i <= 14))
93             this->type = I_Loads;
94         else if ((i >= 15) && (i <= 17))
95             this->type = S;
96         else if ((i >= 18) && (i <= 23))
97             this->type = I;
98         else if ((i >= 24) && (i <= 26))
99             this->type = I_Shifts;
100        else if ((i >= 27) && (i <= 36))
101            this->type = R;
102        else
103            this->type = I_Atomic;
104    }
105 } t_instruction;
106
107 void open_files(char **argv, std::ifstream &asm_file, std::
108                  ofstream &mc_file);
109 void generate_mc(t_instruction *ins);
110 std::string signed_to_binary(const std::string &numberString, int
111                           size);
112
113 #endif

```

main.cpp

```
1 #include "inc/ASM_to_MC.hpp"
2
3 int main(int argc, char **argv)
4 {
5     std::vector<t_instruction> instructions;
6     std::vector<t_instruction>::iterator ins_it;
7     std::ifstream asm_file;
8     std::ofstream mc_file;
9     std::string line, args;
10
11    if (argc != 2)
12        return (1);
13    open_files(argv, asm_file, mc_file);
14    while (getline(asm_file, line))
15        instructions.push_back(t_instruction(line.substr(0, line.find(" ")), \
16                                              line.substr(line.find(" ") + 1)));
17    for (ins_it = instructions.begin(); ins_it != instructions.end(); \
18          ins_it++)
19    {
20        generate_mc(&(*ins_it));
21        mc_file << ins_it->machine_code << std::endl;
22    }
23    asm_file.close();
24    mc_file.close();
25    return (0);
26 }
```

generate_mc.cpp

```
1 #include "../inc/ASM_to_MC.hpp"
2
3 void generate_mc(t_instruction *ins)
4 {
5     short           index_1, index_2;
6     std::string      rd, imm, pcrel_21, rs1, rs2, pcrel_13, shamt
7                 , csr, rs1_or_zimm;
8     std::vector<t_isa>::iterator it;
9
10    for (it = isa.begin(); it != isa.end(); it++)
11        if (it->ins_name == ins->name)
12            break;
13
14    switch (ins->type)
15    {
16        case U:
```

```

16     index_1 = ins->args.find(',');
17     rd = ins->args.substr(0, index_1);
18     if (rd[0] == 'x')
19         rd = rd.substr(1);
20     imm = ins->args.substr(index_1 + 2);
21
22     ins->machine_code = signed_to_binary(imm, 20) + \
23     signed_to_binary(rd, 5);
24     if (ins->name == "lui")
25         ins->machine_code += isa[0].code.substr(10);
26     else if (ins->name == "auipc")
27         ins->machine_code += isa[1].code.substr(10);
28     break;
29 case J:
30     index_1 = ins->args.find(',');
31     rd = ins->args.substr(0, index_1);
32     if (rd[0] == 'x')
33         rd = rd.substr(1);
34     pcrel_21 = ins->args.substr(index_1 + 2);
35
36     ins->machine_code = (signed_to_binary(pcrel_21, 21))[0] + \
37     signed_to_binary(pcrel_21, 21).substr(10, 10) + \
38     (signed_to_binary(pcrel_21, 21))[9] + \
39     signed_to_binary(pcrel_21, 21).substr(1, 8) + \
40     signed_to_binary(rd, 5) + isa[2].code.substr(10);
41     break;
42 case I_Jalr:
43     index_1 = ins->args.find(',');
44     rd = ins->args.substr(0, index_1);
45     if (rd[0] == 'x')
46         rd = rd.substr(1);
47     index_2 = (ins->args.substr(index_1 + 1)).find(',');
48     imm = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
49     rs1 = ins->args.substr(index_1 + index_2 + 3, \
50     ins->args.length() - (index_1 + index_2) - 4);
51     if (rs1[0] == 'x')
52         rs1 = rs1.substr(1);
53
54     ins->machine_code = signed_to_binary(imm, 12) + \
55     signed_to_binary(rs1, 5) + isa[3].code.substr(7, 3) + \
56     signed_to_binary(rd, 5) + isa[3].code.substr(10);
57     break;
58 case B:
59     index_1 = ins->args.find(',');
60     rs1 = ins->args.substr(0, index_1);
61     if (rs1[0] == 'x')
62         rs1 = rs1.substr(1);
63     index_2 = (ins->args.substr(index_1 + 1)).find(',');
64     rs2 = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
65     if (rs2[0] == 'x')
66         rs2 = rs2.substr(1);

```

```

67     pcrel_13 = ins->args.substr(index_1 + index_2 + 3);
68
69     ins->machine_code = (signed_to_binary(pcrel_13, 13))[0] + \
70     (signed_to_binary(pcrel_13, 13)).substr(2, 6) + \
71     signed_to_binary(rs2, 5) + \
72     signed_to_binary(rs1, 5);
73
74     if (ins->name == "beq")
75         ins->machine_code += isa[4].code.substr(7, 3);
76     else if (ins->name == "bne")
77         ins->machine_code += isa[5].code.substr(7, 3);
78     else if (ins->name == "blt")
79         ins->machine_code += isa[6].code.substr(7, 3);
80     else if (ins->name == "bge")
81         ins->machine_code += isa[7].code.substr(7, 3);
82     else if (ins->name == "bltu")
83         ins->machine_code += isa[8].code.substr(7, 3);
84     else if (ins->name == "bgeu")
85         ins->machine_code += isa[9].code.substr(7, 3);
86
87     ins->machine_code += (signed_to_binary(pcrel_13, 13)).substr
88 (8, 4) + \
89     (signed_to_binary(pcrel_13, 13))[1] + "1100011";
90     break;
91 case I_Loads:
92     index_1 = ins->args.find(',');
93     rd = ins->args.substr(0, index_1);
94     if (rd[0] == 'x')
95         rd = rd.substr(1);
96     index_2 = (ins->args.substr(index_1 + 1)).find('(');
97     imm = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
98     rs1 = ins->args.substr(index_1 + index_2 + 3, \
99     ins->args.length() - (index_1 + index_2) - 4);
100    if (rs1[0] == 'x')
101        rs1 = rs1.substr(1);
102
103    ins->machine_code = signed_to_binary(imm, 12) + \
104    signed_to_binary(rs1, 5);
105
106    if (ins->name == "lb")
107        ins->machine_code += isa[10].code.substr(7, 3);
108    else if (ins->name == "lh")
109        ins->machine_code += isa[11].code.substr(7, 3);
110    else if (ins->name == "lw")
111        ins->machine_code += isa[12].code.substr(7, 3);
112    else if (ins->name == "lbu")
113        ins->machine_code += isa[13].code.substr(7, 3);
114    else if (ins->name == "lhu")
115        ins->machine_code += isa[14].code.substr(7, 3);
116
117    ins->machine_code += signed_to_binary(rd, 5) + "0000011";

```

```

17     break;
18
19 case S:
20     index_1 = ins->args.find(',');
21     rs2 = ins->args.substr(0, index_1);
22     if (rs2[0] == 'x')
23         rs2 = rs2.substr(1);
24     index_2 = (ins->args.substr(index_1 + 1)).find(')');
25     imm = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
26     rs1 = ins->args.substr(index_1 + index_2 + 3, \
27     ins->args.length() - (index_1 + index_2) - 4);
28     if (rs1[0] == 'x')
29         rs1 = rs1.substr(1);
30
31     ins->machine_code = \
32     signed_to_binary(imm, 12).substr(0, 7) + \
33     signed_to_binary(rs2, 5) + \
34     signed_to_binary(rs1, 5);
35
36     if (ins->name == "sb")
37         ins->machine_code += isa[15].code.substr(7, 3);
38     else if (ins->name == "sh")
39         ins->machine_code += isa[16].code.substr(7, 3);
40     else if (ins->name == "sw")
41         ins->machine_code += isa[17].code.substr(7, 3);
42
43     ins->machine_code += signed_to_binary(imm, 12).substr(7) + "0100011";
44     break;
45 case I:
46     index_1 = ins->args.find(',');
47     rd = ins->args.substr(0, index_1);
48     if (rd[0] == 'x')
49         rd = rd.substr(1);
50     index_2 = (ins->args.substr(index_1 + 1)).find(',');
51     rs1 = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
52     if (rs1[0] == 'x')
53         rs1 = rs1.substr(1);
54     imm = ins->args.substr(index_1 + index_2 + 3);
55
56     ins->machine_code = signed_to_binary(imm, 12) + \
57     signed_to_binary(rs1, 5);
58
59     if (ins->name == "addi")
60         ins->machine_code += isa[18].code.substr(7, 3);
61     else if (ins->name == "slti")
62         ins->machine_code += isa[19].code.substr(7, 3);
63     else if (ins->name == "sltiu")
64         ins->machine_code += isa[20].code.substr(7, 3);
65     else if (ins->name == "xori")
66         ins->machine_code += isa[21].code.substr(7, 3);
67     else if (ins->name == "ori")

```

```

167     ins->machine_code += isa[22].code.substr(7, 3);
168     else if (ins->name == "andi")
169         ins->machine_code += isa[23].code.substr(7, 3);
170
171     ins->machine_code += signed_to_binary(rd, 5) + "0010011";
172     break;
173 case I_Shifts:
174     index_1 = ins->args.find(',');
175     rd = ins->args.substr(0, index_1);
176     if (rd[0] == 'x')
177         rd = rd.substr(1);
178     index_2 = (ins->args.substr(index_1 + 1)).find(',');
179     rs1 = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
180     if (rs1[0] == 'x')
181         rs1 = rs1.substr(1);
182     shamt = ins->args.substr(index_1 + index_2 + 3);
183
184     if (ins->name == "srai")
185         ins->machine_code = "0100000";
186     else
187         ins->machine_code = "0000000";
188
189     ins->machine_code += signed_to_binary(shamt, 5) + \
190     signed_to_binary(rs1, 5);
191
192     if (ins->name == "slli")
193         ins->machine_code += "001";
194     else
195         ins->machine_code += "101";
196
197     ins->machine_code += signed_to_binary(rd, 5) + "0010011";
198     break;
199 case R:
200     index_1 = ins->args.find(',');
201     rd = ins->args.substr(0, index_1);
202     if (rd[0] == 'x')
203         rd = rd.substr(1);
204     index_2 = (ins->args.substr(index_1 + 1)).find(',');
205     rs1 = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
206     if (rs1[0] == 'x')
207         rs1 = rs1.substr(1);
208     rs2 = ins->args.substr(index_1 + index_2 + 3);
209     if (rs2[0] == 'x')
210         rs2 = rs2.substr(1);
211
212     if ((ins->name == "sub") || (ins->name == "sra"))
213         ins->machine_code = "0100000";
214     else
215         ins->machine_code = "0000000";
216
217     ins->machine_code += signed_to_binary(rs2, 5) + \

```

```

18     signed_to_binary(rs1, 5);

19
20     if (ins->name == "add")
21         ins->machine_code += isa[27].code.substr(7, 3);
22     else if (ins->name == "sub")
23         ins->machine_code += isa[28].code.substr(7, 3);
24     else if (ins->name == "sll")
25         ins->machine_code += isa[29].code.substr(7, 3);
26     else if (ins->name == "slt")
27         ins->machine_code += isa[30].code.substr(7, 3);
28     else if (ins->name == "sltu")
29         ins->machine_code += isa[31].code.substr(7, 3);
30     else if (ins->name == "xor")
31         ins->machine_code += isa[32].code.substr(7, 3);
32     else if (ins->name == "srl")
33         ins->machine_code += isa[33].code.substr(7, 3);
34     else if (ins->name == "sra")
35         ins->machine_code += isa[34].code.substr(7, 3);
36     else if (ins->name == "or")
37         ins->machine_code += isa[35].code.substr(7, 3);
38     else if (ins->name == "and")
39         ins->machine_code += isa[36].code.substr(7, 3);

40
41     ins->machine_code += signed_to_binary(rd, 5) + "0110011";
42     break;
43 case I_Atomic:
44     index_1 = ins->args.find(',');
45     rd = ins->args.substr(0, index_1);
46     if (rd[0] == 'x')
47         rd = rd.substr(1);
48     index_2 = (ins->args.substr(index_1 + 1)).find(',');
49     csr = ins->args.substr(index_1 + 2, index_2 - index_1 + 2);
50     if (csr[0] == 'x')
51         csr = csr.substr(1);
52     rs1_or_zimm = ins->args.substr(index_1 + index_2 + 3);
53     if (rs1_or_zimm[0] == 'x')
54         rs1_or_zimm = rs1_or_zimm.substr(1);

55
56     ins->machine_code = signed_to_binary(csr, 12) +
57     signed_to_binary(rs1_or_zimm, 5);

58
59     if (ins->name == "csrrw")
60         ins->machine_code += isa[37].code.substr(7, 3);
61     else if (ins->name == "csrrs")
62         ins->machine_code += isa[38].code.substr(7, 3);
63     else if (ins->name == "csrrc")
64         ins->machine_code += isa[39].code.substr(7, 3);
65     else if (ins->name == "csrrwi")
66         ins->machine_code += isa[40].code.substr(7, 3);
67     else if (ins->name == "csrrsi")
68         ins->machine_code += isa[41].code.substr(7, 3);

```

```

169     else if (ins->name == "csrrci")
170         ins->machine_code += isa[42].code.substr(7, 3);
171
172     ins->machine_code += signed_to_binary(rd, 5) + "1110011";
173     break;
174 default:
175     break;
176 }
177 }
```

open_files.cpp

```

1 #include "../inc/ASM_to_MC.hpp"
2
3 void open_files(char **argv, std::ifstream &asm_file, std::ofstream
&mc_file)
4 {
5     std::string argv1;
6     std::string asm_file_name;
7
8     argv1 = std::string(argv[1]);
9     asm_file.open(argv1, std::ios::in);
10    if (asm_file.fail() || (argv1.find(".s") == std::string::npos))
11        exit (1);
12    if (argv1.find("/") != std::string::npos)
13        asm_file_name = \
14            argv1.substr(argv1.find_last_of("/", argv1.find(".s")) - argv1.
find_last_of("/"));
15    else
16        asm_file_name = argv1.substr(0, argv1.find(".s"));
17    mc_file.open("tests/mc_files/" + asm_file_name + ".mem", \
18    std::ios::out | std::ios::trunc);
19 }
```

signed_to_binary.cpp

```

1 #include "../inc/ASM_to_MC.hpp"
2
3 std::string signed_to_binary(const std::string &decimalstr, int size
)
4 {
5     int decimal = std::stoi(decimalstr);
6     std::string binarystr;
7     bool isNegative = (decimal < 0) ? true : false;
8
9     if (isNegative)
10        decimal = -decimal;
11    while (decimal > 0)
12    {
13        binarystr = std::to_string(decimal % 2) + binarystr;
```

```
14     decimal /= 2;
15 }
16 while (binarystr.size() < (size_t)size)
17     binarystr = '0' + binarystr;
18 if (isNegative)
19 {
20     for (char& bit : binarystr)
21         bit = (bit == '0') ? '1' : '0';
22     for (int i = binarystr.size() - 1; i >= 0; --i)
23     {
24         if (binarystr[i] == '0')
25         {
26             binarystr[i] = '1';
27             break;
28         }
29         binarystr[i] = '0';
30     }
31 }
32 return (binarystr);
33 }
```