



**Universidad de Valladolid**



**ESCUELA DE INGENIERÍAS  
INDUSTRIALES**

**UNIVERSIDAD DE VALLADOLID**

**ESCUELA DE INGENIERIAS INDUSTRIALES**

**Grado en Ingeniería Electrónica Industrial y Automática**

# **Implementación en una FPGA de un procesador básico basado en MIPS**

**Autor:**

**Martín Gutiérrez, Jonathan**

**Tutor(es):**

**Cáceres Gómez, Santiago**

**Tecnología electrónica**

**Valladolid, Septiembre, 2015**



*Si A es el éxito en la vida, entonces A es igual a x más y más z. El trabajo es la x; la suerte es la y; y la z mantiene tu boca cerrada.*

*Albert Einstein*



## **Agradecimientos**

En primer lugar, me gustaría dar las gracias a mi tutor, Santiago Cáceres, por darme la posibilidad de realizar mi Trabajo Fin de Grado acerca de temas de actualidad y futuro en la electrónica digital, como son los microprocesadores y dispositivos FPGAs.

También quiero agradecerle a Santiago todas las facilidades que me ha aportado a la hora de realizar este Trabajo Fin de Grado, así como sus conocimientos los cuales me han servido de gran ayuda y, su gran disponibilidad para ayudarme en cualquier complicación que me surgiera.

A todos mis compañeros de estudios, que nunca tuvieron duda de que lograría alcanzar mis objetivos, y que han supuesto un pilar fundamental en mi trabajo, y que gracias a su ayuda lo he podido conseguir.

Por supuesto, a mi familia y mis amigos, que siempre han sido un gran apoyo, y que creyeron en mi capacidad para realizar mis estudios.

Muchas gracias a todos.



## **Resumen**

Este Trabajo Fin de Grado consiste en la realización de un microprocesador MIPS básico mediante la implementación sobre un dispositivo FPGA utilizando un lenguaje VHDL.

Se trata de un microprocesador MIPS básico, debido al conjunto de instrucciones que es capaz de cumplir, así como de los programas que con estas instrucciones permite ejecutar.

El microprocesador se implementa en un dispositivo FPGA debido a la versatilidad de programación que estos elementos poseen.

Para ejecutar el conjunto de instrucciones, el microprocesador MIPS básico se compone de diferentes partes o elementos: Contador de programa, memoria de instrucciones, banco de registros, ALU, memoria de datos y elementos que operan de forma aritmética o lógica con los bits que existen a sus entradas.

Este microprocesador podrá llevar a cabo cualquier programa que se le introduzca siempre y cuando se pueda realizar con el conjunto de instrucciones definido.

Todo esto se lleva a cabo en diferentes partes. Primero, se definen los elementos con los que se va a operar: dispositivos (actualidad y futuro), lenguajes de programación, programas empleados... A continuación se realiza el propio microprocesador sobre el dispositivo FPGA y se le realizan las diferentes pruebas para comprobar su funcionamiento.

## **Palabras Clave**

Microprocesador, MIPS, dispositivo, FPGA, tarjeta Basys2, contador de programa, memoria de instrucciones, memoria de datos, registros, banco de registros, ALU, control, control ALU, extensión de signo, desplazamiento izquierda, sumador, multiplexor, instrucción, programa.





## **Abstract**

This final project consists of the implementation of a core MIPS microprocessor through implementation on an FPGA using VHDL language.

The project is a basic MIPS microprocessor because instruction set makes and programs which could execute these instructions.

The microprocessor is implemented in an FPGA device due to the versatility of programming that these elements have.

To run the instruction set, the core MIPS microprocessor is composed of different parts or elements: program counter, instruction memory, registers bank, ALU, data memory and arithmetic or logical operating elements.

This microprocessor could run any program that you introduce, as long as can be done with the instruction set already defined.

All this is done in different parts. First, the elements, which are going to operate, are defined: devices (present and future), programming languages, programs used... After, the microprocessor is implemented in the FPGA device and doing different tests to check its operation.

## **Keywords**

Microprocessor, MIPS, device, FPGA, Basys2, program counter, instruction memory, data memory, registers, registers bank, ALU, control, ALU control, sign extension, left shift, adder, multiplexer, instruction, program.



# Índice

<b>1. INTRODUCCIÓN .....</b>	<b>1</b>
1.1. Antecedentes .....	3
1.2. Objetivos.....	3
1.3. Justificación .....	4
<b>2. METODOLOGÍA .....</b>	<b>9</b>
2.1. Pasos a cumplir para el desarrollo del Trabajo Fin de Grado .....	11
2.2. Definición de microprocesador MIPS. ....	12
2.3. Definición de FPGA.....	16
2.4. Tarjeta Basys 2 .....	18
2.5. Programas utilizados .....	23
2.5.1. Xilinx – ISE.....	23
2.5.2. Adept .....	24
2.5.3. ModelSim .....	25
2.6. Modo de implementar el microprocesador MIPS .....	25
<b>3. DESARROLLO DEL TRABAJO .....</b>	<b>29</b>
3.1. Descripción de las instrucciones que se van a implementar en el microprocesador MIPS .....	31
3.1.1. Instrucciones tipo R .....	32
3.1.2. Instrucciones tipo I .....	38
3.1.3. Instrucciones tipo J .....	44
3.2. Obtención de los elementos que compondrán el microprocesador MIPS.....	45
3.2.1. Elementos comunes a cualquier tipo de instrucción .....	46
3.2.2. Elementos instrucciones tipo R .....	47
3.2.3. Elementos instrucciones tipo I .....	52
3.2.4. Elementos instrucciones tipo J .....	62
3.2.5. Circuito de control .....	64
3.3. Unión de todas las instrucciones: Hardware del microprocesador .....	65
3.3. Creación software del microprocesador .....	71
3.3.1. Lista de elementos hardware .....	71
3.3.2. Implementación de los elementos .....	72
3.3.3. Microprocesador completo. Unión de todos los elementos.....	89
3.3.4. Programas que se ejecutarán en el microprocesador .....	91

<b>4. ANÁLISIS DE LOS RESULTADOS.....</b>	<b>99</b>
4.1. Modo de llevar a cabo el análisis.....	101
4.2. Análisis de las simulaciones .....	101
4.2.1. Registro contador de programa .....	101
4.2.2. Sumador Más 4.....	101
4.2.3. Memoria de instrucciones.....	102
4.2.4. Banco de registros.....	102
4.2.5. Extensión de signo.....	102
4.2.6. Extensión 5_32 bit.....	103
4.2.7. Desplazamiento 2 izquierda (instrucciones tipo J) .....	103
4.2.8. Circuito de control.....	103
4.2.9. Desplazamiento 2 izquierda .....	104
4.2.10. Unidad Aritmético Lógica .....	104
4.2.11. Circuito de control de la ALU.....	105
4.2.12. Sumador .....	106
4.2.13. Memoria de datos .....	106
4.2.14. Unión 28+4 bits.....	106
4.2.15. Multiplexores.....	107
4.3. Análisis de los programas ejecutados.....	107
4.3.1. Programa 1 .....	107
4.3.2. Programa 2 .....	114
4.3.3. Programa 3 .....	118
4.3.4. Programa 4 .....	119
4.4. Análisis a las modificaciones realizadas en el diseño debido al dispositivo FPGA .....	122
<b>5. CONCLUSIONES .....</b>	<b>125</b>
5.1. Cumplimiento de los objetivos.....	127
5.2. Apreciaciones del Trabajo Fin de Grado .....	128
<b>6. BIBLIOGRAFÍA.....</b>	<b>131</b>
<b>ANEXO DE PROGRAMACIÓN.....</b>	<b>135</b>
Registro Contador de programa.....	137
Sumador más 4 (Contador de programa) .....	138
Memoria de instrucciones (programa 1) .....	139
Memoria de Instrucciones (programa 2) .....	142
Memoria de instrucciones (programa 3) .....	145

Memoria de instrucciones (programa 4) .....	148
Desplazamiento 2 izquierda (instrucciones J).....	151
Circuito de control .....	152
Banco de registros .....	154
Extensor de signo.....	155
Extensor 5_32bit.....	156
Desplazamiento 2 izquierda.....	157
Sumador 32 bits.....	157
Control ALU .....	158
Unidad Aritmético Lógica (ALU) .....	159
Memoria de datos .....	163
Conector 28_32bit .....	164
Multiplexores 551 y 552 .....	165
Multiplexor 432 .....	165
Multiplexor Entrada ALU .....	166
Multiplexores 32B1, 32B2 y 32B3.....	166
Multiplexor Control .....	167
Fichero modulo alto nivel (Conexiones).....	168



## Índice de figuras

Figura 1. Esquema de la arquitectura de un microprocesador MIPS.....	5
Figura 2. Esquema básico MIPS (RISC) .....	15
Figura 3. Estructura interna de una FPGA .....	17
Figura 4. FPGA de Xilins .....	17
Figura 5. Tarjeta Basys 2 .....	18
Figura 6: Parte alimentación de la tarjeta Basys 2 .....	19
Figura 7. Conexiones de la parte de programación de la tarjeta Basys 2.....	20
Figura 8. Colocación del jumper JP4 y de IC6 .....	20
Figura 9. Entradas y salidas de la tarjeta Basys 2 .....	21
Figura 10. Conector de 6 pines auxiliar .....	21
Figura 11. Puerto VGA.....	22
Figura 12. Conectores de módulos periféricos .....	22
Figura 13: Contador de programa y memoria de instrucciones.....	46
Figura 14: Unión contador de programa con la memoria de instrucciones.....	47
Figura 15: Banco de registros y ALU .....	47
Figura 16: Formato instrucciones ADD, SUB, OR, AND y SLT.....	49
Figura 17: Conexiones de elementos para instrucciones ADD, SUB, AND, OR y SLT .....	50
Figura 18: Formato instrucciones SLL y SRL.....	51
Figura 19: Conexiones de elementos para instrucciones SLL y SRL.....	52
Figura 20: Memoria de Datos, Extensión de signo, Desplazar dos izquierda y sumador .....	53
Figura 21: Formato instrucción ADDI .....	54
Figura 22: Conexiones de elementos para instrucciones ADDI .....	55
Figura 23: Formato instrucciones SW y LW .....	56
Figura 24: Conexiones de elementos para instrucciones SW.....	57
Figura 25: Conexiones de elementos para instrucciones LW .....	58
Figura 26: Formato instrucciones BEQ y BNE .....	59
Figura 27: Conexiones de elementos para instrucciones BEQ y BNE .....	60
Figura 28: Registro contador, Sumador, Memoria de instrucciones y desplazamiento 2 izquierda .....	62
Figura 29: Formato instrucción J.....	62
Figura 30: Conexiones de elementos para instrucción J.....	63
Figura 31: Circuito de control del microprocesador.....	64
Figura 32: Hardware instrucciones tipo R .....	66
Figura 33: Hardware instrucciones tipo R más ADDI.....	67
Figura 34: Hardware instrucciones tipo R, más ADDI, más LW, más SW.....	68
Figura 35: Hardware instrucciones tipo R más tipo I.....	69
Figura 36: Hardware instrucción tipo R, más tipo I, más tipo J.....	70
Figura 37: Simulación Registro contador (PC) .....	74

Figura 38: Simulación Sumador Más 4 .....	74
Figura 39: Simulación Memoria de Instrucciones .....	74
Figura 40: Simulación Banco de Registros.....	74
Figura 41: Unión final de la memoria de instrucciones al contador de programa ...	75
Figura 42: Simulación Extensión de signo.....	78
Figura 43: Simulación Extensión 5_32 bit.....	78
Figura 44: Simulación Desplazamiento 2izquierda (instrucciones tipo J) .....	78
Figura 45: Simulación circuito de control OpCode 000000.....	78
Figura 46: Simulación circuito de control OpCode 000010.....	78
Figura 47: Simulación circuito de control OpCode 000100.....	78
Figura 48: Simulación circuito de control OpCode 000101.....	80
Figura 49: Simulación circuito de control OpCode 100011.....	80
Figura 50: Simulación circuito de control OpCode 101011.....	80
Figura 51: Simulación circuito de control OpCode 111111.....	80
Figura 52: Simulación Desplazamiento dos izquierda .....	82
Figura 53; Simulación ALU operación suma.....	82
Figura 54: Simulación ALU operación resta .....	82
Figura 55: Simulación ALU operación AND.....	82
Figura 56: Simulación ALU operación OR .....	84
Figura 57: Simulación ALU operación SLL y SRL .....	84
Figura 58: Simulación ALU operación SLT .....	84
Figura 59: Simulación circuito de Control ALU .....	84
Figura 60: Simulación Sumador .....	86
Figura 61: Simulación Memoria de Datos (escritura) .....	86
Figura 62: Simulación Memoria de Datos (lectura) .....	86
Figura 63: Simulación Memoria de Datos .....	86
Figura 64: Simulación Unión 28+4 bits .....	88
Figura 65: Simulación multiplexor de 1 bit.....	89
Figura 66: Microprocesador MIPS.....	90
Figura 67: Señales del microprocesador .....	108



## Índice de Tablas

Tabla 1. Instrucciones implementadas en el microprocesador .....	4
Tabla 2. Elementos a implementar .....	26
Tabla 3. Conjunto de instrucciones implementadas .....	31
Tabla 4: Formato instrucciones tipo R .....	32
Tabla 5: Instrucción tipo ADD .....	32
Tabla 6: Instrucción tipo SUB .....	34
Tabla 7: Instrucción tipo AND .....	34
Tabla 8: Instrucción tipo OR.....	35
Tabla 9: Instrucción tipo SLL .....	36
Tabla 10: instrucción tipo SRL.....	37
Tabla 11: Instrucción tipo SLT .....	38
Tabla 12: Formato instrucciones tipo I .....	38
Tabla 13: Instrucción tipo LW .....	39
Tabla 14: Instrucción tipo SW.....	40
Tabla 15: Instrucción tipo ADDI .....	41
Tabla 16: Instrucción tipo BEQ .....	42
Tabla 17: Instrucción tipo BNE .....	43
Tabla 18: Instrucción tipo J.....	44
Tabla 19: Elementos que forman el microprocesador .....	72
Tabla 20: Salidas circuito de control.....	80
Tabla 21: Tabla de verdad circuito de control ALU .....	85
Tabla 22: Instrucciones programa 1 .....	92
Tabla 23: Desglose instrucciones programa 1.....	93
Tabla 24: Instrucciones programa 2 .....	94
Tabla 25: Desglose instrucciones programa 2.....	95
Tabla 26: Instrucciones programa 3 .....	96
Tabla 27: Instrucciones programa 4 .....	96
Tabla 28: Desglose Instrucciones programa 3.....	97
Tabla 29: Desglose Instrucciones programa 4.....	97
Tabla 30: Registros programa 2.....	117



## Índice de diagramas

Diagrama 1: Ejemplo diagrama árbol .....	23
Diagrama 2. Modo de crear cada elemento o circuito .....	26
Diagrama 3. Modo de implementar el microprocesador.....	28



# **1. Introducción**



## **1.1. Antecedentes**

El desarrollo llevado a cabo del Trabajo Fin de Grado se basa, personalmente, en la complementación del entendimiento de la arquitectura, desarrollo y funcionamiento de un microprocesador MIPS.

La base de la arquitectura, desarrollo y funcionamiento de un microcontrolador MIPS fue expuesta en la asignatura del tercer curso llamada Electrónica Digital y Microprocesadores. De esta asignatura, la docencia fue realizada por el que es mi tutor del Trabajo Fin de Grado, Santiago Cáceres Gómez. Con la base impartida en la asignatura citada se ha implementado un microcontrolador MIPS básico en una tarjeta Basys2 de Xilinx. Esta tarjeta contiene una FPGA (Field Programmable Gate Array) la cual servirá para el desarrollo del trabajo.

La implementación se realiza en lenguaje VHDL, cuya base fue adquirida en la asignatura llamada Métodos y Herramientas de Diseño Electrónico. En esta asignatura el profesor fue Pedro Luis Díez Muñoz.

## **1.2. Objetivos**

El objeto principal de este Trabajo Fin de Grado consiste en la implementación en una tarjeta FPGA (Field Programmable Gate Array) de un microprocesador básico, basado en la arquitectura MIPS, utilizando un lenguaje de programación VHDL.

Un enfoque que se puede dar al presente Trabajo Fin de Grado, es la familiarización con los microprocesadores, en cuanto a posibles estudiantes o personas que quieran emplear el MIPS diseñado, intentando hacer más fácil su comprensión, viendo cómo se comporta el microprocesador de una forma visual, y no sólo teórica, lo cual puede resultar más engorroso.

De forma más específica, el microprocesador MIPS básico implementado podrá ejecutar las siguientes tareas:

- Realizar diferentes operaciones lógicas.
- Realizar diferentes operaciones aritméticas.
- Realizar otro tipo diferente de operaciones, como por ejemplo rotaciones, saltos, ...
- Leer elementos almacenados en memoria de doble puerto (más adelante se explica la importancia del doble puerto).
- Escribir o almacenar diferentes datos o elementos en una memoria RAM interna.
- Modificar los datos o elementos que se tienen alojados en la memoria RAM.

Para poder realizar todas estas operaciones, el microprocesador MIPS ejecutará una serie de instrucciones (tabla 1). Estas instrucciones se explicarán en capítulos posteriores.

Instrucción (inglés)	Instrucción (español)	Mnemonic	Tipo
Add	Suma	Add	R
Subtract	Resta	Sub	R
Bitwise And	Producto lógico AND	And	R
Bitwise Or	Suma lógica OR	Or	R
Shift Left Logical	Rotar Izquierda	Sll	R
Shift Right Logical	Rotar Derecha	Srl	R
Set if Less Than	Activar si es Menor	Slt	R
Add Immediate	Añadir Dato	Addi	I
Load Word	Cargar Dato	Lw	I
Store Word	Escribir Dato	Sw	I
Branch on Equal	Salto si es Igual	Beq	I
Branch on Not Equal	Salto si no es igual	Bne	I
Jump	Salto incondicional	J	J

*Tabla 1. Instrucciones implementadas en el microprocesador*

En definitiva, lo que cualquier microprocesador bien diseñado es capaz de realizar, ya que se trata del diseño de un microprocesador MIPS no muy complejo.

De este objetivo principal, se pueden sacar diferentes partes u objetivos secundarios, sobre todo a nivel personal. Estos objetivos secundarios personales son, por ejemplo, reforzar y entender con mayor claridad la arquitectura, desarrollo y funcionamiento de cualquier microprocesador, o también, reforzar conocimientos desarrollados en la asignatura de Electrónica Digital y Microprocesadores, como puede ser implementar circuitos combinatoriales o secuenciales, o de otra asignatura como Métodos y Herramientas de Diseño Electrónico, donde se imparte conocimientos del lenguaje utilizado (VHDL).

### **1.3. Justificación**

Los objetivos antes expuestos se deben cumplir debido a una serie de razones.

La primera es, implementando un microprocesador MIPS desde su inicio, partiendo de cero, se comprende mejor todos los aspectos que le pueden rodear, desde su modo de funcionamiento, la arquitectura que contiene (memoria, registros, ALUs...), cómo se puede construir, la información que se puede almacenar, etc.

La arquitectura del microprocesador básico basado en MIPS que se pretende implementar, incluye, entre otros, los siguientes elementos:





Hoy en día, en la electrónica digital, hay dos ramas claramente marcadas hacia el progreso. Una de ellas es la rama de los microprocesadores y otra la de dispositivos FPGAs. Por lo tanto, se profundiza un poco en las dos ramas. Ambos elementos tienen muchas ventajas en comparación con sistemas electrónicos más anticuados. Sin embargo, también existen diferencias entre ellos.

Un microprocesador puede ser programado para que realice una serie de acciones, pudiendo no cumplirlas si no está diseñado previamente para realizarla. Con una FPGA, se puede construir el microprocesador que se desee y así que realice todas las opciones que se quiera.

Con ambos se ha reducido enormemente el tamaño de los circuitos electrónicos. Esta reducción de tamaño basada en las técnicas de nanotecnología reduce el consumo de energía, aumenta la velocidad y el poder de procesamiento, lo cual son grandes ventajas con respecto a sistemas electrónicos más antiguos. El mayor desarrollo lo ha llevado a cabo la empresa Intel. En 2008 ya empezó a revolucionar el mercado con un microprocesador de 45 nanómetros. En la actualidad, Intel ya ha conseguido reducir el tamaño hasta 14 nanómetros.

En años pasados, los dispositivos basados en tecnología de FPGA estaban disponibles, únicamente, para los ingenieros que tuvieran un profundo y extenso conocimiento del diseño de hardware digital. Hoy día, esto no es así, pues cualquiera, con unos conocimientos básicos, puede realizar diseños electrónicos sencillos con estos dispositivos.

Los dispositivos FPGAs, desde que Xilinx los inventó en 1984, han pasado de ser sencillos chips de lógica de acoplamiento a reemplazar a los circuitos integrados de aplicación específica (ASICs) y procesadores para procesamiento de señales y aplicaciones de control.

Estos dispositivos, tienen cinco características fundamentales, que los hace ser realmente competitivos en el mercado: [1]

- Muy buen rendimiento
- Tiempo en Llegar al Mercado (rápido desarrollo de prototipos).
- Precio mucho menor que ASICs.
- Fiabilidad
- Mantenimiento a Largo Plazo

Todas estas características, y alguna más que poseen, se desarrollarán con mayor exactitud en el apartado en el que se define qué es un dispositivo FPGA.

Este Trabajo Fin de Grado también se realiza para que, en los años venideros, se pueda utilizar el MIPS diseñado para intentar mejorar la comprensión de posteriores alumnos, tanto del propio microprocesador, como de los dispositivos FPGAs. Cuando se les exponga el temario relacionado con microprocesadores, en la correspondiente asignatura, no será realizada sólo de forma teórica, sino que se podrán ver ejemplos en el propio microprocesador.

El refuerzo de los conceptos desarrollados en las asignaturas antes citadas es otra justificación propia del proyecto. Así una mayor comprensión de cualquier aspecto relacionado, bien con el tema de microprocesadores, o bien con el tema relacionado con los dispositivos FPGAs, quedará firmemente comprendida con este Trabajo Fin de Grado, con el que se intenta dar un paso más hacia el entendimiento de todo lo que rodea a ambos temas.

También personalmente, se eleva mi nivel de programación en lenguaje VHDL, además de mejorar, obviamente, la comprensión de elementos digitales como son los circuitos combinacionales o secuenciales (ALUs, registros, contadores,...), y por supuesto, lo citado anteriormente, mejor entendimiento de los microprocesadores y FPGAs, dos herramientas claves en el futuro electrónico, y por lo tanto, dos herramientas muy valiosas para cualquier ingeniero electrónico.



# 2. Metodología



## **2.1. Pasos a cumplir para el desarrollo del Trabajo Fin de Grado**

Para realizar con éxito la implementación sobre una tarjeta FPGA de un microprocesador básico, basado en la arquitectura MIPS se definen una serie de pasos o hitos a cumplir.

Primero de todo, se realiza una comprensión general de lo que se desea realizar en este proyecto. Para ello, se debe comprender correctamente qué es un microprocesador MIPS, así también qué es un dispositivo FPGA y las funcionalidades que tienen ambos.

Una vez que se tiene seguridad de lo que se desea realizar y cómo realizarlo, se elige el lenguaje de programación, así como el programa idóneo para implementar dicho microprocesador. Se utiliza lenguaje VHDL, el cual viene marcado de inicio por el enunciado del Trabajo de Fin de Grado, y el programa utilizado es proporcionado por Xilinx.

Para iniciar la implementación de un microprocesador, primero de todo se debe definir el conjunto de instrucciones que éste ejecuta. Este conjunto queda definido por tres tipos de instrucciones (R, I y J), que se definirán más adelante. Con estos tres tipos de instrucciones se podrán ejecutar todos los programas que se desean en el microprocesador diseñado. También se definen estos tres tipos debido a su sencillez desde el punto de vista docente, ya que es un enfoque que se le da al Trabajo Fin de Grado.

Una vez definido el conjunto de instrucciones se escogen los elementos o circuitos capaces de llevar a cabo dichas instrucciones (Memorias, Unidad Aritmético Lógica, Banco de Registros, etc).

Después de definir el conjunto de instrucciones y los elementos que lo llevarán a cabo, y antes de comenzar con el diseño del microprocesador, se llega a la conclusión de que implementando cada circuito o elemento que contiene el MIPS por separado, en el futuro será una ventaja para posibles modificaciones, y también, por ejemplo, en el caso de que se utilice para enseñar a próximos alumnos, sean capaces de ver cómo se ha implementado cada elemento y puedan interactuar con ellos.

Dicho esto, se procede a la programación de cada circuito individualmente y, como es lógico, se realizan pruebas de cada uno para comprobar su correcto funcionamiento. Estas pruebas se pueden realizar de dos maneras:

1. Mediante el programa ModelSim, que permite simular el comportamiento de cada elemento.

2. Mediante la tarjeta Basys 2, que posee entradas y salidas para realizar comprobaciones.

La segunda opción es una opción más complicada, pues no existen tantas salidas y entradas en la tarjeta Basys 2 para realizar su comprobación, sin embargo, con la inclusión de distintos multiplexores se puede llegar a visualizar que cada elemento funciona correctamente. Por ejemplo, se incluyen un multiplexor con dos bits de control en la salida del banco de registros para que, modificando estas señales de control, muestre en las salidas (LEDs) los 8 bits menos significativos, los 8 más significativos o los intermedios.

Para realizar la transferencia de los ficheros programados en VHDL a la tarjeta Basys 2, se utiliza el programa Adept, tanto para probar los elementos de manera individual, como más adelante cuando se tenga el microprocesador completamente definido.

Con todos los elementos definidos de manera individual y funcionando correctamente, hay que conectarlos unos con otros, de tal manera que cumplan el esquema de la figura 1, para que así el comportamiento del microprocesador sea el deseado.

Por último queda por realizar diferentes pruebas para corroborar que el microprocesador implementado funciona de manera correcta a lo que se desea. Estas pruebas se realizan con la tarjeta Basys2, introduciendo salidas en cada elemento para poder ver su resultado con los LEDs o los displays que contiene.

## **2.2. Definición de microprocesador MIPS.**

Un microprocesador es el circuito integrado central y más complejo de un sistema informático. Se le puede llamar coloquialmente como el cerebro, el motor, el corazón del sistema. Este ejecuta instrucciones que se le dan a la computadora a muy bajo nivel haciendo operaciones lógicas simples, como sumar, restar, multiplicar y dividir.

Se suele conectar mediante un zócalo, además de incorporarle algún sistema refrigerante para que disipe el calor generado y tenga un funcionamiento más estable.

Algunos datos históricos de los microprocesadores, son por ejemplo, que el primero apareció gracias a la invención de Intel en el año 1971. El modelo fue nombrado como 4004, estaba fabricado con tecnología PMOS y constaba de 4 bits. Con el paso de los años Intel fue mejorando los modelos creados y diferentes competidores quisieron, ante el éxito que se estaba teniendo, entrar en el mercado.



En la actualidad existen multitud de microprocesadores, con características muy superiores a los primeros, como ya se ha dicho, es una senda marcada para el avance de la tecnología digital.

Los microprocesadores son uno de los elementos que han desplazado a la electrónica más tradicional debido a sus dos grandes ventajas:

- Emplear pocos componentes. Se reduce tanto el tamaño, como el consumo y aumenta la fiabilidad debido a su número menor de conexiones entre elementos. Cuantos menos componentes y menos conexiones, más se reduce el coste.
- Programabilidad. Un sistema programable permite modificar la función que realiza sin más que variar el programa, a diferencia de sistemas electrónicos anteriores.

El microprocesador realiza una secuencia de operaciones, pero esta secuencia no es única, sino que es programable mediante instrucciones. Las instrucciones residen en una memoria a la que accede el microprocesador, extrae una instrucción, la decodifica, la ejecuta y vuelve por otra instrucción y así sucesivamente, salvo que reciba una instrucción de parar. [2]

Se puede llegar a confundir microprocesador con microcontrolador, no es lo mismo. Un microcontrolador contiene al microprocesador, es decir, el microprocesador es una parte, muy importante, del microcontrolador. Por decirlo de forma más escueta, el microcontrolador podría ser el ordenador, y el microcontrolador una parte de él.

Existen diferentes tipos de arquitecturas para implementar un microprocesador. En el presente Trabajo Fin de Grado se diseña un microprocesador tipo MIPS.

Un microprocesador MIPS (*Microprocessor without Interlocked Pipeline Stages*) es un sistema digital de cierta complejidad. Su misión consiste en ejecutar los programas de cualquier proceso que se requiera, ejecutando un conjunto de instrucciones que tiene programadas en las cuales se realizan diferentes operaciones lógicas o aritméticas, o se accede a elementos de memoria, instrucciones de salto, transferencia o almacenamiento.

La arquitectura MIPS se desarrolló por primera vez gracias a John Hennessy y unos amigos suyos de Stanford en la década de 1980 (MIPS Computer Systems Inc). Los microprocesadores MIPS son utilizados, entre otros, por Silicon Graphics, Nintendo, Cisco, Sony Play Station 2, consolas de videojuegos y muchos más productos que se utilizan en la vida cotidiana.

El hardware que posee la arquitectura MIPS es pequeño debido a la poca complejidad para decodificar el conjunto de instrucciones que realiza. Se trata de una arquitectura RISC (Reduced Instruction Set Computer).

La arquitectura RISC ejecuta cada instrucción en un solo ciclo. Esto reduce el microprocesador en complejidad y aumenta la velocidad. Características: [3].

- Un conjunto limitado y simple de instrucciones. Se cuenta con un conjunto constituido por instrucciones capaces de ejecutarse en un ciclo de reloj.
- Instrucciones orientadas a los registros con acceso limitado a memoria. Un conjunto de tipo RISC ofrece pocas instrucciones básicas (Load y Store) que pueden guardar datos en la memoria. El resto de ellas operan exclusivamente con registros
- Modos limitados de direccionamiento. Muchas computadoras de tipo RISC ofrecen sólo un modo para direccionar la memoria, generalmente un direccionamiento directo o indirecto de registros con un desplazamiento.
- Un gran banco de registros. Los procesadores de tipo RISC contienen muchos registros de manera que las variables y los resultados intermedios usados durante la ejecución del programa no requieran utilizar la memoria. Con ello se evitan muchas instrucciones del tipo Load y Store.
- Palabra de la instrucción con extensión y formatos fijos. Al hacer idénticos el tamaño y el formato de todas las instrucciones, es posible obtenerlas y decodificarlas por separado. No hay que esperar hasta conocer la extensión de una instrucción anterior a fin de obtener y decodificar la siguiente. Por tanto, esas dos acciones pueden llevarse a cabo en paralelo. En pocas palabras, la decodificación se simplifica.

Los microprocesadores MIPS RISC tienen una arquitectura que incluye: corrección de longitud, sencillo decodificado del formato de instrucción, accesos a memoria limitados para cargar y almacenar instrucciones, unidad de control cableada, un archivo de registro de uso general grande, y todas las operaciones se realizan dentro de los registros del microprocesador. [4].

El MIPS R2000 fue lanzado en 1988, y fue uno de los primeros microprocesadores RISC diseñados. Ofrecía un conjunto de instrucciones muy limpio y un modo de programación "máquina virtual" y automático segmentado.

Que la arquitectura sea segmentada quiere decir que el microprocesador se divide en subpartes, donde cada parte se puede ejecutar en un ciclo de reloj. Esto implica que es posible aumentar la frecuencia de reloj en comparación con un diseño no segmentado. El microprocesador implementado no utiliza el diseño segmentado.

Un microprocesador puede realizar casi cualquier tarea, siempre que previamente sea implementado para tal tarea. Para ellos se define un conjunto de instrucciones y con la combinación de todas o de algunas de estas instrucciones, se llegan a ejecutar los diferentes programas deseados a desarrollar en el microprocesador.

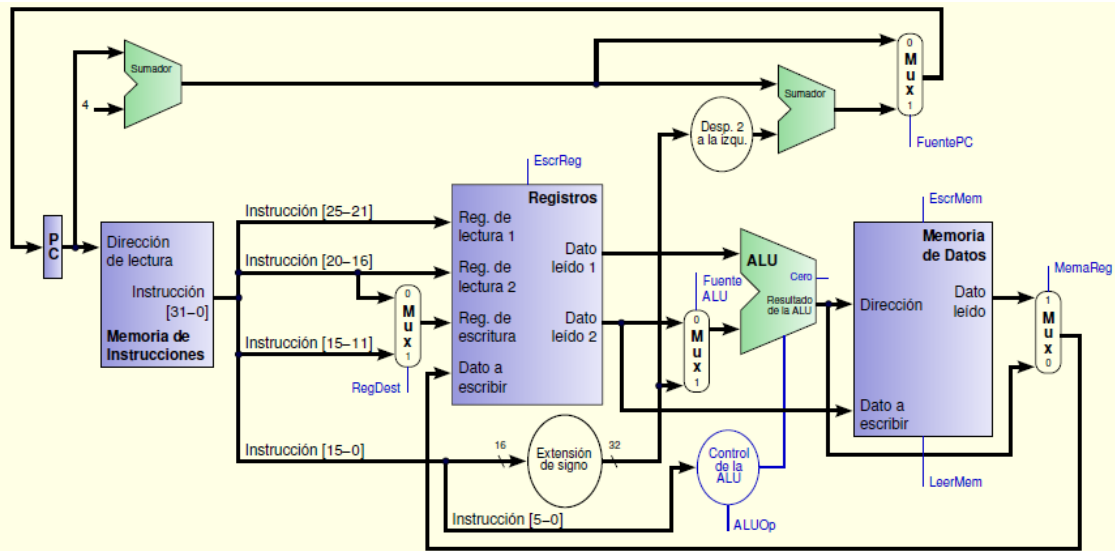


Figura 2. Esquema básico MIPS (RISC)

En la figura 2, obtenida del documento “Diseño del procesador MIPS R2000” [5], se puede observar los elementos básicos que componen un microprocesador MIPS, así como sus conexiones.

Un ejemplo, aunque no el único, de un posible programa a que pueda ejecutar un microprocesador es el siguiente:

- Realizar una multiplicación de dos números contenidos en dos registros diferentes o en una memoria de datos, previamente almacenados. Para ello se necesitarán instrucciones tipo R (add) y tipo I (SW, LW, ADDI, ...). Más adelante se explicará con más detalles el conjunto de instrucciones y los programas que se desarrollarán.

Las instrucciones que se ejecutan en un microprocesador se encuentran alojadas en una memoria interna a la que se accede, con el fin de extraer esta instrucción, descodificarla y ejecutarla. Una vez que se ha ejecutado esta instrucción se vuelve a repetir el proceso con otra instrucción, que puede ser igual, o diferente, y este proceso es indefinido, con la excepción de que se reciba la instrucción de parar.

Por lo tanto, se podría llegar a comentar que un microprocesador constituye la parte del ordenador que se encarga del control y proceso de la información, es decir la Unidad Central de Proceso o CPU. [2].

Esto lo propuso por primera vez Von Neumann, el cual dividió el computador en componentes básicos, y estos componentes permanecen todavía: la CPU o procesador, es el núcleo del computador y contiene todo excepto memoria, entrada y salida. El procesador se divide además en computación y control. [6].

## **2.3. Definición de FPGA**

Un dispositivo FPGA (Field Programmable Gate Array) es un elemento semiconductor, normalmente silicio, el cual se divide en bloques de lógica cuya interconexión y funcionalidad se puede configurar utilizando un lenguaje adecuado de descripción.

Un FPGA es un dispositivo lógico programable (PLD) que puede ser reprogramado cualquier número de veces después de que haya sido fabricado.

Internamente, las FPGAs contienen compuertas de elementos lógicos programables prefabricados llamadas células. Una sola célula puede implementar una red de varias puertas lógicas que se alimentan en los flip-flops. Estos elementos lógicos están dispuestos internamente en una configuración de matriz y se conectan automáticamente unos con otros empleando una red de interconexión programable.

La naturaleza reprogramable de FPGAs, las hace ideales para fines educativos, ya que permite a los estudiantes que intenten tantas iteraciones como sea necesario para corregir y optimizar su diseño.

Hoy en día, también es una herramienta muy útil para realizar cualquier tipo de diseño, pues no se necesitan conexiones externas, debido a su programación interna. Todas las características que poseen las FPGAs se pueden ver a continuación: [4]

1. Rendimiento: Los dispositivos FPGAs exceden la potencia de cómputo de los procesadores digitales de señales aumentando la ejecución secuencial y logrando más en cada ciclo de reloj. Ofrece tiempos de respuesta más veloces y funcionalidad especializada.
2. Tiempo en llegar al mercado: La tecnología FPGA ofrece flexibilidad y capacidades de rápido desarrollo de prototipos para enfrentar los retos de que un producto se libere tarde al mercado. Se puede probar cualquier circuito creado y se puede verificar su correcto funcionamiento sin tener que pasar por un largo proceso de fabricación, como pasa, por ejemplo, en el diseño personalizado de ASIC. Se podrán realizar también todos los cambios deseados posteriormente en el diseño.
3. Precio: El precio de la ingeniería no recurrente de un diseño personalizado ASIC excede considerablemente al de las soluciones de hardware basadas en FPGA. Los requerimientos necesarios al diseñar cualquier sistema se van modificando con el tiempo de desarrollo, y el precio de cambiar incrementalmente los diseños FPGA es insignificante si se compara con el precio de implementar cambios en un diseño ASIC antes de su lanzamiento.
4. Fiabilidad: Los circuitos de un FPGA son una implementación segura de la ejecución de un programa, sin producir fallos. Los sistemas basados en

procesadores frecuentemente implican varios niveles de abstracción para auxiliar a programar las tareas y compartir los recursos entre procesos múltiples. El software a nivel driver se encarga de administrar los recursos de hardware y el sistema operativo administra la memoria y el ancho de banda del procesador. El núcleo de un procesador sólo puede ejecutar una instrucción a la vez, y los sistemas basados en procesadores están siempre en riesgo de que sus tareas se obstruyan entre sí. Los FPGAs, que no necesitan sistemas operativos, minimizan los retos de fiabilidad con ejecución paralela y hardware preciso dedicado a cada tarea.

5. Mantenimiento a largo plazo: Los dispositivos FPGAs son actualizables en campo y no requieren el tiempo y el precio que implica rediseñar un ASIC. Los dispositivos FPGA, al ser reconfigurables, son capaces de mantenerse al tanto con modificaciones a futuro que pudieran ser necesarias. Mientras el producto o sistema se va desarrollando, se puede implementar mejoras funcionales sin la necesidad de invertir tiempo rediseñando el hardware o modificando el diseño de la tarjeta.

Todas estas características hacen que los dispositivos FPGAs sean tan importantes hoy en día y para el futuro en el diseño de circuitos electrónicos, ya que se trata de una herramienta muy potente.

La lógica programable de estos bloques es capaz de implementar todo tipo de funciones, tanto sencillas, como puede ser una puerta lógica, o funciones más complicadas como sistemas que implementen microprocesadores, como por ejemplo, en este caso, que se implementa un microprocesador MIPS básico.

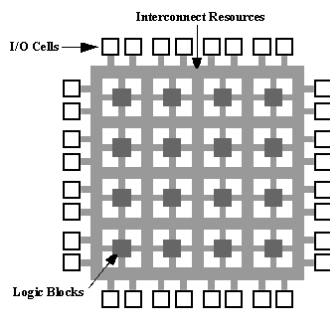


Figura 3. Estructura interna de una FPGA



Figura 4. FPGA de Xilins

Las FPGAs fueron inventados hace menos de 30 años y su influencia en diferentes direcciones en la ingeniería está creciendo continuamente. Hay muchas razones para tal progreso, la más importante de estas razones, es una configurabilidad sencilla y un desarrollo relativamente barato. [7].

Hoy en día hay muchos fabricantes de dispositivos FPGA como son Actel, Altera, Atmel, Cypress, Lucent y Xilinx. Para este proyecto se utilizará una fabricada por Xilins (figura 4).

En la figura 3, se puede ver cómo es la arquitectura interna que tienen los dispositivos FPGAs. Una FPGA está compuesta por un número de entradas y salidas que sirven para poder acceder a los bloques lógicos programables y así configurar el diseño del circuito a crear. Estos bloques de lógica configurable junto con los recursos de memoria RAM que tengan son las dos características que hay que tener en cuenta a la hora de elegir un dispositivo FPGA u otro.

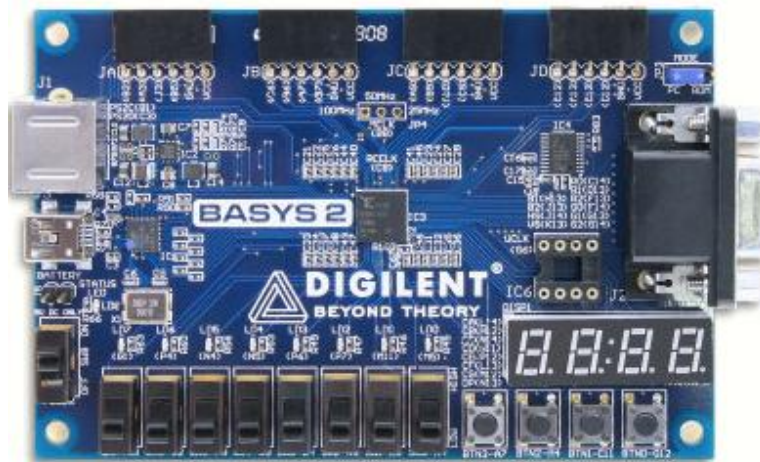
Para realizar la programación de un dispositivo FPGA se tiene diferentes tipos de lenguajes:

- VHDL
- ABEL
- Verilog

Todos estos lenguajes son conocidos como HDL (Hardware Description Language), aunque también existen muchos otros tipos de lenguajes que entran dentro del grupo HDL.

Por todas estas razones, el estudio de estos dispositivos se ha convertido en indispensable para muchos diseñadores de circuitos, microprocesadores, etc. los cuales antes preferían usar sistemas hardware más complejos de implementar y menos fiables debido, sobre todo, a conexiones externas.

## 2.4. Tarjeta Basys 2



*Figura 5. Tarjeta Basys 2*

Todas las figuras de este apartado provienen del documento “Digilent Basys2 Board Reference Manual”.

La tarjeta Basys 2 de Digilent es la tarjeta que se usa para el desarrollo del proyecto. Esta tarjeta posee un dispositivo FPGA de Xilins, cuyo modelo es Spartan 3E-250 CP132.

Esta tarjeta Basys 2 usa un controlador USB Atmel AT90USB2, y ofrece un hardware listo para su adecuado uso, capaz de implementar todo tipo de circuitos que van desde dispositivos lógicos básicos a los controladores complejos.

Todas las características que posee la tarjeta se pueden ver a continuación [8]:

- Xilinx Spartan 3-E FPGA, puertas 100K
- Características FPGA: multiplicadores 18-bit, bloque de doble Puerto RAM 72Kbits, y operaciones de 500MHz+.
- USB 2 puerto de alta velocidad para la configuración FPGA y transferencia de datos (usando el software Adept 2.0)
- XCF02 Plataforma Flash ROM que almacena las configuraciones FPGA indefinidamente
- Frecuencia del oscilador configurable (25, 50, y 100 MHz), además de un socket para un segundo oscilador
- Tres reguladores de voltaje (1.2V, 2.5V, and 3.3V) que permiten el uso de 3.5V-5.5V suministros externos
- 8 LEDs, 4 displays de siete segmentos, 4 pulsadores, 8 interruptores, puerto PS/2, y puerto VGA de 8-bit
- Cuatro cabeceras de 6 pines para los usuarios del sistema I/Os, y sujetan las placas de accesorios de Digilent Pmod
- *Adept 2.0 para implementar los diseños*

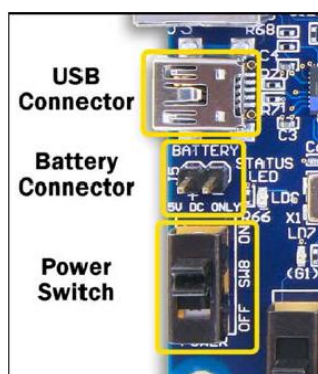


Figura 6: Parte alimentación de la tarjeta Basys 2

La placa Basys 2 se puede dividir en varias partes. La primera es la parte de alimentación, la cual se puede observar en la figura 6.

Esta placa puede ser alimentada por un cable USB, pero también se proporciona un conector de batería, de forma que se puedan utilizar otros suministros externos. Estos suministros externos deberán estar dentro de un rango de 3,5V – 5,5V.

La Basys 2, utiliza un rutado de PCB de cuatro capas, con las capas interiores dedicadas a VCC y GND. La FPGA y los demás circuitos integrados en la placa tienen grandes condensadores cerámicos, de derivación, colocados lo más cerca posible a cada pin VCC, lo que resulta en una fuente de alimentación de muy bajo ruido.

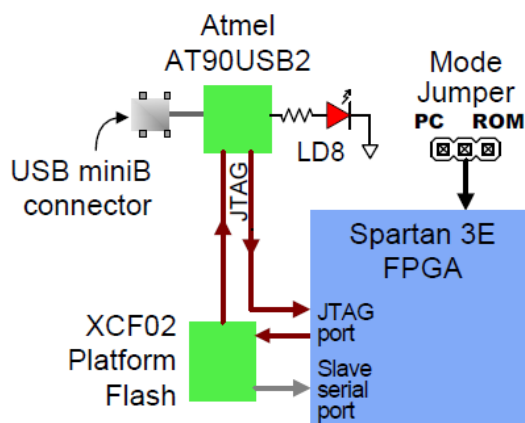


Figura 7. Conexiones de la parte de programación de la tarjeta Basys 2

Otra parte de la placa Basys 2, es donde se produce la configuración o programación de la tarjeta FPGA. Se muestra el esquema que describe sus conexiones en la figura 7. Se puede ver como se accede al dispositivo FPGA para su programación a través del conector mini-usb y el microcontrolador atmel.

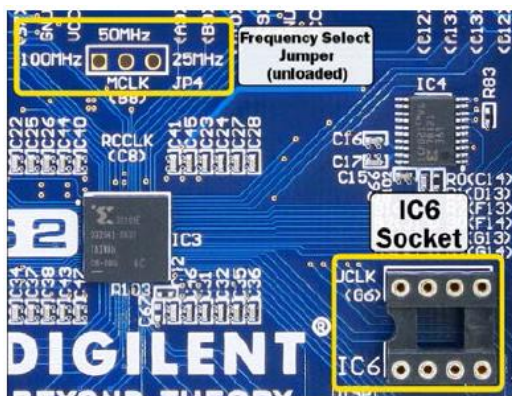


Figura 8. Colocación del jumper JP4 y de IC6

Durante la configuración o programación, un archivo " bit" se transfiere a las células de memoria dentro de la FPGA para definir las funciones lógicas y las interconexiones del circuito a implementar. En este proyecto, se utilizan programas de Xilinx que más adelante se explicarán y comentarán.

La tarjeta Basys 2 incluye un oscilador de silicio configurable por el usuario que produce frecuencias de 25MHz, 50MHz o 100MHz basado en la posición del jumper JP4 (figura 8). También existe la posibilidad de añadir otro oscilador en IC6 (ver figura 8).



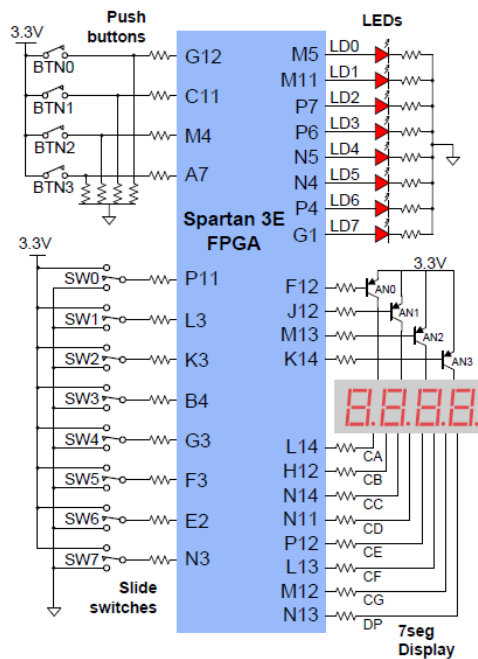


Figura 9. Entradas y salidas de la tarjeta Basys 2

Otra parte que incluye Basys 2, son botones, pulsadores, leds de salida, displays de 7 segmentos,... Todos ellos son las entradas y salidas que tiene la tarjeta Basys 2, y pueden ser visualizados en la figura 9.

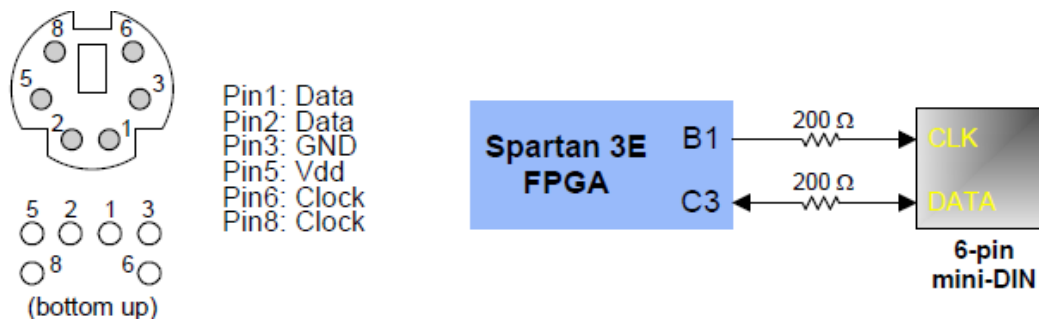


Figura 10. Conector de 6 pines auxiliar

Como se puede ver en la figura 9, la tarjeta Basys 2 tiene un total de:

- 8 interruptores
- 4 botones
- 4 displays de 7 segmentos
- 4 displays de un punto
- 8 leds de salida

La tarjeta Basys 2 también posee, por si el usuario lo requiere, un conector de 6 pines, para poder conectar bien un ratón, o bien, un teclado. Es un puerto de conexión PS/2.

También la tarjeta Basys2 utiliza 10 señales FPGA para crear un puerto VGA con las dos señales de sincronización estándar (HS - Horizontal Sync, y VS - Sincronización vertical). Estas señales se crean para controlar señales de vídeo. El puerto VGA se muestra en la figura 11.

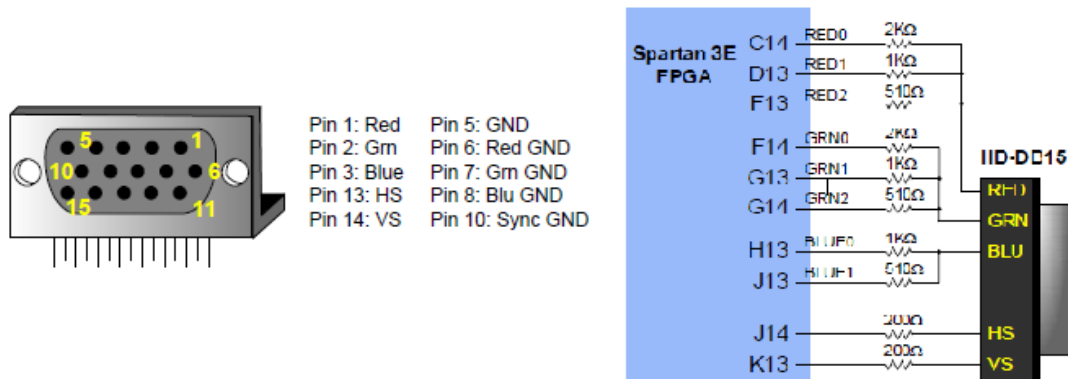


Figura 11. Puerto VGA

Basys2 también proporciona cuatro conectores de los módulos periféricos de 6 pines. Cada conector proporciona Vdd, GND, y cuatro señales FPGA únicas. Estos conectores se pueden ver en la figura 12.

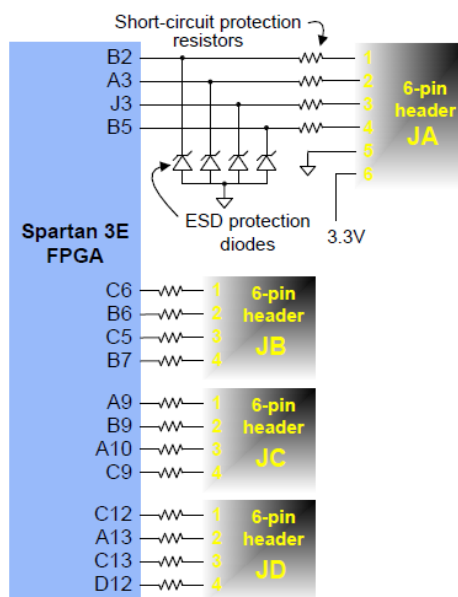


Figura 12. Conectores de módulos periféricos

Como puede llegar a ser lógico, en el presente proyecto no se utilizará todo el potencial que posee la placa Basys 2. [9]

Una vez que ya se tienen definidos tanto los dispositivos FPGAs, como los microprocesadores, MIPS y la tarjeta Basys 2 que se va a emplear en este Trabajo

de Fin de Grado, sólo queda conocer los programas que se utilizarán para el diseño del microprocesador.

## 2.5. Programas utilizados

Para la realización del presente proyecto, es decir, para realizar un microcontrolador mediante la programación de una tarjeta FPGA, se han utilizado diferentes programas, para realizar la programación, compilado, simulación y transferencia de ficheros a la tarjeta Basys 2. Estos programas utilizados son:

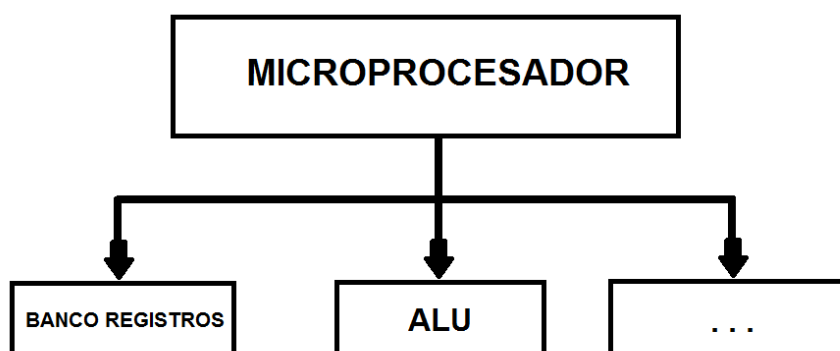
- Xilins – ISE
- Digilent Adept
- ModelSim

A continuación se detalla el funcionamiento de cada uno de ellos.

### 2.5.1. Xilinx – ISE

Xilinx ISE es la herramienta o programa que se utiliza para implementar el diseño del microprocesador MIPS en VHDL. Se trata de un software de diseño gratuito capaz de realizar diseños de funciones para dispositivos FPGA.

Este programa sirve para descargar o programar, en ambas direcciones, el dispositivo FPGA, mediante diseños basados en lenguajes HDL, en el presente Trabajo Fin de Grado el lenguaje es VHDL. En principio también se pueden realizar simulaciones, implementaciones, instalación de dispositivos, y la programación JTAG, aunque no se emplean porque o no son necesarias o, en el caso de simulaciones, se utiliza otro programa llamado ModelSim que es más entendible y los resultados se contrastan de una forma más eficaz.



*Diagrama 1: Ejemplo diagrama árbol*

El programa Xilinx ISE es sencillo en cuanto a la comprensión de su funcionamiento, creación de proyectos, etc. También se trata de una herramienta muy potente que permite crear un proyecto en forma de árbol que es como se desarrollará el Trabajo Fin de Grado. Un ejemplo de proyecto con ficheros en forma de árbol se puede ver en el diagrama 1.

En un proyecto creado con ficheros con arquitectura de árbol, es importante compilar los ficheros de los niveles más bajos primero, e ir ascendiendo en la compilación, porque si no se realiza así, el programa dará fallos en la compilación y el diseño no podrá ser implementado. Este programa permite crear el fichero que está en el nivel más alto, y adjuntar los ficheros de menor nivel, para que se compilen de forma idónea.

Es un programa básico en el desarrollo de este Trabajo Fin de Grado ya que es el programa que genera los archivos que, una vez compilados correctamente, se introducen en el dispositivo FPGA para que se creen las diferentes conexiones en su interior y así se cree el microprocesador MIPS deseado.

Aunque no se va a emplear en un principio, también permite crear proyectos a través de diseños esquemáticos.

### **2.5.2. Adept**

Una vez realizadas todas las compilaciones con el programa Xilins ISE, para realizar la transmisión del fichero programado a la tarjeta Basys 2, se utiliza el programa Adept.

Este programa permite la comunicación con las placas del sistema Digilent y una gran variedad de dispositivos lógicos. Proporciona configuración JTAG y transferencia de datos, además de que añade características de verificación a bordo de E/S, y una conexión de dispositivos de apertura automática.

Es un programa que pertenece a la marca Digilent, y cuyas características son[8]:

- Configura los dispositivos lógicos pertenecientes a Xilinx. Inicia una cadena de exploración y programa FPGAs, CPLDs y PROM. Organiza y realiza un seguimiento de los archivos de configuración.
- Transferencia de datos desde y hacia la FPGA a bordo de la placa base. Lee y escribe en los registros especificados. Carga un flujo de datos a un registro o lee un flujo de datos de un registro.
- Organiza y conecta rápidamente a sus módulos de comunicación.

- Programación de dispositivos Xilinx XCFS, plataforma Flash utilizando archivos .bit o .mcs.
- Programación de Xilinx CPLDs CoolRunner2 utilizando archivos .jed.
- Programación de la mayoría de la serie FPGAs Spartan y Virtex con archivos .bit

Se emplea con el objetivo de transmitir los ficheros generados en el programa Xilinx ISE que servirán para la programación del dispositivo FPGA, es decir, crear las conexiones internas que se necesitan en el diseño del microprocesador que se implementa.

### **2.5.3. ModelSim**

Este programa se utiliza con el fin de realizar las simulaciones de los elementos que se programarán para realizar el microprocesador MIPS.

ModelSim es un software propiedad de Altera, y es capaz de simular cualquier tipo de comportamiento de dispositivos FPGA. No se trata de un software libre, pero hay versiones de prueba con algunas limitaciones. Sin embargo, se ha conseguido simular el funcionamiento de cada elemento que compone el microprocesador.

Soporta diferentes tipos de lenguajes:

- VHDL
- Verilog
- SystemVerilog
- PSL
- SystemC

Como ya se ha dicho, esta parte de simulación se podría realizar con el programa Xilinx ISE, pero el programa ModelSim se puede considerar que es más sencillo de comprender las simulaciones creadas y más vistosas.

## **2.6. Modo de implementar el microprocesador MIPS**

Para implementar el el microprocesador MIPS básico, que se expone en este Trabajo Fin de Grado se han seguido una serie de etapas o pasos.

Primero de todo, y muy importante como es lógico, es realizar una comprensión y entendimiento de cómo ha de funcionar un microprocesador MIPS. Para ello, se ha utilizado los apuntes de la asignatura Electrónica Digital [5].

A partir de aquí, se construye el microprocesador dependiendo de las instrucciones que éste procese. El elegido será uno igual, o muy similar, al que se construye en los apuntes citados anteriormente.

Una vez que se sabe cuál va a ser el funcionamiento y arquitectura del microprocesador, se define el conjunto de instrucciones. Ambas partes, arquitectura y conjunto de instrucciones están completamente relacionadas.

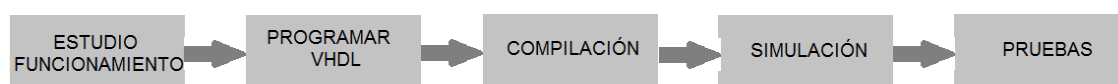
Definido el conjunto de instrucciones, se realiza una lista con los elementos o circuitos combinatoriales y secuenciales que se necesitarán para poder ejecutar dichas instrucciones (tabla 2).

Si se tiene ya la idea de qué elementos o circuitos combinatoriales se van a emplear, se procede a su implementación.

- Contador de programa
- Banco de registros
- Memoria de datos
- Sumadores
- Extensión de signo
- Memoria de instrucciones
- Unidad Aritmético lógica
- Circuito de control
- Multiplexores
- Desplazadores de posición

*Tabla 2. Elementos a implementar*

Para realizar la implementación del microprocesador MIPS, primero se realiza la implementación de cada elemento por separado, para poder comprobar su correcto funcionamiento.



*Diagrama 2. Modo de crear cada elemento o circuito*

Cada elemento sigue el siguiente proceso, el cual se puede ver en el diagrama 2:

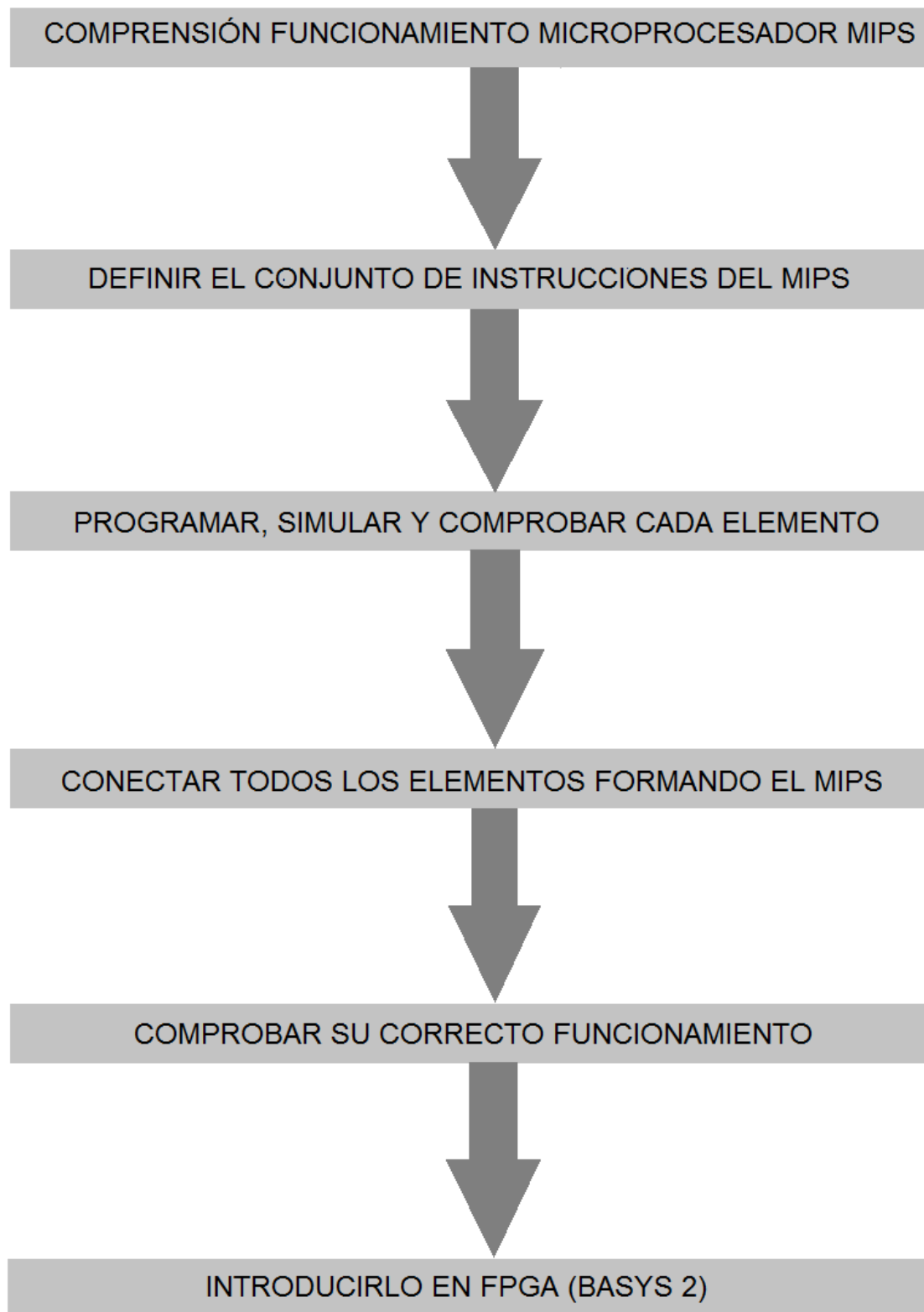
1. Comprender qué funciones debe realizar y cómo debe realizarlas.
2. Realizar implementación, en este caso se utiliza el programa Xilinx ISE.
3. Compilación correcta del elemento.

4. Simulación de todo el comportamiento mediante el programa ModelSim.
5. Prueba si es posible, debido al alto número de bits de cada elemento, con la tarjeta Basys 2.

Una vez que cada elemento ha sido creado, programado, compilado, simulado y comprobado su correcto funcionamiento, se procede a conectar cada componente como es debido, es decir, por ejemplo, la salida de la memoria de instrucciones, que será la instrucción a realizar, como existen muchos campos en la instrucción, cada conjunto de bits tendrá su correspondiente destino, bien sea a alguna entrada del banco de registros, al circuito de control, a algún multiplexor,... y así con todos los elementos que componen el microprocesador MIPS que se está implementando.

Se comprueba el correcto funcionamiento de todo el circuito unido, es decir, de lo que ya es el microprocesador MIPS, y se introduce en la tarjeta Basys 2, es decir, en la FPGA.

El diagrama seguido para la implementación completa del microprocesador MIPS sobre una FPGA se puede ver en el diagrama 3.



*Diagrama 3. Modo de implementar el microprocesador*



# **3. Desarrollo del trabajo**



### 3.1. Descripción de las instrucciones que se van a implementar en el microprocesador MIPS

Para implementar un microprocesador básico basado en una arquitectura MIPS, primero de todo se debe tener claro qué instrucciones debe realizar, qué hace cada instrucción... Por lo tanto, se comienza definiendo el conjunto de instrucciones que realizará el microprocesador.

Este conjunto de instrucciones serán de tres tipos:

- Tipo R. Instrucciones que realizan operaciones aritméticas o lógicas.
- Tipo I. Instrucciones de carga y almacenamiento tanto en registros como en memorias. También en este tipo se encuentran las instrucciones de salto condicional a otra instrucción.
- Tipo J. Instrucciones de salto incondicional.

Las instrucciones fueron expuestas en el primer apartado, pero se pueden ver de nuevo en la tabla 3. Este conjunto de instrucciones se define para ejecutar diferentes programas como multiplicaciones, almacenamientos en memoria, etc.

Instrucción (inglés)	Instrucción (español)	Mnemonic	Tipo	Opcode Field	Function Field
<b>Add</b>	Suma	Add	R	0	32
<b>Subtract</b>	Resta	Sub	R	0	34
<b>Bitwise And</b>	Producto lógico AND	And	R	0	36
<b>Bitwise Or</b>	Suma lógica OR	Or	R	0	37
<b>Shift Left Logical</b>	Rotar Izquierda	Sll	R	0	0
<b>Shift Right Logical</b>	Rotar Derecha	Srl	R	0	2
<b>Set if Less Than</b>	Activar si es Menor	Slt	R	0	42
<b>Add Immediate</b>	Añadir Dato	Addi	I	8	-
<b>Load Word</b>	Cargar Dato	Lw	I	35	-
<b>Store Word</b>	Escribir Dato	Sw	I	43	-
<b>Branch on Equal</b>	Salto si es Igual	Beq	I	4	-
<b>Branch on Not Equal</b>	Salto si no es igual	Bne	I	5	-
<b>Jump</b>	Salto incondicional	J	J	2	-

*Tabla 3. Conjunto de instrucciones implementadas*

Como se puede observar, existen tres tipos diferentes de instrucciones: tipo R, tipo I y tipo J. Todas las instrucciones constan de un total de 32 bits, aunque en un futuro se podría aumentar a 64 bits, con un previo estudio de si es viable la ampliación o no. A continuación se expone qué hace cada instrucción, cómo se ejecuta, formato de la instrucción etc.

### 3.1.1. Instrucciones tipo R

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL	rs	rt	rd	SHAMT	FUNCT	
6	5	5	5	5	6	

Tabla 4: Formato instrucciones tipo R

Estas instrucciones son las también denominadas instrucciones aritmético-lógicas, El formato que tiene este conjunto de instrucciones es el que se puede ver en la tabla 4.

Para realizar la explicación del formato y función de este conjunto de instrucciones, se explica mediante un ejemplo. El ejemplo es la instrucción llamada “add” que es una suma aritmética. La instrucción es invocada de la manera que se observa en la tabla 5.

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	ADD 1 0 0 0 0 0	
6	5	5	5	5	6	

Tabla 5: Instrucción tipo ADD

Los bits del 0 al 5, es decir, los 6 primeros bits empezando por el bit de menor orden, son los bits que indican la función que se va a realizar. En este caso la función que se implementa es “ADD” o suma. En otros casos, en los que se implementen otras funciones tipo R, serán otros bits diferentes a los que se pueden ver en la tabla 5.

Los siguientes 5 bits, los correspondientes a los bits del 6 al 10 de menor orden, corresponden a un campo denominado SHAMT. SHAMT significa “Shift Amount”, campo que se emplea para realizar cambios en los datos de los registros. Estos 5 bits son los que indican desplazamientos de los datos que existen en los registros a los que se accede. Este campo siempre será cero en operaciones como ADD, o no se tiene en cuenta este campo, pero sin embargo, para operaciones como SLL o SRL (operaciones de rotación de los datos) en el cual sí existe este desplazamiento de datos, podrán tener un valor entre 0 y  $2^5 = 32$ .

A continuación existen tres campos, cada uno de 5 bits. Estos tres campos contienen la dirección correspondiente a tres registros del banco de registros. Estos bits son los correspondientes a:

- Registro “Rd”. Bits del 11 al 15. Registro destino.

- Registro “Rt”. Bits del 16 al 20. Registro que puede actuar como fuente o como destino.
- Registro “Rs”. Bits del 21 al 25. Registro fuente.

El número total de registros que se encuentran en el banco serán de 32 registros, por lo tanto existirán 32 direcciones, y dentro de la instrucción puede repetirse la misma dirección en los tres campos, aunque cada campo se nombre de forma diferente, cosa que se hace para designar a cada campo de la instrucción y diferenciarlo de los otros dos.

Por último, existe el campo SPECIAL, o también llamado OpCode, que tiene 6 bits, del bit 26 al 31. En el caso de operaciones tipo R siempre será el conjunto formado por seis ceros: 000000.

El número total de bits de la instrucción sumando cada campo es de 32 bits, y para instrucciones tipo R siempre tiene este formato. Aunque todas las instrucciones tengan el mismo formato, cada tipo de instrucción, dentro de este conjunto tipo R, tiene una misión diferente, y unos bits distintos a las demás.

### 3.1.1.1. Instrucción “ADD”

Instrucción de suma aritmética. La suma se realiza en complemento a dos como se puede ver en el siguiente ejemplo:

$$\begin{array}{r}
 00010101011111011111000001101101 \\
 + 100000101010101010101010100000 \\
 \hline
 10011000001010001001101100001101
 \end{array}
 \begin{array}{l}
 \leftarrow \text{Rs (360575085)} \\
 \leftarrow \text{Rt (-2102744416)} \\
 \leftarrow \text{Rd (-1742169331)}
 \end{array}$$

Se realiza en complemento a dos para tener en cuenta si los datos son positivos o negativos.

El formato de la instrucción se puede ver en la tabla 5 y su llamada es:

- ADD rd, rs, rt

Con esta llamada realizada y teniendo en cuenta el ejemplo anterior, en el registro cuya dirección viene definida por el valor de rd se almacena la suma obtenida de los datos procedentes de los registros cuyas direcciones vienen definidas por rs y rt.

El campo SPECIAL u OpCode es de todos ceros, como en todos los tipo R, y el campo de función es: 100000, diferente que para otras operaciones tipo R.

### 3.1.1.2. Instrucción "SUB"

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SUB 1 0 0 0 1 0	
6	5	5	5	5	6	

Tabla 6: Instrucción tipo SUB

Instrucción de resta aritmética. La resta también se realiza en complemento a dos, como se muestra en el ejemplo:

$$\begin{array}{r}
 11010101011111011111000001101101 \quad \leftarrow \text{Rs}(-713166739) \\
 - 00000010101010101010101010100000 \quad \leftarrow \text{Rt}(44739232) \\
 \hline
 11010010110100110100010111001101 \quad \leftarrow \text{Rd}(-747905971)
 \end{array}$$

El formato que tiene la instrucción "SUB" se puede ver en la tabla 6 y su llamada es la siguiente:

- SUB rd, rs, rt

Cuando se realiza esta llamada y teniendo en cuenta el ejemplo anterior, en el registro cuya dirección viene definida por el valor de rd se almacena la resta obtenida de los datos procedentes de los registros cuyas direcciones vienen definidas por rs y rt.

El campo SPECIAL u OpCode es de todos ceros, como en todas las instrucciones tipo R, y el campo de función es: 100010.

### 3.1.1.3. Instrucción "AND"

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	AND 1 0 0 1 0 0	
6	5	5	5	5	6	

Tabla 7: Instrucción tipo AND

Con esta instrucción se realiza un producto lógico bit a bit de los datos almacenados en las direcciones de los campos de los registros. A continuación se muestra un ejemplo de la operación:

$$\begin{array}{r}
 11010101011111011111000001101101 \quad \leftarrow \text{Rs} \\
 \text{AND } 00000010101010101010101010100000 \quad \leftarrow \text{Rt} \\
 \hline
 00000010001010001010000000100000 \quad \leftarrow \text{Rd}
 \end{array}$$













- LW rt, offset (rs)

Una vez queda invocada la instrucción, lo que desarrolla es cargar el dato que existe en una memoria de datos en el banco de registros. La dirección del registro está dada por el campo rt, y la dirección a la que se accede en memoria para cargar el dato deseado, se obtiene mediante una operación de suma en la Unidad Aritmético Lógica. La suma que se realiza es, los datos obtenidos del registro rs y los datos del campo offset, previamente tratados mediante un ampliador de bits que respeta el signo.

### Ejemplo

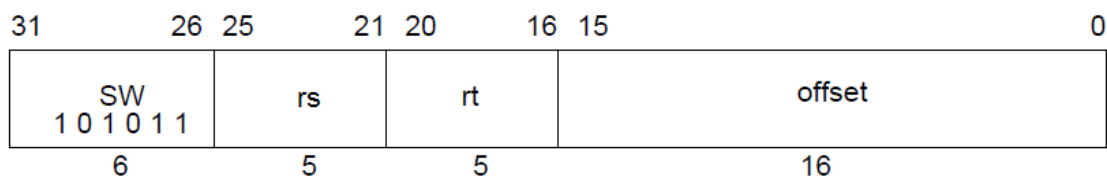
- LW → 100011
- Rt → 01010
- Rs → 10001
- Offset → 1000100010111100

Una vez introducida esta instrucción con los parámetros que se pueden ver arriba, se cargará en el registro Rt, cuya dirección es 01010, los datos obtenidos de la memoria de datos.

La dirección a la que se accede en la memoria de datos es la suma de los datos del registro Rs (dirección 10001), más el dato offset ampliado de signo, que en este caso concreto, basta con añadir 16 unos a la izquierda, es decir, añadir bits más significativos quedando: 11111111111111111000100010111100 (Complemento a dos).

No obstante, cuando se definan todos los elementos que se necesitan para realizar esta y las demás instrucciones, quedará el concepto más claro y conciso.

### **3.1.2.2. Instrucción “SW”**



*Tabla 14: Instrucción tipo SW*

Se denomina “Store Word” o “Escribir Dato”. Viendo la tabla 14, se llega a la conclusión que el formato es el mismo que para la instrucción LW, con la salvedad de que cambia el OpCode. En este caso se trata del identificador creado por los siguientes bits: 101011.





El realizar un salto de instrucción supone una ruptura con la ejecución consecutiva de instrucciones del algoritmo creado.

La invocación que se realiza para ejecutar esta instrucción es:

- BEQ rs, rt, offset

El comportamiento que tiene esta instrucción, después de realizar la invocación anterior, es realizar un salto de instrucción siempre y cuando los registros rs y rt sean idénticos. Con esto no se quiere decir que las direcciones sean idénticas, sino que, los datos contenidos en esas direcciones sean iguales. Si no son iguales, no se produce salto de instrucción.

El salto que se realiza a otra instrucción, viene determinado por el campo denominado como offset.

### Ejemplo

- BEQ → 000100
- Rt → 01010
- Rs → 10001
- Offset → 000000000100001

Con los mismos datos que los ejemplos anteriores, pero cambiando el identificador por el de la instrucción BEQ.

Primero se obtienen los datos que pertenecen al registro rt de dirección 01010, y los datos pertenecientes al registro rs de dirección 100001.

Estos datos serán introducidos directamente a la Unidad Aritmético Lógica para ser comparados entre ellos. Si estos datos son iguales se activa una señal para indicar que se produzca un salto de instrucción, y si no lo son, pues esta señal queda inactiva y no se produce el salto de instrucción. El salto determinado en este caso por el campo offset es de: 000000000100001.

### **3.1.2.5. Instrucción "BNE"**

31	26	25	21	20	16	15	0
BNE 0 0 0 1 0 1		rs	rt		offset		
6		5	5		16		

*Tabla 17: Instrucción tipo BNE*

Se denomina “Branch on Not Equal” o “Salto si No es Igual”. Se trata del caso opuesto a la instrucción anterior, es decir, se produce un salto de instrucción siempre y cuando los dos datos que se comparen no sean iguales. El formato es exactamente al caso anterior cambiando el OpCode: 000101.

La invocación que se realiza para ejecutar esta instrucción es:

- BNE rs, rt, offset

El comportamiento que tiene esta instrucción, después de realizar la invocación anterior, es realizar un salto de instrucción siempre y cuando los registros rs y rt no sean idénticos, es decir, los datos contenidos en las direcciones de los registros no sean iguales. Si son iguales, no se produce salto de instrucción.

El salto que se realiza a otra instrucción, viene determinado por el campo denominado como offset.

**Ejemplo**

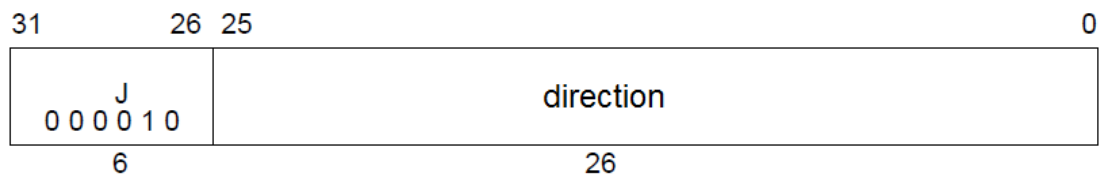
- BNE → 000101
- Rt → 01010
- Rs → 10001
- Offset → 000000000100001

Con los mismos datos que los ejemplos anteriores, pero cambiando el identificador por el de la instrucción BNE.

Primero se obtienen los datos que pertenecen al registro rt de dirección 01010, y los datos pertenecientes al registro rs de dirección 10001.

Estos datos serán introducidos directamente a la Unidad Aritmético Lógica para ser comparados entre ellos. Si estos datos son distintos se activa una señal para indicar que se produzca un salto de instrucción, y si no lo son, pues esta señal queda inactiva y no se produce el salto de instrucción. El salto queda determinado por el campo denominado como offset y en este caso es: 000000000100001.

**3.1.3. Instrucciones tipo J**



*Tabla 18: Instrucción tipo J*



Se trata de una sola instrucción. Se denomina “Jump” o “Salto incondicional”.

Como se puede ver en la tabla 18, el formato es completamente diferente al formato de las instrucciones anteriores, sólo existe una coincidencia. Ésta es que los seis bits más significativos, es decir, los bits que se sitúan entre el 26 y el 31, ambos inclusive, pertenecen al campo denominado OpCode o SPECIAL, y es el identificador de esta instrucción.

Los otros 26 bits pertenecen a otro campo llamado dirección. Este campo determina la extensión del salto de instrucción a realizar.

El cálculo del salto se produce de la siguiente manera. Primero, se desplaza dos bits a la izquierda los datos del campo denominado direction (dirección). Se obtiene 28 bits añadiendo dos ceros en los bits menos significativos de este campo. Segundo, se añaden los 4 bits más significativos que salen después de sumar 4 unidades a la instrucción que se está realizando, es decir, a la salida del contador de programa, para así obtener los 32 bits.

La invocación que se realiza para ejecutar esta instrucción es:

- J direction

### **Ejemplo**

- J → 000010
- Direction → 010100000000000000000000

En este caso el identificado u OpCode indica que se trata de una instrucción tipo J, es decir, se va a realizar un salto de instrucción.

Este salto puede ser hacia delante o hacia detrás, dependiendo de si el número es positivo o negativo (Complemento a 2).

En este ejemplo, como el primer dígito es un cero, se trata de un número positivo, por lo tanto el salto será hacia delante. Este salto será de tantas instrucciones como indica el dato contenido en el campo direction,

## **3.2. Obtención de los elementos que compondrán el microprocesador MIPS**

Todas las figuras empleadas en este apartado se han obtenido del documento “Diseño del procesador MIPS R2000” [5].

El microprocesador MIPS que se diseña constará de una serie de componentes, elementos o circuitos combinacionales o secuenciales. Para determinar cuáles son, lo primero de todo es saber qué instrucciones se van a implementar. Estas instrucciones ya se han desarrollado en el apartado 3.1.

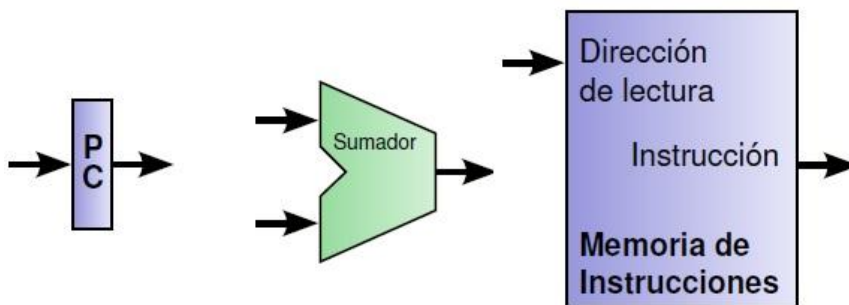
Una vez que se tiene definido el conjunto de instrucciones a implementar, se trata de desglosar qué actividad va a llevar a cabo. Como también está explicado en el capítulo 3.1, sólo queda el último paso para definir qué elementos se necesitan en el desarrollo del microprocesador.

Conforme se necesite realizar una operación u otra, se debe disponer de una serie de componentes. Para ello se desglosa de la siguiente forma:

1. Elementos necesarios para instrucciones tipo R
2. Elementos necesarios para instrucciones tipo I
3. Elementos necesarios para instrucciones tipo J

### **3.2.1. Elementos comunes a cualquier tipo de instrucción**

Para definir el MIPS se deben realizar una serie de pasos teniendo en cuenta las instrucciones implementadas. Los dos primeros pasos son idénticos para todas las instrucciones que posee el microprocesador desarrollado.



*Figura 13: Contador de programa y memoria de instrucciones*

El primer paso consiste en leer en un “contador de programa” (PC) una información. Este contador de programa consiste en un registro a cuya entrada llega una serie de datos y genera en la salida dichos datos después de un ciclo de reloj. Es decir, genera a su salida el estado siguiente que se debe ejecutar.

Se trata básicamente de un contador con posibilidad de carga en paralelo. La carga se produce para las instrucciones de salto, bien sean saltos condicionales o saltos incondicionales.

El contador de programa va generando los siguientes estados gracias a un sumador que se añade al registro, pues así, como su propio nombre indica, se construye el contador, realizando una suma de cuatro unidades para llegar al estado siguiente.

La información se manda a una memoria, tipo ROM, en la cual se leerá el contenido que esta posea. Conforme sea el contenido, esta memoria cargará una instrucción que es previamente almacenada, que es la que indicará la operación u operaciones a realizar dentro del MIPS. Esta memoria se denomina “memoria de instrucciones”.

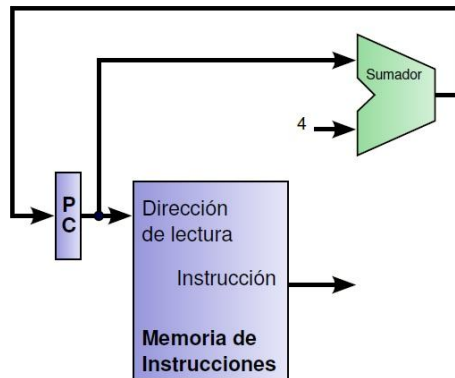


Figura 14: Unión contador de programa con la memoria de instrucciones

A partir de estos dos elementos no siempre se realiza las mismas operaciones, depende de las instrucciones. Aún así, algunas instrucciones usan los mismos elementos pero no con los mismos campos, como se podrá ver a continuación.

### 3.2.2. Elementos instrucciones tipo R

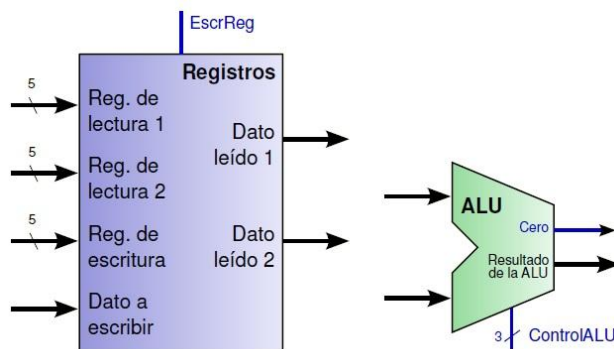


Figura 15: Banco de registros y ALU

Para este primer tipo de instrucciones, teniendo en cuenta que todas son operaciones aritmético-lógicas, a parte del contador de programa y de la memoria de instrucciones ya definidos en el apartado 3.2.1, se necesitarán dos elementos más: un banco de registros, y una Unidad Aritmético lógica.

El primero de los elementos, el banco de registros, es el que recibe primero la instrucción generada en la memoria de instrucciones. En él, y como se puede observar en la figura 15, hay una serie de entradas y salidas:

- Entradas:
  - Registro de lectura 1 (5 bits)
  - Registro de lectura 2 (5 bits)
  - Registro de escritura (5 bits)
  - Dato a escribir (32 bits)
  - EscrReg (habilitación de escritura, 1bit)
  
- Salidas:
  - Dato leído 1 (32 bits)
  - Dato leído 2 (32 bits)

Como se puede ver, el banco de registros posee dos entradas de lectura (dirección de los registros) que a la salida sacan los datos obtenidos en esos registros, y también posee una entrada a otro registro (dirección del registro), que teniendo habilitada la opción de escritura ( $\text{EscrReg} = 1$ ), permite introducir un nuevo dato en él.

Por otro lado, se tiene la Unidad Aritmético Lógica (ALU), elemento destinado a realizar las operaciones deseadas, tanto aritméticas como lógicas. Consta de las siguientes entradas y salidas, tal y como se puede ver en la figura 15:

- Entradas:
  - Entrada 1 (32 bits)
  - Entrada 2 (32 bits)
  - Control ALU (3 bits)
  
- Salidas:
  - Resultado ALU (32 bits)
  - “cero” (1 bit)

A las entradas, denominadas como uno y dos, pueden acceder los datos de los registros leídos en el banco de registros, o pueden acceder otros datos, dependiendo que instrucción sea, como se podrá ver más adelante.

La operación que realiza este componente, viene determinada por los tres bits correspondientes a la entrada llamada Control ALU. Alternando estos tres bits se

consiguen un total de ocho combinaciones, y dependiendo de cuál sea se procederá con una operación de suma, o de resta, o operación lógica and,...

Completada ya la operación, el resultado es obtenido por la salida de 32 bits, con la peculiaridad de que, si este resultado está formado completamente por ceros, es decir, los 32 bits son ceros, la salida denominada como “cero”, tendrá un valor lógico de uno, y si no son todos los bits ceros, pues el valor de esta salida será de cero. Esta salida nos indicará si se produce un salto de instrucción o no como se verá en las instrucciones “BEQ” y “BNE” (estas dos son de tipo I).

A continuación se desglosan todas las opciones que existen con operaciones tipo R para ver las conexiones que existen entre estos cuatro elementos descritos.

### 3.2.1.1. Instrucciones ADD, SUB, OR, AND y SLT

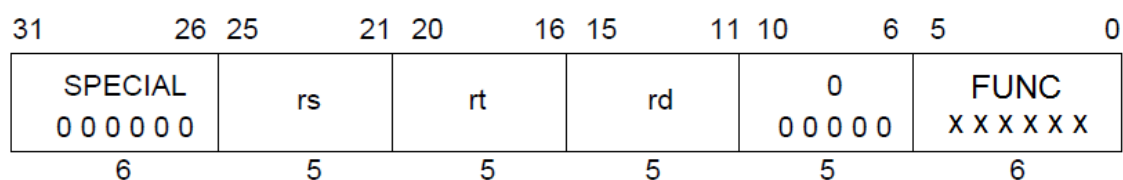


Figura 16: Formato instrucciones ADD, SUB, OR, AND y SLT

Como se puede ver en la figura 16, el formato que tienen las cinco instrucciones es idéntico, la única diferencia es el campo FUNC (función), que será diferente para cada una.

Para todas ellas se emplean los cuatro elementos descritos. Una vez que el contador de programa ha enviado la información a la memoria de instrucciones, y se ha generado la instrucción deseada, con el formato de la figura 16, cada campo será recibido por diferentes elementos.

El primer campo, el denominado como SPECIAL, o también llamado OpCode, se envía a un circuito de control que se verá más adelante cuando se tengan todas las señales de control que se desean. Estas señales de control son por ejemplo la habilitación de escritura en el banco de registros (EscrReg), entre otras muchas.

Los siguientes tres campos, los registro rs, rt, y rd son los que se recogen en el banco de registros. Cada uno de ellos va a una entrada de 5 bits de este banco de registros:

- Rs → registro de lectura 1
- Rt → registro de lectura 2
- Rd → registro de escritura

Estos 5 bits de cada registro corresponden a su dirección dentro del banco, es decir, a la dirección a la que la instrucción manda que se acceda.

El campo denominado SHAMT, que para estas operaciones es todo ceros, no se emplea para nada, pues no afecta a las operaciones a realizar.

El último campo es llevado a otro circuito de control. Este circuito es el que controla la operación que realizará la ALU. Dependiendo de cuál de las cinco operaciones indique la instrucción, el circuito de control generará una combinación de tres bits.

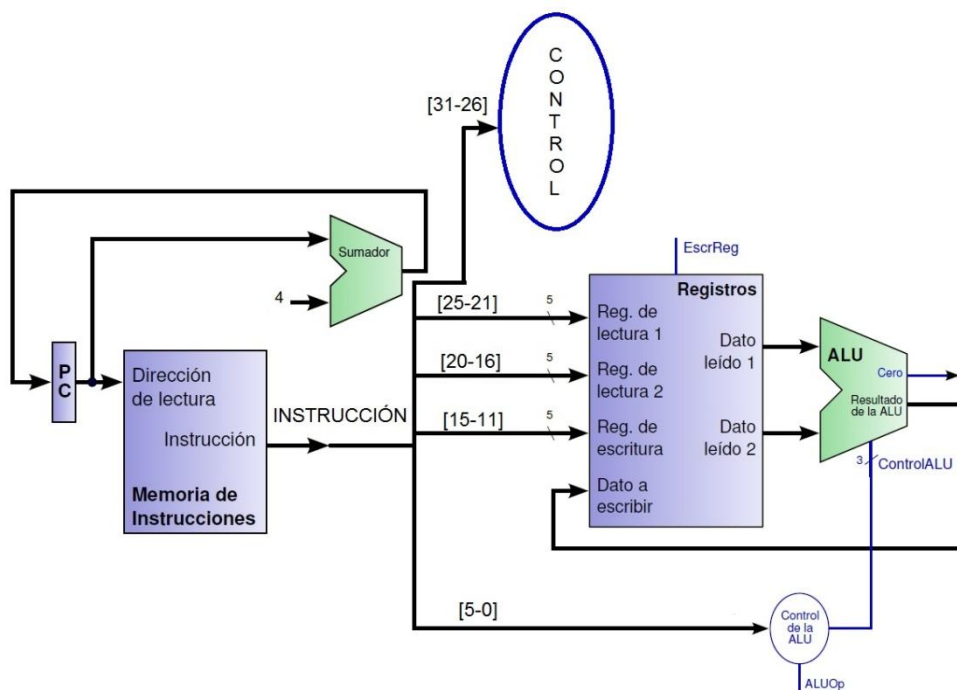


Figura 17: Conexiones de elementos para instrucciones ADD, SUB, AND, OR y SLT

Una vez dirigidos todos los campos de la instrucción, en el banco de registros, como se han introducido dos direcciones de registros para ser leídos, a cada salida del banco se obtienen los datos contenidos en dichos registros.

Los datos de cada registro son introducidos a la Unidad Aritmético Lógica, para que se realice la operación determinada por el circuito de control, o a más alto nivel por la instrucción.

Cuando se haya realizado la operación, la salida de la ALU se llevará a la entrada del registro que se llama "Dato a escribir". Este dato será escrito en la dirección de registro dada por el campo rd de la instrucción, siempre y cuando la habilitación de escritura esté a nivel alto, cosa que siempre estará cuando se realicen cualquiera de estas cuatro instrucciones. En este caso la salida "cero" de la ALU no tiene ninguna utilidad.

El esquema que describe todas las conexiones expuestas se muestran en la figura 17.

### 3.2.1.2. Instrucciones SLL y SRL

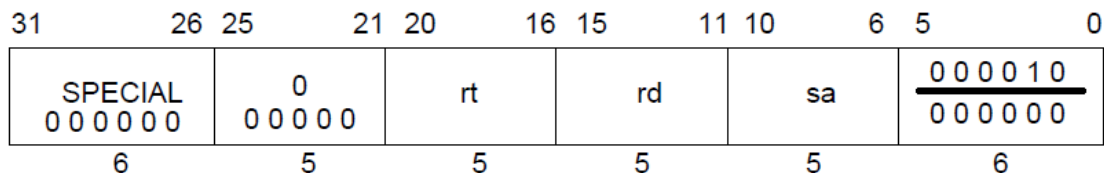


Figura 18: Formato instrucciones SLL y SRL

Como se puede ver en la figura 18, el formato de la instrucción es un poco diferente a las otras instrucciones. La principal diferencia es que se cambia el campo de todo ceros, que en las instrucciones anteriores se almacenaba en el campo SHAMT, y ahora se almacena en el campo que sigue al OpCode. Este campo ahora no se empleará para nada.

Se introducirá en la primera entrada de lectura del banco de registros el campo correspondiente a rt [20-16], y en la entrada de escritura se introducirá el campo rd [15-11].

El campo correspondiente a SHAMT, es decir los bits que van desde el 6 al 10, serán los que determinen el número de bits que se van a desplazar los datos que corresponden al registro cuya dirección ha sido data por rt.

Existen algunas similitudes, como por ejemplo, que el campo SPECIAL u OpCode es exactamente igual, pues es igual para todas las instrucciones tipo R. También se accede al banco de registros, pero sólo a un registro de lectura y otro de escritura.

La división de los campos de la instrucción es exactamente igual, aunque su contenido varíe ligeramente.

Como son sólo dos instrucciones, en la figura 18, se observa las dos combinaciones del campo FUNC posibles: para SRL 000010 y para SLL 000000.

Como se puede intuir, las conexiones no van a ser iguales que las realizadas para las instrucciones del apartado 3.2.2.1. Sin embargo, la primera parte, como ya se ha dicho anteriormente, es igual que en el apartado anterior. Por primera parte se entiende a la formada por el contador de programa (PC) y la memoria de instrucciones.

A partir de que se genera la instrucción, cualquiera de estas dos, es cuando existen las diferencias, aunque el primer campo, el denominado como OpCode y que comprende los bits del 26 al 31, se envían al circuito de control, como en el caso

de las otras instrucciones, y este circuito hará las mismas funciones que para las anteriores, ya que es idéntico OpCode para todas las instrucciones tipo R.

El siguiente campo, el formado por cinco ceros, como ya se ha dicho antes, no se utiliza y no tiene ningún peso en este tipo de instrucciones.

Los dos siguientes campos, se introducen a las entradas del banco de registro, para obtener la dirección de los registros a los que se accede, uno de lectura, y otro de escritura.

El campo SHAMT, comprendido por cinco bits se introduce en la Unidad Aritmético Lógica, no sin antes pasar por un extensor de bits, que añadirá 27 ceros en los bits más significativos, y así completar un total de 32 bits, que es el número de bits que requieren las entradas de la ALU.

Por último, el campo llamado FUNC se enviará al circuito de control denominado como Control ALU, para que este genere la operación que ha de desarrollar la ALU.

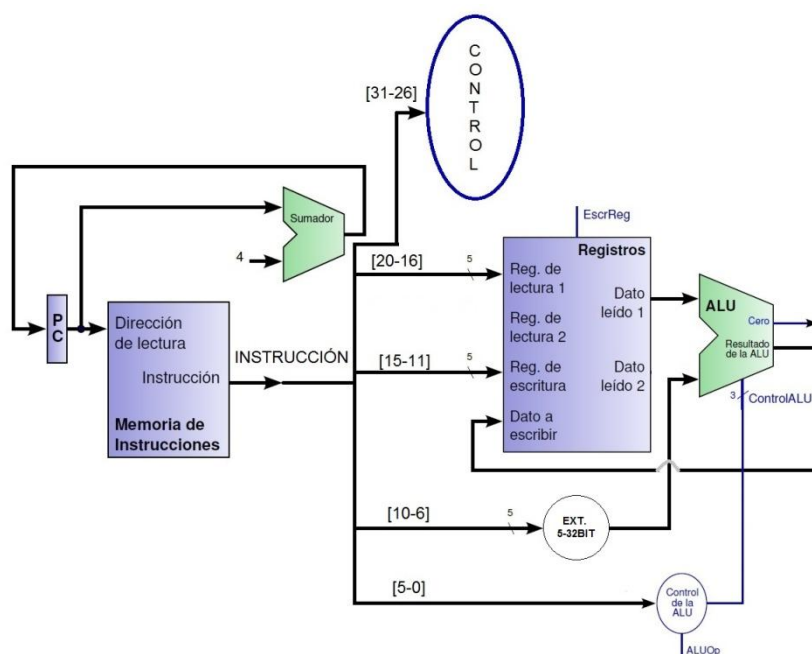


Figura 19: Conexiones de elementos para instrucciones SLL y SRL

### 3.2.3. Elementos instrucciones tipo I

Este tipo de instrucciones, además de utilizar elementos anteriores, como son por ejemplo, el contador de programa, la memoria de instrucciones, el banco de registros o la Unidad Aritmético Lógica, también incorpora otros elementos.



Estos elementos son la memoria de programa y un extensor de signo, para tres instrucciones (Lw, Sw y Addi) y un sumador y otro capaz de realizar un desplazamiento de bits para otras dos instrucciones (Beq y Bne).

Estos elementos citados se pueden ver en la figura 20. El sumador que se emplea en este caso, no realiza la misma función que en el caso de instrucciones tipo R. Éste servirá para realizar el cálculo del salto que se ha de realizar para cambiar de instrucción (Beq y Bne).

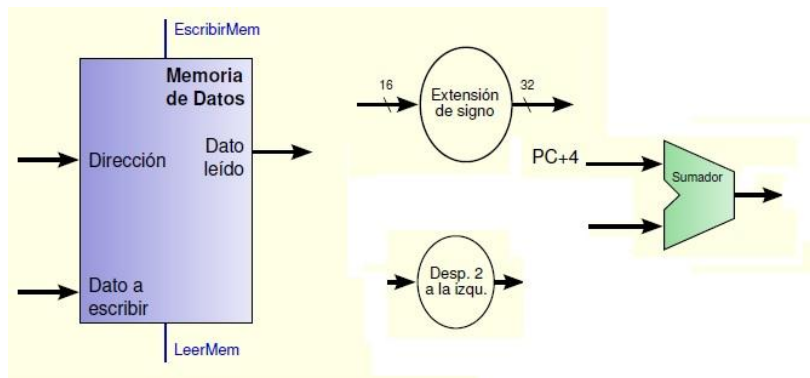


Figura 20: Memoria de Datos, Extensión de signo, Desplazar dos izquierda y sumador

El elemento principal que se añade para este tipo de instrucciones es la memoria de datos, la cual consta de:

- Entradas:
  - Dirección (32 bits)
  - Dato a escribir (32 bits)
  - EscribirMem (1 bit)
  - LeerMem (1 bit)
- Salida:
  - Dato leído

A diferencia de la memoria de instrucciones, explicada anteriormente, la memoria de datos es una memoria tipo RAM. Esto quiere decir, que se trata de una memoria de lectura y también de escritura, mientras que la de instrucciones (ROM), es de solo lectura.

Como se puede ver en la figura 20, se tiene 4 entradas para la memoria de datos, dos de las cuales son de control, una para habilitar la lectura (LeerMem) y la otra

para habilitar la escritura (EscribirMem). Las otras dos entradas indican, la dirección a la que se accede en la memoria (dirección), y el dato que se desea introducir en esa dirección (dato a escribir), siempre y cuando, la habilitación de escritura se encuentre a nivel alto.

La única salida que posee la memoria de datos es la llamada dato leído, la cual se obtiene sólo si la habilitación de lectura se encuentra activada. El dato que se obtiene es el dato almacenado en la dirección de la memoria a la cual se accede.

Los otros tres elementos que se visualizan en la figura 20, son elementos que realizan operaciones determinadas para poder llevar a cabo la instrucción.

El extensor de signo es un elemento que recibe 16 bits, y los aumenta hasta 32 bits, siempre respetando el signo del conjunto de bits a la entrada (complemento a dos). Por ejemplo, si el bit más significativo a la entrada de este elemento es un cero (positivo), se completan los otros 16 bits más significativos con ceros, y si es un uno (negativo), pues se completan con unos.

El elemento llamado Desp. 2 a la izquierda realiza lo siguiente: recibe 32 bits y mueve a todos dos posiciones a la izquierda, añadiendo dos ceros a la derecha, en los bits menos significativos. Destacar que a la salida saca también 32 bits, por lo tanto los dos bits más significativos que llegan a la entrada se pierden.

Por último, el sumador lo que hace es sumar dos conjuntos de bits que le llegan a sus entradas. Los dos son de 32 bits, y el resultado también es de 32 bits. Se diferencia del otro sumador, que en este caso no siempre se suma el valor cuatro, dependerá del salto que se desea realizar.

Como se ha realizado en la instrucciones de tipo R, aquí también se realiza un desglose de cómo actúa cada una, pues se pueden diferenciar también tres grupos.

### 3.2.1.3. Instrucción ADDI

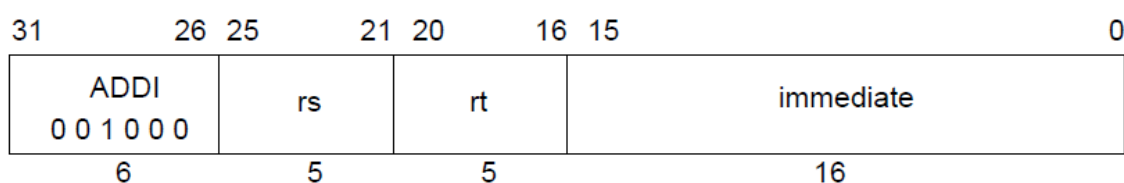


Figura 21: Formato instrucción ADDI

Al observar la figura 21, se puede ver las diferentes partes o grupos en los que se divide la instrucción addi.

El primer campo formado por 6 bits, los más significativos, es el perteneciente al OpCode. Este campo es el que proporciona al circuito de control la información

necesaria para que éste procese que señales mandar a los diferentes elementos que componen el MIPS. Por ejemplo, como en este caso se introduce un nuevo dato en el banco de registros, la entrada de habilitación de escritura deberá estar activa. El circuito de control se explica más adelante.

Los dos siguientes campos o grupos dentro de la instrucción, pertenecientes a los bits que van del 25 al 21 y del 20 al 16, son los campos que contienen la dirección de los dos registros a los que se van a acceder. El primero posee la dirección del registro del que se extraerá un dato, es decir, registro de lectura, y el segundo tiene la dirección del registro donde se realizará la escritura de un nuevo dato, después de realizar una serie de cálculos con el registro leído.

El último grupo de bits es el denominado como immediate (inmediato) en la figura 21. Este campo proporciona un conjunto de 16 bits, los cuales serán sumados al dato obtenido del banco de registros, cuya dirección daba el campo rs. Estos 16 bits pasan previamente por el elemento que aumenta hasta 32 bits respetando el signo, y que se llama Extensión de signo.

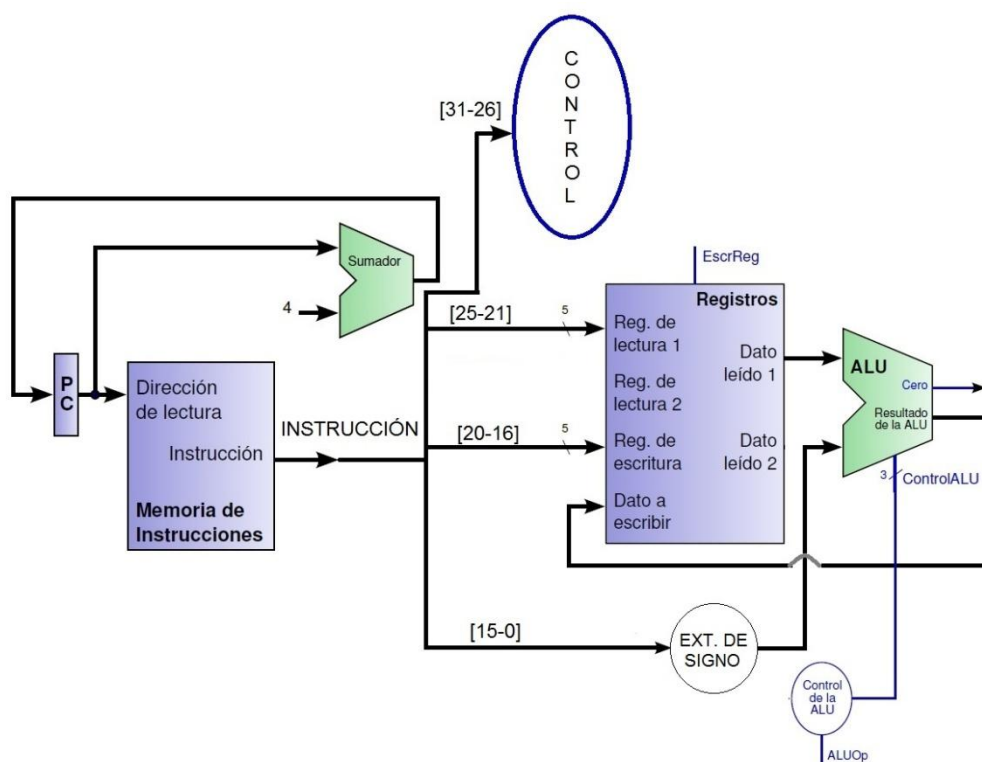


Figura 22: Conexiones de elementos para instrucciones ADDI

Con todos los campos definidos, se puede llegar a la conclusión de que las conexiones entre los elementos no van a ser iguales que para las instrucciones anteriores (tipo R).

Sin embargo, la generación de la instrucción, como ya se ha dicho anteriormente, se realiza igual que en los casos pasados. El contador de programa genera una dirección a la que se accede en la memoria de instrucciones, y ésta, a su salida obtiene la instrucción.

La primera diferencia, es la inclusión del elemento de extensión de signo, para los 16 bits menos significativos de la instrucción. Por lo tanto, los bits correspondientes del 0 al 15, serán introducidos en este elemento. La salida de éste, será introducida a una entrada de la Unidad Aritmético Lógica, para así poder realizar la operación de suma que se lleva a cabo en esta instrucción.

Otra diferencia con las instrucciones tipo R, es que los bits correspondientes al campo denominado como rt, es decir los bits del 16 al 20, son introducidos en la dirección de escritura del banco de registros, para poder escribir el resultado que se obtenga en la ALU en esa dirección.

Existen dos similitudes con las instrucciones tipo R. La primera es el campo OpCode, que como ya se ha dicho, es el que se introduce al circuito de control.

La segunda similitud, aunque en este caso sólo es igual que para el primer conjunto de instrucciones tipo R (ADD, SUB, SLT, AND y OR), el campo cuyos bits son los del intervalo del 21 al 25 (rs), se introduce en el puerto de dirección del registro de escritura 1 del banco de registros, para que éste saque por su salida el dato que se encuentra en esa dirección, y así la ALU pueda realizar la operación de suma, pues la salida del banco de registro se conecta con una entrada de la ALU.

Todas las conexiones que se han explicado, se pueden visualizar en la figura 22.

#### 3.2.1.4. Instrucciones SW y LW

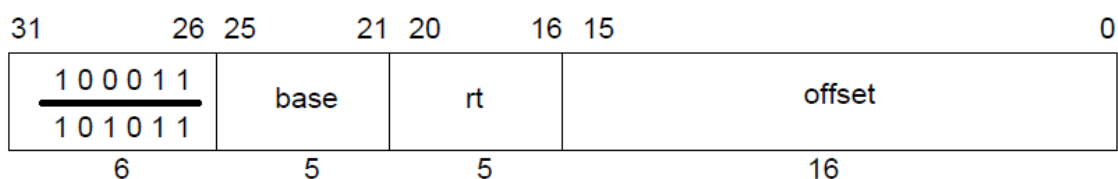


Figura 23: Formato instrucciones SW y LW

Como se puede ver en la figura 23, el formato de estas dos instrucciones es igual al anterior en el aspecto en que se divide en los mismos campos y con las mismas direcciones, aunque se nombren de manera diferente.

En primer lugar tenemos dos nuevos OpCodes: 100011 para la instrucción LW y 101011 para SW. La primera instrucción carga un dato almacenado en la memoria

de datos y lo graba en un registro. La segunda hace la operación inversa, carga el dato de un registro y lo almacena en la memoria de instrucciones.

A continuación, el siguiente campo, aunque esté denominado como base, hace referencia a la dirección de un registro del banco. A partir de aquí conviene diferenciar entre las dos instrucciones.

### 3.2.3.1.1. Instrucción SW

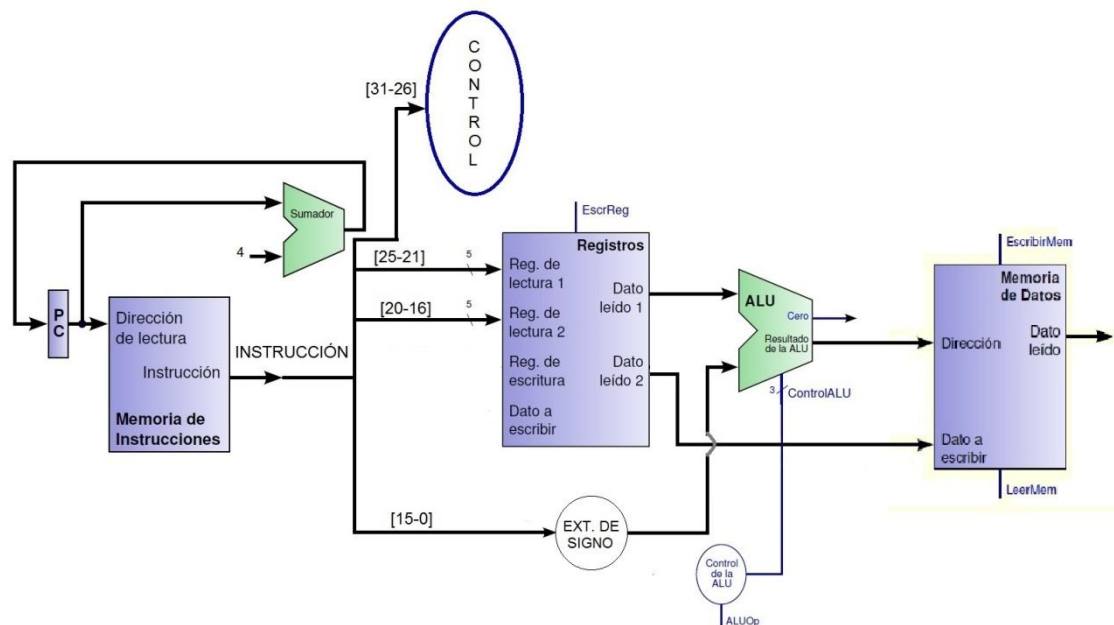


Figura 24: Conexiones de elementos para instrucciones SW

Los dos primeros pasos, como en todas las instrucciones son idénticos, el contador de programa genera una dirección para acceder a la memoria de instrucciones, y así poder tener a su salida la instrucción deseada.

Como el campo OpCode ha quedado explicado cuáles son sus dígitos, simplemente se realiza la conexión de estos 6 bits al circuito de control, para que éste genere las señales de habilitación o deshabilitación que hagan que cada elemento del microprocesador funcione como ha de hacerlo con esta instrucción.

El siguiente campo, como también se ha dicho antes, aunque se nombre como base, corresponde a la dirección de un registro. Esta dirección da como resultado un dato que está almacenado, y a la vez, este dato corresponde con la dirección de la memoria de datos donde se introducirá el nuevo dato.

Sin embargo, la dirección de la memoria no queda definida sólo por el dato del registro, sino que a ese dato, se le debe sumar el campo denominado como offset (16 bits). Pero como este campo necesita ser aumentado hasta 32 bits para poder

ser sumado con el dato del registro, antes de conectarse con la ALU, previamente se pasa por el elemento llamado Extensión de signo.

Tanto la salida de este Extensión de signo, como la salida del banco de registros (Dato leído 1), se conectan a las dos entradas de la Unidad Aritmético Lógica, para que ésta sume ambas, y ya de cómo resultado la dirección de la memoria a la que se ha de acceder.

Por último queda por mencionar el campo rt (bits del 20 al 16). Éste corresponde con otra dirección del banco de registros. Se necesita que esta dirección sea leída para que en la salida llamada Dato leído 2, de un dato de 32 bits, el cual será el que se introduzca en la dirección de memoria antes generada, siempre y cuando esté habilitada la señal de escritura de la memoria de datos.

Todas las conexiones realizadas se pueden visualizar en la figura 24.

### 3.2.3.1.2. Instrucción LW

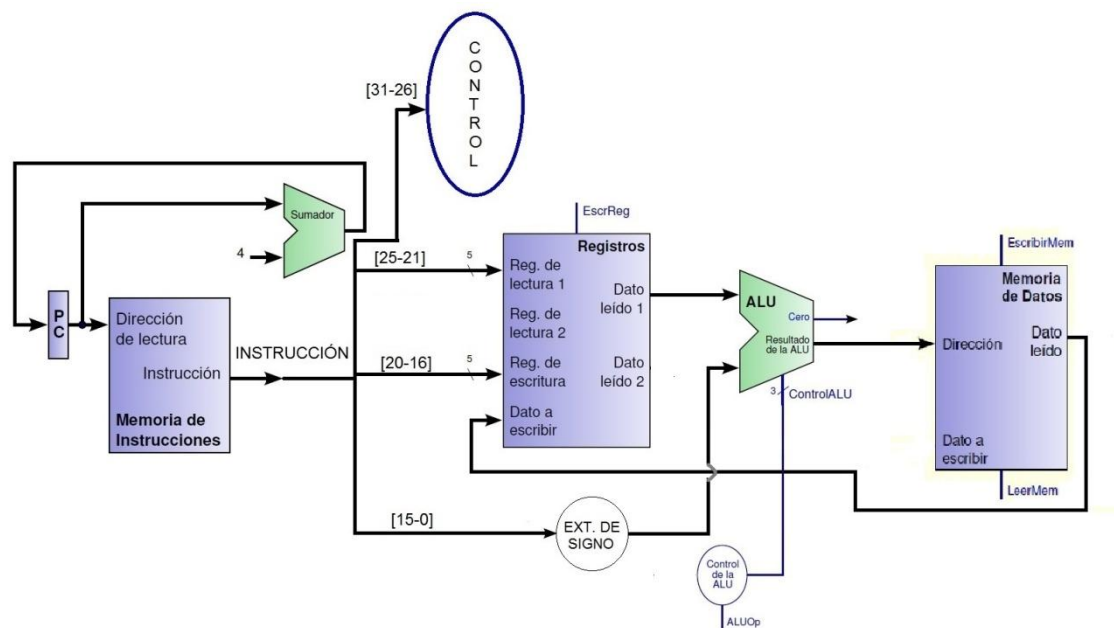


Figura 25: Conexiones de elementos para instrucciones LW

Las conexiones del contador de programa y memoria de instrucciones son idénticas a todas las instrucciones, ya que estos dos primeros pasos siempre son iguales para así poder generar la instrucción deseada, utilizando la dirección de la memoria de instrucciones que indica el contador de programa.

En esta instrucción, el campo OpCode es igual que en todas las demás, se conecta directamente con el circuito de control para que éste genere las señales de

habilitación necesaria para el correcto comportamiento del microprocesador conforme a esta instrucción.

Los demás campos son iguales que la instrucción SW, pero no tienen el mismo comportamiento dentro del microprocesador. En esta instrucción (LW) se carga un dato en el banco de registros procedente de la memoria, y en la instrucción SW se escribe en la memoria un dato procedente del banco de registros.

En este caso, el registro rt (bits del 20 al 16) no da la dirección del dato que se introduce en la memoria, sino que da la dirección del banco de registros en la cual se introducirá el dato que se genera. Por lo tanto, en esta instrucción se carga en el banco de registros un nuevo dato.

El dato que se almacenará en el banco de registros se obtiene de la memoria de datos. Para llegar a él, se debe obtener una dirección de esta memoria. Esta dirección se obtiene mediante los campos denominados base y offset.

Primero el campo base, da la dirección de un registro para obtener un dato de 32 bits que será introducido en una entrada de la Unidad Aritmético Lógica.

El segundo dato a introducir en la ALU será el obtenido del campo offset de 16 bits, pero por este motivo de tener sólo 16 bits, debe ser incrementado, mediante el elemento Extensión de signo, hasta 32 bits.

Ambos elementos serán sumados en la ALU para generar la dirección de la memoria de datos a la que se ha de acceder. Una vez se consiga la dirección, la memoria proporciona un dato de 32 bits también almacenado en esa dirección y, ese dato, será el que se introduzca en el banco de registros en la dirección dada por rt, siempre que la habilitación de escritura lo permita.

Todas las conexiones realizadas se pueden visualizar en la figura 25.

### 3.2.1.5. Instrucciones BEQ y BNE

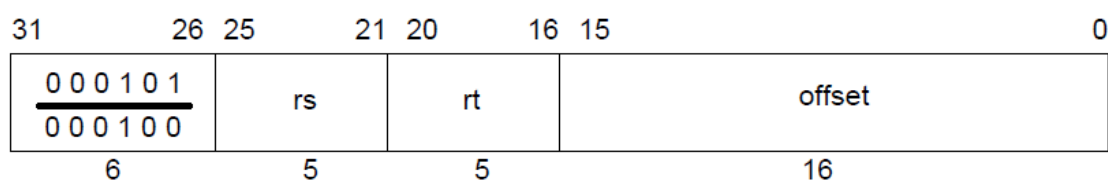


Figura 26: Formato instrucciones BEQ y BNE

El formato de estas dos instrucciones, como se puede comprobar visualizando la figura 26, sigue siendo idéntico a las instrucciones pasadas tipo I, salvo por el nombre de cada campo que es diferente. Ambas instrucciones realizan operaciones

muy diferentes a las instrucciones pasadas, aunque entre ellas dos realizan lo mismo, con una pequeña variación.

La única diferencia que existe entre ellas, es que una realiza un salto de instrucción cuando dos datos obtenidos de los registros son iguales (BEQ) y, la otra, cuando no son iguales (BNE). Por lo tanto ambas instrucciones se explican a la vez, tanto su comportamiento como las conexiones necesarias entre los elementos que las llevan a cabo.

Estas dos instrucciones necesitarán los siguientes elementos:

- Banco de registros
- Unidad Aritmético Lógica
- Extensión de signo
- Desplazamiento 2 bits a la izquierda
- Sumador

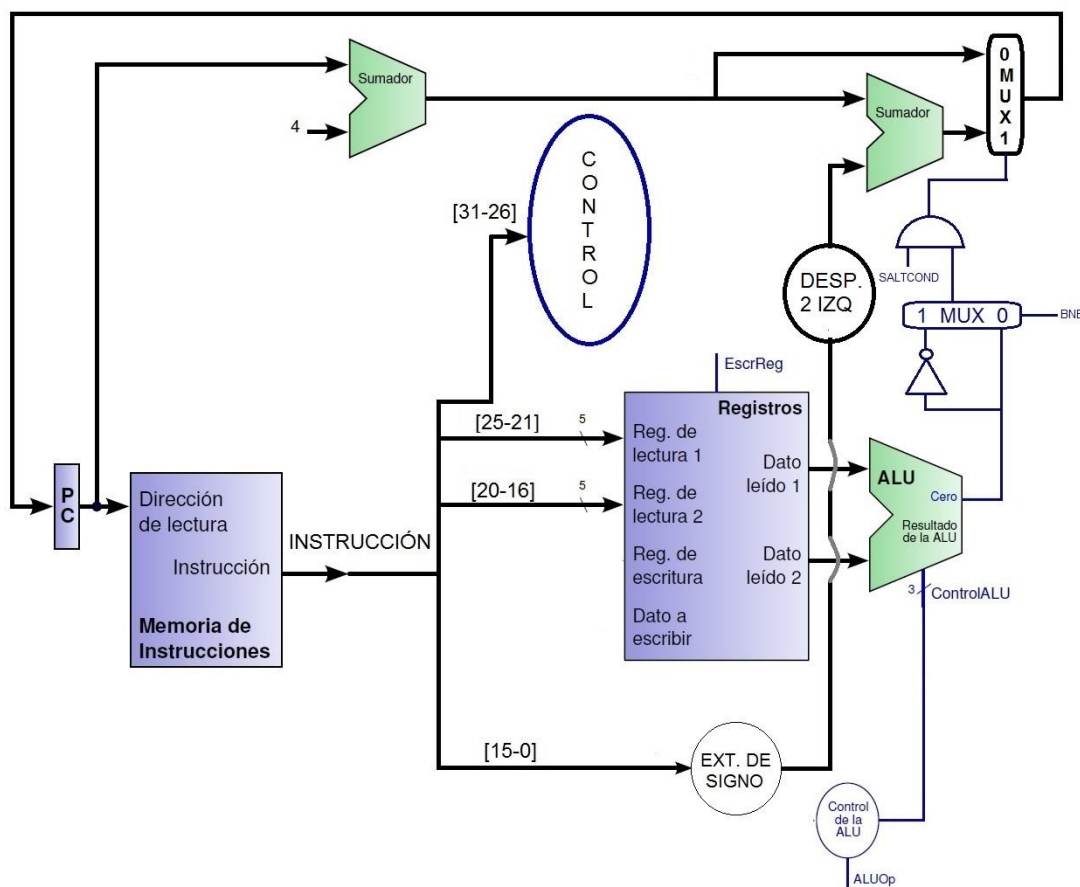


Figura 27: Conexiones de elementos para instrucciones BEQ y BNE



Como se puede comprobar, todos los elementos que se requieren son los mismos que en instrucciones anteriores, pero aquí entra en juego un sumador, que antes no se había llegado a usar. Este sumador simplemente realizará la suma de dos valores para determinar el valor del salto de la instrucción.

En la figura 27 se pueden ver todas las conexiones para estas dos instrucciones que se explican a continuación.

Primero de todo, y como ha sucedido siempre con todas las instrucciones, los dos primeros pasos consisten en que el contador de programa accede con una dirección a la memoria de instrucciones para que ésta genere la instrucción deseada.

A continuación, los 6 bits más significativos de la instrucción, es decir, los que hacen referencia al campo llamado OpCode, se introducen en el circuito de control del microprocesador para que éste habilite las diferentes señales que se requieren en cada elemento.

Los dos siguientes campos, cada uno de 5 bits (del 25 al 21 y del 20 al 16), pertenecen a dos direcciones de registros. Con estas direcciones, se obtiene dos datos de 32 bits que serán introducidos en la ALU para que ésta realice una comparación entre ellos.

Cuando la Unidad Aritmético Lógica haya realizado la comparación, sólo hay dos resultados posibles, que sean iguales o diferentes. Es aquí donde se diferencian las dos instrucciones. Si el resultado de la ALU muestra que ambos datos son iguales, por la salida llamada “Cero” se generará un uno lógico, y si son diferentes un cero lógico.

La salida denominada Resultado de la ALU, aunque no se utiliza en estas dos instrucciones, genera 32 bits del mismo valor, todos ceros si son iguales los datos de los dos registros que se comparan (realizando una resta entre ambos), para que la salida cero se active, y si ambos son distintos, se genera el valor numérico distinto de 0 en esta salida, para que la salida cero no se active.

Para que se produzca el salto de instrucción, debe estar habilitada previamente por el circuito de control la señal llamada “SaltoCondicional”, si no en ningún caso se podría producir el salto, ya que la señal de salida ““cero”” de la ALU se lleva a una puerta AND con esta otra señal.

Sin embargo, como se ha dicho, si el resultado es distinto por esa salida llamada Cero habría un cero lógico, y por lo tanto no se produciría el salto en el caso de la instrucción BNE. Para ello se coloca una bifurcación con un inversor, y luego un multiplexor para poder escoger que señal coger, si BEQ o BNE.

Sólo queda por definir el tamaño del salto a realizar. Viene dado por el campo llamado offset. Este campo de 16 bits se aumenta mediante la extensión de signo hasta 32 bits y se lleva a un sumador. El sumador sumará este dato más lo que sería la siguiente instrucción si no existiera este salto (PC+4+SALTO). El resultado de este sumador será el que se introduzca en el contador de programa, a diferencia de las instrucciones anteriores.

### 3.2.4. Elementos instrucciones tipo J

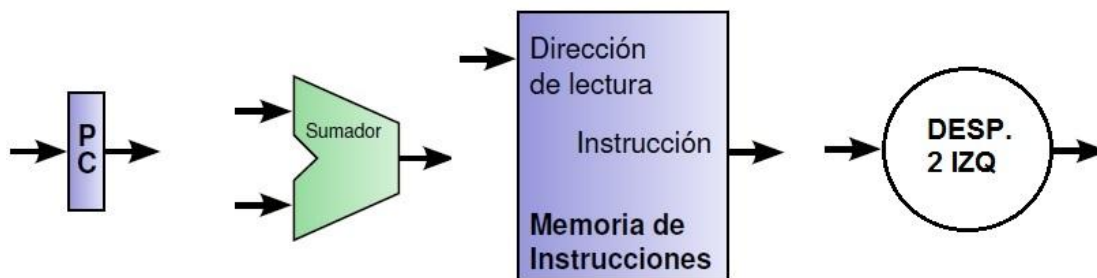


Figura 28: Registro contador, Sumador, Memoria de instrucciones y desplazamiento 2 izquierda

Para este tipo de instrucciones sólo se usan los elementos ya descritos, tanto el contador de programa como la memoria de instrucciones. La única diferencia es que se le añade un elemento que desplaza los bits dos posiciones a la izquierda. Éste no es exactamente igual que el “Desplazamiento 2 a la izquierda” que ya existía para instrucciones anteriores, porque en su entrada tendrá 26 bits, y a su salida habrá 28 bits. Los elementos que se emplean para este tipo de instrucciones se pueden ver en la figura 28.

Para el microprocesador desarrollado, sólo se implementa una instrucción tipo J, y es llamada también instrucción J. Esta instrucción lleva a cabo un salto de instrucción incondicional, es decir, es parecida a BEQ y BNE, pero con la diferencia que este salto se produce siempre que se invoque esta instrucción, con independencia de que los registros sean iguales o no, como ocurre en las instrucciones anteriores.

#### 3.2.1.6. Instrucción J

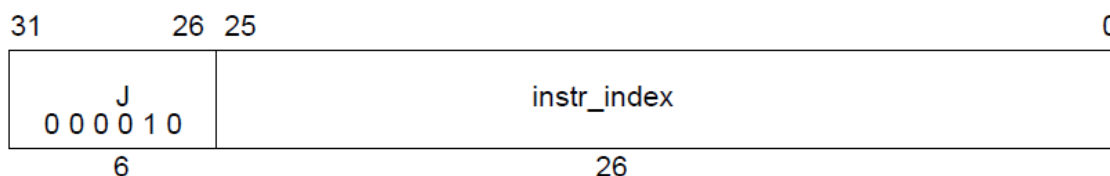


Figura 29: Formato instrucción J

Como se puede ver en la figura 29, el formato es casi completamente diferente a los demás tipos de instrucciones. La única similitud que existe es campo denominado OpCode, que consta como en todas las demás de los 6 bits más significativos.

El segundo campo es ya un campo mucho más extenso, de 26 bits, el cual determinará la dirección del salto de instrucción a realizar.

Como ya es costumbre en todas las instrucciones, los dos primeros pasos son idénticos a las demás instrucciones. El contador de programa accede con una dirección a la memoria de instrucciones, y ésta genera a su salida la instrucción que se lleva a cabo.

Las conexiones que se realizan para esta instrucción se muestran en la figura 30.

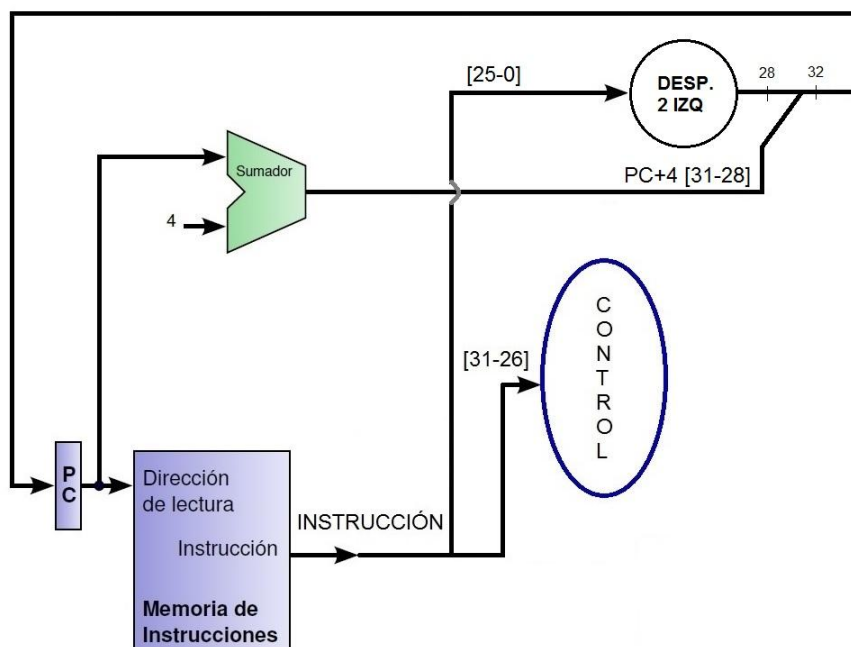


Figura 30: Conexiones de elementos para instrucción J

El primer campo (OpCode), se introduce directamente al circuito de control, para que genere las habilitaciones necesarias, en este caso, con la habilitación de “SaltoIncondicional” será suficiente, pues para llevar a cabo la instrucción no es necesario ni el banco de registros, ni la ALU, ni la memoria de datos.

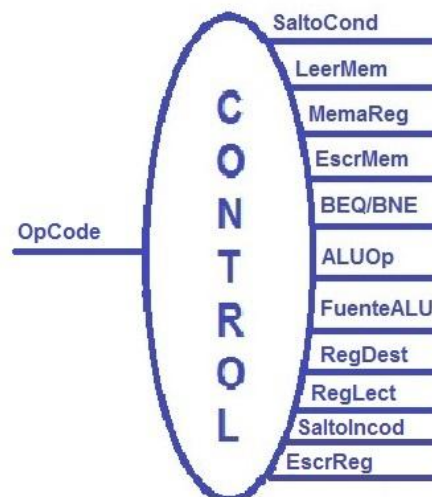
El segundo campo, el comprendido por los bits del 0 al 25, se conecta directamente a un desplazamiento de dos bits a la izquierda. Éste a su salida posee 28 bits, los cuales se componen de dos ceros en los bits menos significativos, y en los otros 26 bits, los que existen a su entrada. Por lo tanto, este desplazamiento, a diferencia del ya existente, no pierde ningún bit.

Una vez se tienen los 28 bits, se realiza una concatenación de bits, hasta llegar a 32. Los 4 bits se colocan en los bits más significativos, y son los provenientes de realizar el cálculo de la instrucción siguiente, es decir, son los 4 bits más significativos que existen a la salida del sumador del contador de programa.

Cuando ya se tienen los 32 bits, y ya por último, como la habilitación de salto incondicional está activada, estos 32 bits se llevan de nuevo al contador de programa para que luego se acceda a la siguiente dirección de memoria de instrucciones.

### **3.2.5. Circuito de control**

Todos los elementos descritos anteriormente, necesitan ser controlados para que su rendimiento sea el deseado, ya que no se realiza un circuito cada vez, sino que se crea un solo circuito, y con la inclusión de multiplexores, dependiendo qué instrucción se ejecute, se utilizarán unos u otros. Estos multiplexores, además de alguna señal de habilitación de otros elementos, se controlarán mediante el siguiente elemento.



*Figura 31: Circuito de control del microprocesador*

El circuito de control consta de una sola entrada, y varias salidas. La entrada consta de 6 bits, y a ella siempre se accede con el OpCode de cada instrucción. La serie de salidas son:

- SaltoCond. Habilidad para realizar las instrucciones BEQ y BNE. Si no se ejecutan ninguna de esas dos instrucciones, su valor será cero, y no se producirá el salto de instrucción.
- LeerMem. Habilidad que tiene la memoria de datos para poder obtener en su salida el dato deseado.

- MemaReg. Señal de control de un multiplexor para diferenciar cuando tomar la conexión de la salida de la ALU hacia el banco de registros o la conexión de la salida de la memoria hacia el banco de registros.
- EscrMem. Habilidad para poder escribir un dato obtenido en una dirección de la memoria de datos.
- Beq/Bne. Distinción para la instrucción de salto. Cuando su valor es 0 significa que se está ejecutando la instrucción BEQ, y cuando es 1, se estará ejecutando la instrucción BNE.
- ALUop. Entrada de dos bits que se conecta con el circuito de control de la ALU para generar las salidas correspondientes junto con el campo FUNC de cada instrucción.
- FuenteALU. Entrada de control a un multiplexor para diferenciar cuando tomar la salida del puerto de lectura dos del banco de registros a la ALU, o la conexión de la salida del extensor de signo a la ALU.
- RegDest. Entrada de control a otro multiplexor de elección de lectura del puerto de escritura del banco de registros. Se trata del mismo comportamiento que el anterior multiplexor, pero variando los bits del 20 al 16 y del 15 al 11.
- RegLect. Entrada de control al multiplexor de elección de lectura del primer puerto de lectura del banco de registros. A las dos entradas del multiplexor llegan dos direcciones de 5 bits, y dependiendo de si RegLect es igual a cero o a uno, a la salida llegan los bits de la instrucción del 25 al 21 o del 20 al 16.
- SaltoIncond. Entrada de control de un multiplexor que habilita la salida de salto cuando se ejecuta la instrucción J. Si se ejecuta otra, el valor es cero.
- EscrReg. Habilidad de escritura del banco de registros. Si esta señal no se encuentra activa (1 lógico), no se podrá realizar la escritura de un dato en una dirección del banco de registros.

### **3.3. Unión de todas las instrucciones: Hardware del microprocesador**

Una vez se tiene definido todo el conjunto de instrucciones, así como los elementos o circuitos que van a ser capaces de implementar dichas instrucciones, se debe diseñar las conexiones a realizar para que el microprocesador funcione conforme a lo deseado, es decir, que sea capaz de implementar todas las instrucciones con un sólo diseño.

Para realizar el diseño completo del microprocesador y unir los elementos que implementan las diferentes instrucciones, se debe ir paso a paso.

Primero de todo, se empieza con el diseño del MIPS que implementan las instrucciones tipo R. En éste apartado también se usan elementos de las figuras contenidas en el documento “Diseño del procesador MIPS R2000” [5].

Se comienza uniendo los elementos hardware de las instrucciones tipo R como ADD, SUB, AND, OR, SLT con SLL y SRL.

Como se puede ver en el apartado anterior, todas las instrucciones tipo R no tienen exactamente el mismo formato, por lo tanto, para realizar un solo hardware que contemple todas las instrucciones tipo R, se necesita modificar los dos diseños. Van a existir dos modificaciones en el diseño hardware.

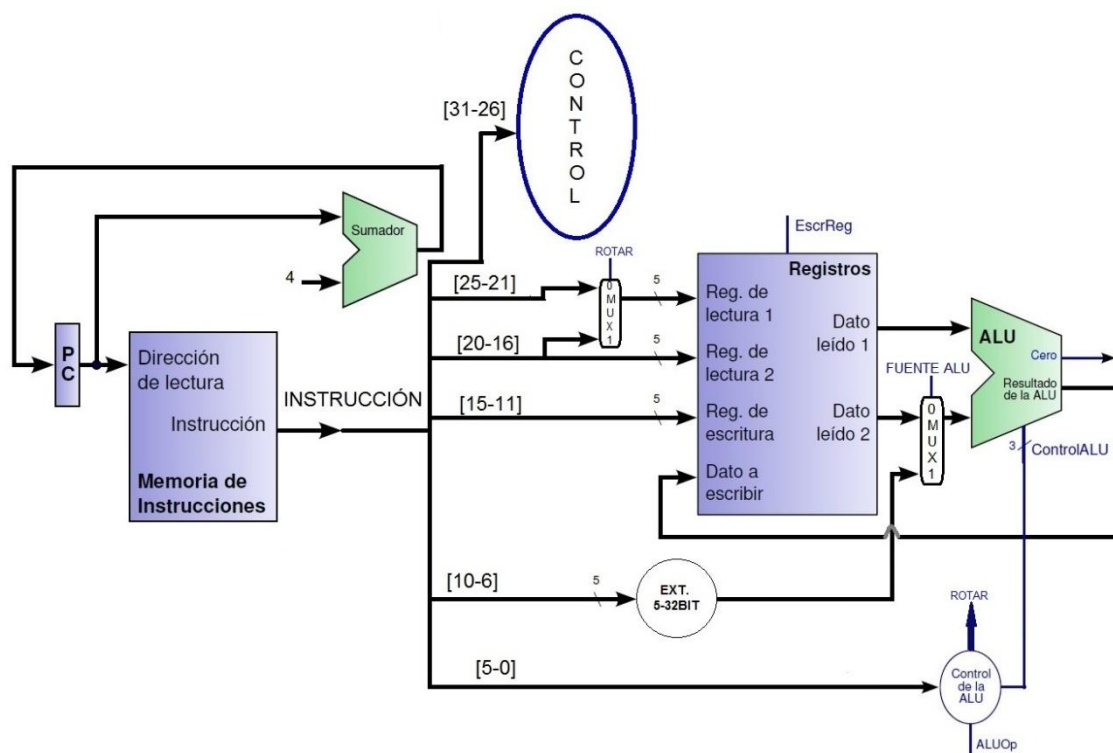


Figura 32: Hardware instrucciones tipo R

El primer cambio que se observa es en las entradas del banco de registros. En las operaciones de rotación, la entrada del banco de registros denominada como “Reg de lectura 1” tiene a su entrada el campo de la instrucción cuyos bits van del 16 al 20, mientras que para el resto es el campo cuyos bits pertenecen al intervalo 25-21. Introduciendo un multiplexor cuya entrada de control modifique que campo escoger de los dos para su salida se soluciona el problema.

La segunda modificación se realiza en una entrada de la ALU. Para las operaciones de rotación se introduce la salida procedente del extensor de 5 a 32 bits y, para el

resto, se introduce en la entrada la salida procedente de la salida 2 del banco de registros. La solución es idéntica a la anterior, es decir, colocando un multiplexor con una señal de control para que decida qué entrada escoger en su salida conforme se esté ejecutando una instrucción u otra.

Ambas modificaciones, así como el hardware completo de todas las instrucciones tipo R se puede ver en la figura 32.

El segundo paso para ir creando un solo hardware de diseño es ir añadiendo las instrucciones tipo I. Se va añadiendo de una en una para hacerlo más sencillo. La primera será la instrucción ADDI.

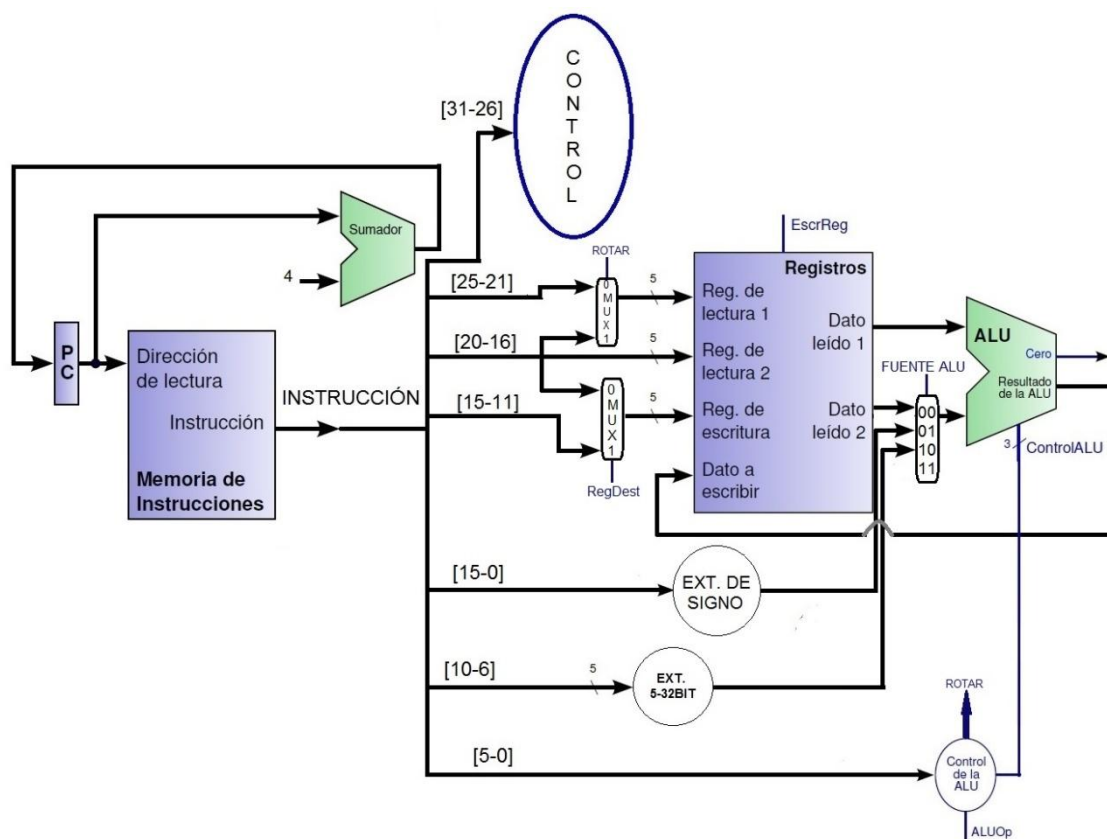


Figura 33: Hardware instrucciones tipo R más ADDI

Para el caso de juntar las instrucciones tipo R y la instrucción ADDI existen dos nuevas modificaciones con respecto al hardware de la figura 32.

La primera modificación está en la entrada del banco de registros denominado "Reg de escritura". Para el caso de las instrucciones tipo R se introduce el campo de la instrucción cuyos bits corresponden al intervalo del 15 al 11, mientras que para la instrucción tipo ADDI el campo que se introduce en esta entrada es el campo cuyos bits van desde el 20 al 16. Esta modificación se realiza como las

anteriores, introduciendo un multiplexor para que a su salida se encuentre un campo u otro. La entrada de control se denomina “RegDest”.

La segunda modificación se realiza en el multiplexor que se colocó a la entrada de la Unidad Aritmético Lógica. Ahora en vez de elegir entre dos salidas de otros elementos, se añade también la salida procedente del extensor de signo. Para conseguirlo hay que aumentar el multiplexor con dos bits de control, quedando una entrada a él libre, utilizándose únicamente las otras tres.

El hardware con las instrucciones tipo R más la instrucción ADDI que se ha explicado hasta ahora se puede ver en la figura 33.

A continuación se añaden otras dos instrucciones, las dos que tienen que ver con la memoria de datos, es decir, las instrucciones LW y SW. Para ello, hay que añadir algún elemento, como por ejemplo la memoria de datos, y modificar el hardware creado hasta el momento.

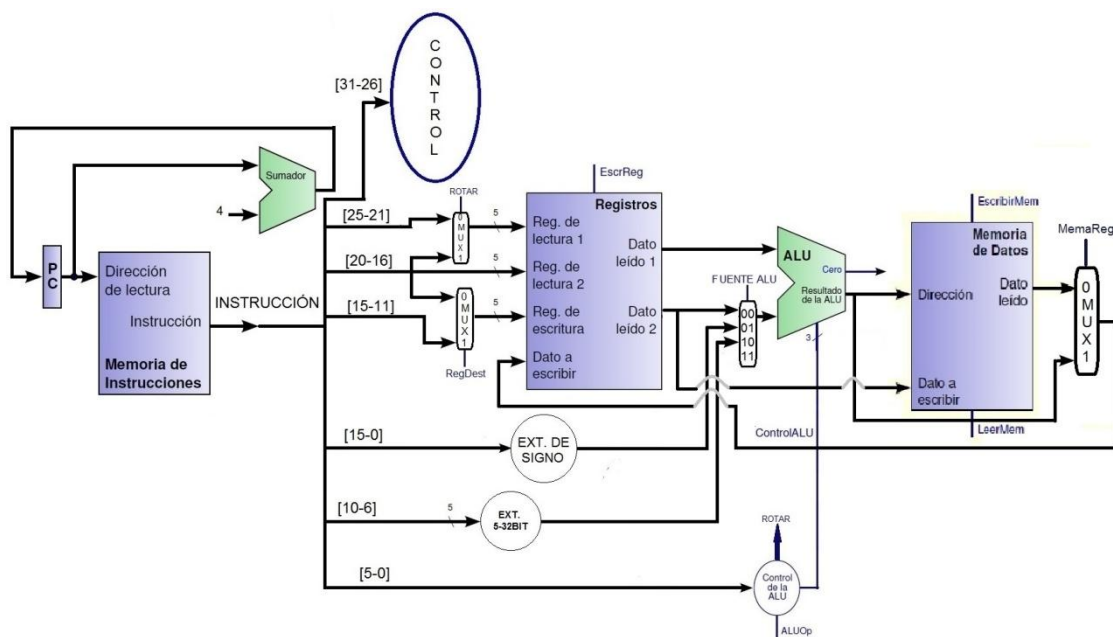


Figura 34: Hardware instrucciones tipo R, más ADDI, más LW, más SW

Al añadir las dos instrucciones que interactúan con la memoria de datos se deben añadir tres modificaciones.

La primera consiste simplemente en añadir el elemento de la memoria de datos, pero como sólo se utiliza para las instrucciones LW y SW hay que realizar la segunda y tercera modificación.

Para la operación LW, se debe introducir en la entrada del banco de registros denominado “Dato a escribir” la salida de la memoria de datos, cosa que no ocurre en las demás instrucciones, en las cuales el dato a escribir en el banco de registros



procede de la salida de la Unidad Aritmético Lógica (aunque no todas las instrucciones realizan escritura en el banco de registros). Se añade un multiplexor capaz de escoger entre la salida de la ALU y de la memoria de datos. La entrada de control se denomina MemaReg.

Para la operación SW, se debe escribir el dato obtenido en el banco de registros de la salida dos, en la memoria de datos, por lo tanto, se conecta la salida dos del banco de registros a la entrada denominada "Dato a escribir" de la memoria de datos. En este caso no hace falta añadir ningún multiplexor.

Las nuevas modificaciones introducidas en el hardware se muestran en la figura 34.

Para completar el hardware de las instrucciones tipo R y tipo I falta añadir las instrucciones de salto condicional. Estas dos instrucciones son las denominadas BEQ y BNE.

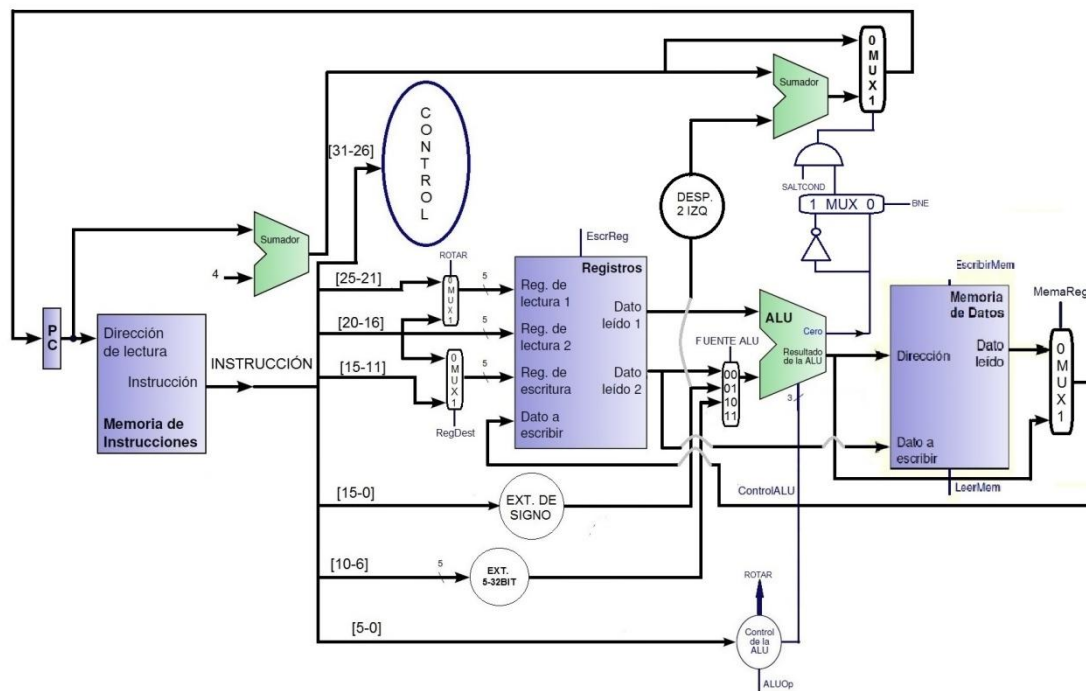


Figura 35: Hardware instrucciones tipo R más tipo I

Terminando de añadir las instrucciones tipo I con el hardware hasta ahora creado, se deben añadir una serie de modificaciones.

La primera es añadir dos elementos que son, el desplazador de bits a la izquierda y un sumador. La entrada del desplazador se conecta con la salida del extensor de signo, sin tener que añadir ningún multiplexor, y la salida del desplazador se conecta a una entrada del sumador. A la otra entrada del sumador se conecta la salida del sumador del contador de programa.



La segunda modificación es añadir a estos 28 bits los cuatro bits más significativos que se obtienen a la salida del sumador procedente del registro contador, y así, completar los 32 bits.

Por último, se añade un multiplexor para elegir entre dos salidas, dependiendo de si se ejecuta la instrucción de salto incondicional o las demás. La entrada de control se denomina “SaltoIncond”.

El hardware completo del microprocesador MIPS capaz de ejecutar todas las instrucciones se muestra en la figura 36. Si se añadieran más instrucciones, se realizarían más modificaciones a este hardware, siempre que sea necesario.

### **3.3. Creación software del microprocesador**

Una vez se tiene claro el diseño hardware del microprocesador, se debe crear el software que implementará dicho microprocesador.

Para ello, como ya se ha expuesto anteriormente, se empleará el programa Xilinx ISE. Se creará en varias etapas:

1. Creación de un elemento o circuito.
2. Simulación del elemento.
3. Prueba con la tarjeta Basys 2.
4. Vuelta al punto uno hasta que se crean todos los elementos que forman el microprocesador.
5. Unión de todos los elementos.
6. Decidir qué programas ejecutar en el microprocesador.
7. Probar si el microprocesador es capaz de desarrollar esos programas.
8. Si se quisiera, añadir tantas instrucciones, mejoras, aumentos de bits, etc. y volver a realizar todos los pasos.

#### **3.3.1. Lista de elementos hardware**

Como ya se tiene el hardware que formará el microprocesador definido, lo primero de todo se realiza la lista de los elementos a definir:

• Registro contador de programa	• Memoria de instrucciones
• Banco de registros	• Unidad Aritmético lógica
• Memoria de datos	• Circuito de control
• Sumadores	• Multiplexores
• Extensión de signo	• Desplazadores de posición

*Tabla 19: Elementos que forman el microprocesador*

Estos son los elementos que se implementan para llevar a cabo las instrucciones definidas con anterioridad.

### **3.3.2. Implementación de los elementos**

Teniendo claro qué elementos se deben implementar con el programa Xilinx ISE, se realiza primero el software de cada elemento y luego se le realizan dos tipos de pruebas:

1. Simulación mediante el programa ModelSim.
2. Prueba con la tarjeta Basys 2.

A continuación se muestra cada elemento diseñado para el microprocesador.

#### **3.3.2.1. *Registro contador de programa***

Para comenzar con la implementación de los elementos que formarán el microprocesador, se comienza a seguir el esquema que se muestra en la figura 36 de izquierda a derecha, y el primer elemento es el registro contador (PC).

La implementación en lenguaje VHDL se muestra en el anexo de implementación que hay al final de este informe.

Con el elemento creado en el programa Xilinx ISE, se procede a comprobar su correcto funcionamiento. Primero se comprueba mediante la simulación creada en el programa ModelSim y que se puede ver en la figura 37. Como se puede ver en esta figura, el registro actualiza su valor en cada flanco de subida del reloj que posee, es decir, conforme cambia el estado actual en cada ciclo, al siguiente ciclo, en la salida del registro se obtiene dicho estado.

Comprobado su comportamiento mediante la simulación, se procede a realizar la verificación del correcto funcionamiento con la tarjeta Basys2. Para ello, en vez de comprobar con 32 bits, se realiza un registro de menor tamaño, de 4 bits, para poder modificar las entradas con los interruptores que la tarjeta posee. El comportamiento es el esperado, y el registro funciona correctamente.

#### **3.3.2.2. Sumador Más 4**

Se trata de un circuito capaz de realizar la suma de un valor de 4 a la salida obtenida en el registro contador. El código que implementa este circuito se puede ver en el anexo de programación.

A continuación se realiza la simulación para comprobar su correcto funcionamiento (ver figura 38).

Comprobada que la simulación es correcta, se realiza una prueba con la tarjeta Basys 2 en la que se ve también que su funcionamiento es el deseado.

#### **3.3.2.3. Contador de programa**

La unión de ambos componentes (registro contador más sumador +4) descritos anteriormente, conforman lo que se puede denominar secuenciador, o contador de programa.

En cada ciclo de reloj aumenta cuatro unidades para acceder a la siguiente instrucción.

#### **3.3.2.4. Memoria de instrucciones**

El siguiente elemento que se implementa es la memoria de instrucciones, cuyo código también es mostrado en el anexo de programación.

Una vez implementada, se realiza la simulación oportuna para comprobar que el comportamiento es el esperado.

En la figura 39 se puede ver cómo introduciendo en la entrada de direcciones el número cero, por la salida de datos se obtiene el valor que hay alojado en esa dirección. Cuando se cambia el valor de la entrada de direcciones, cambia el valor obtenido en la salida después de que exista un flanco ascendente en la señal de reloj.



Con la comprobación de la simulación realizada con éxito, se procede a ver si con la tarjeta Basys2 el comportamiento es el mismo. Para ello se realiza una memoria igual, pero con menos direcciones y con los datos más pequeños, es decir menos bits. Se realiza con 4 bits de direcciones y con 4 bits de datos, y se puede llegar a la conclusión de que su comportamiento es el que se espera.

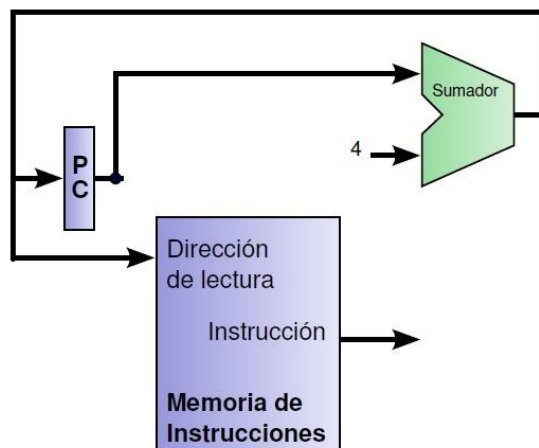


Figura 41: Unión final de la memoria de instrucciones al contador de programa

A parte, cuando estos tres componentes funcionan correctamente por separado, se realiza otra prueba con ellos unidos, para poder ver como se realiza la secuencia de instrucciones. Para ello, se realizan las conexiones oportunas y se transmite el fichero a la tarjeta Basys 2. Las instrucciones se van alternando de la cero a la cuatro, luego a la ocho, doce, dieciséis... Por lo tanto se puede concluir que hasta aquí el comportamiento del MIPS es idóneo.

Con la memoria de instrucciones hay que realizar una pequeña modificación con respecto al esquema del microprocesador completo, debido a que la tarjeta Basys2 tiene una memoria síncrona y posee su propio registro de entrada, por lo tanto, en vez de conectar la salida del registro contador a la entrada de direcciones de la memoria, se debe conectar la salida del sumador, como se puede ver en la figura 41.

### 3.3.2.5. Banco de registros

El siguiente elemento implementado es el banco de registros, y como ocurre con el resto de componentes, su código de implementación se puede ver en el anexo de programación.

Con el código implementado, se sigue el mismo procedimiento que para los demás elementos.

Primero se realiza la simulación que se puede ver en la figura 40. En esta figura no se introducen el total de los registros (32), sino que sólo se pueden observar 8 registros, para así hacer más visual la simulación. Se visualiza que el resultado es el esperado, ya que en las salidas se obtienen los datos que existen en las direcciones de los registros a los que se accede en las entradas de lectura, y estos registros se modifican habilitando la escritura y, con cada flanco ascendente de la señal de reloj, se cambia el dato de la dirección que se introduzca en la entrada “Reg. Escritura”. El dato que se guarda es el que se introduce por la entrada denominada “Dato a escribir”.

A continuación, se procede a la comprobación con la tarjeta Basys2. Igual que para el registro contador, la prueba con la tarjeta no se emplean tantos bits porque no existen suficientes entradas para modificar los datos que existen en el banco. Por lo tanto, se hace un banco de registros idéntico al que se utiliza en el microprocesador pero en vez de 32 registros, se implementa con cuatro registros de cuatro bits cada uno, para poder realizar los cambios con los interruptores y los pulsadores. El comportamiento del banco de registros es el esperado, por lo que se llega a la conclusión de que el elemento queda correctamente implementado.

#### **3.3.2.6. Extensión de signo**

Otro circuito implementado para componer el microprocesador, es el llamado Extensión de signo, cuyo código se muestra también en el anexo de programación. Como ya se ha explicado se trata de un elemento cuya función es aumentar un campo de bits de 16 a 32, respetando siempre el signo de este campo.

Primero, después de realizar la implementación del elemento, se procede con su correspondiente simulación (figura 42).

Como se puede ver en la figura 42, el elemento Extensión de signo aumenta el número de bits que existe en la entrada de 16 a 32. Teniendo en cuenta si el bit más significativo en la entrada es un cero, el número se completa con otros 16 ceros a la izquierda de los 16 primeros, y si el bit más significativo es un uno, se completa con otros 16 unos a la izquierda. Por lo tanto, viendo la simulación se puede llegar a la conclusión de que el resultado obtenido es el idóneo.

Después se realiza la prueba empleando la tarjeta Basys2, pero en este caso también hace falta que el número de bits a la entrada sea menor que 16, para poder cambiar la entrada y así poder realizar la prueba de manera más eficiente. Se realiza con una entrada de 4 bits y a la salida aparecen 8, pero el resto de la implementación es igual. El comportamiento es idéntico al observado en la simulación como cabe esperar. Este elemento queda implementado de forma correcta ya que su funcionamiento ha quedado completamente probado.



### **3.3.2.7. Extensión 5\_32 bit**

A continuación se implementa un componente muy parecido al anterior, pero con varias diferencias. En su entrada en vez de 16 bit, entran 5 bits. Estos bits a la salida se aumentan hasta un valor total de 32, pero los bits que se añaden a la izquierda son siempre ceros, es decir, se añaden un total de 27 ceros en la zona más significativa.

La implementación del circuito se muestra en el anexo de programación.

Se realiza, como en todos los elementos, la simulación para comprobar si el funcionamiento del circuito es el que se espera. La simulación se muestra en la figura 43.

Como se puede ver en esta figura, independientemente de si en el bit más significativo de la entrada existe un cero o un uno, a la salida existen 32 bits, los cuales son 27 ceros en el campo más significativo, y los 5 bits menos significativos son los que existen en la entrada.

La comprobación con la tarjeta Basys2 también se modifica, no en la entrada, que se pueden colocar los 5 bits, pero en la salida sólo se muestran 8 bits, uno por cada led que existe en la tarjeta. La prueba se realiza con éxito y el comportamiento del circuito es el esperado, así que éste queda perfectamente definido.

### **3.3.2.8. Desplazamiento 2 izquierda (operaciones tipo J)**

El siguiente circuito que se implementa, y que también se muestra en el anexo de programación, es el denominado desplazamiento dos izquierda. Este elemento sólo se emplea para operaciones tipo J.

La simulación del elemento denominado Desplazamiento dos izquierda (instrucciones tipo J) se muestra en la figura 44.

Como se puede ver en la figura 44, en la entrada existen 26 bits, y a la salida esos bits se desplazan dos posiciones a la izquierda, añadiendo dos ceros en los bits menos significativos. Los dos bits más significativos no se pierden ya que en la salida existen un total de 28 bits.

También se realiza la comprobación del circuito con la tarjeta Basys2. Para ello se implementa el circuito con alguna modificación. En la entrada se colocan un total de 6 bits, y a la salida 8, para poder ver el desplazamiento de las dos posiciones que se realiza. El comportamiento del circuito es el esperado, por lo que se llega a la conclusión de que el circuito está implementado de una forma correcta.



### **3.3.2.9. Control del microprocesador**

El circuito que controla el funcionamiento del microprocesador es el siguiente a implementar. Genera diferentes señales dependiendo de la instrucción que reciba. A su entrada recibe un total de 6 bits (campo OpCode de cada instrucción) y, genera un total de 10 salidas que controlan el comportamiento del microprocesador.

La implementación del circuito de control del microprocesador también se puede ver en el anexo de programación.

Se realiza la simulación para los OpCode de todas las instrucciones y el resultado se puede ver en las figuras que van desde la 45 a la 51.

Como se puede ver en todas estas figuras, el comportamiento que tiene el circuito, dependiendo del OpCode, corresponde con el resultado deseado, el cual se muestra en la tabla 20.

Para este circuito de control, sí se puede realizar la prueba con la tarjeta Basys2, igual que para la simulación, ya que sólo se tienen seis entradas y hay suficientes interruptores y salidas. Se llega a ver que el resultado es idóneo y que el circuito de control funciona perfectamente.

### **3.3.2.10. Desplazamiento 2 izquierda**

Se trata de un elemento igual que el del apartado 3.3.2.8 pero con alguna diferencia. En este caso este desplazamiento no se usa en instrucciones tipo J, sino en instrucciones tipo I. También se diferencia en el número de bits en la entrada y salida, en este caso, son 32 en ambas, pero el comportamiento es el mismo.

Como en todos los casos anteriores, la implementación de este circuito se puede ver en el anexo de programación.

La simulación que se realiza para este circuito es como la del otro desplazamiento, por lo tanto el comportamiento es el mismo, tal y como se puede ver en la figura 52.

La prueba que se realiza con la tarjeta Basys2 también es muy similar que para el otro circuito de desplazamiento, salvo que se emplean 8 bits de entrada y 8 bits de salida, pues el número de entradas y salidas deben ser iguales. El comportamiento es el esperado, ya que desplaza cada bit dos posiciones a la izquierda, y los dos bits más significativos desaparecen.

+ opcode	000101
regdest	1
saltocond	1
saltoincond	0
leermem	0
memareg	0
+ aluop	01
escrmem	0
fuatealu	0
bne	1
escreg	0

Figura 48: Simulación circuito de control OpCode 000101

+ opcode	100011
regdest	0
saltocond	0
saltoincond	0
leermem	1
memareg	1
+ aluop	00
escrmem	0
fuatealu	1
bne	1
escreg	1

Figura 49: Simulación circuito de control OpCode 100011

+ opcode	101011
regdest	0
saltocond	0
saltoincond	0
leermem	0
memareg	1
+ aluop	00
escrmem	1
fuatealu	1
bne	1
escreg	0

Figura 50: Simulación circuito de control OpCode 101011

+ opcode	111111
regdest	0
saltocond	0
saltoincond	0
leermem	0
memareg	0
+ aluop	00
escrmem	0
fuatealu	0
escreg	0

Figura 51: Simulación circuito de control OpCode 111111

OpCode	RegDest	Escreg	FuenteALU	ALU_OP	BNE	SaltoCond	SaltoIncond	LeerMem	EscreMem	MemaReg
000000	1	1	0	10	X	0	0	0	0	0
000010	1	0	0	10	X	0	1	0	0	0
000100	1	0	0	01	0	1	0	0	0	0
000101	1	0	0	01	1	1	0	0	0	0
100011	0	1	1	00	X	0	0	1	0	1
101011	0	0	1	00	X	0	0	0	1	1
111111	0	0	0	00	X	0	0	0	0	0

Tabla 20: Salidas circuito de control

### **3.3.2.11. Unidad Aritmético Lógica**

El siguiente elemento que se implementa es el encargado de realizar las operaciones aritméticas y/o lógicas. Este componente es la Unidad Aritmético Lógica o ALU. El código de implementación de este componente se puede ver también en el anexo de programación, como el código de todos los demás componentes.

Este componente realiza multitud de operaciones, por lo tanto se deben comprobar todas las operaciones dependiendo de la entrada de control que posee (ContALU), así que también se hacen diferentes pruebas. En cuanto a las simulaciones, se hace una diferente para cada operación. Estas simulaciones se pueden ver en las figuras que van desde la 53 a la 58.

La primera de las figuras, la número 53, trata de comprobar la operación de suma, y tal y como se puede ver, el resultado es bueno, ya que realiza la suma de las dos entradas de forma correcta y sale el resultado por su salida. Si este resultado es de valor cero, la salida denominada como “cero” se activa, si no esta salida siempre estará desactivada.

La segunda de las figuras, la número 54, es la simulación con la que se comprueba la operación de resta. Por la salida se muestra la resta de las dos entradas del componente, como se puede ver en dicha figura. La salida “cero” actúa como en la operación anterior, es decir, si el valor de la operación es cero, esta salida se activa, sino se encuentra desactivada.

La siguiente simulación, la que se muestra en la figura 55, comprueba la operación lógica AND. Esta operación se realiza bit a bit, como se puede ver en dicha simulación. El resultado de la operación entre los bits de cada dato que existe a la entrada se muestra en la salida llamada RestALU. Si este resultado es de valor cero, también la salida llamada “cero” se activa, mientras que si no es de valor cero, la salida no estará activa.

La siguiente simulación es la operación lógica OR. Esta simulación se muestra en la figura 56, y como se puede ver, el resultado que existe en la salida RestALU es el deseado, ya que se realiza la operación de suma lógica bit a bit. Como en todos los casos para este componente, si el resultado es cero, por la otra salida del componente se activa dicha señal.

Otra de las operaciones que existe en este componente, es la operación de desplazamiento, tanto a izquierda como a derecha (SLL o SRL). Esta operación es idéntica a la que se realiza en el componente de Desplazamiento 2 izquierda, aunque se puede variar el número de desplazamiento, desde 0 a 31, y se puede cambiar de sentido, bien sea a izquierda (SLL) o a derecha (SRL).



+ 	entdesp	110001001000000111110010101111	0101010010001001101110010101010
+ 	saldesp	000100100000001111100101011100	010100100001001101110010101000

Figura 52: Simulación Desplazamiento dos izquierda






+ 	datoone	1001001110010001010000111010011	11111111111111111111111111111111
+ 	datotwo	0010111000110000010111110111000	00000000000000000000000000000001
+ 	contalu	010	
+ 	restalu	1100000111000001101000110001011	00000000000000000000000000000000
	zero		

Figura 53; Simulación ALU operación suma






+ 	datoone	1010000011110010100001001101101	
+ 	datotwo	1000010100000111101110111011111	10100000111100101000010011011101
+ 	contalu	110	
+ 	restalu	00011011111010101100100100011110	00000000000000000000000000000000
	zero		

Figura 54: Simulación ALU operación resta






+ 	datoone	10001101011111000000001101000111	
+ 	datotwo	0101110001011111110001000000001	00000000000000000000000000000000
+ 	contalu	000	
+ 	restalu	00001100010111000000001000000001	00000000000000000000000000000000
	zero		

Figura 55: Simulación ALU operación AND

Como se puede observar en la figura 57, el resultado de estas operaciones de rotación o desplazamiento es el deseado, ya que se añaden tantos ceros, a derecha o izquierda del valor introducido por la primera entrada de la ALU, como el número binario que se introduce por la segunda entrada de la ALU.

Y la última operación que es capaz de realizar este componente, es una operación de comparación. El resultado que se obtiene es de valor cero si el dato leído en su primera entrada es mayor o igual que el segundo, o de valor uno si el resultado del primer elemento es menor que el segundo. La simulación hecha se puede ver en la figura 58. Se puede concluir que esta operación también se realiza de forma satisfactoria.

Con todas las operaciones simuladas, se procede a las pruebas con la tarjeta Basys 2, con algunos cambios como para otros elementos, debido al número de entradas y salidas que ésta posee. También las pruebas realizadas se cumplen de forma satisfactoria, por lo tanto se llega a la conclusión de que el elemento implementado en esta ocasión queda perfectamente definido.

#### **3.3.2.12. Circuito de Control de la ALU**

Para que el componente anterior sea capaz de diferenciar cuándo realizar una operación u otra, tiene una señal de control, la cual cambia de valor dependiendo de este circuito de control. Este circuito a su vez depende de dos entradas: la generada por el otro circuito de control llamada ALUOP, y por el último campo de la instrucción (Function) para las instrucciones tipo R.

También genera dos salidas (Rotar y EntALU) para controlar dos multiplexores y así ajustar el funcionamiento del microprocesador.

Para comprobar su correcto funcionamiento, se hace la simulación del circuito. El resultado de esta simulación se muestra en la figura 59.

El resultado de la simulación del circuito que controla la Unidad Aritmético Lógica es el esperado, ya que se corresponde con la tabla de verdad (tabla 21) con la que queda definido este circuito de control.

Como este resultado es bueno, se procede a la prueba del circuito con la tarjeta Basys2. Se prueba el circuito sin realizar ninguna modificación en él ya que, a diferencia de otros elementos, en este caso si se pueden implementar todas las entradas (8) y salidas (5). El resultado obtenido de esta prueba también es el deseado, por lo tanto se concluye que el circuito queda perfectamente implementado.

+	datoone	01110011110010010000110101101000
+	datotwo	10100010011110110001111100011111
+	contalu	001
+	restalu	11110011111110110001111101111111
	zero	

Figura 56: Simulación ALU operación OR

+	datoone	11000101010111011000111000110111					
+	datotwo	00000000000000000000000000000001					
+	contalu	100			101		
+	restalu	000000000000011000101010111011000			11011000111000110111000000000000		
	zero						

Figura 57: Simulación ALU operación SLL y SRL

+	datoone	11101000100001110101011111111000					
+	datotwo	01101111010111101000000110111101			11111111010111101000000110111101		
+	contalu	111					
+	restalu	00000000000000000000000000000000			00000000000000000000000000000001		
	zero						

Figura 58: Simulación ALU operación SLT

+	camfun	UUUUUU			100000		100010		100100		100101		000000		000010		101010	
+	aluop	00		01		10												
+	contalu	010		110		010		110		000		001		101		100		111
	ext_5bit32																	
	rotar																	

Figura 59: Simulación circuito de Control ALU



Código operación	ALUOP	Campo Función	Operación	Acción deseada	ControlALU	Rotar	Ent_ALU
LW	00	XXXXXX	Cargar palabra	Suma	010	0	0
SW	00	XXXXXX	Almacenar palabra	Suma	010	0	0
BEQ/BNE	01	XXXXXX	Saltar si/no es igual	Resta	110	0	0
Tipo R	10	100000	Suma	Suma	010	0	0
Tipo R	10	100010	Resta	Resta	110	0	0
Tipo R	10	100100	And	And	000	0	0
Tipo R	10	100101	Or	Or	001	0	0
Tipo R	10	000000	Rotar izquierda	SLL	101	1	1
Tipo R	10	000010	Rotar derecha	SRL	100	1	1
Tipo R	10	101010	Poner si menor	SLT	111	0	0

*Tabla 21: Tabla de verdad circuito de control ALU*

### 3.3.2.13. Sumador

Otro elemento que se implementa para el microprocesador es un sumador. Es un componente que realiza la suma de dos datos obtenidos a sus entradas. El código que implementa este elemento también es mostrado en el anexo de programación.

Primero se realiza la simulación de este circuito. Esta simulación se muestra en la figura 60 y, como se puede ver, el resultado obtenido en su salida es la suma aritmética de las dos entradas, por lo tanto el comportamiento es el esperado.

Después se lleva a cabo la prueba con la tarjeta Basys 2, pero en vez de sumar datos de 32 bits, se hace la prueba con datos de 4 bits, para poder llegar a visualizar su correcto funcionamiento, como así ha sido. Por lo tanto este circuito también queda perfectamente implementado.

### 3.3.2.14. Memoria de Datos

El elemento de Memoria de Datos es un componente que se implementa como una memoria síncrona de escritura y lectura (RAM). Su código se muestra en el anexo de programación.

Para comprobar su funcionamiento, primero se lleva a cabo su simulación. Unos ejemplos de los resultados obtenidos se pueden ver en las figuras 61, 62 y 63.

+	entsum1	10101000011110100111100000011110	001010100110101001011010010110
+	entsum2	0111010000001110111110110111100	01110101001010001100110110110100
+	salsum	00011100100010010111010111011010	10011111100100110010100000001010

Figura 60: Simulación Sumador



Figura 61: Simulación Memoria de Datos (escritura)

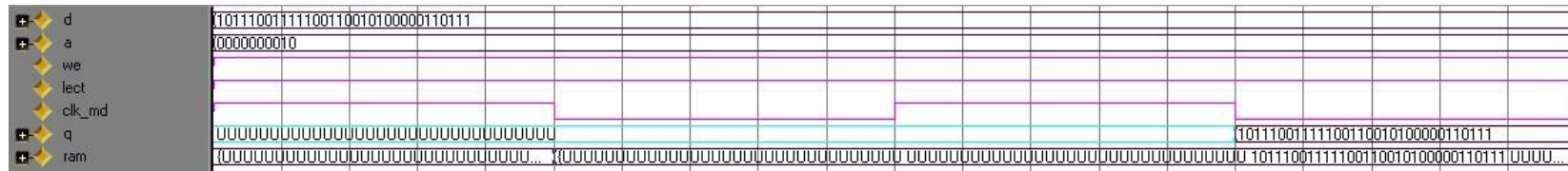


Figura 62: Simulación Memoria de Datos (lectura)

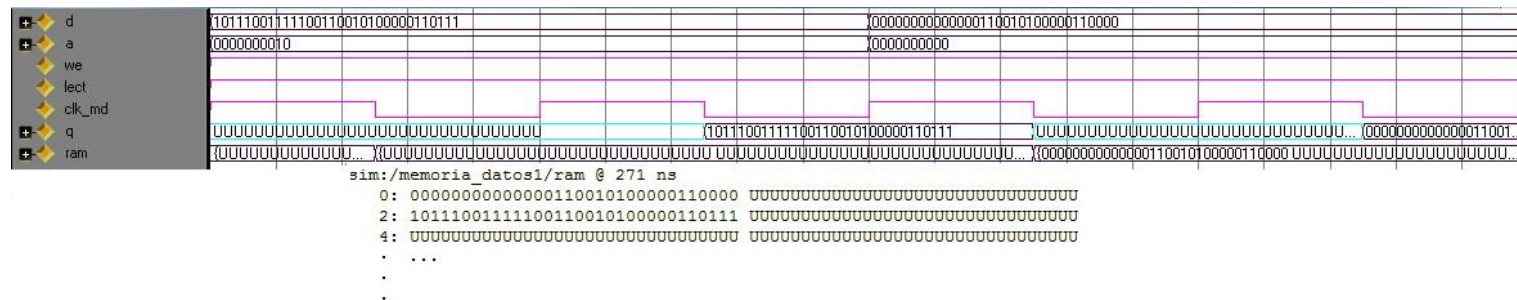


Figura 63: Simulación Memoria de Datos

En la figura 61 se puede ver cómo se realiza la escritura de un dato en una dirección de la memoria. Se observa cómo en la dirección dos de la memoria se escribe el dato que existe a la entrada cuando existe un flanco de bajada en la señal de reloj, siempre que la habilitación de escritura esté activa.

Para realizar la lectura de ese dato, se realiza la simulación de la figura 62. En ella se muestra por la salida el valor leído en la dirección de memoria que existe a la entrada cuando existe un flanco de bajada en la señal de reloj, y a la vez, está habilitada la entrada de control de lectura.

Uniendo ambas operaciones, lectura y escritura, es la simulación mostrada en la figura 63. Se observa, en cada ciclo, cómo primero se escribe el dato en una dirección, y al ciclo siguiente se realiza la lectura de esa dirección. Tanto escritura como lectura se hacen cuando existe un flanco de bajada de la señal de reloj y estén habilitadas las señales de control de escritura o lectura.

A la vista de los resultados obtenidos en estas simulaciones, se puede decir que el comportamiento por el momento de la memoria es el deseado, así que se procede a realizar la prueba de esta memoria con la tarjeta Basys 2.

Para esta segunda prueba, se modifica el número de direcciones que tendrá la memoria a cuatro y, también, se cambia el número de bits que existe en la entrada de datos a cuatro. Estos cambios son hechos para poder visualizar así el comportamiento de la memoria ya que, el número de entradas y salidas es más limitado que 32 bits y 10 direcciones. El resultado obtenido en esta segunda prueba es el esperado, por lo tanto, se llega a la conclusión de que a memoria de datos queda perfectamente implementada.

#### **3.3.2.15. Unión 28+4 bits**

Este no es un elemento hardware como los anteriores. Es un elemento que se implementa con el objetivo de juntar en un dato de 32 bits, dos datos, uno de 28 bits procedente de la salida del desplazamiento 2 izquierda para instrucciones tipo J y, otro de 4 bits procedente de los 4 bits más significativos de la instrucción siguiente (PC+4).

El código de implementación se muestra en el anexo de programación. Para comprobar su funcionamiento se le realiza la simulación que se puede ver en la figura 65.

En esta figura se puede ver cómo el comportamiento de este circuito es el deseado ya que realiza la acción que se le pide de juntar dos datos en uno.

+	ent28	1011100000100000000011000000	0000000000100000000011000000
+	ent4	0100	1111
+	sal32	01001011100000100000000011000000	11110000000000100000000011000000

Figura 64: Simulación Unión 28+4 bits

La prueba con la tarjeta Basys 2 se realiza con la unión de 6+2 bits, por el hecho de siempre, menos entradas y salidas para comprobar el funcionamiento. El resultado es el esperado, así que el componente queda definido de manera eficiente.

### 3.3.2.16. Multiplexores

Por último, para combinar todas las instrucciones, se implementan una serie de multiplexores, que controlan qué camino seguir dependiendo que instrucción se ejecute, es decir, son capaces de escoger qué señal existe a su salida de entre diferentes señales que existen a su entrada.

Se implementan un total de 8 multiplexores, aunque no todos son iguales. Su comportamiento sí que es el mismo, pero la diferencia que hay de unos a otros son el número de bits que hay en sus señales de entrada y salida. A continuación se indica dónde se colocan los multiplexores y el número de bits que contienen sus señales:

- Multiplexor entrada de la dirección de lectura 1 al banco de registros (5 bits).
- Multiplexor entrada de la dirección de escritura al banco de registros (5 bits).
- Multiplexor salida del dato leído 2 del banco de registros y de la salida del elemento extensión de signo (32 bits).
- Multiplexor entrada 2 de la ALU (32 bits).
- Multiplexor para diferenciar instrucciones tipo BEQ de BNE (1 bit).
- Multiplexor para escoger instrucción siguiente o salto condicional (32 bits).
- Multiplexor para escoger instrucción siguiente o salto condicional, o salto incondicional (32 bits).
- Multiplexor salida ALU o salida memoria de datos (32 bits).

Para comprobar su correcto funcionamiento se simula un multiplexor de dos entradas de datos de un bit, con una entrada de control de un bit y una salida, también de un bit (ver figura 65).

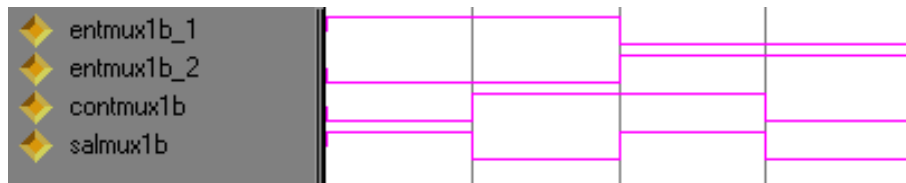


Figura 65: Simulación multiplexor de 1 bit

Como se puede ver, el comportamiento del multiplexor es el deseado, ya que dependiendo de si en la entrada de control existe un cero o un uno, a la salida existe el valor del dato uno o del dato dos respectivamente, tal y como se observa en la figura 65.

A continuación el programa implementado se introduce en la Basys 2 y se realiza la prueba de su funcionamiento. El resultado obtenido es idéntico al de la simulación, por lo que se puede concluir que el comportamiento de los multiplexores que se implementan es el deseado.

### 3.3.3. Microprocesador completo. Unión de todos los elementos.

Para implementar el microprocesador MIPS completo no es suficiente con implementar cada elemento por separado. Hay que realizar una serie de conexiones entre los distintos elementos anteriores. El esquema que se sigue en las conexiones es el esquema del microprocesador mostrado en apartados anteriores, pero para recordar se muestra en la figura 66.

Para unir los componentes creados y crear uno sólo y así crear el microprocesador se hace de la manera que se explica a continuación.

El primer paso ya está hecho, pues es crear cada elemento por separado, cada uno en un archivo o módulo VHDL. Todos estos archivos se crean dentro del mismo proyecto.

A continuación, dentro del mismo proyecto, se debe crear un módulo o archivo vhd nuevo. Este módulo debe ser creado en máximo nivel de jerarquía y, los otros elementos, en el nivel inferior. Esto es así debido a que en el nivel superior se realizan las llamadas a los elementos creados en el nivel inferior, es decir, en este nuevo módulo creado se utilizan los elementos creados anteriormente.

Merece la pena recalcar que, para que el funcionamiento del microprocesador sea bueno, el orden de compilación debe ser: primero los ficheros o archivos del nivel inferior, y luego compilar el fichero de mayor nivel de jerarquía. Si el orden fuera a la inversa, el comportamiento del microprocesador no sería el esperado, o se darían fallos y su funcionamiento no sería adecuado, pudiendo incluso no funcionar.

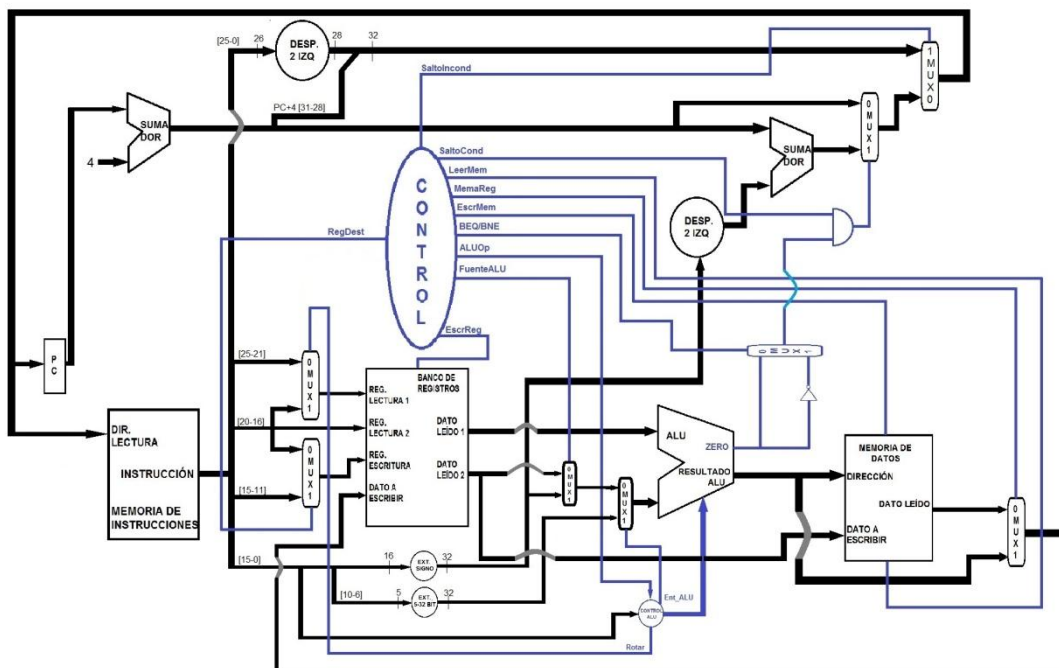


Figura 66: Microprocesador MIPS

Como ya se ha dicho, los elementos del nivel inferior ya se han creado, y como su comportamiento se ha comprobado que es bueno, implica que su compilación ha sido satisfactoria, por lo tanto se está en disposición de crear el archivo o fichero de mayor jerarquía, en el que se realizarán las llamadas a los componentes y se conectarán unos con otros de forma que quede perfectamente definido el microprocesador MIPS que se pretende diseñar.

Este fichero de mayor jerarquía, al cual se le puede llamar fichero "MIPS", tiene una serie de partes bien diferenciadas:

- Librerías utilizadas.
- Entidad del MIPS.
- Arquitectura del MIPS.
  - Declaración de componentes.
  - Declaración de señales la cuales se emplean para la conexión entre los diferentes elementos.
  - Declaración del lugar en que se encuentra la arquitectura de cada componente.
  - Conexión de los componentes entre sí.

Se puede llegar fácilmente a la conclusión de que la estructura más primaria (librerías, entidad y arquitectura) es igual que la estructura empleada para los ficheros de cada componente. Sin embargo, dentro de la arquitectura del microprocesador MIPS varía con respecto a los otros ficheros. En este fichero “MIPS” la estructura es completamente diferente a la estructura de los componentes del nivel inferior.

La primera parte de este fichero es donde se debe declarar todos los componentes creados que componen el microprocesador, así como las entradas y salidas que cada componente posee para que luego puedan ser empleados y conectados unos con otros.

A continuación se deben crear las señales que serán útiles para conectar unos componentes con otros, siguiendo siempre el esquema del microprocesador descrito, el cual se puede ver en la figura 66. Estas señales representan, a modo metafórico, los cables que conectan cada circuito.

Para que los elementos puedan ser empleados, se realizan las llamadas a estos. Para ello se debe especificar en qué lugar, carpeta o proyecto se encuentra la arquitectura de los elementos que se utilizan para completar o crear el microprocesador. Se debe realizar para todos los componentes de la misma manera.

El último paso para crear el microprocesador es definir las conexiones existentes entre cada elemento que compone el microprocesador siguiendo el esquema que se muestra en la figura 66.

Con este fichero (entidad + arquitectura) queda completo el desarrollo del microprocesador MIPS. Si se quiere ver el resultado de alguna salida de algún componente, basta con añadir alguna salida exterior para comprobar su funcionamiento, cosa que se realizará para comprobar el funcionamiento del microprocesador y analizar el resultado obtenido.

La implementación completa del microprocesador MIPS, tanto el fichero de mayor nivel explicado ahora, como los ficheros de cada elemento del nivel inferior, se pueden ver en el anexo de programación. Se puede observar cómo el fichero sigue la estructura descrita anteriormente.

#### **3.3.4. Programas que se ejecutarán en el microprocesador**

Para comprobar si el microprocesador diseñado está bien implementado y su funcionamiento es bueno, se debe probar si es capaz de ejecutar diferentes programas. El conjunto de todos los programas deberán tener todo el conjunto de

instrucciones, para ver que el microprocesador cumple perfectamente con cualquier instrucción descrita en apartados anteriores.

Se implementarán un total de ejemplos de programas de cuatro, suficientes para ver que el microprocesador actúa conforme se desea y realiza todas las instrucciones que existen en el programa. A continuación se explica el desarrollo de los cinco programas.

### 3.3.4.1. Programa 1

El primer programa que se implementa es un programa que se utiliza como ejercicio en la asignatura Electrónica Digital. Este programa consta de un total de 18 instrucciones:

1. <b>ADDI R<sub>0</sub>, R<sub>7</sub>, 8</b>	2. <b>ADDI R<sub>1</sub>, R<sub>7</sub>, -32768</b>
3. <b>ADDI R<sub>2</sub>, R<sub>7</sub>, 32</b>	4. <b>SLT R<sub>4</sub>, R<sub>0</sub>, R<sub>1</sub></b>
5. <b>BEQ R<sub>4</sub>, R<sub>7</sub>, 3</b>	6. <b>ADD R<sub>5</sub>, R<sub>0</sub>, R<sub>7</sub></b>
7. <b>ADD R<sub>0</sub>, R<sub>1</sub>, R<sub>7</sub></b>	8. <b>ADD R<sub>1</sub>, R<sub>5</sub>, R<sub>7</sub></b>
9. <b>SLT R<sub>4</sub>, R<sub>0</sub>, R<sub>2</sub></b>	10. <b>BEQ R<sub>4</sub>, R<sub>7</sub>, 3</b>
11. <b>ADD R<sub>5</sub>, R<sub>0</sub>, R<sub>7</sub></b>	12. <b>ADD R<sub>0</sub>, R<sub>2</sub>, R<sub>7</sub></b>
13. <b>ADD R<sub>2</sub>, R<sub>5</sub>, R<sub>7</sub></b>	14. <b>SLT R<sub>4</sub>, R<sub>1</sub>, R<sub>2</sub></b>
15. <b>BEQ R<sub>4</sub>, R<sub>7</sub>, 3</b>	16. <b>ADD R<sub>5</sub>, R<sub>1</sub>, R<sub>7</sub></b>
17. <b>ADD R<sub>1</sub>, R<sub>2</sub>, R<sub>7</sub></b>	18. <b>ADD R<sub>2</sub>, R<sub>5</sub>, R<sub>7</sub></b>

*Tabla 22: Instrucciones programa 1*

Para llevar a cabo estas instrucciones, se deben introducir en la memoria de instrucciones en código hexadecimal. Pero cómo los elementos del microprocesador utilizan código binario, se desglosa cada instrucción presente en la tabla 22 en código hexadecimal y código binario, tal y como se puede ver en la tabla 23.

Como se puede ver, se implementan instrucciones tipo R y tipo I, pero no tipo J. Dentro de las tipo R, se encuentran las instrucciones “ADD y SLT”, y las tipo I son “ADDI y BEQ”, por lo que tiene cuatro instrucciones diferentes, cambiando el contenido de los registros, pero no de la memoria de datos.

El resultado obtenido al probar este programa se mostrará en el apartado denominado Análisis de Resultados, en el cuál también se realiza una discusión de los resultados obtenidos al realizar dicha prueba.



Instrucción		Código Hexadecimal	Código Binario
1.	ADDI R0, R7, 8	20E00008	00100000111000000000000000001000
2.	ADDI R1, R7, 768	20E10300	00100000111000010000001100000000
3.	ADDI R2, R7, 32	20E20020	00100000111000100000000000010000
4.	SLT R4, R0, R1	0001202A	000000000000000010010000000101010
5.	BEQ R4, R7, 3	10870003	00010000100001110000000000000011
6.	ADD R5, R0, R7	00072820	00000000000000111001010000010000
7.	ADD R0, R1, R7	00270020	00000000001001110000000000100000
8.	ADD R1, R5, R7	00A70820	00000000101001110000100000100000
9.	SLT R4, R0, R2	0002202A	0000000000000000100010000000101010
10.	BEQ R4, R7, 3	10870003	00010000100001110000000000000011
11.	ADD R5, R0, R7	00072820	00000000000000111001010000010000
12.	ADD R0, R2, R7	00470020	00000000010001110000000000100000
13.	ADD R2, R5, R7	00A71020	00000000101001110001000000100000
14.	SLT R4, R1, R2	0022202A	00000000001000100010000000101010
15.	BEQ R4, R7, 3	10870003	00010000100001110000000000000011
16.	ADD R5, R1, R7	00272820	00000000001001110010100000100000
17.	ADD R1, R2, R7	00470820	00000000010001110000100000100000
18.	ADD R2, R5, R7	00A71020	00000000101001110001000000100000

Tabla 23: Desglose instrucciones programa 1

### 3.3.4.2. Programa 2

El segundo programa que se implementa sirve para comprobar el correcto funcionamiento de cada uno de los registros, además de probar todas las instrucciones tipo R.

El programa consta de un total de 32 instrucciones tal y como se puede ver en la tabla 24.

1. ADDI R0, R0, -25	2. ADDI R1, R0, 2
3. ADD R2, R0, R1	4. ADD R3, R0, R1
5. AND R4, R2, R3	6. OR R5, R4, R0
7. SUB R6, R4, R5	8. SLT R7, R7, R6
9. SLT R8, R7, R7	10. OR R9, R8, R5
11. AND R10, R9, R9	12. SLL R11, R10, 6
13. SRL R12, R11, 6	14. OR R13, R12, R0
15. SRL R14, R13, 23	16. SLL R15, R13, 26
17. SUB R16, R15, R14	18. SLT R17, R17, R16
19. ADD R18, R17, R16	20. SUB R19, R18, R17
21. ADD R20, R19, R18	22. AND R21, R20, R19
23. SLL R22, R21, 1	24. OR R23, R22, R21
25. SRL R24, R23, 3	26. ADD R25, R23, R24
27. SUB R26, R25, R23	28. ADD R27, R26, R25
29. OR R28, R27, R26	30. AND R29, R28, R28
31. ADD R30, R30, R29	32. SUB R31, R31, R31

Tabla 24: Instrucciones programa 2

Como se puede ver en la tabla 24, se realizan escrituras en todos los registros, y se utilizan para ello todas las instrucciones tipo R, y para comenzar con valores distintos de cero, la instrucción ADDI.

Para llevar a cabo estas instrucciones, igual que en el programa 1, se deben introducir en la memoria de instrucciones en código hexadecimal. Se desglosa cada instrucción presente en la tabla 24 en código hexadecimal y código binario, tal y como se puede ver en la tabla 25.

El resultado y análisis de este programa se lleva a cabo en el apartado denominado como Análisis de resultados.

### 3.3.4.3. Programa 3

El tercer programa es otro ejercicio perteneciente a la asignatura denominada Electrónica Digital. Se trata de una multiplicación de dos números:  $3 \times 5 = 15$ .

INSTRUCCIÓN	CÓDIGO BINARIO	CÓDIGO HEXADECIMAL
1. ADDI R0, R0, -25	0010000000000000111111111100111	200FFE7
2. ADDI R1, R0, 2	0010000000000000100000000000010	20010002
3. ADD R2, R0, R1	000000000000000010001000000100000	00011020
4. ADD R3, R0, R1	000000000000000010001100000100000	00011820
5. AND R4, R2, R3	0000000010000110010000000100100	00432024
6. OR R5, R4, R0	00000000100000000010100000100101	00802825
7. SUB R6, R4, R5	00000000100001010011000000100010	00853022
8. SLT R7, R7, R6	00000000111001100011100000101010	00E6382A
9. SLT R8, R7, R7	00000000111001110100000000101010	00E7402A
10. OR R9, R8, R5	00000001000001010100100000100101	01054825
11. AND R10, R9, R9	00000001001010010101000000100100	01295024
12. SLL R11, R10, 6	00000000000010100101100110000000	000A5980
13. SRL R12, R11, 6	00000000000010110110000110000010	000B6182
14. OR R13, R12, R0	00000001100000000110100000100101	01806825
15. SRL R14, R13, 23	00000000000011010111010111000010	000D75C2
16. SLL R15, R13, 26	00000000000011010111111101000000	000D7E80
17. SUB R16, R15, R14	0000000111101110100000000100010	01EE8022
18. SLT R17, R17, R16	00000010001100001000100000101010	0230882A
19. ADD R18, R17, R16	00000010001100001001000000100000	02309020
20. SUB R19, R18, R17	00000010010100011001100000100010	02519822
21. ADD R20, R19, R18	00000010011100101010000000100000	0272A020
22. AND R21, R20, R19	00000010100100111010100000100100	0293A824
23. SLL R22, R21, 1	00000000000101011011000001000000	0015B040
24. OR R23, R22, R21	00000010110101011011100000100101	02D5B825
25. SRL R24, R23, 3	00000000000101111100000011000010	0017C0C2
26. ADD R25, R23, R24	00000010111110001100100000100000	02F8C820
27. SUB R26, R25, R23	00000011001101111101000000100010	0337D022
28. ADD R27, R26, R25	00000011010110011101100000100000	0359D820
29. OR R28, R27, R26	00000011011110101110000000100101	037AE025
30. AND R29, R28, R28	00000011100111001110100000100100	039CE824
31. ADD R30, R30, R29	00000011110111011111000000100000	03DDF020
32. SUB R31, R31, R31	00000011111111111111100000100010	03FFF822

Tabla 25: Desglose instrucciones programa 2

Este programa consta de un número de trece instrucciones, las cuales se pueden ver en la tabla 26.

1. ADDI R <sub>16</sub> , R <sub>0</sub> , 3	2. ADDI R <sub>17</sub> , R <sub>0</sub> , 5
3. ADDI R <sub>18</sub> , R <sub>0</sub> , 1	4. ADDI R <sub>19</sub> , R <sub>0</sub> , 15
5. BEQ R <sub>0</sub> , R <sub>19</sub> , 7	6. AND R <sub>20</sub> , R <sub>17</sub> , R <sub>18</sub>
7. BEQ R <sub>0</sub> , R <sub>20</sub> , 1	8. ADD R <sub>22</sub> , R <sub>22</sub> , R <sub>16</sub>
9. ADD R <sub>16</sub> , R <sub>16</sub> , R <sub>16</sub>	10. ADD R <sub>18</sub> , R <sub>18</sub> , R <sub>18</sub>
11. ADDI R <sub>19</sub> , R <sub>19</sub> , -1	12. BEQ R <sub>19</sub> , R <sub>19</sub> , -8
13. BEQ R <sub>19</sub> , R <sub>19</sub> , -1	

Tabla 26: Instrucciones programa 3

Se emplean cuatro tipos de instrucciones: ADDI, ADD, AND Y BEQ. Las 13 instrucciones se desglosan en la tabla 28, donde se puede ver el código binario y hexadecimal de cada instrucción. El resultado que se obtiene al aplicar este programa se expone en el apartado llamado Análisis de resultados.

#### 3.3.4.4. Programa 4

Este cuarto y último programa es una continuación del programa 2, es decir, que son instrucciones que se añaden a las del programa 2 (programa que prueba todos los registros). Se añaden un total de 13 instrucciones como se puede ver en la tabla 27.

1. SW R <sub>0</sub> , 2 (R <sub>31</sub> )	2. LW R <sub>6</sub> , 2 (R <sub>31</sub> )
3. BNE R <sub>0</sub> , R <sub>6</sub> , 2	4. BEQ R <sub>0</sub> , R <sub>6</sub> , 1
5. NOP	6. SW R <sub>1</sub> , 23(R <sub>31</sub> )
7. LW R <sub>10</sub> , 23(R <sub>31</sub> )	8. BNE R <sub>10</sub> , R <sub>0</sub> , 1
9. NOP	10. BEQ R <sub>10</sub> , R <sub>0</sub> , 15
11. BEQ R <sub>31</sub> , R <sub>7</sub> , 2	12. ADD R <sub>31</sub> , R <sub>31</sub> , R <sub>7</sub>
13. J -13	

Tabla 27: Instrucciones programa 4

Como se puede ver se utilizan los valores obtenidos en los registros del programa dos para comprobar el comportamiento de la memoria de datos. Se guardan y cargan valores en esta memoria de datos mediante las instrucciones SW y LW.

También se comprueban el funcionamiento de dos instrucciones que no se han comprobado hasta ahora: BNE y J.

El desglose de todas las instrucciones presentes en este programa se pueden visualizar en la tabla 28,

Con estos cuatro programas se puede ver el comportamiento que tiene el microprocesador implementado.

INSTRUCCIÓN	CÓDIGO BINARIO	CÓDIGO HEXADECIMAL
1. ADDI R16, R0, 3	001000000010000000000000000011	20100003
2. ADDI R17, R0, 5	0010000000100010000000000000101	20110005
3. ADDI R18, R0, 1	00100000000100100000000000000001	20120001
4. ADDI R19, R0, 15	00100000000100110000000000001111	2014000F
5. BEQ R0, R19, 7	00010000000100110000000000000111	10130007
6. AND R20, R17, R18	00000010001100101010000000100100	0232A024
7. BEQ R0, R20, 1	00010000000101000000000000000001	10140001
8. ADD R22, R16, R22	00000010110100001011000000100000	02D0B020
9. ADD R16, R16, R16	00000010000100001000000000100000	02108020
10. ADD R18, R18, R18	00000010010100101001000000100000	02529020
11. ADDI R19, R19, -1	00100010011100111111111111111111	2273FFFF
12. BEQ R19, R19, -8	00010010011100111111111111111000	1273FFF8
13. BEQ R19, R19, -1	00010010011100111111111111111111	1273FFFF

Tabla 28: Desglose Instrucciones programa 3

INSTRUCCIÓN	CÓDIGO BINARIO	CÓDIGO HEXADECIMAL
1. SW R0, 2 (R31)	10101111111000000000000000000010	AFE00002
2. LW R6, 2 (R31)	10001111111001100000000000000010	8FE60002
3. BNE R0, R6, 2	00010100000001100000000000000010	14060002
4. BEQ R0, R6, 1	00010000000001100000000000000001	10060001
5. NOP	00000000000000000000000000000000	00000000
6. SW R1, 23(R31)	101011111110000100000000000001011	AFE1000B
7. LW R10, 23(R31)	100011111110101000000000000001011	8FEA000B
8. BNE R10, R0, 1	00010101010000000000000000000001	15400001
9. NOP	00000000000000000000000000000000	00000000
10. BEQ R10, R0, 15	0001000101000000000000000001111	1140000F
11. BEQ R31, R9, 2	00010011111010010000000000000010	13E90002
12. ADD R31, R31, R9	00000011111010011111100000100000	03E9F820
13. J -13	0000101111111111111111111110011	0BFFFFF3

Tabla 29: Desglose Instrucciones programa 4



# **4. Análisis de los resultados**





## **4.1. Modo de llevar a cabo el análisis**

Como se ha podido comprobar en el apartado anterior, se llevan a cabo dos tipos de pruebas.

La primera es una simulación mediante el programa ModelSim. Esta comprobación es de cada elemento que forma parte del microprocesador por separado. Observando las figuras obtenidas en el apartado anterior, se puede comprobar cómo cada elemento actúa conforme se desea.

La segunda comprobación es mediante la implementación de cuatro programas, los cuales se examinarán a continuación.

## **4.2. Análisis de las simulaciones**

Se realiza una simulación de cada elemento que forma parte del microprocesador. Todas las simulaciones hechas se han mostrado en el apartado anterior, y todas ellas muestran que el resultado de implementar cada elemento es el correcto. No obstante, se realiza un análisis de cada simulación por separado.

### **4.2.1. Registro contador de programa**

La simulación se realiza en la figura 37 (página 75). En ella se puede ver que en su salida (Sal\_PC) existe el valor obtenido en el registro siempre que existe un flanco de subida en su señal de reloj (CLK). El valor que almacena el registro también varía con cada flanco de subida y depende del valor obtenido a la entrada (Ent\_PC).

Este comportamiento es el que se desea para formar parte del microprocesador, ya que este elemento recibe un valor que se sacará por su salida en el siguiente flanco de subida de la señal de reloj, para ir obteniendo las direcciones de las instrucciones.

### **4.2.2. Sumador Más 4**

La simulación de este elemento se muestra en la figura 38 (página 75). Observando la simulación, en la salida del sumador (Sal4) existe el valor obtenido en su entrada (EntPC) sumándole cuatro unidades.

El comportamiento de este elemento se amolda perfectamente a lo que se desea, debido a que junto con el registro contador de programa, forman un registro

contador que va aumentando en 4 unidades la dirección anterior para obtener la siguiente instrucción.

#### **4.2.3. Memoria de instrucciones**

La figura 39 (página 74) es la figura en la que se ve el resultado obtenido al hacer la simulación de la memoria de instrucciones. En ella existe una salida (datos) que tiene los datos obtenidos en la dirección de memoria a la que se accede introduciendo en su entrada (dir) dicha dirección. El valor obtenido a la salida se modifica en cada flanco de subida de la señal de reloj (clk) siempre y cuando también se haya modificado la dirección en su entrada.

Por lo tanto, otro elemento que está bien implementado, ya que para cada programa se introduce en las direcciones de memoria las instrucciones deseadas, y a ellas se accede mediante la dirección que exista a su entrada, que es obtenida del contador de programa.

#### **4.2.4. Banco de registros**

La simulación de este elemento se muestra en la figura 40 (página 74). Sin embargo, en ella no se muestran los 32 registros, sino sólo 8, para así visualizar de forma más sencilla su comportamiento.

En las dos salidas (Dat1 y Dat2) existen los datos que tienen los registros a los que se accede mediante las dos entradas de lectura (Reg1 y Reg2). El valor de los registros se modifica a través de las entradas (Reges y Dates). Con el primero se accede al registro que se desea de los 32 y con el segundo se escribe el dato deseado, siempre y cuando la habilitación de escritura (Escrreg) se encuentre activada. La escritura se realiza en el flanco de subida de la señal de reloj (CLK\_BR).

Visto su comportamiento, el banco de registros implementado queda bien definido ya que realiza las acciones que se piden de él dentro del hardware del microprocesador.

#### **4.2.5. Extensión de signo**

En la figura 42 (página 78) se encuentra la simulación hecha para el elemento denominado Extensión de signo. En ella se observa, que en la salida del elemento (Salex) existen 32 bits. De esos 32 bits, los 16 menos significativos son los mismos a los obtenidos en la entrada del elemento (Entext). Los otros 16 bits son

todos unos o todos ceros, dependiendo de si el bit más significativo de la entrada es un uno o un cero respectivamente.

De nuevo, se comprueba que el funcionamiento obtenido del elemento simulado es el esperado, por lo que el elemento puede formar parte del hardware del microprocesador.

#### **4.2.6. Extensión 5 32 bit**

La simulación de este elemento es la que se puede ver en la figura 43 (página 78).

El resultado de esta simulación indica que el elemento funciona conforme se desea, ya que en la salida (Sal\_sh) existen 32 bits, de los cuales, los 5 bits menos significativos son los bits procedentes en la entrada (Ent\_sh), y los otros 27 son todo ceros para así poder ser introducido el valor de la salida en la entrada de la ALU.

#### **4.2.7. Desplazamiento 2 izquierda (instrucciones tipo J)**

La siguiente figura de las simulaciones, es decir, la figura 44 (página 78) muestra el resultado obtenido al realizar la simulación de este elemento.

En dicha figura, el dato obtenido en la salida del elemento (Sal\_despJ) consta de 28 bits, formados por dos ceros en los bits menos significativos y, los otros 26 bits, los más significativos, son los datos obtenidos en la entrada del elemento (Ent\_despJ), es decir, los mismos 26 bits.

Visto este comportamiento, el elemento creado es idóneo para formar parte del hardware del microprocesador que se desarrolla.

#### **4.2.8. Circuito de control**

Para este circuito, o elemento, se realizan varias simulaciones. Éstas son las que se pueden ver en las figuras que van desde la 45 a 51 (páginas 78 y 80), ambas inclusive.

Este circuito tiene una entrada de 6 bits y, dependiendo del valor que exista en esta entrada, los valores de las salidas se modifican. Estos valores deben coincidir con los valores que existen en la tabla 20 (página 80) para que el funcionamiento del microprocesador sea correcto.

Observando cada una de las figuras (de la 45 a la 51), el resultado obtenido de todas las salidas es el esperado, ya que coincide con los valores que se muestran en la tabla 20. Por lo tanto, se llega a la conclusión de que el circuito de control queda perfectamente implementado y puede pasar a formar parte del hardware del microprocesador.

#### **4.2.9. Desplazamiento 2 izquierda**

El resultado de la simulación de este elemento se puede ver en la figura 52 (página 82). En ella se puede ver cómo en la salida (Saldestp) existen 32 bits, los cuales están formados por 30 bits que existen a su entrada (Entdestp), y por dos ceros, los dos bits menos significativos. Los 30 bits, que se colocan en los bits más significativos, son los 30 bits menos significativos que existen a la entrada por lo que los dos bits más significativos de la entrada se pierden.

Este comportamiento es el deseado, ya que realiza un desplazamiento de dos posiciones de los bits a la izquierda.

#### **4.2.10. Unidad Aritmético Lógica**

Para este elemento, se realizan varias simulaciones, una por cada operación que es capaz de realizar. Estas simulaciones son mostradas en las figuras que van desde la 53 a 58 (páginas 82 y 84), ambas inclusive.

La primera de estas figuras, la número 53, muestra la operación de suma aritmética. Para activar esta operación en la entrada de control de la ALU (Contalu) debe existir el valor 010. Una vez que está activada la operación de sumar, realiza la suma de las otras dos entradas (Datoone y Datotwo) y el resultado es obtenido en la salida (Restalu). Si este resultado es de cero se activa la salida llamada "cero", y, si no lo es, no se activará.

La siguiente figura, la número 54, es en la que se realiza la operación de resta. Se activa mediante el valor 110 en la entrada de control. En la salida se obtiene el valor de resta de las otras dos entradas. La salida denominada como "cero" actúa como se ha descrito antes, es decir, si el resultado es cero se activa, sino, no se activará.

La figura número 55 contiene la simulación de la operación AND, la cual se activa mediante el valor 000 en la entrada de control. Realiza la operación AND bit a bit de las dos entradas y el resultado obtenido se muestra en su salida. La salida llamada "cero" sigue actuando igual que para las operaciones anteriores.

En la figura 56 se realiza la simulación de la operación OR, o suma lógica. En la entrada de control debe existir el número 001. Se hace la operación OR bit a bit de las dos entradas y el resultado se obtiene a través de la salida. De nuevo, la salida “cero” tiene el mismo comportamiento que para las otras operaciones.

La siguiente figura, la número 57, muestra dos operaciones: SLL y SRL. Son las operaciones de rotación y en la entrada de control deben existir los valores 100 y 101 respectivamente. Para la primera se realiza una rotación del dato obtenido en la entrada “Datoone” a la izquierda de tantas posiciones de los bits conforme al valor obtenido por la entrada “Datotwo”, y en la segunda la rotación a la derecha.

Por último, la última simulación de este componente, mostrada en la figura 58, sirve para comprobar el funcionamiento de la operación SLT. En la entrada de control debe aparecer el valor 111. El resultado a la salida depende de si el dato obtenido en la primera entrada es menor que el dato obtenido en la segunda entrada. Si el primero es menor a la salida se obtiene el valor de uno, y sino el valor cero.

Con todas estas simulaciones, se llega a la conclusión que el componente denominado como ALU queda bien implementado, así que puede formar parte del hardware del microprocesador.

#### **4.2.11. Circuito de control de la ALU**

Este circuito es el que genera la señal de control para la ALU y para dos multiplexores. La simulación creada para ver el comportamiento de este circuito es la que se puede ver en la figura 59 (página 84).

La entrada llamada “aluop” tiene tres posibles valores. Con las dos primeras (00 y 01) da igual qué valor se obtenga en la otra entrada (camfun). Para el primer caso la salida (contalu) obtenida tiene el valor 010 (suma) y, para el segundo, tiene el valor 110 (resta).

El otro posible valor en la entrada “aluop” es 10. Con este valor la salida depende también de la entrada “camfun”. Como se puede ver en la tabla 21 (página 85) los valores obtenidos en la simulación coinciden con lo esperado, ya que los valores de la tabla 21 coinciden con los valores obtenidos en la simulación.

Las otras dos salidas, las señales de control de los dos multiplexores, los cuales sirven para las operaciones de rotación, sólo se activan cuando se ejecutan las instrucciones SLL y SRL.

El comportamiento de este circuito es bueno, ya que realiza exactamente lo que se desea de él.

#### **4.2.12. Sumador**

La simulación de este elemento es la mostrada en la figura 60 (página 86). En ella se puede ver que los datos que existen en las dos entradas (Entsum1 y Entsum2) son sumados y el resultado obtenido se saca por la salida del elemento (SalSum).

El comportamiento descrito del sumador coincide perfectamente con la simulación realizada, por lo que se puede introducir en el hardware del microprocesador.

#### **4.2.13. Memoria de datos**

Las simulaciones de la memoria de datos son las que se pueden ver en las figuras 61, 62 y 63 (página 86).

La primera de estas simulaciones muestra cómo se escribe en la memoria. Siempre que esté habilitada la opción de escritura (We) se introduce en la dirección de la memoria dada por la entrada (a) el dato obtenido en la otra entrada (d), la entrada de datos. Esta escritura se realiza en el flanco de bajada de la señal de reloj (CLK\_MD).

En la segunda simulación se realiza la lectura de un dato que existe ya en la memoria. Para ello debe estar activada la habilitación de lectura (lect). Se lee la dirección obtenida en la entrada de direcciones (a), y el valor leído se obtiene en la salida (q). La lectura también se realiza en el flanco de bajada de la señal de reloj (CLK\_MD).

Por último, la tercera simulación muestra las dos operaciones, la escritura y la lectura de la memoria.

Por lo tanto, la memoria implementada está bien definida y se puede introducir en el hardware del microprocesador.

#### **4.2.14. Unión 28+4 bits**

Este elemento se simula en la figura 64 (página 88). En la salida de este elemento (Sal32) se obtiene un dato con un total de 32 bits, de los cuales los 4 bits más significativos pertenecen a la entrada llamada "ent4" y los otros 28 a la otra entrada, la llamada "ent28".

Este comportamiento es el deseado, pues se necesita el elemento que una los 4 bits del contador y la salida del desplazamiento 2 izquierda (instrucciones J), por lo que se puede introducir en el diseño del hardware del microprocesador.

#### **4.2.15. Multiplexores**

En este caso no se simulan todos los multiplexores implementados. Se realiza sólo una simulación de un multiplexor de 1 bit, ya que el comportamiento de los demás es igual. Esta simulación se puede ver en la figura 65 (página 89).

Cuando la señal de control del multiplexor (Contmux1b) tiene el valor de cero, en la salida (salmux1b) existe el valor obtenido en la entrada 1 (Entmux1b\_1), y si existe un uno, el valor obtenido en la entrada 2 (Entmux1b\_2).

El comportamiento de los multiplexores es el adecuado, por lo que se puede introducir en el hardware del microprocesador.

### **4.3. Análisis de los programas ejecutados**

Para realizar una comprobación más eficiente que la simulación, se debe comprobar el microprocesador al completo. Primero se unen todos los componentes como ya se ha explicado, y luego se introducen una serie de programas que implementen todas las instrucciones, y así comprobar el correcto funcionamiento del microprocesador.

Viendo el apartado 3 del Trabajo Fin de Grado, el número de programas que se ejecutan para comprobar el funcionamiento del microprocesador son cuatro. A continuación se analiza el comportamiento de cada uno de ellos.

Para ver los resultados en la tarjeta Basys 2, se muestra en la figura 67 el nombre que se ha dado a cada conexión del microprocesador. El resultado se puede observar gracias a una serie de multiplexores añadidos, ya que no hay suficientes salidas en la tarjeta Basys 2. Todo el código implementado se muestra en el anexo de programación.

#### **4.3.1. Programa 1**

Se trata de un programa formado por 18 instrucciones, y es un programa que se emplea como ejercicio en la asignatura Electrónica Digital para explicar el comportamiento del microprocesador.

Las 18 instrucciones que forman el programa deben introducirse en código hexadecimal en la memoria de instrucciones, en las posiciones múltiples de cuatro, es decir, la primera en la cuatro, la segunda en la ocho, y así sucesivamente. El código hexadecimal de las instrucciones se puede ver en la tabla 23 (página 93).

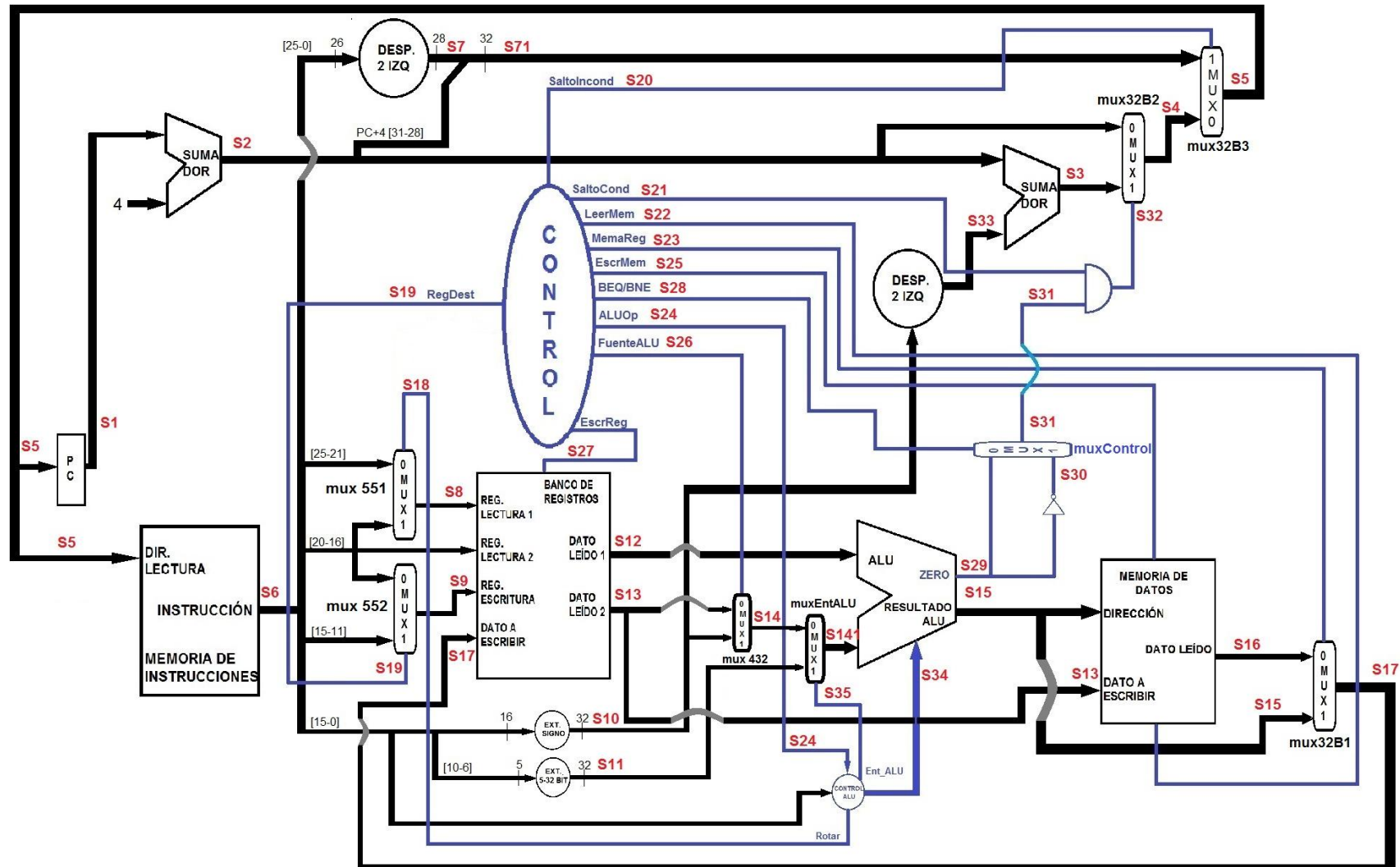


Figura 67: Señales del microprocesador









- Entrada ALU 1 (S12): 00000000000000000000000000000000
- Entrada ALU 2 (S141): 00000000000000000000000000000000
- Resultado ALU (S15): 00000000000000000000000000000000
- Salida extensión de signo (S10): 0000000000000000  
0000000000000011
- Salida sumador (S3): 0000000000000000000000000000111000  
(56)  $\rightarrow 56 / 4 = 14$
- Entrada contador (S5): Salida sumador
- Salto condicional (S21): 1
- BEQ/BNE (S28): 0
- Entrada control multiplexor salto (S32): 1

11)SLT R<sub>4</sub>, R<sub>1</sub>, R<sub>2</sub>. Como R<sub>1</sub>=8 es menor que R<sub>2</sub>=32, se escribe en el registro R<sub>4</sub> el valor 1.

- Salida memoria instrucciones (S6): 000000 00001 00010 00100  
00000 101010
- Entrada registro lectura 1 (S8): 00001
- Entrada registro lectura 2: 00010
- Registro de Escritura (S9): 00100
- Dato a escribir (S17): 000000000000000000000000000000001
- Entrada ALU 1 (S12): 000000000000000000000000000001000
- Entrada ALU 2 (S141): 0000000000000000000000000000100000
- Resultado ALU (S15): Dato a escribir

12)BEQ R<sub>4</sub>, R<sub>7</sub>, 3. El contenido de R<sub>4</sub>=1 es distinto del de R<sub>7</sub>=0, por lo tanto no se produce salto de instrucción.

- Salida memoria instrucciones (S6): 000100 00100 00111  
0000000000000011
- Entrada registro lectura 1 (S8): 00100
- Entrada registro lectura 2: 00111
- Entrada ALU 1 (S12): 000000000000000000000000000000001
- Entrada ALU 2 (S141): 000000000000000000000000000000000
- Resultado ALU (S15): 000000000000000000000000000000001
- Salida extensión de signo (S10): 0000000000000000  
0000000000000011
- Salida sumador (S3): 00000000000000000000000000001001100  
(76)  $\rightarrow 76 / 4 = 19$
- Entrada contador (S5): 0000000000000000000000000000100000  
(64)  $\rightarrow 64 / 4 = 16$
- Salto condicional (S21): 1
- BEQ/BNE (S28): 0
- Entrada control multiplexor salto (S32): 0



La ejecución de este programa se ha hecho de forma satisfactoria debido al buen funcionamiento del microprocesador respecto a las instrucciones ejecutadas. No obstante se necesita realizar más pruebas ya que no se han probado todo el conjunto de instrucciones.

Se puede decir que varias instrucciones funcionan conforme se desea debido al resultado obtenido. Estas instrucciones son las que ya se han ejecutado, ADD, SLT, ADDI, BEQ. A continuación se siguen realizando pruebas para ejecutar más instrucciones.

### 4.3.2. Programa 2

El segundo de los programas a ejecutar sirve para comprobar que en el banco de registros se pueden utilizar los 32 registros que contiene. Para ello se escribe y se lee todos los registros que contiene. También, a la vez, se comprueba en este programa todas las instrucciones tipo R ya que con ellas se puede interactuar con los registros y operar entre ellos.

El programa contiene un total de 32 instrucciones, tal y como se puede observar en la tabla 24 (página 94). Estas 32 instrucciones se deben introducir en la memoria de instrucciones en código hexadecimal como en el caso del programa anterior. El código hexadecimal de cada instrucción se puede ver en la tabla 25 (página 95). La implementación de la memoria de instrucciones para este programa se muestra en el anexo de programación.

Se realiza la compilación del microprocesador con la memoria de instrucciones de este programa y, a continuación se realiza el análisis de las instrucciones que se ejecutan. Sin embargo, para este programa, como son muchas instrucciones y, se repiten varias instrucciones cambiando únicamente los registros con los que se operan, se realiza un único análisis para cada instrucción y al final se añade una tabla con el contenido que existe en cada registro.

- Instrucción 1. ADDI R<sub>0</sub>, R<sub>0</sub>, -25. Se escribe en el registro R<sub>0</sub> el valor -25. Así quedan las señales cuando se ejecuta la instrucción.
  - Salida memoria instrucciones (S6): 001000 00000 00000  
1111111111100111
  - Entrada registro de lectura 1 (S8): 00000
  - Entrada registro escritura (S9): 00000
  - Dato a escribir (S17): 11111111111111111111111111111100111
  - Entrada 1 ALU (S12): 00000000000000000000000000000000
  - Entrada 2 ALU (S141):11111111111111111111111111111100111
  - Resultado ALU: Dato a escribir

- Instrucción 3. ADD R<sub>2</sub>, R<sub>0</sub>, R<sub>1</sub>. Se realiza la suma de los valores obtenidos de los registros R<sub>0</sub>= -25 y R<sub>1</sub>= -23 y el resultado se almacena en el registro R<sub>2</sub>.
  - Salida memoria instrucciones (S6): 000000 00000 00001 00010  
00000 100000
  - Entrada registro de lectura 1 (S8): 00000
  - Entrada registro de lectura 2: 00001
  - Entrada registro escritura (S9): 00010
  - Dato a escribir (S17): 11111111111111111111111111111111010000  
(-48)
  - Entrada 1 ALU (S12): 1111111111111111111111111111111100111  
(-25)
  - Entrada 2 ALU (S141):1111111111111111111111111111111101001  
(-23)
  - Resultado ALU: Dato a escribir
  
- Instrucción 5. AND R<sub>4</sub>, R<sub>2</sub>, R<sub>3</sub>. Se realiza la operación lógica AND de los valores obtenidos de los registros R<sub>2</sub> y R<sub>3</sub> y el resultado se almacena en el registro R<sub>4</sub>.
  - Salida memoria instrucciones (S6): 000000 00010 00011 00100  
00000 100100
  - Entrada registro de lectura 1 (S8): 00010
  - Entrada registro de lectura 2: 00011
  - Entrada registro escritura (S9): 00100
  - Dato a escribir (S17): 11111111111111111111111111111111010000
  - Entrada 1 ALU (S12): 11111111111111111111111111111111010000
  - Entrada 2 ALU (S141): 11111111111111111111111111111111010000
  - Resultado ALU: Dato a escribir
  
- Instrucción 6. OR R<sub>5</sub>, R<sub>4</sub>, R<sub>0</sub>. Se realiza la operación lógica AND de los valores obtenidos de los registros R<sub>2</sub> y R<sub>3</sub> y el resultado se almacena en el registro R<sub>4</sub>.
  - Salida memoria instrucciones (S6): 000000 00100 00000 00101  
00000 100101
  - Entrada registro de lectura 1 (S8): 00100
  - Entrada registro de lectura 2: 00000
  - Entrada registro escritura (S9): 00101
  - Dato a escribir (S17): 1111111111111111111111111111111110111
  - Entrada 1 ALU (S12): 11111111111111111111111111111111010000
  - Entrada 2 ALU (S141): 1111111111111111111111111111111100111
  - Resultado ALU: Dato a escribir

- Instrucción 15. SRL R<sub>14</sub>, R<sub>13</sub>, 23. Se realiza la operación de rotación 23 posiciones hacia la derecha del valor obtenido en el registro R<sub>13</sub> y el resultado se almacena en el registro R<sub>14</sub>.

- Salida memoria instrucciones (S6): 000000 00000 01101 01110 10111 000010
- Entrada registro de lectura 1 (S8): 01101
- Entrada registro escritura (S9): 00101
- Salida EXT 5\_32bit (S11): 00000000000000000000000000000000010111
- Dato a escribir (S17): 0000000000000000000000000111111111
- Entrada 1 ALU (S12): 111111111111111111111111111110111
- Entrada 2 ALU (S141): Salida EXT 5\_32bit
- Resultado ALU: Dato a escribir

- Instrucción 17. SUB R<sub>16</sub>, R<sub>15</sub>, R<sub>14</sub>. Se realiza la resta de los valores obtenidos de los registros R<sub>15</sub>= -603979776 y R<sub>14</sub>= 255 y el resultado se almacena en el registro R<sub>16</sub>.

- Salida memoria instrucciones (S6): 000000 01111 01110 10000 00000 100010
- Entrada registro de lectura 1 (S8): 01111
- Entrada registro de lectura 2: 01110
- Entrada registro escritura (S9): 10000
- Dato a escribir (S17): 1101101111111111111111111000000001 (-603980031)
- Entrada 1 ALU (S12): 111111111111111111111111111100111 (-603979776)
- Entrada 2 ALU (S141):111111111111111111111111111101001 (255)
- Resultado ALU: Dato a escribir

- Instrucción 23. SLL R<sub>22</sub>, R<sub>21</sub>, 1. Se realiza la operación de rotación una posición hacia la izquierda del valor obtenido en el registro R<sub>22</sub> y el resultado se almacena en el registro R<sub>21</sub>.

- Salida memoria instrucciones (S6): 000000 00000 10101 10110 00001 000000
- Entrada registro de lectura 1 (S8): 10101
- Entrada registro escritura (S9): 10110
- Salida EXT 5\_32bit (S11): 0000000000000000000000000000000001
- Dato a escribir (S17): 001001111111111111111111100000000010
- Entrada 1 ALU (S12): 10010011111111111111111110000000001
- Entrada 2 ALU (S141): Salida EXT 5\_32bit
- Resultado ALU: Dato a escribir





anterior realiza los saltos cuando es requerido, y cuando no lo es, no se realizan dichos saltos. Sin embargo se siguen realizando más programas para seguir comprobando el correcto funcionamiento del microprocesador.

### **4.3.3. Programa 3**

En este tercer programa, se implementa otro ejercicio que se expone en la asignatura de Electrónica Digital. Servirá para realizar una mejor comprensión del funcionamiento del microprocesador. El programa realiza una multiplicación de dos números:  $3 \times 5 = 15$ .

Como en los dos programas anteriores, se debe crear la memoria de instrucciones con las instrucciones que desarrollarán el programa. Estas instrucciones se pueden ver en la tabla 28 (página 97), tanto en código binario, como en código hexadecimal. El código decimal es el que se introduce en la memoria de instrucciones, que quedará conformada tal y como se puede ver en el anexo de programación.

Una vez creada, acoplada y compilada la memoria de instrucciones al resto del hardware del microprocesador, se realiza la compilación de éste último y se procede a realizar el análisis de los resultados obtenidos con la tarjeta Basys 2.

El programa consiste en ir comprobando los dígitos del multiplicador, normalmente se va desplazando a la derecha y se comprueba el dígito menos significativo, y desplazando a la izquierda el multiplicando. Si el dígito es uno se suma el nuevo valor del multiplicando al resultado. Si el dígito es cero, tan solo se desplaza el multiplicando.

- Instrucciones de la 1 a la 4. Son instrucciones ADDI para escribir en los registros cuatro valores: 3, 5, 1 y 15. Son los números que se multiplican, el número que guarda una máscara para realizar operaciones, y el resultado final. Estos valores se almacenan respectivamente en los registros R<sub>16</sub>, R<sub>17</sub>, R<sub>18</sub> y R<sub>19</sub>.
- Instrucción 5. BEQ R<sub>0</sub>, R<sub>19</sub>, 7. Como el valor del registro R<sub>0</sub> siempre es cero, sólo se producirá el salto cuando el valor del registro R<sub>19</sub>, sea cero. Esta instrucción se realiza para que el número total de iteraciones sea como máximo de 15.
- Instrucción 6. AND R<sub>20</sub>, R<sub>17</sub>, R<sub>18</sub>. Se realiza una operación para ver si el último bit es un cero o un uno. El valor obtenido se almacena en el registro 20.
- Instrucción 7. BEQ R<sub>0</sub>, R<sub>20</sub>, 1. Comparación entre el valor del registro cero, que siempre tiene el valor cero, con el valor que se obtiene en el registro 20.



Se introducen en la memoria de instrucciones todas las instrucciones, tanto las del segundo programa como las trece que se añaden. La memoria de instrucciones que se implementa para este programa se puede ver en el anexo de programación.

A continuación se explica el comportamiento del programa instrucción a instrucción después de realizar la prueba con la tarjeta Basys 2.

- Las 32 primeras instrucciones, como ya se ha dicho son las que se implementan en el programa 2, por lo tanto su explicación es la mostrada en el apartado 4.3.2.
- Instrucción 33. SW R0, 2 (R31). Instrucción para probar la escritura de un dato en la memoria de datos. Se realiza la escritura en la memoria de datos en la dirección 2, debido a que el registro 31 tiene el dato guardado de cero y se suma un dato de valor dos. El dato que se escribe es el dato que almacena el registro cero.
  - Salida memoria instrucciones (S6): 101011 11111 00000 00000000000000010
  - Entrada registro de lectura 1 (S8): 11111
  - Entrada registro lectura 2: 00000
  - Salida EXT\_Signo (S10): 000000000000000000000000000000010
  - Dirección memoria de datos (S15): 000000000000000000000000000000010
  - Dato a escribir en la memoria de datos (S13): 111111111111111111111111111111100111
  - Entrada 1 ALU (S12): 000000000000000000000000000000000
  - Entrada 2 ALU (S141): Salida EXT\_Signo
  - Resultado ALU: Dirección memoria de datos.
- Instrucción 34. LW R6, 2 (R31). Se comprueba si se ha escrito en la dirección de memoria 2 el dato que se le proporcionó en la instrucción anterior. Para ver si es el mismo valor se carga en otro registro, el seis.
  - Salida memoria instrucciones (S6): 100011 11111 00110 00000000000000010
  - Entrada registro de lectura 1 (S8): 11111
  - Entrada registro escritura (S9): 00110
  - Entrada dato a escribir (S17): 111111111111111111111111111111100111
  - Salida EXT\_Signo (S10): 000000000000000000000000000000010
  - Dirección memoria de datos (S15): 000000000000000000000000000000010
  - Salida memoria de datos (S16): Entrada dato a escribir.
  - Entrada 1 ALU (S12): 000000000000000000000000000000000
  - Entrada 2 ALU (S141): Salida EXT\_Signo
  - Resultado ALU: Dirección memoria de datos.

- Instrucción 35. BNE R<sub>0</sub>, R<sub>6</sub>, 2. Comparación entre los dos registros para ver si el valor es el mismo, y así comprobar que se ha realizado bien la escritura y lectura en la memoria. En el caso de que sean diferentes se realizaría un salto de dos instrucciones. Como los dos registros son iguales no se realiza el salto.
  - Salida memoria instrucciones (S6): 000101 00000 00110 0000000000000010
  - Entrada registro de lectura 1 (S8): 00000
  - Entrada registro de lectura 2: 00110
  - Salida EXT\_Signo (S10): 00000000000000000000000000000010
  - Entrada 1 ALU (S12): 1111111111111111111111111111111100111
  - Entrada 2 ALU (S141): 1111111111111111111111111111111100111
  - Resultado ALU: 000000000000000000000000000000000
  - Salto condicional (S21): 1
  - BNE/BEQ (S28): 1
  - Entrada de control multiplexor de salto condicional: 0
- Instrucción 36. BEQ R<sub>0</sub>, R<sub>6</sub>, 1. Mismo caso que el anterior, pero en este caso se realiza el salto de instrucción ya que la instrucción es “salto si son iguales”.
- Instrucción 37. NOP. No realiza ninguna operación, aunque esta instrucción se salta debido a la instrucción anterior.
- Instrucción 38 y 39. Dos instrucciones para guardar y cargar datos en la memoria de datos en la dirección 23. Primero se escribe el dato que tiene el registro 1, y después este valor se carga en el registro 10.
  - Dato escrito: 1111111111111111111111111111111101001
- Instrucción 40. BNE R<sub>10</sub>, R<sub>0</sub>, 1. Comparación del valor que contienen los dos registros, el cero y el diez. Como son diferentes se producirá el salto de instrucción y salta hasta la instrucción 42.
  - R<sub>10</sub>: 1111111111111111111111111111111101001
  - R<sub>0</sub>: 1111111111111111111111111111111100111
- Instrucción 42. BEQ R<sub>10</sub>, R<sub>0</sub>, 15. Como los datos almacenados en los registros son diferentes no se produce el salto.
- Instrucción 43. BEQ R<sub>31</sub>, R<sub>9</sub>, 2. Como en este caso los datos almacenados siguen sin ser iguales tampoco se produce el salto de instrucción.
- Instrucción 44. ADD R<sub>31</sub>, R<sub>31</sub>, R<sub>9</sub>. Se suma al registro 31, el valor del registro 9 para que en la siguiente iteración, la instrucción anterior salte a la instrucción 46 y así evitar el salto incondicional que se produce en la instrucción 45.

- Instrucción 45. J -13. Se produce un salto incondicional a la primera instrucción de este programa, es decir a la instrucción 33 para volver a probar la memoria de datos, ahora en direcciones diferentes a las direcciones que se accedía antes, debido a que el valor del dato almacenado en el registro 31 cambia en la instrucción anterior y pasa a tener el valor del registro 9 111111111111111111111111111111110111 (-9).

El resultado obtenido con este último programa es el esperado, por lo tanto el microprocesador MIPS queda perfectamente implementado, ya que su funcionamiento es óptimo.

#### **4.4. Análisis a las modificaciones realizadas en el diseño debido al dispositivo FPGA**

Debido a algunas características que posee la tarjeta FPGA, se han realizado diferentes modificaciones respecto al diseño tanto hardware como software del microprocesador.

La primera modificación que se realiza es cambiar la colocación del registro contador debido a los retrasos que pudieran suceder. Se puede ver en la figura 67 (página 106). Como tanto el registro contador como la memoria de instrucciones realizan sus modificaciones con el flanco ascendente de la señal de reloj, para evitar fallos de sincronización, en vez de conectar la salida del registro contador con la memoria de instrucciones, a sus respectivas entradas siempre se coloca la salida del multiplexor que elige el valor siguiente en la cuenta (S5). Se tiene así dos registros funcionando en paralelo, el registro contador y el registro que posee la memoria a su entrada. A este último no se puede acceder.

Señalar también que la memoria de instrucciones es una memoria síncrona ROM. Su lectura se realiza en los flancos ascendentes de la señal de reloj, y no puede ser escrita mediante el empleo del microprocesador, solo puede ser reescrita accediendo a su software de implementación.

En el hardware del microprocesador también se llevan a cabo modificaciones con respecto al hardware del microprocesador MIPS que se ha tomado como base, presente en el documento “Diseño del procesador MIPS R2000” [5]. Se añaden diversos multiplexores, ya que el conjunto de instrucciones implementado es mayor y, por lo tanto, el diseño hardware no será igual. También se añaden elementos como el elemento llamado “Extensión 5\_32bit” o una puerta NOT.

Por último, destacar que la memoria de datos implementada también es una memoria síncrona. La memoria de datos es una memoria RAM, es decir, es una memoria tanto de escritura como de lectura.

Sin embargo, el funcionamiento al realizar las pruebas pertinentes de esta memoria de datos no era el idóneo, y se ha realizado una modificación. Sólo se implementa la escritura de manera síncrona con la pertinente habilitación. La lectura de esta memoria se realiza de manera asíncrona y sin habilitación, es decir, siempre a la salida del elemento existe el dato correspondiente a la dirección que existe en su entrada.

Todas estas modificaciones se han realizado para optimizar el comportamiento del microprocesador MIPS básico teniendo en cuenta las características del dispositivo FPGA en el que se va a implementar.





# 5. Conclusiones



## **5.1. Cumplimiento de los objetivos**

El primer de los objetivos, el principal, es el primer objetivo cumplido, ya que se implementa correctamente y de manera eficiente un microprocesador MIPS básico mediante el uso de un dispositivo FPGA programándole en lenguaje VHDL.

Como ya se ha dicho en el apartado anterior, para que el funcionamiento del microprocesador MIPS básico sea el esperado se han tenido que realizar diversas modificaciones. De esta forma, al funcionar perfectamente, se satisface al 100% este primer objetivo.

Se puede decir que el comportamiento del microprocesador MIPS básico es idóneo ya que realiza todo el conjunto de instrucciones para el que está diseñado y, así ejecuta las diferentes tareas que se desean. Estas tareas son:

- Realiza diferentes operaciones lógicas.
- Realiza diferentes operaciones aritméticas.
- Realiza otro tipo diferente de operaciones, como por ejemplo rotaciones, saltos, ...
- Lee elementos almacenados en memoria de doble puerto.
- Escribe y almacena diferentes datos o elementos en una memoria RAM interna (Memoria de datos).
- Modifica los datos o elementos que se tienen alojados en la memoria RAM (Memoria de datos)

Una vez cumplido el objetivo principal, también se consuman los objetivos personales, debido a que son enfoques que parten de este primero.

A nivel personal, me he enriquecido en diferentes conocimientos que pueden ser beneficiosos para el futuro ya que, como se indica al principio del Trabajo Fin de Grado, los microprocesadores y los dispositivos FPGAs son dos elementos con gran potencial en la electrónica digital del presente y del futuro.

Respecto a este enriquecimiento del conocimiento de estos dispositivos, merece la pena señalar que ya puedo profundizar en temas relacionados con estos dispositivos, pues la base teórica y práctica está afianzada al desarrollar el presente Trabajo Fin de Grado. Al ser dos elementos con la mirada puesta en el futuro, tienen un amplio espectro de uso y de mejora, el cual aún no está definido el límite.

También he afianzado los conocimientos en la programación en lenguaje VHDL. Aunque no he partido de cero, debido a que contaba con una base teórica y práctica adquirida al cursar la asignatura Métodos y Herramientas de Diseño Electrónico, el entendimiento y capacidad de desarrollar programas en este lenguaje ha aumentado considerablemente.

## **5.2. Apreciaciones del Trabajo Fin de Grado**

Después de sacar las conclusiones anteriores, para finalizar el presente Trabajo Fin de Grado, se exponen una serie de apreciaciones sobre todo el trabajo desarrollado.

En primer lugar, el Trabajo Fin de Grado desarrollado se ha terminado de forma satisfactoria, pues todos los objetivos planteados han sido correctamente cumplidos y de forma idónea. Cada uno de estos objetivos, se han ido cumpliendo, partiendo del objetivo principal que es el desarrollo del microprocesador MIPS básico en un dispositivo FPGA.

Como los demás objetivos parten de este objetivo principal también se han ido cumpliendo conforme se ha ido desarrollando el microprocesador.

Segundo, señalar que conforme se han cumplido los objetivos, el enriquecimiento personal en conocimientos desarrollados en este Trabajo Fin de Grado ha sido ampliado extensamente. Esto se ha producido gracias a diversos factores, como son los artículos leídos para conocer aspectos que se desconocían respecto a los microprocesadores o a los dispositivos FPGA, o también gracias a mi tutor, Santiago Cáceres, que me ha proporcionado estos artículos y mucha más información que he ido necesitando para desarrollar el microprocesador MIPS básico programándolo en un dispositivo FPGA en lenguaje VHDL.

Otro aspecto que quiero destacar es la forma de cómo se ha ido desarrollando el microprocesador. Se consideraron diversas opciones, pero la elegida ha sido desarrollar cada parte del microprocesador por separado y luego unirlos. Estas partes o elementos del microprocesador son obtenidos gracias al conjunto de instrucciones que puede ejecutar el microprocesador.

Se consideró la idea de desarrollar el microprocesador como un solo elemento, pero se desechó por dos consideraciones, debido a que se trata de un desarrollo mucho más complejo y menos intuitivo, y que como este diseño se utilizará para explicar el funcionamiento y arquitectura del microprocesador, al realizarlo por partes será más fácil de comprender.

Otro aspecto que creo que es importante señalar, que este microprocesador al ser básico, puede ser la base de otros Trabajos Fin de Grado, aumentando su

complejidad, conjunto de instrucciones, elementos que lo componen (hardware), cantidad de programas que puede ejecutar, etc.

Por último, como este Trabajo Fin de Grado se realiza con el objetivo de que el microprocesador desarrollado sea utilizado en la asignatura “Electrónica Digital” para explicar el temario de microprocesadores con arquitectura MIPS, quiero expresar mi deseo que este microprocesador sea el adecuado para su mejor y más fácil comprensión, y así poder facilitar a mi tutor las clases que imparta en el futuro.



# 6. Bibliografía





- [1] «Introducción a la Tecnología FPGA: Los Cinco Beneficios Principales - National Instruments». Disponible en: <http://www.ni.com/white-paper/6984/es/>.
  
- [2] J. A. Sainz, «Introducción a los microprocesadores». Área de tecnología electrónica. E. U. I. T. I. Vitoria-Gasteiz, 2000.
  
- [3] David Money Harris & Sarah L. Harris. «Digital Design and Computer Architecture (2nd Ed)».
  
- [4] Victor P. Rubio, B.S «A FPGA Implementation of a MIPS RISC Processor for Computer Architecture Education».
  
- [5] Sergio Barrachina Mir. «Diseño del procesador MIPS R2000». Disponible en: [http://www.uv.es/serhocal/docs/7\\_disenyo\\_procesador.pdf](http://www.uv.es/serhocal/docs/7_disenyo_procesador.pdf).
  
- [6] J. A. Álvarez Bermejo, *Estructura de computadores: programación del procesador MIPS y su ensamblador*.
  
- [7] V. Sklyarov, *Synthesis and optimization of FPGA-based systems*. 2014.
  
- [8] «Digilent Inc. - Digital Design Engineer's Source». Disponible en: <http://www.digilentinc.com/Products/Detail.cfm?Prod=BASYS2&NavTop=2&NavSub=649&CFID=10237876&CFTOKEN=4355fe2265042d12-C9F1FD43-5056-0201-02133DB056D93AB2>.
  
- [9] «Digilent Basys2 Board Reference Manual». [www.digilentinc.com](http://www.digilentinc.com).



# Anexo de programación



## Registro Contador de programa

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

entity Reg_PC is
  Port ( Ent_PC : in  STD_LOGIC_VECTOR (31 downto 0); --Entrada al registro contador
        clk_PC : in  STD_LOGIC;                --Entrada de reloj
        Sal_PC : out STD_LOGIC_VECTOR (31 downto 0));--Salida del registro contador
end Reg_PC;

architecture Registro_Contador of Reg_PC is

--Definición del registro
Signal Registro : std_logic_vector (31 downto 0):="00000000000000000000000000000000";

begin

process (clk_PC, Ent_PC)

begin

if clk_PC'event and clk_PC = '1' then --activacion cuando existe flanco ascendente

  Registro <= Ent_PC; --el registro pasa a valer el valor que existe en la entrada

end if;

Sal_PC <= Registro; --por la salida se obtiene el valor actual del registro

end process;

end Registro_Contador;
```

## Sumador más 4 (Contador de programa)

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

entity Sumador_Mas4 is
  Port ( EntPC : in  STD_LOGIC_VECTOR (31 downto 0); --entrada para valor a sumar
        Sal4 : out STD_LOGIC_VECTOR (31 downto 0)); --salida del sumador+4
end Sumador_Mas4;

architecture Sum_Mas4 of Sumador_Mas4 is

begin

Sal4 <= EntPC + "00000000000000000000000000000100";--a la entrada se le suma el valor 4

end Sum_Mas4;
```







```
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000" );
```

begin

```
process (clk_MI) begin  
  if clk_MI'event and clk_MI='1' then --flanco ascendente de la señal de reloj  
    if (en = '1') then  
      Datos <= ROM(conv_integer(Dir));--salida de datos dependiendo de la entrada  
    end if;  
  end if;  
end process;
```

end M\_Ins;





```
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000" );
```

```
begin
```

```
    process (clk_MI) begin  
        if clk_MI'event and clk_MI='1' then    --flanco ascendente de la señal de reloj  
            if (en = '1') then  
                Datos <= ROM(conv_integer(Dir));--salida de datos dependiendo de la entrada  
            end if;  
        end if;  
    end process;
```

```
end M_Ins;
```













```
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000",  
x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000", x"00000000" );
```

```
begin
```

```
    process (clk_MI) begin  
        if clk_MI'event and clk_MI='1' then    --flanco ascendente de la señal de reloj  
            if (en = '1') then  
                Datos <= ROM(conv_integer(Dir));--salida de datos dependiendo de la entrada  
            end if;  
        end if;  
    end process;
```

```
end M_Ins;
```

## Desplazamiento 2 izquierda (instrucciones J)

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

entity Desp2_izqJ is
  Port ( Ent_DesplJ : in  STD_LOGIC_VECTOR (25 downto 0); --entrada desplazamiento J
        Sal_DesplJ : out STD_LOGIC_VECTOR (27 downto 0));--salida desplazamiento J
end Desp2_izqJ;

architecture DesplazamientoJ of Desp2_izqJ is

begin

proceso_desplazar: process(Ent_DesplJ)

  variable z: STD_LOGIC_VECTOR(27 downto 0);

  begin

  z := (others => '0');    --Vector de 28 bits => todos ceros

  z (27 downto 2):= Ent_DesplJ; --Se introduce el vector de entrada en los 26 bits mas significativos

  Sal_DesplJ <= z (27 downto 0);--en la salida se saca la entrada mas dos cero a la derecha

end process;

end DesplazamientoJ;
```

## Circuito de control

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

entity Control_MIP is
  Port ( OpCode : in  STD_LOGIC_VECTOR (5 downto 0);--campo de la instruccion OPCODE
        -----salidas del circuito de control-----
        RegDest : out  STD_LOGIC;
        SaltoCond : out  STD_LOGIC;
        SaltoIncond : out  STD_LOGIC;
        LeerMem : out  STD_LOGIC;
        MemaReg : out  STD_LOGIC;
        ALUop : out  STD_LOGIC_VECTOR (1 downto 0);
        EscrMem : out  STD_LOGIC;
        FuenteALU : out  STD_LOGIC;
        BNE : out  STD_LOGIC;
        EscrReg : out  STD_LOGIC);
end Control_MIP;

architecture Control of Control_MIP is

begin

Arquitectura_Control : process (OpCode)

begin

  if OpCode = "000000" then  --Tipo R

    RegDest <= '1';
    SaltoCond <= '0';
    SaltoIncond <= '0';
    LeerMem <= '0';
    MemaReg <= '0';
    ALUop <= "10";
    EscrMem <= '0';
    FuenteALU <= '0';
    EscrReg <= '1';

  elsif OpCode = "100011" then  --Tipo LW

    RegDest <= '0';
    SaltoCond <= '0';
    SaltoIncond <= '0';
    LeerMem <= '1';
    MemaReg <= '1';
    ALUop <= "00";
    EscrMem <= '0';
    FuenteALU <= '1';
    EscrReg <= '1';

  elsif Opcode = "101011" then  --Tipo SW

    SaltoCond <= '0';
    SaltoIncond <= '0';
```

```

LeerMem <= '0';
ALUop <= "00";
EscrMem <= '1';
FuenteALU <= '1';
EscrReg <= '0';

elsif OpCode = "000100" then --Tipo BEQ

    SaltoCond <= '1';
    SaltoIncond <= '0';
    LeerMem <= '0';
    ALUop <= "01";
    EscrMem <= '0';
    FuenteALU <= '0';
    EscrReg <= '0';
    BNE <= '0';

elsif OpCode = "000101" then --Tipo BNE

    SaltoCond <= '1';
    SaltoIncond <= '0';
    LeerMem <= '0';
    ALUop <= "01";
    EscrMem <= '0';
    FuenteALU <= '0';
    EscrReg <= '0';
    BNE <= '1';

elsif OpCode = "001000" then --Tipo ADDI

    SaltoCond <= '0';
    SaltoIncond <= '0';
    LeerMem <= '0';
    ALUop <= "00";
    EscrMem <= '0';
    FuenteALU <= '1';
    EscrReg <= '1';
    RegDest <= '0';

elsif Opcode = "000010" then --Tipo J

    SaltoIncond <= '1';

else

    RegDest <= '0';
    SaltoCond <= '0';
    SaltoIncond <= '0';
    LeerMem <= '0';
    MemaReg <= '0';
    ALUop <= "00";
    EscrMem <= '0';
    FuenteALU <= '0';
    EscrReg <= '0';
    BNE <= '0';

end if;
end process;
end Control;

```

## Banco de registros

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;

entity Banco_Registros is
Port ( CLKBR : in  STD_LOGIC;           --Señal de reloj
      RegL1 : in  STD_LOGIC_VECTOR (4 downto 0);  --Dirección del Registro de lectura 1
      RegL2 : in  STD_LOGIC_VECTOR (4 downto 0);  --Dirección del Registro de lectura 2
      RegEs : in  STD_LOGIC_VECTOR (4 downto 0);  --Dirección del Registro de escritura
      DatEs : in  STD_LOGIC_VECTOR (31 downto 0); --Dato a escribir en el registro
      EscrReg : in  STD_LOGIC;                  --Habilitación de escritura
      DatL1 : out STD_LOGIC_VECTOR (31 downto 0); --Salida dato 1
      DatL2 : out STD_LOGIC_VECTOR (31 downto 0)); --Salida dato 2
end Banco_Registros;

architecture Reg_Bank of Banco_Registros is

-----Definición de los 32 registros-----
TYPE register_file IS ARRAY ( 0 TO 31 ) OF STD_LOGIC_VECTOR( 31 DOWNT0 0 );

SIGNAL register_array : register_file;

begin

      DatL1 <= register_array(to_integer(unsigned (RegL1))); --salida dato leído 1
      DatL2 <= register_array(to_integer(unsigned (RegL2))); --salida dato leído 2

-----Escritura en el registro deseado-----

Escribir: process (CLKBR, EscrReg, DatEs)

begin

if EscrReg = '1' then           --Debe estar habilitada la señal de control
if CLKBR'event and CLKBR = '1' then  --Ciclo de reloj activado por flanco ascendente

register_array(to_integer(unsigned (RegEs))) <= DatEs;--escribir en el registro

end if;
end if;

end process Escribir;

end Reg_Bank;
```

## Extensor de signo

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

entity Extension_Signo is
  Port ( EntExt : in  STD_LOGIC_VECTOR (15 downto 0); --entrada extensor signo
        SalExt : out STD_LOGIC_VECTOR (31 downto 0)); --salida extensor signo
end Extension_Signo;

architecture Ext_Sig of Extension_Signo is

begin

Signo_Ext: process(EntExt)

--definicion de variables auxiliares
variable y: STD_LOGIC;
variable z: STD_LOGIC_VECTOR(31 downto 0);

begin

  z := (others => '0');    --vector de todos ceros

  z (15 downto 0) := EntExt; --poner los 16 bits en los menos significativos

  y := z(15);             --Coger el bit 16 para ver si es un cero o un uno

-----Extension de signo-----
  if y='0' then
    z (31 downto 16) := "0000000000000000";
  else
    z (31 downto 16) := "1111111111111111";
  end if;

  SalExt <= z;

end process;

end Ext_Sig;
```

## Extensor 5 32bit

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

entity Extensor_shamt is
  Port ( Ent_SH : in  STD_LOGIC_VECTOR (4 downto 0); --entrada Ext5_32bit
        Sal_SH : out STD_LOGIC_VECTOR (31 downto 0));--salida Ext5_32bit
end Extensor_shamt;

architecture Ext5_32 of Extensor_shamt is

begin

  Ext_SH: process(Ent_SH)

variable z: STD_LOGIC_VECTOR(31 downto 0); --variable auxiliar

begin

  z := (others => '0'); --vector de todos ceros

  z (4 downto 0) := Ent_SH; --poner los 5 bits en los menos significativos

  z (31 downto 5) := "00000000000000000000000000";--añadir los 27 bits

  Sal_SH <= z;

end process;

end Ext5_32;
```



## Desplazamiento 2 izquierda

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

entity desp_izq is
  Port ( EntDesp : in  STD_LOGIC_VECTOR (31 downto 0); --entrada desplazamiento izquierda
        SalDesp : out STD_LOGIC_VECTOR (31 downto 0)); --salida desplazamiento izquierda
end desp_izq;

architecture desp_2izq of desp_izq is

begin

proceso_desplazar: process(EntDesp)

  variable z: STD_LOGIC_VECTOR(33 downto 0); --variable auxiliar

  begin

  z := (others => '0'); --poner el vector con todos los bits a cero

  z (33 downto 2):= EntDesp; --poner la entrada en los 32 bits mas significativos

  SalDesp <= z (31 downto 0);--salida los 32 bits menos significativos

end process;

end desp_2izq;
```

## Sumador 32 bits

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

entity Sumador_32bits is
  Port ( EntSum1 : in  STD_LOGIC_VECTOR (31 downto 0); --entrada 1 sumador
        EntSum2 : in  STD_LOGIC_VECTOR (31 downto 0); --entrada 2 sumador
        SalSum : out STD_LOGIC_VECTOR (31 downto 0));--salida sumador
end Sumador_32bits;

architecture Sum_normal of Sumador_32bits is

begin

SalSum <= EntSum1 + EntSum2;

end Sum_normal;
```

## Control ALU

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

entity control_alu is
  Port ( CamFun : in  STD_LOGIC_VECTOR (5 downto 0); --campo funcion de la instruccion
        ALUop : in  STD_LOGIC_VECTOR (1 downto 0); --ALUOP
        --salidas para controlar MIPS
        ContALU : out STD_LOGIC_VECTOR (2 downto 0);
        EXT_5bit32: out STD_LOGIC;
        Rotar : out STD_LOGIC);
end control_alu;

architecture C_ALU of control_alu is
begin

Cont_ALU: process (ALUop, CamFun)
begin

if ALUop="00" then ContALU <= "010";          --suma LW y SW
    Rotar <= '0';
    EXT_5bit32 <= '0';
elsif ALUop="01" then ContALU <= "110";      --resta Beq y Bne
    Rotar <= '0';
    EXT_5bit32 <= '0';
else
    --tipo R
    if CamFun="100000" then ContALU <= "010"; --add
        Rotar <= '0';
        EXT_5bit32 <= '0';
    elsif CamFun="100010" then ContALU <= "110"; --sub
        Rotar <= '0';
        EXT_5bit32 <= '0';
    elsif CamFun="100100" then ContALU <= "000"; --and
        Rotar <= '0';
        EXT_5bit32 <= '0';
    elsif CamFun="100101" then ContALU <= "001"; --or
        Rotar <= '0';
        EXT_5bit32 <= '0';
    elsif CamFun="000000" then ContALU <= "101"; --sll
        Rotar <= '1';
        EXT_5bit32 <= '1';
    elsif CamFun="000010" then ContALU <= "100"; --srl
        Rotar <= '1';
        EXT_5bit32 <= '1';
    elsif CamFun="101010" then ContALU <= "111"; --slt
        Rotar <= '0';
        EXT_5bit32 <= '0';
    else ContALU <= "011";
        Rotar <= '0';
        EXT_5bit32 <= '0';
    end if;
end if;

end process;
end C_ALU;
```

## Unidad Aritmético Lógica (ALU)

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

entity ALU is
  Port ( DatoOne : in  STD_LOGIC_VECTOR (31 downto 0); --entrada 1 ALU
        DatoTwo  : in  STD_LOGIC_VECTOR (31 downto 0); --entrada 2 ALU
        ContALU  : in  STD_LOGIC_VECTOR (2 downto 0); --entrada de control ALU
        RestALU  : out STD_LOGIC_VECTOR (31 downto 0);--salida ALU
        Zero     : out STD_LOGIC;           --salida ALU
  );
end ALU;

architecture Comp_ALU of ALU is
begin

  P_OperALU: Process (DatoOne, DatoTwo, ContALU)

  variable z: STD_LOGIC_VECTOR(63 downto 0);           --variable auxiliar
  variable y: STD_LOGIC_VECTOR(31 downto 0);         --variable auxiliar
  begin

    if ContALU="000" then RestALU <= DatoOne and DatoTwo;    --operacion AND

    -----salida de cero-----
    y := DatoOne and DatoTwo;
    if y = "00000000000000000000000000000000" then Zero<='1';
    else Zero<='0';
    end if;

    elsif ContALU="001" then RestALU <= DatoOne or DatoTwo;  --operacion OR

    -----salida de cero-----
    y := DatoOne or DatoTwo;
    if y = "00000000000000000000000000000000" then Zero<='1';
    else Zero<='0';
    end if;

    elsif ContALU="010" then RestALU <= DatoOne + DatoTwo;  --operacion suma

    -----salida de cero-----
    y := DatoOne + DatoTwo;
    if y = "00000000000000000000000000000000" then Zero<='1';
    else Zero<='0';
    end if;

    -----operacion SRL-----
    elsif ContALU="100" then

      -- Inicialización de datos en cero.
      z := (others => '0');

      -- Poner el dato1 en los primeros 32bits
      z(31 downto 0):= DatoOne;
```

– Dependiendo del dato introducido se cogen los 32bits deseados para la rotacion  
 case DatoTwo is

```

when "00000000000000000000000000000000" => y := z(31 downto 0);
when "00000000000000000000000000000001" => y := z(32 downto 1);
when "00000000000000000000000000000010" => y := z(33 downto 2);
when "00000000000000000000000000000011" => y := z(34 downto 3);
when "00000000000000000000000000000100" => y := z(35 downto 4);
when "00000000000000000000000000000101" => y := z(36 downto 5);
when "00000000000000000000000000000110" => y := z(37 downto 6);
when "00000000000000000000000000000111" => y := z(38 downto 7);
when "00000000000000000000000000001000" => y := z(39 downto 8);
when "00000000000000000000000000001001" => y := z(40 downto 9);
when "00000000000000000000000000001010" => y := z(41 downto 10);
when "00000000000000000000000000001011" => y := z(42 downto 11);
when "00000000000000000000000000001100" => y := z(43 downto 12);
when "00000000000000000000000000001101" => y := z(44 downto 13);
when "00000000000000000000000000001110" => y := z(45 downto 14);
when "00000000000000000000000000001111" => y := z(46 downto 15);
when "0000000000000000000000000010000" => y := z(47 downto 16);
when "0000000000000000000000000010001" => y := z(48 downto 17);
when "0000000000000000000000000010010" => y := z(49 downto 18);
when "0000000000000000000000000010011" => y := z(50 downto 19);
when "0000000000000000000000000010100" => y := z(51 downto 20);
when "0000000000000000000000000010101" => y := z(52 downto 21);
when "0000000000000000000000000010110" => y := z(53 downto 22);
when "0000000000000000000000000010111" => y := z(54 downto 23);
when "0000000000000000000000000011000" => y := z(55 downto 24);
when "0000000000000000000000000011001" => y := z(56 downto 25);
when "0000000000000000000000000011010" => y := z(57 downto 26);
when "0000000000000000000000000011011" => y := z(58 downto 27);
when "0000000000000000000000000011100" => y := z(59 downto 28);
when "0000000000000000000000000011101" => y := z(60 downto 29);
when "0000000000000000000000000011110" => y := z(61 downto 30);
when "0000000000000000000000000011111" => y := z(62 downto 31);
when others => y := DatoOne;

```

end case;

RestALU <= y;

```

-----salida de cero-----
if y="00000000000000000000000000000000" then Zero<='1';
else Zero<='0';
end if;

```

-----fin operacion SRL-----

-----operacion SLL-----

elsif ContALU="101" then

– Inicialización de datos en cero.

```
z := (others => '0');
```

– Dependiendo del dato introducido se coloca el dato1 en el vector z y se cogen siempre los primeros 32bits

case DatoTwo is

```

when "00000000000000000000000000000000" => z(31 downto 0):= DatoOne;
y := z(31 downto 0);

```





## Memoria de datos

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;

entity Memoria_Datos is

generic ( WIDTH : Natural := 32;
         DEPTH : Natural := 10);

Port ( d : in STD_LOGIC_VECTOR (WIDTH - 1 downto 0); --Entrada memoria
      a : in STD_LOGIC_VECTOR (DEPTH - 1 downto 0); --Número de datos almacenados a la
potencia de dos
      we : in STD_LOGIC; --Habilitacion escritura
      lect : in STD_LOGIC; --Habilitacion lectura
      clk_MD : in STD_LOGIC; --Reloj
      q : out STD_LOGIC_VECTOR (WIDTH - 1 downto 0)); --Salida memoria
end Memoria_Datos;

architecture MDAT of Memoria_Datos is

-----definición de la memoria-----
type mem_array is array (0 to 2**DEPTH-1) of std_logic_vector(WIDTH-1 downto 0);

signal RAM : mem_array;

begin

synch_RAM : process (clk_MD) is

begin

if falling_edge (clk_MD) then --flanco descendente de la señal de reloj
q <= RAM(to_integer(unsigned (a))); --salida sincrona

if we = '1' then
RAM(to_integer( unsigned (a))) <= d;--entrada sincrona con habilitacion
end if;
end if;

end process synch_RAM;

end architecture MDAT;
```

## Conector 28 32bit

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

entity Conexion28_32 is
  Port ( Ent28 : in  STD_LOGIC_VECTOR (27 downto 0);
        Ent4  : in  STD_LOGIC_VECTOR (3  downto 0);
        Sal32 : out STD_LOGIC_VECTOR (31 downto 0));
end Conexion28_32;

architecture Conect28_32 of Conexion28_32 is

begin

  Conectar: process (Ent28, Ent4)

  variable z: STD_LOGIC_VECTOR(31 downto 0);

  begin

    z := (others => '0');

    z(27 downto 0):= Ent28;

    z(31 downto 28):= Ent4;

    Sal32 <= z;

  end process;

end Conect28_32;
```



## Multiplexores 551 y 552

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

entity Multiplexor5_5 is
  Port ( EntMux551 : in  STD_LOGIC_VECTOR (4 downto 0);
        EntMux552 : in  STD_LOGIC_VECTOR (4 downto 0);
        ContMux55 : in  STD_LOGIC;
        SalMux55  : out STD_LOGIC_VECTOR (4 downto 0));
end Multiplexor5_5;

architecture mux5_5 of Multiplexor5_5 is

begin

MUX55: process (EntMux551, EntMux552, ContMux55)
begin
if ContMux55 = '0' then SalMux55<=EntMux551;
else SalMux55<=EntMux552;
end if;
end process;

end mux5_5;
```

## Multiplexor 432

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

entity Multiplexor32_4 is
  Port ( EntMux32_4_1 : in  STD_LOGIC_VECTOR (31 downto 0);
        EntMux32_4_2 : in  STD_LOGIC_VECTOR (31 downto 0);
        ContMux32_4  : in  STD_LOGIC;
        SalMux32_4   : out STD_LOGIC_VECTOR (31 downto 0));
end Multiplexor32_4;

architecture Mux32_4 of Multiplexor32_4 is

begin

MUX324: process (EntMux32_4_1, EntMux32_4_2, ContMux32_4)

begin

case ContMux32_4 is
  when '0' => SalMux32_4 <= EntMux32_4_1;
  when others => SalMux32_4 <= EntMux32_4_2;
end case;
end process;
end Mux32_4;
```

## Multiplexor Entrada ALU

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

entity Mux_EntALU is
  Port ( Ent1_MEA : in  STD_LOGIC_VECTOR (31 downto 0);
        Ent2_MEA : in  STD_LOGIC_VECTOR (31 downto 0);
        Cont_MEA : in  STD_LOGIC;
        Sal_MEA : out STD_LOGIC_VECTOR (31 downto 0));
end Mux_EntALU;

architecture M_EA of Mux_EntALU is

begin

Process (Ent1_MEA, Ent2_MEA, Cont_MEA)

begin

if Cont_MEA = '0' then Sal_MEA <= Ent1_MEA;
else Sal_MEA <= Ent2_MEA;

end if;
end process;
end M_EA;
```

## Multiplexores 32B1, 32B2 y 32B3

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

entity Multiplexor32bits is
  Port ( EntMux32_1 : in  STD_LOGIC_VECTOR (31 downto 0);
        EntMux32_2 : in  STD_LOGIC_VECTOR (31 downto 0);
        ContMux32 : in  STD_LOGIC;
        SalMux32 : out STD_LOGIC_VECTOR (31 downto 0));
end Multiplexor32bits;

architecture Mux32B of Multiplexor32bits is
begin

MUX32: process (EntMux32_1, EntMux32_2, ContMux32)
begin

if ContMux32 = '0' then SalMux32 <= EntMux32_1;
else SalMux32 <= EntMux32_2;

end if;
end process;
end Mux32B;
```

## Multiplexor Control

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

entity Multiplexor1bit is
  Port ( EntMux1B_1 : in STD_LOGIC;
        EntMux1B_2 : in STD_LOGIC;
        ContMux1B : in STD_LOGIC;
        SalMux1B : out STD_LOGIC);
end Multiplexor1bit;

architecture Mux_1B of Multiplexor1bit is

begin

MUX1B: process (EntMux1B_1, EntMux1B_2, ContMux1B)

begin

if ContMux1B = '0' then SalMux1B <= EntMux1B_1;
else SalMux1b <= EntMux1B_2;
end if;

end process;

end Mux_1B;
```

## Fichero modulo alto nivel (Conexiones)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;

entity MIPS is
  Port ( clk : in STD_LOGIC;           --reloj del MIPS
        -----multiplexores para comprobar funcionamiento-----
        Ent_cont1: in STD_LOGIC_VECTOR(1 downto 0);
        Ent_cont2: in STD_LOGIC_VECTOR(3 downto 0);
        Sal_MUX: out STD_LOGIC_VECTOR(7 downto 0));
end MIPS;

architecture microprocesador of MIPS is

-----DECLARACION DE COMPONENTES-----
COMPONENT Reg_PC

  Port (Ent_PC : in STD_LOGIC_VECTOR (31 downto 0);
        clk_PC : in STD_LOGIC;
        Sal_PC : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

COMPONENT Sumador_Mas4

  Port ( EntPC : in STD_LOGIC_VECTOR (31 downto 0);
        Sal4 : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

COMPONENT Memoria_Instrucciones

  Port ( clk_MI : in STD_LOGIC;
        en : in STD_LOGIC:='1';
        Dir : in STD_LOGIC_VECTOR (9 downto 0);
        Datos : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

COMPONENT Banco_Registros

  Port ( CLKBR : in STD_LOGIC;
        RegL1 : in STD_LOGIC_VECTOR (4 downto 0);
        RegL2 : in STD_LOGIC_VECTOR (4 downto 0);
        RegEs : in STD_LOGIC_VECTOR (4 downto 0);
        DatEs : in STD_LOGIC_VECTOR (31 downto 0);
        EscrReg : in STD_LOGIC;
        DatL1 : out STD_LOGIC_VECTOR (31 downto 0);
        DatL2 : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;
```

```

COMPONENT Extension_Signo

  Port ( EntExt : in STD_LOGIC_VECTOR (15 downto 0);
        SalExt : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

COMPONENT Extensor_shamt

  Port ( Ent_SH : in STD_LOGIC_VECTOR (4 downto 0);
        Sal_SH : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

COMPONENT Control_MIP

  Port ( OpCode : in STD_LOGIC_VECTOR (5 downto 0);
        RegDest : out STD_LOGIC;
        SaltoCond : out STD_LOGIC;
        SaltoIncond : out STD_LOGIC;
        LeerMem : out STD_LOGIC;
        MemaReg : out STD_LOGIC;
        ALUop : out STD_LOGIC_VECTOR (1 downto 0);
        EscrMem : out STD_LOGIC;
        FuenteALU : out STD_LOGIC;
        BNE : out STD_LOGIC;
        EscrReg : out STD_LOGIC);

end COMPONENT;

COMPONENT Desp2_izqJ

  Port ( Ent_Despl : in STD_LOGIC_VECTOR (25 downto 0);
        Sal_Despl : out STD_LOGIC_VECTOR (27 downto 0));

end COMPONENT;

COMPONENT desp_izq

  Port ( EntDespl : in STD_LOGIC_VECTOR (31 downto 0);
        SalDespl : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

COMPONENT Sumador_32bits

  Port ( EntSum1 : in STD_LOGIC_VECTOR (31 downto 0);
        EntSum2 : in STD_LOGIC_VECTOR (31 downto 0);
        SalSum : out STD_LOGIC_VECTOR (31 downto 0));

end COMPONENT;

```

COMPONENT control\_alu

```
Port ( CamFun : in STD_LOGIC_VECTOR (5 downto 0);
      ALUop : in STD_LOGIC_VECTOR (1 downto 0);
      ContALU : out STD_LOGIC_VECTOR (2 downto 0);
      EXT_5bit32: out STD_LOGIC;
      Rotar : out STD_LOGIC);
```

end COMPONENT;

COMPONENT ALU

```
Port ( DatoOne : in STD_LOGIC_VECTOR (31 downto 0);
      DatoTwo : in STD_LOGIC_VECTOR (31 downto 0);
      ContALU : in STD_LOGIC_VECTOR (2 downto 0);
      RestALU : out STD_LOGIC_VECTOR (31 downto 0);
      Zero : out STD_LOGIC);
```

end COMPONENT;

COMPONENT Memoria\_Datos

```
generic ( WIDTH : Natural := 32;
         DEPTH : Natural := 10);
```

```
Port ( d : in STD_LOGIC_VECTOR (WIDTH - 1 downto 0);
      a : in STD_LOGIC_VECTOR (DEPTH - 1 downto 0);
      we : in STD_LOGIC;
      lect : in STD_LOGIC;
      clk_MD : in STD_LOGIC;
      q : out STD_LOGIC_VECTOR (WIDTH - 1 downto 0));
```

end COMPONENT;

COMPONENT Multiplexor5\_5

```
Port ( EntMux551 : in STD_LOGIC_VECTOR (4 downto 0);
      EntMux552 : in STD_LOGIC_VECTOR (4 downto 0);
      ContMux55 : in STD_LOGIC;
      SalMux55 : out STD_LOGIC_VECTOR (4 downto 0));
```

end COMPONENT;

COMPONENT Multiplexor32\_4 is

```
Port ( EntMux32_4_1 : in STD_LOGIC_VECTOR (31 downto 0);
      EntMux32_4_2 : in STD_LOGIC_VECTOR (31 downto 0);
      ContMux32_4 : in STD_LOGIC;
      SalMux32_4 : out STD_LOGIC_VECTOR (31 downto 0));
```

end COMPONENT;

COMPONENT Multiplexor32bits is

```
Port ( EntMux32_1 : in STD_LOGIC_VECTOR (31 downto 0);
      EntMux32_2 : in STD_LOGIC_VECTOR (31 downto 0);
      ContMux32 : in STD_LOGIC;
      SalMux32 : out STD_LOGIC_VECTOR (31 downto 0));
```

end COMPONENT;

COMPONENT Multiplexor1bit is

```
Port ( EntMux1B_1 : in STD_LOGIC;
      EntMux1B_2 : in STD_LOGIC;
      ContMux1B : in STD_LOGIC;
      SalMux1B : out STD_LOGIC);
```

end COMPONENT;

COMPONENT Conexion28\_32 is

```
Port ( Ent28 : in STD_LOGIC_VECTOR (27 downto 0);
      Ent4 : in STD_LOGIC_VECTOR (3 downto 0);
      Sal32 : out STD_LOGIC_VECTOR (31 downto 0));
```

end COMPONENT;

COMPONENT Mux\_EntALU is

```
Port ( Ent1_MEA : in STD_LOGIC_VECTOR (31 downto 0);
      Ent2_MEA : in STD_LOGIC_VECTOR (31 downto 0);
      Cont_MEA : in STD_LOGIC;
      Sal_MEA : out STD_LOGIC_VECTOR (31 downto 0));
```

end COMPONENT;

-----DECLARACION DE SEÑALES-----

signal s1, s2, s3, s4, s5, s6, s7, s10, s11, s12, s13, s14, s141, s15, s16, s17, s33:  
STD\_LOGIC\_VECTOR (31 downto 0);

--s1:salida PC      s2:salida Sum+4      s3:salida sumador      s4:salida MUX32B2  
--s5:salida MUX32B3      s6:salida MemIns      s7:union PC+4+Desp2IzqJ      s10:salida Ext\_Sig  
--s11:salida Ext5\_32bit      s12:salida BR1      s13:salida BR2      s14:salida MUX432  
--s141:salida MUXEntALU      s15:salida RestALU      s16:salida MemDatos      s17:salida MUX32B1  
--s33:salida Desp2Izq

signal s7: STD\_LOGIC\_VECTOR (27 downto 0); --salida Desp2IzqJ

signal s8, s9: STD\_LOGIC\_VECTOR (4 downto 0);--s8:salida MUX551      s9:salida MUX552

signal s18, s19, s20, s21, s22, s23, s25, s26, s27, s28, s29, s30, s31, s32, s35: STD\_LOGIC;

--s18:Rotar      s19:RegDest      s20:saltoIncondicional      s21:saltoCondicional  
--s22:LeerMem      s23:MemaReg      s25:EschrMem      s26:FuenteALU  
--s27:EschrReg      s28:BNE/BEQ      s29:Zero      s30:Zeronegado  
--s31:salidaMUXSaltCond      s32:salida AND      s35:Ent\_ALU

signal s24: STD\_LOGIC\_VECTOR (1 downto 0); --ALU\_OP

signal s34: STD\_LOGIC\_VECTOR (2 downto 0); --ControlALU

-----señales para ver salidas en la tarjeta BASYS 2-----  
 signal sm1, sm2, sm3, sm4, sm5, sm6, sm7, sm8, sm9, sm10, sm11, sm12, sm13, sm14, sm15,  
 sm16: STD\_LOGIC\_VECTOR (7 downto 0);

-----LUGAR ARQUITECTURA COMPONENTES-----

FOR PC: Reg\_PC USE ENTITY work.Reg\_PC(Registro\_Contador); --Registro Contador  
 FOR Sum4: Sumador\_Mas4 USE ENTITY work.Sumador\_Mas4(Sum\_Mas4); --Sumador  
 instrucción siguiente  
 FOR MI: Memoria\_Instrucciones USE ENTITY work.Memoria\_Instrucciones(M\_Ins); --Memoria de  
 instrucciones  
 FOR D2IJ: Desp2\_izqJ USE ENTITY work.Desp2\_izqJ(DesplazamientoJ); --Desplaza 2 izquierda  
 instrucción J  
 FOR Cntrl: Control\_MIP USE ENTITY work.Control\_MIP(Control); --Control del MIPS  
 FOR BR: Banco\_Registros USE ENTITY work.Banco\_Registros(Reg\_Bank); --Banco de Registros  
 FOR ExSg: Extension\_Signo USE ENTITY work.Extension\_Signo(Ext\_Sig); --Extension signo 16-  
 32  
 FOR Ex532: Extensor\_shamt USE ENTITY work.Extensor\_shamt(Ext5\_32); --Extension shamt 5-  
 32  
 FOR Dlzq: desp\_izq USE ENTITY work.desp\_izq(desp\_2izq); --Desplazamiento 2  
 izquierda  
 FOR Sum32: Sumador\_32bits USE ENTITY work.Sumador\_32bits(Sum\_normal); --Sumador 32  
 bits  
 FOR CntrlALU: control\_alu USE ENTITY work.control\_alu(C\_ALU); --Control ALU  
 FOR MD: Memoria\_Datos USE ENTITY work.Memoria\_Datos(MDAT); --Memoria de datos  
 FOR MUX551: Multiplexor5\_5 USE ENTITY work.Multiplexor5\_5(mux5\_5); --Multiplexor 5 bits  
 entrada y salida  
 FOR MUX552: Multiplexor5\_5 USE ENTITY work.Multiplexor5\_5(mux5\_5); --Multiplexor 5 bits  
 entrada y salida  
 FOR MUX432: Multiplexor32\_4 USE ENTITY work.Multiplexor32\_4(mux32\_4); --Multiplexor 32  
 bits entrada y salida con 4 entradas  
 FOR MUX32B1: Multiplexor32bits USE ENTITY work.Multiplexor32bits(Mux32B); --Multiplexor 32  
 bits entrada y salida  
 FOR MUX32B2: Multiplexor32bits USE ENTITY work.Multiplexor32bits(Mux32B); --Multiplexor 32  
 bits entrada y salida  
 FOR MUX32B3: Multiplexor32bits USE ENTITY work.Multiplexor32bits(Mux32B); --Multiplexor 32  
 bits entrada y salida  
 FOR MUXCONTROL: Multiplexor1bit USE ENTITY work.Multiplexor1bit(Mux\_1B); --Multiplexor  
 control de salto BEQ/BNE  
 FOR CONECT2832: Conexion28\_32 USE ENTITY work.Conexion28\_32(Conect28\_32); --Aumento a  
 32 bits despues del desplazamiento izquierda  
 FOR MUXENTALU: Mux\_EntALU USE ENTITY work.Mux\_EntALU(M\_EA); --Multiplexor  
 entrada ALU  
 FOR OTHERS: ALU USE ENTITY work.ALU(Comp\_ALU); --ALU



```
begin
```

```
-----conexiones según el esquema hardware-----
```

```
s30<= not(s29);
```

```
s32<= s21 and s31;
```

```
PC: REG_PC port map (s5, clk, s1);
```

```
Sum4: Sumador_Mas4 port map (s1, s2);
```

```
MI: Memoria_Instrucciones port map (clk, '1', s5(9 downto 0), s6);
```

```
D2IJ: Desp2_izqJ port map (s6(25 downto 0), s7);
```

```
Cntrl: Control_MIP port map (s6(31 downto 26), s19, s21, s20, s22, s23, s24, s25, s26, s28, s27);
```

```
BR: Banco_Registros port map (clk, s8, s6(20 downto 16), s9, s17, s27, s12, s13);
```

```
ExSg: Extension_Signo port map (s6(15 downto 0), s10);
```

```
Ex532: Extensor_shamt port map (s6 (10 downto 6), s11);
```

```
Dlzq: desp_izq port map (s10, s33);
```

```
Sum32: Sumador_32bits port map (s2, s33, s3);
```

```
CntrlALU: control_alu port map (s6(5 downto 0), s24, s34, s35, s18);
```

```
MD: Memoria_Datos generic map (WIDTH => 32, DEPTH => 10) port map (s13, s15(9 downto 0), s25, s22, clk, s16);
```

```
MUX551: Multiplexor5_5 port map (s6(25 downto 21), s6(20 downto 16), s18, s8);
```

```
MUX552: Multiplexor5_5 port map (s6(20 downto 16), s6(15 downto 11), s19, s9);
```

```
MUX432: Multiplexor32_4 port map (s13, s10, s26, s14);
```

```
MUXENTALU: Mux_EntALU port map (s14, s11, s35, s141);
```

```
Mux32B1: Multiplexor32bits port map (s15, s16, s23, s17);
```

```
Mux32B2: Multiplexor32bits port map (s2, s3, s32, s4);
```

```
Mux32B3: Multiplexor32bits port map (s4, s71, s20, s5);
```

```
MUXCONTROL: Multiplexor1bit port map (s29, s30, s28, s31);
```

```
CONECT2832: Conexion28_32 port map (s7, s2(31 downto 28), s71);
```

```
COMP_ALU: ALU port map (s12, s141, s34, s15, s29);
```

```
-----diseño multiplexores para visualizar las señales del MIPS-----
```

```
Process (clk, Ent_cont1, Ent_cont2)
```

```
begin
```

```
case Ent_cont1 is
```

```
when "00" => sm1 <= s6(7 downto 0);
```

```
when "01" => sm1 <= s6(15 downto 8);
```

```
when "10" => sm1 <= s6(23 downto 16);
```

```
when others => sm1 <= s6(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm2 <= s12(7 downto 0);
```

```
when "01" => sm2 <= s12(15 downto 8);
```

```
when "10" => sm2 <= s12(23 downto 16);
```

```
when others => sm2 <= s12(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is  
  
when "00" => sm3 <= s141(7 downto 0);  
when "01" => sm3 <= s141(15 downto 8);  
when "10" => sm3 <= s141(23 downto 16);  
when others => sm3 <= s141(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm4 <= s17(7 downto 0);  
when "01" => sm4 <= s17(15 downto 8);  
when "10" => sm4 <= s17(23 downto 16);  
when others => sm4 <= s17(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm5 <= s15(7 downto 0);  
when "01" => sm5 <= s15(15 downto 8);  
when "10" => sm5 <= s15(23 downto 16);  
when others => sm5 <= s15(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm6 <= s13(7 downto 0);  
when "01" => sm6 <= s13(15 downto 8);  
when "10" => sm6 <= s13(23 downto 16);  
when others => sm6 <= s13(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm7 <= s14(7 downto 0);  
when "01" => sm7 <= s14(15 downto 8);  
when "10" => sm7 <= s14(23 downto 16);  
when others => sm7 <= s14(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm8 <= s16(7 downto 0);  
when "01" => sm8 <= s16(15 downto 8);  
when "10" => sm8 <= s16(23 downto 16);  
when others => sm8 <= s16(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm9 <= s2(7 downto 0);  
when "01" => sm9 <= s2(15 downto 8);  
when "10" => sm9 <= s2(23 downto 16);  
when others => sm9 <= s2(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm10 <= s71(7 downto 0);  
when "01" => sm10 <= s71(15 downto 8);  
when "10" => sm10 <= s71(23 downto 16);  
when others => sm10 <= s71(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm11 <= s1(7 downto 0);  
when "01" => sm11 <= s1(15 downto 8);  
when "10" => sm11 <= s1(23 downto 16);  
when others => sm11 <= s1(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm12 <= s10(7 downto 0);  
when "01" => sm12 <= s10(15 downto 8);  
when "10" => sm12 <= s10(23 downto 16);  
when others => sm12 <= s10(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm13 <= s4(7 downto 0);  
when "01" => sm13 <= s4(15 downto 8);  
when "10" => sm13 <= s4(23 downto 16);  
when others => sm13 <= s4(31 downto 24);
```

```
end case;
```

```
case Ent_cont1 is
```

```
when "00" => sm14 <= s5(7 downto 0);  
when "01" => sm14 <= s5(15 downto 8);  
when "10" => sm14 <= s5(23 downto 16);  
when others => sm14 <= s5(31 downto 24);
```

```
end case;
```

```

case Ent_cont1 is

when "00" => sm15(0) <= s18;
    sm15(1) <= s19;
    sm15(2) <= s27;
    sm15(3) <= s26;
    sm15(4) <= s35;
    sm15(6 downto 5) <= s24;
    sm15(7) <= s28;

when "01" => sm15(2 downto 0) <= s34;
    sm15(3) <= s22;
    sm15(4) <= s25;
    sm15(5) <= s23;
    sm15(6) <= s21;
    sm15(7) <= s32;

when "10" => sm15(0) <= s20;
    sm15(2 downto 1) <= "00";
    sm15(7 downto 3) <= s8;

when others => sm15(2 downto 0) <= "000";
    sm15(7 downto 3) <= s9;

end case;

```

```

case Ent_cont1 is

when "00" => sm16 <= s3(7 downto 0);
when "01" => sm16 <= s3(15 downto 8);
when "10" => sm16 <= s3(23 downto 16);
when others => sm16 <= s3(31 downto 24);

end case;

```

```

case Ent_cont2 is

when "0000" => Sal_MUX <= sm1;
when "0001" => Sal_MUX <= sm2;
when "0010" => Sal_MUX <= sm3;
when "0011" => Sal_MUX <= sm4;
when "0100" => Sal_MUX <= sm5;
when "0101" => Sal_MUX <= sm6;
when "0110" => Sal_MUX <= sm7;
when "0111" => Sal_MUX <= sm8;
when "1000" => Sal_MUX <= sm9;
when "1001" => Sal_MUX <= sm10;
when "1010" => Sal_MUX <= sm11;
when "1011" => Sal_MUX <= sm12;
when "1100" => Sal_MUX <= sm13;
when "1101" => Sal_MUX <= sm14;
when "1110" => Sal_MUX <= sm15;
when others => Sal_MUX <= sm16;

end case;

```

```

end process;
end microprocesador;

```