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 segmentado basado en MIPS

Autor:

Gómez Hernández, Jonatan

Tutor:

Cáceres Gómez, Santiago

Tecnología Electrónica

Valladolid, julio de 2017.

Este trabajo es el resultado del apoyo incondicional que he recibido en todo momento por parte de mi familia, mi pareja y mis amigos. Sin ellos no habría sido posible superar los momentos de flaqueza en los que la imposibilidad de encontrar las soluciones acertadas y poder avanzar te hacían querer tirar la toalla.

Mención especial para mi tutor, Santiago Cáceres Gómez, el cual me ayudó desde el primer día y me puso todas las facilidades posibles, estando siempre disponible para mí. Además de un trabajo impecable como profesional, ha sabido guiarme también fuera de ese ámbito y me ha mostrado su total confianza. Nunca dudó de que pudiera realizar esta tarea de la manera que él creía adecuada. Espero haber estado a la altura.

Agradecer a mis compañeros de grado, que me han aconsejado siempre que se lo he pedido y me han prestado toda la ayuda que estaba en su mano sin esperar nada a cambio.

Por último agradecer a todos los profesores que me han tenido como alumno en estos años, porque de una manera u otra me han ayudado a llegar a dónde hoy me encuentro.

Por todo esto este trabajo llevará mi nombre, pero tiene el de todos ellos, y sin la suma de cada uno no se podría haber llevado a cabo.

GRACIAS.

RESUMEN

Este Trabajo Fin de Grado consiste en la implementación en una placa Basys 3 de un Microprocesador de tipo MIPS segmentado o *pipeline*, el cual está programado en VHDL utilizando la herramienta Vivado Design, del fabricante Xilinx.

Es segmentado siguiendo la evolución natural que han tenido este tipo de aparatos, pasando de poder ejecutar una única instrucción por ciclo de reloj a varias en el mismo tiempo, es decir, poder dividir su potencial o capacidad en varias etapas que irán ejecutando partes de la instrucción global simultáneamente, consiguiendo así que se realice cada instrucción de una manera mucho más rápida.

En el presente trabajo se explicará detalladamente el proceso de diseño y creación del microprocesador segmentado, así como las comprobaciones pertinentes para cerciorarnos que, una vez finalizado, funciona de manera correcta para cualquier combinación de instrucciones pertenecientes a su repertorio.

PALABRAS CLAVE

Microprocesador, MIPS, Segmentación, FPGA, VHDL.

ABSTRACT

This Final Degree Project consists in the implementation of a segmented MIPS microprocessor or pipeline in a Basys 3 board, which is programmed in VHDL, using the Vivado Design tool, from Xilinx fabricant.

It is segmented following the natural development that this sort of devices have had: going from executing just one single instruction per clock cycle to several instructions in the same time, what means, being able to divide its capacity in several stages which will execute part of the global task simultaneously, in order to achieve the result of each instruction in a much more fast way.

In the current paperwork, the design process and the creation of the segmented microprocessor as well as the relevant checks to probe that, once it is completed, will work properly to any combination of instructions within its repertoire, will be detailed explained.

KEYWORDS

Microprocessor, MIPS, Pipeline, FPGA, VHDL.

Índice de contenidos

Índice de Ilustraciones

Índice de tablas

Capítulo 1. Introducción

1.1. Justificación y relevancia

"La integración de microprocesadores en FPGAs es una estrategia muy potente en el mundo de la electrónica digital. Siendo posible integrar sistemas mixtos en los que se combinen procesador y lógica convencional, agilizando las tareas y ahorrando espacio." (Álvaro Padierna Díaz, 2009)

"MIPS es una arquitectura diseñada para optimizar la segmentación en unidades de control y para facilitar la generación automática de código máquina por parte de los compiladores, por ese motivo es la más extendida comercialmente por los fabricantes."(«Arquitectura MIPS», 2016)

Eligiendo un modelo que trabaje de manera segmentada ganaremos notablemente en rendimiento ya que el ciclo de reloj de funcionamiento se ve reducido en una relación cercana al número de etapas o segmentos en que se divida.

1.2. Contexto

La realización del trabajo se ha llevado a cabo con una placa Basys 3. Esta elección fue tomada por el tutor, por lo que no fue tarea mía decidir si esta placa cumplía mejor o peor las condiciones para realizar el trabajo requerido. Como la Basys 3 tiene arquitectura Artix-7 va asociada a Xilinx y, por tanto, a su herramienta principal de diseño, Vivado. Es decir, que a la hora de comenzar el trabajo ya era conocida la herramienta principal a utilizar. La única decisión que había que tomar en este sentido era la elección del lenguaje de programación. Este aspecto se decidió rápidamente ya que la propuesta de mi tutor coincidía con mi preferencia, programar en VHDL. A ambos nos parecía un lenguaje con más posibilidades que Verilog, que era la otra opción posible. Además en mi caso, VHDL me era más familiar, ya que es un lenguaje con el que ya había trabajado y con el cual me siento cómodo programando.

1.3. Aprendizaje previo

Los conocimientos relacionados pero adquiridos previamente a la realización de este trabajo han sido varios. En primer lugar, estaría la base de programación en VHDL que adquirí en la asignatura Métodos y Herramientas de Diseño Electrónico, gracias a mi profesor José Manuel Mena Rodríguez. Como paso siguiente aparecería la asignatura de Electrónica Digital, impartida por mi tutor en este trabajo, Santiago Cáceres Gómez, en la cual aprendí el funcionamiento del MIPS monociclo que servirá como base para este trabajo. Y final y paralelamente con la anterior cursé Sistemas Electrónicos Reconfigurables, impartida por Francisco José de Andrés Rodríguez Trelles, en cuyos laboratorios y clases teóricas se impartían conocimientos de programación en VHDL, aplicados a los dispositivos lógicos programables, por lo que me aportó una gran ayuda a la hora de mejorar mi programación con este lenguaje.

Este trabajo aparece como la continuación de uno realizado anteriormente consistente en la implementación en una placa Basys 2 del MIPS monociclo, por lo que siguiendo la evolución lógica el siguiente paso era implementar el MIPS segmentado en una FPGA (en este caso el modelo siguiente al del anterior trabajo). Aunque en este trabajo se hubieran encontrado la totalidad de conocimientos e información para realizar la primera parte del presente trabajo se ha decidido no consultarlo ni tomarlo como referencia en ningún momento para que el aprendizaje base fuera propio y la línea de desarrollo no estuviera tan marcada. De todas formas, este trabajo ha ayudado a mi tutor, Santiago Cáceres, a impartir de una manera más clara y precisa los conocimientos relacionados con este tipo de microprocesador en la asignatura Electrónica Digital, la cual he tomado como referencia en todo momento, y también le ha servido para asesorarme a mí de una manera más precisa a la hora de plantear y desarrollar mi trabajo, por lo que la aportación realizada por Jonatan Martín Gutiérrez y su trabajo ha estado muy presente.

1.4. Objetivos

Se pasan a explicar detalladamente los objetivos básicos que se han marcado para este Trabajo Fin de Grado, los cuales se analizarán una vez finalizado el mismo en el capítulo 6.

1.4.1 Objetivos Generales

- **Diseñar y programar el MIPS segmentado (*pipeline*)**: este es el objetivo final y general del trabajo. Se basa en construir un microprocesador segmentado con una estructura de tipo MIPS (Microprocessor without Interlocked Pipeline Stages, es decir, microprocesador sin bloqueos en las etapas de segmentación) en lenguaje VHDL e implementarlo en una placa Basys 3 con la estructura FPGA basada en Artix-7. Este procesador debería ser capaz de ejecutar todas las instrucciones para las que ha sido diseñado. Se engloba dentro del tipo RISC ya que tiene un repertorio de instrucciones reducido.

1.4.2. Objetivos Específicos

- **Programar en VHDL**: VHDL es un lenguaje definido por el IEEE usado por ingenieros y científicos para describir circuitos digitales o modelar fenómenos científicos respectivamente. El objetivo es familiarizarse poco a poco con dicho lenguaje, comenzando con circuitos sencillos para más tarde poder conseguir el nivel de programación que se requiere para el trabajo.

- **Realizar diseños digitales orientados a FPGAs**: se buscará ser capaz de diseñar diferentes tipos de circuitos digitales de tal manera que estos sean puedan ser implementados de manera correcta en estructuras FPGAs. Todos los elementos y sus conexiones tendrán que ser pensados para que su concordancia con este tipo de estructura sea máxima.

- **Diseñar y programar el MIPS monociclo**: Este objetivo será un paso previo del general, enfocado al aprendizaje y a la creación de elementos básicos. Las condiciones serán las mismas salvo que el microprocesador que se implementará en la placa Basys 3 será de tipo monociclo y no segmentado. Las instrucciones deberán ejecutarse a lo largo de un ciclo de reloj, habiendo solo una instrucción dentro del circuito simultáneamente.

- **Diseñar y programar módulos funcionales independientes**: se buscará diseñar y programar diferentes módulos o conjuntos de elementos. El objetivo consistirá en que realicen tareas específicas independientes. Podrán ser aprovechados individualmente o formar parte de un

esquema superior. En este sentido mediante la unión progresiva de estas estructuras se intentará lograr el objetivo general explicado en la página anterior, por lo que cuando se diseñen estos módulos se hará de tal manera que las tareas que vayan a realizar puedan ser parte en un esquema general del MIPS segmentado.

Capítulo 2. Metodología

2.1. Pasos a seguir

A continuación se muestra los pasos a seguir, tanto en forma de diagrama como numerada. Esta será la metodología que se seguirá para intentar cumplir de manera correcta y eficiente el objetivo final de este trabajo, aunque también será aplicable a algún objetivo específico:

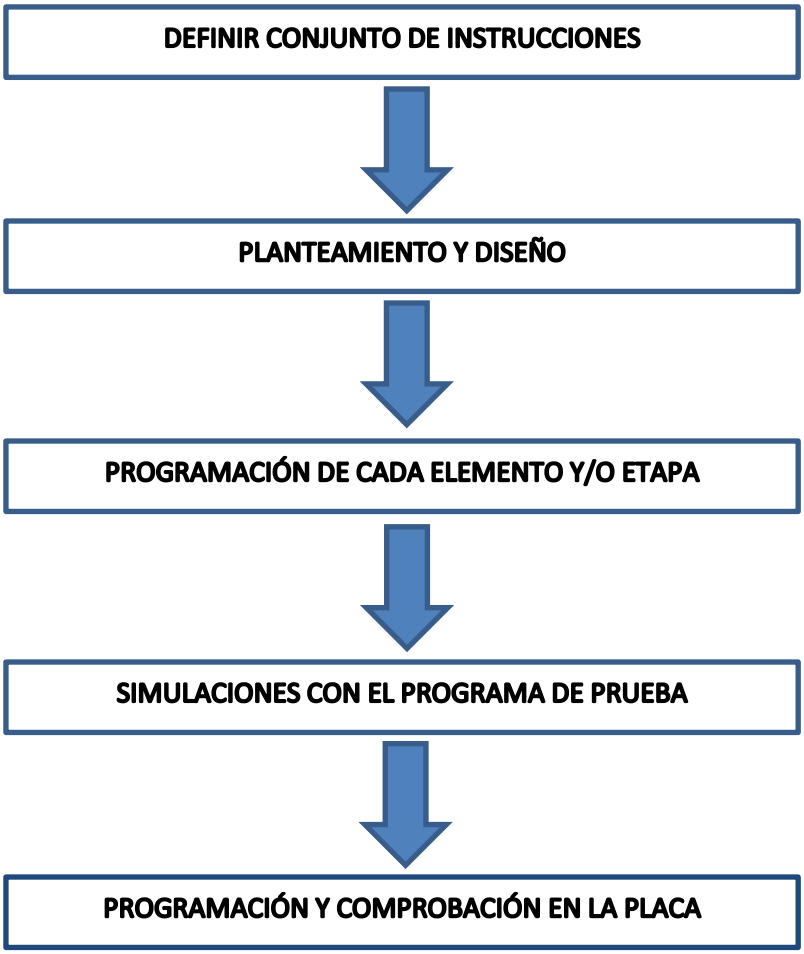

Ilustración 1: Diagrama de la metodología

1. **Elección de las diferentes instrucciones que vamos a implementar**. Es necesario pensar detenidamente qué acciones queremos que haga nuestro procesador antes de empezar a realizar código. Las tareas que sea capaz de llevar a cabo vienen marcadas por las diferentes instrucciones para las que se cree, ya que software y hardware se diseñarán de diferentes maneras según el tipo de instrucciones que seamos capaces de ejecutar.

2. **Esquema y diseño de nuestro Hardware**. Una vez ya sabemos lo que queremos hacer pasaríamos a abordar el cómo vamos a hacerlo. Se diseña un circuito con diferentes elementos (combinacionales, secuenciales y memorias) y con unas señales de control que los dirijan. Este circuito debe de ser capaz de ejecutar todas las instrucciones seleccionadas en el apartado anterior y para ello es muy importante elegir bien los componentes y sus conexiones, además del orden en el que se dispongan.

3. **Creación de componentes y etapas**. Es hora de empezar con el software, es decir, la programación. Será necesario programar todos los elementos mencionados en el apartado anterior. Una vez hecho esto se tendrán que unir los unos con los otros para formar el esquema global final. Es posible unir varios componentes para formar etapas independientes y simplificar el problema, como ya se ha explicado en la página 24.

4. **Simulaciones.** A través de la herramienta de simulación utilizada se comprobará si los resultados obtenidos coinciden con los esperados. Antes de esto se deberá haber creado uno o varios programas de prueba. En ellos se probarán todas situaciones que sean de interés, teniendo que conocer perfectamente el resultado teórico que deseamos que realice y como plasmar eso en el lenguaje máquina.

5. **Implementación en la Basys 3.** El paso final de esta metodología consistirá en introducir el programa simulado dentro de la placa, la cual deberá asignar los elementos correspondientes para su ejecución. Una vez programada se deberá comprobar si realiza las tareas que el programa le pide de la misma manera que lo hacía en la simulación, es decir, de la manera esperada cuando diseñamos el programa de prueba correspondiente.

"La metodología top-down consiste en que un diseño complejo se divide en diseños más sencillos que se puedan diseñar (o describir) más fácilmente. La metodología bottom-up consiste en construir un diseño complejo a partir de módulos, ya diseñados, más simples. En la práctica, un diseño usa generalmente ambas metodologías."(«VHDL», 2017) Por lo tanto nuestro diseño tiene parte de las dos metodologías, top-down en la primera parte ya que ya era conocido el esquema global del MIPS monociclo, y bottom-up en la parte de segmentación ya que a partir de ir modificando los módulos que se tenían en el monociclo se llega al procesador segmentado.

Capítulo 3. Herramientas

3.1. Herramientas utilizadas

Basys-3

"La tarjeta BASYS 3 es una plataforma de desarrollo bastante completa basada en la FPGA ARTIX-7 de la empresa XILINX. La tarjeta cuenta con: una FPGA de alto rendimiento (XC7A35T-1CPG236C), varios puertos (USB y VGA). Con esta tarjeta se puede implementar los principales diseños combinaciones y secuenciales. Una de sus principales características es que cuenta con: 16 interruptores (SW), 5 botones pulsadores (push boton), 16 led, una pantalla de 7 segmentos de 4 dígitos y 4 conectores de expansión PMOD; suficientes para realizar una gran variedad de pruebas sin la necesidad de construir hardware adicional. Con los 4 conectores PMOD se pueden utilizar para agregar hardware adicional y darle nuevas características a la tarjeta aumentando las capacidades.

La FPGA ARTIX-7 instalado en la BASYS 3 esta optimizado para ofrecer un alto rendimiento y mejores características que los modelos anteriores, las principales características de este dispositivo son:

- *33,280 celdas lógicas en 5200 partes (cada parte tiene 4 LUTs de 6 entradas y 8 flip-flop)*
- *Bloque de memoria RAM de alta velocidad de 1800Kbits*
- *5 administradores de reloj con lazo seguidor de fase (PLL)*
- *90 sectores dedicados para DSP*
- *Reloj interno de 450MHz*
- *Convertidor analógico/digital incorporado en la FPGA*

Los principales componentes de la tarjeta de desarrollo BASYS 3 se muestran en la figura siguiente. Los números indicados en la figura y la tabla que le sigue muestran la ubicación de los principales componentes.

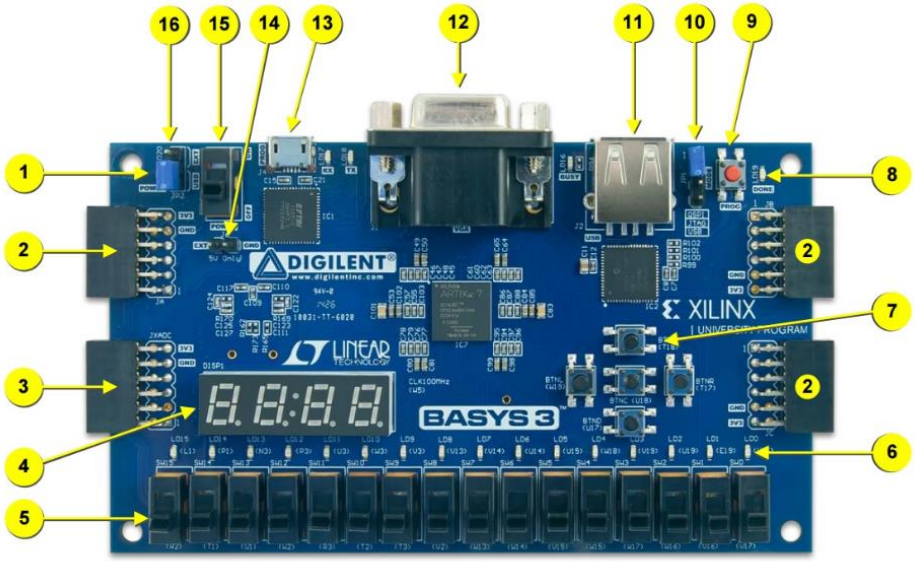

Ilustración 2: Placa Basys 3 **(«Basys 3 Artix-7 FPGA Board - Lógica Programable»)**

Nº	Descripción	Nº	Descripción
1	Led indicador de voltajes correctos	9	Botón de reset de la FPGA (restaura la configuración de la FPGA)
2	Conectores PMOD	10	Jumper para seleccionar el modo de configuración
3	Conector PMOD que también puede ser utilizado por el ADC interno de la FPGA	11	Conector USB HOST
4	Pantalla de 7 segmentos de 4 dígitos	12	Conector para monitores VGA
5	Interruptores	13	Conector micro-USB utilizado para la configuración y para comunicaciones UART
6	LEDs	14	Jumper para alimentación externa
7	Botones pulsadores	15	Interruptor de encendido
8	LED que indica cuando el proceso de configuración ha terminado	16	Jumper que selecciona cual es la fuente de alimentación. Selecciona entre el conector micro-USB o la alimentación externa.

Tabla 1. Componentes Basys 3 **(«Basys 3 Artix-7 FPGA Board - Lógica Programable»)**

La BASYS 3 está diseñada para ser programada por el nuevo ambiente de desarrollo de XILINX llamado VIVADO DESIGN SUITE, en la cual hay nuevas herramientas de diseño que facilitan el desarrollo de proyectos. Se ejecuta más rápido y permite un mejor uso de los recursos de la FPGA y permite a los diseñadores explorar nuevas alternativas de diseño. El editor de sistemas (System Editor) incluye un analizador lógico dentro del chip, la herramienta de síntesis de alto nivel y además de otras herramientas de última generación. En la versión gratuita del VIVADO DESIGN SUITE llamada WEBPACK se pueden crear proyectos en la BASYS 3 fácilmente sin costo adicional."(«Basys 3 Artix-7 FPGA Board - Lógica Programable»)

Como no es relevante su funcionamiento interno o sus especificaciones para este trabajo no se indagará más en este tema, pero tanto en la página web de DIGILENT(«Digilent Store») como en el portal del cual se ha obtenido toda esta información (referenciado al final del anterior párrafo) hay más contenido disponible sobre esta tarjeta de desarrollo.

Vivado

"*Vivado es un software desarrollado por Xilinx para la síntesis en alto nivel y el análisis de diseños y circuitos electrónicos en HDL. A diferencia de ISE que se basó en ModelSim para la simulación, el Sistema de Edición Vivado incluye un simulador propio para la lógica incorporada. Utiliza una cadena de herramientas que convierte el código C en la lógica programable. Vivado permite a los desarrolladores compilar sus diseños, realizar análisis de tiempo, examinar diagramas del tipo RTL, simular la reacción de un diseño a diferentes estímulos, y configurar el dispositivo de destino con el programador.*

Vivado es compatible con dispositivos de alta capacidad más nuevos, y acelera el diseño de la lógica programable y I / O (entrada/salida). Vivado está preparado para acoplarse perfectamente con las placas FPGA de Xilinx, más en concreto está dirigido a sus placas más grandes, por lo que nuestra Artix-7 entra perfectamente dentro de esta categoría."(«Xilinx Vivado», 2017) Nosotros hemos usado la versión WebPack gratuita que ofrece a los diseñadores un entorno de diseño limitado, pero más que suficiente para realizar nuestro trabajo.

VHDL

"VHDL es un lenguaje definido por el IEEE (Institute of Electrical and Electronics Engineers) usado por ingenieros y científicos para describir circuitos digitales o modelar fenómenos científicos respectivamente. VHDL es el acrónimo que representa la combinación de VHSIC y HDL, donde VHSIC es el acrónimo de Very High Speed Integrated Circuit y HDL es a su vez el acrónimo de Hardware Description Language. Aunque puede ser usado de forma general para describir cualquier circuito digital se usa principalmente para programar PLD (Programable Logic Device - Dispositivo Lógico Programable), FPGA (Field Programmable Gate Array), ASIC y similares.

Dentro del VHDL hay varias formas con las que se puede diseñar el mismo circuito y es tarea del diseñador elegir la más apropiada entre las siguientes:

***Funcional o Comportamental**: Se describe la forma en que se comporta el circuito digital, se tiene en cuenta solo las características del circuito respecto al comportamiento de las entradas y las salidas. Esta es la forma que más se parece a los lenguajes de software ya que la descripción puede ser secuencial, además de combinar características concurrentes. Estas sentencias secuenciales se encuentran dentro de los llamados procesos en VHDL. Los procesos son ejecutados en paralelo entre sí, y en paralelo con asignaciones concurrentes de señales y con las instancias a otros componentes.*

***Flujo de datos**: Se describen asignaciones concurrentes (en paralelo) de señales.*

***Estructural**: Se describe el circuito con instancias de componentes. Estas instancias forman un diseño de jerarquía superior, al conectar los puertos de estas instancias con las señales internas del circuito, o con puertos del circuito de jerarquía superior. Es la recomendada cuando el diseño digital se vuelve complejo o está conformado por múltiples bloques de hardware.*

*-**Mixta**: combinación de todas o algunas de las anteriores."*(«VHDL», 2017)

Para el trabajo se ha decidido usar la forma estructural, ya que me parece la más adecuada ante un trabajo de esta envergadura. Dentro de cada componente se utiliza el diseño funcional por lo que se podría decir que el proyecto global utiliza un diseño mixto, siendo solo estructural en su programa principal desde el cual se llama al resto de proyectos.

Capítulo 4. Desarrollo

4.1. Microprocesador Monociclo

Antes de comenzar a elaborar nuestro diseño se van a aclarar algunos conceptos importantes. Para la realización de este trabajo se utilizará la arquitectura Harvard (memoria de datos y memoria de instrucciones) por ser más moderna y evitar problemas de riesgo tales como el acceso a la memoria, además de conseguir un rendimiento mucho mayor.

Hay que saber distinguir entre arquitectura e implementación, dos conceptos bases a la hora de entender un procesador:

-Arquitectura: repertorio de instrucciones, registros, modelo de excepciones, manejo de memoria virtual, mapa de direcciones físicas y otras características comunes.

-Implementación: forma en que los procesadores específicos aplican la arquitectura.

4.1.1. Instrucciones

Como ya se ha comentado anteriormente la primera decisión relevante que hay que tomar para diseñar un microprocesador es qué tipo de tareas y funciones queremos poder llevar a cabo. Estas vienen marcadas por las instrucciones, que van a tener un tamaño de 32 bits, representados como 8 números hexadecimales. Como pequeño repaso, las tipos principales de instrucciones y su estructura son:

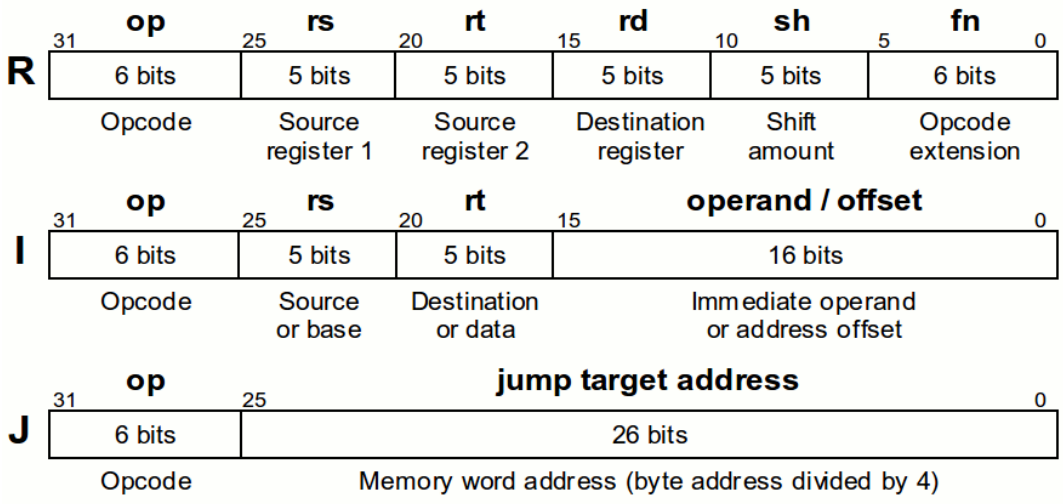

Ilustración 3. Formato de cada tipo de instrucción (Universidad Tecnológica de la Mixteca, s. f.)

Tipo R: está compuesto por las instrucciones aritméticas y lógicas. Los primeros 5 bits corresponden al código de operación, que siempre será una combinación de 6 ceros consecutivos cuya utilidad es la de indicar el tipo de instrucción. A continuación 10 bits destinados a los registros fuente, es decir indicadores de los registros desde los que se van a obtener los datos para operar, 5 de esos bits son para la primera fuente (rs) y 5 para la segunda (rt). 5 bits de campo destino o rd, que nos dice el registro en el que se va a almacenar el resultado de la operación, 5 bits de shamt o desplazamiento para poder hacer un direccionamiento indirecto y, para finalizar, 5 bits para el campo *function* (función), que indicará que hacer con esos valores.

Tipo I: es el formato utilizado por las instrucciones de transferencia de datos, saltos condicionales e instrucciones con operandos inmediatos. El código operación ocupa los primeros 6 bits y es diferente dependiendo la instrucción a realizar. Los siguientes 5 bits nos direccionan al registro fuente (rs) pero en esta ocasión los 5 siguientes, es decir los 5 correspondientes a rt se utilizan como destino y no como fuente. Por último los 16 bits finales se utilizan para representar el desplazamiento o el operando inmediato.

Tipo J: instrucciones de salto incondicional o bifurcación. Destinan 6 bits como identificador de tipo, igual que las otras y el resto (26 bits) para la dirección.

La decisión fue tomada a partir del procesador visto en la asignatura Electrónica Digital en la cual se estudia el funcionamiento de un MIPS monociclo con las instrucciones más básicas y se comprueba que con esas instrucciones básicas se pueden llevar a cabo multitud de tareas. Por tanto el repertorio de instrucciones disponible es el siguiente:

Tipo R

- Add: suma el valor leído de rs con el de rt y lo almacena en rd. Su campo función (*function*) tiene el valor “100000”.

- Sub: resta el valor leído de rs con el de rt y lo almacena en rd. Su campo función tiene el valor “100010”.

- Slt: compara rs con rt y si el primero es menor pone a 1 (binario) rd. Su campo función tiene el valor “001000”.

- AND: operación lógica AND entre rs y rt cuyo resultado es almacenado en rd. Su campo función tiene un valor “100100”.

- OR: operación lógica OR entre rs y rt cuyo resultado es almacenado en rd. Su campo función tiene un valor “100101”.

Tipo I

- Sw: su código de operación es 43 en binario. Coge (lee) el dato almacenado en rt y lo almacena en la dirección de memoria formada por el contenido de rs más un desplazamiento.

- Lw: su código de operación es 35 en binario. Coge (lee) el dato almacenado en la dirección de memoria formada por el contenido del registro rs más un desplazamiento y lo almacena en el rt.

- Addi: su código de operación es 8 en binario. Suma el valor leído del registro rs con una constante (con signo) y almacena el resultado en rt.

- Beq: su código de operación es 4 en binario. Compara los valores de los registros rs y rt y si son iguales salta a la dirección formada por la suma de la señal de salida del contador de programa, el número 4 en binario y un desplazamiento. La unión de las dos primeras señales forma la señal PC+4 de la cual se hablará con mayor detalle en apartados siguientes.

Tipo J

- J: su código de operación es 2 en binario. Los 26 bits posteriores indican la dirección a la que se salta, sin condición, al ejecutarse. Esta dirección estará formada por los 4 valores más significativos de la señal PC+4 mencionada antes concatenados con los 26 primeros bits de la instrucción desplazados dos posiciones a la izquierda.

A partir de estas instrucciones sería muy fácil añadir una gran variedad de instrucciones extras que solo necesitarían de pequeñas modificaciones en

el software, hardware, o ambos y de la unidad de control. Algunos ejemplos serían: stli, XOR, bne, etc.

Aunque no se haya explicado aquí hay que recordar que la forma de llamar a estas instrucciones sigue un orden que podría no corresponderse con nuestro pensamiento más lógico o asociativo. Por ejemplo, cuando se escribe add $1, $2, $3 lo lógico sería pensar que se mantiene el orden de la estructura de la instrucción y rs es $1, rt es $2 y rd es $3. Sin embargo esto no es así, en este caso la operación llevada a cabo es $1 = $2 + $3 (con signo), por lo que $1 es rd mientras que $2 y $3 son rs y rt respectivamente. Esta precaución hay que tenerla en el resto de instrucciones pero no es algo elegido por el programador, sino una forma de sintaxis que se puede consultar en cualquier libro de procesadores, y que una vez aprendida nos servirá para cualquiera de los procesadores que utilicemos. Un buen artículo para consultar este y otros apartados básicos de los procesadores sería el blog de Alma Celeste Flores Martínez(Alma Celeste Flores Martínez, s. f.).

4.1.2. Diseño

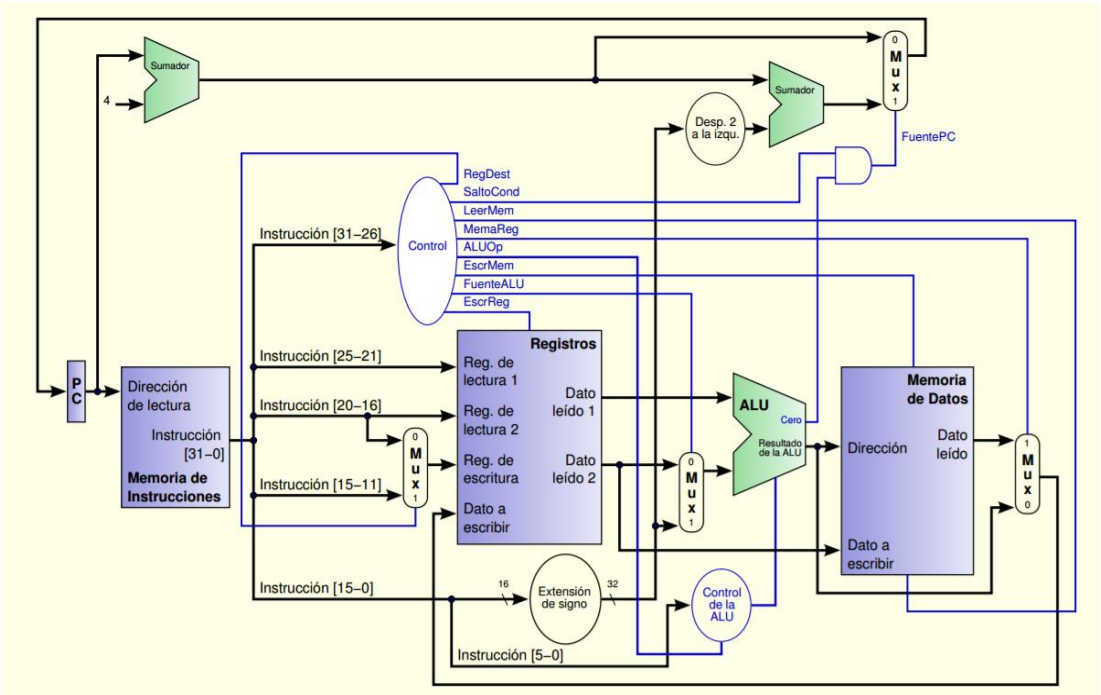

Ilustración 4. Esquema MIPS monociclo (Sergio Barrachina Mir, 2006)

Para la realización de este trabajo se ha tomado como referencia este esquema del MIPS básico monociclo que se estudia en la asignatura Electrónica Digital, la cual utiliza el trabajo realizado por Sergio Barrachina Mir(Sergio Barrachina Mir, 2006).

Como se puede apreciar sigue una estructura cíclica, que no tiene entradas ni salidas y que en definitiva no forma parte de una arquitectura mayor. En la práctica el microprocesador siempre estaría integrado dentro de un sistema mayor y tendría entradas y salidas para poderse comunicar con él y así poder utilizar los resultados de las operaciones que el microprocesador efectúa. En negro se representa las señales de datos que recorren el circuito con la información de la instrucción (datos y resultados) y en azul se representan las señales de control que le dicen a los elementos del microprocesador que hacer con las señales de datos.

Al esquema anterior le falta añadir la circuitería necesaria en la parte superior para poder ejecutar la instrucción j de salto incondicional. El conjunto quedaría así:

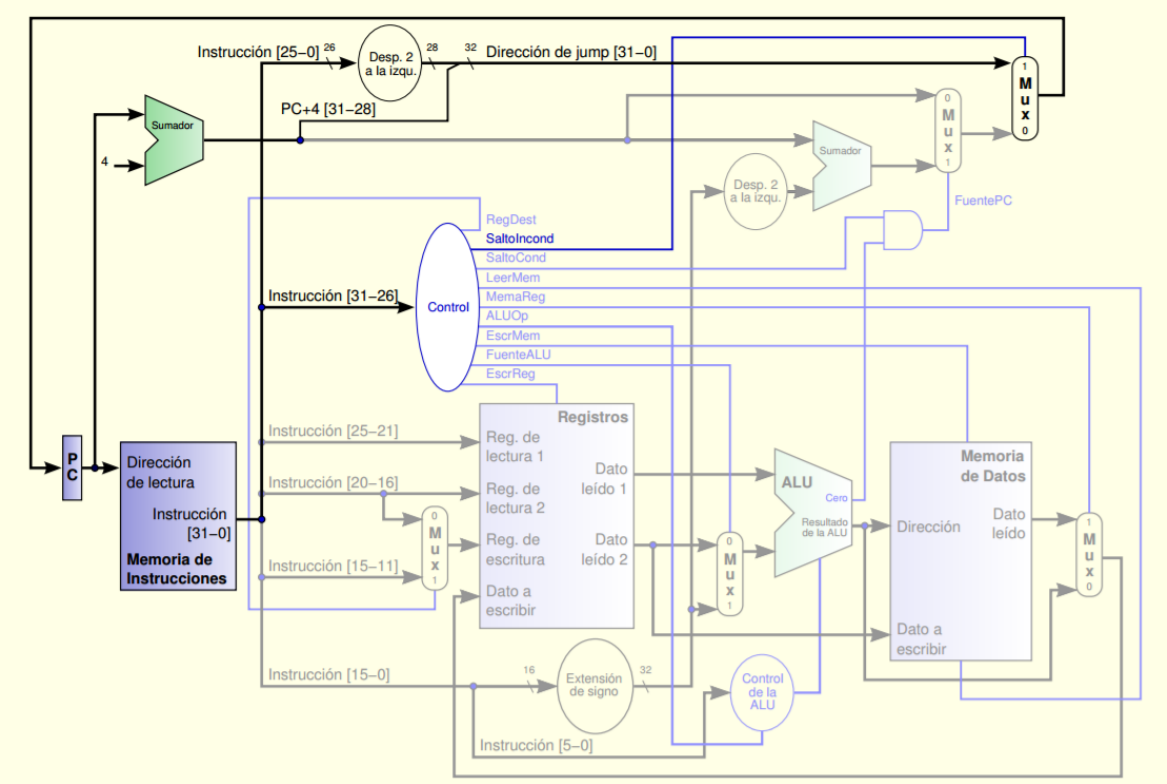

Ilustración 5. Circuitería salto incondicional (Sergio Barrachina Mir, 2006)

Se ha tenido que añadir una señal extra de control, de nombre Salto Incond y marcada en azul para poder indicar cuando se desea hacer el salto incondicional y cuando no.

La única diferencia de mi esquema final con respecto a este es que he optado por unir los dos multiplexores de la parte superior del esquema en uno solo de 4 entradas y 1 salida, al que le entran las dos señales de control (FuentePC y SaltoIncond). Esta acción habrá que tenerla en cuenta a la hora de elegir los valores de la unidad de control. Se pasa de un comportamiento secuencializado, en el cual si la señal SaltoIncond está activa da igual lo que valga la señal FuentePC, a uno en paralelo, donde ambas señales tendrán la misma importancia y por tanto tendrán que tener siempre valores bien definidos para poder ejecutar el tipo de salto que deseemos.

En color más marcado se encuentra toda la circuitería que utiliza el salto incondicional. A partir de ahora cada vez que citemos los esquemas de la asignatura Electrónica Digital para el MIPS básico monociclo se estará haciendo referencia a los encontrados en esta página y la anterior.

4.2. Programación del Microprocesador Monociclo

Como se ha explicado en el apartado anterior este trabajo se basa en un esquema ya dado, y por lo tanto no es tarea del autor diseñar el microprocesador monociclo, sino entender completamente su funcionamiento, tanto global como de cada uno de sus componentes. La razón por la que el microprocesador monociclo está incluido en este trabajo es debido a que se ha decidido implementarlo en VHDL y de esta manera utilizarlo como base para el microprocesador segmentado, que es el tipo de microprocesador que queremos obtener al finalizar este trabajo. El código principal del microprocesador monociclo se encuentra en el ANEXO 1.

4.2.1. Componentes

En este apartado se va a explicar el funcionamiento de cada componente y si se ha tenido que hacer alguna modificación con respecto a los teóricos usados en los esquemas de referencia. No se mostrará código

como tal, el cual estará disponible en el ANEXO 2 al final de esta memoria. Las aclaraciones pertinentes se realizarán a través de diagramas de flujo y referencias al código de la parte final.

Contador de Programa (CP)

El contador de programa es un registro que nos va otorgando un valor de 32 bits (ya que es nuestro tamaño de instrucción) cada flanco de subida del ciclo de reloj, proporcionándonos de esta manera un número asociativo a cada instrucción, es decir, un orden. El primer número será un conjunto de 32 ceros y los siguientes valores se recibirán por la señal correspondiente con la salida del multiplexor en la parte superior del esquema.

A la salida de nuestro contador de programa se encuentra un sumador que añade el valor 4 (en binario "100") a nuestro actual valor. La razón de sumarle 4 y no otro valor es que los dos bits menos relevantes se reservan para futuras utilidades. En nuestro procesador no nos han hecho falta para ninguna acción pero se ha decidido mantenerlos ya que aportan al programa un punto de versatilidad ante nuevas utilidades que se le puedan dar en el futuro.

La razón de que este sumador se haya implementado en un código conjunto con el contador de programa no ofrece ninguna ventaja de rendimiento, sino simplemente una forma más compacta y precisa de tener las dos señales de salida (CP y CP+4), además de ahorrarnos otro *project* para definir dicho sumador y otra señal en el programa principal para conectar ambos.

Si se quisiera separar sumador de contador de programa como se hace en el esquema principal no habría ningún impedimento, simplemente habría que copiar el código del sumador individual en un nuevo *project* y modificar el programa principal, añadiendo otro componente. Las señales de entrada serían el valor de CP y un valor fijo 4 (VHDL acepta de igual manera que se ponga el número 4 en decimal o binario, solo que de esta última manera tiene que estar representado en un tamaño de 32 bits).

Memoria de instrucciones

La memoria de instrucciones es una memoria ROM (solo lectura) la cual tiene almacenada la secuencia de instrucciones que queremos ejecutar, es decir, el programa que se llevará a cabo. La manera de introducir esas instrucciones es mediante programación, dentro del *proyect* ROM destinado al código de esta memoria.

Recibe la señal del contador de programa sin los dos primeros bits (en el apartado del contador de programa se explica el porqué) y da de salida la instrucción de 32 bits que se encontraba almacenada en esa dirección. En la práctica solo recibirá parte de esos 32 debido a que la placa Basys 3 no permite implementar un tamaño de memoria tan grande como 2^32.

Partiendo de este punto se nos plantean dos opciones: usar los módulos de memoria de la Basys 3 o la sintetización, es decir, que se utilicen parte de los registros de la placa como memoria. Esta decisión es transcendental para el devenir del trabajo ya que de la primera forma las memorias de nuestro microprocesador serían síncronas, mientras que para el segundo caso las memorias podrían ser asíncronas.

Si planteamos el problema desde el punto de vista de la velocidad de funcionamiento, es decir, de la frecuencia o ciclo de instrucción, se verá rápidamente que la opción de disponer de memorias asíncronas siempre es mejor. El asincronismo nos dará el valor correcto en mayor número de ocasiones y para velocidades mayores que el sincronismo. Debido a esto los modelos que se toman como ejemplo usan este tipo de elementos.

La decisión adoptada en un primer lugar será realizar el microprocesador monociclo y segmentado de la misma manera que en las referencias utilizadas, es decir, usando memorias asíncronas, las cuales actualizan sus señales de entrada y salida en todo momento. Al finalizar esta tarea se analizará si los resultados obtenidos son satisfactorios y si es necesario o no realizar otro tipo de modelo.

El asincronismo usado utilizando VHDL consiste en la sensibilidad de las entradas, es decir, se actualiza el valor de salida cada vez que cualquiera de las entradas cambia su valor.

Extensor de signo

El extensor de signo se basará en un circuito que tendrá como entrada los primeros 16 bits de la instrucción (salida de la ROM) y completará con unos hasta 32 bits si el bit en la posición 15 de la instrucción es un 1. De lo contrario, si es un 0, los 16 bits añadidos a la izquierda serán ceros. Esto se hace así para respetar el valor del número en complemento a 2.

Desplazadores

Disponemos de dos desplazadores en nuestro hardware y ambos realizan la misma acción. Son desplazadores de 2 a la izquierda por lo se basan en añadir 2 ceros en las posiciones menos significativas de la entrada.

Aunque se podría realizar un código común para ambos, aparecerán separados en el ANEXO 2. La razón es que los tamaños de entrada y salida son diferentes.

El primer desplazador se utilizará para desplazar los 26 bits menos significativos de la instrucción, convirtiéndolos en 28. Esta acción forma parte del cálculo de la dirección de salto incondicional.

El segundo desplazador por su parte recibirá la señal de salida del extensor de signo y le añadirá 2 ceros a la derecha, “perdiendo” los dos bits más significativos, ya que el tamaño de la entrada y la salida deben de ser el mismo.

Sumadores

Los sumadores (elementos que suman las señales de entrada y muestran el resultado en una única salida) utilizados para este esquema son tres. Por un lado se encuentra el sumador integrado en el contador de programa, indicado anteriormente y por otro el sumador integrado dentro de la ALU.

El sumador restante y que será un elemento individual con código propio sumará la señal CP+4, la cual es la salida del primer sumador, y por el otro la señal desplazada del extensor de signo, explicado con anterioridad.

Sumando estas dos señales, ambas de 32 bits, obtendremos una nueva señal de 32 bits que no aceptará desbordamiento (*overflow)*.

Multiplexores

Los multiplexores son los elementos más simples del microprocesador y nos servirán para elección de unas señales frente a otras. Todos los multiplexores del mismo tamaño podrán ser referenciados con el mismo nombre en el código final, aunque sean elementos diferentes, gracias a que se usará la técnica de instanciación.

Para el microprocesador monociclo tendremos cuatro multiplexores, tres de los cuales son iguales de tamaño 2 a 1, pero uno de ellos elegirá entre señales de 5 bits mientras que los otros dos elegirán entre señales de 32 bits. El último de estos multiplexores tendrá un tamaño de 4 a 1, aunque solo tres de esas entradas serán utilizadas. A todos estos multiplexores se les conectará diferentes señales de control que decidirán el valor de la salida y de las que se hablará más adelante.

Banco de Registros

Nuestro banco de registros tendrá un tamaño de 32x32, es decir, 32 registros que almacenarán valores de 32 bits. Dispondrá de cuatro entradas, además de la señal de reloj y la habilitación de escritura. Tres de estas entradas tendrán un tamaño de 5 bits para hacer referencia a los registros de los cuales se va a leer y al registro en el que se va a escribir, en el caso de que la señal de habilitación de escritura esté activa. En ese caso se utilizaría también la última de estas entradas, que sería el dato de 32 bits preparado para almacenarse en el registro correspondiente. Esta opción de escritura se realizará de manera síncrona, utilizando la primera mitad del ciclo de ejecución, mientras que la lectura será asíncrona, utilizando el mismo método explicado para las memorias en la página 37. Como datos de salida tendremos los dos valores almacenados en los registros referenciados a la entrada.

En un principio todos los registros tendrán almacenados 32 ceros como forma de inicialización, pero solo uno de ellos, el registro 0, no podrá cambiar de valor, teniendo deshabilitada la función de escritura. También hay

que recordar que la manera más común de referenciar a los registros es con t seguido de un subíndice entre 0 y 7 para los registros entre el 8 y el 15 respectivamente, y con la letra s usada de la misma manera para los siguientes, es decir, que si vemos una referencia al registro S_3 se estará refiriendo al registro número 19 de los 32 de nuestro banco.

Unidad de Control

Se trata de un circuito combinacional que recibe los 6 bits más significativos de la instrucción (salida de la ROM) y los convierte en diferentes señales según la siguiente tabla:

CO_5	CO_4	CO_3	CO_2	CO_1	CO_0	SaltIncond	RegDest	FuenteALU	MemaReg	EscrReg	LeerMem	EscrMem	SaltoCond	ALUOp1	ALUOp0
0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0
1	0	0	0	1	1	0	0	1	1	1	1	0	0	0	0
1	0	1	0	1	1	0	0	1	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1
0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0

Tabla 2. Entradas y salidas unidad de control

Las señales CO corresponden al código de operación y son los 6 bits de entrada que acabamos de explicar. Las salidas corresponden a cada una de las señales de control que controlarán cada instrucción, diciéndole a cada elemento qué deben hacer con el dato. Como se puede comprobar coinciden con las señales de control que se indicaban en el esquema de la página 32.

Cada fila corresponde con una instrucción y las señales de control asociadas a ellas, todas las de tipo R tienen el código de operación común y por tanto necesitan las mismas señales de control. Las siguientes filas ordenadas de arriba hacia abajo corresponden con las siguientes instrucciones: *lw*, *sw*, *beq*, *addi* y *j*.

En algunos casos hay señales de control que no tienen relevancia y podrían valer tanto 1 como 0, representándose normalmente con una X. Para ganar en simplificación se ha optado por poner 0 en lugar de X, quedando la tabla que acabamos de analizar.

Para optimizar el código en el programa general no se utilizarán 10 señales de control (las salidas de este circuito) si no que se utilizará una señal de tamaño 10 cuyo valor se corresponda con las señales de control, manteniendo el orden visto en la tabla, es decir, siendo ALUOp0 el bit menos significativo del conjunto y SaltoIncond el bit más significativo.

Control de la ALU

El control de la ALU como su propio nombre indica es un circuito combinacional con la tarea de decidir que tarea debe hacer la ALU, es decir, controlarla para que realice la acción correcta para cada instrucción. Para saber qué tipo de instrucción es y por tanto qué señal mandarle a la ALU este circuito tiene como entradas el campo *function*, que son los 6 bits menos significativos de una instrucción y ALUOp, que son dos señales de control que sirven para que el control de la ALU diferencie instrucciones. La tabla de funcionamiento del control de la ALU es la siguiente:

Código operación	ALUOp	Campo de función	Operación	Acción deseada	ControlALU
LW	00	XXXXXX	*cargar palabra*	suma	010
SW	00	XXXXXX	*almacenar palabra*	suma	010
Branch Equal	01	XXXXXX	*saltar si igual*	resta	110
R-type	10	100000	*suma*	suma	010
R-type	10	100010	*resta*	resta	110
R-type	10	100100	*and*	and	000
R-type	10	100101	*or*	or	001
R-type	10	101010	*poner si menor*	slt	111

Tabla 3. Entradas y salidas control ALU (Sergio Barrachina Mir, 2006)

Como en casos anteriores las X representan valores que pueden ser 0 o 1 de manera indiferente ya que no afectan a la salida. Tanto en las instrucciones de tipo I como en las de salto incondicional el campo de función o *function* forma parte del operando inmediato o de la dirección relativa o absoluta a la que saltar. En definitiva este campo solo tendrá relevancia a nivel de control de la ALU para distinguir las diferentes instrucciones de tipo R.

Que se puede simplificar de la siguiente manera:

ALUOp	Campo de función	ControlALU
00	XXXXXX	010
01	XXXXXX	110
1X	XX0000	010
1X	XX0010	110
1X	XX0100	000
1X	XX0101	001
1X	XX1010	111

Tabla 4. Simplificación control ALU (Sergio Barrachina Mir, 2006)

A partir de esta tabla programaremos el control de la ALU. Si nos fijamos en la primera tabla notamos que no están descritas ni la instrucción *addi* ni la instrucción j ya que el autor no las tuvo en cuenta. De igual manera no se necesita ninguna modificación en este apartado para añadirlas. En la unidad de control se estudió las señales de control que genera cada instrucción y se puede comprobar fácilmente que *addi* se comporta para este componente como una *lw* o *sw*. En el caso de la instrucción *j* da igual la señal que mandemos a la ALU ya que no es necesario realizar ninguna operación con ella.

ALU

La unidad aritmético-lógica o ALU es el circuito combinacional encargado de realizar las operaciones aritméticas o lógicas que necesita cada instrucción. Como señal de entrada tiene los dos valores con los que opera y el control de la ALU que decide qué hacer con ellos. Los datos recibidos serán de 32 bits al igual que la salida mientras que la unidad de control tiene un tamaño de 3 bits. Además de estas señales habrá 1 bit de salida que se pondrá a 1 en el caso de que el resultado sea igual a 0. De esta manera podremos realizar correctamente la instrucción *beq*, que solo salta en el caso de que los dos valores leídos sean iguales.

Nuestra ALU tendrá implementadas las siguientes operaciones: AND y OR como lógicas y suma, resta y menor que como aritméticas. Esta última operación se realiza para la instrucción s*tl,* la cual consiste en escribir un 1 binario en un determinado registro si A es menor que B. La ALU comparará ambos valores y su resultado valdrá 1 en binario si esa condición se cumple.

Memoria de Datos

La memoria de datos es una memoria RAM, lo que significa que permite tanto escribir como leer. El razonamiento del tamaño que podemos programar es el mismo que con la memoria de instrucciones, por lo que como entrada de este componente solo podremos recibir algunos de los 32 bits que se obtienen a la salida de la ALU. El número de bits menos significativos que elijamos representará la dirección, tanto de escritura como de lectura.

Además de esta señal se tendrá como entrada el valor a escribir, de 32 bits, procedente del segundo puerto de lectura del banco de registros. Por último como entradas tendremos dos señales de control: EscrMem (que cuando está activa habilita la escritura) y LeerMem (que cuando está activa habilita la lectura). Como salida tendremos únicamente el dato almacenado en la dirección indicada, de 32 bits, siempre y cuando la lectura esté activa.

La solución adoptada con respecto al sincronismo será la misma que para la memoria de instrucciones, es decir, que las salidas serán asíncronas, ya que actualizarán su valor cada vez que lo haga cualquier valor de entrada.

La mayoría de estos elementos serán utilizados de manera íntegra y sin modificaciones en el microprocesador segmentado, por lo que realizar el MIPS monociclo no solo tiene un carácter académico para aprender VHDL y entender el funcionamiento base, sino que también tiene parte funcional, ya que será trabajo realizado para la programación del microprocesador segmentado.

4.2.2. Señales

Dentro de algunos componentes se declaran señales, estas se utilizan como forma de evitar que haya inconsistencia de valores, ya que la forma de programación de VHDL es totalmente secuencial. Otro método posible es utilizando variables, cuya diferencia principal con respecto a las señales es que se declaran dentro del process y que actualizan su valor instantáneamente, y no al final del proceso como las señales. Dependiendo del comportamiento deseado puede ser más acertado utilizar una u otra técnica.

En el programa principal también se declararán señales, están se usan para realizar la instanciación. Este método consiste en llamar desde el programa principal a los diferentes componentes que queremos utilizar. En cualquiera de los programas principales que se encontrarán en los "ANEXOS" al final de esta memoria se podrá encontrar esta forma de programar.

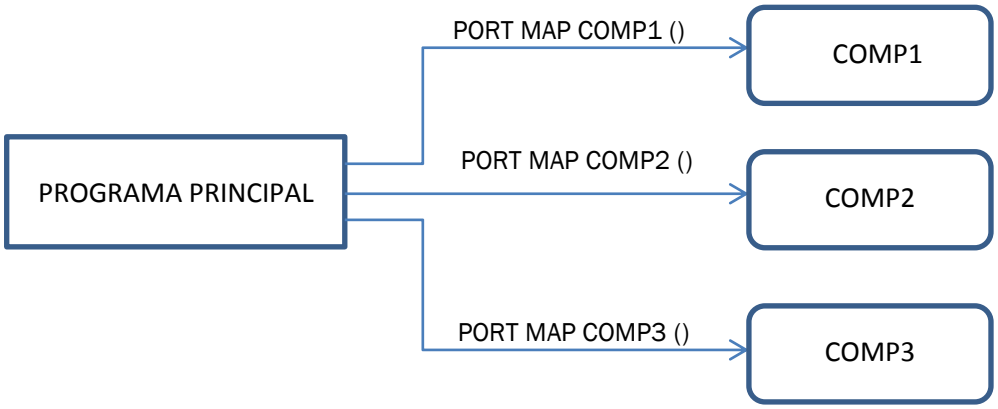

Ilustración 6. Instanciación de componentes

En esta imagen se representa la instanciación más básica. El programa principal llamará a los diferentes componentes (COMP1, COMP2 y COMP3) a través de un "mapeo", que en el código aparecerá como PORT MAP. Dentro de los paréntesis se declararán las asociaciones de cada variable del componente con la señal del programa principal correspondiente.

4.2.3. Programas de prueba

Los programas de prueba tendrán la finalidad como su propio nombre indica de probar el procesador para ver si funciona de manera adecuada, tanto simulando con Vivado como en la propia placa Basys 3. Para el microprocesador monociclo no existe ninguna combinación de instrucciones críticas, es decir, en las que haya un riesgo de fallo mayor que en las demás. Por lo tanto el programa de prueba creado para este tipo de MIPS tendrá que utilizar todas las instrucciones pero no en un orden determinado. Como es lógico debemos conocer perfectamente el funcionamiento de cada instrucción y saber qué es lo que queremos que haga nuestro programa.

Como en la asignatura de Electrónica Industrial impartida por mi tutor, Santiago Cáceres Gómez, se nos enseña el funcionamiento del procesador monociclo y el lenguaje ensamblador a través de dos programas ejemplo cuya función es realizar la multiplicación 3*5=15 de dos maneras diferentes. Teniendo este material disponible he decidido utilizar estos dos programas como programas de prueba y añadir un tercero, que sería el único diseñado íntegramente por mí, que consistirá en realizar las instrucciones no comprobadas en los dos anteriores y así completar todo el repertorio. El código ensamblador de cada programa sería:

PROGRAMA 1

nº	PC	Etiqueta	Operación	Instrucción	Registros
1	0x00000000		s1 = (zero) + 3	addi $s1, $zero, 0x0003	s1 = multiplicando
2	0x00000004		s2 = (zero) + 5	addi $s2, $zero, 0x0005	s2 = contador/mltr
3	0x00000008	FOR	If s2 = (zero) goto FIN	beq $s2, $zero, FIN	
4	0x0000000C		s2 = s2 + (-1)	addi $s2, $s2, 0xffff	
5	0x00000010		s3 = s3 + s1	add $s3, $s3, $s1	s3 = resultado
6	0x00000014		If (zero) = (zero) goto FOR	beq $zero, $zero, FOR	Salto incondicional
7	0x00000018	FIN	If (zero) = (zero) goto FIN	beq $zero, $zero, FIN	Salto incondicional

Tabla 5. Programa 1 («Material de la asignatura Electrónica Digital»)

La forma de realizar la multiplicación de este programa se basa en almacenar los números 3 y 5 en diferentes registros para luego realizar un bucle for (con dos instrucciones beq) que vaya sumando el valor que hay en el registro s1, que es 3, tantas veces como el valor de s2, que es 5, y acumula el resultado en otro registro, que en este caso será s3. Disminuir el indice, es decir el valor 5, hasta 0, se obtiene sumando -1 al valor de ese registro en cada iteración, lo cual se realiza con una instrucción addi. Una vez completado el bucle *for* se llega a la última instrucción, que saltará siempre a sí misma, dándole un final redundante al programa.

La columna número de la izquierda se corresponde con el valor de PC, que es la columna siguiente, y representan el orden de las instrucciones, pero eso no significa que se ejecuten una detrás de otra, si no que es el orden en el que se introducirán en la memoria de instrucciones. Las etiquetas son una forma de referenciar una instrucción para que al programar nos resulte más sencillo. La columna de Operación sería el pseudocódigo el cual explica cada instrucción pero de una manera mucho más intitutiva. La columna instrucción nos muestra el código ensamblador. Por último en la columna Registros se indica el qué se almacena en cada registro y la acción de los saltos incondicionales.

Si al código ensamblador le añadimos el código máquina nos queda:

PC	Instrucción	Binario (memoria de instrucciones)	Hexadec.
0x00000000	addi $s1, $zero, 0x0003	001000 00000 10001 0000000000000011	0x20110003
0x00000004	addi $s2, $zero, 0x0005	001000 00000 10010 0000000000000101	0x20120005
0x00000008	beq $s2, $zero, 0x0003	000100 10010 00000 0000000000000011	0x12400003
0x0000000C	addi $s2, $s2, 0xffff	001000 10010 10010 1111111111111111	0x2252FFFF
0x00000010	add $s3, $s3, $s1	000000 10011 10001 10011 00000 100000	0x02738820
0x00000014	beq $zero, $zero, 0xfffc	000100 00000 00000 1111111111111100	0x1000FFFC
0x00000018	beq $zero, $zero, 0xffff	000100 00000 00000 1111111111111111	0x1000FFFF

Tabla 6. Código ensamblador programa 1 («Material de la asignatura Electrónica Digital»)

PROGRAMA2

nº	PC	Etiqueta	Operación	Instrucción	Registros
1	0x00000000		s1 = (zero) + 3	addi $s1, $zero, 0x0003	s1 = multiplicando
2	0x00000004		s2 = (zero) + 5	addi $s2, $zero, 0x0005	s2 = multiplicador
3	0x00000008		s3 = (zero) + 1	addi $s3, $zero, 0x0001	s3 = mascara
4	0x0000000C		s4 = (zero) + 15	addi $s4, $zero, 0x000f	s4 = contador
5	0x00000010	FOR	If s4 = (zero) goto FIN	beq $s4, $zero, FIN	
6	0x00000014		s5 = s2 AND s3	AND $s5, $s2, $s3	s5 = temporal
7	0x00000018		if s5 = (zero) goto NoSUMA	beq $s5, $zero, NoSUMA	
8	0x0000001C		s6 = s6 + s1	add $s6, $s1, $s6	s6 = resultado
9	0x00000020	NoSUMA	s1 = s1 + s1	add $s1, $s1, $s1	s1 << 1
10	0x00000024		s3 = s3 + s3	add $s3, $s3, $s3	s3 << 1
11	0x00000028		s4 = s4 + (-1)	addi $s4, $s4, 0xffff	cont = cont -1
12	0x0000002C		if (zero) = (zero) goto FOR	beq $zero, $zero, FOR	Salto incondicional
13	0x00000030	FIN	If (zero) = (zero) goto FIN	beq $zero, $zero, FIN	Salto incondicional

Tabla 7. Programa 2 («Material de la asignatura Electrónica Digital»)

La forma de calcular la multiplicación de este segundo programa es a través de una máscara. Se almacenan el 3 y el 5 en los mismos registros que antes pero esta vez también se almacena un 1 y un 15 en los siguientes. Este último hará las veces de contador. De esta manera se vuelve a entrar en un bucle *for* que esta vez opera de manera diferente. El procedimiento consistiría en ir comprobando el dígito que ocupa la posición menos significativa del multiplicador. Si este es 1 debemos sumar el multiplicando a una variable, resultado, que contiene las sumas parciales. Si es cero no añadimos nada a dicha variable. Para comprobar si el dígito menos significativo del multiplicador es 1 o 0 utilizamos una máscara que tiene un 1 en el dígito menos significativo y un cero en el resto de los dígitos. Se realiza una operación AND entre el multiplicador y la máscara para saber si se debe sumar el multiplicando a la variable que almacena las sumas parciales. Para comprobar todos los dígitos del multiplicador tenemos que ir desplazándolo a la derecha en cada ciclo de reloj. Esta acción la realizaremos multiplicando por 2, o lo que es lo mismo sumando el número con él mismo. Por otro lado, el multiplicando se va desplazando a la izquierda a medida que el multiplicador se desplaza a la derecha (en realidad desplazaremos la máscara hacia la izquierda ya que es lo que nos permite el elemento disponible). Si el dígito menos significativo del multiplicador es uno, se suma el nuevo valor del multiplicando al resultado. Si el dígito menos significativo del multiplicador es cero, tan solo se desplaza el multiplicando. Añadiendo el código máquina la tabla quedaría:

PC	Instrucción	Binario (memoria de instrucciones)	Hexadecimal
0x00000000	addi $s1, $zero, 0x0003	001000 00000 10001 0000000000000011	0x20110003
0x00000004	addi $s2, $zero, 0x0005	001000 00000 10010 0000000000000101	0x20120005
0x00000008	addi $s3, $zero, 0x0001	001000 00000 10011 0000000000000001	0x20130001
0x0000000C	addi $s4, $zero, 0x000f	001000 00000 10100 0000000000001111	0x2014000F
0x00000010	beq $s4, $zero, 0x0007	000100 00000 10100 0000000000000111	0x10140007
0x00000014	and $s5, $s2, $s3,	000000 10010 10011 10101 00000 100100	0x0253A824
0x00000018	beq $s5, $zero, 0x0001	000100 00000 10101 0000000000000001	0x10150001
0x0000001C	add $s6, $s1, $s6	000000 10110 10001 10110 00000 100000	0x02D1B020
0x00000020	add $s1, $s1, $s1	000000 10001 10001 10001 00000 100000	0x02318820
0x00000024	add $s3, $s3, $s3	000000 10011 10011 10011 00000 100000	0x02739820
0x00000028	addi $s4, $s4, 0xffff	001000 10100 10100 1111111111111111	0x2294FFFF
0x0000002C	beq $zero, $zero, 0xfff8	000100 00000 00000 1111111111111000	0x1000FFF8
0x00000030	beq $zero, $zero, 0xffff	000100 00000 00000 1111111111111111	0x1000FFFF

Tabla 8. Código ensamblador programa 2 («Material de la asignatura Electrónica Digital»)

La gran ventaja de usar estos dos programas ya dados no reside únicamente en que el código ya está elaborado y no hay que diseñarlo de cero sino también en que en la parte de simulación se podrá comparar las simulaciones realizadas con las que se nos proporcionan en estos ejemplos. De esta manera es muy cómodo comparar comportamientos y verificar que nuestro microprocesador funciona como deseamos.

PROGRAMA 3

Como ya se ha comentado para la prueba del monociclo no existen riesgos respectivos al orden o combinación de instrucciones, por lo que simplemente nos limitaremos a probar el conjunto de instrucciones que no se ha realizado todavía. Estas instrucciones son *stl*, *sw, lw, OR, sub* y *j*. El tercer programa de prueba quedará de la siguiente manera:

PC	Instrucción	Hexadecimal
0x00000000	addi $S1, $Szero, 0x0003	0x20110003
0x00000004	addi $S2, $Szero, 0X0005	0x20120005
0x00000008	stl $S3, $S1, $S2	0x0232982A
0x0000000C	sw $S3, 0000($S1)	0xAE330000
0x00000010	lw $S4, 0000($S1)	0x8E340000
0x00000014	OR $S5, $S1, $S2	0x0232A825
0x00000018	sub $S6, $S1, $S2	0x0232B022
0x0000001C	j 0x000000	0x08000000

Tabla 9. Programa 3

El programa consiste en almacenar los valores 3 y 5 en los primeros registros S como se hizo en los programas anteriores. A continuación, se opera con ellos dos para realizar el resto de instrucciones cuyos valores se van almacenando en los registros consecutivos a los ya utilizados.

Este programa también se podrá utilizar para el microprocesador segmentado, con el afán de probar el mayor número de opciones posibles, pero será necesario añadir otro u otros programas debido a que aparecen riesgos relativos a la segmentación.

4.3. Microprocesador Segmentado

La segmentación es una técnica de implementación de unidades de control que permite tratar las instrucciones en serie dividiéndolas en etapas o fases. Con un único cauce de ejecución de instrucciones es posible mantener ejecutándose simultáneamente varias instrucciones, cada una en una fase distinta.

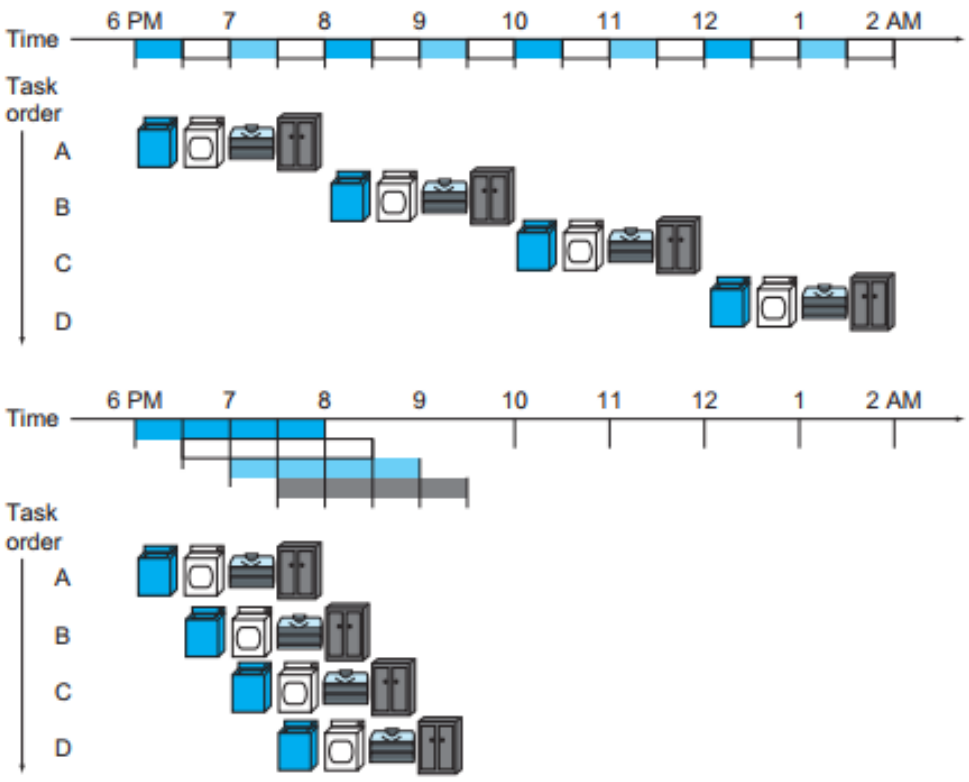

Ilustración 7. Segmentación en una colada (Patterson & Hennessy, 2014, p. 273)

El ejemplo de las etapas de una colada representado en esta figura es muy utilizado para explicar la segmentación. Se puede apreciar claramente como combinando y segmentando los elementos disponibles conseguimos hacer cuatro coladas completas en mucho menos tiempo que haciendo la colada completa de cada una de ellas antes de empezar con la siguiente.

La principal razón para llevar a cabo una segmentación es la ganancia de rendimiento. Al dividir en fases puede haber tantas instrucciones como fases en el interior del procesador y ejecutarse todas ellas en el mismo tiempo en el que un monociclo estaría ejecutando una sola acción. Por tanto, esta mejora del rendimiento viene dada por una mejor distribución de los elementos, consiguiendo reducir notablemente el tiempo de ciclo de ejecución y, por consiguiente, aumentar la velocidad.

El coste asociado a realizar la segmentación es la utilización de más lógica, el hardware y el software de nuestro microprocesador aumentará considerablemente para poder llevar a cabo las mismas tareas con esta nueva forma de tratamiento de las instrucciones.

La referencia principal que se va a tomar para realizar la segmentación va a ser el libro Estructura y Diseño de Computadores de Hennessy y Patterson, aunque como ya se comentó anteriormente esta parte del trabajo tendrá mucha parte de investigación y diseño propio, debido a que es la principal de este trabajo. Siempre que se haga referencia al libro nos estaremos refiriendo a este en concreto, tanto en su versión inglesa como en castellano.

4.3.1. Instrucciones

Las instrucciones que podrá realizar nuestro microprocesador segmentado serán las mismas que para el monociclo ya que estas se decidieron pensando en nuestro procesador en general. El segmentar no hace unas instrucciones más fáciles que otras por lo que si aquellas nos permitían realizar una amplia gama de acciones de manera relativamente fácil, en esta situación se comportarán exactamente igual. La lista de instrucciones que puede ejecutar el procesador se encuentra en el apartado instrucciones del MIPS monociclo.

4.3.2. Diseño

El primer aspecto que debemos decidir es el número de etapas que va a tener nuestro microprocesador segmentado. Para separar unas de otras se utilizan registros, que almacenarán los valores de las señales que decidamos durante un ciclo de reloj, para que así en el siguiente ciclo la etapa siguiente disponga de la información guardada. La siguiente gráfica nos ayuda a tomar esta decisión:

Ilustración 8. Relación etapas-mejora rendimiento (Patterson & Hennessy, 2000, p. 510)

Por lo que observamos una segmentación que arroja unos buenos resultados de eficiencia es la de dividir el MIPS monociclo en cinco fases mediante cuatro registros. Además al tomar como referencia un MIPS monociclo básico encontramos bastantes facilidades para que el tamaño de la segmentación sea este. La disposición de los registros (en azul) con respecto a los elementos principales sería la siguiente:

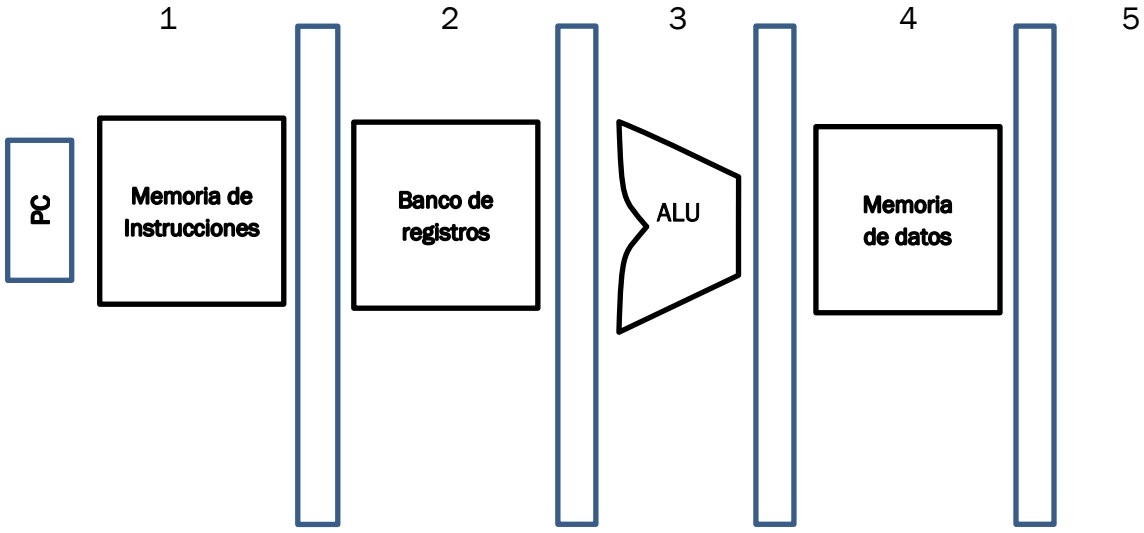

Ilustración 9. Segmentación básica

Cada una de las cinco etapas tendrá con un nombre asociado a su función. En orden ascendente según los números mostrados en la imagen anterior son:

IF (búsqueda): en esta etapa se asocia el valor del contador de programa con la instrucción a realizar, es decir, el número que refleja el orden de cada instrucción con el valor de cada instrucción para que realice las tareas deseadas.

ID (decodificación): en esta etapa se decodifica la instrucción para generar las señales de control que acompañarán a la instrucción. Además se decide la instrucción siguiente a realizar y se obtienen los valores con los que se va a trabajar en la ALU.

EX (ejecución): Se opera con la ALU obteniendo los resultados necesarios para poder ejecutar la instrucción correctamente.

MEM (memoria): Se utiliza la memoria RAM (si es necesaria) ya sea para leer o para escribir.

WB (escritura): Se decide el valor a escribir y se escribe el mismo en el banco de registros.

Cada instrucción que entre en el microprocesador pasará por todas las etapas, estando su instrucción posterior en la etapa anterior. Esto mismo se puede observar en este ejemplo de una secuencia de instrucciones aleatoria:

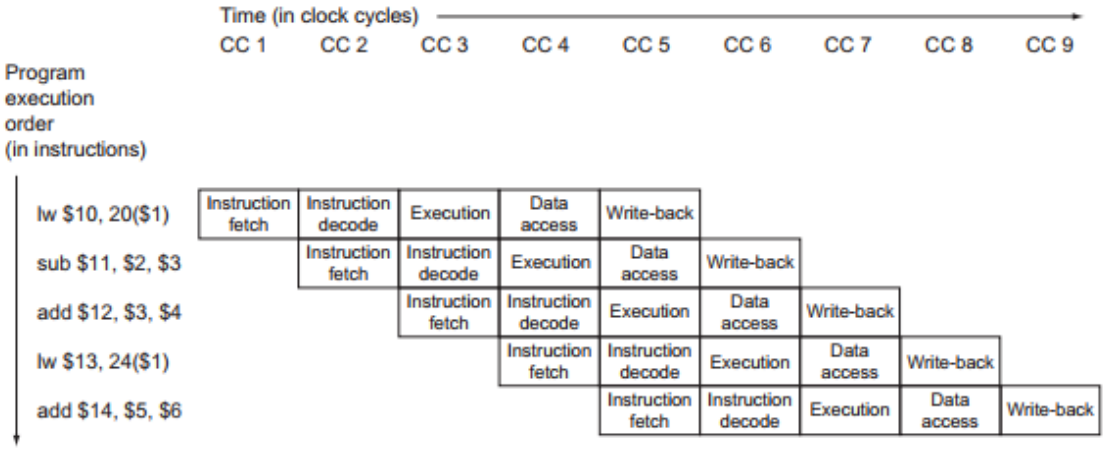

Ilustración 10. Segmentación por etapas (Patterson & Hennessy, 2014, p. 299)

RIESGOS

Hay que tener en cuenta que al diseñar un procesador segmentado aparecen nuevos problemas o riesgos que no existían con el procesador monociclo. Estos se pueden dividir en 3 clases principalmente:

- Riesgos estructurales: van ligados al hardware y se producen cuando hay cierta combinación de instrucciones que el procesador no puede soportar.

- Riesgos de control: son aquellos ligados a las señales de la unidad de control y aquellas que modifican el PC, como los saltos.

- Riesgos de datos: se producen cuando una instrucción necesita para su tarea un valor que se ha modificado en las instrucciones anteriores y que todavía no ha llegado a su destino (se encuentra en alguna de las etapas de la segmentación).

Para solucionar estos problemas utilizaremos diferentes técnicas, intentando no modificar los elementos originales realizados para el MIPS monociclo siempre que sea posible. Estas se explicarán con mayor detalle en el apartado programación que está a continuación de este.

4.4. Programación del Microprocesador Segmentado

Los nuevos componentes y conexiones que van a aparecer en este microprocesador serán parte de las diferentes soluciones que se han tomado para evitar los riesgos explicados en el apartado anterior. Ningún elemento es eliminado con respecto al esquema inicial por lo que si algún circuito explicado para el monociclo no aparece aquí será porque su programación es la misma y únicamente se añade dicho componente con su código correspondiente al nuevo esquema. El código principal del microprocesador segmentado se encuentra en el ANEXO 3.

4.4.1. Componentes

A continuación se pasa a describir por separado cada solución adoptada, explicando todos y cada uno de los componentes que se han tenido que modificar o añadir a nuestro circuito. La manera de explicarlos será la misma que la utilizada para la programación del microprocesador monociclo, pero aportando más detalles que faciliten la comprensión. Todos y cada uno de los códigos correspondientes a los elementos que se describan se encontrarán en el ANEXO 4.

1. RIESGOS ESTRUCTURALES

Como ya se ha comentado anteriormente utilizamos dos memorias, una para datos y otra para instrucciones, lo que se define como arquitectura Harvard. Con este tipo de arquitectura eliminamos los riesgos estructurales referentes a las memorias, que son los más habituales en este tipo de estructura. Con el diseño de segmentación basado en el MIPS básico

monociclo no nos encontramos con ningún riesgo estructural, razón por la cual no habrá que disponer ningún elemento para evitarlos o neutralizarlos.

2. RIESGOS DE CONROL

Estos riesgos residen en las variables de control y en la circuitería necesaria para poder realizar los saltos a tiempo. Para las variables de control no hace falta realizar ninguna modificación en la unidad de control implementada para el monociclo, ya que las señales son las mismas. El único cambio referente a este tipo de riesgo consiste en ir almacenando dichas señales de control en los registros, para que a cada etapa le lleguen sus correspondientes señales, es decir, las de su instrucción, y no las de otra. A medida que pasemos de un registro a otro las señales de control almacenadas en él serán menos, ya que quedarán menos por utilizar al estar cada vez más cerca de la finalización de esa instrucción.

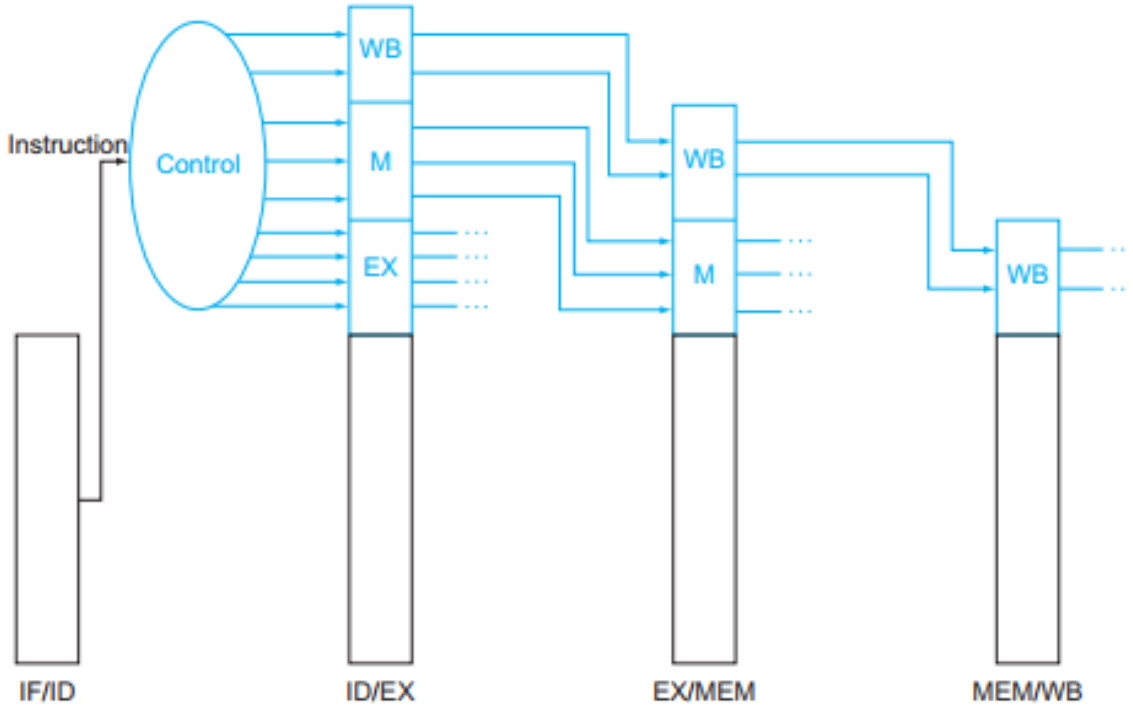

Ilustración 11. Propagación señales de control (Patterson & Hennessy, 2014, p. 303)

Para los saltos la tarea es más compleja. Lo primero es llevar toda la circuitería de decisión y salto a la etapa más próxima del inicio posible, que será la etapa de decodificación, ya que es la primera en la que se tiene las señales de control correspondientes a cada instrucción. Esto se hace para poder saber si hay que realizar un salto o no de la manera más rápida posible y así evitar que instrucciones que no se deben realizar avancen por las fases de nuestro microprocesador. El mayor problema viene ligado a la señal cero, la cual es necesaria para realizar el salto. Para obtener el valor de la señal cero en la etapa de decodificación utilizaremos dos elementos, beq e igualdad, además de dos multiplexores 3 a 1.

Beq

El elemento beq se encarga de generar la señal de control para dos multiplexores que tendrán como entrada el dato leído del banco de registros, el resultado de la ALU en la etapa de ejecución y el resultado de la ALU en la etapa de Memoria respectivamente.

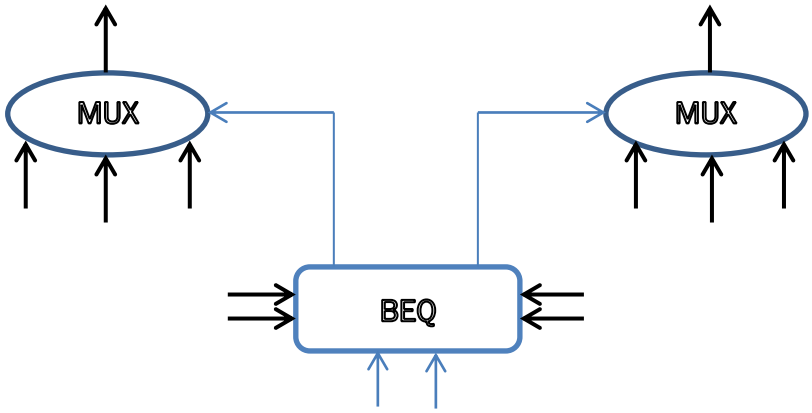

Ilustración 12. Elemento BEQ

Las señales azules de la figura representan señales de control y las de color negro señales normales. Básicamente esta estructura funciona como un anticipador como los que se explicarán posteriormente. Una manera sencilla de entender su funcionamiento es a través de pseudocódigo.

Si (Registro_lectura_1 = Registro_destino_EX

AND EscrReg_EX = 1) entonces

Salida_1 = “01” –Mux escoge el resultado de la ALU en EX

Sino Si (Registro_lectura_1 = Registro_destino_MEM

AND EscrReg_MEM = 1) entonces

Salida_1 = “10” –Mux escoge el resultado de la ALU en MEM

Sino

Salida_1 = “00” –Mux escoge la lectura del banco de registros

Se comprueba si el registro de lectura coincide con alguno de los registros destino de las dos etapas posteriores y si la señal de control que habilita la escritura en esa etapa está activa. Como hay dos registros de lectura habrá también dos multiplexores y esta estructura se deberá repetir, teniendo dos salidas diferentes.

Igualdad

Se ha llamado así al componente cuya tarea es comprobar si cuando la instrucción en ejecución es *beq* las dos salidas de los multiplexores anteriores son iguales.

Si (Salida_mux_1 = Salida_mux_2 AND AluOP = ”01”) entonces

Señal cero = 1

Si no

Señal cero = 0

La igualdad también se podría haber hecho con una combinación de puertas XNOR y AND. Las puertas lógicas XNOR serían tantas como bits tengan los elementos (32 en este caso), y a su entrada tendrían cada par de bits de los dos datos que coinciden en su índice. La puerta AND por su parte

recibiría todas las salidas de las puertas. Esta es una solución más teórica e incluso más elegante pero ya que VHDL nos permite utilizar la expresión igualdad (=) y la tiene perfectamente implementada la utilizamos con el propósito de ahorrar código.

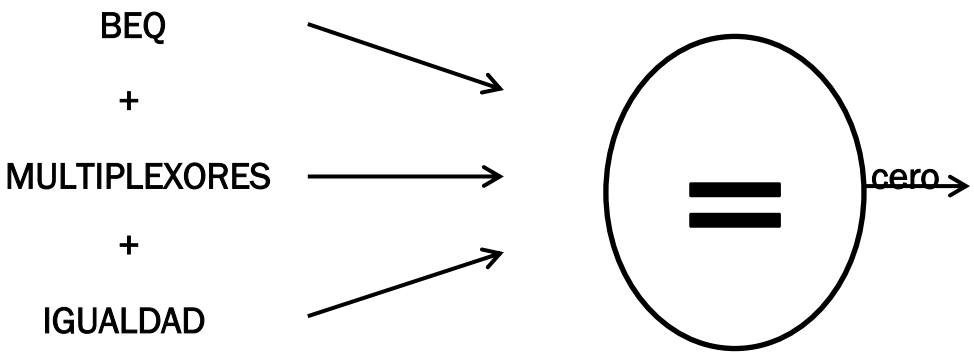

Ilustración 13. Anticipación señal cero

Este será el símbolo para representar conjuntamente toda la circuitería necesaria para obtener el valor de la señal cero en la etapa de decodificación. En el esquema final se utilizará esta abreviatura para ahorrar en espacio y facilitar la comprensión del funcionamiento global. En dicho esquema no tendrá entradas y la única salida será la señal cero.

Una vez obtenidos todos los valores referentes al salto en la etapa de decodificación siguen existiendo problemas. Cuando esta etapa está en curso la instrucción siguiente ya se encuentra en la etapa de búsqueda. De modo que habrá que evitar que esta instrucción ocasiones cambio en nuestro sistema.

Una de las formas de conseguir esto es implementar una pequeña memoria de 1 o 2 bits, dependiendo de la eficacia que queramos que tenga, la cual almacena la predicción. Esta predicción consiste en recordarle al microprocesador si la última vez que ejecutó esa instrucción se efectuó un salto o no, y así poder anticiparse para la actual. En el caso de que se fallara en la predicción se introduciría una instrucción no deseada que habría que eliminar posteriormente, cuando la circuitería de saltos de la etapa de decodificación nos haya informado de si había que tomar el salto o no.

Otra posible solución sería la de introducir instrucciones independientes a los saltos detrás de los mismos, para que en el caso de que sean tomados no haya que anular la instrucción en curso. Esta es la solución más compleja para el programador ya que la elección de instrucciones es muy complicada, y muchas veces no tendremos tantas instrucciones independientes como saltos haya por lo que no mejora tanto la eficiencia para el "gasto" en software que supone.

La última posible solución, y en este caso tomada para resolver los riesgos de salto, se basa en introducir al multiplexor 4 a 1 la señal CP+4 a la salida del sumador (dentro del contador de programa) y no esperar a la etapa de decodificación. De esta manera siempre se toma esa señal como correcta, y en el caso de que una etapa después se tome un salto, habrá que anular la instrucción que se está ejecutando en la fase de búsqueda. Esta solución es tanto mejor cuantos menos saltos haya en nuestro programa. Como normalmente las instrucciones de salto suponen una minoría con respecto al resto de instrucciones suele ser una solución bastante sencilla y que arroja unos buenos resultados de rendimiento. La forma de eliminar la instrucción que se ha "colado" en el circuito se explicará en la parte de componentes.

3. RIESGOS DE DATOS

Este tipo de riesgos son los más frecuentes en la estructura MIPS. Como ya se ha explicado ocurren cada vez que queremos utilizar un dato que todavía no ha sido escrito, ya que se encuentra en alguna de las etapas siguientes. Debido a esto deberemos crear componentes que nos proporcionen esos valores todavía no escritos en los momentos previos a tener que hacer las operaciones. Esta tarea se llevará a cabo gracias a varias unidades de anticipación y una unidad de detección de riesgos. Cada una de ellas llevará ligada sus correspondientes multiplexores y estarán destinadas a evitar uno de los riesgos de datos.

Unidad anticipación escritura (UAE)

Para el microprocesador segmentado existe la posibilidad de querer leer un valor que va a ser escrito en ese mismo ciclo por otra instrucción anterior. Este comportamiento no podía ocurrir en el monociclo porque no existía ninguna instrucción en nuestro repertorio que quisiera leer un valor

que iba a escribir ella misma. Como no sabemos cuánto tiempo se tarda en escribir ese valor en el registro y luego ser leído deberemos anticipar ese valor a la salida, intentando conseguir la mayor velocidad de funcionamiento posible.

En el libro usado como referencia esta anticipación se realiza dentro del mismo banco de registros. Como en nuestro caso se ha optado por no cambiar los códigos iniciales se decide usar una unidad de anticipación de lectura, con multiplexores, fuera del propio banco de registros. De esta manera conseguimos el mismo resultado manteniendo los bloques principales intactos.

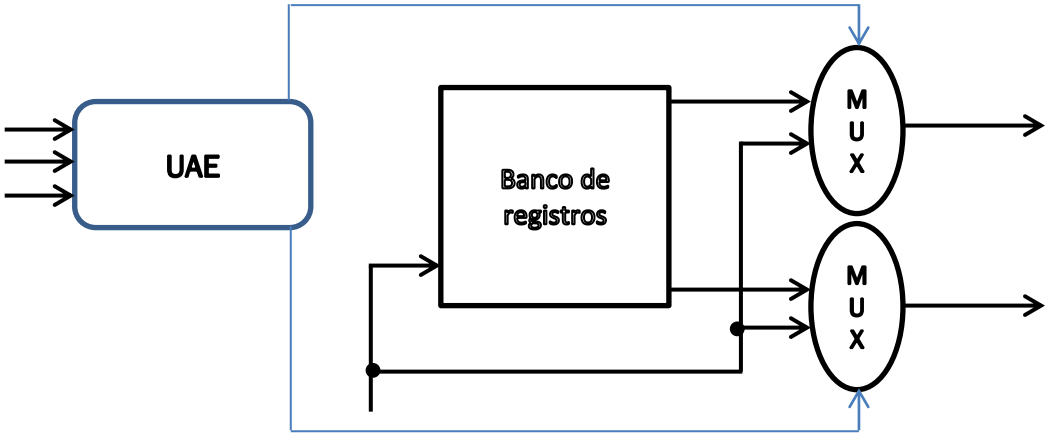

Ilustración 14. Anticipación en la escritura

Salida_1 = 0 y Salida_2 = 0 –Inicialización dentro del *process*

Si (Registro_escritura=Registro_lectura_1

AND Registro_escritura /= “00000”) entonces

Salida_1 = 1

Si (Registro_escritura=Registro_lectura_2

AND Registro_escritura /= “00000”) entonces

Salida_2 = 1

Este elemento en definitiva detecta si el registro de escritura coincide con cualquiera de los dos registros de lectura, mandándole la señal necesaria a cada multiplexor para que anticipe el resultado. El pseudocódigo posterior al esquema nos ayuda a tener una mejor comprensión del componente.

Unidad de anticipación

Se encuentra en la fase de ejecución (EX) y junto con los multiplexores asociados se encarga de anticipar los valores necesarios a las entradas de la ALU.

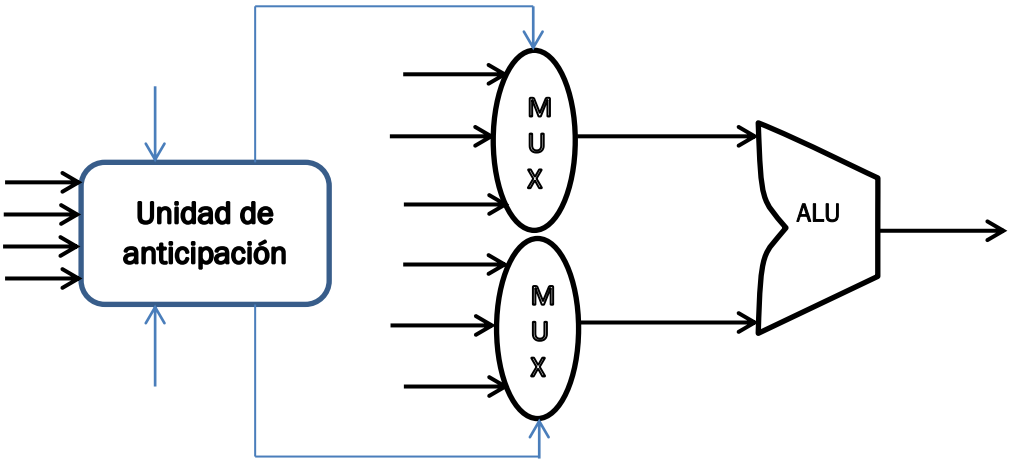

Ilustración 15. Unidad de anticipación

Este componente comprueba si el registro del cual se ha leído en la instrucción actual coincide con el registro destino de alguna de las 2 etapas siguientes (que contendrán las 2 instrucciones anteriores a esta), y si además esa instrucción tiene la señal de habilitación de la escritura activa. Como a la ALU entran dos señales con diferentes valores serán necesarios dos multiplexores. El primero tendrá que seleccionar entre el valor leído del registro de lectura 1, el resultado de la ALU en la etapa de ejecución o el resultado de la ALU en la etapa de memoria. El segundo tendrá las mismas entradas salvo que el valor leído será del registro de lectura 2 y no del 1. La representación en pseudocódigo de este funcionamiento sería la siguiente:

```
Salida_1 = “00” Salida_2 = “00” –Inicialización dentro del process
    Si (EscrReg_MEM = 1 AND Reg_Destino_MEM /= “00000”) entonces
        Si (Reg_Destino_MEM = Registro_lectura_1) entonces
            Salida_1 = “10”
        Si (Reg_Destino_MEM = Registro_lectura_2) entonces
            Salida_2 = “10”
    Si (EscrReg_WB = 1 AND Reg_Destino_WB /= “00000”) entonces
        Si (Reg_Destino_WB = Registro_lectura_1) entonces
            Salida_1 = “01”
        Si (Reg_Destino_WB = Registro_lectura_2) entonces
            Salida_2 = “01”
```

Un apunte importante es recordar que no hay que eliminar el multiplexor controlado por la señal FuenteALU, el cual seleccionaba la entrada al puerto segundo de la ALU. Este multiplexor se encontrará ahora entre el segundo multiplexor de anticipación y la ALU. Como entradas tendrá el valor extendido y la salida del segundo multiplexor de esta unidad.

Unidad de detección Riesgos

La unidad de riesgos es un componente que se emplaza en la etapa de decodificación y que nos ayuda a solucionar el único riesgo de datos que no se puede solucionar con anticipaciones. Se trata del riesgo asociado a la acción de intentar leer un registro el cual va a ser escrito por la instrucción *lw*, es decir, que el dato leído procede de la memoria de datos.

No disponemos del valor deseado hasta que no salga por el puerto de salida de la memoria de datos, lo cual ocurre en un instante que desconocemos, por lo que no se podrá anticipar el valor en ese mismo ciclo. Este tipo de estrategia se hace siempre buscando la mayor frecuencia de funcionamiento, debido a lo cual debemos garantizar que todos los valores que se van a utilizar en una etapa han sido guardados por la etapa anterior en

el registro que separa a ambos, para así disponer de ellos en cuanto se detecte un flanco de subida

Sabiendo esto para realizar esta anticipación de la misma manera en la que se han realizado las otras necesitaríamos poder "viajar en el tiempo", como se muestra en esta imagen:

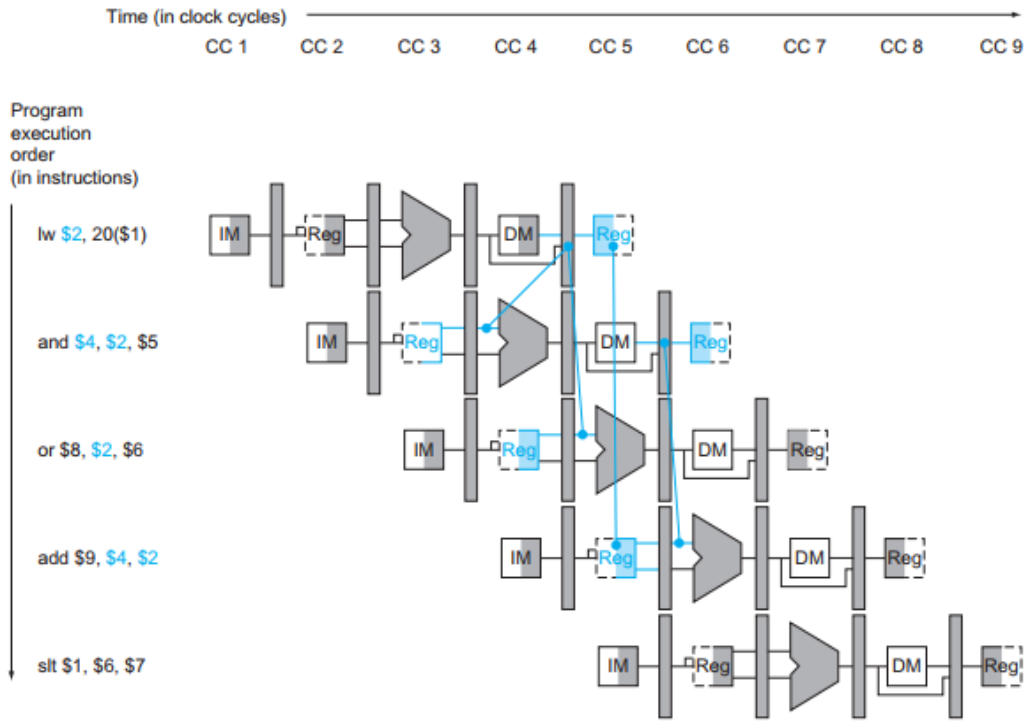

Ilustración 16. Anticipación de dato leído por memoria (Patterson & Hennessy, 2014, p. 313)

Será necesario que el microprocesador se "pare" en sus primeas etapas (IF e ID) repitiendo la instrucción que tienen en ejecución para que la instrucción *lw*, que se encontrará en la etapa de ejecución (EX), pueda avanzar una etapa más. Esto se consigue creando unas señales de control que actúen sobre el contador de programa y sobre el primer registro, haciendo que ambos reciban como entrada el valor que tienen en la salida.

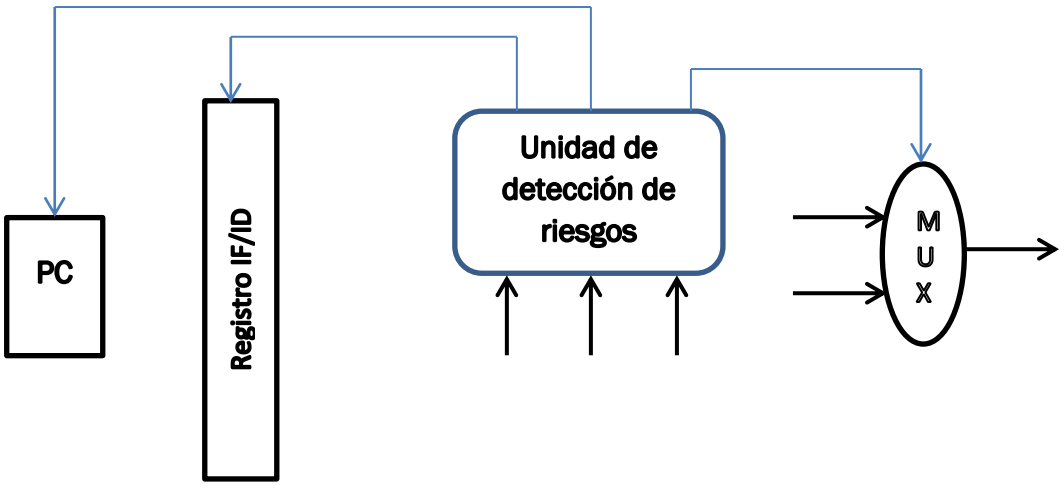

Ilustración 17. Unidad de detección de riesgos

El multiplexor es necesario para evitar la propagación de la instrucción que se encuentra en la etapa de decodificación. Poniendo todas las señales de control con el valor 0 esa instrucción no provocará ningún efecto en el circuito. Como entradas tendrá la señal de control habitual y una señal del mismo tamaño solo de ceros. A continuación se muestra el pseudocódigo con el fin de aclarar más el funcionamiento de este componente:

```
Salida_1 = 0 y Salida_2 = 0 y Salida_3 = 0 –Inicialización dentro del process
    Si (LeerMem_EX = 1) entonces
        Si (Registro_Destino_EX = Registro_lectura_1 OR
        Registro_Destino_EX = Registro_lectura_2) entonces
            Salida_1 = 1 –Manda repetir instrucción a PC
            Salida_2 = 1 –Manda repetir instrucción a registro IF/ID
            Salida_3 = 1 –Escoge la señal de ceros en el mux
```

Una vez hecho esto el dato leído a la salida de la memoria de datos será detectado por las anticipaciones normales, ya que estará almacenado en el registro MEM/WB. La anticipación se realizará cuando la instrucción que lee el registro esté en la etapa de ejecución (EX) y *lw* se encuentre en la de

escritura (WB), quedando en el medio la instrucción NOP que hemos introducido.

La mayoría de estos componentes que comparan registros excluyen al registro número 0, porque tiene un valor constante igual a 32 ceros en binario y no podrá ser modificado, por lo que no importa si se va a habilitar o no la escritura en él.

4.4.2. Señales

Para la parte del MIPS segmentado aparecerán nuevas señales debido a que hay nuevos componentes. Se ha decidido añadir una F al final del nombre de las señales que se engloban dentro de la etapa de búsqueda, una D a las de la etapa de decodificación, una X a las de la etapa de ejecución, una M para las que integran la etapa de Memoria y por último una W para aquellas señales que se encuentren dentro de la etapa de escritura.

Estas señales se han nombrado de la manera más intuitiva posible, tanto para la parte del MIPS monociclo como para la del segmentado. En los casos en que se necesiten aclaraciones habrá comentarios junto a estas señales, como en el resto del código.

4.4.3. Módulos

Como es lógico los módulos programados del microprocesador segmentado coinciden con las etapas en las que hemos dividido el esquema, es decir, cada módulo estará compuesto por los componentes que hay entre registros. Por tanto las acciones de estos serán las mismas que las explicadas en la página 54 de este trabajo.

Para realizar este microprocesador se ha seguido una estrategia contraria a la utilizada para el MIPS monociclo. Mientras que para éste se seguía una metodología top-down porque ya se conocía el esquema final, en este modelo hemos utilizado una metodología bottom-up, ya que se ha ido construyendo el modelo complejo a partir de módulos más simples.

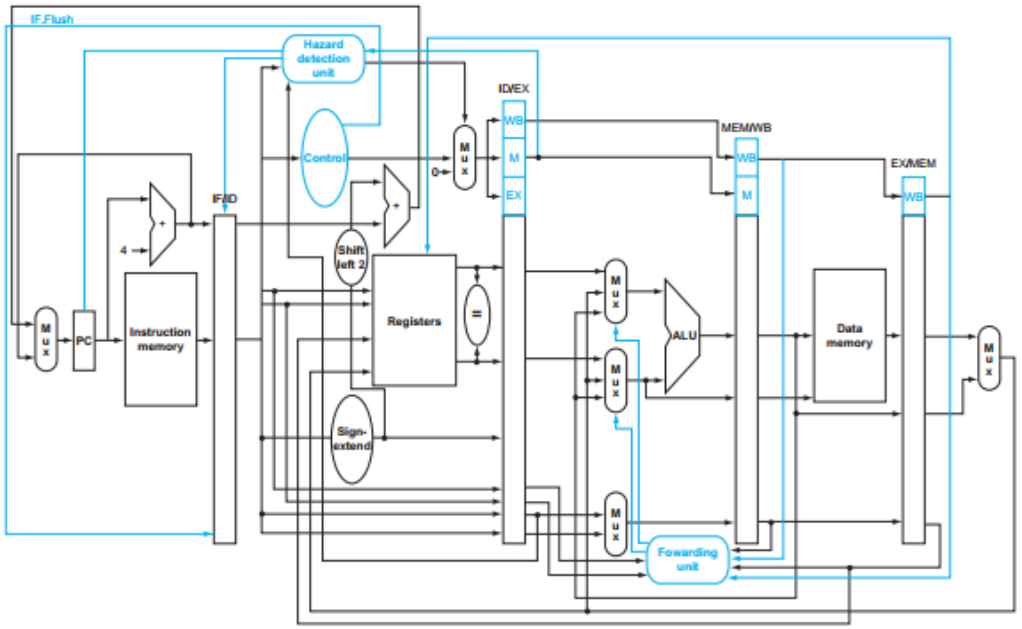

Ilustración 18. Ruta de datos procesador segmentado (Patterson & Hennessy, 2014, p. 325)

En la figura podemos observar como ha quedado el circuito final en el libro usado como referencia. Además la figura siguiente nos mostrará de manera más ampliada una zona de dicho circuito que puede que sea algo confusa.

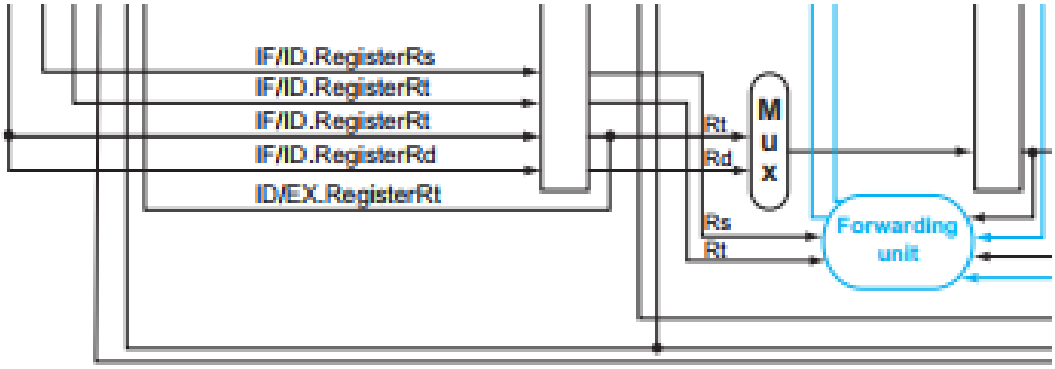

Ilustración 19. Registro destino (Patterson & Hennessy, 2014, p. 316)

Este esquema es prácticamente idéntico al que se ha obtenido, salvo por unos pequeños detalles ya mencionados que se pasan a recordar:

1. Hay que añadir la circuitería necesaria para realizar el salto incondicional, exactamente igual que se hace en la página 38 de este trabajo.

2. Se debe añadir el multiplexor 2 a 1 controlado por la señal de control FuenteALU a la salida del segundo multiplexor de anticipación (el que va al segundo puerto de entrada de la ALU).

3. El símbolo de igualdad representa todo los elementos explicados para la determinación del valor de la señal cero.

4. Para este esquema la anticipación en la escritura va integrada dentro del banco de registros.

4.4.4. Programas de prueba

Además de los dos programas de los que disponemos como ejemplo de la asignatura Electrónica Digital y del realizado para el microprocesador monociclo que ya han sido explicados detalladamente en el este mismo apartado del capítulo anterior, para el microprocesador segmentado debemos crear un programa que pruebe no solo las instrucciones que todavía no han sido probadas, si no determinadas combinaciones de instrucciones que provocan riesgos y que deben ser corregidas, como se ha mostrado en apartados anteriores de este capítulo.

Una vez probados los 3 programas anteriores se comprueba que la única situación de peligro que no se ha verificado es la combinación de *lw* seguida de una instrucción que lea del mismo registro en el que va a escribir. Por lo tanto, diseñamos un programa única y exclusivamente para analizar dicha situación:

PC	Instrucción	Hexadecimal
0x00000000	addi $S1, $Szero, 0x0003	0x20110003
0x00000004	addi $S2, $Szero, 0X0005	0x20120005
0x00000008	sw $S1, 0005($S2)	0xAE510005
0x0000000C	lw $S0, 0005($S2)	0x8E500005
0x00000010	sw $S0, 0006($S2)	0xAE500006

Tabla 10. Programa 4

En este simple programa se almacena el valor 3 y 5 en los registros 17 y 18 respectivamente. Posteriormente la instrucción *sw* almacena en la dirección 0x0000000a (valor del registro 18 + valor inmediato 5) el valor que está almacenado en el registro 17, es decir, 3. A continuación *lw* lee de la memoria de datos el valor recientemente almacenado para guardarlo en el registro 16. Finalmente, este valor es leído por otra instrucción *sw* para almacenarlo en la dirección posterior a la que lo hizo la primera *sw*, que será la dirección 0x0000000b (valor del registro 18 + valor inmediato 6).

4.5. Comprobaciones

En la parte de programación de componentes para el microprocesador monociclo se eligió sintetizar las memorias de manera que la placa utilizase registros para implementarlas y pudieran funcionar de manera asíncrona, dejando abierto el tamaño que deberían tener. El tamaño mínimo se ha fijado en 2^10, es decir, 1024 instrucciones para cada memoria. Hasta ese tamaño el microprocesador tiene una capacidad bastante aceptable y podría desarrollar programas complejos. Para una capacidad menor a esta deberíamos buscar alguna alternativa.

Una vez programado y probado con el simulador de Vivado, se procedió a implementar el microprocesador de tipo segmentado en la placa Basys 3. Al intentar llevar a cabo esta acción la herramienta nos informa de que el número de celdas lógicas, de las cuales hay 33,280 ha sido sobrepasado provocando que no pueda ser posible implementar dicho procesador con este tamaño seleccionado. Como es lógico la solución más sencilla es reducir este tamaño, lo cual nos puede servir para seguir utilizando todo lo explicado anteriormente perdiendo únicamente espacio de almacenamiento y dejando nuestro modelo más enfocado a trabajo didáctico que a funcionar dentro de un sistema más potente.

Otra solución posible es utilizar una de las memorias (preferiblemente la memoria de datos) como síncrona, de manera que el programa nos implementaría la misma en una de las memorias disponibles de la placa. Esta solución es viable para el tamaño que hemos fijado como el mínimo aceptable pero no es la elegida finalmente, ya que si se iba a hacer una programación mixta y se iban a perder la mayoría de beneficios que da el asincronismo era más acertado implementar ambas memorias como síncronas.

Si se quiere tener una cantidad de espacio de almacenamiento "decente" es necesario utilizar un dispositivo de mayor potencia o programar las memorias del microprocesador de manera síncrona. Esta última opción es la que se ha decidido realizar y su desarrollo se encuentra en el capítulo siguiente.

4.6. Microprocesador segmentado en Basys 3

En esta sección se explicarán todas las modificaciones, tanto de software como de hardware necesarias para pasar del microprocesador segmentado que utiliza memorias asíncronas a uno que utiliza memorias síncronas. El código principal de este microprocesador segmentado se encuentra en el ANEXO 5 y el de los componentes a modificar en el ANEXO 6.

Memorias

La solución adoptada ha sido realizar las operaciones de escritura en la primera mitad del ciclo de instrucción y la escritura en el segundo. El problema principal asociado a dicha solución es que tendremos que hacer determinadas tareas en la mitad de tiempo que si las memorias fueran asíncronas y tendremos una frecuencia máxima de funcionamiento algo menor.

Como la herramienta Vivado Design está preparada para programar placas que integren estructuras del tipo FPGA entiende la estructura de una memoria como síncrona, por lo que si se programan como tal, será más sencillo que el compilador entienda lo que se quiere llevar a cabo y nos utilice los componentes acertados, y no otros existentes en la placa que podrían no funcionarnos de la manera deseada.

Un punto importante de esta solución es que si nuestro contador de programa y los registros de segmentación trabajan en flanco de subida, como en este caso, nuestra escritura no podrá hacerse en dicho flanco, porque las señales no habrán llegado. Esto se corrige con una simple modificación que consiste en que la escritura se haga durante todo el semiciclo positivo, es decir, que se escriba durante todo el tiempo que la señal de reloj vale 1 y no solo en el momento que esta señal cambia su valor de 0 a 1. El

comportamiento sigue siendo síncrono porque las entradas vienen de un registro previo activado en flanco de subida.

Unidad de anticipación de lectura

Este componente con sus respectivos multiplexores que integrábamos en el modelo asíncrono desaparecería. La razón es que trabajando con memorias síncronas la frecuencia de ejecución es menor, y se decidió primar la simplicidad por encima de la velocidad. Como en este modelo se van a tener que realizar ciertas acciones en medio ciclo de instrucción no es necesario disponer de los valores leídos por el banco de registros al principio del mismo. Además, se ha comprobado que el banco de registros no es el componente que más tarda en ejecutar sus acciones, siendo este la memoria de datos de tipo RAM. Debido a lo cual, eliminando este componente, simplificamos el código sin aumentar nuestra velocidad de funcionamiento.

Unidad de anticipación

La unidad de anticipación para este modelo añadiría una nueva señal. Esta se activará siempre en los casos en los que lo hacía la unidad de riesgos, es decir, cuando la instrucción que se encuentre en la etapa de ejecución sea una *lw* y la instrucción que se encuentra en la etapa de decodificación quiere leer el valor del registro en el cual va a escribir la *lw*.

El motivo de que ahora este problema se solucione con una anticipación normal y no con la unidad de riesgos es meramente simplificativo. Mientras que en el anterior modelo buscábamos velocidad y no permitíamos utilizar valores que no estuvieran previamente almacenados en registros, en el caso de este nuevo modelo utilizamos el valor a la salida de la memoria de datos en el mismo momento que es obtenido. Como esta memoria ahora es síncrona nos proporcionará el valor en el flanco de bajada y será en ese instante cuando ese valor se introduzca en la anticipación para obtener el valor correcto.

Durante el primer medio ciclo la anticipación ya estará activa pero el valor anticipado será uno incorrecto. Esto no es inconveniente ya que los valores se almacenan en el segundo ciclo. Como es de esperar al realizar este procedimiento se pierde en velocidad, ya que tenemos que tener un ciclo de

ejecución lo suficientemente grande para que entre el flanco de bajada y el de subida nos dé tiempo a operar en la ALU con ese valor y que el resultado esté a la entrada del registro siguiente antes del siguiente flanco. Simplemente se ha limitado a seguir la nueva preferencia de este modelo, que será la sencillez en el código y en el hardware. Una ventaja significativa de este modelo y referente en particular a este componente es que gracias a esta modificación no será necesario introducir ninguna instrucción NOP (todo ceros), por lo que tendremos un modelo que funcionará de manera ininterrumpida excepto cuando haya un salto, tanto condicional como incondicional.

Si para este componente se intentara ganar en velocidad mientras que en otros se intentara reducir la complejidad nos quedaría un modelo con un objetivo mixto, el cual será mucho menos intuitivo para alguien que quiera entender lo realizado en este trabajo.

Unidad de Riesgos

Por lo explicado para la unidad de anticipación este elemento dejaría de tener sentido en este nuevo modelo y será eliminado. No solo se elimina el componente si no las señales que generaba y que se introducían tanto en los dos primeros registros (PC e IF/ID) como en el multiplexor a la salida de la unidad de control.

Esas señales quedarán eliminadas del programa principal y por tanto los registros mencionados también sufrirán una ligera modificación, mucho menor que la de los componentes explicados en este apartado. Siguiendo con esto el multiplexor a la salida de la unidad de control ya no sería necesario por lo que será eliminado de nuestro esquema final.

4.7. Métrica del ciclo de reloj

En este apartado vamos a decidir la velocidad o frecuencia de funcionamiento máxima a la que puede trabajar nuestro microprocesador segmentado y, por tanto, su ciclo de ejecución.

La medición del rendimiento de un microprocesador es una tarea compleja, dado que existen diferentes tipos de "cargas" que pueden ser procesadas con diferente efectividad por procesadores de la misma gama.

El simulador de la herramienta Vivado te deja elegir el tiempo de ciclo de ejecución, pero no se puede utilizar para esta tarea debido a que la simulación aparece correctamente para ciclos de instrucción tan pequeños que no se podrían implementar en la placa, es decir, que darían fallos.

Un método para averiguar el valor real de nuestro ciclo sería el de "prueba y error". Esta estrategia puede ser efectiva, pero es muy laboriosa. Consiste en programar el reloj en el archivo master de la Basys 3, poniendo el valor que deseemos, y comprobando si una vez programada la placa se llega al resultado final esperado. Se iría incrementando esa velocidad hasta que se produjera fallo, es decir, que algunas de las señales del final del programa no coincidan con las deseadas. Este método arroja algún riesgo, un ejemplo aplicado a nuestro caso particular es que Vivado Design nos permite programar la placa para velocidades que obtienen el resultado final del programa correctamente pero que tienen errores de conexiones. Esto lo sabemos porque la misma herramienta nos muestra que hay determinados pines que no se han podido conexionar de la manera correcta. Aun ocurriendo esto es muy posible que el programa que tenemos almacenado en la memoria de instrucciones funcione, ya que el porcentaje de pines que pueden o que dan fallos es menor comparado con el total, pero como queremos garantizar que el microprocesador funciona perfectamente ante cualquier programa o situación no podemos aceptar esos fallos.

Por lo tanto, la mejor manera de averiguar nuestra velocidad máxima de funcionamiento es con el apartado *timing* de la herramienta Vivado Design, el cual nos dice si para la velocidad a la que actualmente tenemos programada la placa existe algún riesgo de conexión de pines o de ejecución de componentes (normalmente el riesgo asociado a la realización de una tarea es muy pequeño, y tendremos un gran margen con la velocidad finalmente elegida). Utilizando esta estrategia encontramos que la velocidad máxima a la que nuestro modelo no desarrolla ningún riesgo es la siguiente:

Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk_pin	{0.000 7.000}	14.000	71.429

Tabla 11. Frecuencia máxima de funcionamiento

El resultado obtenido es que nuestro microprocesador puede trabajar correctamente a unos 70MHz, lo que se traduce en un ciclo de ejecución de 14 ns.

Tras esto se vuelve a introducir el programa en la placa eligiendo esta velocidad para comprobar por un lado que los resultados finales siguen siendo los correctos y por el otro que no existe ningún pin del circuito que no haya sido conectado de la manera correcta, como se pude ver en la siguiente tabla.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,000 ns	Worst Hold Slack (WHS):	0,119 ns	Worst Pulse Width Slack (WPWS):	6,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1365	Total Number of Endpoints:	1365	Total Number of Endpoints:	1348

All user specified timing constraints are met.

Tabla 12. Resumen temporal

La tabla nos muestra que para esa velocidad no hay valores negativos, teniendo tiempo para realizar todas y cada una de las tareas. En la parte de *Hold* se aprecia el margen de tiempo que se tiene para el peor caso de funcionamiento, es decir, el tiempo que "sobraría" en caso de realizar la tarea que necesite un tiempo mayor.

Programando una de las dos memorias como síncrona y la otra como asíncrona se incrementaría la frecuencia de funcionamiento a unos 80-85Mhz, es decir, que funcionaría correctamente par un ciclo de ejecución de 12ns. Como ya se ha mencionado se ha intentado evitar programaciones híbridas o mixtas debido a lo cual el modelo final elegido será el MIPS segmentado con memoria de datos y de instrucciones de carácter síncrono. Por esta razón la implementación de este modelo solo se llevó a cabo a modo de comprobación y como no se va a considerar como una solución final aceptable no se adjuntan tablas o imágenes.

4.8. Resultado Final

El diseño adoptado finalmente es más lento pero más compacto en cuanto a código, siendo la programación más fiel a los materiales disponibles (mejor aprovechamiento), lo que hace que sea más sencilla su aplicación en otros componentes y su comprensión para otros usuarios.

Vivado Design nos proporciona diferentes tablas que nos sirven para verificar que todo se ha realizado de la manera esperada. La utilización de los componentes de la Basys 3 para nuestro modelo final sería:

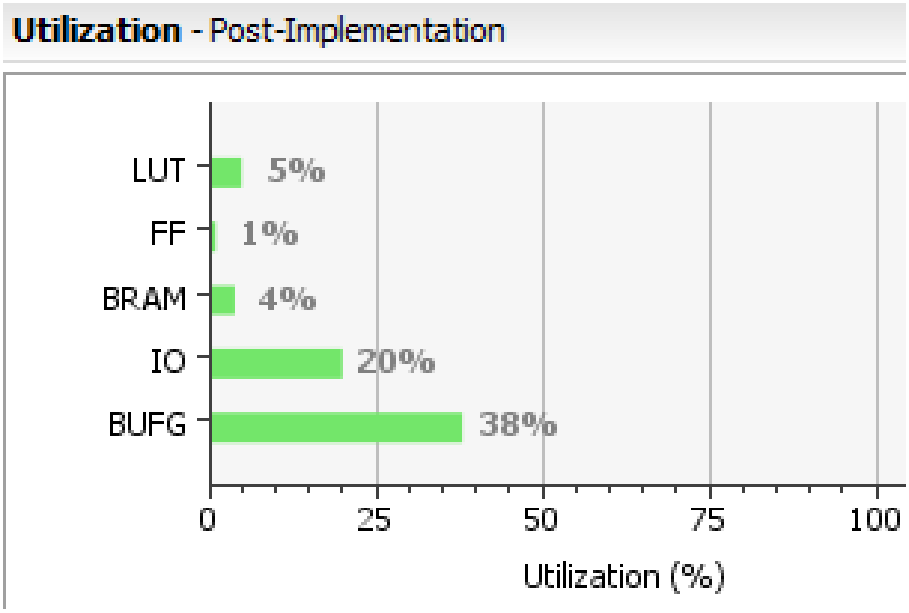

Ilustración 20. Porcentaje utilización

Resource	Utilization	Available	Utilization %
LUT	1110	20800	5.34
FF	309	41600	0.74
BRAM	2	50	4.00
IO	21	106	19.81
BUFG	12	32	37.50

Tabla 13. Utilización Basys 3

Todos los valores se encuentran en un límite aceptable, no teniendo utilizada ni la mitad de la capacidad de ningún tipo de elemento disponible. Con el 4% de utilización de las memorias para el tamaño actual, 1024 instrucciones, es bastante posible que se pudiera incrementar su capacidad hasta las 2048 o 4096 instrucciones para cada una de ellas.

En el apartado de la implementación como en el de la distribución de energía los valores obtenidos siguen la dinámica anterior, logrando valores totalmente correctos, como se puede ver a continuación:

Implementation	
Conflict nets:	0
Unrouted nets:	0
Partially routed nets:	0
Fully routed nets:	2262

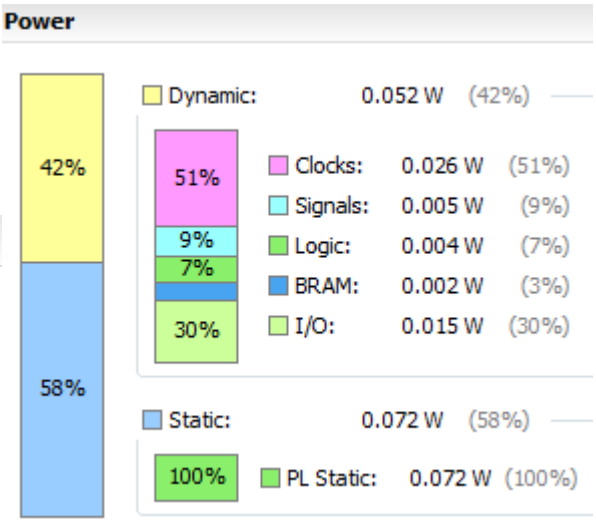

Tabla 14. Valores implementación/ Ilustración 21. Distribución energética

El esquema final del microprocesador segmentado realizando todos los cambios para trabajar con memorias síncronas se encuentra en la siguiente página. Las señales en azul representan señales de control, mientras que en negro representan señales normales.

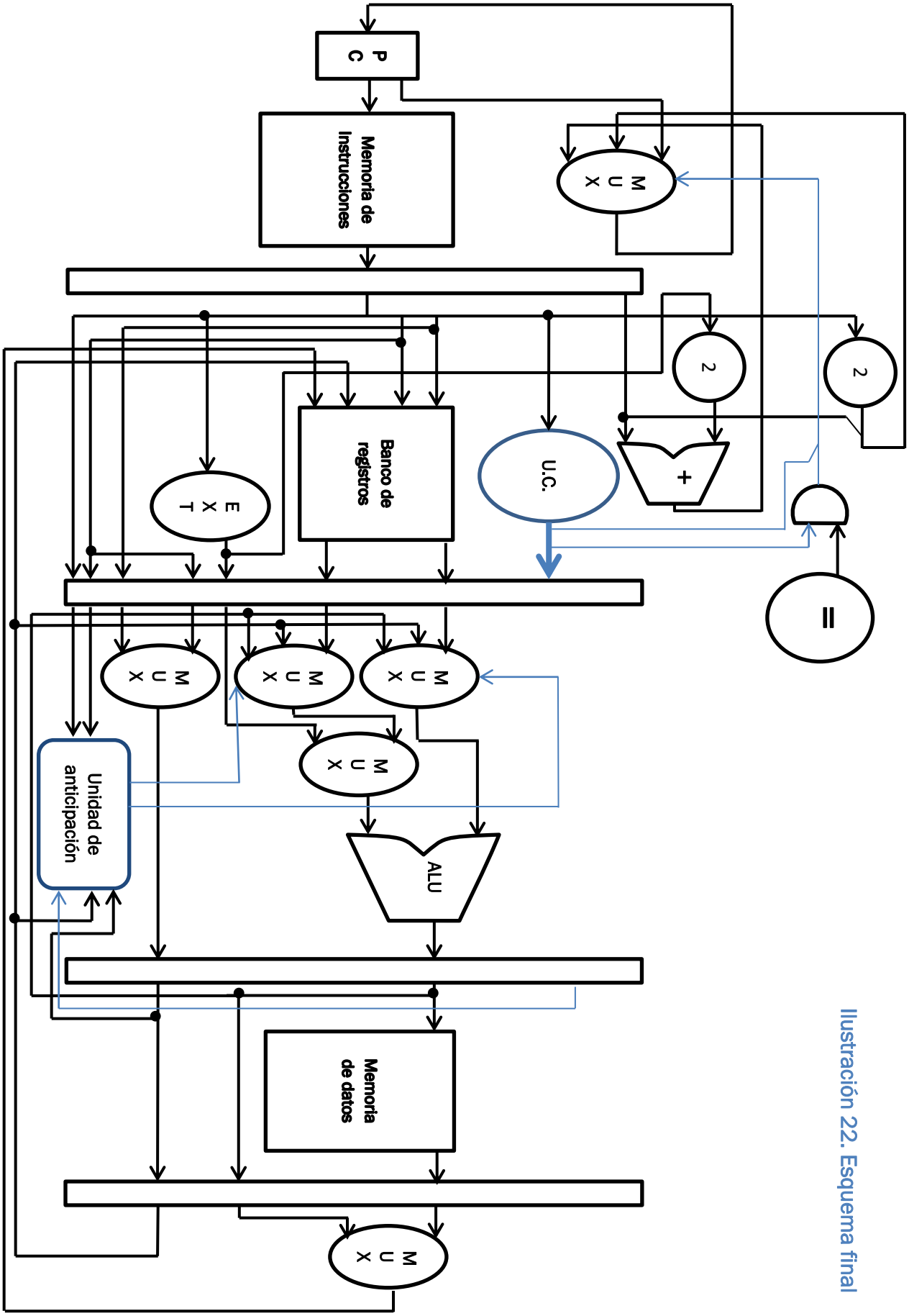

Ilustración 22. Esquema final

Capítulo 5. Simulaciones

5.1. Simulaciones Microprocesador Segmentado

Para hacer las simulaciones se ha utilizado la metodología test bench, consistente en crear un archivo de simulación con todas las señales de entrada y los valores que van a tener a lo largo de la ejecución. De esta manera cuando se arranca la simulación ya obtenemos el resultado final por pantalla. Esta forma de trabajar se aprendió durante la etapa de familiarización con VHDL.

Con la intención de que el trabajo no sea demasiado denso y se ciña lo máximo posible al objetivo principal marcado inicialmente no se mostrarán simulaciones de los modelos no implementados en la placa.

A continuación se pasa a mostrar las imágenes comentadas del correcto funcionamiento de los diversos programas de prueba. Las señales disponibles para mostrar son demasiadas para que se puedan apreciar en una imagen por lo que se han dejado las que he considerado más relevantes. Del mismo modo la mayoría de los programas se alargan demasiado en el tiempo por lo que se mostrará el desarrollo inicial del problema y finalmente los resultados esperados. Tanto la simulación como la implementación de estos programas de prueba han sido mostradas y verificadas por el tutor de este trabajo, Santiago Cáceres Gómez.

PROGRAMA 1

Repasando lo explicado en el apartado 4.2.3. de esta memoria el primer programa de prueba consiste en la realización de la multiplicación 3*5 utilizando una estructura repetitiva, que suma el primer número al mismo registro tantas veces como el valor del segundo número tomado como iterador. Por consiguiente el valor que se espera almacenar en el registro correspondiente es F, es decir, 15 en hexadecimal.

Con este programa se podrá verificar las anticipaciones a la entrada de la ALU, el salto condicional y la anticipación de lectura. Si el desarrollo y el resultado final coinciden con los esperados, el microprocesador segmentado estará trabajando de manera correcta. Teniendo un microprocesador monociclo ya implementado o material teórico de la simulación a realizar se puede comprobar el funcionamiento de manera más rápida, sin tener que ir revisando una a una todas y cada una de las instrucciones que se van ejecutando en el circuito.

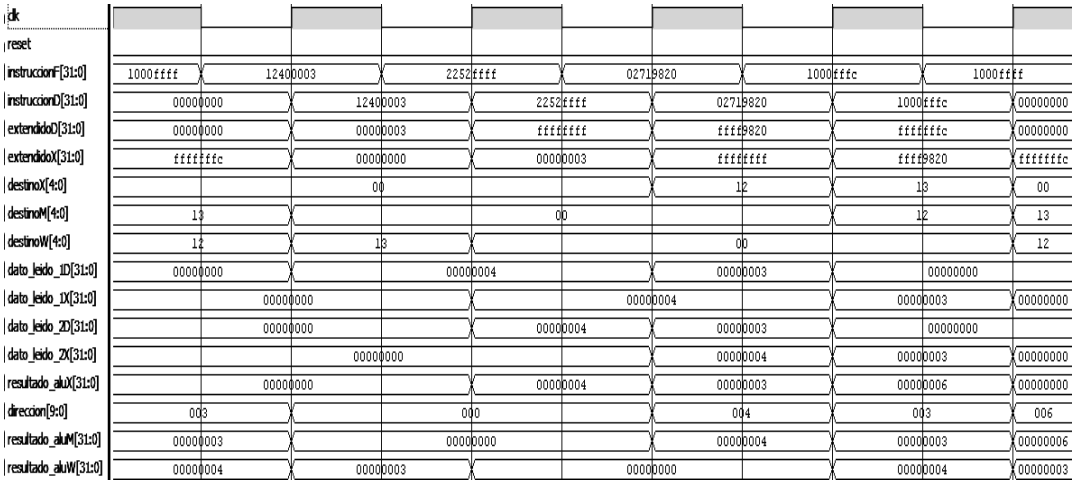

Ilustración 23. Prueba segmentación

En esta primera imagen se ha pretendido mostrar el correcto funcionamiento de la segmentación. Para ello se han seleccionados algunas de las principales señales antes y después de pasar por alguno de los registros del circuito. Cuando llega un flanco de subida en el reloj (señal *clk* situada en la parte superior) los valores de las señales pre registro pasan a la señal post registro, realizando lo que conocemos como propagación.

Tanto para el programa 1 como para el 2 se dispone de simulaciones con el desarrollo y los resultados correctos por lo que se emplearán a modo de comparativa con las simulaciones obtenidas para nuestro modelo con Vivado Design. Estas simulaciones están realizadas para el microprocesador monociclo por lo que el momento de obtener el valor correcto no es el mismo.

En un principio se va a mostrar el arranque del programa con el desarrollo de las primeras instrucciones para corroborar los valores que se van obteniendo antes de llegar a la parte de repetición.

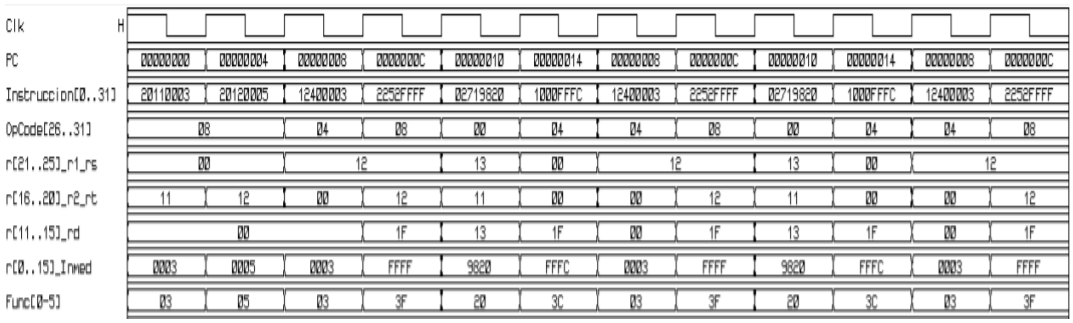

Ilustración 24. Simulación teórica 1 («Material de la asignatura Electrónica Digital»)

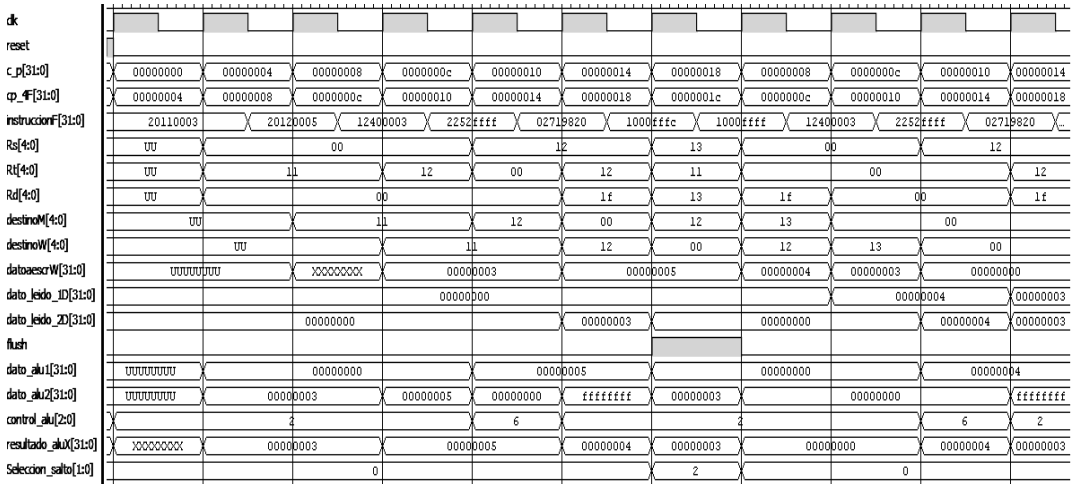

Ilustración 25. Simulación Programa 1

Todos los programas realizados necesitan de una inicialización que viene proporcionada por la señal reset. Esta señal se pondrá a valor 1 para proporcionar un primer valor al contador de programa y después permanecerá a 0 durante el resto del programa, permitiendo que este avance. En el momento que la señal de reset se vuelva a poner a 1 el programa volverá a empezar. Esta señal reset será común para todos los registros, incluido el contador de programa y se puede añadir a la ROM si queremos que no nos muestre ninguna instrucción hasta que el ciclo de reloj no empiece, pero también funciona correctamente sin ella. En la práctica la señal reset será un interruptor de los que disponemos en la placa mientras que la señal de reloj la proporcionaremos a partir de un pulsador para comprobar el correcto funcionamiento.

El valor de ciclo de ejecución que se ha tomado para esta y para el resto de simulaciones es de 10 ns, es decir una frecuencia de 100MHz. Esta velocidad nos daría un fallo en nuestra implementación pero las simulaciones nos permiten cualquier valor y como este es el que viene por defecto se ha decidido no modificarlo.

Las letras U representan señales sin un valor definido, porque todavía no han recibido sus entradas. Si la letra que aparece es una X significará que hay un valor erróneo, esto aparece porque alguna de las señales ya ha llegado y otra u otras todavía se encuentran en U, pero no supone ningún problema ya que cuando la primera instrucción llegue a su altura ya dispondrán de los valores correctos para operar y nunca nos volverá a mostrar una señal de error.

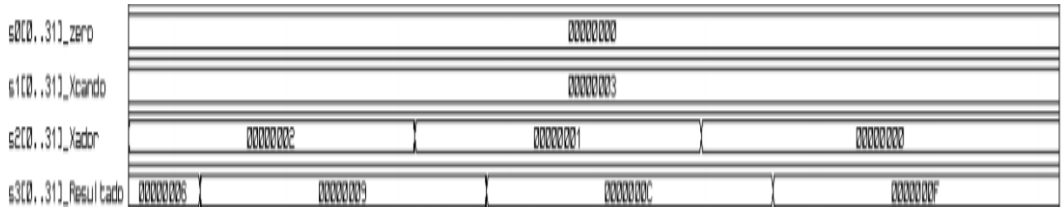

Ilustración 26. Resultados teóricos 1 («Material de la asignatura Electrónica Digital»)

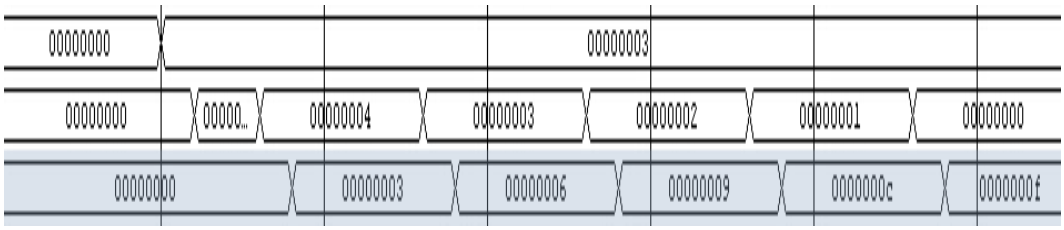

Ilustración 27. Resultados 1

Si recordamos el valor que nos debe proporcionar este programa y lo comparamos con las simulaciones ejemplo que se nos proporcionan en la asignatura de Electrónica Digital vemos que coinciden a la perfección y que, por tanto, funciona correctamente. Se escribe el valor 3 y 5 y este segundo se va disminuyendo una unidad cada vez que suma 3 al registro 19, en el que se almacenará el resultado. Al ponerse a 0 el registro 18 se realizará la última suma y comprobamos que en el registro S_3 se encuentra almacenado el valor "0000000f" o 15 (en decimal) que esperábamos.

PROGRAMA 2

El programa 2 explicado detalladamente en la página 49 del presente trabajo pretendía realizar la misma tarea que el anterior pero de manera diferente. El procedimiento en esta ocasión consiste en ir comprobando el dígito que ocupa la posición menos significativa del multiplicador. Si este es 1 debemos sumar el multiplicando a una variable, resultado, que contiene las sumas parciales. Si es cero no añadimos nada a dicha variable. Para comprobar si el dígito menos significativo del multiplicador es 1 o 0 utilizamos una máscara que tiene un 1 en el dígito menos significativo y un cero en el resto de los dígitos.

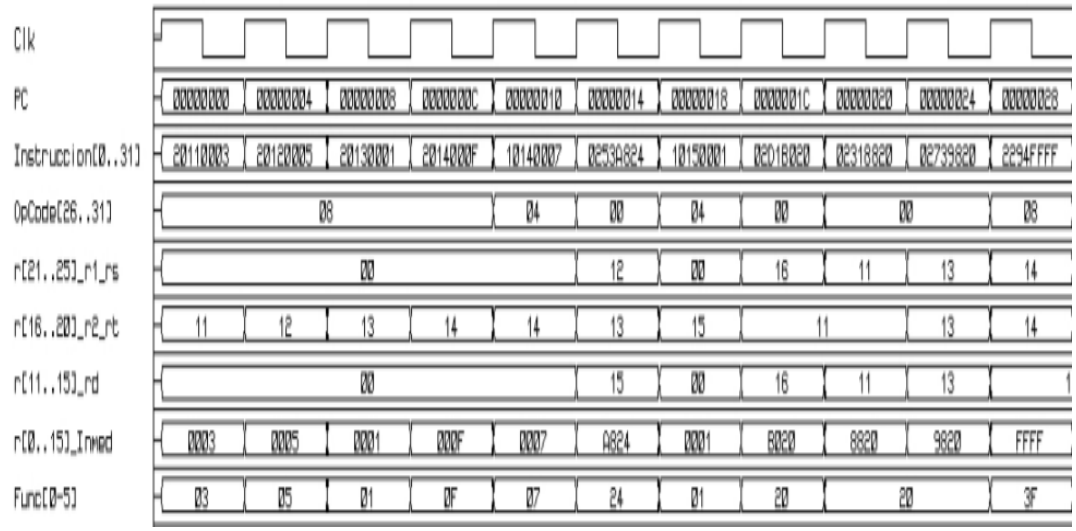

Ilustración 28. Simulación teórica 2 («Material de la asignatura Electrónica Digital»)

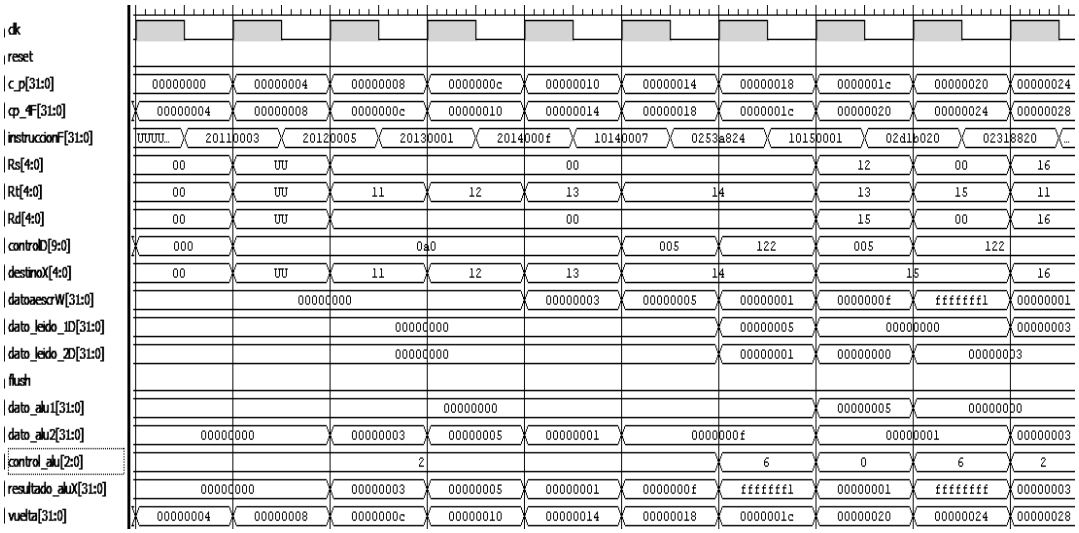

Ilustración 29. Simulación Programa 2-1

Volvemos a mostrar el inicio del programa y su desarrollo tras haber realizado un ciclo de reloj con el reset y dejarlo luego en valor constante 0. Podemos comparar con el ejemplo que disponemos como en el programa anterior y vemos que concuerdan a la perfección.

Como el programa 2 es el que más instrucciones ejecuta de todos los realizados se añadirá otra imagen que muestre el momento en el que las señales adquieren sus valores finales, dando por concluido el programa.

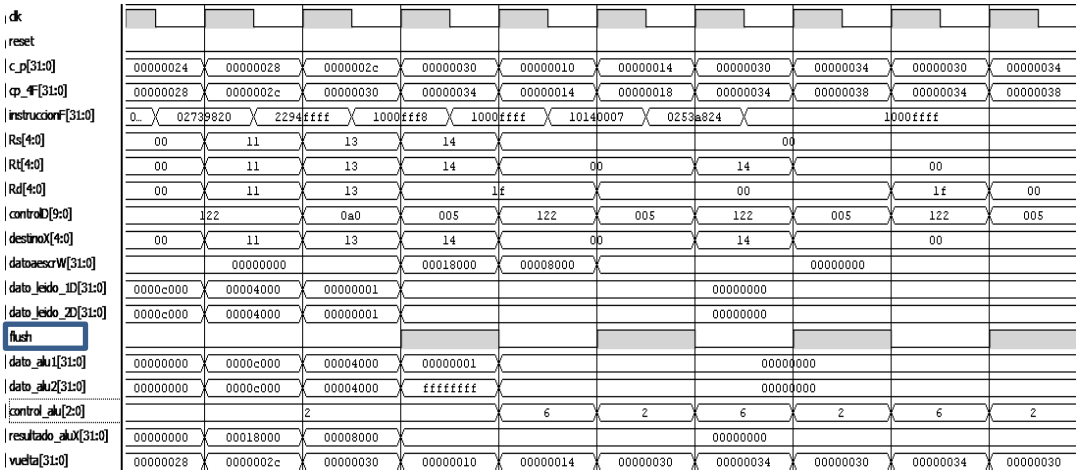

Ilustración 30. Simulación Programa 2-2

Como ya explicado en los apartados de programación la decisión de salto se toma en la etapa de decodificación por lo que la instrucción anterior a un salto tomado se debe anular, es decir, no debe afectar de ninguna manera al programa. La señal *flush* dentro del recuadro azul representa este hecho, ya que se pone a 1 cada vez que un salto se ha realizado, haciendo que la instrucción que está en esos momentos en la etapa de búsqueda aparezca como una NOP en el siguiente ciclo y así poder seguir desarrolando el programa de la manera correcta.

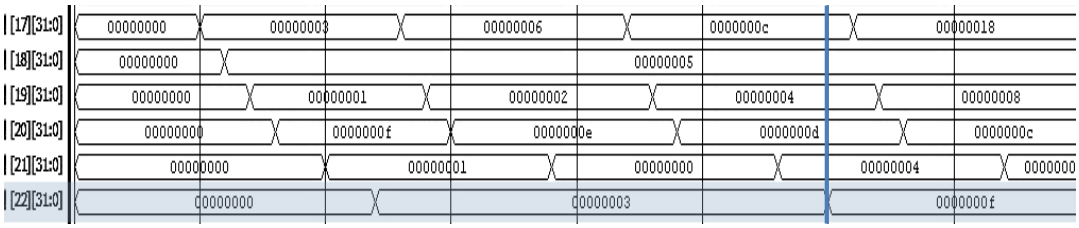

Ilustración 31. Resultados 2-1

La línea vertical azul indica el momento en el que la máscara añade por suma por segunda y última vez el valor que se encuentra en el registro 22 con el del 17 para almacenarlo en el mismo registro 22. Por tanto en este programa se obtiene el valor deseado muy pronto y este permanecerá hasta el final del programa sin sufrir ninguna modificación.

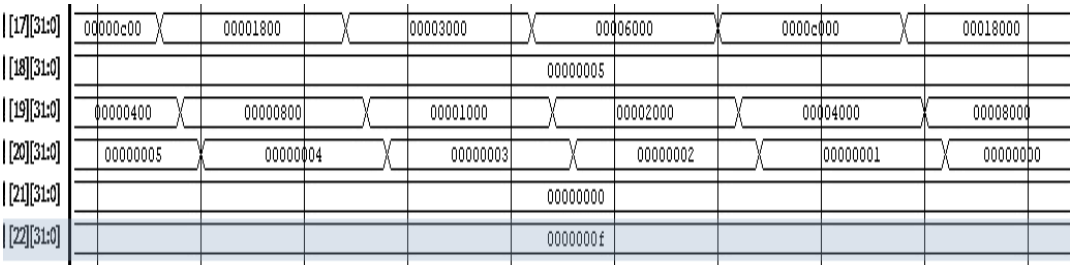

Ilustración 32. Resultados 2-2

Como acabamos de decir el programa tiene que seguir avanzando, haciendo hasta 15 ciclos de varias instrucciones, análogo a un *FOR*. Además de conseguir el resultado final correcto el resto de valores finales de cada uno de los registros son idénticos a los teóricos. En esta ocasión no se ha adjuntado la imagen correspondiente porque no se apreciaban los valores con la suficiente claridad.

PROGRAMA 3

El programa 3 se encuentra explicado en la página 50 de este trabajo dentro del apartado programas de prueba. A grandes rasgos consiste en almacenar los valores 3 y 5 en los primeros registros S como se hizo en los programas anteriores. A continuación, se opera con esos valores para realizar todas y cada una de las instrucciones que todavía no se han probado en los programas anteriores. Los resultados de estas operaciones se irán almacenando progresivamente en los registros desde S_3 a S_7.

Una vez más comenzaremos mostrando el inicio de programa para ver el correcto arranque y propagación de los valores de todas las señales elegidas. En esta ocasión ya no se dispone de material teórico con el que comparar por lo que se tendrá que conocer más si cabe el desempeño del programa, ya que para verificar el correcto funcionamiento de este habrá que seguir una a una todas las señales disponibles, sabiendo cual es el resultado esperado y corroborando que concuerda con el obtenido.

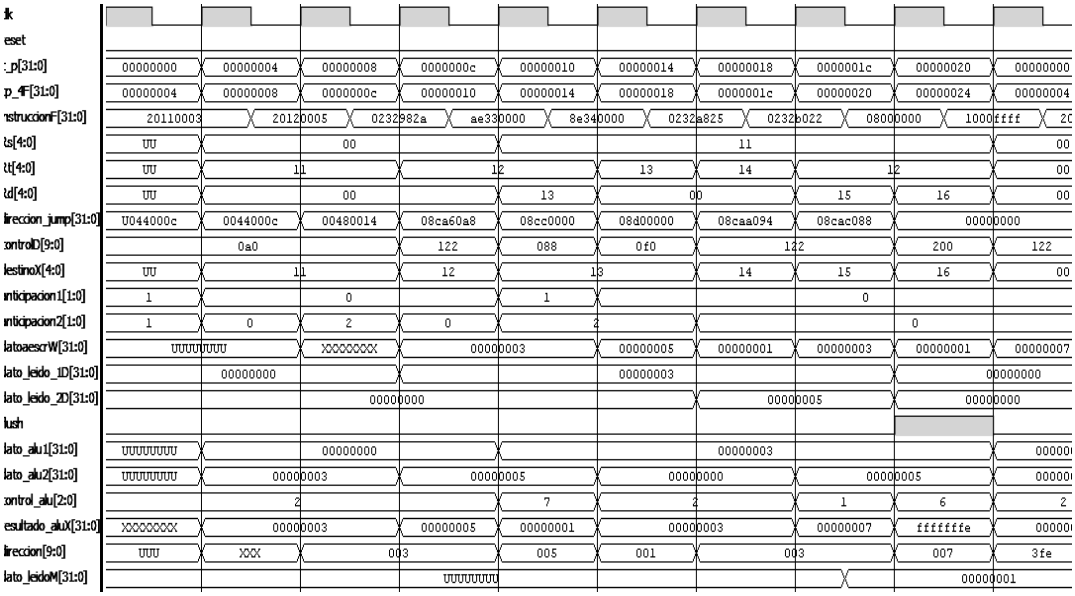

Ilustración 33. Simulación Programa 3

Para finalizar la simulación de este sencillo programa se mostrarán los valores de los registros y direcciones de memoria relevantes. Estas señales se denominan internas y no aparecen en la simulación principal. Será necesario añadirlas una vez se arranca el simulador, yendo al componente correspondiente y seleccionándolas. De esta manera cuando el tiempo de simulación avance apareceran junto a las señales del programa principal.

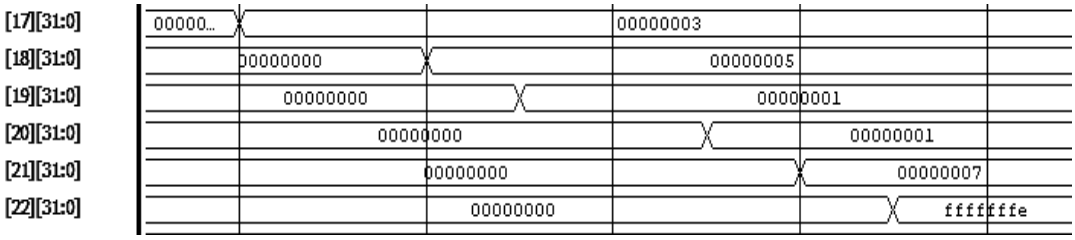

Ilustración 34. Resultados 3-1

Ilustración 35. Resultados 3-2

Tanto en el banco de registros como en la memoria de datos se obtienen los valores deseados en el orden correcto. Al igual que en programas anteriores se marca en azul la señal más relevante y su valor.

PROGRAMA 4

El programa 4 explicado en la página 69 de esta memoria trata de comprobar el correcto funcionamiento del único riesgo que todavía no se ha comprobado, una instrucción *lw* que escriba en el registro del cual va a leer la instrucción posterior a ella. En esta ocasión se ha elegido mostrar el programa y la simulación con *sw* como instrucción siguiente para intentar hacerlo más gráfico pero en la práctica también se ha verificado con una señal *addi* y con las señales de tipo R.

En primer lugar se almacena el valor 3 y 5 en los registros 17 y 18 respectivamente. Posteriormente la instrucción *sw* almacena en la dirección 0x0000000a (valor del registro 18 + valor inmediato 5) el valor que está almacenado en el registro 17, es decir, 3. A continuación *lw* lee de la memoria de datos el valor recientemente almacenado para guardarlo en el registro 16. Finalmente, este valor es leído por otra instrucción *sw* para almacenarlo en la dirección posterior a la que lo hizo la primera *sw*, que será la dirección 0x0000000b (valor del registro 18 + valor inmediato 6).

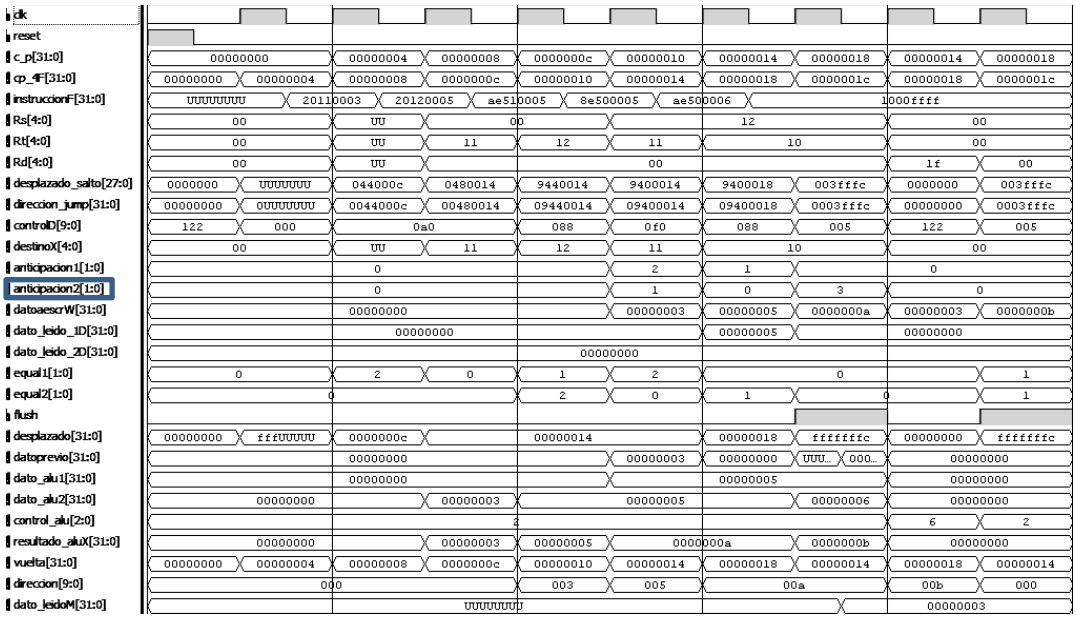

Ilustración 36. Simulación Programa 4

Este cuarto programa es el más breve y específico por lo que se ha podido capturar integramente en una imagen, siendo bastante sencilla su comprensión. Por este motivo se ha añadido alguna señal extra, como pueden ser las señales que equal 1 y 2 que controlan los multiplexores a la entrada del componente igualdad, o también los valores anticipación 1 y 2 los cuales controlan los multiplexores previos a la ALU (sin contar el dirigido por la señal de control FuenteALU). Todas estas señales también aparecen en el resto de programas y adoptan los valores correctos pero se ha entendido que su inclusión con el resto de señales podría restar claridad a la comprensión.

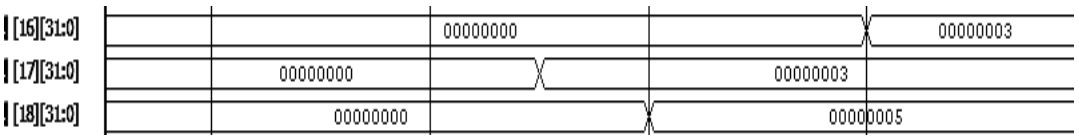

Ilustración 37. Resultados 4-1

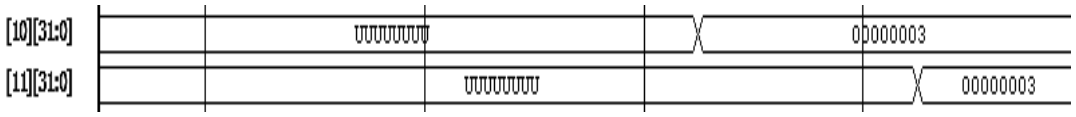

Ilustración 38. Resultados 4-2

La primera imagen referente a los resultados muestra los registros relevantes dentro del banco de registros. La segunda por su parte nos refleja el cambio de valor en las direcciones significativas dentro de la memoria de datos. Si se quisiera hacer desaparecer las señales U de la memoria de datos habría que realizar una inicialización de la misma manera que se realizó para el banco de registros, aunque no supone ningún beneficio práctico.

Se ha podido comprobar que el riesgo asociado a la combinación *lw* seguida de una instrucción de lectura ha sido corregido correctamente sin necesidad de la unidad de riesgos, simplemente añadiendo otra señal de anticipación en la unidad del mismo nombre (se aprecia en el momento en que la señal anticipación 2 marcada en azul adopta el valor 3).

Capítulo 6. Conclusiones

6.1 Cumplimiento de los objetivos

Se procede a repasar cada uno de los objetivos marcados al principio de este trabajo para verificar uno a uno si se han cumplido o no.

Adquirir conocimientos de programación en VHDL: La familiarización con el entorno de Vivado ha sido total, adquiriendo un nivel notable de programación en VHDL. Tras un periodo de aprendizaje realizando los componentes básicos de la asignatura Electrónica Digital se adquirió el nivel necesario para comenzar este trabajo y durante la realización del mismo se ha incrementado dicho nivel.

Diseñar y programar módulos funcionales independientes: Se ha logrado que todos los componentes y módulos funcionen correcta e individualmente y siendo lo más fieles posible a los teóricos utilizados como referencia. Por tanto, todo ese material podrá ser utilizado en el futuro, tanto por el programador como por todo aquel que lo desee, sin necesidad de conocer aspectos específicos e internos sobre ellos, es decir, bastando con lo aprendido teóricamente.

Diseñar y programar el MIPS monociclo: La parte de diseño no se ha realizado porque se tomó como referencia un esquema ya realizado del cual se conocía su funcionamiento a la perfección. Para la segunda parte del objetivo, la programación, se ha conseguido obtener un microprocesador MIPS y RISC monociclo que funciona perfectamente para todo su repertorio de instrucciones siempre que las memorias que utilice sean síncronas o que sean asíncronas de un tamaño suficientemente pequeño para poder implementarlas con registros.

Realizar diseños digitales orientados a FPGAs: El microprocesador realizado finalmente se integra a la perfección con una estructura típica FPGA y en este caso en concreto con la placa Basys 3 y sus componentes. Todos y cada uno de los elementos son identificados correctamente por el programa, utilizando los componentes que deseábamos una vez programada la placa. Además se ha logrado una buena disposición de los mismos sin ningún gasto desproporcionado y con una implementación final perfecta.

Diseñar y programar el MIPS segmentado (*pipeline*): Se ha realizado a partir de los componentes y la estructura base del MIPS básico monociclo un microprocesador segmentado o *pipeline* también de tipo MIPS y RISC que realiza correctamente las instrucciones de su repertorio, sea cual sea su orden. La ejecución se realiza en 5 etapas o *stages* y nos da un resultado perfecto para una velocidad de 70Mhz. Este microprocesador podrá estar implementado dentro de un sistema mayor u operar en solitario.

6.2. Posibles mejoras

La alternativa empleada para superar el problema de la implementación puede considerarse como una mejora en sí, ya que aunque trabaje a menor velocidad que su homólogo asíncrono nos permite tener un microprocesador segmentado perfectamente eficiente en la placa Basys 3, como deseábamos en un principio.

Como ya se ha indicado si se quiere representar de manera íntegra el esquema utilizado como referencia e implementarlo en la Basys 3 deberemos reducir el tamaño de las memorias. Esta "mejora" solo sería tal desde el punto de vista teórico, es decir, si queremos implementar el microprocesador monociclo o segmentado desde un carácter didáctico y no funcional.

Otra posible mejora sería intentar que el microprocesador no tuviera ningún tipo de interrupción, es decir, que siempre fuese ejecutando la instrucción correcta sin necesidad de ningún tipo de señal NOP. Esto se puede conseguir si el contador de programa trabaja en flanco de bajada, y a continuación tiene una ROM asíncrona o una ROM que trabaje en paralelo con él. De esta manera en la etapa de decodificación se realizaría la decisión del salto, (necesitaríamos los valores leídos del banco de registros al principio del ciclo) que llegaría en medio ciclo a la entrada del contador de programa, y a partir de ahí tendríamos otro medio ciclo para obtener la instrucción correspondiente a partir de la salida dada por este componente y almacenarla en el registro IF/ID. Como es lógico esta medida nos haría perder velocidad de funcionamiento, porque necesitamos un ciclo de ejecución mayor. Se debería cambiar un componente básico como es el contador de programa y como ya se ha repetido a lo largo del presente trabajo se ha querido mantener los componentes iniciales siempre que ha sido posible. Además el programa tendría una estructura híbrida o mixta entre el sincronismo y el asincronismo, algo que se ha intentado evitar a toda costa.

6.3. Líneas futuras

La evolución natural de este trabajo estará enfocada a la integración de las excepciones dentro de nuestro microprocesador, tanto a nivel de Hardware como de Software. Siguiendo este camino se llegará posteriormente a la predicción o planificación dinámica del microprocesador segmentado construyendo lo que se conoce como el MIPS segmentado superescalar. Si se quiere mantener la línea de trabajo que se ha seguido para este trabajo se recomienda que el libro de referencia para estas evoluciones será *Arquitectura de Computadores: Un enfoque Cuantitativo*, de Hennessy y Patterson.

Capítulo 7. Referencias

7.1. Bibliografía

Sergio Barrachina Mir. (2006). *Diseño del procesador MIPS R2000* (Didáctico). Jaume I, Castellón de la Plana. Recuperado a partir de http://www.uv.es/serhocal/docs/7_disenyo_procesador.pdf

Patterson, D. A., & Hennessy, J. L. (2000). *Estructura y diseño de computadores. (1ed) Volumen 2*. (E. E. i Vila, Trad.) (Edición: 1). Barcelona: Editorial Reverte.

Patterson, D. A., & Hennessy, J. L. (2014). *Computer organization and design: the hardware/software interface* (Fifth edition). Amsterdam ; Boston: Elsevier/Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier.

Material de la asignatura Electrónica Digital. (s. f.). Recuperado a partir de https://campusvirtual.uva.es/

7.2. Webs de consulta

Alma Celeste Flores Martínez. (s. f.). Arquitectura de computadoras. Recuperado a partir de https://is603arquicom2016.wordpress.com/

Álvaro Padierna Díaz. (2009, junio). *Microprocesador sintetizable en FPGA* (Trabajo Fin de Grado). Pontificia de Comillas, Madrid. Recuperado a partir de https://www.iit.comillas.edu/pfc/resumenes/4a4635e5a6b76.pdf

Arquitectura MIPS. (2016, mayo 20). Recuperado a partir de https://is603arquicom2016.wordpress.com/acerca-de/

Basys 3 Artix-7 FPGA Board - Lógica Programable. (s. f.). Recuperado a partir de https://sites.google.com/site/logicaprogramable/calculadoras/fpga/basys-3-artix-7-fpga-board

Electrical Engineering Store, FPGA, Microcontrollers and Instrumentation | Digilent. (s. f.). Recuperado a partir de http://store.digilentinc.com/

Universidad Tecnológica de la Mixteca. (s. f.). Curso: Arquitectura de Computadoras. Recuperado a partir de http://mixteco.utm.mx/~merg/AC/

VHDL. (2017, abril 12). En *Wikipedia, la enciclopedia libre*. Recuperado a partir de https://es.wikipedia.org/w/index.php?title=VHDL&oldid=98295111

Xilinx Vivado. (2017, junio 22). En *Wikipedia*. Recuperado a partir de https://en.wikipedia.org/w/index.php?title=Xilinx_Vivado&oldid=786885410

Anexos

1. MIPS Monociclo

MICROPROCESADOR MONOCICLO

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

entity monociclo is
Generic(N : natural := 32; M : natural := 3);
    Port (
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          opt: in STD_LOGIC_VECTOR (2 downto 0);
          muestra : out STD_LOGIC_VECTOR (15 downto 0)
          );
end monociclo;

architecture arch_monociclo of monociclo is

  component cp
     PORT(
          clk,rst : in STD_LOGIC;
          entrada : in STD_LOGIC_VECTOR (31 downto 0);
          sal1, sal2 : out STD_LOGIC_VECTOR (31 downto 0)
         );
  end component;

  component ROM
     PORT(
           direc : in STD_LOGIC_VECTOR (9 downto 0);
           instruccion : out STD_LOGIC_VECTOR (31 downto 0)
         );
  end component;

  component desplazador_j
     Port (
           instruc : in STD_LOGIC_VECTOR (25 downto 0);
           desplazado : out STD_LOGIC_VECTOR (27 downto 0)
           );
  end component;

  component UC

     Port (
           tipo : in STD_LOGIC_VECTOR (5 downto 0);
           salida : out STD_LOGIC_VECTOR (9 downto 0)
           );

  end component;

  component banco_reg
       PORT(
           clk, rst : in STD_LOGIC;
           wr : in STD_LOGIC;
           reg_lectura_1 : in STD_LOGIC_VECTOR (4 downto 0);
           reg_lectura_2 : in STD_LOGIC_VECTOR (4 downto 0);
```

```
          reg_escritura : in STD_LOGIC_VECTOR (4 downto 0);
          dato_escrito : in STD_LOGIC_VECTOR (31 downto 0);
          dato1 : out STD_LOGIC_VECTOR (31 downto 0);
          dato2 : out STD_LOGIC_VECTOR (31 downto 0)
         );
  end component;

component ext
     Port (
          A : in STD_LOGIC_VECTOR (15 downto 0);
          B : out STD_LOGIC_VECTOR (31 downto 0)
         );
end component;

component multi
   Port (
          I0,I1: in STD_LOGIC_VECTOR (31 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(31 downto 0) );
end component;

component desplazador
   Port (
          extensor : in STD_LOGIC_VECTOR (31 downto 0);
          desplazado : out STD_LOGIC_VECTOR (31 downto 0)
          );
end component;

component multi2
   Port(
          I0,I1: in STD_LOGIC_VECTOR (4 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(4 downto 0)
          );

end component;

component CA
   Port (
          ALUOp : in STD_LOGIC_VECTOR (1 downto 0);
          funcion : in STD_LOGIC_VECTOR (5 downto 0);
          resultado : out STD_LOGIC_VECTOR (2 downto 0)
          );
end component;

component alu
  Port (
         A : in STD_LOGIC_VECTOR (N-1 downto 0);
         B : in STD_LOGIC_VECTOR (N-1 downto 0);
         sel : in STD_LOGIC_VECTOR (M-1 downto 0);
         zero: out STD_LOGIC;
         resultado: inout STD_LOGIC_VECTOR(N-1 downto 0)
         );
end component;

component sumador
  Port (
          cp4 : in STD_LOGIC_VECTOR (31 downto 0);
          desplazado : in STD_LOGIC_VECTOR (31 downto 0);
```

```
        salida : out STD_LOGIC_VECTOR (31 downto 0)
        );
end component;

component multisalto
   Port (
        I0,I1,I2: in STD_LOGIC_VECTOR (31 downto 0);
        c : in STD_LOGIC_VECTOR (1 downto 0);
        s : out STD_LOGIC_VECTOR(31 downto 0)
        );
end component;

component ram2
   Port (
        EscrMem : in std_logic;
        LeerMem: in std_logic;
        radd : in std_logic_vector(9 downto 0);
        wadd : in std_logic_vector(9 downto 0);
        data_in : in std_logic_vector(31 downto 0);
        data_out : out std_logic_vector(31 downto 0)
       );
end component;


signal c_p: std_logic_vector(31 downto 0); --salida contador de programa
signal cp_4: std_logic_vector(31 downto 0); --salida primer sumador
signal instruccion: std_logic_vector(31 downto 0); --salida de la ROM
signal desplazado_salto: std_logic_vector(27 downto 0);
signal direccion_jump: std_logic_vector (31 downto 0); --dirección salto
signal control: std_logic_vector(9 downto 0);
signal extendido: std_logic_vector(31 downto 0);
signal destino: std_logic_vector(4 downto 0); --registro de escritura
signal datoaescr: std_logic_vector(31 downto 0); --valor a escribir
signal dato_leido_1: std_logic_vector(31 downto 0); --lectura 1 banco registros
signal desplazado: std_logic_vector (31 downto 0);
signal dato_leido_2: std_logic_vector(31 downto 0); --lectura 2 banco registros
signal dato_alu: std_logic_vector(31 downto 0); --entrada 2 puerto ALU
signal control_alu: std_logic_vector(2 downto 0);
signal cero: std_logic;
signal resultado_alu: std_logic_vector(31 downto 0);
signal salida_sumador: std_logic_vector(31 downto 0);
signal Fuente_PC: std_logic;
signal Seleccion_salto: std_logic_vector(1 downto 0); --control siguiente instrucción
signal vuelta: std_logic_vector(31 downto 0); --siguiente instrucción
signal direccion: std_logic_vector(9 downto 0);
signal dato_leido: std_logic_vector(31 downto 0); --lectura de la RAM

begin

PASO1: cp PORT MAP (clk => clk, rst=>reset, entrada => vuelta, sal1 => c_p, sal2 => cp_4);
PASO2: ROM PORT MAP (clk => clk, direc=> c_p(11 downto 2), instruccion => instruccion);
PASO3: desplazador_j PORT MAP(instruc=>instruccion(25 downto 0),
desplazado=>desplazado_salto);
direccion_jump <= cp_4(31 downto 28) & desplazado_salto;
PASO4: UC PORT MAP(tipo=>instruccion(31 downto 26), salida=>control);
PASO5: multi2 PORT MAP (I0=>instruccion(20 downto 16), I1=>instruccion(15 downto 11),
c=>control(8), s=>destino);
PASO6: banco_reg PORT MAP (clk=>clk, rst=>reset, wr=>control(5),
reg_lectura_1=>instruccion(25 downto 21), reg_lectura_2 => instruccion(20 downto 16),
```

```
reg_escritura=> destino, dato_escrito=> datoaescr, dato1 => dato_leido_1,
dato2 => dato_leido_2);
PASO7: ext PORT MAP(A=>instruccion(15 downto 0),B=>extendido);
PASO8: desplazador PORT MAP(extensor=>extendido, desplazado=>desplazado);
PASO9: multi PORT MAP(I0=>dato_leido_2, I1=>extendido, c=>control(7), s=>dato_alu);
PASO10: CA PORT MAP(ALUOp=>control(1 downto 0), funcion=>instruccion(5 downto 0),
resultado=>control_alu);
PASO11: ALU PORT MAP(A=>dato_leido_1, B=>dato_alu, sel=>control_alu, zero=>cero,
resultado=>resultado_alu);
PASO12: sumador PORT MAP(cp4=>cp_4, desplazado=>desplazado,
salida=>salida_sumador);
Fuente_PC <= control(2) AND cero;
Seleccion_salto <= Fuente_PC & control(9);
PASO13: multisalto PORT MAP(I0=>cp_4, I1=>direccion_jump, I2=>salida_sumador,
c=>Seleccion_salto, s=>vuelta);
direccion<=resultado_alu(9 downto 0);
PASO14: ram2 PORT MAP(EscrMem=>control(3), LeerMem=>control(4),
radd=>dirección,wadd=>direccion, data_in=>dato_leido_2, data_out=> dato_leido);
PASO15: multi PORT MAP(I0=>resultado_alu, I1=>dato_leido, c=>control(6), s=>datoaescr);

   with opt select
      muestra <= instruccion(15 downto 0) when "000",
            instruccion(31 downto 16) when "001",
            dato_leido(15 downto 0) when "010",
            dato_leido(31 downto 16) when "011",
            "000000" & control when others;

end arch_monociclo;
```

2. Elementos comunes

CONTADOR DE PROGRAMA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity cp is
    Port ( clk, rst: in STD_LOGIC;
           entrada: in STD_LOGIC_VECTOR (31 downto 0);
           sal1, sal2 : out STD_LOGIC_VECTOR (31 downto 0)--cp y cp4
           );
end cp;
architecture arch_cp of cp is
begin
process (clk, rst, entrada)
begin
    if (rst = '1') then
       sal1 <= (others=>'0');
       sal2 <= (others=>'0');
    elsif(clk'event AND clk = '1') then
          if(entrada = "11111111111111111111111111111111") then
             sal1 <= (others => '0');
             sal2 <= ("00000000000000000000000000000100");
          else
             sal1 <= entrada;
             sal2 <= entrada + 4;
          end if;
    end if;
end process;
```

ROM ASÍNCRONA

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity ROM is
   Port (
        direc : in STD_LOGIC_VECTOR (9 downto 0);
        instruccion : out STD_LOGIC_VECTOR (31 downto 0));
end ROM;
architecture arch_ROM of ROM is
signal aux: std_logic_vector(9 downto 0);
TYPE rom1 is ARRAY (0 to 1023) of STD_LOGIC_VECTOR (31 downto 0);
signal memoria : rom1:=(

--                  PARTE COMUN
                    x"20110003",
                    x"20120005",

--                   PROGRAMA1
                    x"12400003",
                    x"2252FFFF",
                    x"02719820",
                    x"1000FFFC",


--                   PROGRAMA2
                    x"20130001",
                    x"2014000F",
                    x"10140007",
                    x"0253A824",
                    x"10150001",
                    x"02D1B020",
                    x"02318820",
                    x"02739820",
                    x"2294FFFF",
                    x"1000FFF8",

--                   PROGRAMA3
                    x"0232982A",
                    x"AE330000",
                    x"8E340000",
                    x"0232A825",
                    x"0232B022",
                    x"08000000",
             others =>x"1000FFFF" );--bucle final comun
begin

process(direc)
begin
            instruccion <= memoria(conv_integer(direc));
end process;
end arch_ROM;
```

DESPLAZADOR SALTO

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

entity desplazador_j is
   Port ( instruc : in STD_LOGIC_VECTOR (25 downto 0);
          desplazado : out STD_LOGIC_VECTOR (27 downto 0));
end desplazador_j;

architecture arch_desplazador_j of desplazador_j is


begin

   desplazado <= instruc & "00";

end arch_desplazador_j;
```

UNIDAD DE CONTROL

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

entity UC is
   Port ( tipo : in STD_LOGIC_VECTOR (5 downto 0);
          salida : out STD_LOGIC_VECTOR (9 downto 0));
end UC;

architecture arch_UC of UC is
begin

   with tipo select
         salida <= "0100100010" when "000000", --Tipo R
                   "0011110000" when "100011", --LW
                   "0010001000" when "101011", --SW
                   "0000000101" when "000100", --BEQ
                   "0010100000" when "001000", --ADDI
                   "1000000000" when "000010", --J
                   "0000000000" when others;

end arch_UC;
```

EXTENSOR DE SIGNO

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

entity ext is
   Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
          B : out STD_LOGIC_VECTOR (31 downto 0));
end ext;

architecture arch_ext of ext is
signal aux: std_logic_vector(31 downto 0);
begin
process(A)

begin
   if (A(15) = '0') then
     aux <= "0000000000000000" & A(15 downto 0);
   else
     aux <= "1111111111111111" & A(15 downto 0);
   end if;

end process;

B <= aux;

end arch_ext;
```

DESPLAZADOR 31 BITS

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

entity desplazador is
   Port ( extensor : in STD_LOGIC_VECTOR (31 downto 0);
          desplazado : out STD_LOGIC_VECTOR (31 downto 0));
end desplazador;

architecture arch_desplazador of desplazador is

signal aux: std_logic_vector(33 downto 0);

begin

       aux<= extensor & "00";
       desplazado <= aux(31 downto 0);

end arch_desplazador;
```

BANCO DE REGISTROS

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity banco_reg is
   Port ( clk,rst : in STD_LOGIC;
          wr : in STD_LOGIC;
          reg_lectura_1 : in STD_LOGIC_VECTOR (4 downto 0);
          reg_lectura_2 : in STD_LOGIC_VECTOR (4 downto 0);
          reg_escritura : in STD_LOGIC_VECTOR (4 downto 0);
          dato_escrito : in STD_LOGIC_VECTOR (31 downto 0);
          dato1 : out STD_LOGIC_VECTOR (31 downto 0);
          dato2 : out STD_LOGIC_VECTOR (31 downto 0));
end banco_reg;

architecture arch_banco_reg of banco_reg is

TYPE banco is ARRAY (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
signal registro : banco:= (others => "00000000000000000000000000000000");

begin


process(clk, rst, wr, registro, reg_escritura, dato_escrito, reg_lectura_1, reg_lectura_2,
dato_escrito)


begin


   if rst /= '0' then

      registro <= (others => "00000000000000000000000000000000");

   elsif clk ='1' and wr = '1' and  reg_escritura /= "00000" then

      registro(conv_integer(reg_escritura)) <= dato_escrito;

   end if;

  dato1 <= registro(conv_integer(reg_lectura_1));
  dato2 <= registro(conv_integer(reg_lectura_2));


end process;

end arch_banco_reg;
```

MULTIPLEXOR 2 A 1

```
entity multi is
   Port ( I0,I1: in STD_LOGIC_VECTOR (31 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(31 downto 0));
end multi;

architecture multi_arch of multi is

begin
   with c select
      s <= I0 when '0',
           I1 when others;
end multi_arch;
```

MULTIPLEXOR 2 A 1 (5 bits)

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

entity multi2 is
   Port ( I0,I1: in STD_LOGIC_VECTOR (4 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(4 downto 0));
end multi2;

architecture arch_multi2 of multi2 is

begin
   with c select
      s <= I0 when '0',
           I1 when others;
end arch_multi2;
```

MULTIPLEXOR 3 A 1

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

entity multisalto is
   Port ( I0,I1,I2: in STD_LOGIC_VECTOR (31 downto 0);
          c : in STD_LOGIC_VECTOR (1 downto 0);
          s : out STD_LOGIC_VECTOR(31 downto 0));
end multisalto;

architecture arch_multisalto of multisalto is

begin
   with c select
      s <= I0 when "00",
           I2 when "10",
           I1 when others; --Se necesita por sintaxis pero no habrá nunca “11”

end arch_multisalto;
```

CONTROL DE LA ALU

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

entity CA is
   Port ( ALUOp : in STD_LOGIC_VECTOR (1 downto 0);
          funcion : in STD_LOGIC_VECTOR (5 downto 0);
          resultado : out STD_LOGIC_VECTOR (2 downto 0));
end CA;

architecture arch_CA of CA is
begin
   process(ALUOp, funcion)
      begin
      case aluop is
         when "00" => resultado <= "010"; -- lw, sw y addi
         when "01" => resultado <= "110"; -- beq
         when others => case funcion(3 downto 0) is      -- R-type
            when "0000" => resultado <= "010";
            when "0010" => resultado <= "110";
            when "0100" => resultado <= "000";
            when "0101" => resultado <= "001";
            when "1010" => resultado <= "111";
            when others => resultado <= "---";
         end case;
      end case;
   end process;
end arch_CA;
```

ALU

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

entity alu is
Generic(N : natural := 32; M : natural := 3);
   Port ( A : in STD_LOGIC_VECTOR (N-1 downto 0);
          B : in STD_LOGIC_VECTOR (N-1 downto 0);
          sel : in STD_LOGIC_VECTOR (M-1 downto 0);
--          zero: out STD_LOGIC;
          resultado: inout STD_LOGIC_VECTOR(N-1 downto 0)
          );
end alu;

architecture arch_alu of alu is

begin
process (sel,A,B)

      Variable solucion : std_logic_vector(N-1 downto 0); --Si no quiero ver desbordamiento N-1
      Variable auxiliar : std_logic_vector(M-1 downto 0);

      begin

         --INICIALIZACION
         solucion := "00000000000000000000000000000000";
         auxiliar := sel;

         case auxiliar is
            when "000" => solucion := (A AND B);
            when "001" => solucion := (A OR B);
            when "010" => solucion := (A + B);
            when "110" => solucion := (A - B);
            when "111" =>
               if (A < B) then
                  solucion := "00000000000000000000000000000001";--slt
               else
                  solucion := solucion;
               end if;
            when others => NULL;
         end case;

         -- Comprobamos si el resultado es 0
         if solucion = 0 then
            Zero <= '1';
         else
            Zero <= '0';
         end if;

          -- Resultado final
          resultado <= solucion;
      end process;

end arch_alu;
```

SUMADOR

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

entity sumador is
    Port (
            cp4 : in STD_LOGIC_VECTOR (31 downto 0);
          desplazado : in STD_LOGIC_VECTOR (31 downto 0);
         salida : out STD_LOGIC_VECTOR (31 downto 0));
end sumador;

architecture arch_sumador of sumador is

begin

    salida <= cp4 + desplazado;

end arch_sumador;
```

RAM ASÍNCRONA

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


entity ram2 is
Port (
      EscrMem : in std_logic;
      LeerMem: in std_logic;
      radd : in std_logic_vector(9 downto 0);
      wadd : in std_logic_vector(9 downto 0);
      data_in : in std_logic_vector(31 downto 0);
      data_out : out std_logic_vector(31 downto 0)
   );
end ram2;


architecture arch_ram2 of ram2 is

type ram is array(0 to 1023) of std_logic_vector(31 downto 0);
signal ram1_1 : ram;


begin


process(data_in, wadd, EscrMem)
begin
      if EscrMem = '1' then
         ram1_1(conv_integer(wadd)) <= data_in;
      end if;

end process;

process(ram1_1, LeerMem, radd)
begin

      if LeerMem = '1' then
         data_out <= ram1_1(conv_integer(radd));
      end if;

end process;
end arch_ram2;
```

3. MIPS segmentado con memorias asíncronas

MIPS SEGMENTADO ASÍNCRONO

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

entity segmentado is
Generic(N : natural := 32; M : natural := 3);
   Port (
         clk : in STD_LOGIC;
         reset : in STD_LOGIC;
         opt: in STD_LOGIC_VECTOR (2 downto 0);
         muestra : out STD_LOGIC_VECTOR (15 downto 0)
         );
end segmentado;

architecture arch_segmentado of segmentado is

component cp
     PORT(
         clk, rst, EscrPC : in STD_LOGIC;
         entrada : in STD_LOGIC_VECTOR (31 downto 0);
         sal1, sal2 : out STD_LOGIC_VECTOR (31 downto 0)
        );
 end component;

 component ROM
    PORT(

          direc : in STD_LOGIC_VECTOR (9 downto 0);
          instruccion : out STD_LOGIC_VECTOR (31 downto 0)
       );
 end component;

 component reg1
    Port (
         ent : in STD_LOGIC_VECTOR (63 downto 0);
         clk, EscrF, Flush : in std_logic;
         sal : out STD_LOGIC_VECTOR (63 downto 0)
         );
 end component;

 component UR
   Port (
        LeerMemX: in std_logic;
        RtX, RsD, RtD: in std_logic_vector(4 downto 0);
        EscrPC, EscrF, riesgo: out std_logic
         );
 end component;

 component desplazador_j
    Port (
          instruc : in STD_LOGIC_VECTOR (25 downto 0);
```

```
        desplazado : out STD_LOGIC_VECTOR (27 downto 0)
        );
end component;

component UC

   Port (
        tipo : in STD_LOGIC_VECTOR (5 downto 0);
        salida : out STD_LOGIC_VECTOR (9 downto 0)
        );

end component;

component UAE
   Port (
        Reg_escritura : in STD_LOGIC_VECTOR (4 downto 0);
        Reg_lectura1 : in STD_LOGIC_VECTOR (4 downto 0);
        Reg_lectura2 : in STD_LOGIC_VECTOR (4 downto 0);
        anticipacion3 : out STD_LOGIC;
        anticipacion4 : out STD_LOGIC
        );
end component;

component banco_reg
    PORT(
        clk, reset : in STD_LOGIC;
        wr : in STD_LOGIC;
        reg_lectura_1 : in STD_LOGIC_VECTOR (4 downto 0);
        reg_lectura_2 : in STD_LOGIC_VECTOR (4 downto 0);
        reg_escritura : in STD_LOGIC_VECTOR (4 downto 0);
        dato_escrito : in STD_LOGIC_VECTOR (31 downto 0);
        dato1 : out STD_LOGIC_VECTOR (31 downto 0);
        dato2 : out STD_LOGIC_VECTOR (31 downto 0)
       );
end component;

component beq
    Port (
         EscrRegX, EscrRegM: in STD_LOGIC;
         Rs, Rt, RdX, RdM: in STD_LOGIC_VECTOR(4 downto 0);
         control_beq1, control_beq2: out STD_LOGIC_VECTOR(1 downto 0)
        );
end component;

component igualdad
    Port (
         AluOP: in STD_LOGIC_VECTOR(1 downto 0);
         datoleido1, datoleido2: in STD_LOGIC_VECTOR (31 downto 0);
         zero : out STD_LOGIC
          );
end component;

component vaciado
      Port (
           Seleccion_salto: in STD_LOGIC_VECTOR(1 downto 0);
```

```
            flush: out STD_LOGIC
            );
end component;

component controlmux
    Port (
          Flush : in STD_LOGIC;
          riesgo : in STD_LOGIC;
          mux : out STD_LOGIC
          );
end component;

component multicontrol
    Port (
          I0: in STD_LOGIC_VECTOR (9 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(9 downto 0)
          );
end component;

component ext
     Port (
           A : in STD_LOGIC_VECTOR (15 downto 0);
           B : out STD_LOGIC_VECTOR (31 downto 0)
         );
end component;

component reg2
    Port (
         ent : in STD_LOGIC_VECTOR (120 downto 0);
         clk : in std_logic;
         sal : out STD_LOGIC_VECTOR (120 downto 0)
         );
end component;

component UA
        Port (
             EscrRegM, EscrRegW: in STD_LOGIC;
             Rs, Rt, RdM, RdW: in STD_LOGIC_VECTOR(4 downto 0);
             anticiparA, anticiparB: out STD_LOGIC_VECTOR(1 downto 0)
        );
end component;

component multi
    Port (
          I0,I1: in STD_LOGIC_VECTOR (31 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(31 downto 0) );
end component;

component desplazador
    Port (
         extensor : in STD_LOGIC_VECTOR (31 downto 0);
         desplazado : out STD_LOGIC_VECTOR (31 downto 0)
         );
```

```
end component;

component multi2
   Port(
        I0,I1: in STD_LOGIC_VECTOR (4 downto 0);
        c : in STD_LOGIC;
        s : out STD_LOGIC_VECTOR(4 downto 0)
        );

end component;

component CA
   Port (
        ALUOp : in STD_LOGIC_VECTOR (1 downto 0);
        funcion : in STD_LOGIC_VECTOR (5 downto 0);
        resultado : out STD_LOGIC_VECTOR (2 downto 0)
        );
end component;

component alu
  Port (
       A : in STD_LOGIC_VECTOR (N-1 downto 0);
       B : in STD_LOGIC_VECTOR (N-1 downto 0);
       sel : in STD_LOGIC_VECTOR (M-1 downto 0);
       resultado: inout STD_LOGIC_VECTOR(N-1 downto 0)
       );
end component;

component sumador
  Port (
        cp4 : in STD_LOGIC_VECTOR (31 downto 0);
        desplazado : in STD_LOGIC_VECTOR (31 downto 0);
        salida : out STD_LOGIC_VECTOR (31 downto 0)
        );
end component;

component reg3
   Port (
       ent : in STD_LOGIC_VECTOR (72 downto 0);
       clk : in std_logic;
       sal : out STD_LOGIC_VECTOR (72 downto 0)
       );
end component;

component multisalto
   Port (
        I0,I1,I2: in STD_LOGIC_VECTOR (31 downto 0);
        c : in STD_LOGIC_VECTOR (1 downto 0);
        s : out STD_LOGIC_VECTOR(31 downto 0)
        );
end component;

 component ram2
   Port (
        EscrMem : in std_logic;
```

```
        LeerMem: in std_logic;
        radd : in std_logic_vector(9 downto 0);
        wadd : in std_logic_vector(9 downto 0);
        data_in : in std_logic_vector(31 downto 0);
        data_out : out std_logic_vector(31 downto 0)
       );
end component;

component reg4
   Port (
       ent : in STD_LOGIC_VECTOR (70 downto 0);
       clk : in std_logic;
       sal : out STD_LOGIC_VECTOR (70 downto 0)
       );
end component;


signal c_p: std_logic_vector(31 downto 0);
signal cp_4F: std_logic_vector(31 downto 0);
signal cp_4D: std_logic_vector(31 downto 0);
signal instruccionF: std_logic_vector(31 downto 0);
signal instruccionD: std_logic_vector(31 downto 0);
signal Rs: std_logic_vector(4 downto 0);
signal Rt: std_logic_vector(4 downto 0);
signal Rd: std_logic_vector(4 downto 0);
signal desplazado_salto: std_logic_vector(27 downto 0);
signal direccion_jump: std_logic_vector(31 downto 0);
signal EscrPC: std_logic; -- señal de mantenimiento de señal CP
signal EscrF: std_logic; --señal de mantenimiento de señal registro IF/ID
signal riesgo: std_logic; --señal de control del mux, multiplexor de control
signal controlprevio: std_logic_vector(9 downto 0); --salida de UC previa al mux
signal controlD: std_logic_vector(9 downto 0);
signal controlX: std_logic_vector(9 downto 0);
signal controlM: std_logic_vector(3 downto 0);
signal controlW: std_logic_vector(1 downto 0);
signal dato_correcto1: std_logic_vector(31 downto 0); --dato leído 1 post multi
signal dato_correcto2: std_logic_vector(31 downto 0); --dato leído 2 post multi
signal dato1: std_logic_vector(31 downto 0); --valor anticipado 1 para beq
signal dato2: std_logic_vector(31 downto 0); --valor anticipado 2 para beq
signal extendidoX: std_logic_vector(31 downto 0);
signal extendidoD: std_logic_vector(31 downto 0);
signal destinoX: std_logic_vector(4 downto 0);
signal destinoM: std_logic_vector(4 downto 0);
signal destinoW: std_logic_vector(4 downto 0);
signal anticipacion1: std_logic_vector(1 downto 0); --anticipación ALU 1
signal anticipacion2: std_logic_vector(1 downto 0); --anticipación ALU2
signal anticipacion3: std_logic; --señal anticipación puerto lectura 1
signal anticipacion4: std_logic; --señal anticipación puerto lectura 2
signal datoaescrM: std_logic_vector(31 downto 0);
signal datoaescrW: std_logic_vector(31 downto 0);
signal dato_leido_1D: std_logic_vector(31 downto 0);
signal dato_leido_2D: std_logic_vector(31 downto 0);
signal dato_leido_1X: std_logic_vector(31 downto 0);
signal dato_leido_2X: std_logic_vector(31 downto 0);
signal equal1: std_logic_vector(1 downto 0); --anticipacion 1 para señal cero
```

```
  signal equal2: std_logic_vector(1 downto 0); --anticipacion 2 para señal cero
  signal flush: std_logic; --señal indicación salto tomado
  signal desplazado: std_logic_vector(31 downto 0);
  signal datoprevio: std_logic_vector(31 downto 0); --salida multi anticipacion 2
  signal dato_alu1: std_logic_vector(31 downto 0);
  signal dato_alu2: std_logic_vector(31 downto 0);
  signal control_alu: std_logic_vector(2 downto 0);
  signal cero: std_logic;
  signal resultado_aluX: std_logic_vector(31 downto 0);
  signal resultado_aluM: std_logic_vector(31 downto 0);
  signal resultado_aluW: std_logic_vector(31 downto 0);
  signal salida_sumador: std_logic_vector(31 downto 0);
  signal Fuente_PC: std_logic;
  signal Seleccion_salto: std_logic_vector(1 downto 0);
  signal vuelta: std_logic_vector(31 downto 0);
  signal direccion: std_logic_vector(9 downto 0);
  signal dato_leidoM: std_logic_vector(31 downto 0);
  signal dato_leidoW: std_logic_vector(31 downto 0);

begin

--ETAPA IF

PASO1: multisalto PORT MAP(I0=>cp_4F, I1=>direccion_jump, I2=>salida_sumador,
c=>Seleccion_salto, s=>vuelta);
PASO2: cp PORT MAP (clk=>clk, rst=>reset, EscrPC=>EscrPC, entrada=>vuelta, sal1=>c_p,
sal2=>cp_4F);
PASO3: ROM PORT MAP (direc=> c_p(11 downto 2), instruccion => instruccionF);
PASO4: reg1 PORT MAP (ent(63 downto 32)=>instruccionF, ent(31 downto 0)=>cp_4F,
clk=>clk, EscrF=>EscrF, Flush=>Flush, sal(63 downto 32)=>instruccionD, sal(31 downto
0)=>cp_4D);

--ETAPA ID


PASO5: desplazador_j PORT MAP(instruc=>instruccionD(25 downto 0),
desplazado=>desplazado_salto);
direccion_jump <= cp_4D(31 downto 28) & desplazado_salto;
PASO6: UC PORT MAP(tipo=>instruccionD(31 downto 26), salida=>controlprevio);
PASO7: UR PORT MAP(LeerMemX=>controlX(4), RtX=>Rt, RsD=>instruccionD(25 downto 21),
RtD=>instruccionD(20 downto 16), EscrPC=>EscrPC, EscrF=>EscrF, riesgo=>riesgo);
PASO8: UAE PORT MAP(Reg_escritura=>destinoW, Reg_lectura1=>instruccionD(25 downto
21), Reg_lectura2=>instruccionD(20 downto 16), anticipacion3=>anticipacion3,
anticipacion4=>anticipacion4);
PASO9: multi PORT MAP(I0=>dato_leido_1D, I1=>datoaescrW, c=>anticipacion3,
s=>dato_correcto1);
PASO10: multi PORT MAP(I0=>dato_leido_2D, I1=>datoaescrW, c=>anticipacion4,
s=>dato_correcto2);
PASO11: banco_reg PORT MAP (clk=>clk, rst=>reset, wr=>controlW(0),
reg_lectura_1=>instruccionD(25 downto 21), reg_lectura_2=>instruccionD(20 downto 16),
reg_escritura=>destinoW, dato_escrito=>datoaescrW, dato1=>dato_leido_1D,
dato2=>dato_leido_2D);
PASO12: beq PORT MAP(EscrRegX=>controlX(5), EscrRegM=>controlM(2),
Rs=>instruccionD(25 downto 21), Rt=>instruccionD(20 downto 16), RdX=>destinoX,
RdM=>destinoM, control_beq1=>equal1, control_beq2=>equal2);
```

```
PASO13: multisalto PORT MAP(I0=>dato_correcto1, I1=>resultado_aluX,
I2=>resultado_aluM, c=>equal1, s=>dato1);
PASO14: multisalto PORT MAP(I0=>dato_correcto2, I1=>resultado_aluX,
I2=>resultado_aluM, c=>equal2, s=>dato2);
PASO15: igualdad PORT MAP (AluOP=>controlprevio(1 downto 0), datoleido1=>dato1,
datoleido2=>dato2, zero=>cero);
Fuente_PC <= controlprevio(2) AND cero;
Seleccion_salto <= Fuente_PC & controlprevio(9);
PASO16: vaciado PORT MAP(Seleccion_Salto=>Seleccion_Salto, flush=>flush);
PASO17: multicontrol PORT MAP(I0=>controlprevio, c=>riesgo, s=>controlD);
PASO18: ext PORT MAP(A=>instruccionD(15 downto 0),B=>extendidoD);
PASO19: desplazador PORT MAP(extensor=>extendidoD, desplazado=>desplazado);
PASO20: sumador PORT MAP(cp4=>cp_4D, desplazado=>desplazado,
salida=>salida_sumador);
PASO21: reg2 PORT MAP(ent(120 downto 111)=>controlD, ent(110 downto
79)=>dato_correcto1,                                    ent(78 downto 47)=>dato_correcto2,
ent(46 downto 15)=>extendidoD,                                ent(14 downto
10)=>instruccionD(25 downto 21), ent(9 downto 5)=>instruccionD(20 downto 16),
ent(4 downto 0)=>instruccionD(15 downto 11), clk=>clk, sal(120 downto 111)=>controlX,
sal(110 downto 79)=>dato_leido_1X, sal(78 downto 47)=>dato_leido_2X,
sal(46 downto 15)=>extendidoX, sal(14 downto 10)=>Rs, sal(9 downto 5)=>Rt,
        sal(4 downto 0)=>Rd);

--ETAPA EX

PASO22: multi2 PORT MAP (I0=>Rt, I1=>Rd, c=>controlX(8), s=>destinoX);
PASO23: UA PORT MAP(EscrRegM=>controlM(2), EscrRegW=>controlW(0), Rs=>Rs, Rt=>Rt,
RdM=>destinoM, RdW=>destinoW, anticiparA=>anticipacion1, anticiparB=>anticipacion2);
PASO24: multisalto PORT MAP (I0=>dato_leido_1X, I1=>datoaescrW, I2=>resultado_aluM,
c=>anticipacion1, s=>dato_alu1);
PASO25: multisalto PORT MAP (I0=>dato_leido_2X, I1=>datoaescrW, I2=>resultado_aluM,
c=>anticipacion2, s=>datoprevio);
PASO26: multi PORT MAP (I0=>datoprevio, I1=>extendidoX, c=>controlX(7), s=>dato_alu2);
PASO27: CA PORT MAP (ALUOp=>controlX(1 downto 0), funcion=>extendidoX(5 downto 0),
resultado=>control_alu);
PASO28: ALU PORT MAP (A=>dato_alu1, B=>dato_alu2, sel=>control_alu,
resultado=>resultado_aluX);

PASO29: reg3 PORT MAP (ent(4 downto 0)=>destinoX, ent(36 downto 5)=>datoprevio,
ent(68 downto 37)=>resultado_aluX, ent(72 downto 69)=>controlX(6 downto 3), clk=>clk,
sal(4 downto 0)=>destinoM, sal(36 downto 5)=>datoaescrM, sal(68 downto
37)=>resultado_aluM, sal(72 downto 69)=>controlM);

--ETAPA MEM

direccion<=resultado_aluM(9 downto 0);
PASO30: ram2 PORT MAP(EscrMem=>controlM(0), LeerMem=>controlM(1), radd=>direccion,
wadd=>direccion, data_in=>datoaescrM, data_out=>dato_leidoM);
PASO31: reg4 PORT MAP(ent(4 downto 0)=>destinoM, ent(36 downto 5)=>resultado_aluM,
ent(68 downto 37)=>dato_leidoM, ent(69)=>controlM(2), ent(70)=>controlM(3), clk=>clk,
sal(4 downto 0)=>destinoW, sal(36 downto 5)=>resultado_aluW,
sal(68 downto 37)=>dato_leidoW, sal(70 downto 69)=>controlW);
```

```
--ETAPA WB

PASO32: multi PORT MAP(I0=>resultado_aluW, I1=>dato_leidoW, c=>controlW(1),
s=>datoaescrW);

   with opt select
      muestra <= instruccionF(15 downto 0) when "000",
            instruccionF(31 downto 16) when "001",
            dato_leidoW(15 downto 0) when "010",
            dato_leidoW(31 downto 16) when "011",
            "000000" & controlprevio when others;

end arch_segmentado;
```

4. Elementos MIPS segmentado con memorias asíncronas

CONTADOR DE PROGRAMA

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

entity cp is
    Port ( clk, rst, EscrPC: in STD_LOGIC;
           entrada: in STD_LOGIC_VECTOR (31 downto 0);
           sal1, sal2 : out STD_LOGIC_VECTOR (31 downto 0)--cp y cp4
           );
end cp;

architecture arch_cp of cp is
signal cp: STD_LOGIC_VECTOR(31 downto 0):=(others=>'0');

begin

process (clk, rst, EscrPC, entrada)
begin

    if (rst = '1') then
        cp <= (others=>'0');
    elsif (EscrPC = '1') then
        cp <= cp;
    else
        cp <= entrada;
    end if;

    if(clk'event AND clk = '1') then
           if(cp = "11111111111111111111111111111111") then
              sal1 <= (others => '0');
              sal2 <= cp+4;
           else
              sal1 <= cp;
              sal2 <= cp + 4;
           end if;
    end if;

end process;

end arch_cp;
```

REGISTRO 1

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

entity reg1 is
   Port (
   ent : in STD_LOGIC_VECTOR (63 downto 0);
   clk, EscrF, Flush : in std_logic;
   sal : out STD_LOGIC_VECTOR (63 downto 0));
end reg1;

architecture arch_reg1 of reg1 is

   begin

   process(clk)
   begin

      if Flush = '1' then
         if clk'event and clk ='1' then

            sal(63 downto 32) <= "00000000000000000000000000000000";
            sal(31 downto 0) <= ent(31 downto 0);

          end if;

      elsif EscrF = '0' then
         if clk'event and clk = '1' then

            sal(63 downto 0) <= ent(63 downto 0);

         end if;
      end if;
   end process;

end arch_reg1;
```

REGISTRO 2

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

entity reg2 is
   Port (
   ent : in STD_LOGIC_VECTOR (120 downto 0);
   clk : in std_logic;
   sal : out STD_LOGIC_VECTOR (120 downto 0));
end reg2;

architecture arch_reg2 of reg2 is
   begin

   process(clk)
   begin
      if clk'event and clk = '1' then
           sal <= ent;
      end if;

end process;
end arch_reg2;
```

REGISTRO 3

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

entity reg3 is
   Port (
   ent : in STD_LOGIC_VECTOR (72 downto 0);
   clk : in std_logic;
   sal : out STD_LOGIC_VECTOR (72 downto 0));
end reg3;

architecture arch_reg3 of reg3 is
   begin

   process(clk)
   begin
      if clk'event and clk = '1' then
           sal <= ent;
      end if;

end process;
end arch_reg3;
```

REGISTRO 4

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

entity reg4 is
   Port (
   ent : in STD_LOGIC_VECTOR (70 downto 0);
   clk : in std_logic;
   sal : out STD_LOGIC_VECTOR (70 downto 0));
end reg4;

architecture arch_reg4 of reg4 is

   begin

   process(clk)
   begin
      if clk'event and clk = '1' then
            sal <= ent;
      end if;

end process;
end arch_reg4;
```

UNIDAD DE RIESGO

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

entity UR is
  Port (
       LeerMemX: in std_logic;
       RtX, RsD, RtD: in std_logic_vector(4 downto 0);
       EscrPC, EscrF, riesgo: out std_logic
   );
end UR;

architecture arch_UR of UR is
begin

process(LeerMemX, RtX, RsD, RtD)
begin

EscrPc <= '0';
EscrF <= '0';
riesgo <= '0';

    if(LeerMemX = '1') then
       if(RtX=RsD OR RtX=RtD) then

          EscrPc<='1';
          EscrF<='1';
          riesgo <= '1';
       end if;
    end if;

end process;

end arch_UR;
```

UNIDAD ANTICIPACIÓN DE LECTURA

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

entity UAE is
   Port ( Reg_escritura : in STD_LOGIC_VECTOR (4 downto 0);
          Reg_lectura1 : in STD_LOGIC_VECTOR (4 downto 0);
          Reg_lectura2 : in STD_LOGIC_VECTOR (4 downto 0);
          anticipacion3 : out STD_LOGIC;
          anticipacion4 : out STD_LOGIC);
end UAE;

architecture arch_UAE of UAE is
begin
process(Reg_escritura, Reg_lectura1, Reg_lectura2)
begin
   anticipacion3 <= '0';
   anticipacion4 <= '0';
   if(Reg_escritura=Reg_lectura1 and Reg_escritura /= "00000") then
      anticipacion3 <= '1';
   end if;
   if(Reg_escritura=Reg_lectura2 and Reg_escritura /= "00000") then
      anticipacion4 <= '1';
   end if;
end process;
end arch_UAE;
```

BEQ

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

entity beq is
    Port (
        EscrRegX, EscrRegM: in STD_LOGIC;
        Rs, Rt, RdX, RdM: in STD_LOGIC_VECTOR(4 downto 0);
        control_beq1, control_beq2: out STD_LOGIC_VECTOR(1 downto 0)
        );
end beq;

architecture arch_beq of beq is

begin

process(EscrRegX, EscrRegM, Rs, Rt, RdX, RdM)
begin

   if(Rs = RdX AND EscrRegX = '1') then
     control_beq1 <= "01";
   elsif (Rs = RdM AND EscrRegM = '1') then
     control_beq1 <= "10";
   else
     control_beq1 <= "00";
   end if;

   if(Rt = RdX AND EscrRegX = '1') then
     control_beq2 <= "01";
   elsif (Rt = RdM AND EscrRegM = '1') then
     control_beq2 <= "10";
   else
     control_beq2 <= "00";
   end if;

end process;


end arch_beq;
```

IGUALDAD

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

entity igualdad is
    Port (
          AluOP: in STD_LOGIC_VECTOR(1 downto 0);
          datoleido1, datoleido2: in STD_LOGIC_VECTOR (31 downto 0);
          zero : out STD_LOGIC);
end igualdad;

architecture arch_igualdad of igualdad is

begin
process(AluOP, datoleido1, datoleido2)
begin

    if(datoleido1 = datoleido2 AND AluOP ="01") then
       zero <= '1';
    else
       zero <= '0';
    end if;

end process;
end arch_igualdad;
```

VACIADO

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

entity vaciado is
      Port (
          Seleccion_salto: in STD_LOGIC_VECTOR(1 downto 0);
          flush: out STD_LOGIC
       );
end vaciado;

architecture arch_vaciado of vaciado is

begin

   process(Seleccion_salto)
   begin

      if(Seleccion_salto = "01" OR Seleccion_salto= "10") then
         flush<= '1';
      else
         flush<='0';
      end if;
   end process;
end arch_vaciado;
```

MULTICONTROL

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

entity multicontrol is
   Port ( I0: in STD_LOGIC_VECTOR (9 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(9 downto 0));
end multicontrol;

architecture arch_multicontrol of multicontrol is

begin
   with c select
      s <= "0000000000" when '1',
           I0 when others;
end arch_multicontrol;
```

UNIDAD DE ANTICIPACIÓN

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

entity UA is
        Port (
            EscrRegM, EscrRegW: in STD_LOGIC;
            Rs, Rt, RdM, RdW: in STD_LOGIC_VECTOR(4 downto 0);
            anticiparA, anticiparB: out STD_LOGIC_VECTOR(1 downto 0)
        );
end UA;

architecture arch_UA of UA is

begin

process(EscrRegM, EscrRegW, Rs, Rt, RdM, RdW)
begin

   anticiparA <= "00";
   anticiparB <= "00";

   if(EscrRegM /= '0' AND RdM /= "00000") then
      if(RdM = Rs) then
         anticiparA <= "10";
      end if;
      if(RdM = Rt) then
         anticiparB <= "10";
      end if;
   end if;

   if(EscrRegW /= '0' AND RdW /= "00000") then
      if(RdW = Rs) then
         anticiparA <= "01";
      end if;
      if(RdW = Rt) then
         anticiparB <= "01";
      end if;
   end if;

end process;

end arch_UA;
```

5. MIPS segmentado con memorias síncronas

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

entity segmentado is
Generic(N : natural := 32);
    Port (
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          opt: in STD_LOGIC_VECTOR (2 downto 0);
          muestra : out STD_LOGIC_VECTOR (15 downto 0)
          );
end segmentado;

architecture arch_segmentado of segmentado is

component cp
     PORT(
          clk, rst : in STD_LOGIC;
          entrada : in STD_LOGIC_VECTOR (31 downto 0);
          sal1, sal2 : out STD_LOGIC_VECTOR (31 downto 0)
         );
 end component;

 component ROM
     PORT(
           clk, rst : in STD_LOGIC;
           direc : in STD_LOGIC_VECTOR (9 downto 0);
           instruccion : out STD_LOGIC_VECTOR (31 downto 0)
         );
 end component;

 component reg1
     Port (
          ent : in STD_LOGIC_VECTOR (63 downto 0);
          clk, Flush, rst : in std_logic;
          sal : out STD_LOGIC_VECTOR (63 downto 0)
          );
 end component;

 component UR
   Port (
        LeerMemX: in std_logic;
        RtX, RsD, RtD: in std_logic_vector(4 downto 0);
        EscrPC, EscrF, riesgo: out std_logic
         );
 end component;

 component desplazador_j
```

```
    Port (
         instruc : in STD_LOGIC_VECTOR (25 downto 0);
         desplazado : out STD_LOGIC_VECTOR (27 downto 0)
         );
end component;

component UC

    Port (
         tipo : in STD_LOGIC_VECTOR (5 downto 0);
         salida : out STD_LOGIC_VECTOR (9 downto 0)
         );

end component;

component banco_reg
     PORT(
         clk, rst : in STD_LOGIC;
         wr : in STD_LOGIC;
         reg_lectura_1 : in STD_LOGIC_VECTOR (4 downto 0);
         reg_lectura_2 : in STD_LOGIC_VECTOR (4 downto 0);
         reg_escritura : in STD_LOGIC_VECTOR (4 downto 0);
         dato_escrito : in STD_LOGIC_VECTOR (31 downto 0);
         dato1 : out STD_LOGIC_VECTOR (31 downto 0);
         dato2 : out STD_LOGIC_VECTOR (31 downto 0)
        );
end component;

component beq
     Port (
          EscrRegX, EscrRegM: in STD_LOGIC;
          Rs, Rt, RdX, RdM: in STD_LOGIC_VECTOR(4 downto 0);
          control_beq1, control_beq2: out STD_LOGIC_VECTOR(1 downto 0)
         );
end component;

component igualdad
     Port (
          AluOP: in STD_LOGIC_VECTOR(1 downto 0);
          datoleido1, datoleido2: in STD_LOGIC_VECTOR (31 downto 0);
          zero : out STD_LOGIC
           );
end component;

component vaciado
       Port (
            Seleccion_salto: in STD_LOGIC_VECTOR(1 downto 0);
            flush: out STD_LOGIC
            );
end component;
```

```
component controlmux
    Port (
          Flush : in STD_LOGIC;
          riesgo : in STD_LOGIC;
          mux : out STD_LOGIC
          );
end component;

component multicontrol
    Port (
          I0: in STD_LOGIC_VECTOR (9 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(9 downto 0)
          );
end component;

component ext
     Port (
           A : in STD_LOGIC_VECTOR (15 downto 0);
           B : out STD_LOGIC_VECTOR (31 downto 0)
         );
end component;

component reg2
    Port (
         ent : in STD_LOGIC_VECTOR (118 downto 0);
         clk,rst : in std_logic;
         sal : out STD_LOGIC_VECTOR (118 downto 0)
         );
end component;

component UA
 Port (
       EscrRegM, EscrRegW, LeerMemM: in STD_LOGIC;
       Rs, Rt, RdM, RdW: in STD_LOGIC_VECTOR(4 downto 0);
       anticiparA, anticiparB: out STD_LOGIC_VECTOR(1 downto 0)
       );
end component;

component multi
    Port (
          I0,I1: in STD_LOGIC_VECTOR (31 downto 0);
          c : in STD_LOGIC;
          s : out STD_LOGIC_VECTOR(31 downto 0) );
end component;

component desplazador
    Port (
          extensor : in STD_LOGIC_VECTOR (31 downto 0);
          desplazado : out STD_LOGIC_VECTOR (31 downto 0)
          );
```

```
end component;

component multi2
   Port(
        I0,I1: in STD_LOGIC_VECTOR (4 downto 0);
        c : in STD_LOGIC;
        s : out STD_LOGIC_VECTOR(4 downto 0)
        );

end component;

component CA
   Port (
        ALUOp : in STD_LOGIC_VECTOR (1 downto 0);
        funcion : in STD_LOGIC_VECTOR (5 downto 0);
        resultado : out STD_LOGIC_VECTOR (2 downto 0)
        );
end component;

component alu
  Port (
       A : in STD_LOGIC_VECTOR (N-1 downto 0);
       B : in STD_LOGIC_VECTOR (N-1 downto 0);
       sel : in STD_LOGIC_VECTOR (2 downto 0);
       resultado: inout STD_LOGIC_VECTOR(N-1 downto 0)
       );
end component;

component sumador
  Port (
        cp4 : in STD_LOGIC_VECTOR (31 downto 0);
        desplazado : in STD_LOGIC_VECTOR (31 downto 0);
        salida : out STD_LOGIC_VECTOR (31 downto 0)
        );
end component;

component reg3
   Port (
       ent : in STD_LOGIC_VECTOR (72 downto 0);
       clk,rst : in std_logic;
       sal : out STD_LOGIC_VECTOR (72 downto 0)
       );
end component;

component multisalto
   Port (
       I0,I1,I2,I3: in STD_LOGIC_VECTOR (31 downto 0);
       c : in STD_LOGIC_VECTOR (1 downto 0);
       s : out STD_LOGIC_VECTOR(31 downto 0)
       );
end component;
```

```
 component ram2
   Port (
        Clk, rst : in std_logic;
        EscrMem : in std_logic;
        LeerMem: in std_logic;
        radd : in std_logic_vector(9 downto 0);
        wadd : in std_logic_vector(9 downto 0);
        data_in : in std_logic_vector(31 downto 0);
        data_out : out std_logic_vector(31 downto 0)
       );
end component;

component reg4
   Port (
        ent : in STD_LOGIC_VECTOR (70 downto 0);
        clk,rst : in std_logic;
        sal : out STD_LOGIC_VECTOR (70 downto 0)
        );
end component;


signal c_p: std_logic_vector(31 downto 0);
signal cp_4F: std_logic_vector(31 downto 0);
signal cp_4D: std_logic_vector(31 downto 0);
signal instruccionF: std_logic_vector(31 downto 0);
signal instruccionD: std_logic_vector(31 downto 0);
signal Rs: std_logic_vector(4 downto 0);
signal Rt: std_logic_vector(4 downto 0);
signal Rd: std_logic_vector(4 downto 0);
signal desplazado_salto: std_logic_vector(27 downto 0);
signal direccion_jump: std_logic_vector(31 downto 0);
signal controlD: std_logic_vector(9 downto 0);
signal controlX: std_logic_vector(7 downto 0);
signal controlM: std_logic_vector(3 downto 0);
signal controlW: std_logic_vector(1 downto 0);
signal dato1: std_logic_vector(31 downto 0);
signal dato2: std_logic_vector(31 downto 0);
signal extendidoX: std_logic_vector(31 downto 0);
signal extendidoD: std_logic_vector(31 downto 0);
signal destinoX: std_logic_vector(4 downto 0);
signal destinoM: std_logic_vector(4 downto 0);
signal destinoW: std_logic_vector(4 downto 0);
signal anticipacion1: std_logic_vector(1 downto 0);
signal anticipacion2: std_logic_vector(1 downto 0);
signal datoaescrM: std_logic_vector(31 downto 0);
signal datoaescrW: std_logic_vector(31 downto 0);
signal dato_leido_1D: std_logic_vector(31 downto 0);
signal dato_leido_2D: std_logic_vector(31 downto 0);
signal dato_leido_1X: std_logic_vector(31 downto 0);
signal dato_leido_2X: std_logic_vector(31 downto 0);
```

```
  signal equal1: std_logic_vector(1 downto 0);
  signal equal2: std_logic_vector(1 downto 0);
  signal flush: std_logic;
  signal desplazado: std_logic_vector(31 downto 0);
  signal datoprevio: std_logic_vector(31 downto 0);
  signal dato_alu1: std_logic_vector(31 downto 0);
  signal dato_alu2: std_logic_vector(31 downto 0);
  signal control_alu: std_logic_vector(2 downto 0);
  signal cero: std_logic;
  signal resultado_aluX: std_logic_vector(31 downto 0);
  signal resultado_aluM: std_logic_vector(31 downto 0);
  signal resultado_aluW: std_logic_vector(31 downto 0);
  signal salida_sumador: std_logic_vector(31 downto 0);
  signal Fuente_PC: std_logic;
  signal Seleccion_salto: std_logic_vector(1 downto 0);
  signal vuelta: std_logic_vector(31 downto 0);
  signal direccion: std_logic_vector(9 downto 0);
  signal dato_leidoM: std_logic_vector(31 downto 0);
  signal dato_leidoW: std_logic_vector(31 downto 0);

begin

--ETAPA IF

PASO1: multisalto PORT MAP(I0=>cp_4F, I1=>direccion_jump, I2=>salida_sumador,
I3=>cp_4F, c=>Seleccion_salto, s=>vuelta);
PASO2: cp PORT MAP (clk=>clk, rst=>reset, entrada=>vuelta, sal1=>c_p, sal2=>cp_4F);
PASO3: ROM PORT MAP (clk => clk, rst=>reset, direc=> c_p(11 downto 2),
instruccion=>instruccionF);
PASO4: reg1 PORT MAP (ent(63 downto 32)=>instruccionF, ent(31 downto 0)=>cp_4F,
rst=>reset, clk=>clk, Flush=>Flush, sal(63 downto 32)=>instruccionD, sal(31 downto
0)=>cp_4D);

--ETAPA ID

PASO5: desplazador_j PORT MAP(instruc=>instruccionD(25 downto 0),
desplazado=>desplazado_salto);
direccion_jump <= cp_4D(31 downto 28) & desplazado_salto;
PASO6: UC PORT MAP(tipo=>instruccionD(31 downto 26), salida=>controlD);
PASO7: banco_reg PORT MAP (clk=>clk, rst=>reset, wr=>controlW(0),
reg_lectura_1=>instruccionD(25 downto 21), reg_lectura_2 => instruccionD(20 downto 16),
reg_escritura=>destinoW, dato_escrito=>datoaescrW, dato1=>dato_leido_1D,
dato2=>dato_leido_2D);
PASO8: beq PORT MAP(EscrRegX=>controlX(4), EscrRegM=>controlM(2),
Rs=>instruccionD(25 downto 21), Rt=>instruccionD(20 downto 16), RdX=>destinoX,
RdM=>destinoM, control_beq1=>equal1, control_beq2=>equal2);
PASO9: multisalto PORT MAP(I0=>dato_leido_1D, I1=>resultado_aluX, I2=>resultado_aluM,
I3=>dato_leido_1D, c=>equal1, s=>dato1);
PASO10: multisalto PORT MAP(I0=>dato_leido_2D, I1=>resultado_aluX, I2=>resultado_aluM,
I3=>dato_leido_2D, c=>equal2, s=>dato2);
```

```
PASO11: igualdad PORT MAP (AluOP=>controlD(1 downto 0), datoleido1=>dato1,
datoleido2=>dato2, zero=>cero);
Fuente_PC <= controlD(2) AND cero;
Seleccion_salto <= Fuente_PC & controlD(9);
PASO12: vaciado PORT MAP(Seleccion_Salto=>Seleccion_Salto, flush=>flush);
PASO13: ext PORT MAP(A=>instruccionD(15 downto 0),B=>extendidoD);
PASO14: desplazador PORT MAP(extensor=>extendidoD, desplazado=>desplazado);
PASO15: sumador PORT MAP(cp4=>cp_4D, desplazado=>desplazado,
salida=>salida_sumador);
PASO16: reg2 PORT MAP(ent(118 downto 113)=>controlD(8 downto 3),
ent(112 downto 111)=>controlD(1 downto 0), ent(110 downto 79)=>dato_leido_1D,
ent(78 downto 47)=>dato_leido_2D, ent(46 downto 15)=>extendidoD,
ent(14 downto 10)=>instruccionD(25 downto 21), ent(9 downto 5)=>instruccionD(20 downto
16),                    ent(4 downto 0)=>instruccionD(15 downto 11), clk=>clk,
rst=>reset,                    sal(118 downto 111)=>controlX, sal(110 downto
79)=>dato_leido_1X,                    sal(78 downto 47)=>dato_leido_2X, sal(46
downto 15)=>extendidoX, sal(14 downto 10)=>Rs,    sal(9 downto 5)=>Rt, sal(4 downto
0)=>Rd);

--ETAPA EX

PASO17: multi2 PORT MAP (I0=>Rt, I1=>Rd, c=>controlX(7), s=>destinoX);
PASO18: UA PORT MAP(EscrRegM=>controlM(2), EscrRegW=>controlW(0),
LeerMemM=>controlM(1), Rs=>Rs, Rt=>Rt, RdM=>destinoM, RdW=>destinoW,
anticiparA=>anticipacion1, anticiparB=>anticipacion2);
PASO19: multisalto PORT MAP (I0=>dato_leido_1X, I1=>datoaescrW, I2=>resultado_aluM,
I3=>dato_leidoM, c=>anticipacion1, s=>dato_alu1);
PASO20: multisalto PORT MAP (I0=>dato_leido_2X, I1=>datoaescrW, I2=>resultado_aluM,
I3=>dato_leidoM, c=>anticipacion2, s=>datoprevio);
PASO21: multi PORT MAP (I0=>datoprevio, I1=>extendidoX, c=>controlX(6), s=>dato_alu2);
PASO22: CA PORT MAP (ALUOp=>controlX(1 downto 0), funcion=>extendidoX(5 downto 0),
resultado=>control_alu);
PASO23: ALU PORT MAP (A=>dato_alu1, B=>dato_alu2, sel=>control_alu,
resultado=>resultado_aluX);

PASO24: reg3 PORT MAP (ent(4 downto 0)=>destinoX, ent(36 downto 5)=>datoprevio,
ent(68 downto 37)=>resultado_aluX, ent(72 downto 69)=>controlX(5 downto 2), clk=>clk,
rst=>reset, sal(4 downto 0)=>destinoM, sal(36 downto 5)=>datoaescrM,
sal(68 downto 37)=>resultado_aluM, sal(72 downto 69)=>controlM);

--ETAPA MEM

direccion<=resultado_aluM(9 downto 0);
PASO25: ram2 PORT MAP(Clk=>clk, rst=>reset, EscrMem=>controlM(0),
LeerMem=>controlM(1), radd=>direccion, wadd=>direccion, data_in=>datoaescrM,
data_out=>dato_leidoM);
PASO26: reg4 PORT MAP(ent(4 downto 0)=>destinoM, ent(36 downto 5)=>resultado_aluM,
ent(68 downto 37)=>dato_leidoM, ent(69)=>controlM(2), ent(70)=>controlM(3), clk=>clk,
rst=>reset, sal(4 downto 0)=>destinoW, sal(36 downto 5)=>resultado_aluW,
sal(68 downto 37)=>dato_leidoW, sal(70 downto 69)=>controlW);
```

```
--ETAPA WB

PASO27: multi PORT MAP(I0=>resultado_aluW, I1=>dato_leidoW, c=>controlW(1),
s=>datoaescrW);

   with opt select
      muestra <= instruccionF(15 downto 0) when "000",
            instruccionF(31 downto 16) when "001",
            dato_leidoW(15 downto 0) when "010",
            dato_leidoW(31 downto 16) when "011",
            dato_leidoM(15 downto 0) when "100",
            "000000" & controlD when others;

end arch_segmentado;
```

6. Elementos MIPS segmentado con memorias síncronas

REGISTRO 1

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

entity reg1 is
   Port (
   ent : in STD_LOGIC_VECTOR (63 downto 0);
   clk, Flush, rst : in std_logic;
   sal : out STD_LOGIC_VECTOR (63 downto 0));
end reg1;
architecture arch_reg1 of reg1 is
   begin
   process(clk, rst)
   begin
      if rst = '1' then
         sal <= (others=>'0');
      elsif clk'event and clk ='1' then
         if Flush = '1' then
            sal(63 downto 32) <= "00000000000000000000000000000000";
            sal(31 downto 0) <= ent(31 downto 0);
         else
            sal(63 downto 0) <= ent(63 downto 0);
         end if;
      end if;
   end process;
   end arch_reg1;
```

REGISTRO 2

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

entity reg2 is
    Port (
    ent : in STD_LOGIC_VECTOR (118 downto 0);
    clk,rst : in std_logic;
    sal : out STD_LOGIC_VECTOR (118 downto 0));


end reg2;

architecture arch_reg2 of reg2 is

    begin
    process(clk, rst)
    begin
        if rst = '1' then
            sal <= (others => '0');
        elsif clk'event and clk = '1' then
            sal <= ent;
        end if;
end process;
end arch_reg2;
```

REGISTRO 3

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

entity reg3 is
   Port (
   ent : in STD_LOGIC_VECTOR (72 downto 0);
   clk, rst : in std_logic;
   sal : out STD_LOGIC_VECTOR (72 downto 0));
end reg3;

architecture arch_reg3 of reg3 is

   begin

   process(clk,rst)
   begin

      if rst = '1' then
           sal <= (others=>'0');
      elsif clk'event and clk = '1' then
           sal <= ent;
      end if;
end process;
end arch_reg3;
```

REGISTRO 4

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

entity reg4 is
   Port (
   ent : in STD_LOGIC_VECTOR (70 downto 0);
   clk,rst : in std_logic;
   sal : out STD_LOGIC_VECTOR (70 downto 0));
end reg4;

architecture arch_reg4 of reg4 is

   begin

   process(clk,rst)
   begin

       if rst = '1' then
            sal <= (others=>'0');
       elsif clk'event and clk = '1' then
            sal <= ent;
       end if;
end process;
end arch_reg4;
```

ROM

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity ROM is
   Port (
        clk,rst : in STD_LOGIC;
        direc : in STD_LOGIC_VECTOR (9 downto 0);
        instruccion : out STD_LOGIC_VECTOR (31 downto 0));
end ROM;
architecture arch_ROM of ROM is
signal aux: std_logic_vector(9 downto 0);
TYPE rom1 is ARRAY (0 to 1023) of STD_LOGIC_VECTOR (31 downto 0);
signal memoria : rom1:=(

--              PARTE COMUN
                x"20110003",
                x"20120005",

--               PROGRAMA1
                x"12400003",
                x"2252FFFF",
                x"02719820",
                x"1000FFFC",


--               PROGRAMA2
                x"20130001",
                x"2014000F",
                x"10140007",
                x"0253A824",
                x"10150001",
                x"02D1B020",
                x"02318820",
                x"02739820",
                x"2294FFFF",
                x"1000FFF8",

--               PROGRAMA3
                x"0232982A",
                x"AE330000",
                x"8E340000",
                x"0232A825",
                x"0232B022",
                x"08000000",

--               PROGRAMA 4
                x"AE510005",
                x"8E500005",
                x"AE500006",

          others =>x"1000FFFF" );--bucle final comun
```

```
begin

process(clk,rst)
begin

   if rst/='0' then
         instruccion <= x"00000000";
   elsif clk'event and clk='0' and rst/='1' then
         instruccion <= memoria(conv_integer(direc));
   end if;
end process;

end arch_ROM;
```

UNIDAD DE ANTICIPACIÓN

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

entity UA is
        Port (
            EscrRegM, EscrRegW, LeerMemM: in STD_LOGIC;
            Rs, Rt, RdM, RdW: in STD_LOGIC_VECTOR(4 downto 0);
            anticiparA, anticiparB: out STD_LOGIC_VECTOR(1 downto 0)
        );
end UA;

architecture arch_UA of UA is
begin

process(EscrRegM, EscrRegW, Rs, Rt, RdM, RdW, LeerMemM)
begin

   anticiparA <= "00";
   anticiparB <= "00";

   if(EscrRegM /= '0' AND RdM /= "00000" AND LeerMemM /= '1') then
      if(RdM = Rs) then
         anticiparA <= "10";
      end if;
      if(RdM = Rt) then
         anticiparB <= "10";
      end if;
   end if;

   if(EscrRegW /= '0' AND RdW /= "00000") then
      if(RdW = Rs) then
         anticiparA <= "01";
      end if;
      if(RdW = Rt) then
         anticiparB <= "01";
      end if;
   end if;

   if(LeerMemM = '1') then
      if(RdM=Rs) then
         anticiparA <= "11";
      end if;
      if(RdM=Rt) then
         anticiparB <= "11";
      end if;
   end if;
end process;

end arch_UA;
```

MULTIPLEXOR 4 A 1

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

entity multisalto is
   Port ( I0,I1,I2,I3: in STD_LOGIC_VECTOR (31 downto 0);
          c : in STD_LOGIC_VECTOR (1 downto 0);
          s : out STD_LOGIC_VECTOR(31 downto 0));
end multisalto;

architecture arch_multisalto of multisalto is

begin
   with c select
      s <= I0 when "00",
           I2 when "10",
           I3 when "11",
           I1 when others;

end arch_multisalto;
```

RAM

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


entity ram2 is
Port (  Clk, rst : in std_logic;
        EscrMem : in std_logic;
        LeerMem: in std_logic;
        radd : in std_logic_vector(9 downto 0);
        wadd : in std_logic_vector(9 downto 0);
        data_in : in std_logic_vector(31 downto 0);
        data_out : out std_logic_vector(31 downto 0)
    );
end ram2;


architecture arch_ram2 of ram2 is

type ram is array(0 to 1023) of std_logic_vector(31 downto 0);
signal ram1_1 : ram;


begin


process(Clk, rst, wadd, EscrMem)
begin
    if rst /= '0' then
            ram1_1<=(others=>x"00000000");
    elsif Clk'event and Clk = '0' then
        if EscrMem = '1' then
            ram1_1(conv_integer(wadd)) <= data_in;
        end if;
    end if;

end process;


process(Clk, LeerMem, radd)
begin
    if Clk'event and Clk = '0' then
        if LeerMem = '1' then
            data_out <= ram1_1(conv_integer(radd));
        end if;
    end if;
end process;
end arch_ram2;
```