CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN COMPUTACIÓN

CINVESTAV
IPN
ADQUISICION
DE LIBROS

"Diseño e Implementación de un microprocesador RISC en VHDL"

Tesis que presenta

Lic. Sergina Ascelli Shayá Zamudio Vissuet

Para Obtener el Grado de

Maestra en Ciencias

En la Especialidad de

Ingeniería Eléctrica

Director de la Tesis: Dr. Adriano de Luca Pennacchia

México, D.F. Mayo 2003

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

Introducción

El empleo de FPGAs (*Field Programmable Gate Arrays*) tiene ciertas ventajas, entre ellas la flexibilidad que se obtiene al desarrollar un prototipo. Un microprocesador o microcontrolador comercial tiene ciertas funciones, y al adquirirlo ya no es posible modificarlas. En cambio, un FPGA puede ser reprogramado, en poco tiempo, con las funciones específicas que requiere un proyecto. Esto da lugar al cómputo reconfigurable, que tiene ciertas ventajas sobre dispositivos con una función fija, por ejemplo: reducción de costos, mayor desempeño y confiabilidad, además de implementación de interfaces especializadas y de alta velocidad. Así, podemos pensar que el desarrollar un sistema digital de esta manera puede ser útil con fines académicos y en algunas aplicaciones industriales.

Para la descripción de procesos dentro del FPGA se requiere un lenguaje de alto nivel que permita varios niveles de abstracción en el diseño, tales como Verilog o VHDL. En este caso se empleará VHDL (VHSIC *Very High Speed Integrated Circuits* Hardware Description Language), que desde 1987 se convirtió en un estándar de la IEEE. En términos generales podemos decir que VHDL es un lenguaje similar al C, pero que además permite la descripción de modelos de hardware.

Un diseño puede dividirse en varios bloques especializados en cierta función, y por medio de VHDL se detallará su comportamiento, así como sus entradas y salidas, que los interconectan. Al comprobar su funcionamiento pueden detectarse errores y corregirse, todo desde un nivel alto. Todo esto le da a VHDL cierta facilidad de uso en el diseño de circuitos.

Al desarrollar este trabajo de tesis se diseñó en VHDL un microprocesador RISC (*Reduced Instruction Set Computer*) y finalmente se simuló su comportamiento en un FPGA. La arquitectura en la que se basa este proyecto es la UAM-RISC [7], cuyos fines son didácticos. En consecuencia uno de los objetivos de este proyecto es exponer detalladamente el ciclo de diseño de un RISC por medio del lenguaje VHDL. Este lenguaje de alto nivel permite describir el comportamiento de un sistema digital como una entidad con entradas y salidas. Posteriormente el diseño puede ser probado e implementado en un FPGA. Las herramientas que utilizamos pertenecen al ambiente de desarrollo de Xilinx Inc., e incluyen interfaces para

el diseño, implementación y simulación de electrónica digital, a través de una computadora personal. El diseño de nuestro microprocesador incluye las características de un sistema RISC:

- un pequeño conjunto de instrucciones,
  - para accesar a la memoria sólo existen instrucciones de tipo LOAD y STORE,
  - ejecución en *pipeline* (segmentación o línea de ejecución en varias etapas),
  - las instrucciones no contienen microcódigo que se deba decodificar,
  - las instrucciones son de 32 bits, y pueden incluir el código de operación, la dirección del o de los registros fuente, la dirección del registro destino, la dirección de la localidad de memoria a la que debe saltar la ejecución, o incluso un operando de valor constante,
  - un conjunto de registros de propósito general.

Estas características se describe en el capítulo 1, enfatizando la importancia del *pipeline*. Además detallamos la estructura interna de los FPGAs, que favorece el desempeño de algunos circuitos. También nos parece importante referir aplicaciones que puede tener el cómputo reconfigurable en diversas áreas de la tecnología. Posteriormente hablamos del ambiente de desarrollo de este proyecto, y esto hace necesario describir el lenguaje VHDL y mencionar las ventajas de emplearlo. Xilinx Foundation Series tiene diversas interfaces, para el diseño, implementación y pruebas de un circuito. En ese capítulo mencionamos de forma superficial cada una de estas interfaces.

La arquitectura UAM – RISC II, sintetizada en un FPGA, tiene diversos componentes, que se describen a detalle en el segundo capítulo: unidad de control, contador de Programa, memorias de Datos y de Instrucciones, ALU, comparador, varios multiplexores, Bloque de registros, Registros del *pipeline* y una unidad para detectar dependencia de datos. Además resulta de gran importancia hablar de las etapas de procesamiento de este *pipeline* y del conjunto de instrucciones que empleará el procesador.

Posteriormente, el capítulo 3 detalla las características del FPGA empleado en la implementación, y se dan razones que justifican la elección de este modelo para el diseño de un *pipeline*. También se describen brevemente las herramientas empleadas para implementar y simular. Mientras se ejecuta el *Flow Engine* necesario para la implementación se generan diversos reportes que van reflejando los resultados de cada fase. En ese capítulo se examina e interpreta dicha información, exponiendo los fragmentos más relevantes en el texto.

Además se detallan de las restricciones de tiempo aplicadas a nuestro diseño. Estas son relevantes dado su efecto sobre el desempeño del sistema.

Finalmente el capítulo presenta varias pruebas del funcionamiento del microprocesador en un FPGA. Valiéndonos de una herramienta de simulación analizamos la ejecución de instrucciones de diferente tipo en cada una de las 5 etapas del *pipeline*, exponiendo incluso casos de dependencia de datos.

En el capítulo 4 se tratan varios temas: limitaciones y posibles mejoras realizables a partir del diseño del UAM - RISC II, así como una perspectiva a futuro del desarrollo de los FPGAs y de sus aplicaciones.

Índice

Capítulo 1 Antecedentes

1.1 Arquitecturas RISC

1.1.1 Microprocesadores

Los elementos básicos de un procesador son una memoria de instrucciones/datos, una unidad aritmética/lógica y una unidad de control que supervisa la manera en que circulan las señales de entrada/salida entre los otros dispositivos. El concepto de arquitectura de computadoras no se restringe simplemente a la estructura física [10]. Podemos decir que una computadora es un sistema integrado por el hardware, un conjunto de instrucciones, un sistema operativo, programas de aplicación e interfaces del usuario. Por lo tanto un sistema de cómputo demuestra su poder al coordinar los esfuerzos realizados por todos estos elementos. Desde el punto de vista del lenguaje ensamblador, la arquitectura de una computadora se describe a partir de su conjunto de instrucciones, que incluye el código de operación, modos de direccionamiento, registros, manejo de la memoria virtual, etc. Desde la perspectiva de la implementación en hardware un procesador está organizado en CPUs, buses, microcódigo, *pipelines*, memoria física, cachés, etc. El estudio de la arquitectura de una computadora debe cubrir ambos enfoques.

1.1.2 Arquitecturas RISC

Existen varios tipos de microprocesadores: RISC, CISC, superescalares, *superpipelined*, *VLIW*, etc. Los que nos interesan en este proyecto son los RISC (*Reduced Instruction Set Computer*), y además comentaremos las características de los microprocesadores CISC (*Complex Instruction Set Computer*). Las diferencias existentes entre ambas familias de microprocesadores pueden suavizarse al construir modelos con características de ambos tipos. A continuación se presentan las principales diferencias, de acuerdo con Kwai [10]:

Característica de la arquitectura	CISC	RISC
Número de instrucciones y su formato	El número de instrucciones es grande y tienen formatos variables que van de los 16 a los 64 bits	Contiene pocas instrucciones, que tienen un formato fijo de 32 bits; la memoria están basadas en registros
Modos de direccionamiento	12 – 24	Limitado de 3 a 5
Registros de propósito general y memoria	De 8 a 24 registros. La mayoría tiene una sola memoria para datos y para instrucciones, aunque en diseños más recientes ya están separadas	De 32 a 192 registros. La mayoría tiene una memoria para datos y otra para instrucciones
Reloj y CPI	33 – 50 MHz (1992) con CPI entre 2 y 5	50 – 150 MHz (1992) con un CPI de 1 para la mayoría de las instrucciones, pero con CPI promedio de 1.5
Control	La mayoría tiene microcódigo, empleando una memoria de control (ROM), pero los CISC modernos también usan control a base de componentes alambrados	La mayoría tiene un control alambrado sin emplear una memoria de control

Tabla 1. 1 Características RISC y CISC

Incrementar la velocidad de ejecución ha sido una motivación constante para buscar mejoras en los microprocesadores. Al crear los CISC se pensó en un programa compuesto por una serie de microinstrucciones, cada una ejecutable con una sola función de control. Debido al avance tecnológico que permite cada vez mayor integración, ha sido posible implementar un mayor número de microinstrucciones; éstas pueden ser de longitud variable; existe un número grande de operaciones que hacen referencia a la memoria, a través de varios modos de direccionamiento; el número de ciclos por instrucción (CPI) varía entre 1 y 20. Algunos procesadores de esta familia son: *Intel i486, M6 8040, VAX/8600, IBM 390*. En FPGA encontramos al *CZ80CPU Microprocessor* (compatible con el *Zilog Z80*), entre otros.

Posteriormente se pensó en simplificar los programas, reduciendo el número de instrucciones posibles. Esto se logró con macroinstrucciones que la unidad de control debe decodificar. Así, por ejemplo, es posible leer los 2 ó más registros fuente en un solo ciclo de reloj. Si la ejecución se organiza en etapas, mientras se leen los registros, el resto de los dispositivos del microprocesador quedan libres para atender a otras instrucciones. Los procesadores RISC [10] poseen, como su nombre lo indica, un pequeño conjunto de instrucciones con un formato fijo; la ejecución puede realizarse en varias etapas de *pipeline*; además tienen un número reducido de formas de accesar a la memoria de datos, separada de la memoria de instrucciones; la mayoría de las instrucciones están basadas en registros, por lo que estos microprocesadores cuentan con gran número de registros de propósito general; el número de instrucciones por ciclo (CPI) es de 1-2; la velocidad puede ser variable, pues depende del avance tecnológico logrado hasta ese momento. Como resultado de la uniformidad del tamaño de las instrucciones y del tiempo de ejecución, los intervalos de espera y el tiempo de decodificación se reducen al mínimo, al reducir el número de ciclos necesarios para cada instrucción (idealmente 1). Estos factores contribuyen a incrementar significativamente la velocidad de cómputo. Esta velocidad se puede ver reducida por las instrucciones de ramificación, o por las dependencias de datos ocasionadas dentro del *pipeline*, lo que nos lleva a un promedio de 1.5 instrucciones por ciclo. Entre los primeros microprocesadores RISC encontramos al *Intel i860, SPARC CY76C601, Motorola M88100, IBM RS/6000*. Asimismo se han implementado algunos procesadores de este tipo en FPGA como *IP (Intelectual Property): V8-uRISC 8-bit Microprocessor, ARC 32-bit Configurable RISC Processor, AX1610 16-bit RISC Processor, MicroBlaze Soft RISC Processor*.

La implementación de un RISC en un FPGA resulta muy conveniente, dada la estructura en *pipeline* del microprocesador. Como ya hemos dicho, en este tipo de arquitecturas varios dispositivos pueden estar activos a un mismo tiempo ya que en cada una de las etapas del *pipeline* hay una instrucción ejecutándose. Al estar constituido un FPGA por miles de bloques lógicos (independientes y a la vez interconectados) soporta esta ejecución simultánea, lo que conlleva una mejor utilización de los recursos del FPGA. Por lo tanto la organización en *pipeline* de los diferentes dispositivos o funciones que integran al microprocesador optimiza el performance del FPGA.

1.1.3 El paralelismo funcional

El paralelismo puede presentarse en diversas formas, tales como *pipelining* (segmentación), concurrencia, datos paralelos, *time sharing*, multitareas, multiprogramación, etc. En las computadoras escalares se han introducido técnicas de *prefetch* que nos permiten ejecutar al mismo tiempo la lectura de la instrucción, la decodificación y la ejecución. Esto habilita el paralelismo funcional [10], que puede tener 2 enfoques: el primero es el usar múltiples unidades funcionales simultáneamente, otro es el *pipelining* con varios niveles de procesamiento. El UAM - RISC II emplea éste último para ejecutar 5 instrucciones de forma paralela, a través de un *pipeline* de 5 etapas.

1.1.4 *Pipeline* lineal

Un microprocesador con un *pipeline* lineal contiene una serie de etapas conectadas en secuencia para ejecutar una instrucción tras otra. Cada instrucción es leída en la etapa 1 y procesada en las siguientes etapas, en cada ciclo de reloj. El resultado final se obtiene en la última etapa del *pipeline*; una vez que hay instrucciones en todas las etapas se obtiene un resultado en cada ciclo de reloj.

La ejecución típica de una instrucción comprende una secuencia de operaciones [10]: lectura de la instrucción, decodificación, lectura de operandos, ejecución (incluyendo accesos a memoria) y escritura de resultados. Estas fases son ideales para ejecutarse en un pipeline lineal. Cada fase puede necesitar uno o más ciclos para ejecutarse, dependiendo del tipo de instrucción y de memoria. Podemos modelar un procesador con base en esta estructura de pipeline. En la práctica, la mayoría de los pipelines tienen entre 2 y 15 etapas; muy pocos pipeline se diseñan con más de 10 etapas en computadoras reales. Para hacer una correcta elección del número de etapas se debe maximizar la razón performance/costo.

Normalmente encontramos tres tipos de pipeline [28]:

- De 3 etapas: lectura de la instrucción, decodificación y escritura de resultados.
- De 4 etapas: lectura de la instrucción, decodificación, ejecución y escritura de resultados.
- De 5 etapas: lectura de la instrucción, decodificación, ejecución, acceso a memoria y escritura de resultados.

La aceleración que se obtiene de un pipeline [28] está en función del algoritmo a ejecutar, dependiendo de cuánto paralelismo permite y cuantos conflictos ocurren dentro del pipeline. La aceleración puede calcularse de la siguiente manera:

$$\text{Aceleración} = \frac{PD}{1 + PSC} = \frac{\text{No. de etapas}}{1 + \text{No. de Ciclos que se detiene la ejecución}}$$

El denominador de esta fórmula se refiere al número de ciclos que debe detenerse la ejecución para resolver conflictos. De acuerdo al tipo de instrucciones que ejecuta el UAM - RISC II, la ejecución únicamente se detiene 2 ciclos en instrucciones de salto. A partir de lo anterior podríamos calcular la aceleración para diferentes longitudes de pipeline en este procesador:

- Para 3 etapas: 1.0
- Para 4 etapas: 1.3
- Para 5 etapas: 2.5

De esta forma podemos ver que resulta apropiado el diseño de un pipeline de 5 etapas para el UAM - RISC II.

Hay varias definiciones importantes relacionadas con la ejecución en *pipeline*, que nos permiten describir mejor a un microprocesador [10]:

Ciclo de instrucción del *pipeline*. Período de reloj del *pipeline* de instrucciones, es decir, el tiempo requerido para que cada fase del *pipeline* se complete. Es conveniente que todas

las etapas tengan un ciclo con la misma duración, pues así se optimiza la ejecución en paralelo de varias instrucciones.

Latencia entre instrucciones. Es el tiempo (en ciclos) requerido entre 2 instrucciones consecutivas; cuando es mayor que 1 ocurre una subutilización del *pipeline*. En el microprocesador desarrollado, la latencia es de 1 y por lo tanto cada ciclo se lee una nueva instrucción (excepto en los casos de ramificación).

Tasa de instrucciones ejecutadas. Es el número de instrucciones ejecutadas por ciclo, también es llamado grado de un procesador superescalar. El UAM - RISC II es un procesador escalar ya que solo ejecuta 1 instrucción por ciclo, y por lo tanto su grado es 1.

Latencia para operaciones simples. Las operaciones simples son la mayoría de las ejecutadas por una computadora, tales como *adds, loads, stores, branches, moves,* etc. Las operaciones complejas requieren una latencia mayor, por ejemplo calcular una división. Esta latencia se mide en número de ciclos. Todas las operaciones que el UAM - RISC II es capaz de ejecutar son simples y tienen latencia 1.

Conflicto de recursos. Existe cuando 2 ó más instrucciones requieren utilizar la misma unidad funcional al mismo tiempo. Ya que en el UAM - RISC II se ejecuta sólo 1 instrucción por ciclo no llega a ocurrir que 2 instrucciones soliciten al mismo tiempo el uso de la ALU.

Un procesador escalar se ejecuta con datos escalares. El más simple ejecuta instrucciones cuyo resultado es un entero, usando operandos de punto fijo. Otros procesadores pueden ejecutar operaciones enteras y de punto flotante. El UAM - RISC II procesa sólo valores enteros.

1.2 FPGAs y Cómputo Reconfigurable

Un FPGA es un arreglo de unidades de procesamiento, y tanto su función como sus interconexiones se programan después de la fabricación. El emplear FPGAs da lugar al cómputo reconfigurable, ya que las operaciones a ejecutar están definidas por bits de configuración, que indican a cada compuerta y a cada interconexión cómo comportarse. Dentro de un FPGA se implementan tareas conectando elementos básicos, limitados dentro de un área. En contraste, en los procesadores tradicionales las operaciones se van ejecutando en secuencia a través del tiempo, empleando registros o memorias para almacenar los resultados intermedios [6]. A partir de lo anterior podemos decir que un FPGA es un dispositivo de arquitectura fija y al mismo tiempo programable, que ofrece un ambiente de desarrollo productivo y eficiente, tanto para el software como para el hardware. Como resultado de estas características, es posible lograr que un diseño en FPGA se adapte exactamente a las necesidades de la aplicación. Entre las ventajas de la integración de los FPGAs con otros dispositivos están la reducción de costos, mayor desempeño y confiabilidad. También hay un gran potencial en su flexibilidad y escalabilidad para implementar interfaces especializadas y de alta velocidad [23]. Por ejemplo, hay estructuras de software y algoritmos (que pueden descomponerse en procesos paralelos que no se bloqueen entre sí) que pueden alcanzar una velocidad significativa en el procesamiento y en la transferencia de datos al ser implementados en hardware.

Entre los antecesores de los FPGAs se encuentran los ASICs (*Application-Specific Integrated Circuits*), circuitos que están enfocados total o parcialmente a una función. También existen los Dispositivos de Lógica Programable (PLDs); los más simples están formados por Arreglos de Lógica Programable (PLAs) y su configuración se determina aplicando cierto voltaje a cada

una de sus conexiones. Posteriormente los dispositivos programables se hicieron mucho más complejos, dando lugar a los CPLDs y a los FPGAs (Field Programmable Gate Arrays). Los FPGAs de Xilinx son implementados con tecnología de memoria RAM estática. A diferencia de la mayoría de los dispositivos de lógica programable su configuración es volátil y debe restablecerse cada vez que se activa. Están formados por arreglos de celdas lógicas, cada una de las cuales puede ser programada para realizar diversas funciones combinatorias y secuenciales. Además de estas celdas hay múltiples líneas de interconexión programables y búfers de 3er estado, así como un oscilador *on-chip* [29].

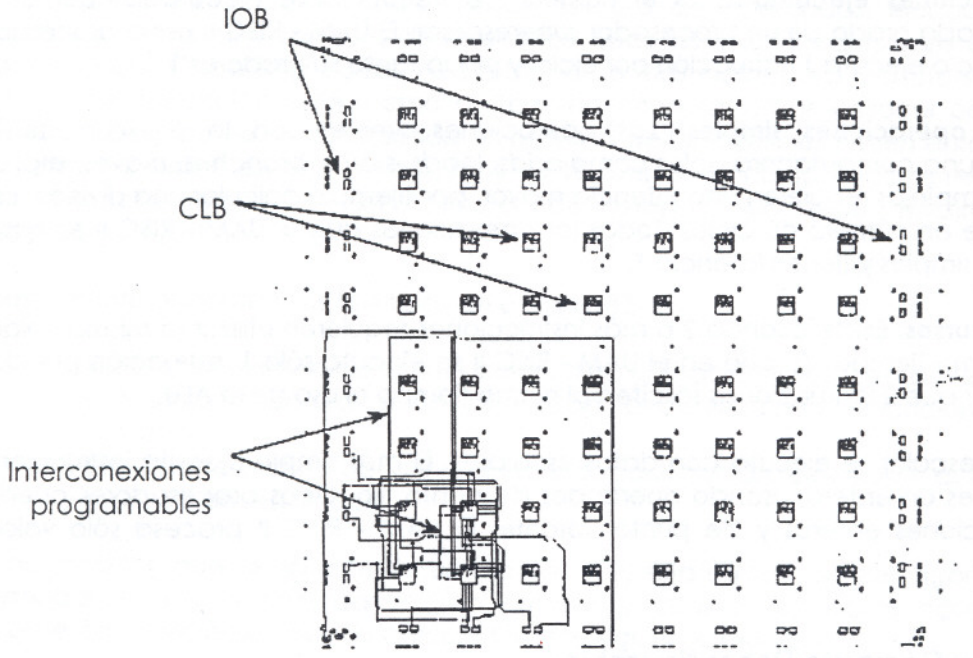


Figura 1.1 Diseño interior de un FPGA de Xilinx [17]

Las unidades básicas del FPGA son el CLB y el IOB. Cada CLB incluye dos generadores de funciones de 16 bits (F o G), un generador de función de 8 bits (H), dos registros (*flip-flops* o *latches*) y un multiplexor. Por medio de un generador de funciones se puede implementar una función de hasta 5 variables; para funciones con más variables se conectan dos o más generadores. La parte de lógica combinatoria de un CLB emplea una *Look-Up Table* (LUT) para implementar funciones booleanas. De esta manera se pueden generar 2 funciones lógicas independientes de hasta 4 variables cada una, o una sola función de 5 variables. Al emplear un lenguaje de alto nivel como VHDL todos estos detalles del mapeo no son visibles para el diseñador [13]. El IOB (*Input/Output Block*) permite la conexión de la lógica interior del FPGA con el exterior. Cada uno de ellos controla un pin del dispositivo, configurable como señal de entrada, salida o bidireccional. Algunos de los componentes de un IOB son: 2 flip-flops, 1 buffer de salida y otro de entrada, varios multiplexores [24].

De acuerdo a sus características, los FPGAs de Xilinx están agrupados en familias [24]. Los Spartan son dispositivos que incluyen algunas de las características de la arquitectura de la familia XC4000. Su diseño puede alcanzar un gran rendimiento. Se caracterizan también por su bajo costo. Dentro de la familia Virtex encontramos dispositivos que tienen gran número de compuertas lógicas (algunos rebasan el millón), y por lo tanto un número enorme de CLBs (más de 6000), además tienen un conjunto de interconexiones rápidas y flexibles; por sus características esta familia es útil para diseños grandes y complejos. Los FPGAs XC4000 son más reducidos que los Virtex (hasta 180,000 compuertas y 3136 CLBs), pero su arquitectura también es apropiada para diseños complejos y de gran densidad. Este microprocesador

reprogramable, el UAM - RISC II, fue diseñado en el dispositivo XC4085XL, elegido por su gran capacidad. Empleamos una PC con sistema operativo Windows NT 4.0 corriendo a 900MHz con 256Kb en RAM.

Aunque el microprocesador está sintetizado sobre un dispositivo específico, se diseñó para poder implementarse en diferentes modelos de FPGA, es decir, para ser independiente de la tecnología, ya que no contiene descripciones específicas para una familia de FPGAs y tampoco emplea componentes diseñados a través de LogiBlox [25]. Ésta es una herramienta gráfica para crear módulos dentro de nuestro diseño a partir de una plantilla con varios parámetros que permiten adaptar su funcionalidad. Existen plantillas para contadores, multiplexores, memorias, registros de corrimiento, compuertas lógicas, etc. Los dispositivos creados con esta herramienta aprovechan las características de la arquitectura en que se implementan. Existen dispositivos prediseñados de Logiblox para diversas familias de FPGAs, como: XC300A, XC3100A, XC4000E, XC4000EX y XC5200. Es por esta razón que se decidió no utilizar Logiblox en este proyecto, y de esta manera el diseño del UAM - RISC II conserva la propiedad de portabilidad del lenguaje VHDL.

Al implementar en un FPGA podemos identificar diversas ventajas sobre un circuito fabricado con funciones fijas [27]:

- Podemos evitar que nuestro microprocesador o microcontrolador caiga en la obsolescencia, al reprogramarlo con nuevas funciones. De esta forma, podemos considerar que su período de vida útil es de mayor duración.
- Si contamos con el código fuente del microprocesador nunca tendremos el problema de que se ha dejado de fabricar el modelo o de que no hay en existencia. Al tener el código fuente, ya sea que lo hayamos diseñado o comprado, es posible implementar una nueva copia del chip.
- Además se podría portar el microprocesador a otra familia de FPGAs de Xilinx.
- Si cambian los requerimientos del diseño es posible hacer las modificaciones.
- También nos es útil la reprogramación cuando cambian los estándares o protocolos en los que interviene el microprocesador en FPGA.

En [5] se mencionan otras ventajas de implementar un microprocesador en FPGA:

- **Bajo costo de implementación.** En el caso de series pequeñas o de microprocesadores que requieren una reconfiguración frecuente, la programación de las conexiones entre las compuertas lógicas de un FPGA tiene mucho menor costo que la producción de un ASIC diferente.
- **Mayor productividad.** El reconfigurar un microprocesador es similar a guardar un nuevo programa de software en memoria. Como resultado de esto, el costo del diseño de un microprocesador reconfigurable depende más de las industrias del software que del hardware.
- **Mayor velocidad.** Los FPGAs pueden llegar a procesar datos a velocidades mayores que un microprocesador de propósito general ya programado. (Dado el avance tecnológico de los microprocesadores actuales, esta afirmación ya no es aplicable).
- **Mayor control sobre los datos.** Todos los registros internos, los buses de datos y de dirección pueden ser observados continuamente y controlados.

Además podemos agregar que uno de los usos más comunes de los microprocesadores es el control de procesos. Para realizar cambios en un control puede ser necesario modificar el software contenido en la memoria de instrucciones. Y al emplear un FPGA los cambios pueden ser tanto en las instrucciones como en la estructura del hardware, lo que también es una ventaja importante.

Los requerimientos de costos bajos y tiempos de diseño cortos constantemente se hacen más estrictos, en las soluciones que los ASICs dan a las pequeñas y medianas empresas (PYMES).

Éstas, por lo general, no cuentan con un experto en diseños para VLSI/ASIC, y tienen presupuestos limitados en las fases de diseño y de desarrollo de prototipos, y generalmente requieren aplicaciones muy específicas. Por otro lado, la tecnología de los FPGAs cumple con la mayoría de los requerimientos de las PYMES en cuanto a una disminución de los costos, de los riesgos, así como tiempos de diseño cortos [15]. Se está convirtiendo en una práctica normal que los productores de ASICs ofrezcan núcleos de hardware sintetizables, construidos en VHDL. Estos modelos tienen el objetivo de explotar la reusabilidad y la portabilidad del VHDL [3], con el fin de reducir costos y tiempos. Además estos dispositivos pueden ser probados en todas sus funciones y esto da la posibilidad de disminuir la tasa de error, lo que los hace atractivos para diseños complejos. También para algunas aplicaciones que tienen una demanda muy pequeña en el mercado. Es por eso que pueden ser ideales para pequeñas compañías o para universidades, que pueden lograr que el hardware basado en FPGA se adapte hasta cumplir totalmente con sus necesidades, en lugar de emplear microprocesadores o microcontroladores con funciones fijas; si se requieren modificaciones sobre éstos, la inversión se habrá perdido [16]. En contraste un FPGA puede ser reprogramado para convertirse en un microprocesador total o ligeramente diferente, aunque por supuesto también hay que realizar una inversión inicial para su adquisición.

En [27] encontramos algunos casos más específicos en los que la reconfigurabilidad del microprocesador es una característica importante:

- Hay estándares de comunicación a alta velocidad (IEEE 1394, IEEE 1355, USB 2.0, etc.) que pueden tener variaciones de acuerdo a la región del mundo en la que nos encontramos. En tal caso nos sería útil poder realizar las modificaciones pertinentes en el microprocesador involucrado en el estándar.
- Al haber diseñado o adquirido un microprocesador o microcontrolador sería posible realizar pequeños cambios sobre él para controlar diversos motores, sin necesidad de intercambiarlo por otro.

La velocidad a la que ocurren los cambios tecnológicos demanda poder reprogramar el hardware. Los fabricantes de FPGAs han creado herramientas que auxilian a los usuarios en esta tarea. Ahora, con la tecnología de Xilinx, un equipo de diseñadores es capaz de realizar cambios y optimizaciones a lo largo del diseño del sistema, generando mayor integración y desempeño, con un menor tiempo para llegar al mercado. Incluso pueden realizarse cambios al hardware y al software cuando ya está en manos del cliente, para corregir errores o implementar nuevas funciones [23]; esto les da a los FPGAs flexibilidad ilimitada. Cuando cambian los requerimientos de un sistema, un diseñador puede modificar el código fuente del diseño en VHDL y volver a sintetizarlo en el mismo FPGA. Esto da la posibilidad de actualizar hardware/software. Por ejemplo, podría bajarse de Internet el código optimizado para reconfigurar el FPGA, del mismo modo que actualmente se hace con una actualización de software [14].

Hace 17 años los FPGAs se limitaban principalmente a prototipos y a producciones en pequeños volúmenes. Hoy, debido a que ha disminuido su precio[1] e incrementado su flexibilidad, también se emplean los FPGAs en diversas aplicaciones con volúmenes grandes, donde el costo es importante, especialmente en las que requieren ser reprogramadas en tiempo real para cumplir con los estándares que continuamente están cambiando y con las demandas de los consumidores de esta era digital.

---

[1] Spartan o SpartanXL con precios desde $2.95 dólares [12]. Precios menores a $10 dólares (XCV50 de 50K compuertas, en grandes volumenes [13]. El FPGA Spartan-II XC2S150 con 150,000 compuertas tiene un precio de lista de $15 dólares en grandes volumenes [30]. Los precios han bajado de aprox. $1000 dólares en 1995 hasta menos de $100 dólares [26].

El 70% de la venta de FPGAs es consumido por aplicaciones de redes y comunicaciones [2], ya que los FPGAs son capaces de satisfacer los incesantes cambios e innovaciones en este tipo de aplicaciones, y además se pueden ajustar al amplio rango de nuevos estándares que continuamente surgen. Otras aplicaciones de los FPGAs están en tecnologías de video digital y *Bluetooth* [21]. Un ejemplo interesante lo encontramos en una grabadora de video [2] (*Personal Video Recorder*) que puede almacenar hasta 60 horas de programación de televisión. Este aparato contiene un módem que permite reconfigurarlo diariamente, por la noche. A través de una conexión a Internet se lee un flujo de bits que contiene la nueva configuración del FPGA que controla algunos de los dispositivos internos. De esta forma se pueden agregar funcionalidades a la grabadora e incluso arreglar errores sin la intervención directa de su usuario. Como consecuencia se alarga la vida útil del aparato.

También encontramos que en 2000 se fabricó una cámara digital para el mercado infantil[2] que contiene un FPGA de la familia Spartan [2]. A la fecha encontramos cómputo configurable en gran variedad de aplicaciones, tales como procesamiento de imágenes, compresión, procesos químicos por computadora, rastreo de objetos, controladores fuzzy, entre muchos otros [12]. También encontramos ejemplos en los que en un ciclo se obtienen cálculos que a un procesador le pueden llevar decenas o cientos de ciclos: algoritmos de decriptación como el RSA (*Rivest-Shamir-Adelman*), análisis de secuencias de DNA, procesamiento de señales digitales (*DSP*), emulación de microprocesadores, implementación de algoritmos criptográficos [6].

También se han empleado los FPGAs en aplicaciones como [9]:

- *Distributed Virtual Computer.* Computadora reconfigurable que permite conectar varios EVC1. Se implementó en un par de XC4013E. La reconfigurabilidad dinámica es una característica importante de este sistema, ya que permite modificar los protocolos de comunicación a través de la red de EVC1s, nodo por nodo.
- *ENABLE++.* Multiprocesador escalable implementado en FPGAs de la familia XC4000 de Xilinx. El núcleo del microprocesador realiza los cálculos, y contiene 16 FPGAs XC4013. Es expandible gracias a que tiene hasta 7 módulos en los que pueden conectarse otros dispositivos de hardware (RAMs, microprocesadores, etc.)
- *Functional Memory Computer.* Es otro microprocesador, que ocupa 13 FPGAs de la familia XC4000. Ocupa 12 de ellos como una memoria que se conecta mediante un bus de 32 bits con el FPGA restante.
- *HARP1.* Microprocesador RISC de 32 bits implementado en un Xilinx 3195 que se conecta con dos módulos RAM de 16 bits (32K).
- *RISC4005.* Microprocesador de 16 bits. Se trata de un RISC con un *pipeline* de 4 etapas. Posee 16 registros de propósito general y ocupa el 75% de un XC4005. Puede ser implementado en un FPGA de mayor tamaño, de la familia XC4000 de Xilinx.
- Entre las aplicaciones más específicas encontramos al *VA1000,* para la visualización de imágenes médicas y para generar volúmenes, implementado en dos XC4010.

Por los buenos resultados que se han obtenido se espera que haya una gran demanda por parte de la iniciativa privada en un futuro cercano. Y las universidades podrían participar en esto, al incluir en sus cursos los principios y aplicaciones de los dispositivos reconfigurables como la base de la programación estructural [12]. De esta forma podemos considerar que la tecnología de arquitectura de computadoras ya no es exclusiva de las compañías fabricantes de hardware, ahora también es accesible a los estudiantes, que pueden llegar a crear prototipos que les permitan comprender el funcionamiento de los microprocesadores y microcontroladores, empleando herramientas computacionales para el diseño de circuitos.

[2] JAMCAM 3.0 de KBGear, con un costo alrededor de 100 dólares. [15, 2]

1.3 Ambiente de Desarrollo

Todas las etapas del desarrollo de este microprocesador se realizaron mediante el ambiente del *Project Manager* de Xilinx *Foundation 4.1i*. [22] Actualmente está disponible una nueva versión de este software, ISE 4.1. Se trata de una interfaz aún más amigable, que presenta en diferentes secciones de la pantalla la jerarquía de archivos, las diferentes etapas del desarrollo con sus respectivos reportes, así como la herramienta activa en ese momento. Además cuenta con nuevos algoritmos para la implementación, en el *Place & Route*.

En primer lugar se realizó el diseño en VHDL. En una etapa posterior se realizó la Simulación Funcional de cada uno de los elementos y de todo el conjunto. Posteriormente, al no encontrar errores en el diseño, se realizó la Implementación (para convertir la descripción en alto nivel en elementos básicos dentro del FPGA).

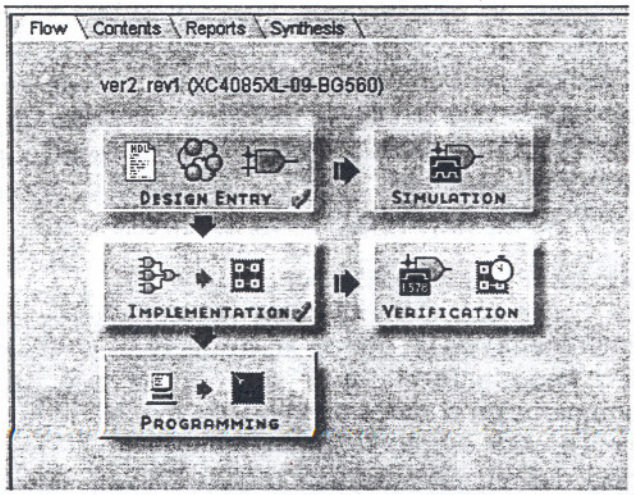


Figura 1.2 *Project Manager*

Finalmente se realizó la Simulación con retardos (*Verification*), que nos da resultados más acertados sobre el funcionamiento real de los dispositivos que hemos programado. En la figura 1.3 podemos observar el flujo del diseño a partir de la codificación en VHDL, empleando las herramientas de Xilinx - *Project Manager*.

1.3.1 Lenguaje VHDL

Los lenguajes de descripción de hardware se emplean para detallar la arquitectura y el comportamiento de sistemas electrónicos discretos, con el propósito de modelar, simular y probar dichos sistemas. Pueden soportar el empleo de varios niveles de descripción cuando por ejemplo, se combinan instrucciones estructurales con descripciones del comportamiento. De esta forma es posible describir un diseño en un nivel alto, y después refinar el diseño a nivel de componente.

Existen algunas ventajas al emplear un HDL [23]:

- Es posible verificar la funcionalidad del diseño desde las primeras etapas del diseño e incluso simularlo. Esto nos permite saber si el diseño es correcto.
- Una descripción HDL puede ser independiente de la tecnología, y por lo tanto puede ser usada posteriormente para generar este diseño en una tecnología diferente.
- VHDL, como la mayoría de los lenguajes de alto nivel, es fuertemente orientado a tipos.

- Las herramientas de Xilinx sintetizan y optimizan la lógica, para convertir automáticamente una descripción VHDL en una implementación a nivel de compuerta.

Actualmente VHDL es uno de los lenguajes para descripción de hardware que más se emplea. Este lenguaje tiene cierto parecido con C, pero no debe ser considerado un lenguaje para software, ya que sus instrucciones se enfocan a funciones de hardware. Además puede describir el procesamiento de datos de forma concurrente. Por lo tanto, el entendimiento que se logre de este lenguaje depende de nuestros conocimientos de hardware y de lenguajes de programación. VHDL facilita la capacidad de modelado a diferentes niveles: compuerta, registro o sistema. Además es posible describir un sistema digital previamente representado por un diagrama de bloques, máquina de estados finitos o tabla de verdad [1]. Originalmente fue diseñado para modelar el comportamiento de hardware ya existente, no para especificar la función de nuevos dispositivos [29]. Por lo tanto hay algunas instrucciones de VHDL no sintetizables, por ejemplo: los que incluyen especificaciones de tiempo (*wait, after*), las estructuras que no tienen un tamaño definido (arreglos de tamaño *n*, árboles dinámicos) y los números de punto flotante.

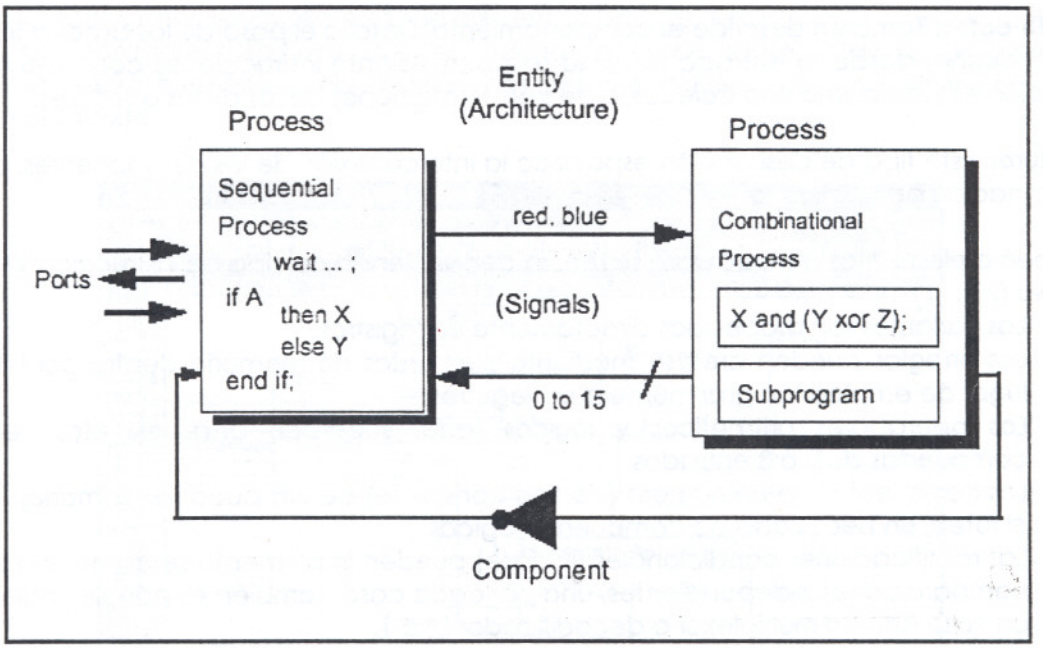


Figura 1.3 Modelo de hardware descrito en VHDL [23]

La síntesis ocurre cuando una herramienta automatizada convierte una descripción de alto nivel en hardware, de forma análoga a un compilador que transforma un programa de software en lenguaje máquina. En 1999 la IEEE definió un nuevo estándar con un subconjunto de sentencias VHDL sintetizables.

VHDL describe una entidad [1, 17, 23], ya sea componente o circuito, como una parte visible (nombre de la entidad y sus conexiones) y otra invisible (el algoritmo de la entidad y sus interconexiones). Luego de haber definido una entidad se convierte en parte de una biblioteca, y por lo tanto es posible reutilizarla en ese diseño o en otro. Una entidad VHDL tiene una o más entradas y salidas que se pueden conectar con otros dispositivos. Está compuesta por procesos o componentes que operan de forma concurrente y que se comunican a través de señales. Estas señales tienen una fuente (*driver*) y uno o más destinos, además de un tipo. Además, cada entidad (Fig. 1.4) está definida por una declaración de

señales y por una arquitectura, que comprende procesos con operaciones aritméticas o lógicas, asignación de señales o instrucciones para instanciar componentes, e incluso llamadas a subprogramas. Como parte de esta descripción del comportamiento de la entidad pueden existir una forma de inicialización (*reset event*), de sincronización (*clock event*) y de asignación de las señales de salida (*output event*).

VHDL es un lenguaje fuertemente orientado a tipos. Cada señal interna, externa o variable tiene un tipo definido. Existe gran variedad de tipos, entre los que podemos mencionar: ***bit, bit_vector, std_logic, std_logic_vector, integer.***

Las instrucciones de VHDL se dividen en categorías de acuerdo a su nivel de abstracción [23, 25]. Estos niveles pueden combinarse:

Algorítmica. Describe el comportamiento del diseño. Expresa los aspectos funcionales o algorítmicos del diseño, describiendo la transformación de señales. A través de instrucciones secuenciales detalla la función del diseño, pero no necesariamente cómo la realiza. No es visible la forma en que las funciones de entrada/salida se descomponen en funciones elementales más pequeñas.

Flujo de datos. También describe el comportamiento. Detalla el paso de los datos a través de todo el diseño, desde su entrada hasta la salida, mediante instrucciones concurrentes. Una operación es vista como una colección de transformaciones de los datos o señales.

Estructural. Este tipo de descripción especifica la interconexión de los componentes, a través de la creación de instancias.

El mapeo a elementos de hardware se realiza dependiendo del tipo de instrucción VHDL, por ejemplo [20]:

- Las variables son mapeadas directamente en registros.
- Los arreglos pueden crearse mediante elementos de memoria dentro del FPGA, en lugar de emplear un gran número de registros.
- Los operadores aritméticos y lógicos (*add, substract, and, or,* etc.) requieren compuertas de 2 ó 3 entradas
- Los corrimientos de un número (de tipo constante) de bits pueden ser manejados por el ruteo, sin necesidad de compuertas lógicas.
- Las ramificaciones condicionales (*if, case*) pueden implementarse como una serie de comparaciones independientes, una por cada caso. También es posible implementar un *case* con un multiplexor o decodificador 2^n *a 1*.

1.3.2 Herramientas de Diseño e Implementación

El *Schematic Editor* [22] permite diseñar un circuito con elementos predefinidos que se encuentran en bibliotecas de Xilinx y con elementos nuevos de la biblioteca del proyecto actual. La interfaz está integrada con el *HDL Editor* y con el *Logic Simulator*, lo que facilita tanto la etapa de diseño como la de pruebas. Los diseños pueden abarcar varias hojas; también pueden tener varios niveles de jerarquía o uno solo. En la siguiente figura se muestra un diseño con varios niveles jerárquicos.

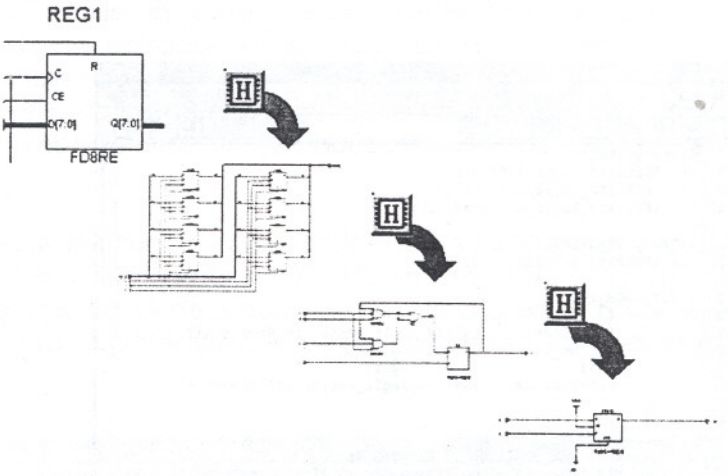


Figura 1.4 Diseño jerárquico de un registro de 8 bits

En el *Schematic Editor* hay bibliotecas de elementos básicos, tales como compuertas lógicas, *buffers*, comparadores, *flip-flops*, *latches*, memorias y contadores. Hay varias formas de diseñar un nuevo elemento; una de ellas nos permite definir cada uno de los pines que integran el circuito.

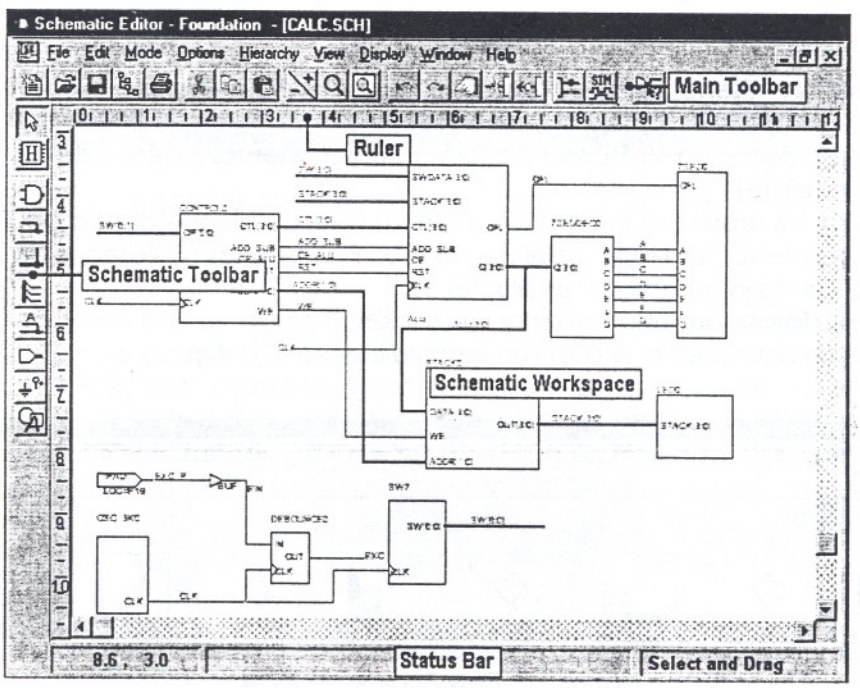


Figura 1.5 Interfaz del *Schematic Editor* de Xilinx

Después de haber descrito la estructura del nuevo elemento, mediante el *HDL Editor* es posible definir la función que éste ejecuta, mediante sentencias VHDL. Los diferentes elementos del diseño pueden conectarse por medio de líneas y buses, o por nombre (y en tal caso las conexiones no son visibles, sólo el nombre de las señales). Esta interfaz nos permite verificar la presencia de errores de sintaxis antes de sintetizar el diseño. Esto significa que se convierten descripciones VHDL a un formato interno de Xilinx al sintetizar. Las

descripciones VHDL se traducen en lógica, en forma de un bloque de lógica combinatoria, un *latch* o un registro.

Figura 1.6 Interfaz del *HDL Editor* de Xilinx

Al implementar un diseño en el FPGA, Xilinx sigue el *Flow Engine*, cuyas etapas son las siguientes:

- Las instrucciones en VHDL se transforman en elementos básicos de las bibliotecas de Xilinx (*Translate*)
- Optimizar las representaciones del diseño a nivel de bloque dentro del FPGA (*Map*)
- Mapear la estructura lógica del diseño a un FPGA de las bibliotecas de Xilinx (*Place*)
- Configurar las interconexiones (*Route*)
- Generar datos para una posterior simulación (*Timing*)
- Traducir el diseño en el FPGA a un archivo binario (*Configure*).

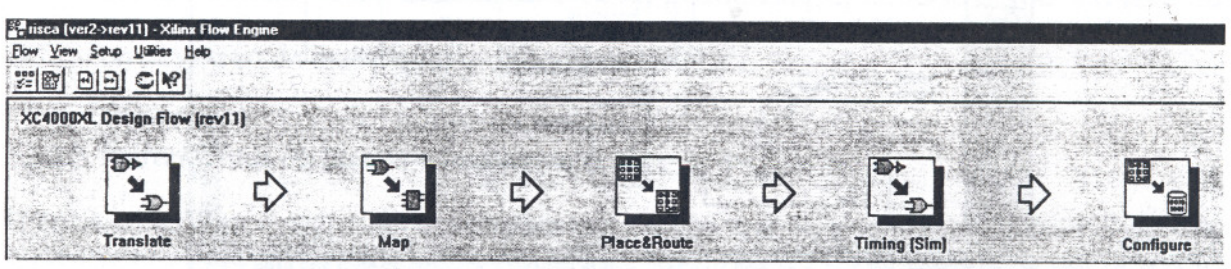


Figura 1.7 *Flow Engine* de Xilinx

1.3.3 Herramienta para Pruebas

El *Logic Simulator* o Simulador es una herramienta que nos permite verificar el funcionamiento del diseño antes de implementarlo [22]. Generalmente se realiza después de haber completado un diseño en el *Schematic Editor*. A través de esta interfaz se indican las señales a visualizar, y un medio práctico para ello es el *Script Editor*. Un bus puede aparecer como tal

o como múltiples líneas de un bit, también se puede indicar en qué notación se mostrará la señal; a las señales de entrada se les puede asignar valores. Se define la duración del período de las señales de reloj; incluso se puede elegir simular solamente una parte del diseño. Esta interfaz permite generar resultados que también pueden ser visibles al mismo tiempo en el *Schematic Editor*, lo que puede resultar de mucha ayuda en la etapa de pruebas.

La simulación puede ser Funcional o considerar los retardos ocasionados por la lectura o escritura, o por la transmisión de una señal de un dispositivo a otro (*Timing Simulation*). Así se puede verificar la velocidad de ejecución del circuito en el FPGA seleccionado. También pueden detectar rutas o *paths* críticos, o la existencia de violaciones de los tiempos de *set-up* o *hold*.

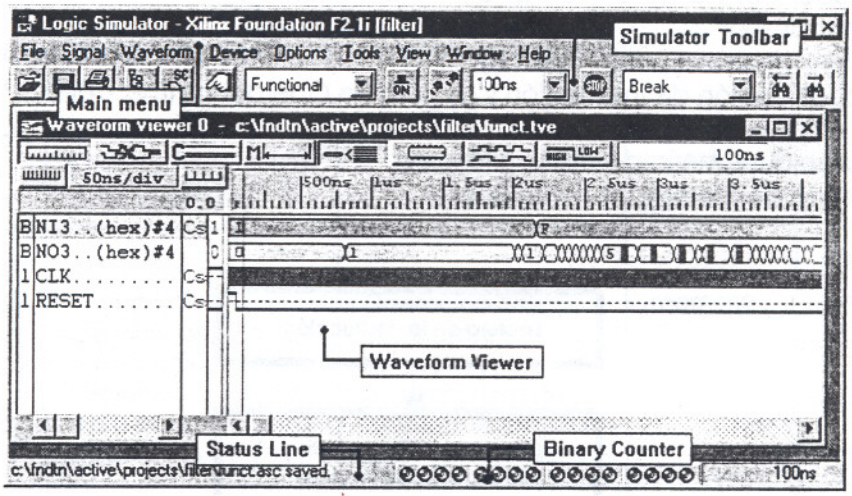


Figura 1.8 Interfaz del *Logic Simulator* de Xilinx

A lo largo del capítulo hemos descrito los RISC y la ejecución en *pipeline*, ya que son características propias del núcleo del microprocesador que se desarrolló en este proyecto. Además hemos hablado del cómputo reconfigurable y de las múltiples aplicaciones que actualmente tiene y que potencialmente tendrá a futuro. A partir de esto podemos afirmar que los FPGAs se convertirán en un elemento de gran importancia en el desarrollo tecnológico, con implicaciones a nivel académico, industrial y comercial.

Capítulo 2 Diseño del UAM – RISC II

2.1 Descripción del UAM – RISC II

2.1.1 Las Etapas del *Pipeline*

El diseño y programación de este microprocesador se basan en la descripción de la UAM - RISC [7], una arquitectura de 32 bits desarrollada en el departamento de Ingeniería Eléctrica de la UAM con fines didácticos orientados hacia el entendimiento de los conceptos básicos de una arquitectura RISC, así como de su funcionamiento.

El microprocesador consta de 5 etapas, que le permiten ejecutar 5 instrucciones de forma paralela. Las etapas son:

1. Lectura de la Instrucción (*Fetch*)
2. Decodificación de Instrucción y Lectura de Operandos (*Decode*)
3. Ejecución (*Execute*)
4. Acceso a memoria de datos (*Memory*)
5. Escritura de Resultados (*Write Back*)

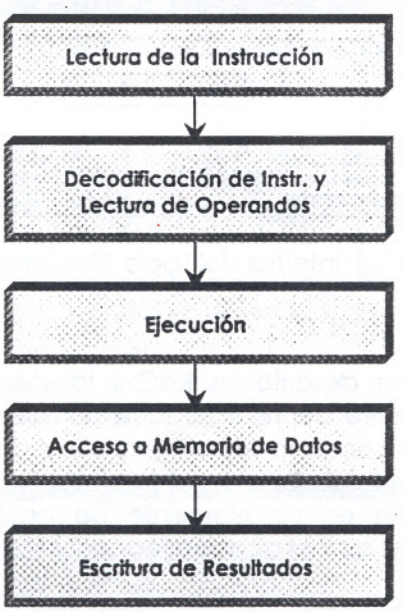


Figura 2.1 *Pipeline* lineal

Para describir el comportamiento del *pipeline* podemos analizar las primeras cinco instrucciones leídas de memoria. En el primer ciclo de reloj (T1) la primera instrucción (I1) se encuentra en la etapa *Fetch*. En el siguiente ciclo ésta pasa a la etapa de Decodificación, al mismo tiempo que otra instrucción se encuentra en Lectura. En el tercer ciclo de reloj ya hay 3 instrucciones ejecutándose.

A partir del quinto ciclo se ejecutan **5 instrucciones en paralelo**, ya que hay una instrucción en cada etapa del pipeline, por ejemplo, mientras se escribe el resultado de I1, también se realiza la Lectura de I5. **En cada uno de los ciclos subsiguientes se obtendrá el resultado de una instrucción**: en T6 se escribe el resultado de I2, en T7 el resultado de I3, y así sucesivamente. Esto lo podemos ver en la figura 2.2.

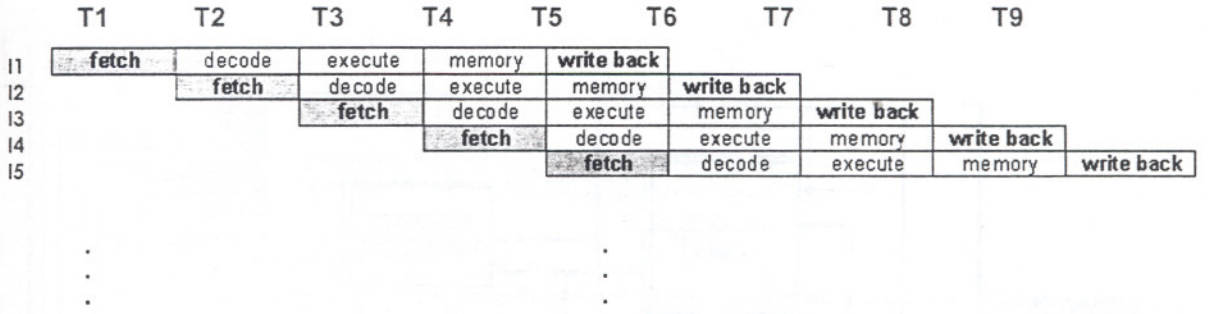


Figura 2.2 Secuencia de instrucciones dentro del *pipeline*

Las operaciones de cada etapa se mencionan a continuación:

1 LECTURA ***(fetch)***	**2 DECODIFICACIÓN** ***(decode)***	**3 EJECUCIÓN** ***(execute)***	**4 MEMORIA** ***(memory)***	**5 ESCRITURA** ***(write back)***
- *La unidad de* conteo genera una dirección - Lectura de instrucción - Se detecta la dependencia de datos	- *Se decodifica la* instrucción - Decisión de ramificación - *Lectura de registros* para obtener operandos	- *Operación de la ALU* - Operación del comparador - *Se resuelve la dependencia de* datos, seleccionando operandos para ALU o comparador	- Acceso a memoria	- Actualización de Registros

La estructura del microprocesador emplea 4 Registros del *pipeline* y 16 unidades principales:

- unidad de conteo
- memoria de datos
- memoria de instrucciones
- unidad de dependencia de datos (*Forward Unit*)
- unidad de control
- bloque de registros de propósito general
- unidad aritmética y lógica (ALU)
- comparador
- 8 multiplexores

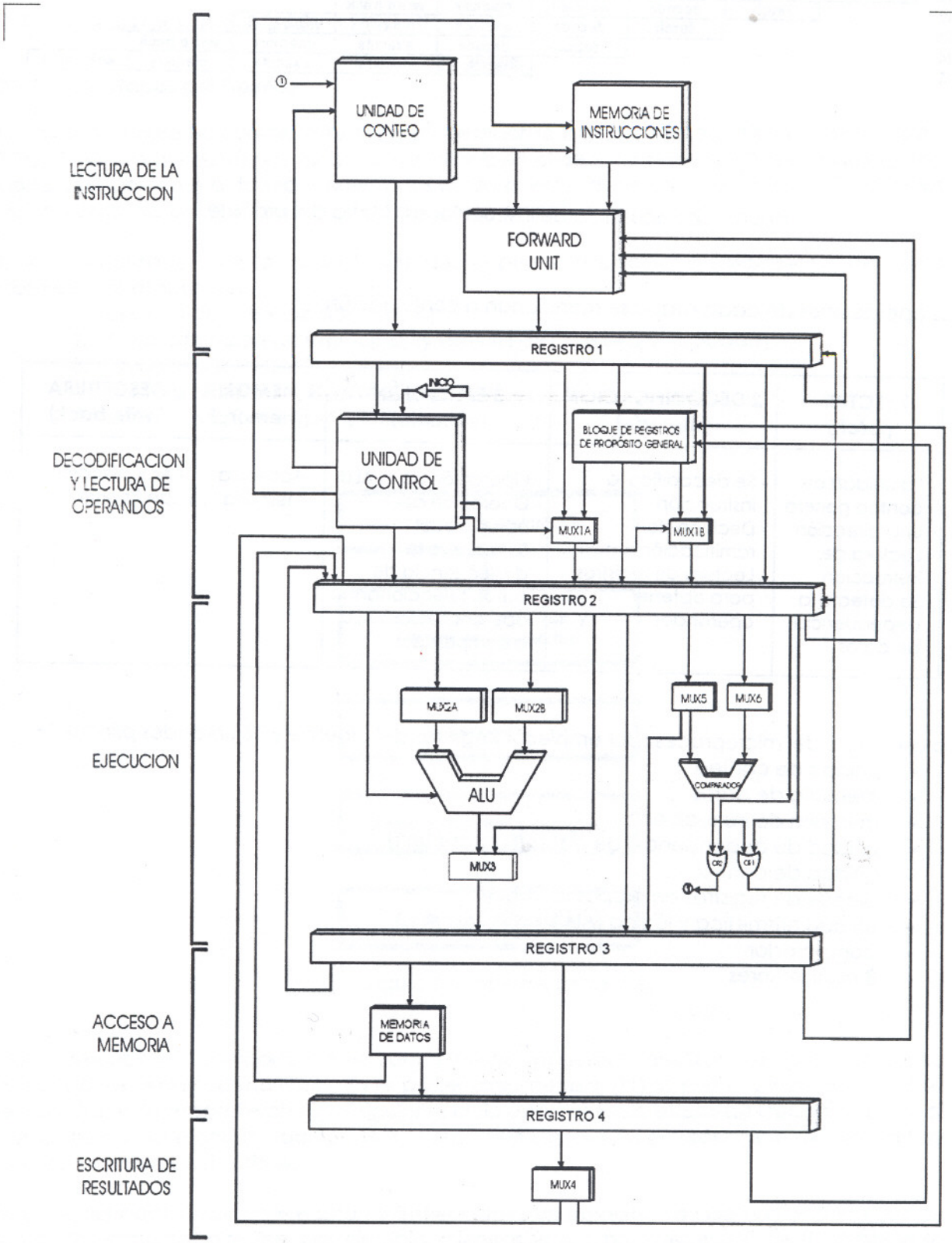


Figura 2.3 Diagrama de bloques

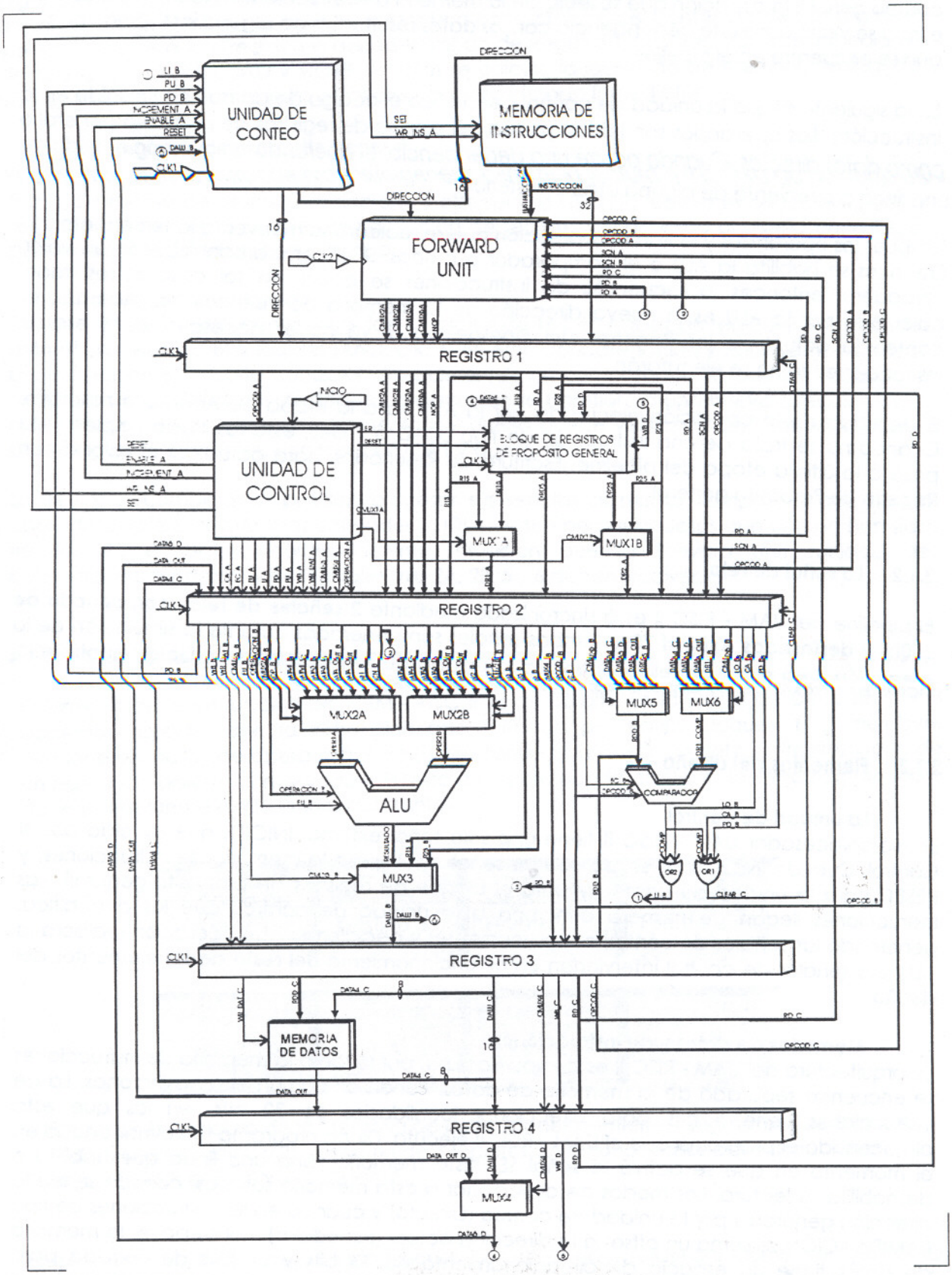


Figura 2.4 Diagrama del microprocesador UAM - RISC II

La lectura de cada nueva instrucción se realiza en la etapa *Fetch*, ya que la unidad de conteo genera la dirección que se leerá de la memoria de instrucciones. Además, en esta etapa se verifica si existe dependencia con los datos resultantes de alguna instrucción que aún se encuentre en el *pipeline*.

En la siguiente etapa la unidad de control decodifica el código de operación (incluido en la instrucción). Los operandos son leídos, ya sea del bloque de registros de propósito general o como datos directos. Cuando ocurre una dependencia, el operando en cuestión es leído de una línea proveniente de alguna etapa posterior.

Después los operandos y el tipo de operación que se realizará se mueven a la tercera etapa. Entonces se habilita la ALU o el comparador o ambos. Si el comparador obtiene un valor verdadero, entonces la secuencia de instrucciones se altera. En tal caso el resultado calculado por la ALU es la nueva dirección de la memoria de instrucciones. Además, el contenido actual de los Registros del *Pipeline* 1 y 2 ya no es necesario, pues esas 2 instrucciones ya no se ejecutarán.

El resultado de la operación ejecutada por la ALU va a la etapa de Acceso a memoria. Cuando no se trata de una instrucción de tipo LOAD o STORE, este resultado simplemente pasa a la última etapa del *pipeline*, Escritura de Resultados, para actualizar el valor de un Registro de Propósito general.

2.1.2 La señal de reloj

El *pipeline* del UAM - RISC II está sincronizado mediante **2 señales de reloj, con período de 250ns y desfasadas entre sí 70ns.** Ambas señales son generadas durante la simulación de la ejecución. Una señal de reloj controla a la Unidad de Dependencia de Datos; el otro reloj sincroniza al resto de los dispositivos, incluyendo a los Registros del *pipeline*.

2.1.3 Elementos del diseño

La unidad de control

El microprocesador UAM - RISC II tiene una sola señal externa, INICIO, que es leída por el Control. Cuando INICIO= '1' se activan las señales SET (para la memoria de instrucciones) y RESET (para la unidad de conteo y para el bloque de registros de propósito general). Las instrucciones llegan de manera asíncrona a la unidad de control, que las decodifica, generando las señales de control correspondientes y decidiendo qué operación realizará la ALU. Las señales de control intervienen en el funcionamiento del resto de los elementos del diseño.

Memorias de datos y de instrucciones

La arquitectura del UAM - RISC II es de tipo *Harvard*, por lo tanto la memoria de instrucciones se encuentra separada de la memoria de datos. Tienen en común el ser asíncronas. La de Instrucciones tiene 5 bits para direccionar localidades de 32 bits, en las que está almacenado el programa que el UAM - RISC II ejecuta. Dicho programa fue almacenado en el momento en que se activó la señal SET. Esta memoria tiene una línea que habilita o deshabilita su lectura. Los modos de direccionar a esta memoria son dos: cuando se lee la dirección generada por la unidad de conteo (Directo) y cuando en las instrucciones de tipo RAMIFICACIÓN se suma un offset a la dirección actual (Inmediato). Por su parte, la memoria de datos tiene un espacio de direccionamiento de 16 bits y un bus de entrada para actualizar el contenido de localidades de 8 bits. También tiene una línea de habilitación que da la posibilidad de leer o escribir en ella. Existen tres modos de direccionamiento: en

algunas instrucciones los últimos 16 bits representan una localidad de la memoria de datos (modo Directo); también es posible realizar una operación con el contenido de dos registros, y el resultado será la dirección a acceder (modo de Registros); hay instrucciones inmediatas de tipo OPERACIÓN, LOAD y STORE, en ellas se calcula la localidad de memoria realizando una operación entre un registro y un valor constante (modo Inmediato).

Los registros de propósito general

El conjunto de 32 registros de propósito general permite la lectura o escritura de palabras de 8 bits. El proceso de escritura es síncrono y obedece a CLK1; la lectura es asíncrona, pues se realiza en el momento que llega la dirección del registro. De acuerdo al diseño en VHDL, es posible leer 3 registros al mismo tiempo.

La ALU y el comparador

El UAM - RISC II cuenta con una unidad funcional de 16 bits, capaz de realizar 8 instrucciones aritméticas (ADD, SUB) y lógicas (AND, OR, XOR, NOT, concatenación y corrimientos lógicos). Las instrucciones de salto condicional requieren de un comparador, ya que sólo cuando la comparación realizada es verdadera la ejecución se transfiere a una nueva dirección de la memoria de instrucciones. Este elemento del diseño puede verificar si dos cantidades de 8 bits son iguales, diferentes, una mayor o menor que la otra.

La ALU y el comparador pueden procesar sus señales al mismo tiempo, ya que tienen operadores diferentes. Ambas entidades VHDL pueden necesitar datos que aún no han sido escritos en el registro correspondiente, situación que la *Forward Unit* (unidad de dependencia de datos) y los multiplexores 2A, 2B, 5 y 6 deben resolver.

2.1.4 El conjunto de instrucciones

El formato de la instrucción del UAM - RISC II es de 32 bits. En la Etapa de Decodificación podemos apreciar que la instrucción está formada por varios bloques. El primer byte contiene el código de operación, el segundo contiene la dirección del registro destino (o de un operando para el comparador). El resto de los bits puede contener la dirección de uno o dos registros fuente u operandos constantes.

Figura 2.5 La palabra de la Memoria de Instrucciones contiene 4 segmentos de 8 bits: el código de operación y la dirección de 3 registros. En algunos casos estas direcciones son sustituidas por valores constantes.

El microprocesador posee 31 instrucciones que se pueden clasificar como de tipo LOAD, STORE, OPERACIÓN y RAMIFICACIÓN. Requieren 1 ó 2 operandos, y pueden llegar a utilizar la ALU y el comparador simultáneamente; en tal caso se requieren 4 operandos (los 3 bytes finales de la instrucción actual y la dirección actual).

El conjunto de instrucciones del UAM - RISC II se puede clasificar de la siguiente manera:

Categoría	Instrucción básica
Aritméticas	ADD, SUB, NOT
Lógicas	AND, OR, XOR
Corrimiento	SLL, SRL
Movimiento de datos	LOAD, STORE
Salto	BEQ, BGA, BLE, BNEQ, JUMP, CALL, RETURN

Tabla 2. 1 Instrucciones por categoría

Las características de las 31 instrucciones se presentan a continuación:

CÓDIGO DE OPERACIÓN	INSTRUCCIÓN	DESCRIPCIÓN	OPERACIÓN QUE REALIZA
TIPO OPERACIÓN			
00001010	ADD	Adición de 2 registros	(RD) <= (R1S) + **(R2S)**
00001011	ADDI	Adición de 1 registro y 1 valor constante	(RD) <= (R1S) + **R2S**
00000100	AND	AND de 2 registros	(RD) <= (R1S) AND **(R2S)**
00000101	ANDI	AND de 1 registro y 1 valor constante	(RD) <= (R1S) AND **R2S**
00011010	NOT	Inversión de los bits de un registro	(RD) <= NOT **(R2S)**
00011011	NOTI	Inversión de los bits de un valor constante	(RD) <= NOT **R2S**
00000110	OR	OR de 2 registros	(RD) <= (R1S) OR **(R2S)**
00000111	ORI	OR de 1 registro y 1 valor constante	(RD) <= (R1S) OR **R2S**
10000111	SLL	Corrimiento lógico a la izquierda de n bits de un registro	(RD) <= (R1S) SLL **R2S**
10001000	SRL	Corrimiento lógico a la derecha de n bits de un registro	(RD) <= (R1S) SRL **R2S**
00000010	SUB	Sustracción entre 2 registros	(RD) <= (R1S) - **(R2S)**
00000011	SUBI	Sustracción entre 1 registro y 1 valor constante	(RD) <= (R1S) - **R2S**
00001000	XOR	XOR de 2 registros	(RD) <= (R1S) XOR **(R2S)**
00001001	XORI	XOR de 1 registro y 1 valor constante	(RD) <= (R1S) XOR **R2S**

TIPO LOAD			
10000001	LOAD	Almacena una palabra de memoria de datos en un registro	(RD) <= MEMORIA[**(R1S)** & **R2S**]
10000000	LOEX		(RD) <= MEMORIA[**(R1S)** & **(R2S)**]
10000010	LOFF		(RD) <= MEMORIA[**R1S** & **R2S**]
TIPO STORE			
10000101	STORE	Almacena un registro en una localidad de memoria de datos	MEMORIA[**(R1S)** & **R2S**] <= **(RD)**
10000100	STOEX		MEMORIA[**(R1S)** & **(R2S)**] <= **(RD)**
10000110	STOFF		MEMORIA[**R1S** & **R2S**] <= **RD**
TIPO RAMIFICACIÓN			
00010000	BEQ	Cambio de dirección cuando dos registros son iguales	if (RD) = **(R1S)** then DIRECCION <= SCN + R2S
00010001	BEQI	Cambio de dir. cuando un registro y una constante son iguales	if (RD) = **R1S** then DIRECCION <= SCN + R2S
00010100	BGA	Cambio de dirección cuando un registro es mayor que otro	if (RD) > **(R1S)** then DIRECCION <= SCN + R2S
00010101	BGAI	Cambio de dirección cuando un registro es mayor que una constante	if (RD) > **R1S** then DIRECCION <= SCN + R2S
00010110	BLE	Cambio de dirección cuando un registro es menor que otro	if (RD) < **(R1S)** then DIRECCION <= SCN + R2S
00010111	BLEI	Cambio de dirección cuando un registro es menor que una constante	if (RD) < **R1S** then DIRECCION <= SCN + R2S
00010010	BNEQ	Cambio de dirección cuando dos registros no son iguales	if (RD) <> **(R1S)** then DIRECCION <= SCN + R2S
00010011	BNEQI	Cambio de dirección cuando un registro y una constante no son iguales	if (RD) <> **R1S** then DIRECCION <= SCN + R2S
00011000	CALL	Llamada a subrutina	$SC_n = SC_{n+1}$, DIRECCION:= R1S & R2S
00011001	JUMP	Cambio de dirección	DIRECCION <= SCN + R2S
1110000	RETURN	Regreso de una subrutina	$SC_n = SC_{n-1}$, si n >= 1 $SC_n = SC_0$, si n = 0

Tabla 2. 2 Descripción de las instrucciones

2.1.5 Instrucciones tipo OPERACIÓN

Los 8 bits más significativos contienen el código de operación que la unidad de control decodifica. Los bits 23 a 16 contienen a la señal RD_A, que es la dirección del registro en que se escribirá el resultado final. Los bits 15 a 8 pertenecen a la señal R1S_A, dirección del registro que contiene el primer operando. El último byte es la señal R2S_A. La palabra completa es de tipo std_logic_vector, con 32 bits.

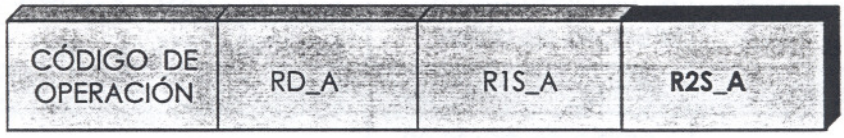


Figura 2.6 Formato de las instrucciones tipo OPERACIÓN

Las operaciones que caen dentro de esta categoría son:

- ADD
- ADDI
- AND
- ANDI
- OR
- ORI
- SLL
- SRL
- SUB
- SUBI
- XOR
- XORI
- NOT
- NOTI

Las siguientes instrucciones necesitan un par de operandos, y puede ocurrir que el segundo operando sea el contenido de un registro o un valor constante (Fig. 2.7). En otras palabras la señal R2S_A puede indicar la dirección de un registro que posteriormente será leído o puede ser un valor inmediato. El resultado de estas instrucciones también es un std_logic_vector de 8 bits, pero se le agregan ocho 0's a la izquierda para formar una palabra de 16 bits.

Figura 2.7 Formato de las instrucciones ADD, ADDI, AND, ANDI, OR, ORI, SUB, SUBI, XOR, XORI

- **ADD, ADDI**

Este par de instrucciones realiza la adición de dos señales de tipo std_logic_vector. Los paréntesis indican que debe leerse el contenido de ese registro; por ejemplo **(R2S_A)** implica que debe leerse el contenido del registro número R2S_A; cuando no está entre paréntesis **R2S_A** indica que esta señal contiene ya el operando.

ADD (RD_A) <= (R1S_A) + **(R2S_A)**
ADDI (RD_A) <= (R1S_A) + **R2S_A**

- **AND, ANDI**

Realizan el AND entre dos bytes de tipo std_logic_vector.

AND (RD_A) <= (R1S_A) AND **(R2S_A)**
ANDI (RD_A) <= (R1S_A) AND **R2S_A**

- **OR, ORI**

Realizan la operación OR de dos std_logic_vector.

OR	(RD_A) <= (R1S_A) OR **(R2S_A)**
ORI	(RD_A) <= (R1S_A) OR **R2S_A**

- **SUB, SUBI**

Realizan la sustracción entre dos bytes de tipo std_logic_vector. El primer operando debe ser mayor o igual que el segundo; en caso contrario el resultado de la sustracción no será un número negativo, sino "00000000".

SUB	(RD_A) <= (R1S_A) - **(R2S_A)**
SUBI	(RD_A) <= (R1S_A) - **R2S_A**

- **XOR, XORI**

Realizan el XOR de dos bytes de tipo std_logic_vector.

XOR	(RD_A) <= (R1S_A) XOR **(R2S_A)**
XOR I	(RD_A) <= (R1S_A) XOR **R2S_A**

Las instrucciones de corrimiento realizan el corrimiento de n bits. El resultado de estas instrucciones también es un std_logic_vector de 8 bits. El primer byte, contiene la dirección RD_A del registro donde se almacenará el resultado del corrimiento. El siguiente bloque de 8 bits contiene la dirección del registro al cual se aplicará el corrimiento. El número de bits n que se correrán está representado por el valor R2S_A.

Figura 2.8 Formato de las instrucciones SLL, SRL

- **SLL, SRL**

Estas instrucciones realizan el corrimiento de n bits sobre el contenido de un registro. El valor **R2S_A** indica el número n de bits que se corren en **(R1S_A)**.

SLL	(RD_A) <= (R1S_A) SLL **R2S_A**
SRL	(RD_A) <= (R1S_A) SRL **R2S_A**

Las instrucciones para invertir el valor de un std_logic_vector de 8 bits son NOT y NOTI. El resultado de estas instrucciones es del mismo tipo.

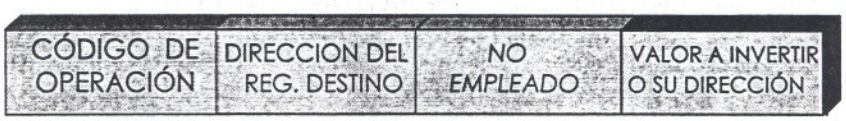


Figura 2.9 Formato de las instrucciones NOT, NOTI

- **NOT, NOTI**

Este par de instrucciones realiza la negación de un std_logic_vector de 8 bits. En la instrucción NOT se invierten los bits del valor contenido en el registro **R2S_A.** En la instrucción NOTI se invierte el valor constante **R2S_A**; en ambas el resultado se almacena en el registro **RD_A.**

NOT (RD_A) <= NOT **(R2S_A)**
NOTI (RD_A) <= NOT **R2S_A**

Para comprender mejor cómo se ejecutan las instrucciones OPERACIÓN vamos a describir su procesamiento en cada una de las etapas. Para ello tomemos como ejemplo XORI.

XORI - Etapa 1

La ejecución de esta etapa ocurre de forma similar con cualquier tipo de instrucción. En caso de que la señal PU_B esté activa en el flanco de subida de CLK1, el Subcontador de Rutina se cambia por el siguiente y empieza a contar a partir del número leído de SCN_C[15:0]. Si PD_B está activa se cambia al contador anterior. Pero si ambas son igual a 0 la secuencia de conteo permanece en el mismo contador, que en este caso sería el 0. Ya sea que haya cambio o no de contador, en el flanco de subida de CLK1 la unidad de conteo genera DIRECCION[15:0] si están habilitadas las señales INCREMENT_A y ENABLE_A. Supongamos que el valor de DIRECCION[15:0] es "0000000000001001" = 9_{10}.

La señal DIRECCION[15:0] es leída por la memoria de instrucciones, y si WR_INS_A = '0' se realiza la lectura de dicha localidad para obtener la INSTRUCCION[31:0], que en este caso es "00001001_00000111_00000101_00000001".

Cuando ocurre el flanco de subida de CLK2 la *Forward Unit* verifica si existe una dependencia de datos con otra instrucción dentro del *pipeline*. Para ello lee la INSTRUCCION[31:0], la DIRECCION[15:0] actual y las direcciones de las 2 instrucciones anteriores: SCN_A[15:0] y SCN_B[15:0]. Al leer la instrucción actual se incluye el código de operación: "00001001". También es necesario identificar el código de las 3 instrucciones anteriores: OPCOD_A[7:0], OPCOD_B[7:0] y OPCOD_C[7:0], así como la dirección de sus registros destino: RD_A[7:0], RD_B[7:0] y RD_C[7:0].

Supongamos ahora que ninguna de estas instrucciones ocasiona dependencia de datos, entonces la *Forward Unit* generará los siguientes valores: NOP= '0', CMUX2A[2:0]= "010", CMUX2B[2:0]= "010", CMUX5A[1:0]= "10", CMUX6A[1:0]= "10".

Posteriormente, en el flanco de subida de CLK1 el Registro 1 del *pipeline* almacena las señales correspondientes a los multiplexores 2A, 2B, 5A y 6A, así como NOP; también almacena la instrucción y su dirección. Si CLEAR_C estuviera habilitada, todas las señales de salida serían '0's y esto sería consecuencia de que otra instrucción, en la etapa 4, provocó un salto en la ejecución. Pero si suponemos que CLEAR_C= '0', entonces el valor contenido en estas señales pasará a otras señales de salida que tienen la terminación "_A". De esta forma el Registro 1 lee INSTRUCCIÓN[31:0], DIRECCION[15:0], CMUX2A[2:0]= "010", CMUX2B[2:0]= "010", CMUX5A[1:0]= "10", CMUX6A[1:0]= "10" y NOP= '0'.

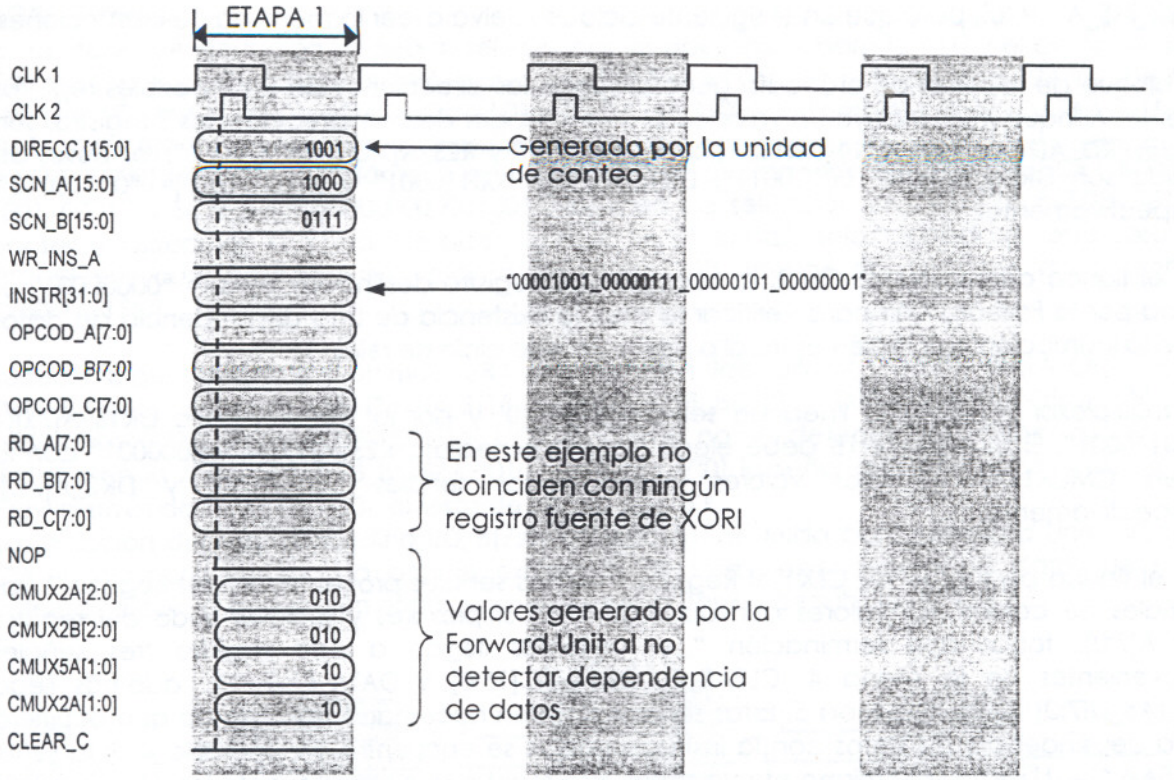


Figura 2.10 Diagrama de la Etapa 1 de XORI

XORI - Etapa 2

Cuando el Registro 1 lee la instrucción la descompone en varias señales: OPCOD_A[7:0]= "00001001", RD_A[7:0]= "00000111", R1S_A[7:0]= "00000101", R2S_A[7:0]= "00000001"; su dirección se propaga a SCN_A[15:0]= "00000000_00001001".

En esta fase el elemento más importante es la unidad de control. La instrucción actual no es la primera en ejecutarse, así que la señal INICIO debe estar inactiva y por lo tanto las señales SET y RESET también permanecen inactivas. Cuando la unidad de control lee OPCOD_A[7:0]= "00001001" determina que la operación que la ALU realizará es XOR y genera OPERACION_A[3:0]= "0100". También genera las señales de control correspondientes a esta instrucción:

CA_A = '0', porque XORI no es una instrucción de salto.
CMUX1A = '0', porque el 1er operando de XORI es el contenido del registro R1S_A[7:0].
CMUX1B = '1', porque el 2° operando de XORI es el valor constante "00000001".
CMUX3_A = '0', pues el mux. 3 seleccionará el resultado de la operación XORI.
CMUX4_A = '1', pues el mux. 4 seleccionará el resultado para actualizar el reg. destino.
EC_A = '0', ya que XORI no necesita el comparador.
ENABLE_A = '1', para que la unidad de conteo permanezca habilitada.
ER = '1', para habilitar la lectura del registro.
EU_A = '1', para habilitar a la ALU.
INCREMENT_A = '1', para que la unidad de conteo permanezca generando instrucciones.
LI_A = '0', porque XORI no va a generar una nueva dirección para la U. de Conteo
PD_A = '0', porque XORI no implica regresar al contador anterior.
PU_A = '0', porque XORI no implica cambiar al siguiente contador.
WB_A = '1', para habilitar posteriormente la escritura en el registro destino "00000111"

WR_DAT_A = '0', ya que no se escribirá en la memoria de datos.
WR_INS_A = '0', para que en el siguiente ciclo se vuelva a leer la memoria de instrucciones

Al bloque de registros de propósito general llegan las direcciones de los 3 posibles registros fuente. Aunque XORI incluye un valor constante en lugar de una dirección, los 3 registros son leídos: RD_A[7:0]= "00000111", R1S_A[7:0]="00000101" y R2S_A[7:0]= "00000001"; los buses de salida son DRDS_A[7:0]= "00110011", DR1S_A[7:0]= "00110001" y DR2S_A[7:0]= "00110010", respectivamente.

En el flanco de subida de CLK2, la dirección del registro destino RD_A[7:0]= "00000001" es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos con la instrucción que recién entró al *pipeline* en este ciclo de reloj.

El multiplexor 1A tiene la línea de selección en '0' y por lo tanto elige a DR1S_A[7:0]= "00110001". El multiplexor 1B debe elegir al dato inmediato R2S_A[7:0]= "00000001" con la línea CMUX1B= '1'. Estos valores pasan a las señales DR1_A[7:0] y DR2_A[7:0], respectivamente.

En el flanco de subida de CLK1 el Registro 2 lee las señales provenientes del Registro 1, las señales de control, los valores derivados de los multiplexores y el valor leído del registro RD_A[7:0], todas con terminación "_A". Además llegan a este registro tres señales provenientes de la etapa 4: CLEAR_C, DATA4_C[15:0] y DATA_OUT[7:0]; además llega DATA5_D[7:0] desde la etapa 5. Estas señales pueden necesitarse en caso de que ocurriera una dependencia de datos con la instrucción que se encuentra en la etapa 4. Si la señal CLEAR_C= '1', tendría el mismo efecto sobre este registro que sobre el anterior. Al convertirse en salidas del Registro 2 cada una de las señales tomará el sufijo "_B".

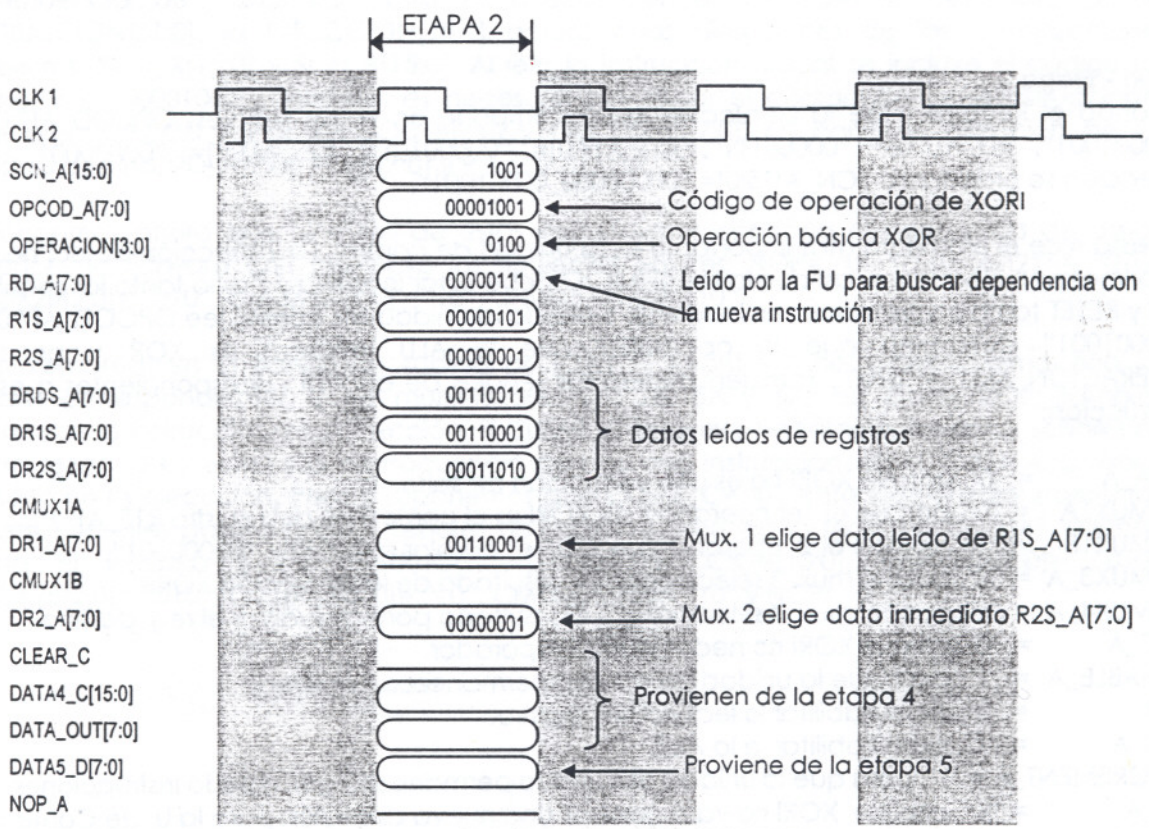


Figura 2.11 Diagrama de la Etapa 2 de XORI

XORI - Etapa 3

En la Etapa de Ejecución se deben seleccionar los operandos para la ALU y el comparador. Esto se logra gracias a los multiplexores 2A, 2B, 5 y 6. Por ejemplo, el mux. 2A recibe las señales DATA4_C[15:0], DATA_OUT[7:0] y DATA5_D[7:0]. También lee DATA4_B[15:0], DATA_OUT_B[7:0] (ambas de la instrucción que en este momento ya está en etapa 5), DATA5_B[7:0] (de la instrucción que en este flanco de subida salió del *pipeline*), DR1_B[7:0]= "00110001" y SCN_B= "0000000000001001". Su línea de selección es CMUX2A_B[2:0]= "010" por ser la instrucción XORI, ya que además NOP_B= '0', el dato seleccionado en este caso es DR1_B[7:0], que propaga su valor a los ocho bits menos significativos de la señal OPER1A[15:0].

Sucede algo similar con el mux. 2B, que también lee DATA4_C[15:0], DATA_OUT[7:0] y DATA5_D[7:0], además de DATA4_B[15:0], DATA_OUT_B[7:0] y DATA5_B[7:0]. Otras señales leídas son DR2_B[7:0]= "00000001" y NOP_B. Como no se detectó una dependencia de datos, con la línea CMUX2B_B[2:0]= "010" es DR2_B[7:0] la señal que pasa a los bits menos significativos de OPER2B[15:0]; el resto de los bits son '0's.
La dirección del registro destino RD_B[7:0]= "00000111" es leída por la *Forward Unit* cuando CLK2= '1', para verificar la posible existencia de una dependencia de datos de esta instrucción con la que recién entró al *pipeline* con el flanco de subida de CLK1.

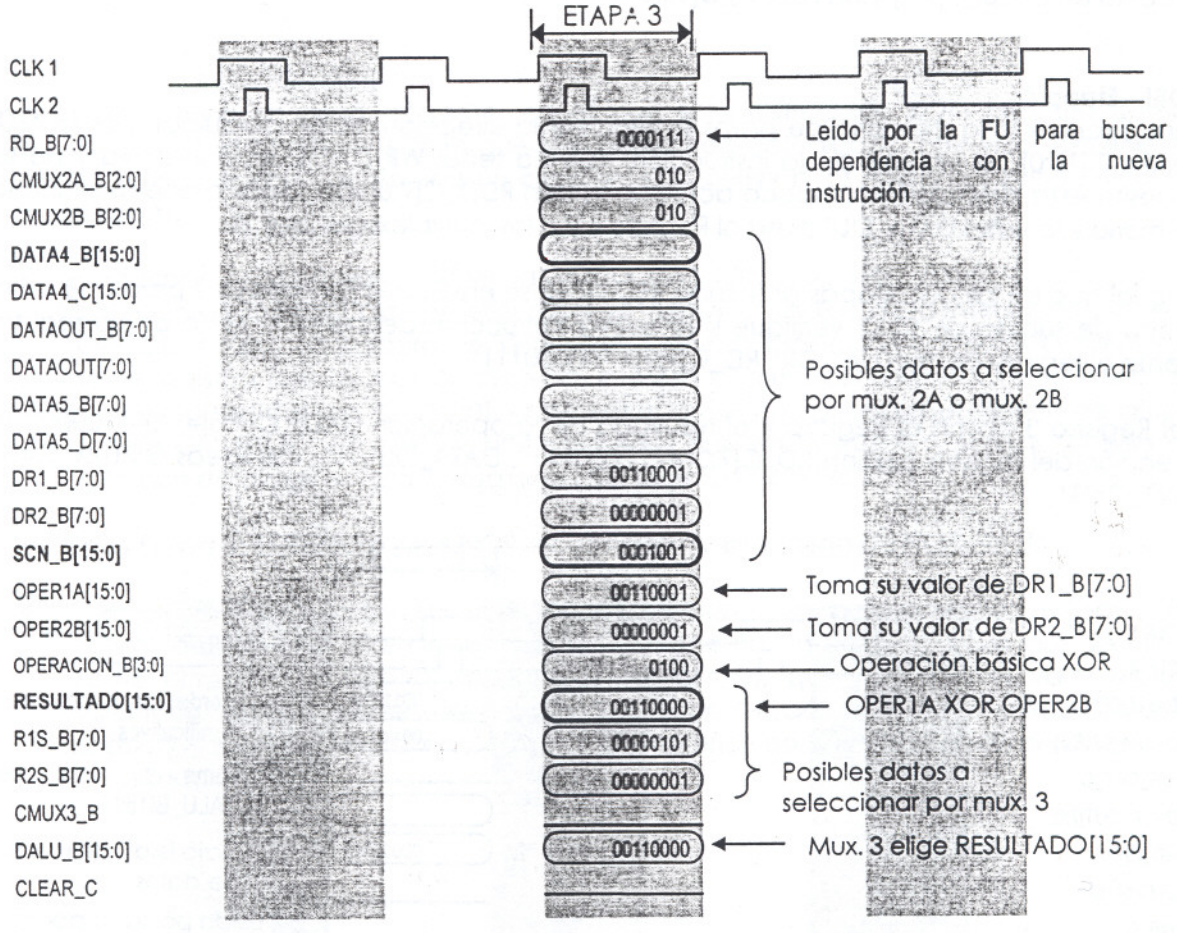


Figura 2.12 Diagrama de la Etapa 3 de XORI

OPER1A[15:0] y OPER2B[15:0] son leídos por la ALU, que debe estar habilitada. De acuerdo con OPERACION_A[3:0]= "0100" calcula *"00110001" XOR "00000001" = "00110000"* y lo

almacena en RESULTADO[15:0], agregándole ocho '0's empezando por el bit más significativo.

El mux. 3 lee RESULTADO[15:0], así como las señales R1S_B[7:0]= "00000101" y R2S_B[7:0]= "00000001". Con la línea de selección CMUX3_B= '0' elige el valor generado por la ALU, que pasa a la señal DALU_B[15:0].

Al mismo tiempo el comparador recibe las señales DR1_COMP[7:0] y RDD_B[7:0], seleccionadas por los mux. 5 y 6 respectivamente. Pero como la señal EC_B está inactiva no se realiza ninguna comparación y la señal COMP conserva el valor '0'. La señal CLEAR_C depende del OR de tres señales: CA_B (llamada a subrutina), PD_B (regreso de subrutina), COMP (resultado de la comparación). En este caso las 3 señales tienen el valor '0'. CLEAR_C= '0' es leída por los Registros 1 y 2. Por su parte LI_B, la línea de control que permitiría que la unidad de conteo lea una nueva dirección, está inactiva como resultado del OR entre COMP y LO_B. Esto implica que no habrá cambios sobre los Registros 1 y 2 ni sobre la unidad de conteo.

En el siguiente flanco de subida de CLK1 el Registro 3 lee algunas señales de salida del Registro 2: OPCOD_B[7:0], RD_B[7:0], CMUX4_B, WB_B y WR_DAT_B. También es leído el valor RDD_B[7:0], dato de salida del MUX5A. El caso de la señal DALU_B[15:0] es especial, ya que se copia en DALU_C[7:0] y en DATA4_C[15:0].

XORI - Etapa 4

Para acceder a la memoria de datos se requiere la dirección recién calculada por la ALU, DATA4_C[15:0]. Al tratarse de la instrucción, XORI la señal WR_DAT_C está inactiva y no se necesita esta dirección y tampoco actualizarla con RDD_C[7:0]. De todas maneras se lee la memoria y la señal DATA_OUT pasa al Registro 4 y a los multiplexores 2A y 2B.

Al igual que en las dos etapas anteriores RD_C[7:0] se envía a la *Forward Unit* para que en el flanco de subida de CLK2 verifique si la nueva instrucción del *pipeline* tiene como registro fuente al registro destino de XORI, RD_C[7:0]= "00000111".

Del Registro 3 pasan al Registro 4 el resultado de la operación DALU_C[7:0]= "00110000", la dirección del registro destino RD_C[7:0]= "00000111", DATA_OUT[7:0], las líneas CMUX4_C='1' y WB_C='1'.

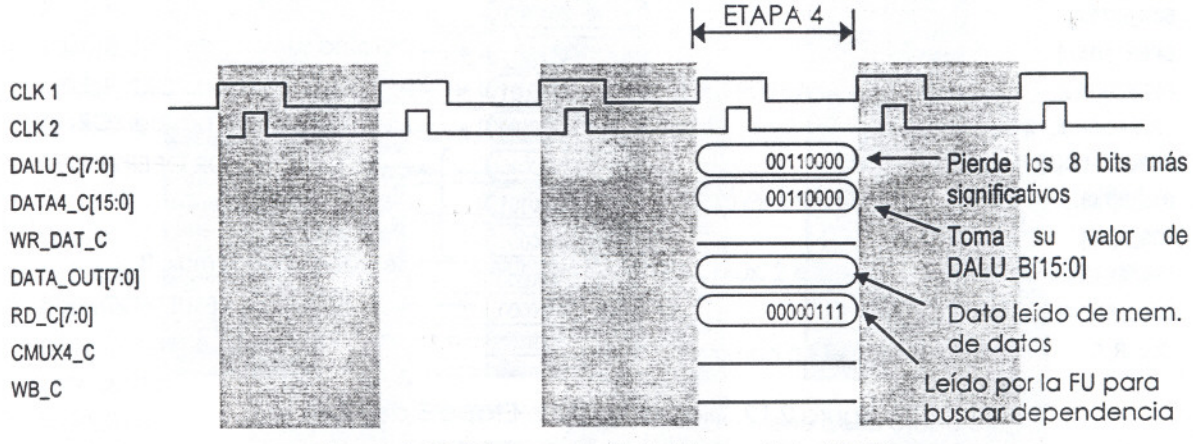


Figura 2.13 Diagrama de la Etapa 4 de XORI

XORI - Etapa 5

En esta etapa el multiplexor 4 lee DATA_OUT_D[7:0] y DALU_D[7:0], eligiendo a este último para propagarlo a la señal DATA5_D[7:0]. La señal WB_D= '1' y por lo tanto DATA5_D[7:0] = "00110000" actualiza el contenido del registro RD_D[7:0]= "00000111". Esta etapa se ejecuta de forma similar con todas las instrucciones que actualizan uno de los registros de propósito general (tipo OPERACIÓN y LOAD).

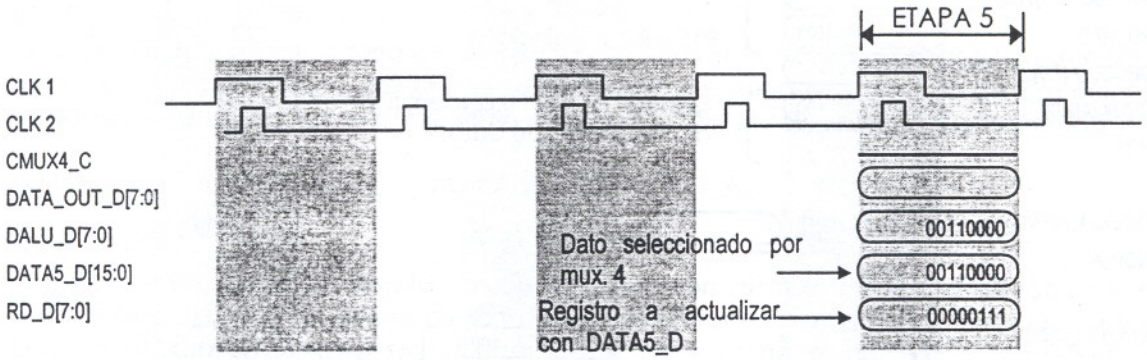


Figura 2.14 Diagrama de la Etapa 5 de XORI

Supongamos ahora un caso diferente, en el que esta instrucción XORI necesita como operando el contenido de un registro que aún no ha sido actualizado. Este valor pertenece a la instrucción que entró al *pipeline* **un ciclo** antes. Ocurrirían los siguientes cambios en la ejecución de XORI:

- *En la Etapa 1*: La *Forward Unit* lee la INSTRUCCION[31:0], la DIRECCION[15:0] actual, la dirección de la instrucción anterior (SCN_A[15:0]) y el código de la instrucción anterior (OPCOD_A[7:0]). La palabra de 32 bits que forma la instrucción contiene la dirección de los registros fuente, en este caso nos interesa el primero, "00000101", que coincide con el registro destino RD_A[7:0]. Al detectar la existencia de dependencia de datos genera CMUX2A[2:0]= "100" y NOP= '1' para que en la etapa 3 el mux. 2A elija al dato que contiene el valor actualizado del registro R1S_A[7:0].

- *En la Etapa 2*: El Registro 2 lee NOP_A= '1' y permite la salida de NOP_B= '1'.

- *En la Etapa 3* (Ejecución): Los mux. 2A y 2B reciben su línea de selección, además de NOP_B; ambas influyen en la selección de los operandos para la ALU. El mux. 2A elige como primer operando a la señal DATA4_C[15:0], que contiene el valor **recién calculado** por la ALU para la instrucción anterior. Así, aún cuando la instrucción que ocasionó la dependencia era la inmediata anterior, **no hay ningún retraso** al resolver esta situación.

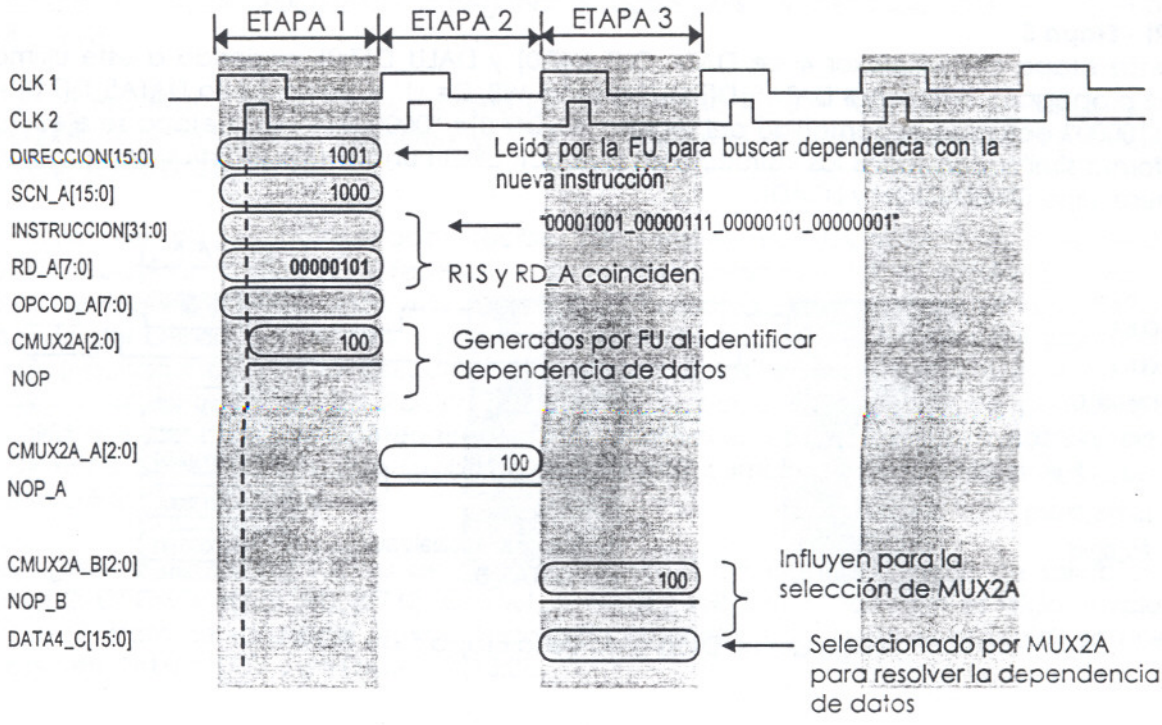


Figura 2.15 Diagrama de XORI con dependencia de datos

2.1.6 Instrucciones tipo LOAD

Con las operaciones LOAD del UAM - RISC II es posible leer un elemento de la memoria de datos y posteriormente almacenar este valor en un registro. Los primeros 8 bits contienen el código de operación; los 2 bytes finales pueden contener la dirección de uno o dos registros, o directamente una dirección de memoria.

Figura 2.16 Formato de las instrucciones LOAD

En la figura podemos observar 2 bloques de color más oscuro. El valor que contiene cada uno de estos bloques puede ser interpretado como la dirección de un registro o como un valor constante. Cuando este valor se considera como dirección, el registro correspondiente será leído en la etapa 2.

A partir de estos valores se obtiene un nuevo par de bytes, que formarán un std_logic_vector de 16 bits. Este vector contendrá la dirección de memoria a leer. Todo esto lo podemos ver más claramente al analizar las diferentes operaciones que caen dentro de esta categoría:

- **LOEX**

Permite la lectura de una localidad de memoria, para posteriormente almacenar la palabra leída en el registro RD_A. Para calcular la dirección de memoria es necesario leer 2 registros. El contenido de ambos se concatenará en la etapa de Ejecución.

LOEX (RD_A) <= MEMORIA[**(R1S_A)** & **(R2S_A)**]

- **LOAD**

Lee una localidad de la memoria de datos, para posteriormente almacenar su contenido en el registro RD_A. Es necesario calcular la dirección de memoria en la etapa de Ejecución: el valor leído del registro R1S_A se concatena con el valor constante representado por R2S_A.

LOAD (RD_A) <= MEMORIA[**(R1S_A)** & **R2S_A**]

- **LOFF**

En esta instrucción no es necesario calcular la dirección de memoria. Esta dirección viene directamente en los últimos 16 bits de la palabra de instrucción. Posteriormente, en la etapa de Ejecución, el multiplexor 3 seleccionará este valor como el necesario para leer una localidad de memoria.

LOFF (RD_A) <= MEMORIA[**R1S_A** & **R2S_A**]

A continuación se detalla un ejemplo de la ejecución de LOFF en cada una de las etapas del *pipeline*:

LOFF - Etapa 1

En el flanco de subida de CLK1 el Registro 1 del *pipeline* almacena las señales correspondientes a los mux. 2A, 2B, 5A y 6A; también almacena la instrucción, su dirección y NOP.

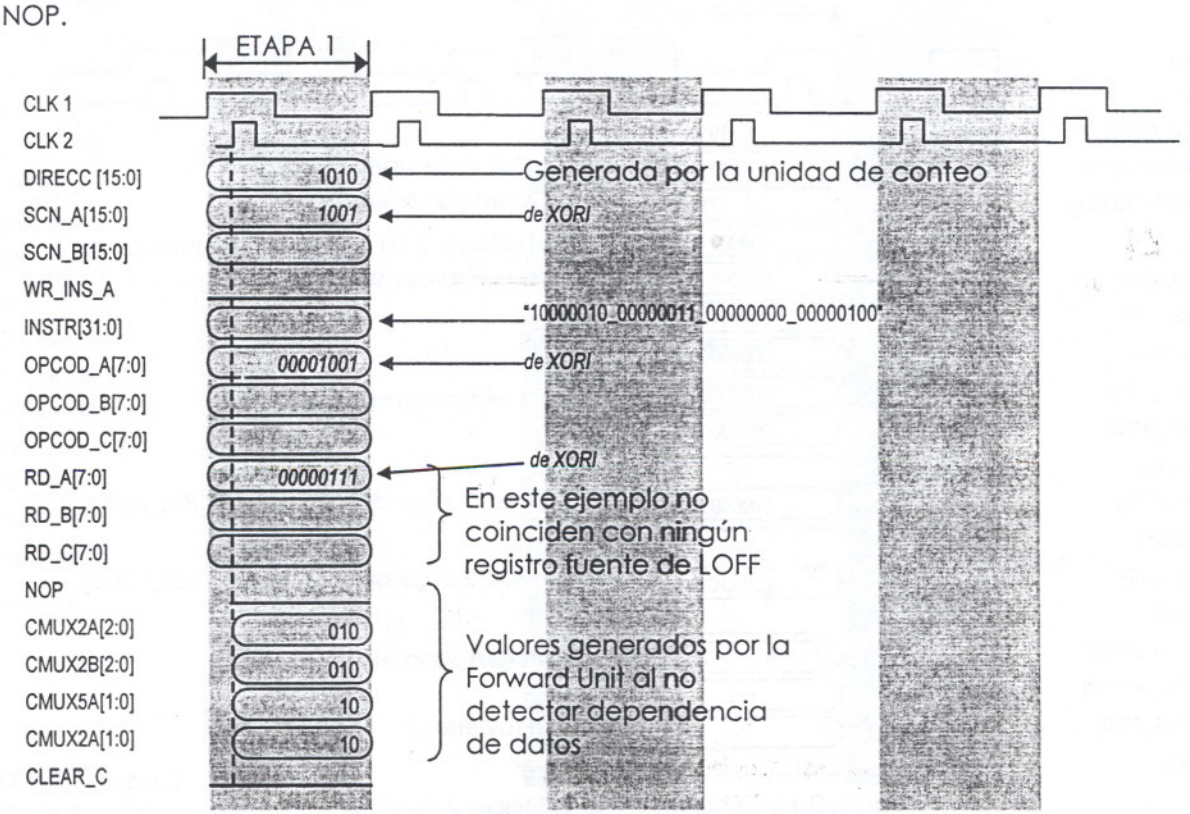


Figura 2.17 Diagrama de la Etapa 1 de LOFF

LOFF - Etapa 2

Si CLEAR_C= '0', las señales leídas por el Registro 1 en la etapa anterior propagarán su valor a: CMUX2A_A[2:0]= "010", CMUX2B_A[2:0]= "010", CMUX5A_A[1:0]= "10", CMUX6A_A[1:0]= "10", OPCOD_A[7:0]= "10000010", RD_A[7:0]= "00000011", R1S_A[7:0]= "00000000", R2S_A[7:0]= "00000100", SCN_A[15:0]= "00000000_00001010" y NOP_A= '0'.

Cuando la unidad de control decodifica LOFF, OPCOD_A[7:0]= "10000010", determina que la ALU no realizará ninguna operación y por ello genera OPERACION_A[3:0]= "1111"; el resto de las señales de control generadas son:

CA_A = '0', LOFF no es una instrucción de salto.
CMUX1A = '1', el 1er operando de LOFF es la constante "00000000".
CMUX1B = '1', el 2° operando de LOFF es la constante "00000100".
CMUX3_A = '1', pues el mux. 3 seleccionará DALU_B[15:0]= R1S_B[7:0] & R2S_B[7:0]
CMUX4_A = '0', pues el mux. 4 seleccionará el valor leído de la Mem. de instrucciones.
EC_A = '0', ya que LOFF no necesita el comparador.
ENABLE_A = '1', para que la unidad de conteo permanezca habilitada.
ER = '0', para deshabilitar la lectura de registros.
EU_A = '0', para deshabilitar la ALU.
INCREMENT_A = '1', para que la unidad de conteo permanezca generando instrucciones.
LI_A = '0', porque LOFF no va a generar una nueva dirección para la U. de Conteo
PD_A = '0', porque LOFF no implica regresar al contador anterior.
PU_A = '0', porque LOFF no implica cambiar al siguiente contador.
RESET = '0', anteriormente ya se estableció que el contador 0 debe iniciar en 0.
SET = '0', anteriormente ya se escribieron las instrucciones en la memoria
WB_A = '1', para habilitar posteriormente la escritura en el registro destino "00000011"
WR_DAT_A = '0', para leer de la memoria de datos.
WR_INS_A = '0', para que en el siguiente ciclo se vuelva a leer la Mem. de instrucciones.

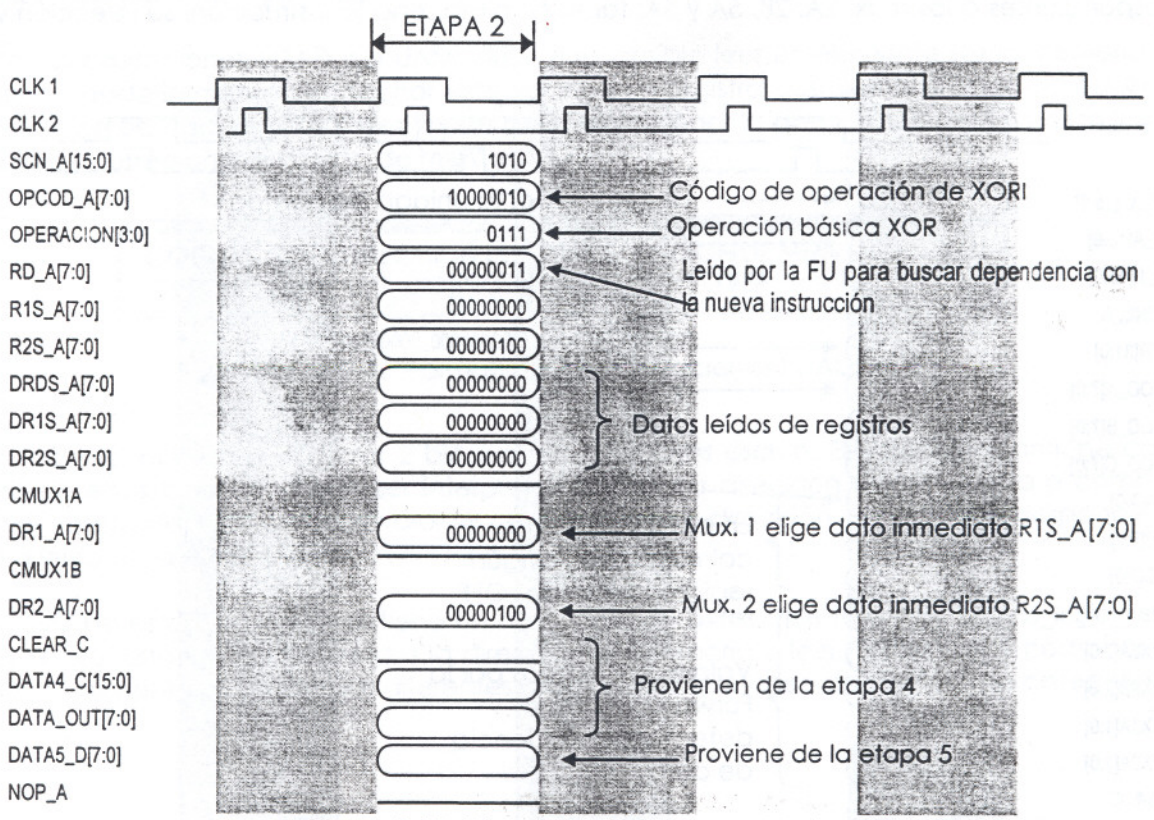


Figura 2.18 Diagrama de la Etapa 2 de LOFF

Al bloque de registros de propósito general llegan las direcciones de los 3 posibles registros fuente (que no serán leídos, pues ER= '0'): R1S_A[7:0]= "00000000", R2S_A[7:0]="00000100" y RD_A[7:0]= "00000011"; los 3 buses de salida son "00000000".

La dirección del registro destino RD_A[7:0]= "00000011" es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos con la instrucción que ingresó al *pipeline* en este flanco de subida de CLK2.
El multiplexor 1A tiene la línea de selección en '1' y por lo tanto elige al valor constante R1S_A[7:0]= "00000000". El multiplexor 1B también debe elegir al dato inmediato R2S_A[7:0]= "00000100" con la línea CMUX1B= '1'. Estos datos pasan a las señales DR1_A[7:0] y DR2_A[7:0], respectivamente.

En el flanco de subida de CLK1 el Registro 2 almacena las señales provenientes del Registro 1, las señales de control, los valores derivados de los multiplexores y el valor leído de RD_A[7:0], todas con terminación "_A". Además lee CLEAR_C, DATA4_C[15:0], DATA_OUT[7:0] y DATA5_B[7:0]; estos buses pueden necesitarse en caso de una dependencia de datos con la instrucción que se encuentra en la Etapa de Acceso a memoria. CLEAR_C= '0' no tiene ningún efecto sobre este registro. Las señales de salida tienen la terminación "_B".

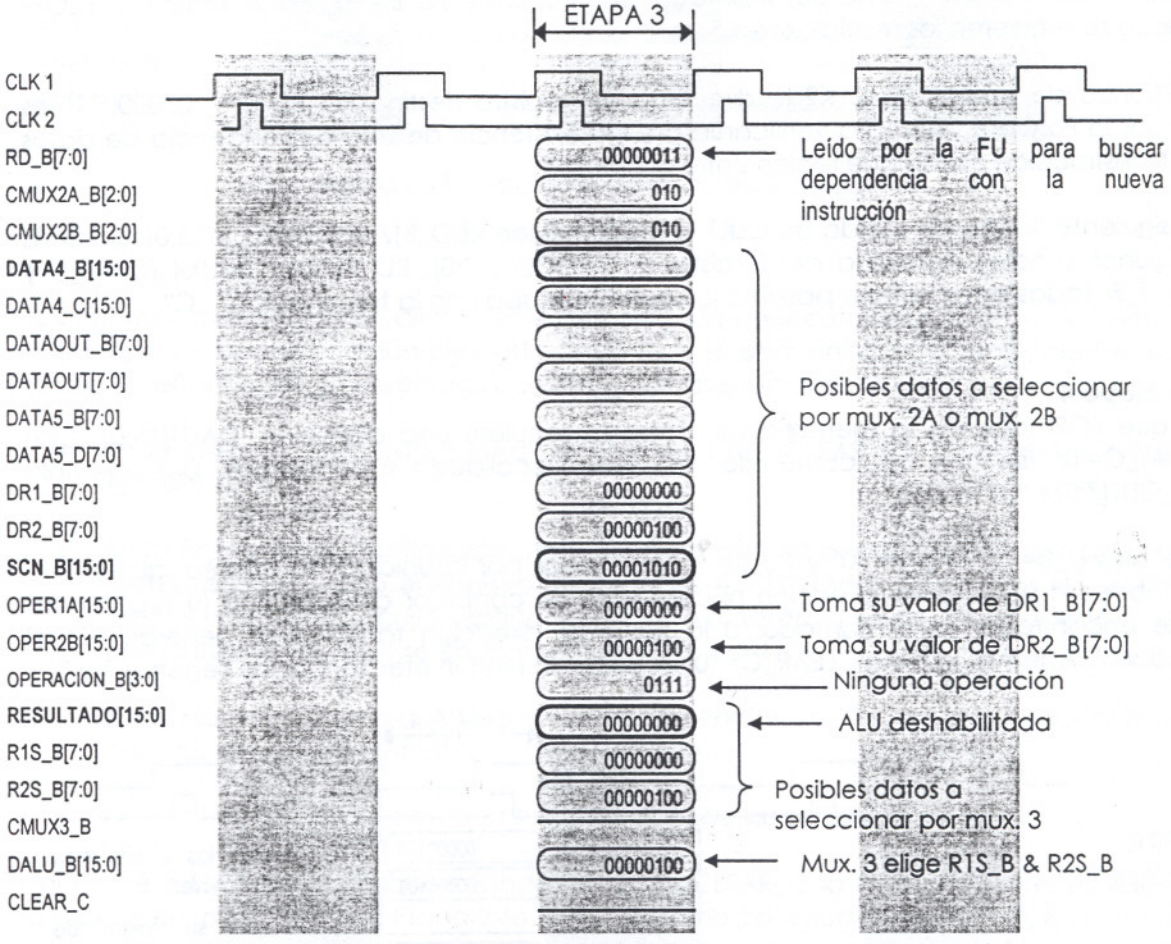


Figura 2.19 Diagrama de la Etapa 3 de LOFF

LOFF - Etapa 3

En la Etapa de Ejecución se deben seleccionar los operandos para la ALU y el comparador. Esto se logra empleando los multiplexores 2A, 2B, 5 y 6. El mux. 2A recibe señales de varias

etapas: DATA4_C[15:0], DATA_OUT[7:0] (etapa 4) y DATA5_D[7:0] (etapa 5), DATA4_B[15:0], DATA_OUT_B[7:0] (de la etapa 2, pero pertenecen a la instrucción que en este momento ya está en etapa 5), DATA5_B[7:0] (de la etapa 2, pero proviene de la instrucción que en este flanco de subida salió del *pipeline*), DR1_B[7:0]= "00000000" y SCN_B= "00000000_00001010" (de la instrucción actual). Su línea de selección es CMUX2A_B[2:0]= "010" y actúa junto con NOP_B='0'. Por lo tanto el dato seleccionado es DR1_B[7:0], que propaga su valor a los ocho bits menos significativos de OPER1A[15:0].

El funcionamiento del mux. 2B es similar: también lee DATA4_C[15:0], DATA_OUT[7:0] y DATA5_D[7:0], DATA4_B[15:0], DATA_OUT_B[7:0] y DATA5_B[7:0], además de DR2_B[7:0]= "00000100". Como no se detectó una dependencia de datos en la primera etapa, NOP_B='0'; con este valor y con CMUX2B_B[2:0]= "010", OPER1B[15:0] toma el valor de DR2_B[7:0].

OPER1A[15:0] y OPER1B[15:0] son leídos por la ALU, que está deshabilitada. Entonces el valor de RESULTADO[15:0] es "00000000_00000000". El mux. 3 lee RESULTADO[15:0], así como las señales R1S_A[7:0]= "00000000" y R2S_A[7:0]= "00000100". Con CMUX3_A = '1' elige R1S_A & R2S_A, de 16 bits, y lo asigna a DALU_B[15:0].

El comparador, también se ha deshabilitado, por medio de EC_B= '0'. Por lo tanto para LOFF tampoco se requieren los multiplexores 5 y 6.

En el flanco de subida de CLK2 la dirección del registro destino RD_B[7:0]= "00000011" es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos de esta instrucción con la que recién entró al *pipeline*.

En el siguiente flanco de subida de CLK1 el Registro 3 lee RDD_B[7:0] y DALU_B[15:0], además de algunas señales de salida del Registro 2: OPCOD_B[7:0], RD_B[7:0], CMUX4_B, WB_B y WR_DAT_B. Todas estas señales pasan a la siguiente etapa con la terminación "_C".

LOFF - Etapa 4

Para que LOFF accese la memoria de datos se requiere una dirección, DALU[15:0]. Con WR_DAT_C='0' se lee el contenido de esta localidad, supongamos por ejemplo DATA_OUT[7:0]= "01100010".

Las señales LI_B= '0', PD_B= '0' y PU_B= '0' son leídas por la unidad de conteo, que por lo tanto no leerá una nueva dirección ni cambiará de contador de subrutina (si suponemos que se encontraba en el contador 0 la siguiente dirección también se generará en el contador 0). Además la línea CLEAR_C= '0' es leída, sin ningún efecto, por los Registros 1 y 2.

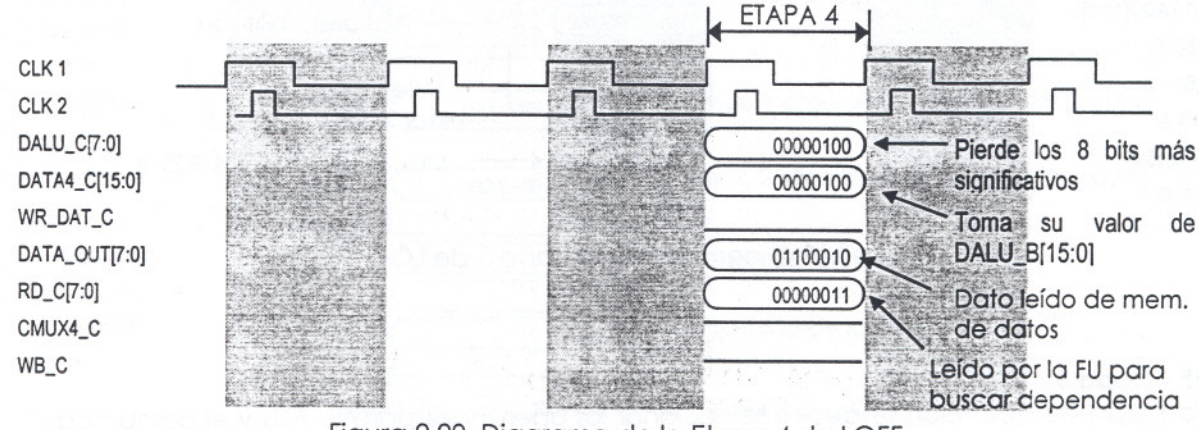


Figura 2.20 Diagrama de la Etapa 4 de LOFF

Al igual que en las dos etapas anteriores RD_C[7:0] se envía a la *Forward Unit* para verificar en CLK2= '1' si la nueva instrucción del *pipeline* tiene como registro fuente al registro destino de LOFF, RD_C[7:0]= "00000011".

Del Registro 3 pasan al Registro 4 el resultado de la operación DALU_C[7:0]= "00000100", la dirección del registro destino RD_C[7:0]= "00000011", DATA_OUT_C[7:0], las líneas CMUX4_C='1' y WB_C='1'.

LOFF - Etapa 5

En la última etapa de ejecución de LOFF, el multiplexor 4 elige DATA_OUT_D[7:0] para asignarlo a la señal DATA5_D[7:0] = "01100010", que actualiza el registro RD_D[7:0]= "00000011".

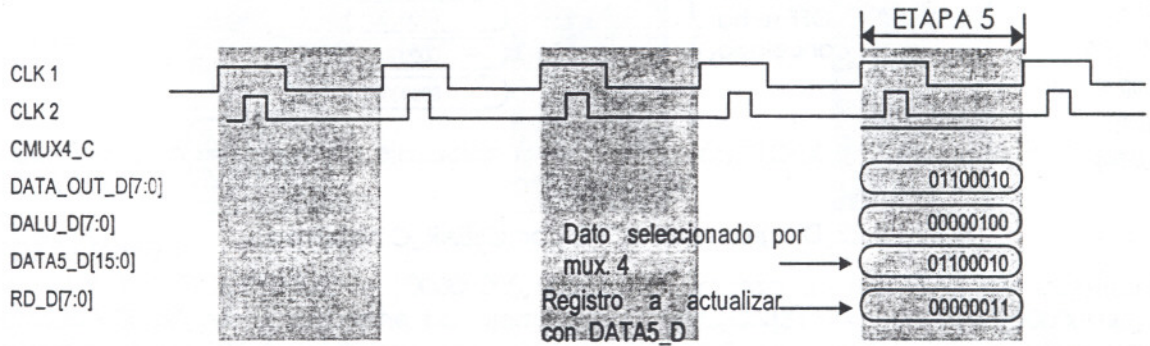


Figura 2.21 Diagrama de la Etapa 5 de LOFF

Supongamos ahora un caso diferente: LOFF comienza a ejecutarse, y cuando se encuentra en la Etapa de Decodificación otra instrucción (que se encuentra en la Etapa de Ejecución) habilita la señal CLEAR_C. Enseguida describiremos las etapas de LOFF.

- *En la Etapa 1* (Lectura): cuando el Registro 1 lee CLEAR_C= '1' asigna a todas sus señales '0's.
- *En la Etapa 2* (Decodificación): la unidad de control decodifica la instrucción de la forma usual. Cuando el Registro 2 lee CLEAR_C= '1' todas sus señales de salida (de control y de datos), cuyo destino es la Etapa 3, serán '0's. Lo mismo sucederá con la nueva instrucción que se encontraba en la Etapa 1.
- *En la Etapa 3* (Ejecución): se leen todas las señales en '0's, y por lo tanto la ALU y el comparador SE DESHABILITAN.
- *En las Etapas 4 y 5*: se continúa con la lectura de las señales en '0's.

A partir de esto concluimos que cuando se activa CLEAR_C la instrucción que en ese ciclo de reloj se encontraba en la Etapa 2 (o Etapa 1) ya no continúa su ejecución. En el siguiente ciclo entrará una nueva instrucción al *pipeline*, que ya se ejecutará normalmente. Por lo tanto, cuando se activa CLEAR_C se pierden 2 ciclos de reloj.

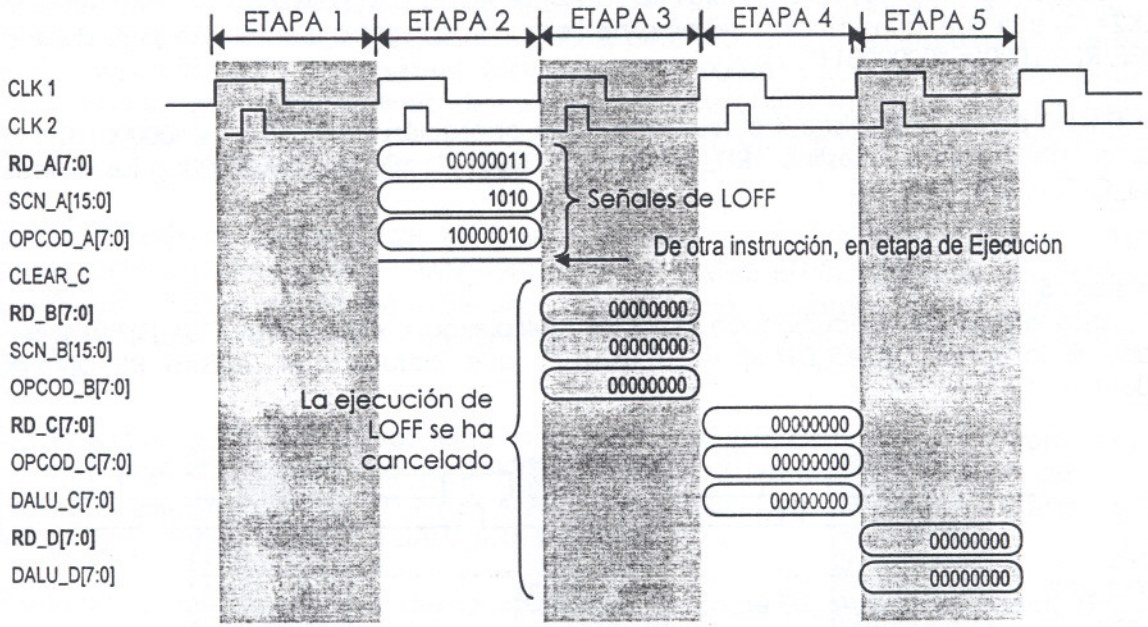


Figura 2.22 Diagrama de LOFF, con CLEAR_C habilitada

2.1.7 Instrucciones tipo STORE

Nos permiten actualizar el contenido de un elemento de la memoria de datos. En la siguiente figura podemos observar el formato que tienen estas instrucciones: el código de operación, seguido por la dirección del registro RD_A y después las 2 señales R1S_A y R2S_A. En este caso RD_A no es la dirección del registro destino, sino la dirección del registro a leer; posteriormente este valor se almacenará en la memoria.

Figura 2.23 Formato de las instrucciones STORE

Las operaciones que caen en esta categoría son:

- **STOEX**

La forma de obtener la dirección de memoria es similar a las instrucciones de tipo LOAD, como lo podemos ver a continuación. Esta instrucción requiere la lectura de 3 registros: RD_A, R1S_A y R2S_A. El primero contiene la dirección del registro cuyo valor será almacenado en memoria. Al concatenar en la etapa de Ejecución el contenido de los otros 2 registros resultará la localidad de memoria requerida.

STOEX MEMORIA[**(R1S_A)** & **(R2S_A)**] <= (RD_A)

- **STORE**

En primer lugar se lee el registro RD_A. También se lee R1S_A, y este valor se concatena con la constante R2S_A para obtener una localidad de la memoria de datos (un std_logic_vector de 16 bits). De esta manera se podrá accesar dicha localidad para actualizarla con el valor de RD_A.

STORE MEMORIA[**(R1S_A)** & **R2S_A**] <= (RD_A)

- **STOFF**

Al igual que en la instrucción LOFF no es necesario calcular la dirección de memoria, pues se encuentra como valor constante en la palabra de instrucción. De forma similar, el multiplexor 3 seleccionará este valor como la localidad de la memoria de datos a actualizar.

STOFF MEMORIA[**R1S_A** & **R2S_A**] <= RD_A

A continuación se detalla la ejecución de la instrucción STOEX en cada una de las etapas del *pipeline*:

STOEX - Etapa 1

La señal DIRECCION[15:0] = "00000000_00001011" accesa la memoria de instrucciones cuando WR_INS_A= '0' y obtiene, por ejemplo, INSTRUCCIÓN[31:0]= "10000100_00000100_ 00000001_00000010". En el flanco de subida de CLK1 el Registro 1 del *pipeline* almacena las señales correspondientes a los mux. 2A, 2B, 5A y 6A; también almacena la instrucción actual y su dirección. Si CLEAR_C= '0', el valor contenido en estas señales será leído por el Registro 1: INSTRUCCIÓN[31:0], DIRECCION[15:0], CMUX2A[2:0]= "010", CMUX2B[2:0]= "010", CMUX5A[1:0]= "10", CMUX6A[1:0]= "10" y NOP_A= '0'.

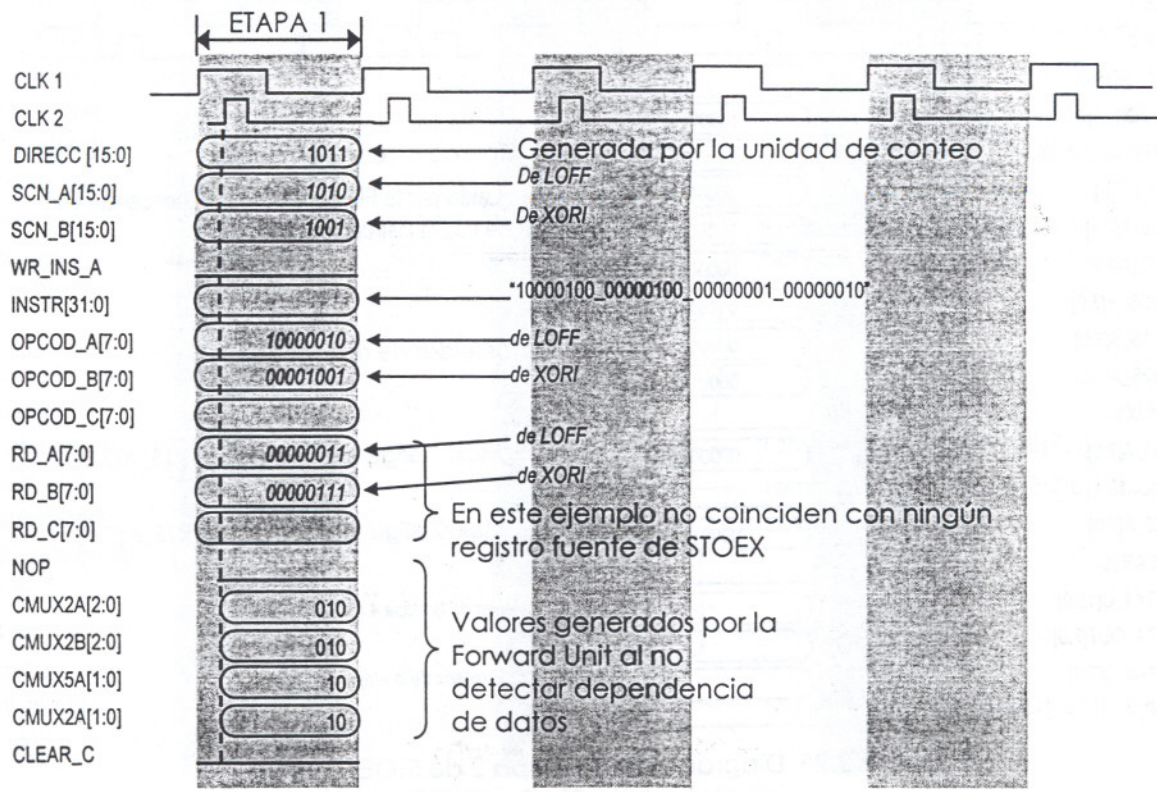


Figura 2.24 Diagrama de la etapa 1 de STOEX

STOEX - Etapa 2

Al leer INSTRUCCIÓN[31:0] y DIRECCION[15:0] se obtienen las siguientes señales de salida de Registro 1: OPCOD_A[7:0]= "10000100", RD_A[7:0]= "00000100", R1S_A[7:0]= "00000001", R2S_A[7:0]= "00000010", SCN_A[15:0]= "00000000_00001011".

Cuando la unidad de control decodifica OPCOD_A[7:0]= "10000100" determina que la ALU realizará una concatenación, generando la señal OPERACION_A[3:0]= "0101". A continuación se describen las señales de control generadas:

CA_A = '0', STOEX no es una instrucción de salto.
CMUX1A = '0', el 1er operando de STOEX es el contenido del registro "00000001".
CMUX1B = '0', el 2° operando de STOEX es el contenido del registro "00000010".
CMUX3_A = '0', el mux. 3 seleccionará el RESULTADO[15:0] de la ALU.
CMUX4_A = '0', el mux. 4 seleccionará el valor leído de la Mem. de instrucciones.
EC_A = '0', ya que STOEX no necesita el comparador.
ENABLE_A = '1', para que la unidad de conteo permanezca habilitada.
ER = '1', para habilitar la lectura de registros.
EU_A = '1', para habilitar la ALU.
INCREMENT_A = '1', para que la unidad de conteo continúe generando instrucciones.
LI_A = '0', no va a generar una nueva dirección para la unidad de conteo
PD_A = '0', porque STOEX no implica regresar al contador anterior.
PU_A = '0', porque STOEX no implica cambiar al siguiente contador.
RESET = '0', anteriormente ya se estableció que el contador 0 debe iniciar en 0.
SET = '0', anteriormente ya se escribieron las instrucciones en la memoria
WB_A = '0', para deshabilitar la actualización de registros
WR_DAT_A = '1', para escribir en la memoria de datos.
WR_INS_A = '0', para que en el siguiente ciclo se vuelva a leer la Mem. de instrucciones.

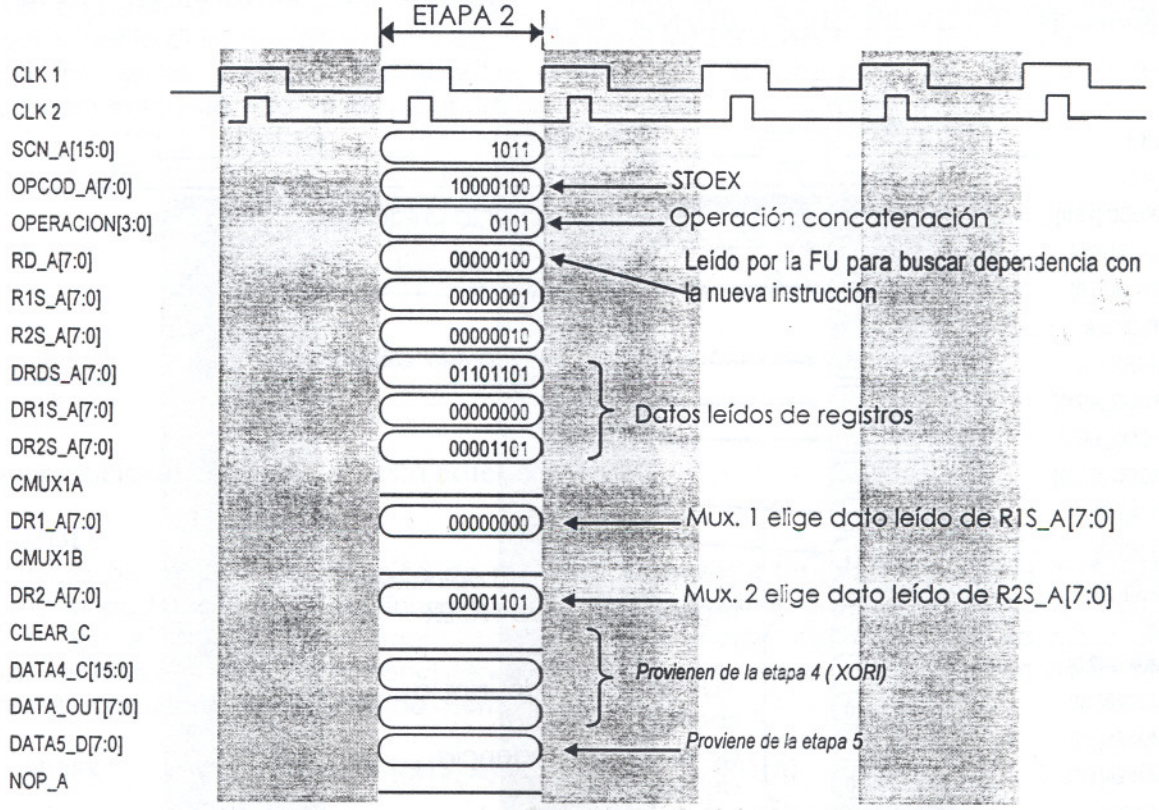


Figura 2.25 Diagrama de la etapa 2 de STOEX

Al bloque de registros de propósito general llegan las direcciones de los 3 posibles registros fuente: R1S_A[7:0]= "00000001", R2S_A[7:0]="00000010" y RD_A[7:0]= "00000100". La dirección RD_A[7:0] también es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos en CLK2='1'.

El multiplexor 1A tiene la línea de selección en '0' y por lo tanto elige al valor leído del primer registro fuente DR1S_A[7:0]= "00000000". El multiplexor 1B también elige al dato leído DR2S_A[7:0]= "00001101" con la línea CMUX1B= '0'. Estos datos pasan a las señales DR1_A[7:0] y DR2_A[7:0], respectivamente. En este caso también se requiere el dato contenido en RD_A[7:0], que es la señal DRDS_A[7:0]= "01101101".

En el flanco de subida de CLK1 el Registro 2 almacena las señales provenientes del Registro 1, las señales de control, los valores derivados de los multiplexores y el valor leído de RD_A[7:0], todas con terminación "_A". Además lee CLEAR_C, DATA4_C[15:0], DATA_OUT[7:0] y DATA5_B[7:0]; estos buses podrían necesitarse en una dependencia de datos con la instrucción que se encuentra en la etapa 4 del *pipeline*, pero este no es el caso. Por su parte, CLEAR_C= '0' no tiene ningún efecto sobre este registro. Las señales de salida del Registro 2 tienen la terminación "_B".

STOEX - Etapa 3

Aquí se eligen los operandos para la ALU y el comparador mediante los multiplexores 2A, 2B, 5 y 6, que funcionan de forma similar a la instrucción LOFF, que ya explicamos. En el flanco de subida de CLK2 la dirección del registro destino RD_B[7:0]= "00000100" es leída por la

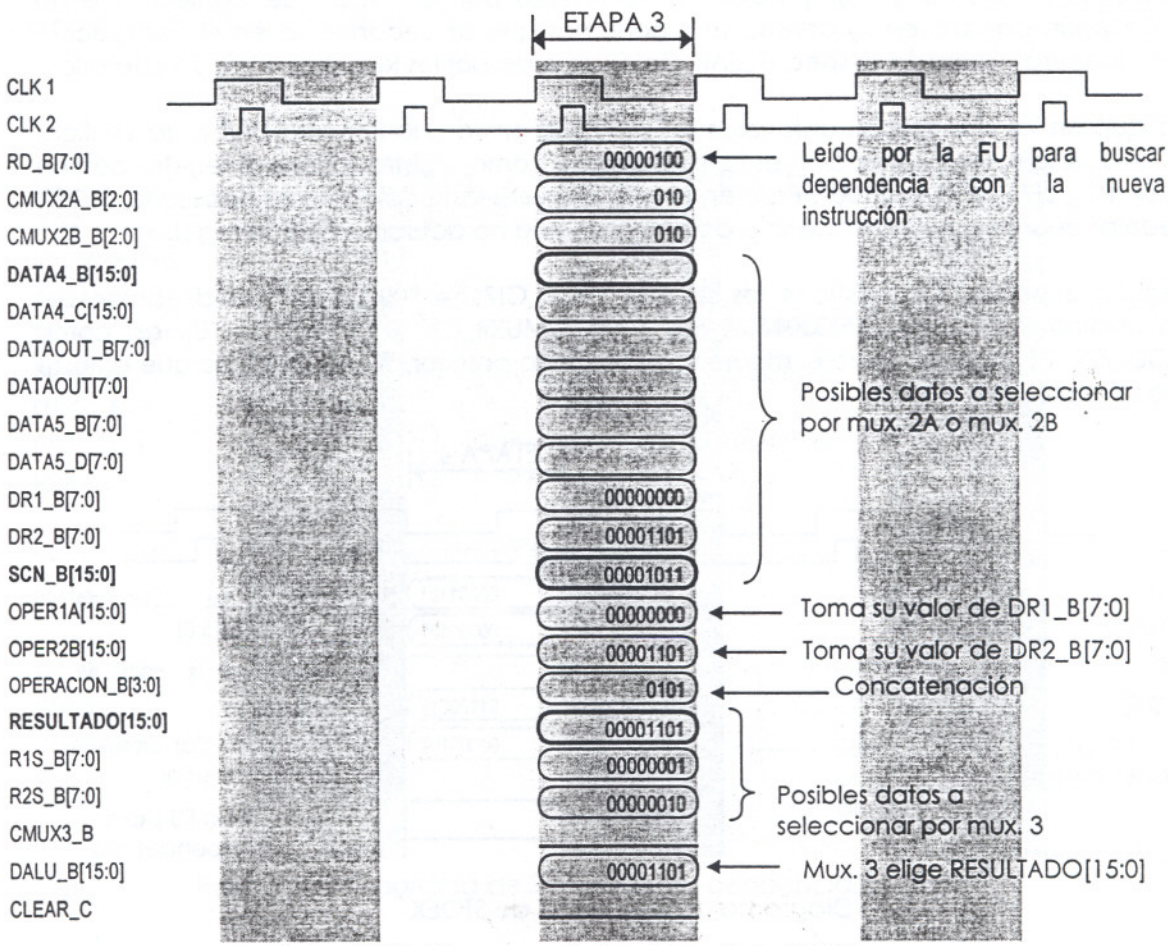


Figura 2.26 Diagrama de la etapa 3 de STOEX

Forward Unit para verificar la posible existencia de una dependencia de datos de esta instrucción con la que recién entró al *pipeline*.

Las señales DR1S_A[7:0] y DR2S_A[7:0] se convierten en esta etapa en OPER1A[15:0] y OPER1B[15:0] son leídos por la ALU, habilitada por EU_B= '1'. Entonces los concatena, ya que a la instrucción STOEX le corresponde OPERACION_A[3:0]= "0101". La señal RESULTADO[15:0] tiene el valor "00000000_00001101". El mux. 3 lo lee, así como a las señales R1S_A[7:0]= "00000001" y R2S_A[7:0]= "00000010". Con CMUX3_A = '0' se elige RESULTADO[15:0], que se propaga a DALU_B[15:0]. Aunque no se requieran datos para el comparador, el multiplexor 5 debe seleccionar un dato, y ya que no se ha detectado una dependencia de datos, elige a DRDS_A[7:0] y lo asigna a RDD_B[7:0].

El comparador se encuentra deshabilitado en esta instrucción, con EC_B= '0'. En el siguiente flanco de subida de CLK1 el Registro 3 lee DALU_B[15:0] y RDD_B[7:0], además de algunas señales de salida del Registro 2: OPCOD_B[7:0], RD_B[7:0], CMUX4_B, WB_B y WR_DAT_B. Todas estas señales pasan a la siguiente etapa con la terminación "_C".

STOEX - Etapa 4

Para actualizar la memoria de datos se accesa la dirección calculada por la ALU, DATA4_C[15:0]= "00000000_00001101". Este valor se encontraba en la señal DALU_B[15:0], y al salir del Registro 3 una copia de esta señal se asigna a DALU_C[7:0] y otra a DATA4_C[15:0]. Con WR_DAT_C='1' se actualiza el contenido de esta localidad con el valor RDD_C[7:0]= "01101101".

Las señales LI_B= '0', PD_B= '0' y PU_B= '0' son leídas por la unidad de conteo, que no realizará ningún cambio en su conteo (si suponemos que se encontraba en el contador 0, ahí permanecerá). También la línea CLEAR_C= '0' es leída por los Registros 1 y 2, sin alterarlos.

Al igual que en las dos etapas anteriores RD_C[7:0] se envía a la *Forward Unit* para verificar en CLK2= '1' si la nueva instrucción del *pipeline* tiene como registro fuente al registro destino de STOEX, RD_C[7:0]= "00000100". Pero en el caso de esta instrucción no es necesario. STOEX no puede ocasionar una dependencia de datos, ya que no actualiza ningún registro.

Del Registro 3 pasan al Registro 4 las señales DALU_C[7:0]= "00001101", la dirección del registro destino RD_C[7:0]= "00000100", las líneas CMUX4_C= '0' y WB_C= '0', así como DATA_OUT_C[7:0] (que conserva el mismo valor del ciclo anterior, "01100010", porque en este ciclo no se realizó una lectura).

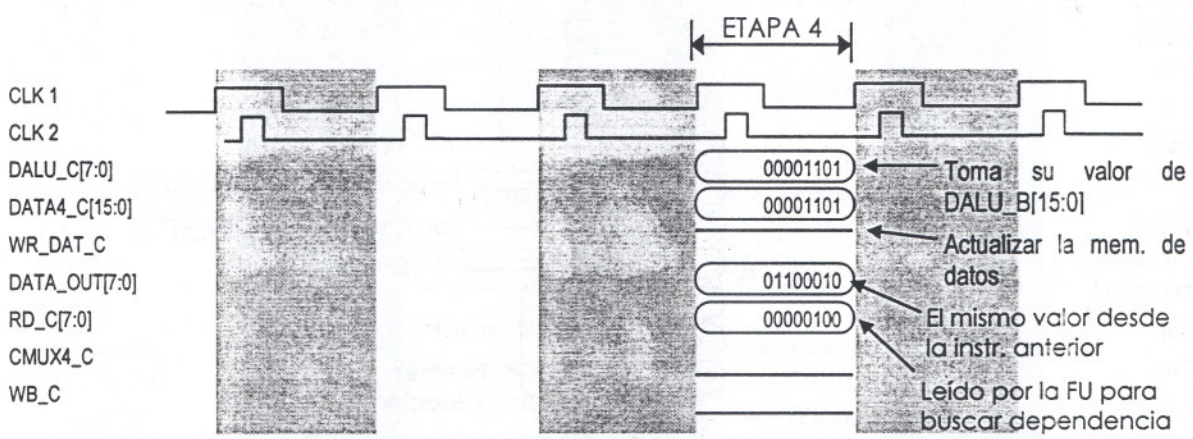


Figura 2.27 Diagrama de la etapa 4 de STOEX

STOEX - Etapa 5

El multiplexor 4 elige DATA_OUT_D[7:0] para propagarlo a DATA5_D[7:0]= "01100010". Sin embargo, ya que WB_D= '0', este valor no se empleará. En una instrucción de tipo OPERACIÓN este valor actualizaría al registro destino RD_D[7:0]= "00000100". En el caso de las instrucciones de tipo STORE, podemos considerar que su ejecución ha terminado en la etapa 4 y por lo tanto no requieren la etapa de Escritura de Resultados.

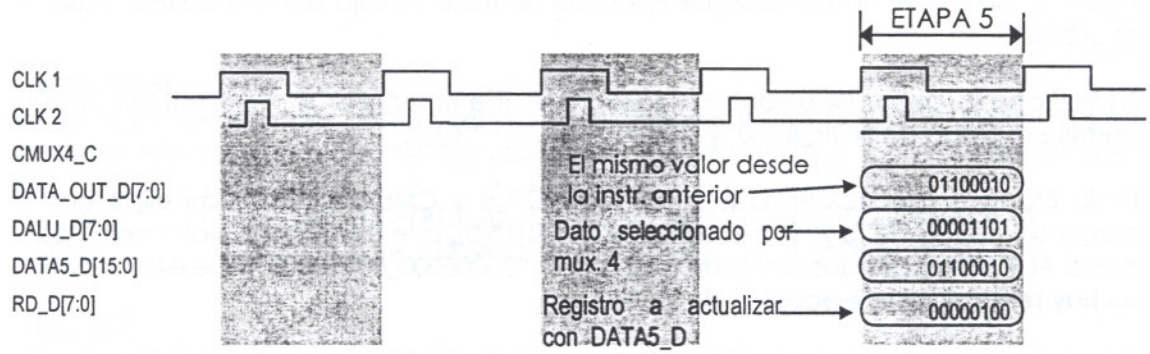


Figura 2.28 Diagrama de la etapa 5 de STOEX

Veamos ahora un caso diferente: supongamos que STOEX necesita como operando el contenido de un registro que aún no ha sido actualizado, y que este valor pertenece a la instrucción que entró al *pipeline* **dos ciclos** antes (XORI). Entonces ocurrirían los siguientes cambios en la ejecución de STOEX:

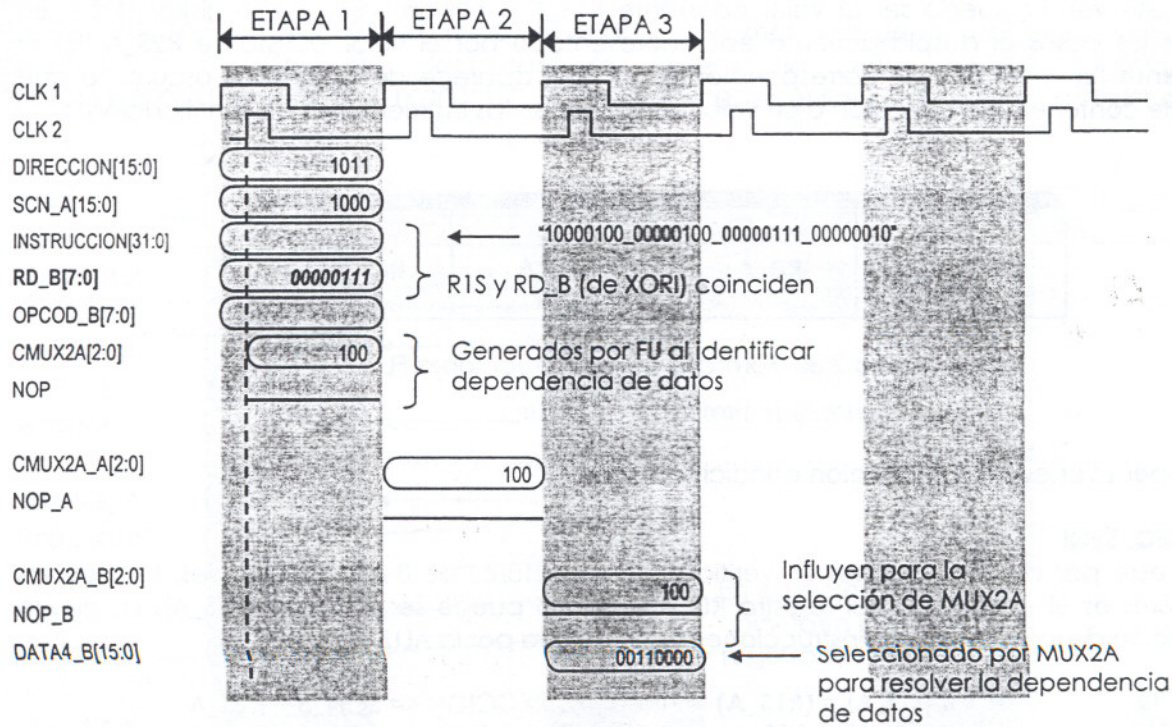


Figura 2.29 Diagrama de STOEX con dependencia de datos

- *En la Etapa 1* (Lectura): La *Forward Unit* lee la INSTRUCCION[31:0], la DIRECCION[15:0] actual, la dirección de las 2 instrucciones anteriores (SCN_A[15:0] y SCN_B[15:0]) y su código de operación (OPCOD_A[7:0] y OPCOD_B[7:0]). La palabra INSTRUCCION[31:0] contiene la dirección de los registros fuente. Supongamos que la dirección del primer registro de STOEX tiene un nuevo valor, que coincide con el registro destino de XORI, es decir, R1S= "00000111" = RD_B[7:0]. Al detectar la existencia de dependencia de datos, la *Forward Unit* genera CMUX2A[2:0]= "100" y NOP= '0' para que en la Etapa de Ejecución el mux. 2A elija al dato correcto como operando.

- *En la Etapa 2 (Decodificación):* El Registro 2 recibe la señal NOP_A y CMUX2A_A[2:0]; permite la salida de NOP_B= '0' y CMUX2A_B[2:0]= "100".

- *En la Etapa 3* (Ejecución): El mux. 2B lee NOP_B y CMUX2A_B[2:0]; así elige como primer operando de la ALU a la señal DATA4_B[15:0], que contiene el valor calculado por la ALU para XORI dos ciclos de reloj antes ("00000000_00110000"). De esta manera **no hay retrasos** en la ejecución de STOEX.

2.1.8 Instrucciones tipo RAMIFICACIÓN

El UAM - RISC II contiene instrucciones de salto condicional e incondicional. En primer lugar veremos las que requieren una comparación. Cuando el comparador produce un resultado verdadero es posible modificar la secuencia de ejecución. La nueva dirección es calculada a partir de un desplazamiento sobre la dirección actual.

La comparación se realiza entre el contenido del registro RD_A y otro **std_logic_vector** de 8 bits. Este vector puede ser el valor constante R1S_A o el contenido del registro R1S_A. En todos los casos el desplazamiento está representado por el valor constante R2S_A. En la siguiente figura el bloque correspondiente a R1S_A aparece de color más oscuro ya que puede contener una dirección o un valor constante en las instrucciones de ramificación.

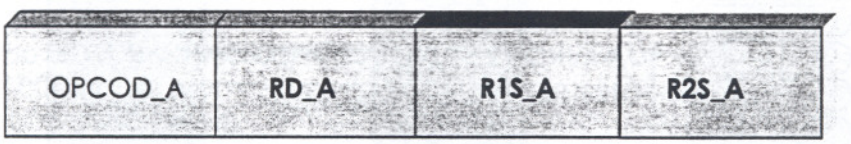


Figura 2.30 Formato de las instrucciones FLOW

Las operaciones de ramificación condicional son:

- **BEQ, BEQI**

Con este par de instrucciones se verifica si dos vectores de 8 bits son iguales. Uno de los vectores es el contenido del registro RD_A y el otro puede ser R1S_A o (R1S_A). La nueva dirección de la memoria de instrucciones es calculada por la ALU.

BEQ	if (RD_A) = **(R1S_A)**	then	DIRECCION <= SCN_B + R2S_A
BEQI	if (RD_A) = **R1S_A**	then	DIRECCION <= SCN_B + R2S_A

- **BNEQ, BNEQI**

Aquí se comprueba si ambos operandos son diferentes. Si así sucede ya no se ejecutará la siguiente instrucción, sino la nueva instrucción de la localidad DIRECCION + R2S_A. Cuando los operandos son iguales la secuencia de ejecución no es alterada.

BNEQ	if (RD_A) <> **(R1S_A)**	then	DIRECCION <= SCN_B + R2S_A
BNEQI	if (RD_A) <> **R1S_A**	then	DIRECCION <= SCN_B + R2S_A

- **BGA, BGAI**

Para que se ejecute un cambio en la memoria de instrucciones es necesario que el contenido del registro RD_A sea mayor que el contenido de R1S_A o que la constante R1S_A.

BGA	if (RD_A) > **(R1S_A)**	then	DIRECCION <= SCN_B + R2S_A
BGAI	if (RD_A) > **R1S_A**	then	DIRECCION <= SCN_B + R2S_A

- **BLE, BLEI**

Cuando se cumple la condición de que el operando (RD_A) sea menor que (R1S_A) o que R1S_A entonces la dirección actual tiene un desplazamiento. Cuando no se cumple tampoco hay cambios en la dirección de la memoria de instrucciones.

BLE	if (RD_A) < **(R1S_A)**	then	DIRECCION <= SCN_B + R2S_A
BLEI	if (RD_A) < **R1S_A**	then	DIRECCION <= SCN_B + R2S_A

A continuación se describe un ejemplo de la ejecución de BLEI a través del *pipeline*:

BLEI - Etapa 1

DIRECCION[15:0] = "00000000_00001100" es leída por la memoria de instrucciones cuando WR_INS_A = '0'. De esta localidad se obtiene la señal INSTRUCCIÓN[31:0]= "00010111_00000101_00010111_00000110". Si CLEAR_C= '0' en el flanco de subida de CLK1 el

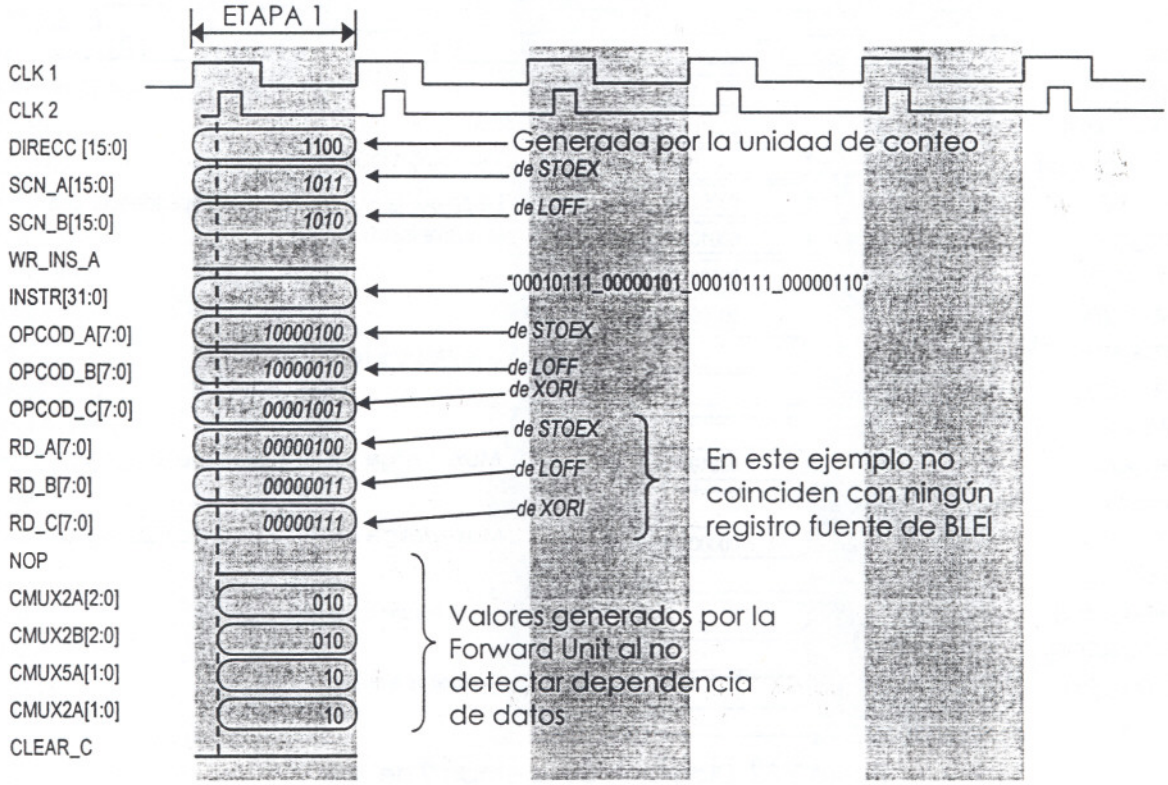


Figura 2.31 Diagrama de la etapa 1 de BLEI

Registro 1 del *pipeline* almacena las señales INSTRUCCIÓN[31:0], DIRECCION[15:0], CMUX2A[2:0]= "010", CMUX2B[2:0]= "010", CMUX5A[1:0]= "10", CMUX6A[1:0]= "10" y NOP='0'.

BLEI - Etapa 2

El Registro 1 tiene como salidas, además de las líneas de los multiplexores, a OPCOD_A[7:0]= "00010111", RD_A[7:0]= "00000101", R1S_A[7:0]= "00010111", R2S_A[7:0]= "00000110" y SCN_A[15:0]= "00000000_00001100". Cuando la unidad de control decodifica OPCOD_A[7:0]= "00010111" determina que la ALU realizará una suma, asignando a la señal OPERACION_A[3:0] el valor "0001". A continuación se describen las señales de control generadas:

CA_A = '1', BLEI es instrucción de salto.
CMUX1A = '1', deja pasar el valor constante R1S_A "00010111".
CMUX1B = '1', deja pasar el valor constante R2S_A "00000110".
CMUX3_A = '0', el mux. 3 seleccionará el RESULTADO[15:0] de la ALU.
CMUX4_A = '1', pues el mux. 4 también seleccionará RESULTADO[15:0].
EC_A = '1', ya que BLEI empleará el comparador.
ENABLE_A = '1', para que la unidad de conteo permanezca habilitada.
ER = '1', para habilitar la lectura de registros.
EU_A = '1', para habilitar la ALU.
INCREMENT_A = '1', para que la unidad de conteo continúe generando instrucciones.
LI_A = '1', ya que BLEI genera una nueva dirección para la unidad de conteo.
PD_A = '0', porque BLEI no implica regresar al contador anterior.
PU_A = '0', porque BLEI no implica cambiar al siguiente contador.
RESET = '0', anteriormente ya se estableció que el contador 0 debe iniciar en 0.
SET = '0', anteriormente ya se escribieron las instrucciones en la memoria.
WB_A = '0', para deshabilitar la actualización de registros.
WR_DAT_A = '0', para deshabilitar la escritura en la memoria de datos.
WR_INS_A = '0', para que en el siguiente ciclo se vuelva a leer la Mem. de instrucciones.

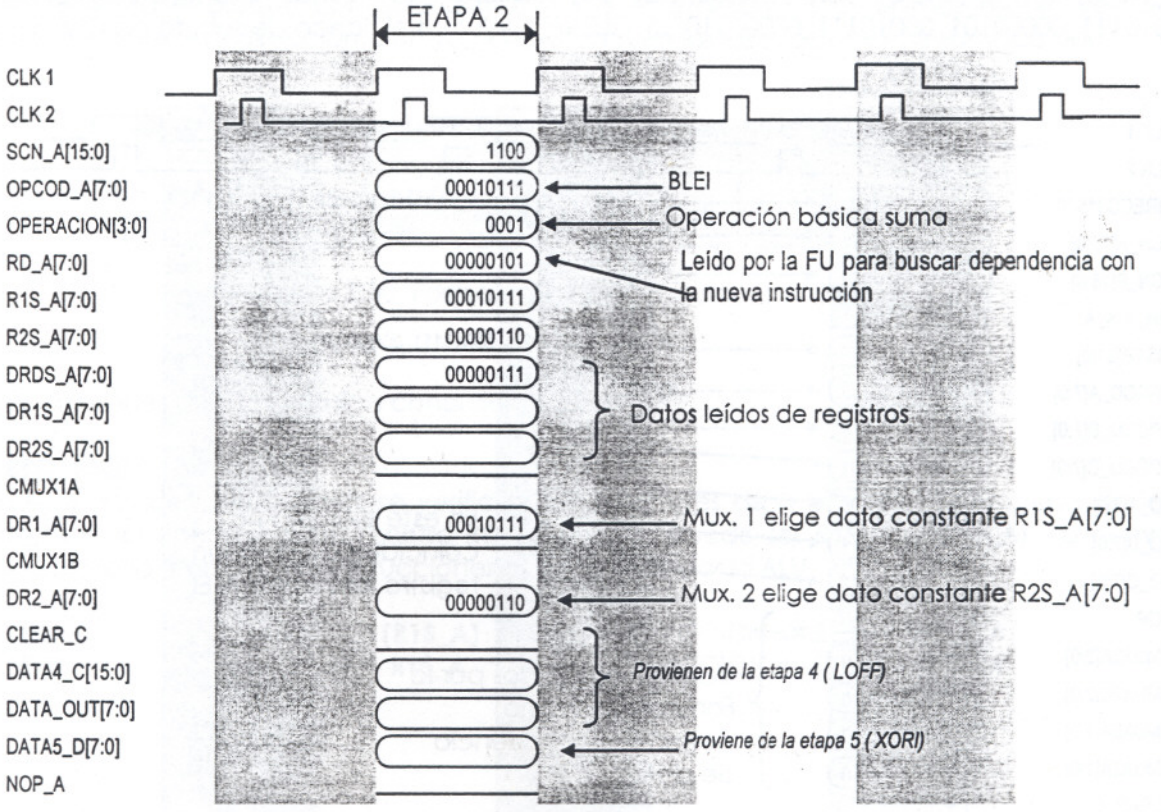


Figura 2.32 Diagrama de la etapa 2 de BLEI

Al bloque de registros de propósito general llegan las direcciones de los 3 posibles registros fuente: R1S_A[7:0]= "00010111", R2S_A[7:0]= "00000110" y RD_A[7:0]= "00000101". En este caso también se requiere el dato contenido en RD_A[7:0], en la señal DRDS_A[7:0]= "00000111".

El multiplexor 1A tiene su línea de selección activa y por lo tanto elige el valor constante R1S_A[7:0]. El multiplexor 1B también elige a la constante R2S_A[7:0] con la línea CMUX1B= '1'. Estos datos pasan a las señales DR1_A[7:0] y DR2_A[7:0], respectivamente.

En el flanco de subida de CLK1 el Registro 2 almacena las señales provenientes del Registro 1, las señales de control, los valores derivados de los multiplexores y el valor leído de RD_A[7:0], todas con terminación "_A". Además lee CLEAR_C, DATA4_C[15:0], DATA_OUT[7:0] y DATA5_B[7:0]; estos buses podrían necesitarse en una dependencia de datos con la instrucción que se encuentra en la etapa 4 del *pipeline* (XORI, de tipo OPERACIÓN)

Aquí se eligen los operandos para la ALU y el comparador. Por tratarse de la instrucción BLEI y por no haberse detectado en la primera etapa dependencia de datos (NOP_B= '0'), el mux. 2A tiene como línea de selección CMUX2A_B[2:0]= "001", para elegir SCN_B[15:0]. El mux. 2B, con CMUX2B_A[2:0]= "001", elige el valor de R2_B[7:0] y agrega ocho 0's a la izquierda, para que el operando OPER1B [15:0] tenga 16 bits.

En el flanco de subida de CLK2 la dirección del registro destino RD_B[7:0]= "00000101" es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos de esta instrucción con la que recién entró al *pipeline*.

OPER1A[15:0] y OPER1B[15:0] son leídos por la ALU, que realiza la suma de ambas cantidades, pues el Control decodificó esta instrucción como una adición y generó OPERACION_A[3:0]= "0001". La señal RESULTADO[15:0] tiene el valor *SCN_B + R2S_B = "00000000_00001100" + "00000000_00000110" = "00000000_00010010"*. El mux. 3 elige RESULTADO[15:0] y lo asigna a DALU_B[15:0].

También para el comparador hay que elegir operandos. El primer operando, ya que no se detectó dependencia de datos, es la constante DRDS_B[7:0], seleccionada con la señal CMUX5_B[1:0]= "10". Por lo tanto la señal RDD_B[7:0] toma el valor "00000111". Para decidir cuál será el segundo vector a comparar se emplea el multiplexor 6. Con la línea de selección CMUX6_B[1:0]= "10" se elige la constante DR1_B[7:0], que propaga su valor a DR1_COMP[7:0].

El comparador verifica si el contenido del registro RD es menor que el valor constante en R1S (RD < R1S). En este ejemplo, podemos observar que RDD_B[7:0]= "00000111" y R1S_B[7:0]= "00010111"; por lo tanto la condición se cumple. Este elemento del diseño genera COMP= '1' . Se calcula el OR de esta señal con LO_B='0' y se obtiene LI_B= '1'. Además el OR entre COMP, PD_B='0' y CA_B='0' resulta en CLEAR_C='1'. Esto ocasionará que en la siguiente etapa la unidad de conteo modifique su secuencia: en el siguiente ciclo del *pipeline* debería generarse la dirección "00000000_00001111"; pero en lugar de ello se leerá la nueva dirección calculada por la ALU: "00000000_00010010", y a partir de este número continuará el conteo. Además se borrará el contenido de los Registros 1 y 2.

Si hubiera ocurrido el caso contrario, en el que la condición RD < R1S no se cumple el valor de COMP sería '0'. Por lo tanto LO_B='0' y CLEAR_C='0', y la secuencia de instrucciones no se vería modificada.

En el siguiente flanco de subida de CLK1 el Registro 3 lee DALU_B[15:0] y RDD_B[7:0], además de algunas señales de salida del Registro 2: OPCOD_B[7:0], RD_B[7:0], CMUX4_B, WB_B y WR_DAT_B. Todas estas señales pasan a la siguiente etapa con la terminación "_C".

BLEI - Etapa 4

Las instrucciones de tipo RAMIFICACIÓN no accesan a la memoria de datos. La señal WR_DAT_C está inactiva y aunque DATA_OUT no se necesita, esta señal pasa al Registro 4.

Las señales PD_B= '0' y PU_B= '0' son leídas por la unidad de conteo, que no cambiará el número de contador (si suponemos que se encontraba en el contador 0, ahí permanecerá). LI_B= '1' provoca que la unidad de conteo lea una nueva dirección a partir de la cual seguirá el conteo. La señal DALU_B[15:0] contiene esta dirección.

En el flanco de subida de CLK1, los Registros 1 y 2 leen CLEAR_C= '1' y borran su contenido. Esto significa que las señales de salida no son copia de las entradas a estos registros. Todos los bits de salida son 0's. Hay más implicaciones: las instrucciones que entraron al *pipeline* después de BLEI ya no se ejecutarán. Por ejemplo, hemos dicho que BLEI se encontraba en la

dirección 12 de la memoria de instrucciones. Si BLEI ya se encuentra en la Etapa 4, debe haber una instrucción en la etapa 2 con dirección 14 vaciando sus señales en el Registro 2, y otra instrucción en la etapa 1 con dirección 15 dando valores al Registro 1. El contenido de ambos registros será ignorado y la unidad de conteo, al leer una nueva dirección, generará una nueva instrucción para la etapa 1.

Al igual que en las dos etapas anteriores RD_D[7:0] se envía a la *Forward Unit* para verificar en CLK2= '1' si la nueva instrucción del *pipeline* tiene como registro fuente al registro destino de BLEI, RD_C[7:0]= "00000101". Pero en el caso de esta instrucción es innecesaria tal comparación. BLEI no puede ocasionar una dependencia de datos, ya que no actualiza ningún registro.

Del Registro 3 pasan al Registro 4 las señales DALU_C[7:0]= "00010010", la dirección del registro destino RD_C[7:0]= "00000101", las líneas CMUX4_C= '1' y WB_C= '1', así como DATA_OUT_C[7:0] (que conserva el mismo valor del ciclo anterior, "01100010", porque en este ciclo no se realizó una lectura).

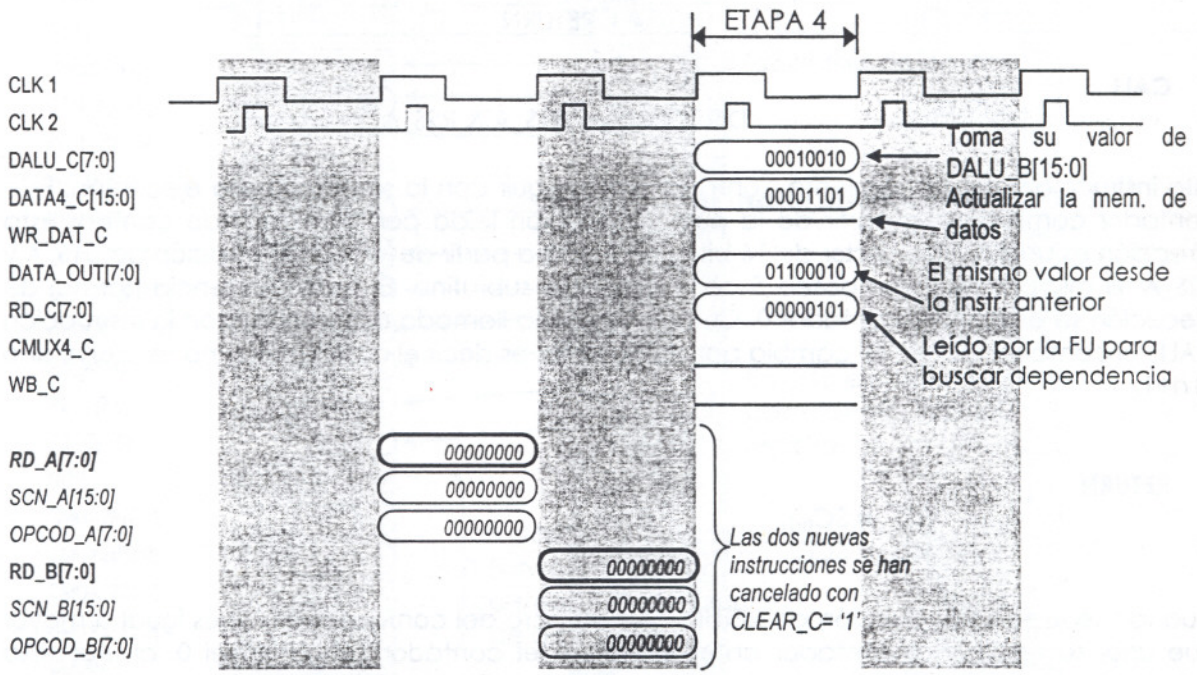


Figura 2.34 Diagrama de la etapa 4 de BLEI

BLEI - Etapa 5

Al igual que las instrucciones de tipo STORE, la instrucción BLEI (y el resto de instrucciones de tipo RAMIFICACIÓN) no requieren la etapa de Escritura de Resultados.

Si suponemos que en esta instrucción se detectó una dependencia de datos, la ejecución de BLEI sería diferente. La comparación a realizar para BLEI tendría como operando el contenido de un registro que aún no ha sido actualizado. Supongamos que este valor pertenece a la instrucción que entró al *pipeline* **tres ciclos** antes (XORI). Ocurrirían los siguientes cambios en la ejecución de BLEI:

- En la Etapa 1 (Lectura): La *Forward Unit* lee la INSTRUCCION[31:0], la DIRECCION[15:0] actual, la dirección de las instrucciones anteriores (SCN_A[15:0] y SCN_B[15:0]) y su código de operación (OPCOD_A[7:0], OPCOD_B[7:0] y OPCOD_C[7:0]). La palabra INSTRUCCION[31:0] contiene la dirección de los registros fuente, en este caso RD.

Supongamos que este registro coincide con el registro destino de XORI, es decir, RD= "00000111" = RD_C[7:0]. Al detectar la existencia de dependencia de datos, la *Forward Unit* genera CMUX5[1:0]= "00" y NOP= '0' para resolver la dependencia de datos.

- *En la Etapa 2 (Decodificación):* El Registro 2 lee la señal NOP_A= '0' y CMUX5_A[2:0]= "00" y en el mismo flanco de subida permite la salida de CMUX5_B y NOP_B.

- En la Etapa 3 (Ejecución): El mux. 5 elige como primer operando del comparador a la señal DATA5_D[7:0], que contiene el valor calculado por la ALU para XORI tres ciclos de reloj antes (*"00000000_00110000"*). De esta manera **no hay retrasos** en la ejecución de BLEI.

Para ejecutar saltos incondicionales la UAM - RISC II cuenta con 3 instrucciones, que son de llamada a una subrutina y regreso de ella:

- CALL
- JUMP
- RETURN

- **CALL**

$SC_n = SC_{n+1}$, DIRECCION:= **R1S_A** & R2S_A

Esta instrucción elige el siguiente contador para seguir con la secuencia de ejecución. Este contador comenzará a partir de la nueva dirección leída por la unidad de conteo; esta dirección es un std_logic_vector de 16 bits, generado a partir de la concatenación de R1S_A y R2S_A. El microprocesador posee 5 contadores de subrutina. En una secuencia normal de ejecución se emplea el contador 0. Cuando hay una llamada a subrutina con la instrucción CALL, el contador actual se cambia por el siguiente, es decir el contador ya no será el *n*, sino el *n+1*.

- **RETURN**

$SC_n = SC_{n-1}$, si $n \geq 1$
$SC_n = SC_0$, si $n = 0$

Cuando se lee una instrucción de RETURN y el número del contador actual es igual o mayor que uno, se regresa al contador anterior. Pero si el contador actual es el 0, al leer esta instrucción la secuencia de conteo permanecerá sin cambios. Hasta que el microprocesador lee una instrucción RETURN puede salir de la subrutina en la que entró con CALL. Con esta instrucción regresa al contador anterior, y continúa la secuencia en la dirección en que se encontraba antes del salto. Estas 2 instrucciones tienen una ejecución mucho más simple que el resto. En la etapa 3 activan CLEAR_B para borrar el contenido de los Registros 1 y 2; ambas instrucciones provocan que se cambie de contador de subrutina; además, CALL indica en qué dirección reiniciar el conteo.

- **JUMP**

DIRECCION <= SCN_B + R2S_A

La nueva dirección de la memoria de instrucciones que se leerá es calculada por la ALU cuando se encuentra una instrucción JUMP. La dirección es de 16 bits, y el corrimiento es R2S_A, de 8 bits. Como ejemplo analicemos la ejecución de CALL:

CALL - Etapa 1

Esta instrucción es la única que activa la línea PU_B, pero en una etapa posterior. En este momento, supongamos que las instrucciones anteriores que aún se encuentran en ejecución no han activado esta línea. En el flanco de subida de CLK1 el contador actual, supongamos que es el número 0, genera una nueva dirección si están activas las señales ENABLE_A e INCREMENT_A, por ejemplo DIRECCION[15:0]= "00000000_00001101".

Esta localidad de la memoria de instrucciones es leída cuando WR_INS_A = '0' y se obtiene la señal INSTRUCCIÓN[31:0]= "00011000_11111111_00000000_00000100".

En el flanco de subida de CLK1, si CLEAR_C= '0', el Registro 1 del *pipeline* lee las señales de esta etapa: INSTRUCCIÓN[31:0], DIRECCION[15:0], CMUX2A[2:0]= "010", CMUX2B[2:0]= "010", CMUX5A[1:0]= "10", CMUX6A[1:0]= "10", y NOP= '0'.

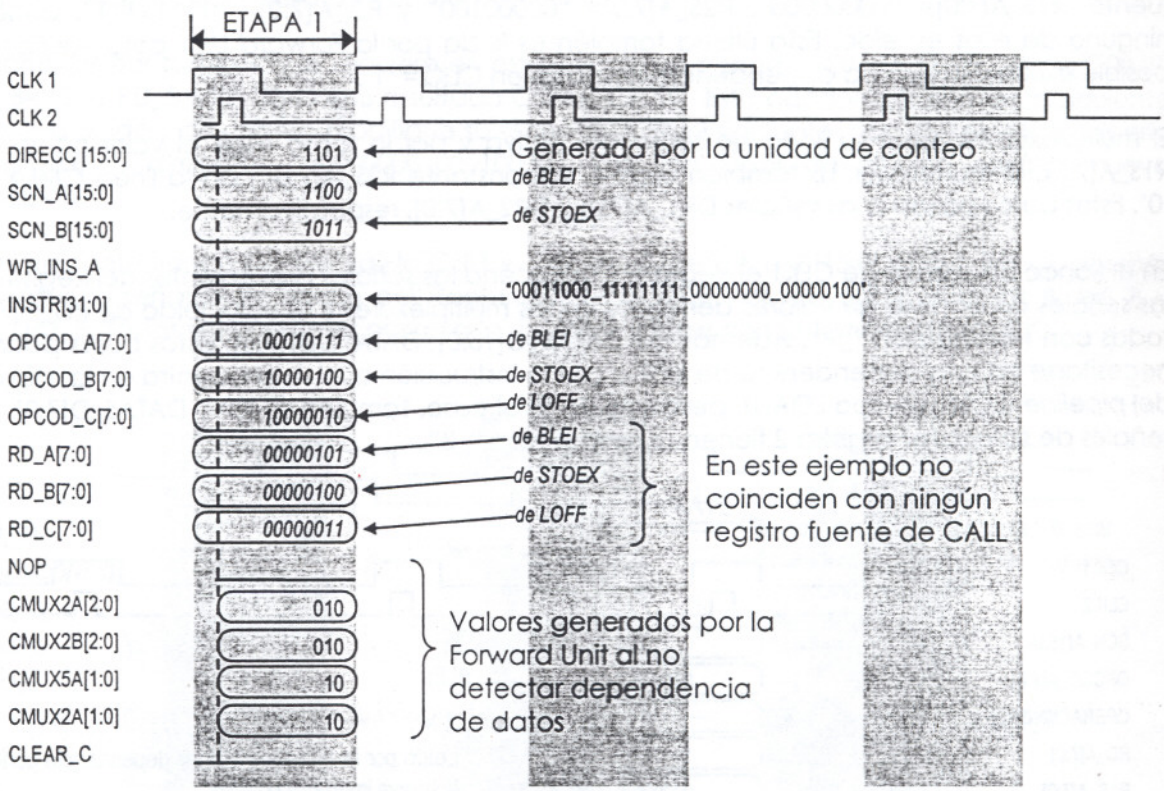


Figura 2.35 Diagrama de la etapa 1 de CALL

CALL - Etapa 2

Las señales INSTRUCCIÓN[31:0] y DIRECCION[15:0] se descomponen en nuevas señales para esta etapa: OPCOD_A[7:0]= "00011000", RD_A[7:0]= "11111111", R1S_A[7:0]= "00000000", R2S_A[7:0]= "00000100", SCN_A[15:0]= "00000000_00001101". Cuando la unidad de control decodifica OPCOD_A[7:0]= "00011000" determina que la ALU no realizará ninguna operación y entonces genera la señal OPERACION_A[3:0]= "1111". A continuación se describen las señales de control generadas:

CA_A = '1', CALL es instrucción de salto.
CMUX1A = '1', para dejar pasar el valor constante R1S_A "00000000".
CMUX1B = '1', para dejar pasar el valor constante R2S_A "00000100".
CMUX3_A = '1', el mux. 3 seleccionará los 2 valores constantes R1S_A y R2S_A.
CMUX4_A = '0', mux. 4 seleccionará el dato leído de memoria (que no usará).
EC_A = '0', CALL no empleará el comparador.

ENABLE_A = '1', para que la unidad de conteo permanezca habilitada.
ER = '0', para deshabilitar la lectura de registros.
EU_A = '0', para deshabilitar la ALU.
INCREMENT_A = '0', para que la unidad de conteo no genere una nueva instrucción.
LI_A = '1', CALL genera una nueva dirección para la unidad de conteo.
PD_A = '0', porque CALL no implica regresar al contador anterior.
PU_A = '1', porque con CALL la ejecución se transfiere al siguiente contador.
RESET = '0', anteriormente ya se estableció que el contador 0 debe iniciar en 0.
SET = '0', anteriormente ya se escribieron las instrucciones en la memoria.
WB_A = '0', para deshabilitar la actualización de registros.
WR_DAT_A = '0', para deshabilitar la escritura en la memoria de datos.
WR_INS_A = '0', para que en el siguiente ciclo se vuelva a leer la Mem. de instrucciones.

Al bloque de registros de propósito general llegan las direcciones de los 3 posibles registros fuente: R1S_A[7:0]= "00000000", R2S_A[7:0]= "00000100" y RD_A[7:0]= "11111111", aunque ninguno de ellos es leído. Esta última también es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos en CLK2= '1'.

El multiplexor 1A tiene su línea de selección activa y por lo tanto elige el valor constante R1S_A[7:0]. El multiplexor 1B también elige a la constante R2S_A[7:0] con la línea CMUX1B= '0'. Estos datos pasan a las señales DR1_A[7:0] y DR2_A[7:0], respectivamente.

En el flanco de subida de CLK1 el Registro 2 almacena las señales provenientes del Registro 1, las señales de control, los valores derivados de los multiplexores y el valor leído de RD_A[7:0], todas con terminación "_A". Además lee DATA4_C[15:0], DATA_OUT[7:0]; estos buses podrían necesitarse en una dependencia de datos con la instrucción que se encuentra en la etapa 4 del *pipeline* (LOFF, de tipo LOAD), pero este no es el caso. También se lee y DATA5_D[7:0]. Las señales de salida del Registro 2 tienen la terminación "_B".

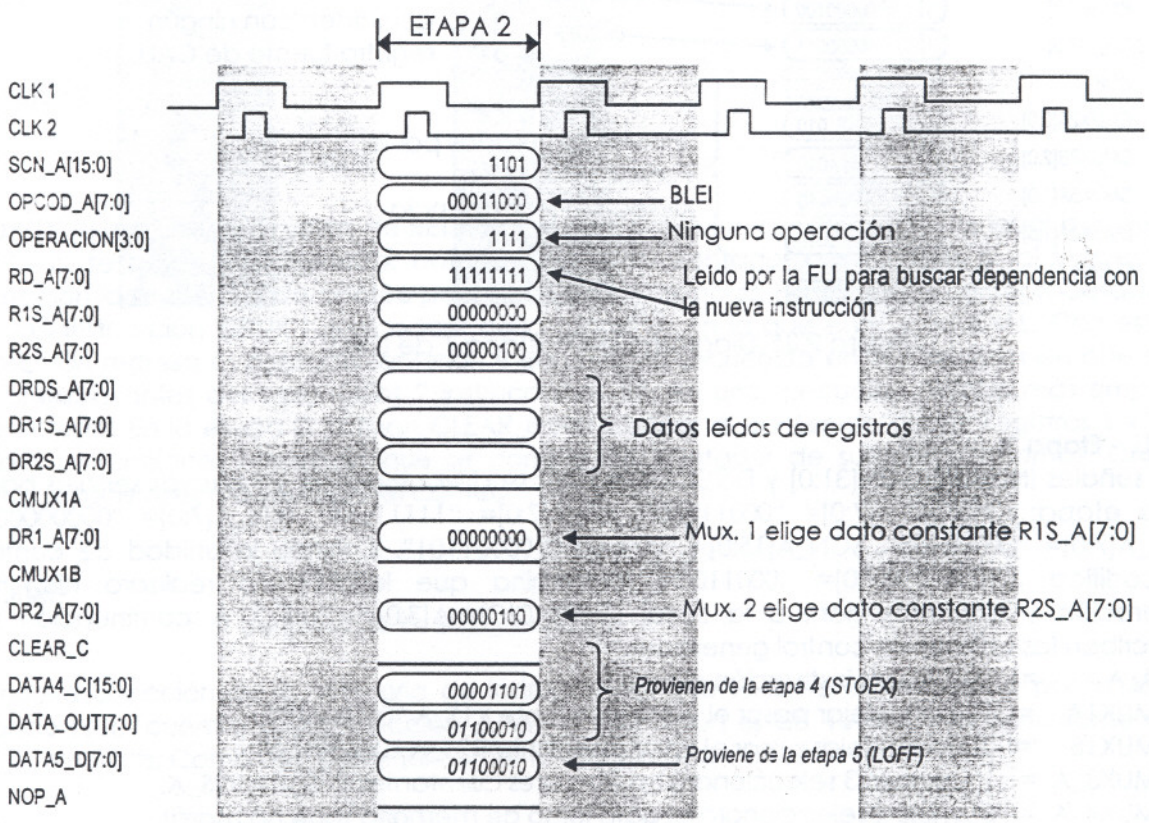


Figura 2.36 Diagrama de la etapa 2 de CALL

CALL - Etapa 3

En esta etapa se eligen los operandos para la ALU mediante los mux. 2A y 2B, aunque CALL no los necesita.

En el flanco de subida de CLK2 la dirección del registro destino RD_B[7:0]= "11111111" es leída por la *Forward Unit* para verificar la posible existencia de una dependencia de datos de esta instrucción con la que recién entró al *pipeline*.

OPER1A[15:0] y OPER1B[15:0] son leídos por la ALU, que se encuentra deshabilitada por EU_B= '0', y por lo tanto la señal RESULTADO[15:0] toma el valor "00000000_00000000".

El mux. 3, con su línea CMUX3_A = '0' elige las señales R1S_B[7:0]= "00000000" y R2S_B[7:0]= "00000100" y las concatena para generar DALU_B[15:0], que en la siguiente etapa será la dirección que lea la unidad de conteo.

El comparador tampoco se requiere en esta ocasión, y genera COMP= '0'. El OR de COMP, CA_B= '1' y PD_B= '0' da como resultado CLEAR_C= '1'. Esto ocasionará que en la siguiente etapa la unidad de conteo modifique su secuencia. En el siguiente ciclo del *pipeline* debería generarse la dirección "00000000_00010000". Pero en lugar de ello se leerá la nueva dirección.

En el siguiente flanco de subida de CLK1 el Registro 3 lee DALU_B[15:0] y RDD_B[7:0], además de algunas señales de salida del Registro 2: OPCOD_B[7:0], RD_B[7:0], CMUX4_B, WB_B y WR_DAT_B. Todas estas señales pasan a la siguiente etapa con la terminación "_C".

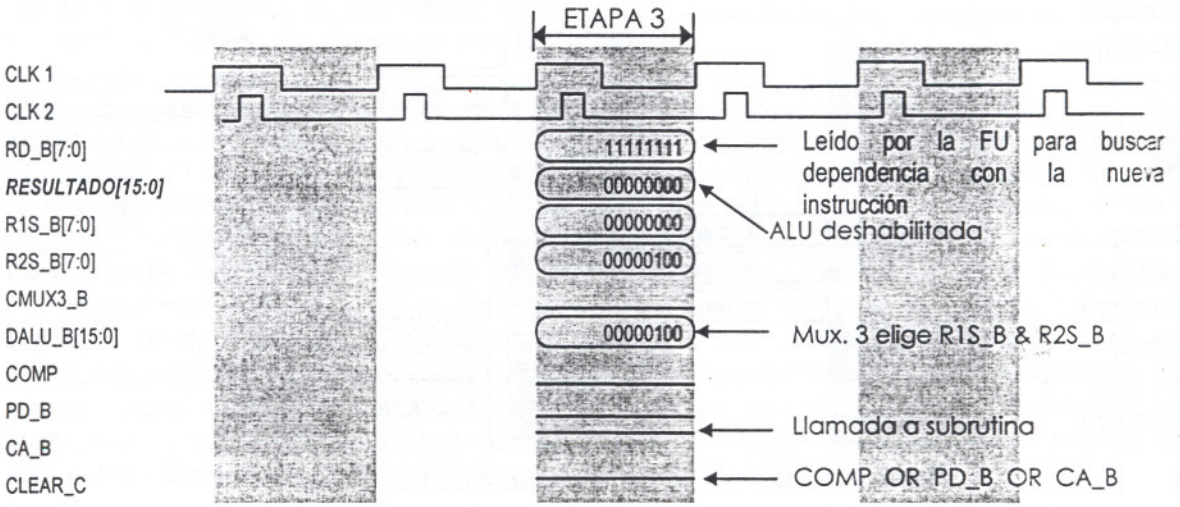


Figura 2.37 Diagrama de la etapa 3 de CALL

CALL - Etapa 4

En las instrucciones de tipo RAMIFICACIÓN no hay acceso a la memoria de datos. La señal WR_DAT_C está inactiva y aunque DATA_OUT no se necesita, esta señal es leída por el Registro 4. Las señales PD_B= '0' y PU_B= '1' son leídas por la unidad de conteo, que cambiará al siguiente contador (si suponemos que se encontraba en el contador 0, cambiará al 1). Además LI_B= '1' provoca que en la subida de CLK1 la unidad de conteo lea la nueva dirección a partir de la cual seguirá el conteo. La señal DALU_B[15:0] tiene el valor "00000000_00000100".

En el flanco de subida de CLK1, los Registros 1 y 2 leen CLEAR_C= '1' y borran su contenido. Esto significa que las señales de salida no son copia de las entradas a estos registros. Todos los bits de salida son 0's. Hay más implicaciones: las instrucciones que entraron al *pipeline* después de CALL ya no se ejecutarán. Como ejemplo hemos supuesto que CALL se localiza en la dirección 13 de la memoria de instrucciones. Si CALL ya se encuentra en la Etapa 4, debe haber una instrucción en la etapa 2 con dirección 15 vaciando sus señales en el Registro 2, y otra instrucción en la etapa 1 con dirección 16 dando valores al Registro 1. El contenido de ambos registros será ignorado y la unidad de conteo, al leer una nueva dirección, generará una nueva instrucción para la etapa 1.

Al igual que en las dos etapas anteriores, la *Forward Unit* verifica en CLK2= '1' si la nueva instrucción del *pipeline* tiene como registro fuente al registro destino de CALL, RD_C[7:0]= "11111111". Pero en el caso de esta instrucción no es necesaria tal comparación, pues CALL no puede ocasionar una dependencia de datos, ya que no actualiza ningún registro.

Del Registro 3 pasan al Registro 4 las señales DALU_C[7:0]= "00010001", la dirección del registro destino RD_C[7:0]= "11111111", las líneas CMUX4_C= '1' y WB_C= '0', así como DATA_OUT_C[7:0] (que conserva el mismo valor del ciclo anterior, "01100010", porque en este ciclo no se realizó una lectura).

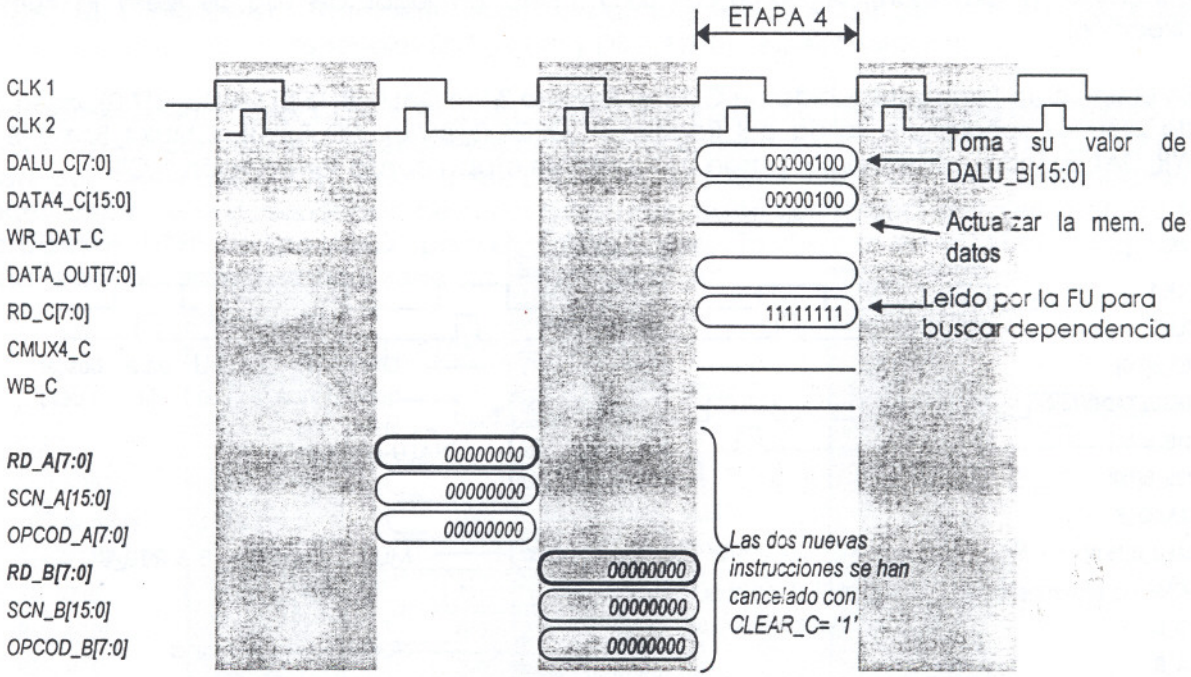


Figura 2.38 Diagrama de la etapa 4 de CALL

CALL - Etapa 5

La instrucción CALL (y el resto de instrucciones RAMIFICACIÓN) no necesita la etapa de Escritura de Resultados.

2.1.9 Resolviendo la Dependencia de Datos

Cuando una instrucción llega a la Etapa de Ejecución es posible que uno o ambos operandos requeridos aún no estén disponibles en el registro correspondiente. Entonces ocurre una dependencia de datos. La *Forward Unit* resuelve esta situación ocasionando el

mínimo de retrasos en el *pipeline*, pues la identifica desde la etapa de Lectura de la Instrucción.

No todas las instrucciones ocasionan una dependencia de datos, tal es el caso de STORE y RAMIFICACIÓN, que no actualizan el contenido de un registro. En el siguiente cuadro se presentan los casos en que puede llegar a ocurrir una dependencia de datos:

ETAPA	**1, LECTURA** (instrucción actual = *i*)	**2, DECODIF. Y OPERANDOS** *(i-1)*	**3, EJECUCIÓN** *(i-2)*	**4, ACCESO A MEMORIA** *(i-3)*
SEÑAL A EXAMINAR	INSTRUCCION[31:0]	OPCOD_A[7:0]	OPCOD_B[7:0]	OPCOD_B[7:0]
TIPO DE INSTRUCCIÓN	OPERACIÓN, LOAD, STORE o RAMIFICACIÓN	**CASO 1:** OPERACIÓN o LOAD		
			CASO 2: OPERACIÓN o LOAD	
				CASO 3: OPERACIÓN o LOAD

Tabla 2. 3 Casos de dependencia de datos

Ahora detallaremos cada una de estas 3 posibles situaciones de dependencia. En el caso 1, la dependencia de datos ocurre entre la instrucción actual y la que entró en el ciclo anterior al *pipeline*, es decir, entre instrucciones sucesivas. El conflicto ocurre porque la instrucción actual (de cualquier tipo) tiene como operando al registro destino de la instrucción de tipo OPERACIÓN o LOAD que se encuentra en la etapa 2, apenas en decodificación. Obviamente este dato no estará listo en el momento que se requiere. Gracias a las modificaciones realizadas en el diseño del UAM - RISC II, no es necesario "insertar una burbuja", es decir, no se detiene la ejecución de la instrucción actual durante un ciclo de reloj. Lo que sucede es la activación de la señal NOP, que indicará a los multiplexores de la etapa 3 que deben seleccionar al dato que se acaba de calcular. De esta forma la instrucción actual podrá continuar su ejecución con un dato válido, aunque éste no haya sido escrito aún en el registro correspondiente (Figura 2.13).

En el segundo caso, la dependencia ocurre entre la instrucción actual y la que entró al *pipeline* 2 ciclos antes, que se encuentra en la etapa de Ejecución. En la etapa 1 de la instrucción actual *Forward Unit* detecta el conflicto cuando la instrucción actual (de cualquier tipo) requiere leer el registro destino de la instrucción (de tipo OPERACIÓN o LOAD) que se encuentra en la etapa 3, y que por lo tanto no tiene un valor actualizado. El resultado calculado continuará su ejecución a través del *pipeline*, y una copia de su valor irá a la etapa 2, donde inmediatamente formará parte de la instrucción actual, resolviendo el conflicto (Figura 2.14).

En el 3er caso se observa que también puede ocurrir una dependencia de datos con la instrucción que entró 3 ciclos antes, y que por lo tanto está en la Etapa de Acceso a memoria. Este caso se resuelve de forma similar al caso 2. Ahora la instrucción actual *i* necesita el valor contenido en el registro destino de la instrucción *i-3*, que aún no ha sido

actualizado. El resultado ya calculado continuará su ejecución a través del *pipeline*, al tiempo que una copia de su valor irá a la etapa 2 de la instrucción actual, para convertirse en uno de los operandos de la ALU o del comparador (Figura 2.15).

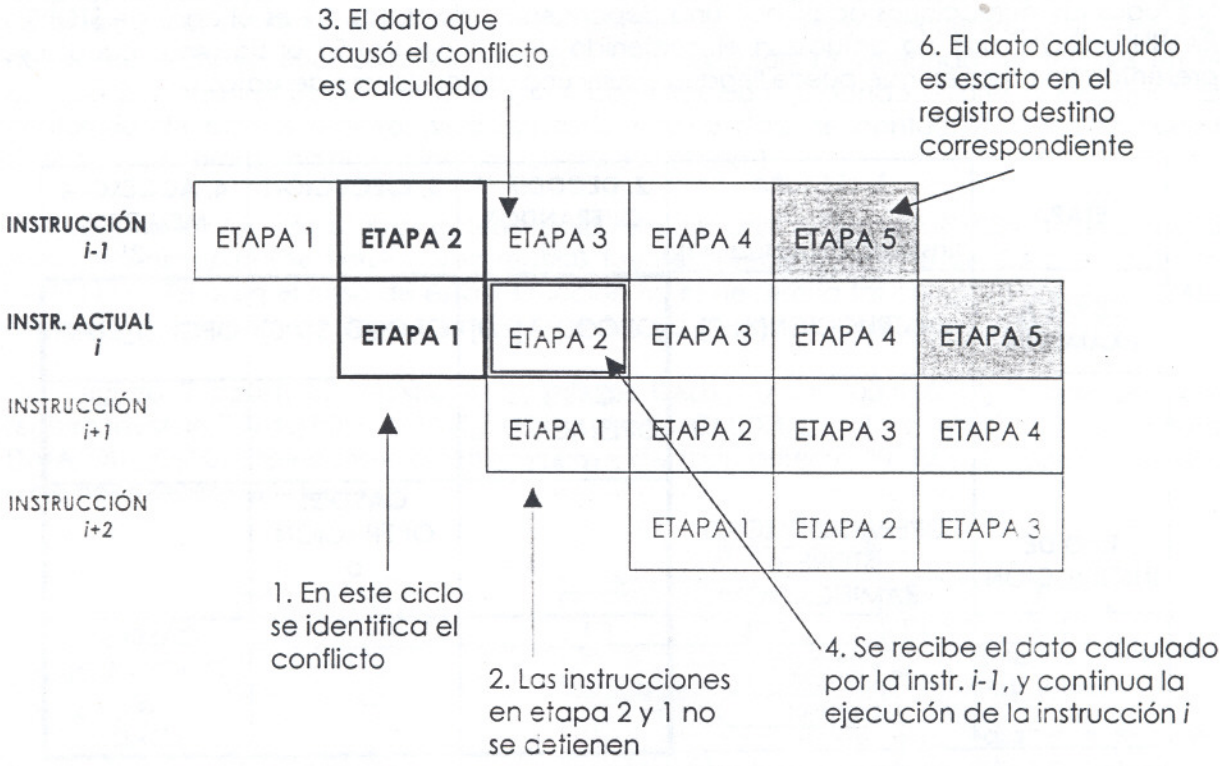


Figura 2.39 Dependencia de datos entre 2 instrucciones inmediatas (Caso 1)

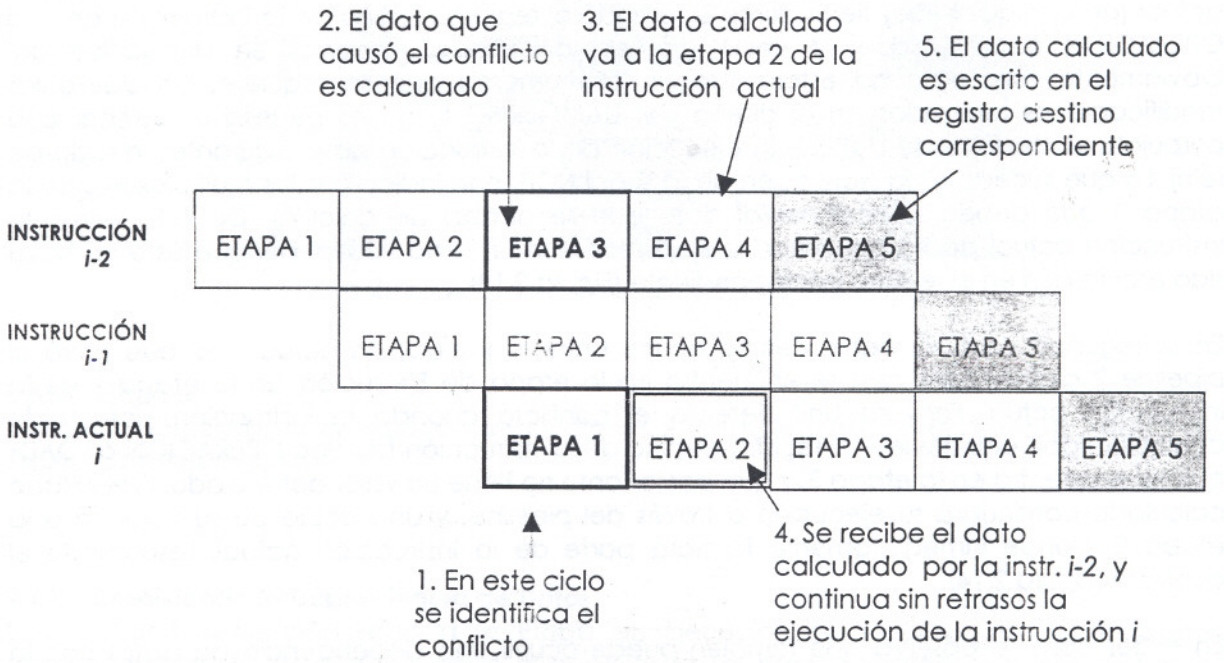


Figura 2.40 Dependencia de datos entre las instrucciones *i*, *i-2* (Caso 2)

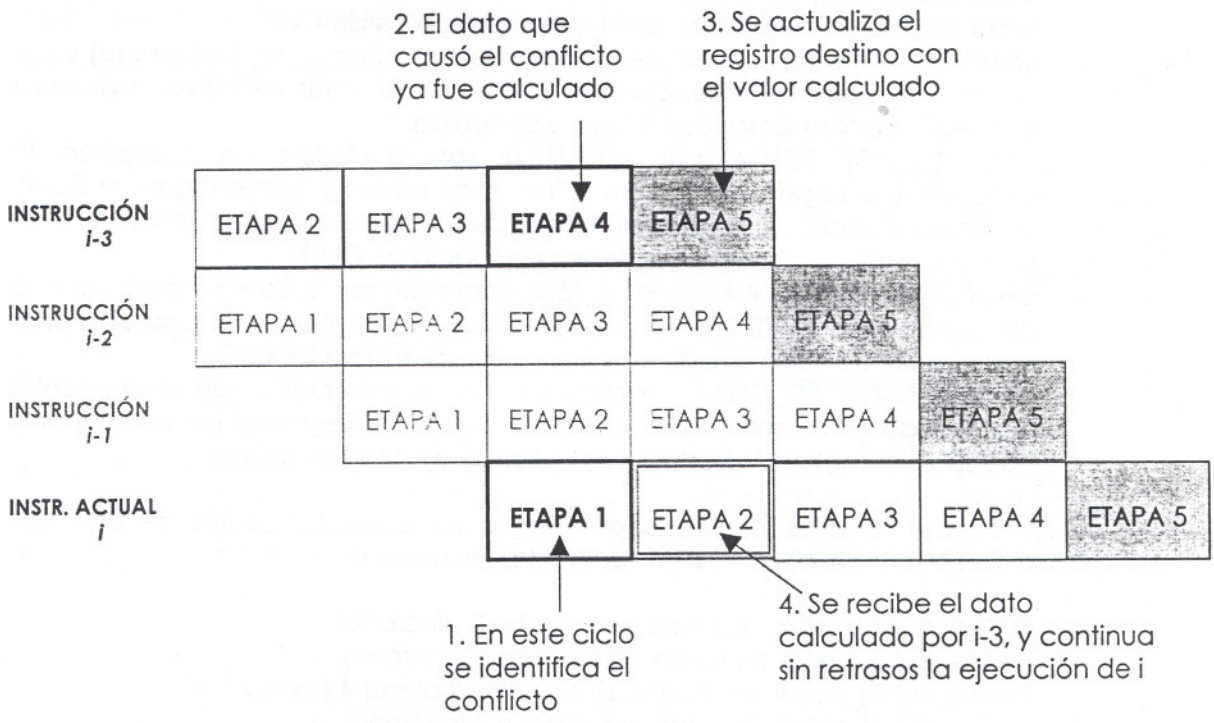


Figura 2.41 Dependencia de datos entre las instrucciones *i*, *i-3* (Caso 3)

Después de analizar estos 3 casos generales de dependencia de datos podemos concluir que la *Forward Unit* puede resolver esta situación con ayuda de los multiplexores de la Etapa de Ejecución sin ocasionar retrasos, lo que representa una ventaja significativa en el diseño del UAM - RISC II.

2.2 Diseño del UAM – RISC II

La arquitectura que a continuación se puntualiza se basa en la descripción del UAM – RISC [3]. En el UAM – RISC II, además de presentarse la especificación en VHDL, se realizaron extensiones al diseño del UAM – RISC. Este microprocesador cuenta con una estructura de *pipeline* con 5 etapas. Luego que las señales son procesadas en una etapa durante un ciclo del reloj, se almacena su valor en un registro con el siguiente flanco de subida de CLK1.

2.2.1 La señal de reloj

En este microprocesador existen 2 relojes: CLK1 y CLK2. El primero de ellos sincroniza las 5 etapas del *pipeline*, ya que todos los registros del *pipeline* reciben la señal de CLK1. Hay además otros elementos sincronizados por CLK1: la unidad de conteo y el bloque de registros de propósito general. La señal CLK2 se hace necesaria para resolver la dependencia de datos. Al analizar anteriormente la ejecución de diferentes tipos de instrucciones explicamos que la *Forward Unit* realiza diversas comparaciones, por ejemplo de los registros fuente de la instrucción que recién entró al *pipeline* contra los registros destino de las 3 instrucciones que empezaron su ejecución antes, y que todavía se encuentran dentro del *pipeline*. Todas estas direcciones están disponibles en el flanco de subida de CLK1, lo que nos haría pensar que no es necesaria otra señal de reloj. Sin embargo, el detectar correctamente una dependencia de datos demandan el uso de un segundo reloj, cuyo flanco de subida ocurra después del de CLK1. Veamos cada una de las señales que recibe la *Forward Unit*:

- DIRECCIÓN[15:0], es generada por la unidad de conteo (síncrona), por lo tanto llega a la *Forward Unit* en el flanco de subida de CLK1.
- INSTRUCCIÓN[31:0], es leída de la memoria de instrucciones **(asíncrona)** y por lo tanto su valor no es estable en CLK1= '1'. Su valor definitivo lo alcanza alrededor de 50ns después del flanco de subida.
- OPCOD_A[7:0], SCN_A[15:0], RD_A[15:0] son el código de operación, la dirección y el registro destino de la instrucción que se encuentra en la Etapa de Decodificación. Además son señales de salida del Registro 1 (síncrono), por lo que su valor es estable en el flanco de subida de CLK1.
- OPCOD_B[7:0], SCN_B[15:0], RD_B[15:0] corresponden a la instrucción que se encuentra Ejecución. Son señales de salida del Registro 2 (síncrono), por lo que su valor también es estable en el flanco de subida de CLK1.
- OPCOD_C[7:0], RD_C[15:0] corresponden a la instrucción que se encuentra accesando a la memoria de datos. Son salidas del Registro 3 (síncrono), por lo que su valor también es estable en el flanco de subida de CLK1.

Cuando efectivamente se detecta una dependencia de datos hay varias señales que pueden ayudar a resolverla sin ocasionar retrasos en la ejecución del *pipeline*:

- DATA4_C[7:0], señal de salida del Registro 3 (síncrono)
- DATA4_B[7:0], señal de salida del Registro 2 (síncrono)
- DATA5_D[7:0], señal seleccionada por el multiplexor 4 **(asíncrono)**
- DATA5_B[7:0], señal de salida del Registro 3 (síncrono)
- DATA_OUT[7:0], palabra leída de la memoria de datos **(asíncrona)**
- DATA_OUT_B[7:0], señal de salida del Registro 2 (síncrono)
- SCN_B[15:0], señal de salida del Registro 2 (síncrono)
- DR1_B[7:0], DR2_B[7:0], ambas son salidas del Registro 2 (síncrono)

A partir de esto podemos concluir que las señales INSTRUCCIÓN[31:0], DATA5_D[7:0], DATA_OUT[7:0] pueden no tener un valor estable en el flanco de subida de CLK1. Y si la *Forward Unit* realizara sus comparaciones para identificar una dependencia de datos en este flanco podríamos incurrir en un error. Por esto es preciso utilizar la segunda señal de reloj, CLK2, en este diseño.

En el diseño esquemático una señal de reloj se considera como una señal externa. Por lo tanto, requiere un IPAD *(input pad)* y un buffer (BUFG) para amplificar la señal. Sin embargo, no hay forma de describir una señal del reloj en el *Schematic Editor*. Solamente desde el *Logic Simulator* podemos especificar las características de este tipo de señales periódicas.

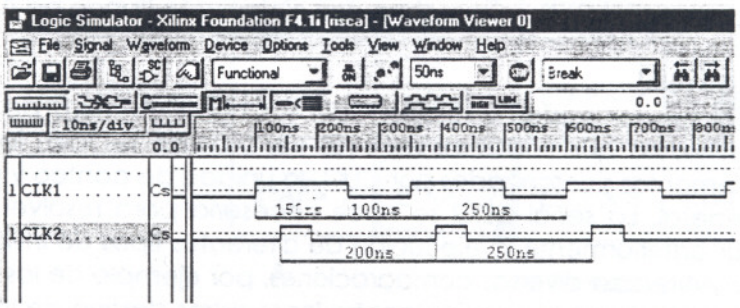


Figura 2.42 Señales CLK 1 y CLK 2 en el Logic Simulator. Ambos relojes tienen un período de 250ns. CLK 1 permanece en nivel alto 150ns, y 100ns en bajo. CLK 2 permanece 50ns en 1, y 200ns en 0 lógico.

2.2.2 Etapa 1 Lectura de la Instrucción *(Instruction Fetch)*

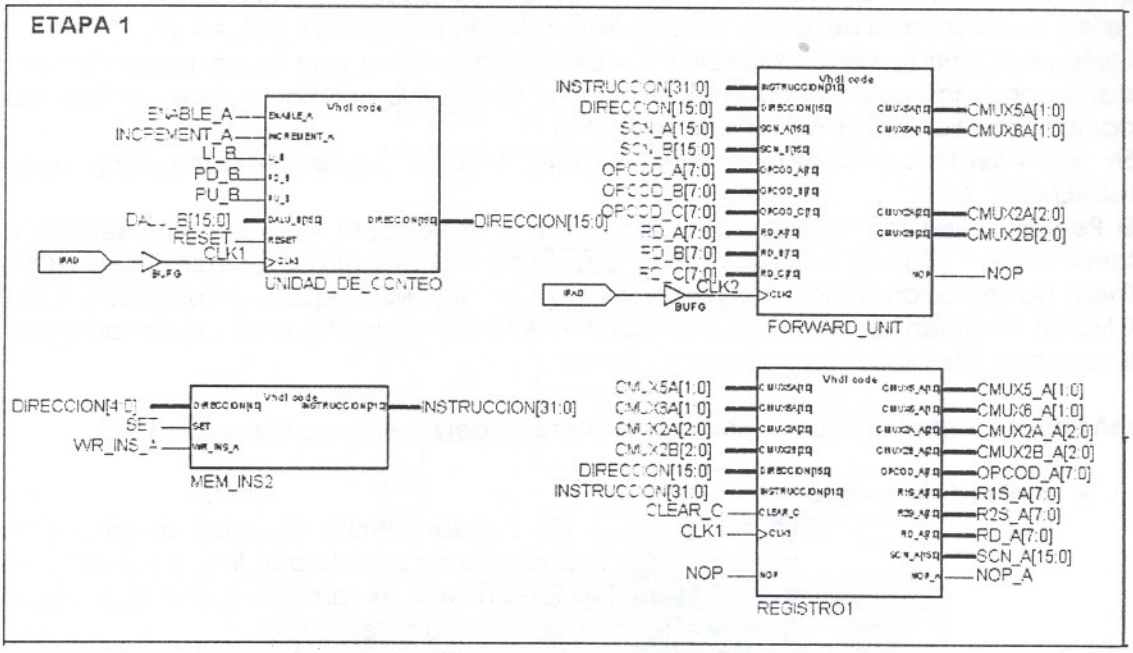


Figura 2.43 Elementos de la 1ª etapa del *pipeline*: Unidad de conteo, Memoria de datos, Memoria de Instrucciones, *Forward Unit*, Registro 1.

La etapa de Lectura de la Instrucción incluye a:

- unidad de conteo
- memoria de instrucciones
- Unidad de Dependencia de Datos (*Forward Unit*)
- Registro 1 (Registro del *pipeline*)

Durante esta etapa la unidad de conteo genera una dirección de la memoria de instrucciones. La palabra contenida en esta dirección de memoria sale al bus de instrucciones y se almacena en el Registro 1 del *pipeline*.

Al mismo tiempo, la instrucción es leída por la Unidad de Dependencia de Datos, que verificará si el contenido de alguno de los registros fuente aún no ha sido actualizado. Cuando esto ocurre, dependiendo del tipo de instrucción que generó la dependencia, la señal NOP puede habilitarse, para indicar que la dependencia es con la instrucción inmediata anterior, y no es necesario retrasar la ejecución durante un ciclo de reloj. También puede modificar las señales que controlan los multiplexores 2A y 2B (que seleccionan los operandos de la ALU), 5 y 6 (para el comparador). Esto con el propósito de poder leer el valor actualizado antes que sea escrito en el registro fuente.

De manera más específica, esto es lo que ocurre en la Etapa 1:

1. Se escriben las instrucciones en la memoria de instrucciones (sólo al inicio de la ejecución del microprocesador), cuando la señal INICIO ='1'.
2. El número de contador de subrutina se incrementa cuando PU_C ='1' o disminuye cuando PD_C ='1'.
3. La unidad de conteo lee una nueva dirección para alterar la secuencia de ejecución (cuando una instrucción de tipo Flow se encuentra en la etapa 4 y se ha cumplido la condición de salto que establece).

4. Se habilita al contador actual cuando ENABLE_A ='1', para que genere una dirección de la memoria de instrucciones. Sólo puede llegar a deshabilitarse cuando ocurre un salto en la secuencia de ejecución, ya que en tal caso se especificará una nueva dirección.
5. Se lee una localidad de la memoria de instrucciones cuando WR_INS_A ='0'.
6. La *Forward Unit* genera CMUX2A y CMUX2B; si determina que existe dependencia de datos con la instrucción que entró al *pipeline* en el ciclo anterior, puede habilitar NOP y cambiar el valor de estas líneas de selección.
7. Se incrementa el contador actual (para leer la siguiente instrucción) cuando INCREMENT_A ='1'.
8. El Registro 1 lee las señales y las deja pasar a la siguiente etapa (como señales con terminación "**_A**"). Las entradas son: el bus de direcciones, el bus de instrucción, NOP, las líneas de selección de los multiplexores 2A, 2B, 5 y 6 (CMUX2A[2:0], CMUX2B[2:0], CMUX5, CMUX6). También lee CLEAR_C, y si el COMPARADOR la habilitó, a las señales de salida se les asignan '0's.

Las señales de control que intervienen en esta etapa del *pipeline* son las siguientes:

- **unidad de conteo**

CLK1	- En cada flanco de subida permite la salida de la dirección generada por el contador de subrutina activo. Y si la instrucción actual es de tipo RAMIFICACIÓN también se lee una dirección que modificará la secuencia de ejecución.
RESET	- Se activa cuando la señal externa INICIO ='1'. Asigna el contador 0 y la dirección 0 como inicio del conteo, es decir, el contador activo es el 0, y éste genera la dirección 0.
INCREMENT_A	- Incrementa el conteo.
ENABLE_A	- Habilita el conteo.
LI_B	- Llama a un dato (DALU_B[15:0]) para modificar la secuencia de conteo.
PD_B	- Regresa a un contador anterior.
PU_B	- Avanza al siguiente contador.

- **memoria de instrucciones**

SET	- Se habilita cuando la señal externa INICIO ='1'. Entonces se inicializa la memoria, es decir, se escribe cada una de las instrucciones del arreglo de memoria.
WR_INS_A	- Si su valor es '0', indica lectura de una instrucción.

- **unidad de dependencia de datos**

CLK2	- Cuando tiene un flanco de subida, examina si se cumplen las condiciones para una dependencia de datos (al menos uno de los registros fuente coincide con el registro destino de alguna instrucción anterior, que todavía está en el *pipeline*).
NOP	- Esta unidad genera la señal de control NOP. Cuando existen las condiciones de dependencia de datos con la instrucción anterior a la actual, esta señal se activa. En caso contrario permanece inactiva.

- **Registro 1 (Registro del *pipeline*)**

CLK1 - En cada flanco de subida de CLK1, los valores de todas las señales generadas en esta etapa se almacenan en Registro 1. Al mismo tiempo, las deja pasar a la etapa 2 (excepto cuando CLEAR_C ='1'). Cuando sucede esto, el nombre de la señal cambia, ya que se le agrega "_A" al final.

CLEAR_C - Cuando está habilitada todas los bits de salida son '0'.

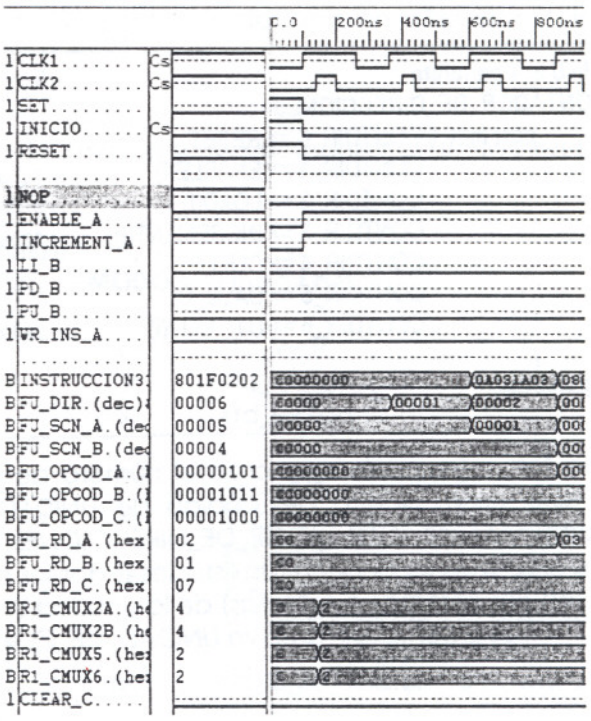


Figura 2.44 El Logic Simulator muestra las señales involucradas en la etapa de Lectura de la Instrucción

A continuación se detalla cada uno de los dispositivos involucrados en la etapa de Lectura de la Instrucción.

2.2.2.1 La unidad de conteo

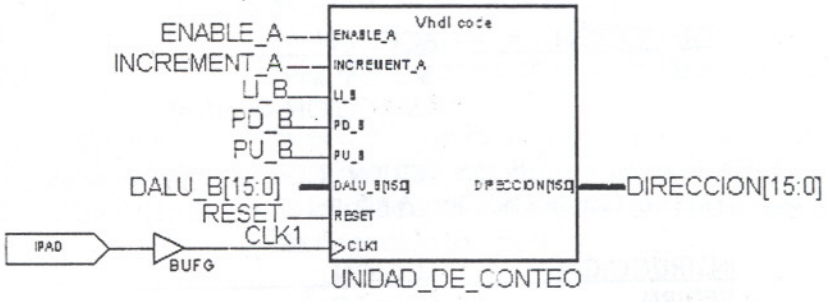


Figura 2.45 Esta entidad VHDL agrupa a 5 instancias del Contador de Direcciones

La unidad de conteo produce la sucesión de direcciones que se leerán en la memoria de instrucciones. La entidad está integrada por 5 contadores de subrutina, generados en VHDL como componentes. Al hablar de componentes nos referimos a dispositivos generados como elementos de la entidad, por medio de un *for... generate*. Sólo se programa una vez el comportamiento del componente y en cada iteración del ciclo se crea (se instancia) una copia idéntica. Esta función VHDL nos fue útil al programar el conjunto de contadores de direcciones y el conjunto de registros de propósito general. Cada uno de los componentes se instancia [17] al mapear sus puertos con las señales internas o externas de la entidad a la que pertenece:

```
for i in 0 to 4 generate                                --5 contadores
  CONTADOR_DE_DIRECCIONES
        port map ( CONTDIR_0 => RESET,
                   CONTDIR_1 => ENABLE(i),              -- señal interna
                   CONTDIR_2 => LOAD(i),                -- señal interna
                   CONTDIR_3 => INCREMENT(i),           -- señal interna
                   CONTDIR_4 => CLK1,
                   CONTDIR_5 => DALU_B,
                   CONTDIR_6 => DIR_OUT(i)              -- señal interna
        );
end generate;
```

Figura 2.46 Ciclo *for ... generate* que crea el número de instancias indicado por el subíndice *i*; en este caso, dentro del código de UNIDAD_DE_CONTEO se crean 5 instancias del componente CONTADOR_DE_DIRECCIONES. En el mapeo podemos observar que a la izquierda se encuentran listados todos los puertos del componente, y a la derecha las señales (internas o externas) de la entidad UNIDAD_DE_CONTEO.
Segmento del archivo UNIDAD_DE_CONTEO.HDL (Ver Apéndice B)

La entidad unidad de conteo se rige por el CLK1. En cada flanco de subida del CLK1 se genera una nueva dirección, y si es el caso, también se lee una dirección. Cuando RESET ='1' se elige al contador 0. Posteriormente, con la señal PU_B ='1' se puede cambiar al siguiente contador, empezando el conteo en la dirección indicada por DALU_B[15:0] (generada en la Etapa de Ejecución). Con PD_B ='1' se regresa al contador anterior, y a la dirección en que éste se encontraba.

Cuando la unidad de control decodifica una instrucción CALL, habilita las señales CA_A y PU_A. Esta última indica un cambio del contador de subrutina actual (SC_n) al siguiente contador de subrutina (SC_{n+1}). El nuevo contador comenzará a generar direcciones a partir del offset de 16 bits DALU_B[15:0].

INSTRUCCIÓN	ACCIÓN
CALL	$SC_n = SC_{n+1}$
	DIRECCION <= offset

El complemento de CALL es la instrucción RETURN. Con ésta se regresa al contador de subrutina anterior, SC_{n+1}, pues el Control habilitará PD_B:

INSTRUCCIÓN	ACCIÓN	
RETURN	$SC_n = SC_{n-1}$	para $n \geq 1$
	$SC_n = SC_n$	para $n = 0$

Cada uno de los contadores está sincronizado con el CLK1, cuando éste tiene un flanco de subida sucede lo siguiente:

- Si la señal RESET ='1', el contador activo genera la dirección 0.
- Cuando la señal ENABLE_A se convierte en ENABLE(*i*) del contador *i*, dicho contador está habilitado y permite la salida de la dirección actual.
- Cuando la señal INCREMENT_A se convierte en INCREMENT(*i*) del contador *i*, la dirección actual se incrementa en 1 (permitiendo así leer la siguiente instrucción).
- En el caso de que LI_B ='1', y por lo tanto también LOAD(*i*) ='1', la nueva dirección se lee de DALU_B[15:0].

A continuación se presentan la entidad y sus componentes, junto con la descripción de sus señales.

Entidad unidad de conteo

Nombre de la señal	MODO	Tipo de dato	ORIGEN
CLK_1	in	std_logic	- U. Control
ENABLE_A	in	std_logic	- U. Control
INCREMENT_A	in	std_logic	- OR1 del comparador
LI_B	in	std_logic	- Registro 2
PD_B	in	std_logic	- Registro 2
PU_B	in	std_logic	- U. Control
RESET	in	std_logic	- Mux. 3
DALU_B[15:0]	in	std_logic_vector (15 downto 0)	**DESTINO**
DIRECCIÓN[15:0]	out	std_logic_vector (15 downto 0)	- Mem. de instrucciones, *Forward Unit*, Registro 1

Nombre de la señal	Descripción
CLK_1	- Reloj 1, que controla a la unidad de conteo y a sus contadores
ENABLE_A	- Habilita el conteo
INCREMENT_A	- Incrementa el conteo
LI_B	- Llama a un dato (DALU_B[15:0]) para modificar la secuencia de conteo
PD_B	- Regresa al contador anterior
PU_B	- Avanza al siguiente contador
RESET	- Asigna el contador 0 y la dirección 0 como inicio del conteo
DALU_B[15:0]	- Es el dato que modificará la secuencia de conteo, es el bus de dirección (*input*)
DIRECCION[15:0]	- Es la dirección generada por el contador actual, es el bus de dirección (*output*)

Componente contador de direcciones

Nombre del puerto	MODO	Tipo de dato
CONTDIR_0	in	std_logic
CONTDIR_1	in	std_logic
CONTDIR_2	in	std_logic
CONTDIR_3	in	std_logic
CONTDIR_4	in	std_logic
CONTDIR_5[15:0]	in	std_logic_vector (15 downto 0)
CONTDIR_6[15:0]	inout	std_logic_vector (15 downto 0)

Nombre del puerto	MAPEADO A	
CONTDIR_0	- RESET	(señal externa)
CONTDIR_1	- ENABLE(i)	(señal interna, copia *i* de la señal externa ENABLE _A)
CONTDIR_2	- LOAD(i)	(copia *i* de la señal LI_B)
CONTDIR_3	- INCREMENT(i)	(copia *i* de la señal INCREMENT_A)
CONTDIR_4	- CLK_1	(señal externa)
CONTDIR_5[15:0]	- DALU_B[15:0]	(señal externa)
CONTDIR_6[15:0]	- DIR_OUT(i)	(señal interna, una copia de ella va a la DIRECCION que genera la Unidad de Conteo; también retroalimenta al contador *i*)

2.2.2.2 Memoria de instrucciones

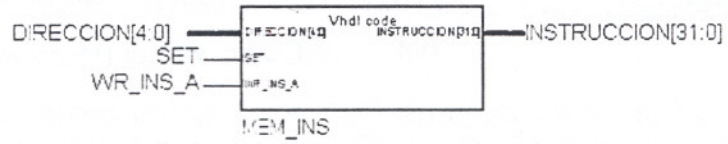


Figura 2.47 Memoria de instrucciones de 32 bits, con direcciones de 16 bits

Esta ROM asíncrona contiene el programa, es decir, las instrucciones que se ejecutarán durante toda la actividad del microprocesador. Recibe las direcciones generadas por la unidad de conteo.
Cuando la señal SET ='1', todas las instrucciones se almacenan en memoria. La señal WR_INS_A determina cuándo se realizará una lectura pues cuando está en 0, la señal DIRECCION[4:0] contiene la localidad de memoria que se leerá; y el contenido de esta localidad pasará al bus de instrucción de 32 bits. Ya que la memoria de instrucciones es un arreglo de palabras, para acceder a una de sus localidades se necesita un número de tipo entero. Por lo tanto, debe hacerse la conversión de la señal DIRECCION[4:0], de std_logic_vector de 5 bits a un entero.

Inicialmente esta memoria tenía un bus de direcciones de 16 bits, ya que la unidad de conteo genera una dirección de 16 bits. Se redujo a solamente 5 bits por dos razones: la capacidad del FPGA elegido, además de que el tiempo que requiere sintetizar e implementar una memoria crece si el espacio de direccionamiento es mayor. Se programaron 3 dispositivos de memoria iguales, con diferentes instrucciones, para realizar pruebas. Una de ellas (MEM_INS) contiene instrucciones de tipo OPERACIÓN y LOAD. La segunda (MEM_INS2) incluye instrucciones de tipo OPERACIÓN y de salto incondicional. La tercera memoria (MEM_INS3) contiene además instrucciones de salto condicional.

A continuación se detalla cada señal de la memoria de instrucciones.

Nombre de la señal	MODO	Tipo de dato	ORIGEN
DIRECCION[4:0]	in	std_logic_vector (4 downto 0)	- U. Conteo
WR_INS_A	in	std_logic	- U. Control
SET	in	std_logic	- U. Control
			DESTINO
INSTRUCCION[31:0]	out	std_logic_vector (31 downto 0)	- Registro 1

Nombre de la señal	Descripción
DIRECCION[4:0]	- Es la dirección de la memoria de instrucciones que se leerá.
WR_INS_A	
SET	- Cuando = '0', indica lectura. - Cuando = '1', escribe cada una de las instrucciones en la localidad que le corresponde.
INSTRUCCION[31:0]	- Es el bus de instrucciones (output).

2.2.2.3 Unidad de Dependencia de Datos (*Forward Unit*)

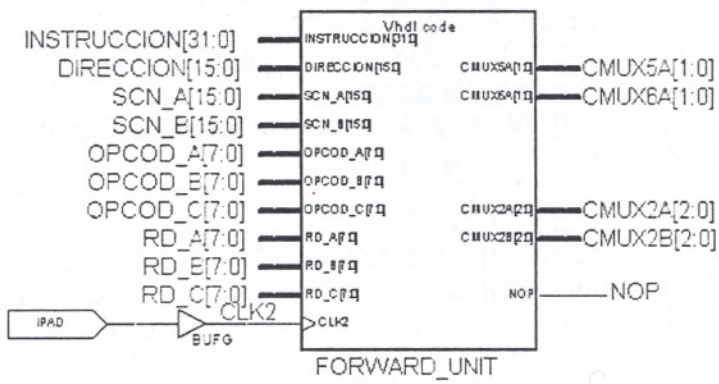


Figura 2.48 Unidad de Dependencia de Datos (*Forward Unit*)

Cuando una instrucción llega a la Etapa de Ejecución es posible que los operandos que necesita aún no estén disponibles en el registro correspondiente. Entonces ocurre una dependencia de datos. La *Forward Unit* es la pieza fundamental para resolverla evitando los retrasos en el *pipeline*, ya que se anticipa a esta situación, identificándola desde la etapa de Lectura de la Instrucción.

Este elemento se sincroniza con CLK2, ya que su flanco de subida ocurre después del de CLK1, para dar tiempo a que el bus de instrucciones ya tenga un valor estable. Cuando la unidad de conteo genera una dirección para leer la memoria de instrucciones con CLK1, en el siguiente flanco de subida de CLK2 pasan a la *Forward Unit* esta dirección y la instrucción correspondiente. *Forward Unit* verifica si los dos ciclos de reloj anteriores tienen direcciones diferentes a la actual. Para ello lee las direcciones SCN_A[15:0] y SCN_B[15:0], (que provienen del Registro 1 y del Registro 2, respectivamente). Si fueran iguales a la dirección actual, el ciclo de reloj actual no podría dar lugar a una dependencia de datos.

Además es preciso identificar de qué tipo son las instrucciones que se encuentran en cada una de las etapas del *pipeline*. De la palabra de instrucción se toman los primeros 8 bits, correspondientes al código de operación actual. También se leen OPCOD_A[7:0], OPCOD_B[7:0] y OPCOD_C[7:0], códigos de las 3 instrucciones anteriores, que ahora se encuentran en las etapas de Decodificación, Ejecución y Acceso a memoria, respectivamente. Para la instrucción actual nos interesa especialmente saber si es de tipo STORE (ya que tiene 3 operandos: RS1, RS2 y RD) o RAMIFICACIÓN (con 4 operandos; de ellos RS1 y RD se leen del Bloque de Registros.)

Si alguna de las otras 3 instrucciones en las siguientes etapas del *pipeline* es de tipo OPERACIÓN o LOAD, es posible que ocasionen una dependencia de datos, pues la etapa de Escritura de Resultados en un registro es utilizada por estas instrucciones. Se compara la dirección del registro destino de las 3 instrucciones anteriores (RD_A[7:0], RD_B[7:0] y RD_C[7:0]) con las direcciones de los registros fuente de la instrucción actual. Si alguno de ellos coincide, entonces ocurre una dependencia de datos y la línea de selección de al menos un multiplexor (MUX2A, 2B para la ALU; MUX5, 6 para el comparador) será modificada. A continuación enlistamos las comparaciones realizadas en el código de la *Forward Unit*.

Cuando la instrucción *i-1* es de tipo OPERACIÓN:

- Si el registro destino de la instrucción *i-1* (RD_A[7:0]) es igual al primer registro fuente actual (R1S[7:0]), entonces NOP= '1' y CMUX2A[2:0]= "100".
- Si el registro destino de la instrucción *i-1* es igual al segundo registro fuente actual (R2S[7:0]), entonces NOP= '1' y CMUX2B[2:0]= "100".
- Si RD_A[7:0] = RD[7:0] y la instrucción actual es de tipo STORE o RAMIFICACIÓN, entonces NOP= '1' y CMUX5[1:0]= "01".
- Si RD_A[7:0] = R1S[7:0] y la instrucción actual es de tipo RAMIFICACIÓN, entonces NOP= '1' y CMUX6[1:0]= "01".

Cuando la instrucción *i-1* es de tipo LOAD:

- Si RD_A[7:0] es igual a R1S[7:0], entonces NOP= '1' y CMUX2A[2:0]= "011".
- Si RD_A[7:0] es igual a R2S[7:0], entonces NOP= '1' y CMUX2B[2:0]= "011".
- Si RD_A[7:0] es igual a RD[7:0] y la instrucción actual es de tipo STORE o RAMIFICACIÓN, entonces NOP= '1' y CMUX5[1:0]= "11".
- Si RD_A[7:0] es igual a R1S[7:0] y la instrucción actual es de tipo RAMIFICACIÓN, entonces NOP= '1' y CMUX6[1:0]= "11".

Cuando la instrucción *i-2* es de tipo OPERACIÓN:

- Si RD_B[7:0] = R1S[7:0], entonces CMUX2A[2:0]= "100".
- Si RD_B[7:0] = R2S[7:0], entonces CMUX2B[2:0]= "100".
- Si RD_B[7:0] = RD[7:0] y la instrucción actual es de tipo STORE o RAMIFICACIÓN, entonces CMUX5[1:0]= "01".
- Si RD_B[7:0] = R1S[7:0] y la instrucción actual es de tipo RAMIFICACIÓN, entonces NOP= '1' y CMUX6[1:0]= "01".

Cuando la instrucción *i-2* es de tipo LOAD:

- Si RD_B[7:0] = R1S[7:0], entonces CMUX2A[2:0]= "100".
- Si RD_B[7:0] = R2S[7:0], entonces CMUX2A[2:0]= "100".
- Si RD_B[7:0] = RD[7:0] y la instrucción actual es de tipo STORE o RAMIFICACIÓN, entonces CMUX5[1:0]= "11".
- Si RD_B[7:0] = R1S[7:0] y la instrucción actual es de tipo RAMIFICACIÓN, entonces NOP= '1' y CMUX6[1:0]= "11".

Cuando la instrucción i-3 es de tipo OPERACIÓN o LOAD:

- Si RD_C[7:0] = R1S[7:0], entonces CMUX2A[2:0]= "000".
- Si RD_C[7:0] = R2S[7:0], entonces CMUX2B[2:0]= "000".
- Si RD_C[7:0] = RD[7:0] y la instrucción actual es de tipo STORE o RAMIFICACIÓN, entonces CMUX5[1:0]= "00".
- Si RD_C[7:0] = R1S[7:0] y la instrucción actual es de tipo RAMIFICACIÓN, entonces NOP= '1' y CMUX6[1:0]= "00".

Cuando la dirección de un registro fuente resulta igual a RD_A[7:0], la dependencia es con la instrucción inmediata anterior y entonces se activa la señal NOP, con el fin de que los multiplexores 2A y 2B seleccionen señales cuyo valor está recién calculado. Cuando RD_B[7:0] o RD_C[7:0] ocasionan la dependencia no se activa NOP, y tampoco hay retrasos.

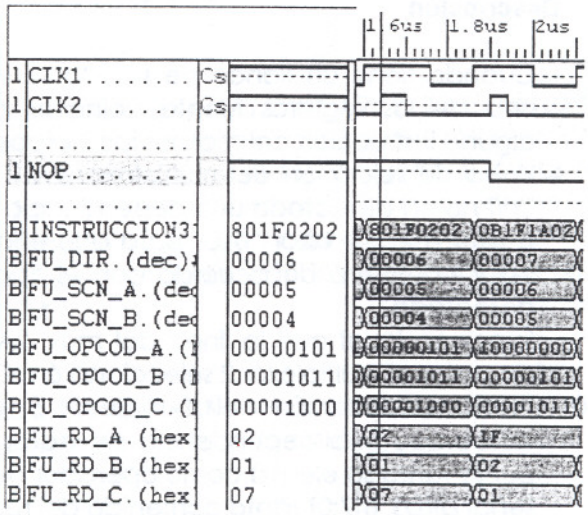


Figura 2.49 Comparación de direcciones de los registros fuente y destino

En la figura 2.23 podemos observar cómo se comparan las direcciones de los registros durante la simulación. En la primera columna se encuentran los nombres de las señales, algunos de los cuales sufrieron variaciones para presentarlos en la simulación. Así, en la primera el nombre FU_DIR corresponde a la señal DIRECCIÓN[15:0] que es leída por la *Forward Unit*, su valor se encuentra en las dos columnas de la derecha. El primer bus identifica a la señal INSTRUCCIÓN[31:0], en notación decimal. La dirección de su registro destino es 1F. Sus registros fuente son "02" y "02". La posición en memoria de la instrucción actual es DIR[7:0], y va seguida por las posiciones anteriores (SCN_A[15:0]) y (SCN_B[15:0]). En la figura coincide el valor de RD_A[7:0]= 02 con ambos registros fuente y por lo tanto NOP está activa.

Enseguida se especifica cada señal que entra o sale de la *Forward Unit*.

Nombre de señal	MODO	Tipo de dato	ORIGEN
CLK_2	in	std_logic	
DIRECCION[15:0]	in	std_logic_vector (15 downto 0)	- U. Conteo
INSTRUCCION[31:0]	in	std_logic_vector (31 downto 0)	- U. Conteo
OPCOD_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
OPCOD_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
OPCOD_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3

RD_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
RD_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
RD_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3
SCN_A[15:0]	in	std_logic_vector (15 downto 0)	- Registro 1
SCN_B[15:0]	in	std_logic_vector (15 downto 0)	- Registro 2
			DESTINO
CMUX2A[2:0]	out	std_logic_vector (2 downto 0)	- Registro1
CMUX2B[2:0]	out	std_logic_vector (2 downto 0)	- Registro1
CMUX5A[1:0]	out	std_logic_vector (1 downto 0)	- Registro1
CMUX6A[1:0]	out	std_logic_vector (1 downto 0)	- Registro1
NOP	out	std_logic	- Registro1

Nombre de la señal	**Descripción**
CLK_2	- Cuando tiene un flanco de subida se verifica si al menos uno de los registros fuente coincide con el registro de
CMUX2A[2:0]	alguna instrucción anterior, en las 3 etapas siguientes - Línea de selección del multiplexor 2A. Cuando la *Forward Unit* no ha detectado una dependencia de datos le asigna a esta línea el valor "010" para que elija a DR1_B[7:0]. Este valor cambiará de acuerdo con el tipo de dependencia
CMUX2B[2:0]	detectada.
CMUX5A[1:0]	- Funciona igual que la línea del mux. 2A; sin dependencia de datos el multiplexor 2 seleccionará a DR2_B[7:0].
CMUX6A[1:0]	- Línea de selección del multiplexor 5. Cuando no se ha detectado una dependencia de datos su valor es "10" y por lo tanto se elegirá como operador del comparador a la señal DRDS_B[7:0] (dato contenido del registro RD)
DIRECCION[15:0]	- Trabaja de forma similar a CMUX5A[1:0]. Cuando no se ha detectado una dependencia de datos su valor es "10" y el multiplexor 6 seleccionará la señal DR1_B[7:0]
INSTRUCCION[31:0]	- Localidad de memoria de la cual se leyó la instrucción actual
NOP	- Palabra de 4 bytes, que contiene el código de operación, y la dirección de 3 registros - Cuando está activa, significa que se encontró una dependencia de datos ocasionada por la instrucción
OPCOD_A [7:0]	anterior a la actual - Código de operación de la instrucción que se encuentra en
OPCOD_B[7:0]	la siguiente etapa (Decodificación) - Código de operación de la instrucción que se encuentra en
OPCOD_C[7:0]	la etapa 3 (Ejecución) - Código de operación de la instrucción que se encuentra en
RD_A[7:0]	la etapa 4 (Acceso a memoria)
RD_B[7:0]	- Dirección del registro destino de la instrucción en etapa 2
RD_C[7:0]	- Dirección del registro destino de la instrucción en etapa 3
SCN_A[15:0]	- Dirección del registro destino de la instrucción en etapa 4
SCN_B[15:0]	- Localidad de memoria de la instrucción en etapa 2 - Localidad de memoria de la instrucción en etapa 3

2.2.2.4 Registro 1 (Registro del *pipeline*)

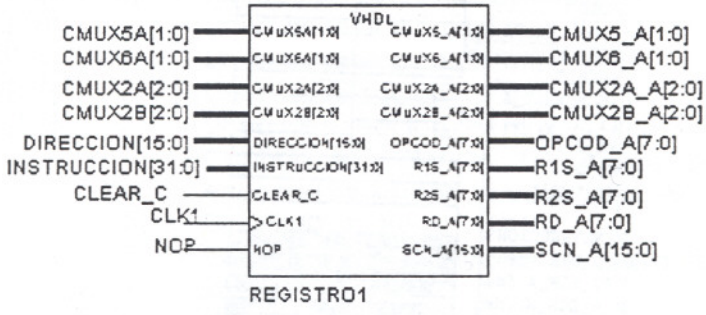


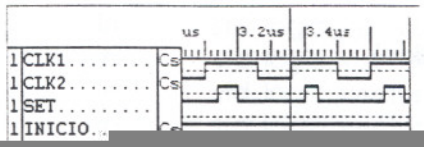
Figura 2.50 Registro 1 del *pipeline*

Los cuatro Registros que existen en la UAM - RISC II son pieza fundamental del *pipeline*. Son ellos los que imponen el sincronismo dentro del microprocesador. Existen elementos no síncronos (como las memorias y la unidad de control) cuyas señales de salida son leídas al mismo tiempo que las señales de elementos síncronos (como la unidad de conteo). Además, hemos visto (en la sección 1.1.4) que cada una de las etapas *pipeline* debe tener la misma duración. Esto se resuelve sincronizando los 4 registros del *pipeline* con la misma señal de reloj, en este caso CLK1.

De esta forma, cada vez que CLK1 tiene un flanco de subida, el Registro 1, al igual que los otros tres, lee las señales de datos y de control de una etapa, y las deja pasar a la siguiente etapa.

Cuando la *Forward Unit* detecta que la instrucción actual ocasiona una dependencia de datos, posiblemente también asigne NOP ='1'. También puede ocurrir que al mismo tiempo que se está leyendo la instrucción actual, en la etapa de Ejecución haya una instrucción de tipo RAMIFICACIÓN, y por lo tanto el comparador está habilitado. Cuando la comparación es falsa, COMP ='0' y si además no se está ejecutando una instrucción CALL o RETURN, entonces se obtendrá CLEAR_C= '0', lo que no altera al resto de las señales leídas por el Registro 1. Por el contrario, si la comparación resulta verdadera, la señal CLEAR_C se habilitaría y como resultado, la secuencia de instrucciones será alterada en el siguiente ciclo de reloj. Y por lo tanto las instrucciones que se encuentran en las primeras etapas ya no deben continuar en el *pipeline*. Cuando el Registro 1 lee CLEAR_C ='1' "borra" su contenido, asignando ceros a todos los bits de las señales de salida.

En la siguiente figura (2.26) se muestra un fragmento de la simulación en el que CLEAR_C se activa. Esto ocurre durante un ciclo de reloj. Y en el siguiente ciclo (señalado con la línea vertical) podemos ver sus efectos. En la figura se aprecia que ha asignado '0' a las señales de salida del Registro 1: SCN_A[15:0], OPCOD_A[7:0], RD_A[7:0]; del Registro 2: SCN_B[15:0], OPCOD_B[7:0], RD_B[7:0].


us
3.2us
3.4us
1 CLK1.....
1 CLK2.....
1 SET.....
1 INICIO..

Nombre de señal	Descripción
DIRECCION[15:0]	- Localidad de memoria de la instrucción actual (en la Etapa de Lectura)
INSTRUCCION[31:0]	- Instrucción actual, de 32 bits
CMUX2A[2:0]	- Línea de selección del mux. 2A
CMUX2B[2:0]	- Línea de selección del mux. 2B
CMUX5A[1:0]	- Línea de selección del mux. 5
CMUX6A[1:0]	- Línea de selección del mux. 6
NOP	- Cuando su valor es '1', indica que la dependencia encontrada es con la instrucción anterior (en la Etapa de Decodificación)
CLEAR_C	- Cuando está habilitada se borra el contenido de este Registro
CMUX2A_A[2:0]	- Línea de selección del mux. 2A
CMUX2B_A[2:0]	- Línea de selección del mux. 2B
CMUX5_A[1:0]	- Línea de selección del mux. 5
CMUX6_A[1:0]	- Línea de selección del mux. 6
NOP_A	- Indica que la dependencia es con la instr. anterior
OPCOD_A[7:0]	- Código de operación de la instrucción actual
R1S_A[7:0][7:0]	- Dirección de un registro o valor constante
R2S_A[7:0]	- Dirección de un registro o valor constante
RD_A[7:0]	- Dirección de un registro o valor constante
SCN_A[15:0]	- Dirección de la memoria de la instrucción actual

2.2.3 Etapa 2 Decodificación y Lectura de Operandos (*Decode & Operand Fetch*)

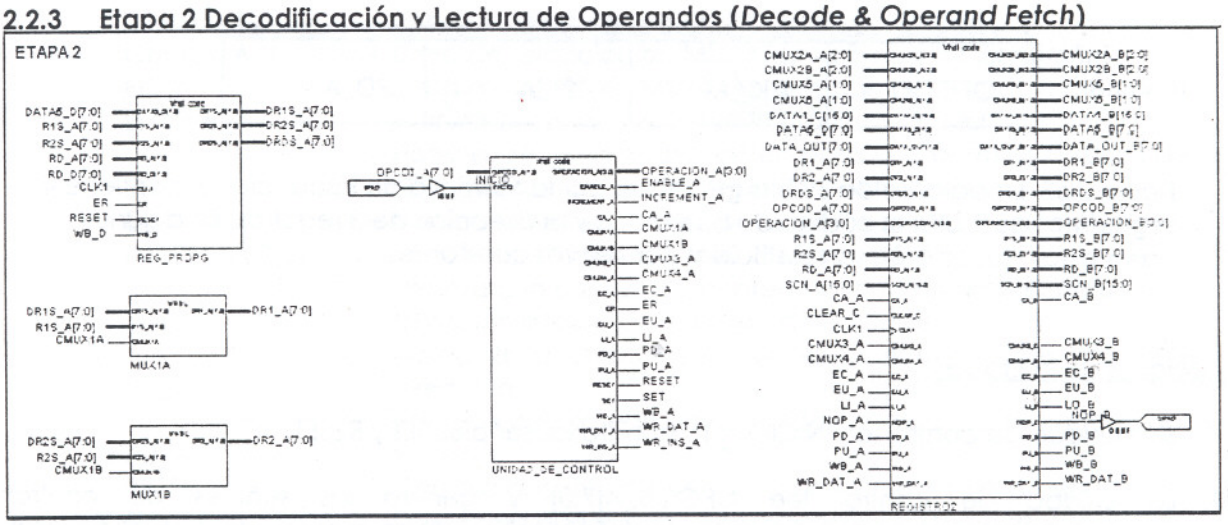


Figura 2.52 Elementos de la 2ª etapa del *pipeline*: Unidad de Control, Bloque de Registros de Propósito General, Multiplexor 1A, Multiplexor 1B, Registro 2.

Esta etapa decodifica la instrucción RISC de 32 bits, al señalar qué tipo de instrucción se ejecutará, además de generar bits de control que se distribuirán a través de todo el microprocesador. Es decir, desde esta etapa la unidad de control decide qué harán los otros dispositivos.

La etapa de Decodificación de la Instrucción y Lectura de Operandos emplea los siguientes elementos:

- unidad de control
- bloque de registros de propósito general
- multiplexor 1A
- multiplexor 1B
- Registro 2 (Registro del *pipeline*)

Cuando comienza la ejecución del *pipeline* se debe habilitar durante algunos *nseg.* la señal externa INICIO, pues de esta manera se habilitarán RESET (para la unidad de conteo) y SET (para la memoria de instrucciones).

Los primeros 8 bits de la instrucción leída de la memoria de instrucciones son decodificados por el Control. Al identificar al Código de Instrucción genera 18 señales de control para algunos multiplexores, así como para el comparador, unidad de conteo, memoria de instrucciones, memoria de datos y Bloque de Registros. Además determina qué operación efectuará la ALU.

Los datos contenidos en los siguientes 3 bloques de 8 bits son localidades del bloque de registros de propósito general. En algunos casos los registros fuente son sustituidos por un valor constante, por lo tanto los multiplexores de esta etapa tienen la tarea de elegir entre el valor leído o la constante. También llegan de la de Escritura de Resultados la dirección del registro destino y el dato que se escribirá en él para actualizarlo.

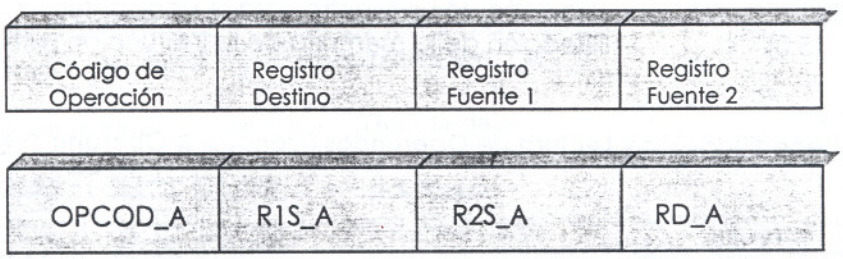


Figura 2.53 La palabra de 32 bits generada por la Memoria de Instrucciones contiene 4 segmentos de 8 bits: el código de operación y la dirección de 3 registros. En algunos casos estas direcciones son sustituidas por valores constantes.

Acciones de la Etapa 2

1. Si la unidad de control lee INICIO ='1', habilita las señales SET y RESET.

2. La unidad de control lee OPCOD_A[7:0] y genera las señales de control correspondientes. Además determina qué operación básica realizará la ALU, al generar la señal OPERACION_A[3:0].
3. Si RESET ='1' se le asignan 0's al contenido de todos los registros.

4. Si el bloque de registros de propósito general está habilitado para lectura (ER ='1'), se lee el registro indicado en R1S_A[7:0]; su contenido se vacía en el bus DR1S_A[7:0]. Lo mismo ocurre con R2S_A[7:0] y RD_A[7:0]; los datos leídos son DR2S_A[7:0] y DRDS_A[7:0], respectivamente.

5. Si CLK1='1' y si el Bloque de Registros está habilitado para escritura (WB_D ='1'), el contenido del bus DATA5_D[7:0] se escribe en el registro RD_D[7:0].

6. El multiplexor 1A recibe su línea de selección del Control. Con ella elige entre el dato leído DR1S_A[7:0] y el dato constante R1S_A[7:0]. Este valor pasa a DR1_A[7:0].

7. El multiplexor 1B elige entre el dato leído DR2S_A[7:0] y la constante R2S_A[7:0]. Este valor pasa a DR2_A[7:0].

8. Cuando CLK1 ='1', el Registro 2 almacena las señales (la mayoría de ellas **con terminación "_A"**) y las deja pasar a la siguiente etapa **(como señales con terminación "_B")**. Las señales de entrada son: el bus de dirección; las líneas de selección de los multiplexores 2A, 2B, 3, 4, 5 y 6; el bus que va a la memoria de datos; el bus que va a actualizar el Bloque de Registros; el bus que proviene de la memoria de datos; las direcciones de los registros fuente; el contenido del registro RD_A[7:0]; los datos seleccionados por los multiplexores 1A y 1B; el código de operación y la operación básica que le corresponde; además almacena varias señales de control. También lee CLEAR_C, y si se encuentra habilitada a todas las señales de salida de este Registro se les asignan '0's.

Las **señales de control** generadas y leídas en esta etapa de Decodificación y Lectura de Operandos son:

- **unidad de control**

CA_A	- Se habilita cuando el código de operación corresponde a CALL.
CMUX1A	- Línea de selección del multiplexor MUX1A (que también se encuentra en esta etapa)
CMUX1B	- Línea de selección de MUX1B (en esta etapa)
CMUX3_A	- Línea de selección de MUX3 (en la Etapa de Ejecución)
CMUX4_A	- Línea de selección MUX4 (en la Etapa de Acceso a memoria)
EC_A	- Habilita el comparador (en la Etapa de Ejecución)
ENABLE_A	- Habilita a la unidad de conteo para que permita la salida de la dirección actual (en la Etapa De Lectura)
ER	- Habilita la lectura en el Bloque de Registros (en esta etapa)
EU_A	- Habilita la ALU (en la Etapa de Ejecución)
INCREMENT_A	- Habilita a la unidad de conteo para incrementar una unidad la dirección actual (en la Etapa de Lectura)
LI_A	- Indica a la unidad de conteo que debe leer una nueva dirección
PD_A	- Provoca que se cambie al contador de subrutina actual por el siguiente
PU_A	- Cambia al contador de subrutina actual por el anterior
RESET	- Determina que el contador 0 genere la dirección 0
SET	- Permite que las instrucciones sean escritas en la memoria de instrucciones (en la Etapa de Lectura)
WB_A	- Indica que habrá una escritura en el registro destino (en la etapa de Escritura de Resultados)
WR_DAT_A	- Habilita la lectura o escritura de memoria de datos (en la Etapa de Acceso a memoria)
WR_INS_A	- Habilita la lectura de memoria de instrucciones (en la Etapa de Lectura)
	-

- **bloque de registros de propósito general**

CLK1	- Cuando tiene un flanco de subida permite la escritura
ER	- Cuando está habilitada permite la lectura
RESET	- Cuando está habilitada el contenido de todos los registros son 0's
WB_D	- Cuando está habilitada permite la escritura

- **multiplexor 1A**

CMUX1A	- Línea de selección

- **multiplexor 1B**

CMUX1B	- Línea de selección

- **Registro 2 (Registro del *pipeline*)**

CLK1	- En cada flanco de subida de CLK1, los valores de todas las señales generadas en esta etapa se almacenan en Registro 2 (excepto cuando CLEAR_C ='1'.) Al mismo tiempo, las deja pasar a la etapa 3. Cuando sucede esto, el **nombre de la señal** cambia, ya que su terminación ahora será **"_B"**
CLEAR_C	- Cuando está habilitada todos los bits de salida son 0
CA_A	- Se habilita cuando el código de operación corresponde a CALL
CMUX3_A	- Línea de selección de MUX3
CMUX4_A	- Línea de selección de MUX4
EC_A	- Habilita el comparador
EU_A	- Habilita la ALU
LI_A	- Cuando está habilitada la U. de Conteo lee una nueva dirección
NOP_A	- Si está habilitada indica que en la Etapa de Lectura se detectó una dependencia de datos con la instrucción anterior.
PD_A	- Cuando está habilitada la U. de Conteo disminuye el número de contador de subrutina
PU_A	- Cuando está habilitada la U. de Conteo incrementa el número de contador de subrutina
WB_A	- Habilita la escritura o lectura de un registro
WR_DAT_A	- Habilita la escritura o lectura de la memoria de datos

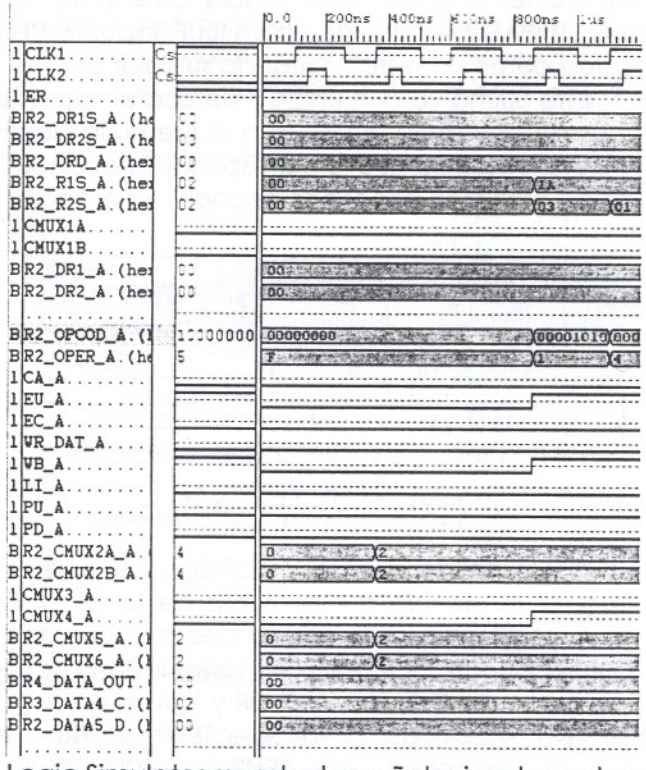


Figura 2.54 El Logic Simulator muestra las señales involucradas en la Etapa de Decodificación y Lectura de Operandos

A continuación se describen el diseño de cada uno de los elementos de esta etapa.

2.2.3.1 La unidad de control

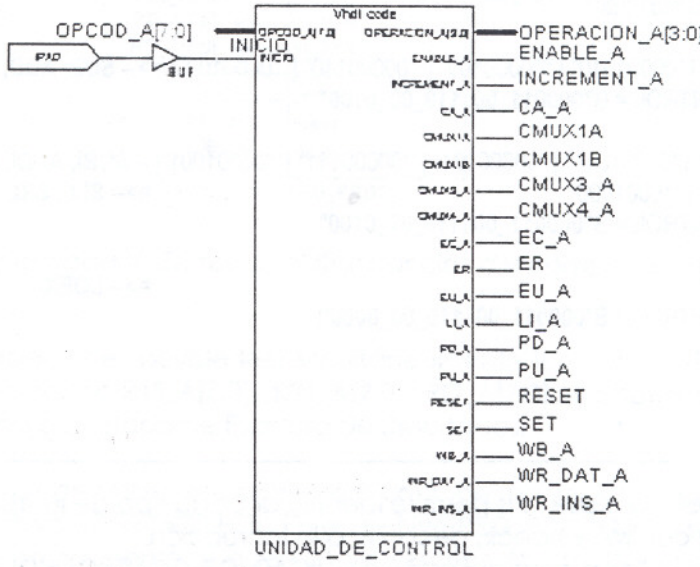


Figura 2.55 En la unidad de control se genera la mayoría de las señales de control, el resto proviene de la unidad de dependencia de datos

Este dispositivo asíncrono decide, junto con la *Forward Unit*, qué acciones deben realizarse en el resto del diseño del *pipeline*. Al leer el código de operación de la instrucción actual genera 18 señales de control. Además, permite inicializar los valores del microprocesador en el 1er ciclo de reloj.

La señal INICIO es la única señal externa al microprocesador. En el diagrama del *Schematic Editor* debe estar conectada a un IPAD (*input pad*) y a un IBUF (*input buffer*) para amplificar la señal. Esta señal externa actúa como un *switch*. Cuando su valor es '1' y el Control la lee, habilita las señales SET y RESET; ésta última va a la unidad de conteo para que el contador 0 empiece a generar direcciones a partir de la 0; SET va a la memoria de instrucciones para que cada palabra se escriba en la localidad que le corresponde; y cuando RESET también es leída por el bloque de registros de propósito general se asignan '0's a cada registro.

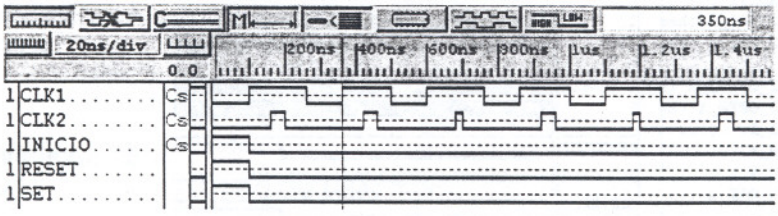


Figura 2.56 Señales INICIO, RESET y SET

En la siguiente figura vemos cómo se decodifica el código de operación utilizando el lenguaje VHDL. Para identificar el contenido de los 8 bits de la señal OPCOD_A[7:0], se emplea la sentencia case ... is. A cada caso se le asigna una palabra de control mediante when... => .... Esta palabra de control puede ser igual para varias instrucciones, o ser única. Por ejemplo, a los códigos de operación de **SUB, ADD, AND, OR** y **XOR** se les asigna la palabra de control[3] **B"000011_001110_00_0100"**, por medio de la variable TEMP_CONTROL. A los códigos de **SUBI, ADDI, ANDI, ORI, XORI, SLL** y **SRL** les corresponde TEMP_CONTROL:= **B"000011_001110_01_0100"**. Podemos observar que ambas palabras de control difieren sólo en el bit 4. Cuando este bit está en '1' lógico indica lectura de un valor constante. También hay palabras asignadas a una sola instrucción. Por ejemplo, el código TEMP_CONTROL:= **B"000011_001110_00_0000"** le corresponde únicamente a **LOEX** .

```
case OPCOD_A(7 downto 0) is

   when "00000010" | "00001010" | "00000100" | "00000110" | "00001000" =>-- SUB, ADD, AND, OR, XOR
        TEMP_CONTROL:= B"000011_001110_00_0100";

   when "00000011" | "00001011" | "00000101" | "00000111" | "00001001"| -- SUBI, ADDI, ANDI,ORI,XORI
        "10000111" | "10001000"                                          =>-- SLL, SRL
        TEMP_CONTROL:= B"000011_001110_01_0100";

   when "10000000"                                                       =>-- LOEX
        TEMP_CONTROL:= B"000011_001110_00_0000";
                        .
                        .
                        .
```

Figura 2.57 La sentencia *case ... is* permite identificar cada tipo de instrucción y generar señales de control, es decir, decodifica la instrucción.
Segmento del archivo UNIDAD_DE_CONTROL.HDL (Ver Apéndice)

[3] En las señales de tipo std_logic_vector es válido dividir el vector en segmentos por medio de un guión bajo (_). Si se utiliza al menos un guión se vuelve necesario agregar una B (de binario) antes del vector, con el fin de que no se confunda con una cadena de caracteres no numéricos.

Posteriormente la variable que contiene la palabra de control se divide en 18 bits, cada uno correspondiente a una señal externa de la unidad de control. Por ejemplo a la señal **ER** se le asigna el bit 8 de la variable **TEMP_CONTROL**, a **EU_A** el bit 7, a **EC_A** el bit 6, a **CMUX1A** el 5, etc.

```
ER      <= TEMP_CONTROL(8);
EU_A    <= TEMP_CONTROL(7);
EC_A    <= TEMP_CONTROL(6);
CMUX1A  <= TEMP_CONTROL(5);.
   .
   .
   .
```

Figura 2.58 Asignación de bits de la palabra de control a las señales generadas por la Unidad de Control.

Segmento del archivo UNIDAD_DE_CONTROL.HDL (Ver Apéndice B)

2.2.3.2 Bloque de registros de propósito general

Se trata de un conjunto de 32 registros de 8 bits cada uno. Ya que la dirección de un registro es de 8 bits, queda abierta la posibilidad de implementar 2^8 registros, en lugar de sólo 2^5. Es posible leer y escribir en estos registros. Cuando son leídos, su valor puede convertirse en un operando, y en una etapa posterior pueden almacenar el resultado de la instrucción.

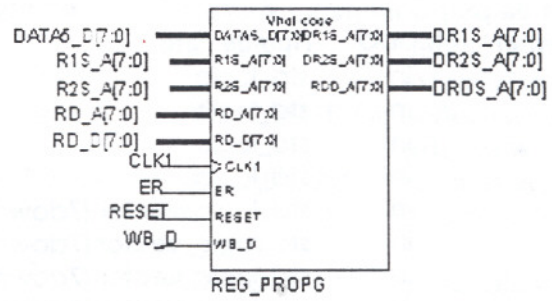


Figura 2.59 Esta entidad VHDL reúne a 32 instancias del componente Registro

La lectura es asíncrona, y es posible leer simultáneamente los tres registros contenidos en el formato de la instrucción: R1S_A[7:0], R2S_A[7:0], RD_A[7:0]. Por otro lado, la escritura es síncrona, y concierne a la etapa de Escritura de Resultados.

El código VHDL que describe el comportamiento de un registro (componente [17]) se encuentra en el archivo REGISTROS.HDL. En el archivo REG_PROPG.HDL se hace nuevamente la declaración del componente, y además se describe la arquitectura de la entidad que instancia múltiples copias de REGISTROS. En la siguiente figura se presenta el ciclo for que mapea cada puerto del componente con una señal interna o externa de la entidad. El uso de señales internas permite que una copia del dato que entra a la entidad REG_PROPG llegue a cada componente; de la misma manera, es factible habilitar simultáneamente para lectura o escritura varios registros, ya que cada uno cuenta con sus propias líneas, independientes del resto.

Para poder sintetizar sin errores la entidad es necesario: primero salvar y sintetizar el archivo .HDL que contiene la descripción del componente (REGISTROS.HDL); en segundo lugar salvar y sintetizar el archivo con la arquitectura de la entidad que instancia el componente (REG_PROPG.HDL). De lo contrario, no se enlazarán los componentes con la entidad, y por lo tanto su funcionamiento no será el programado.

```
INSTANCIAR_BLOQUE: for i in 0 to 31 generate        -- 32 registros
INSTANCIA: REGISTROS
    port map (REGISTROS_1 => IN_REG,                -- señal interna
              REGISTROS_2 => WRITE_ENABLE(i), -- señal interna
              REGISTROS_3 => READ_ENABLE(i),  -- señal interna
              REGISTROS_4 => CLK1,
              REGISTROS_5 => OUT_REG(i)         -- señal interna
              );
end generate;
```

Figura 2.60 En el mapeo, a la izquierda se listan los puertos del componente, y a la derecha las señales (internas o externas) de la entidad REG_PROPG. Las cadenas INSTANCIAR_BLOQUE e INSTANCIA son etiquetas.

Segmento del archivo REG_PROPG.HDL (Ver Apéndice)

Para explicar mejor la forma en que interactúan la entidad y sus componentes es necesario detallar cada una de las señales.

Entidad bloque de registros de propósito general

Nombre de la señal	MODO	Tipo de dato	ORIGEN
CLK_1	in	std_logic	
ER	in	std_logic	- U. Control
RESET	in	std_logic	- U. Control
WB_D	in	std_logic	- U. Control
DATA5_D[7:0]	in	std_logic_vector (7downto 0)	- Mux. 4
R1S_A[7:0]	in	std_logic_vector (7downto 0)	- Registro 1
R2S_A[7:0]	in	std_logic_vector (7downto 0)	- Registro 1
RD_A[7:0]	in	std_logic_vector (7downto 0)	- Registro 1
DR1S_A[7:0]			**DESTINO**
DR2S_A[7:0]	out	std_logic_vector (7downto 0)	- Mux. 1A
DRDS_A[7:0]	out	std_logic_vector (7downto 0)	- Mux. 1B
	out	std_logic_vector (7downto 0)	- Registro 2

Nombre de la señal	Descripción
CLK_1	- Reloj 1, que controla a los 32 registros (componentes)
ER	- Cuando está habilitada, habilita la lectura de cualquiera de los 32 registros. Una copia de ER pasa a la señal interna READ_ENABLE(i), y de ahí al registro *i*.
RESET	- Cuando su valor es '1' habilita la señal WRITE_ENABLE(i) y asigna a la señal IN_REG un vector de 0's.
WB_D	- Cuando es '1' habilita el registro *i* para escritura. Una copia de su valor se propaga a la señal WRITE_ENABLE(*i*).
	- Es el resultado de la instrucción que se encuentra en la etapa

DATA5_D[7:0]	de Escritura de Resultados. Su valor se asigna a la señal interna IN_REG.
R1S_A[7:0]	- Dirección de un registro a leer. (Para acceder a un registro es necesario convertir la señal de tipo std_logic_vector a integer)
R2S_A[7:0]	- Dirección de un registro a leer; en algunas instrucciones es un valor constante.
RD_A[7:0]	- En las instrucciones de tipo OPERACIÓN y LOAD se trata de la dirección del registro destino. En el tipo RAMIFICACIÓN es la dirección de un operando del comparador. Para las instrucciones STORE es la dirección del registro que debe leerse (o el dato directamente) para luego almacenarse en la memoria de datos
DR1S_A[7:0]	- Dato leído del registro R1S_A[7:0]
DR2S_A[7:0]	- Dato leído del registro R2S_A[7:0]
DRDS_A[7:0]	- Dato leído del registro RD_A[7:0]

Componente Registro

Nombre del puerto	MODO	Tipo de dato
REGISTROS_1	in	std_logic_vector (7 downto 0)
REGISTROS_2	in	std_logic
REGISTROS_3	in	std_logic
REGISTROS_4	in	std_logic
REGISTROS_5	out	std_logic_vector (7 downto 0)

Nombre del puerto	MAPEADO A	
REGISTROS_1	- IN_REG	(señal interna, copia de la señal externa DATA5_D[7:0])
REGISTROS_2	- WRITE_ENABLE(i)	(señal interna, copia *i* de la señal externa WB_D)
REGISTROS_3	- READ_ENABLE(i)	(señal interna, copia *i* de la señal externa ER)
REGISTROS_4	- CLK_1	
REGISTROS_5	- DIR_OUT(i)	(señal interna, una copia de ella se propaga al bus DR1S_A[7:0], DR2S_A[7:0] o al DRDS_A[7:0], según corresponda)

2.2.3.3 Multiplexores 1A y multiplexor 1B

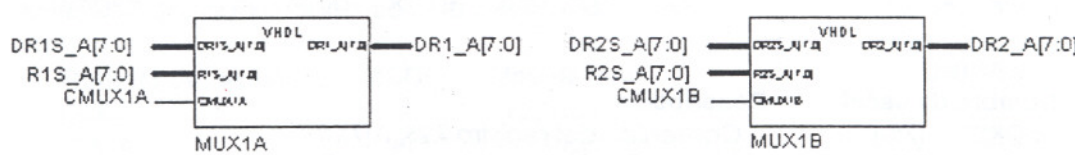


Figura 2.61 Multiplexores de la Etapa 2

Ambos multiplexores tienen la función de elegir entre un dato leído del bloque de registros de propósito general y un dato constante, incluido en la instrucción. Como ya hemos visto, el formato de la instrucción es de 32 bits, dividido en 4 secciones de 8 bits cada una. De estas

secciones 3 pueden contienen direcciones de registros o valores constantes. Independientemente del tipo de instrucción, cuando las palabras leídas (DR1S_A[7:0] y DR2S_A[7:0]) pasan al multiplexor se decide si se utilizarán o no. Esto se logra mediante una sentencia *case ... is*.

```
case CMUX1A is

   when '0' =>   DR1_A[7:0] <=  R1S_A;        -- la instrucción requiere un valor constante
   when '1' =>   DR1_A[7:0] <= DR1S_A;        -- la instrucción requiere el contenido de un registro

end case;
```

Figura 2.62 Sentencia case ... *is* del multiplexor 1A

Segmento del archivo MUX1A.HDL (Ver Apéndice)

Veamos las características de cada una de sus señales.

multiplexor 1A

Nombre de señal	**MODO**	**Tipo de dato**	**ORIGEN**
DR1S_A[7:0]	in	std_logic_vector (7 downto 0)	- B.de Registros
R1S_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
CMUX1A	in	std_logic	- U. Control
			DESTINO
DR1_A[7:0]	out	std_logic_vector (7 downto 0)	- Registro 2

Nombre de señal	**Descripción**
DR1S_A[7:0]	- Contenido del registro R1S_A[7:0]
R1S_A[7:0]	- Dirección de un registro, especificada en la instrucción
CMUX1A	- Cuando = 0, elige a DR1S_A[7:0] Cuando = 1, elige a R1S_A[7:0]
DR1_A[7:0]	- Contiene al dato elegido por el multiplexor 1ª

multiplexor 1B

Nombre de señal	**MODO**	**Tipo de dato**	**ORIGEN**
DR2S_A[7:0]	in	std_logic_vector (7 downto 0)	- B.de Registros
R2S_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
CMUX2A[2:0]	in	std_logic	- U. Control
			DESTINO
DR2_A[7:0]	out	std_logic_vector (7 downto 0)	- Registro 2

Nombre de señal	**Descripción**
DR2S_A[7:0]	- Contenido del registro R2S_A[7:0]
R2S_A[7:0]	- Dirección de un registro, especificada en la instrucción
CMUX2A[2:0]	- Cuando = 0, elige a DR2S_A[7:0] Cuando = 1, elige a R2S_A[7:0]
DR2_A[7:0]	- Contiene al dato elegido por el multiplexor 1B

2.2.3.4 Registro 2 (Registro del *pipeline*)

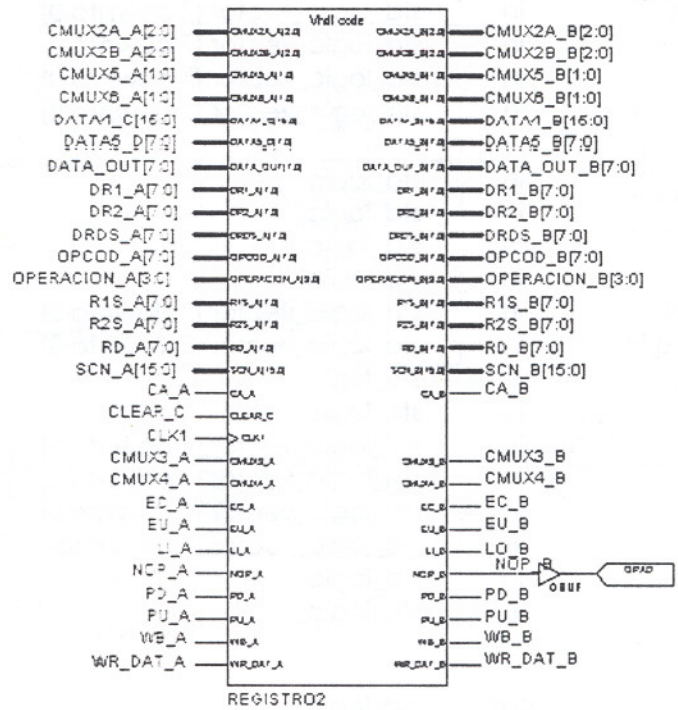


Figura 2.63 Registro 2 del *pipeline*

Este registro es el más grande del *pipeline*, es decir almacena mayor cantidad de bits. Es el paso intermedio **entre las etapas 2 y 3**, y por lo tanto recopila la mayoría de las señales de control, así como el bus de direcciones, las direcciones de registros, y el dato leído de ellos, el código de operación y su operación. Casi todas estas señales tienen terminación "**_A**", ya que provienen de la Etapa de Decodificación y Lectura, aunque también las hay con terminaciones "_C" y "_D". Al salir de este registro, las señales tienen el sufijo "**_B**". También lee la señal CLEAR_C, reacciona igual que el Registro 1. Cuando la *Forward Unit* ha detectado una dependencia, es este Registro el que recibe los datos de la etapa de Acceso a memoria o de Escritura de Resultados (DATA4_C, DATA_OUT[7:0], DATA5_D[7:0]) para posteriormente elegir el más adecuado. En realidad en cada ciclo de reloj estas líneas llegan al Registro 2, pero es sólo en caso de dependencia cuando una de ellas requiere ser utilizada. Y es la *Forward Unit* quien decide qué línea será seleccionada por los multiplexores 2A y 2B.

Las señales de entrada y salida de este registro son:

Nombre de señal	MODO	Tipo de dato	ORIGEN
CLK_1	in	std_logic	
CA_A	in	std_logic	- U. Control
CLEAR_C	in	std_logic	- OR1 del comparador
CMUX2A_A[2:0]	in	std_logic_vector (2 downto 0)	- Forward U.
CMUX2B_A[2:0]	in	std_logic_vector (2 downto 0)	- Forward U.
CMUX3_A	in	std_logic	- U. Control
CMUX4_A	in	std_logic	- U. Control

CMUX5_A[1:0]	in	std_logic_vector (1 downto 0)	- Forward U.
CMUX6_A[1:0]	in	std_logic_vector (1 downto 0)	- Forward U.
DATA4_C[15:0]	in	std_logic_vector (15 downto0)	- Registro 3
DATA5_D[7:0]	in	std_logic_vector (7 downto 0)	- Mux. 4
DATA_OUT[7:0]	in	std_logic_vector (7 downto 0)	- Mem. Instr.
DR1_A[7:0]	in	std_logic_vector (7 downto 0)	- Mux. 1A
DR2_A[7:0]	in	std_logic_vector (7 downto 0)	- Mux. 1B
DRSD_A[7:0]	in	std_logic_vector (7 downto 0)	- B. de Registros
EC_A	in	std_logic	- U. Control
EU_A	in	std_logic	- U. Control
LI_A	in	std_logic	- U. Control
NOP	in	std_logic	- Forward U.
OPCOD_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
OPERACION_A[3:0]	in	std_logic_vector (3 downto 0)	- U. Control
PD_A	in	std_logic	- U. Control
PU_A	in	std_logic	- U. Control
R1S_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
R2S_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
RD_A[7:0]	in	std_logic_vector (7 downto 0)	- Registro 1
SCN_A[15:0]	in	std_logic_vector (15 downto0)	- Registro 1
WB_A	in	std_logic	- U. Control
WR_DAT _A	in	std_logic	- U. Control
			DESTINO
CA_B	out	std_logic	- OR1 del comp.
CMUX2A_B[2:0]	out	std_logic_vector (2 downto 0)	- Mux. 2A
CMUX2B_B[2:0]	out	std_logic_vector (2 downto 0)	- Mux. 2B
CMUX3_B	out	std_logic	- Mux. 3
CMUX4_B	out	std_logic	- Registro 3
CMUX5_B[1:0]	out	std_logic_vector (1 downto 0)	- Mux. 5
CMUX6_B[1:0]	out	std_logic_vector (1 downto 0)	- Mux. 6
DATA4_B[15:0]	out	std_logic_vector (15 downto0)	- Mux. 2A, mux.2B
DATA5_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 2A, mux.2B
DATA_OUT_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 2A, mux.2B
DR1_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 2A
DR2_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 2B
DRDS_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 5
EC_B	out	std_logic	- comparador
EU_B	out	std_logic	- ALU
LO_B	out	std_logic	- OR2 del comp.
OPCOD_B[7:0]	out	std_logic_vector (7 downto 0)	- comparador, *Forward Unit*
OPERACION_B[3:0]	out	std_logic_vector (3 downto 0)	- ALU
PD_B	out	std_logic	- U. de Conteo, OR1 del Comp.
PU_B	out	std_logic	- U. de Conteo
R1S_B	out	std_logic_vector (7 downto 0)	- Mux. 3
R2S_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 3
RD_B[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 3
SCN_B[15:0]	out	std_logic_vector (15 downto0)	- mux.2A, *Forward Unit*
WB_B	out	std_logic	- Registro 3
WR_DAT _B	out	std_logic	- Registro 3

2.2.4 Etapa 3 Ejecución y Comparación *(Execute)*

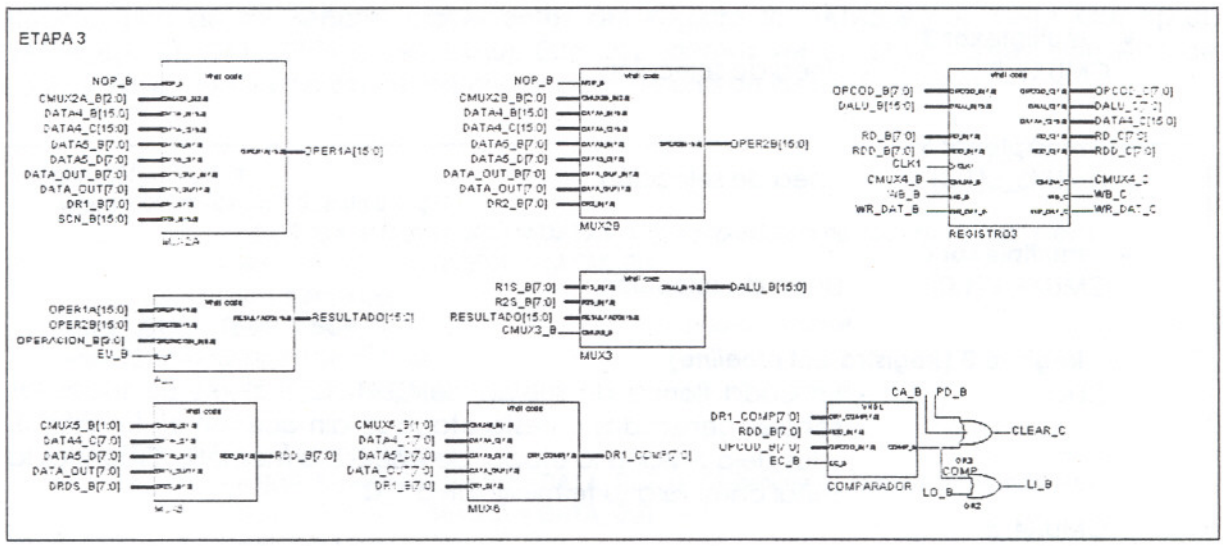


Figura 2.64 Etapa de Ejecución

La etapa de Ejecución necesita los siguientes elementos:

- multiplexor 2A
- multiplexor 2B
- ALU
- multiplexor 3
- multiplexor 5
- multiplexor 6
- comparador
- Registro 3 (Registro del *pipeline*)

Las señales de control involucradas en esta fase del *pipeline* son:

- **ALU**

  EU_B - Cuando está en '1' lógico habilita a la ALU para que realice una operación básica

- **comparador**

  EU_B - Cuando está activa habilita al comparador para que realice una comparación

  COMP - Resultado de la comparación, puede habilitar CLEAR_C o LI_B

  CA_A - Indica una llamada a subrutina, puede habilitar CLEAR_C

  PD_B - Indica regreso de una subrutina, puede habilitar CLEAR_C

  LO_B - Indica si la unidad de conteo realizará la lectura de una nueva dirección, ya que puede habilitar LI_B

  LI_B - Indica si la unidad de conteo cambiará su secuencia de instrucciones

  CLEAR_C - Cuando está habilitada borra el contenido de los Registros 1 y 2

- **multiplexor 2A**

  CMUX2A_B[2:0] - Línea de selección

- **multiplexor 2B**
  CMUX2B_B[2:0] - Línea de selección

- **multiplexor 3**
  CMUX3_B - Línea de selección

- **multiplexor 5**
  CMUX5_B[1:0] - Línea de selección

- **multiplexor 6**
  CMUX6_B[1:0] - Línea de selección

- **Registro 3 (Registro del *pipeline*)**
  CLK1 - En cada flanco de subida de CLK1, los valores de todas las señales generadas en esta etapa se almacenan en Registro 3, y las deja pasar a la etapa de Acceso a memoria. Entonces la señal cambiará su terminación a "**_C**".

  CMUX4_B
  WB_B
  WR_DAT_B

  CMUX4_C
  WB_C
  WR_DAT_C

2.2.4.1 Multiplexores 2A y 2B

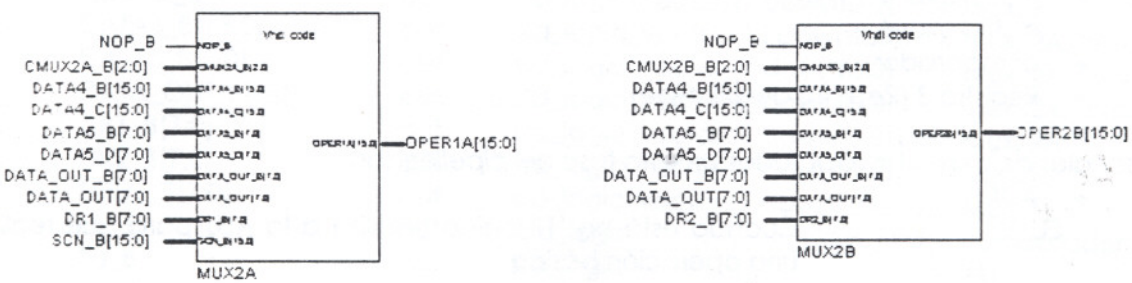


Figura 2.65 Multiplexores que seleccionan los operandos de la ALU

Los multiplexores 2A y 2B, así como los 5 y 6, son más complejos ya que su decisión depende de su línea de control y del estado de la señal NOP_B. Se diseñaron de esta manera para resolver la dependencia de datos que puede llegar a ocasionarse entre un par de instrucciones consecutivas. En la etapa de Lectura, *Forward Unit* asigna un valor a las señales CMUX2A[2:0], CMUX2B[2:0], CMUX5A[1:0] y CMUX6A[1:0]. Este valor depende del tipo de instrucción y de haber identificado o no un caso de dependencia. Estas señales de control se propagan a la Etapa 2 (con terminación "_A") y a la Etapa 3 (ahora con terminación "_B"). Así, obtenemos las señales CMUX2A_B[2:0] y CMUX2B_B[2:0], que son leídas por os multiplexores. Cuando NOP_B está habilitada indica que al pasar por la Etapa 1 la instrucción actual se detectó una dependencia de datos con la instrucción que le precedía. De esta forma, el multiplexor puede elegir qué operando es el adecuado, es decir, el dato que

acaba de ser generado, aún antes de que llegue al Registro 2. Estas señales son: DATA5_D[7:0], que proviene del mux. 4 en la Etapa de Escritura; DATA_OUT[7:0], el dato leído de la memoria de datos en Etapa de acceso a memoria; DATA4_C[15:0], el valor recién calculado por la ALU en el ciclo de reloj anterior. Cuando NOP_B= '0' estos multiplexores elegirán una de las señales provenientes del Registro 2: DATA5_B[7:0], DATA_OUT_B[7:0], DATA4_B[15:0], DR1_B[7:0] o DR2_B[7:0]. Esto lo podemos ver en el siguiente fragmento de código VHDL, que muestran las sentencias *case* de uno de los multiplexores.

```
case CMUX2A_B is
 when "000" => for i in 7 downto 0 loop
                  if NOP_B = '1' then TEMP(i) := DATA5_D(i); -- dependencia de datos con la instr. previa
                  else                TEMP(i) := DATA5_B(i);
                  end if; end loop;
 when "001"=>  TEMP := SCN_B;                              -- ejecucion normal
 when "010" => for i in 7 downto 0 loop
                  TEMP(i) := DR1_B(i);                     -- ejecucion normal
               end loop;
 when "011" => for i in 7 downto 0 loop
                  if NOP_B = '1' then TEMP(i) := DATA_OUT(i); -- dependencia de datos con la instr. previa
                  else                TEMP(i) := DATA_OUT _B(i);
                  end if; end loop;
 when "100" => for i in 7 downto 0 loop
                  if NOP_B = '1' then TEMP(i) := DATA4_C(i); -- dependencia de datos con la instr. previa
                  else                TEMP(i) := DATA4_B(i);
                  end if; end loop;
 when others => null;
end case;
OPER1A <= TEMP;
```

Figura 2.66 *Segmento del archivo MUX2A.HDL (Ver Apéndice B)*

2.2.4.2 ALU

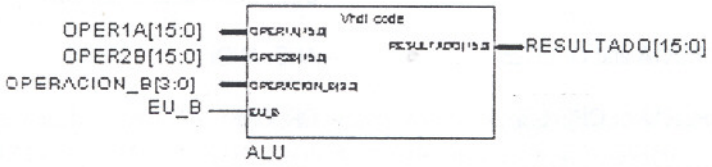


Figura 2.67 Etapa de Ejecución

El código VHDL de la ALU ocupa varias funciones para el manejo de tipos de datos, así como para realizar las operaciones de suma y resta. Las operaciones de corrimiento sólo es posible aplicarlas a un *bit_vector*, y ya que todas las señales de la UAM - RISC II son de tipo *std_logic_vector*, es necesario realizar conversiones entre ambos tipos. Para ello se crearon (dentro del mismo archivo que describe la arquitectura de la ALU) las funciones *SLVECTOR_a_BITVECTOR(operador_slv)* y *BITVECTOR_a_SLVECTOR(operador_bv)*. Para ejecutar la suma existe la función *F_SUMA(operador1, operador2, signo)*, donde el signo indica si se realizará una adición o una sustracción, con '0' ó '1', respectivamente. Aquí cabe señalar que el microprocesador no maneja números negativos, por lo que cuando el resultado de una sustracción es de este tipo, el resultado se cambiará a un vector de 0's.

A partir de la equivalencia de OPERACION_B[3:0] (generada anteriormente por la unidad de control, con terminación "**_A**") con las instrucciones se diseñó la arquitectura de la ALU:

SEÑAL OPERACION_B[3:0]	DESCRIPCIÓN	INSTRUCCIONES
0000	RESTA	SUB, SUBI
0001	SUMA	ADD, ADDI BEQ, BEQI BNEQ, BNEQI, BGA, BGAI BLE, BLEI, JMP
0010	AND	AND, ANDI
0011	OR	OR, ORI
0100	XOR	XOR, XORI
0101	CONCATENACIÓN	LOEX, LOAD, STOEX, STORE
0110	SLL	SLL
0111	SRL	SRL
1000	NOT	NOT, NOTI

Tabla 2. 4 Operaciones básicas ejecutadas por la ALU

El código VHDL de la ALU consiste principalmente en una sentencia *case...is*, que identifica el tipo de operación que se ejecutará. Todas ellas tienen 2 operandos, excepto NOT con sólo 1. Los operandos, OPER1A[15:0] y OPER2B[15:0], son señales externas del tipo *std_logic_vector(15 downto 0)*. El resultado de cualquiera de las instrucciones se almacena en la variable TEMP. Posteriormente este valor es asignado a la señal RESULTADO[15:0], también de 16 bits.

```
case OPERACION_B is

when "0000" =>   TEMP:= F_SUMA(OPER1A, OPER2B, '1');      -- SUB, SUBI

when "0001" =>   TEMP:= F_SUMA(OPER1A, OPER2B, '0');      -- ADD, ADDI, BEQ, BEQI, BNEQ, BNEQI
                                                           -- BGA, BGAI, BLE, BLEI, JMP

when "0010" =>   TEMP:= OPER1A and OPER2B;                 -- AND, ANDI

when "0011" =>   TEMP:= OPER1A or OPER2B;                  -- OR, ORI

when "0100" =>   TEMP:= OPER1A xor OPER2B;                 -- XOR, XORI

when "0101"=>    TEMP:= OPER1A(7 downto 0) & OPER2B(7 downto 0);  -- LOEX, LOAD, STOEX, STORE
```

Figura 2.68 La sentencia case ... *is* permite identificar cada tipo de instrucción y generar señales de control, es decir, decodifica la instrucción.*Segmento del archivo ALU.HDL (Ver Apéndice)*

En el caso de "0000" y "0001" podemos observar que se realiza una llamada a la función *F_SUMA( )*, y que el último argumento indica el signo del segundo operando. En el caso de las instrucciones RAMIFICACIÓN es necesario sumar los 16 bits del primer operando con un offset, que implica un desplazamiento. Las demás operaciones solamente requieren 8 bits, y los bits restantes (del 15 al 8) son siempre 0's. En las operaciones SLL y SRL es necesario

especificar de cuántos bits es el corrimiento (Figura 2.41). En primer lugar se hace una conversión de *std_logic_vector* a un *bit_vector*, y este valor se almacena en TEMPBIT. Se enumeran todos los casos posibles: corrimiento de 0 bits, corrimiento de 1 bit y hasta un máximo de 8 bits. Este resultado se vuelve a convertir en *std_logic_vector*. También hay algunas instrucciones para las cuales no es necesario que la ALU realice una operación, tal es el caso de: LOFF, STOFF, CALL, RETURN.

```
case OPERACION_B is

when "0110" =>    TEMPBIT:= SLVECTOR_a_BITVECTOR(OPER1A);    -- SLL
case OPER2B(3 downto 0) is
        when "0000" => TEMP:= BITVECTOR_a_SLVECTOR(TEMPBIT sll 0);
        when "0001" => TEMP:= BITVECTOR_a_SLVECTOR(TEMPBIT sll 1);
        ...
        when "1000" => TEMP:= BITVECTOR_a_SLVECTOR(TEMPBIT sll 8);
when others => TEMP:= BITVECTOR_a_SLVECTOR(TEMPBIT sll 8);   -- con OPER2B> 8, TEMPBIT son ceros
end case;

when others =>    TEMP:= (others =>'0');                    -- LOFF, STOFF, CALL, RETURN,
                                                            NOP, HALT,
```

Figura 2.69 Se muestra parte de la sentencia *case ... is* que ayuda a realizar el corrimiento de un número específico de bits. También podemos ver que hay algunas instrucciones que no necesitan un valor calculado por la ALU.

Segmento del archivo ALU.HDL (Ver Apéndice)

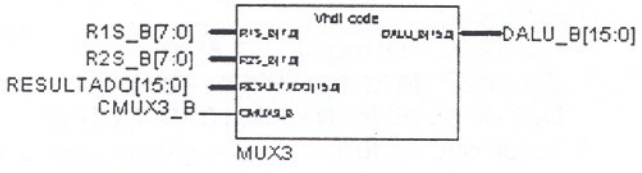


Figura 2.70 Multiplexor 3

La función de multiplexor 3 es muy sencilla: debe elegir el valor recién calculado por la ALU o el par de bytes con valores constantes (contenidos en la palabra de 32 bits de la instrucción actual). Estos dos bytes se concatenan dentro del multiplexor, por lo que la señal de salida es siempre de 16 bits.

Enseguida se presenta la parte medular del código fuente de este dispositivo.

```
case CMUX3_B is
 when '0' =>
          DALU_B <= RESULTADO;                  -- SELECCIONA EL DATO CALCULADO POR LA ALU
 when '1' =>
          DALU_B <= R1S_B &  R2S_B;             -- SELECCIONA EL DATO INMEDIATO (R1 & R2) DE
                                                   LA INSTRUCCION
 when others =>
          DALU_B <= RESULTADO;
end case;
```

Figura 2.71 *Segmento del archivo MUX3.HDL (Ver Apéndice B)*

2.2.4.4 Multiplexor 5 y multiplexor 6

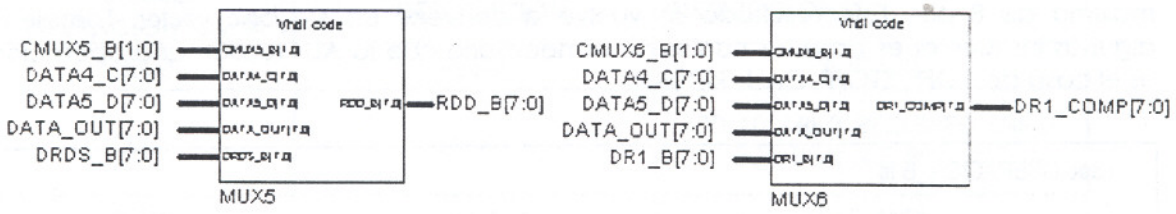


Figura 2.72 Multiplexores 5 y 6

Como ya se ha mencionado, estos multiplexores generan los operandos para el comparador. Reciben señales que provienen de las etapas de Lectura de Operandos (DRDS_B[7:0]), de Acceso a memoria (DATA_OUT[7:0]) y de Escritura de Resultados (DATA5_D[7:0]).

multiplexor 5

Nombre de señal	MODO	Tipo de dato	ORIGEN
DRDS_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
DATA4_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3
DATA_OUT[7:0]	in	std_logic_vector (7 downto 0)	- Mem. Datos
DATA5_D[7:0]	in	std_logic_vector (7 downto 0)	- Mux. 4
CMUX5_B[1:0]	in	std_logic_vector (1 downto 0)	- Registro 2
RDD_B[7:0]	out	std_logic_vector (7 downto 0)	**DESTINO** - comparador

Nombre de señal	Descripción
DRDS_B[7:0]	- Dato leído del registro RD_B[7:0]
DATA4_C[7:0]	- Dirección de la memoria de datos para lectura/escritura
DATA_OUT[7:0]	- Dato leído de la dirección DATA4_C[7:0]
DATA5_D[7:0]	- Valor que actualizará el registro destino de la instrucción que se encuentra en la Etapa 5
CMUX5_B[1:0]	- Línea de selección para el primer dato del comparador
RDD_B[7:0]	- Dato elegido por el multiplexor 5

multiplexor 6

Nombre de señal	MODO	Tipo de dato	ORIGEN
DR1_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
DATA4_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3
DATA_OUT[7:0]	in	std_logic_vector (7 downto 0)	- Mem. Datos
DATA5_D[7:0]	in	std_logic_vector (7 downto 0)	- Mux. 4
CMUX6_B[1:0]	in	std_logic_vector (1 downto 0)	- Registro 2
DR1_COMP	out	std_logic_vector (7 downto 0)	**DESTINO** - comparador

Nombre de señal	Descripción
DR1_B[7:0]	- Dato leído del registro R1_A[7:0] o valor constante
DATA4_C[7:0]	- Dirección de la memoria de datos para lectura/escritura
DATA_OUT[7:0]	- Dato leído de la dirección DATA4_C[7:0]
DATA5_D[7:0]	- Valor que actualizará el registro destino de la instrucción que se encuentra en la Etapa 5
CMUX6_B[1:0]	- Línea de selección para el segundo dato del comparador
DR1_COMP	- Dato elegido por el multiplexor 6

2.2.4.5 Comparador

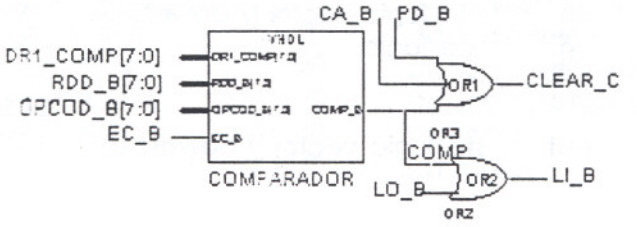


Figura 2.73 El Comparador es necesario en las instrucciones de tipo FLOW

Cuando el microprocesador lee una instrucción de ramificación el comparador decide si se modificará la secuencia de ejecución. Para ello confronta las señales RDD_B[7:0] y DR1_COMP; si la comparación resulta verdadera se habilita la señal COMP. Después esta señal se propaga a CLEAR_B.

El comparador lee el código de operación contenido en los primeros 8 bits de la instrucción, y que en esta etapa es la señal OPCOD_B[7:0]. De acuerdo con este valor es la comparación efectuada cuando la señal que lo habilita (EC_B) está habilitada:

INSTRUCCIÓN	COMPARACIÓN
BEQ BEQI	Igual que =
BNEQ BNEQI	Diferente /=
BGA BGAI	Mayor que >
BLE BLEI	Menor que <

Tabla 2.5 Tipos de comparaciones

Las instrucciones que terminan en "I" tienen un operando inmediato. Así, los operandos de BEQ, BNEQ, BGA y BLE son: el contenido de los registros RD y RS1. Los operandos de BEQI, BNEQI, BGAI y BLEI son: el contenido de RD y el valor constante RS1. Ya que debe leerse el contenido de al menos un registro para realizar una comparación, es posible que el comparador también se vea involucrado en una dependencia de datos. Por ello necesita de los multiplexores 5 y 6, para elegir el valor correcto de sus operandos.

Al mismo tiempo que el comparador manipula estas señales, la ALU también tiene un par de operadores: la dirección de esta instrucción (SCN_B[15:0]) y un offset o desplazamiento

(DR2_B[7:0]). Cuando la comparación es verdadera, el resultado generado por la ALU será la nueva dirección a partir de la cual seguirá la ejecución del programa.

A continuación se detallan las señales que entran y salen del comparador:

Nombre de la señal	MODO	Tipo de dato	ORIGEN
RDD_B[7:0]	in	std_logic_vector (7 downto 0)	- Mux. 5
DR1_COMP	in	std_logic_vector (7 downto 0)	- Mux. 6
OPCOD_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
EC_B	in	std_logic	- Registro 2
			DESTINO
COMP	out	std_logic_vector (7 downto 0)	- OR1, AND1

Nombre de señal	Descripción
RDD_B[7:0]	- Es el primer operando del contador. Es la señal elegida por el mux. 5
DR1_COMP	- Segundo operando. Señal elegida por mux. 6
OPCOD_B[7:0]	- Código de operación de la instrucción que se encuentra en la etapa actual (Ejecución)
EC_B	- Cuando está activa habilita al comparador
COMP	- Contiene el valor de la comparación (0 ó 1)

La señal COMP entra a las compuertas OR2 y OR3. El resultado de *COMP OR LO_B* es el valor de la señal LI_B, que va a la unidad de conteo para modificar la dirección que debe leerse de memoria. Cuando la señal *COMP OR CA_B OR PD_B* dan como resultado '1', CLEAR_C= '1'. Esto significa que CLEAR_C se habilita cuando la comparación de una instrucción tipo RAMIFICACIÓN resulta verdadera o cuando la instrucción en etapa de en la etapa de Ejecución sea una llamada de subrutina o el regreso de una subrutina.

2.2.4.6 Registro 3 (Registro del *pipeline*)

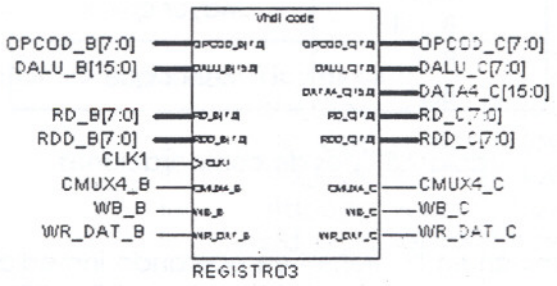


Figura 2.74 Registro 3 del *pipeline*

Este registro da paso a la Etapa 4, de Acceso a memoria. Almacena los valores del código de operación y el resultado de la ALU. Además la dirección del registro destino y su contenido. Entre las señales de control se encuentran las que habilitan la escritura en el bloque de registros de propósito general y en la memoria de datos. Todas las señales que resultan de la etapa de Ejecución tienen terminación "**_B**", y al pasar a la Etapa 4 cambian a

la terminación "**_C**". La señal DALU_B[15:0] se copia en dos señales: DATA4_C[15:0], y sus 8 bits menos significativos en DALU_C[7:0].

A continuación se presentan las señales de entrada y salida del Registro 3:

Nombre de señal	MODO	Tipo de dato	ORIGEN
CLK_1	in	std_logic	
CMUX4_B	in	std_logic	- Registro 2
DALU_B[15:0]	in	std_logic(15 downto 0)	- Registro 2
OPCOD_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
RD_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
RDD_B[7:0]	in	std_logic_vector (7 downto 0)	- Registro 2
WB_B	in	std_logic	- Registro 2
WR_DAT _B	in	std_logic	- Registro 2
			DESTINO
CMUX4_C	out	std_logic	- Registro 4
DALU_C[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 4
DATA4_C[15:0]	out	std_logic_vector (15 downto 0)	- Mem. Datos
OPCOD_B[7:0]	out	std_logic_vector (7 downto 0)	- Forward U.
RD_C[7:0]	out	std_logic_vector (7 downto 0)	- Bloque de Reg.
RDD_C[7:0]	out	std_logic_vector (7 downto 0)	- Mem. Datos
WB_C	out	std_logic	- Bloque de Reg.
WR_DAT _C	out	std_logic	- Mem. Datos

2.2.5 Etapa 4 Accesar a memoria de datos *(Memory Access)*

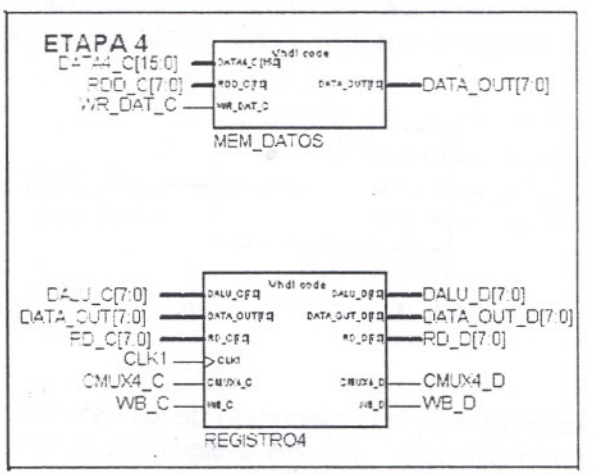


Figura 2.75 Etapa de lectura/escritura en la Memoria de Datos

En esta fase del *pipeline* es posible leer la dirección de memoria que la ALU ha calculado, y en la siguiente se escribirá esta palabra en un registro. También podemos almacenar en la dirección calculada de la memoria de datos el contenido de un registro previamente leído.

En esta fase del *pipeline* son necesarios 2 dispositivos:

- memoria de datos

- Registro 4 (Registro del *pipeline*)

Las señales de control involucradas son:

- **memoria de datos**
  WR_DAT_C - Cuando la línea tiene el valor '0' se habilita la lectura; cuando está habilitada permite la escritura

- **Registro 4 (Registro del *pipeline*)**
  CLK1 - En cada flanco de subida de CLK1, los valores de todas las señales generadas en esta etapa son leídos por el Registro 4. Al mismo tiempo, las deja pasar a la última etapa. Cuando sucede esto, el nombre de la señal cambia y su terminación será **"_D"**.
  CMUX4_C - Línea de selección del multiplexor 4
  WB_C - Cuando está activa habilita un registro para escritura

2.2.5.1 Memoria de datos

Se trata de una memoria asíncrona de 8 bits, con espacio direccionable de 16 bits, lo que implica 2^{16} direcciones de memoria. Ya que el espacio en el FPGA es limitado únicamente se implementaron 2^5 direcciones, y el bus permanece con 16 bits, pensando en futuras actualizaciones que requieran ese número de bits.

La unidad de control generó previamente la señal WR_DAT_**A** ='0' cuando se encontró con una instrucción de lectura de la memoria de datos, es decir, de tipo LOAD. En este caso, al llegar a la etapa de Acceso a memoria, la señal ya se ha convertido en WR_DAT_**C** ='0'. Cuando ésta llega a la memoria de datos, la palabra contenida en la localidad DATA4_C[15:0] pasa al bus de salida DATA_OUT[7:0]. Por el contrario, cuando WR_DAT_**C** ='1', la palabra contenida en el bus de datos RDD_C[7:0] se escribe en memoria.

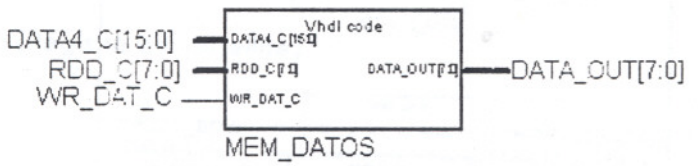


Figura 2.76 Memoria de datos, diseñada en VHDL

Si la memoria fuera síncrona retardaría la ejecución de las instrucciones, ya que en un flanco de subida de CLK1 leería la dirección, y hasta el siguiente flanco permitiría la salida de la palabra de memoria. De esta forma, con una memoria asíncrona, el acceso se puede realizar en un solo ciclo de reloj.

Las instrucciones de Acceso a memoria son de tipo LOAD y STORE. Por ejemplo:

LOEX (RD):= M[(RS1) + (RS2)]

La instrucción LOEX implica que los registros fuente RS1 y RS2 sean leídos y que en la Etapa de Ejecución el contenido de ambos sea concatenado por la ALU. El resultado de esta

operación es una señal de tipo std_logic_vector de 16 bits, que pasará a la señal DATA4_C[15:0]. Para acceder a una posición de este arreglo de memoria es necesario que la dirección sea un número de tipo entero, por lo tanto se tiene que aplicar esta conversión al número DATA4_C[15:0]. Después se lee esa posición y la palabra pasa al bus de datos DATA_OUT[7:0]. En la siguiente etapa, de escritura, la palabra leída se almacenará en el registro RD.
Otro ejemplo es la instrucción STORE:

STORE M[(RS1) + RS2]:= (RD)

Cuando el control la decodifica generará las señales correspondientes para leer el registro RD; además hará que el contenido del RS1 se concatene con el valor RS2 en la ALU. Éste resultado también es un std_logic_vector de 16 bits, que se propagará a DATA4_C[15:0]. También en este caso, al ser convertido en un entero, DATA4_C[15:0] indica la dirección de memoria de datos en donde se almacenará RDD_C[7:0] (el contenido de RD).

Nombre de la señal	MODO	Tipo de dato	ORIGEN
DATA4_C[15:0]	in	std_logic_vector (15 downto 0)	- Registro 3
RDD_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3
WR_DAT_C	in	std_logic	- U. Control
			DESTINO
DATA_OUT[7:0]	out	std_logic_vector (7 downto 0)	- Registro 2, Registro4

Nombre de señal	Descripción
DATA4_C[15:0]	- Es la dirección de la memoria de datos en la que se escribirá o leerá
RDD_C[7:0]	- Es el dato que se escribe en la dirección indicada (DATA4_C[15:0]) de la memoria de datos. Es el bus de datos (*input*)
WR_DAT_C	- Cuando = '0', indica lectura. Cuando = '1', escritura
DATA_OUT[7:0]	- Es el dato que se lee de la dirección indicada (DATA4_C[15:0]) de la memoria de datos. Es el bus de datos (*output*)

2.2.5.2 Registro 4 (Registro del *pipeline*)

Figura 2.77 Registro de la Etapa 4, de Acceso a Memoria

El último registro del *pipeline* es el más pequeño, ya que solamente almacena el resultado de la ALU, el valor leído de la memoria de datos, la dirección del registro destino y la línea que habilita la escritura de resultados (para la siguiente etapa), así como la línea de selección del multiplexor 4.

Nombre de señal	MODO	Tipo de dato	ORIGEN
CLK_1	in	std_logic	
CMUX4_C	in	std_logic	- Registro 3
DALU_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3
DATA_OUT[7:0]	in	std_logic_vector (7 downto 0)	- Mem. de Datos
RD_C[7:0]	in	std_logic_vector (7 downto 0)	- Registro 3
WB_C	in	std_logic	- Registro 3
CMUX4_D			**DESTINO**
DALU_D[7:0]	out	std_logic	- Mux. 4
DATA_OUT_D[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 4
RD_D[7:0]	out	std_logic_vector (7 downto 0)	- Mux. 4
WB_D	out	std_logic_vector (7 downto 0)	- Bloque de Reg.
	out	std_logic	- Bloque de Reg.

Nombre de señal	Descripción
CMUX4_D	- Cuando es igual a '0', elige a DATA_OUT_D[7:0]. Cuando es igual a '1' elige a DALU_D[7:0]
DALU_D[7:0]	- Resultado de la ALU o de (R1S_B[7:0] & R2S_B[7:0])
DATA_OUT_D[7:0]	- Palabra leída de la memoria de datos
RD_D[7:0]	- Dirección del registro que se actualizará en la siguiente etapa
WB_D	- Cuando está activa habilita la escritura en un registro

2.2.6 Etapa 5 Escribir Resultados *(Write Back)*

La última etapa del *pipeline* se encarga de escribir en un registro destino el resultado calculado por la ALU o un valor leído de memoria. Esta etapa requiere dos elementos del UAM - RISC II:

- multiplexor 4
- bloque de registros de propósito general

Ya hemos detallado el funcionamiento del Bloque de Registros en la Etapa 2 del *pipeline* (Fig. 2.33), por lo tanto a continuación sólo se presenta el mux. 4:

2.2.6.1 Multiplexor 4

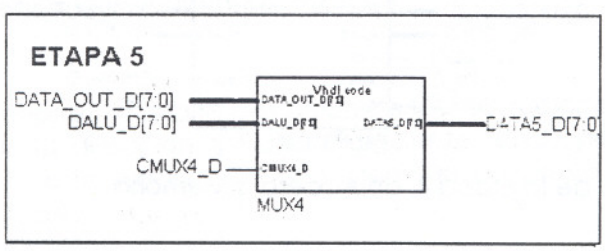


Figura 2.78 Multiplexor de la última etapa del *pipeline*

Este dispositivo se encarga de seleccionar qué valor actualizará el registro destino. Puede ser la señal DALU_D[7:0], que contiene el resultado calculado por la ALU en una instrucción de

tipo OPERACIÓN. También puede tratarse de la señal DATA_OUT_D[7:0], que es el contenido de una localidad de memoria de datos de una instrucción de tipo LOAD.

Nombre de la señal	**MODO**	**Tipo de dato**	**ORIGEN**
DATA_OUT_D[7:0]			
DALU_D[7:0]	in	std_logic_vector (7 downto 0)	- Registro 4
CMUX4_D	in	std_logic_vector (7 downto 0)	- Registro 4
	in	std_logic	- Registro 4
			DESTINO
DATA5_D[7:0]	out	std_logic_vector (7 downto 0)	- Bloque de Registros

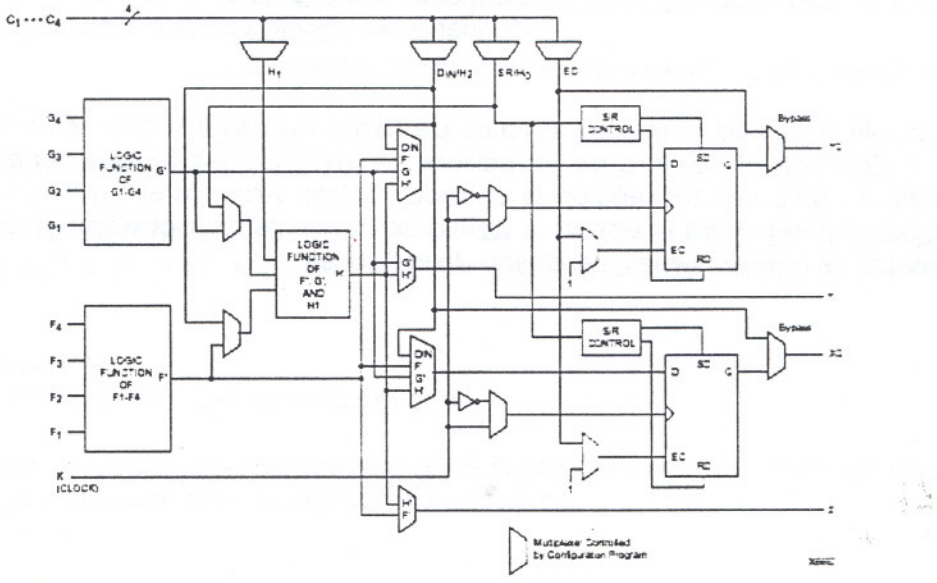
Nombre de señal	**Descripción**
DATA_OUT_D[7:0]	
DALU_D[15:0]	- Palabra leída de la memoria de datos - Es el valor calculado por la ALU o el vector constante (R1S_B[7:0] & R2S_B[7:0]), que actualizará el registro destino
CMUX4_D	- Cuando está deshabilitada elige a DATA_OUT_D[7:0]. En caso contrario elige a DALU_D[7:0]
DATA5_D[7:0]	- Contiene al dato elegido por el multiplexor 4

Este capítulo nos ha permitido describir de forma detallada cómo está diseñado el UAM - RISC II. Esto con el objetivo de especificar en VHDL la función que tiene cada uno de los dispositivos, así como sus señales de entrada y salida. Además el capítulo incluye fragmentos del código programado cuando se consideró conveniente, por ejemplo: cuando se crearon instancias de componentes, del diseño de la ALU, etc.

Capítulo 3 Simulación e Implementación

3.1 Características del FPGA utilizado

Ya que los FPGAs de la familia XC4000 tienen gran cantidad de *flip-flops*, resulta conveniente emplearlos en diseños con *pipeline*. En este tipo de arquitecturas se divide el procesamiento en subfunciones que se ejecutan en paralelo, y los resultados se almacenan en estos *flip-flops* (Figura. 3.1). El dispositivo en el que se implementó este proyecto pertenece a la familia XC4000XL de Xilinx [24]. Su arquitectura incluye gran número de interconexiones, que le permiten al FPGA aceptar diseños complejos. También contienen numerosos bloques I/O (IOBs). Tiene disponibles 8 señales de reloj que se distribuyen a lo largo de todo el FPGA. Y llegan a soportar velocidades de hasta 80 MHz. Cada CLB dentro del dispositivo contiene 3 generadores de funciones booleanas que se implementan como LUTs (*Look Up Tables*), en realidad elementos de memoria. Los generadores F y G son de 4 entradas. El generador H tiene 3 señales de entrada, dos de las cuales pueden ser el resultado de F y G, o pueden ser externas al CLB. A continuación podemos ver, la estructura de un CLB perteneciente a la familia XC4000.



Simplified Block Diagram of XC4000 Series CLB (RAM and Carry Logic functions not shown)

Figura 3.1 Estructura interna de un CLB de la familia XC4000 [24]

Las características de nuestro dispositivo están resumidas en la siguiente tabla. Podemos ver que se trata de un modelo grande, con gran número de *flip-flops*, CLBs y IOBs. Ahora podemos ver las razones para elegir este dispositivo: el ser apropiado para el diseño de un *pipeline*, su tamaño, y el permitir más de una señal de reloj.

Device	Logic Cells	Max Logic Gates (No RAM)	Max. RAM Bits (No Logic)	Typical Gate Range (Logic and RAM)*	CLB Matrix	Total CLBs	Number of Flip-Flops	Max. User I/O
XC4085XL	7448	85,000	100,352	55,000 - 180,000	56 x 56	3,136	7,168	448

Tabla 3.1 Características del FPGA XC4085XL [24]

3.2 Simulación e Implementación del Diseño

Se diseñó cada uno de los elementos del núcleo del microprocesador y se simuló su funcionamiento. Después de esto se organizaron las diferentes etapas del *pipeline*, y se agregó un registro entre cada par de etapas. A partir de esto se realizó la simulación de todo el diseño para probar su funcionamiento, se implementó y se volvió a probar la ejecución de todo el conjunto, ahora considerando los retardos.

Esta etapa del proyecto resultó la más compleja, ya que se realizaron numerosos cambios en el diseño con el propósito de sincronizar las señales dentro del *pipeline*. Cada modificación en el código o en la estructura de los dispositivos hizo necesario repetir el ciclo de simulación funcional, implementación y simulación con retardos.

3.2.1 Herramientas

El comportamiento de cada elemento se diseñó mediante el *HDL Editor* y el *Schematic Editor* de Xilinx, de los que ya hemos hablado en la sección 1.3.2. Durante la etapa de diseño, para comprobar que la descripción que hemos hecho es correcta, es necesario verificar la sintaxis de este archivo. Enseguida, para verificar su funcionamiento se sintetiza el diseño. Ambas tareas las realiza la aplicación *DPM*, de *Synopsys*.

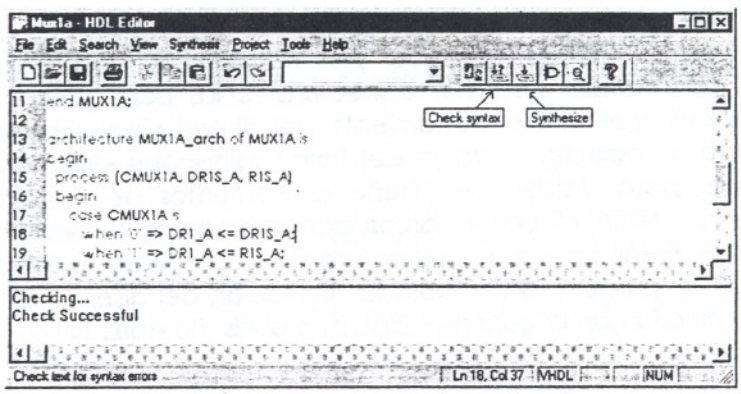


Figura 3.2 Herramientas para verificar sintaxis y sintetizar

Después de realizar estos pasos para cada uno de los elementos de este núcleo de un microprocesador RISC, se organizaron las 5 etapas del procesamiento, agregando los 4 registros del *pipeline* necesarios. En este caso resulta muy práctico el uso del *Schematic Editor*, pues nos permite visualizar cada elemento, con sus señales y buses de I/O.

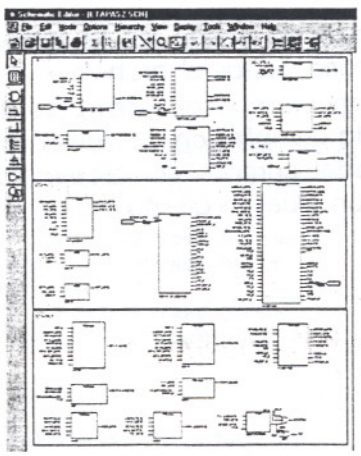


Figura 3.3 Herramienta para diseño esquemático

En un *pipeline* es imprescindible la sincronización entre las diferentes etapas. En este caso, para lograrlo fue necesario realizar numerosas simulaciones, con el fin de ir perfeccionando el diseño. Para ello empleamos el *Logic Simulator*, en la opción de simulación funcional. Ya que el número de señales involucradas en la ejecución del microprocesador es muy grande, lo mejor es crear un archivo que llame a todas las señales del diseño, y que necesitamos visualizar; para ello empleamos el *Script Editor*.

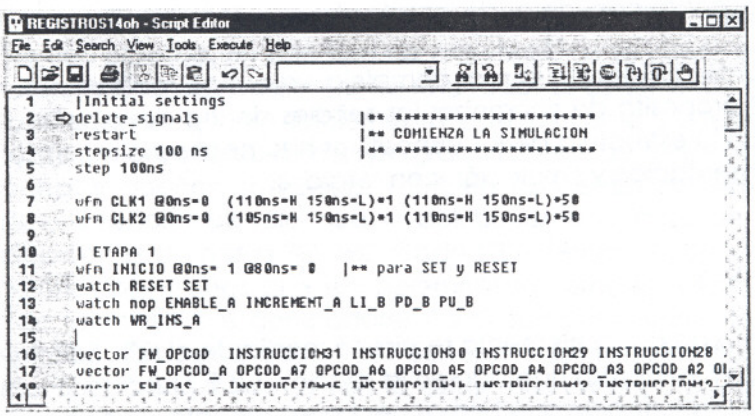


Figura 3.4 Herramientas para crear y ejecutar *scripts*

Con esta herramienta podemos organizar las señales como líneas y buses, además de indicarles el orden en que deben aparecer y su notación al ejecutar la simulación. Más adelante veremos con detalle las simulaciones realizadas para probar el UAM - RISC II. Después de comprobar que el comportamiento del diseño es el deseado, pasamos a la implementación. Ésta es realizada a través del *Flow Engine*, que ejecuta automáticamente diferentes programas para: traducir el diseño a elementos del FPGA, verificar que las restricciones de tiempo sean válidas y comprobar la ausencia de señales desconectadas (*Translate*); mapear el diseño lógico a la estructura del FPGA *(Map);* acomodar y conectar el diseño *(Place & Route);* producir datos para la simulación del diseño *(Timing);* y finalmente producir un archivo binario con la configuración que el diseño dará al FPGA *(Configure)*. (Ver Figura 1.8).

Estas tareas se realizan de forma automática al elegir en el *Project Manager* la opción *Implementation,* y su ejecución es transparente al usuario. También desde esta interfaz podemos indicar el nivel de esfuerzo para *Place & Route*. Para implementar este microprocesador elegimos el nivel alto, que requiere más tiempo para completarse. Este nivel se sugiere en diseños que no cumplen con el performance deseado, o cuando producen algún error con el nivel medio o bajo durante el *Place* [22].

Otra forma de mejorar el performance de un diseño son las restricciones de tiempo (*Timing constraints*). Para agregarlas es necesario editar el archivo *.ucf que se encuentra en el directorio del proyecto. Estas restricciones son evaluadas durante la etapa PAR (*Place & Route*). En caso de que el diseño pueda cumplir con las restricciones, es posible que mejore su desempeño. Cuando no las puede cumplir, se produce un error y es necesario modificarlas y reiniciar la implementación. Nuestro diseño contiene algunas restricciones, que serán analizadas más adelante.

3.2.2 Reportes generados

Cuando ha terminado la implementación es posible revisar los diversos reportes que se han generado en cada una de las etapas del *Flow Engine: Translate, Map, Place & Route, Timing, Configure* (ver figura 1.8).

Entre la información que encontramos en el *Implementation Log File* están el nombre del FPGA elegido y su velocidad. *Covermode* se refiere a la principal restricción para mapear el diseño al dispositivo: área o velocidad. El elegir esta última implica que habrá funciones en paralelo dentro del FPGA, y por lo tanto serían necesarios más CLB's. Para implementar nuestro diseño indicamos que el mapeo se realice por área, y por lo tanto, con funciones seriales que pueden emplearse recursivamente, se ocupará el mínimo posible de CLB's.

```
Using target part "4085xlbg560-09".
MAP xc4000xl directives:
     Partname = "xc4085xl-09-bg560".
    Covermode = "area".
```

El *Map Report (map.mrp)* presenta el número de elementos del FPGA que nuestro diseño ha utilizado. Este resumen también lo encontramos en el reporte **.par (Place & Route)*[4]. Podemos observar que se encuentra ocupado el 90% de los CLB's. Esto se debe principalmente a que al implementar las funciones requeridas para el diseño de este núcleo de un microprocesador se requieren gran parte de las LUTs de 4 entradas y algunas de 3 entradas. Además podemos ver reflejada la existencia de 2 señales de reloj, al ser utilizados 2 *Global Clock Buffers (BUFGLS)*. Sería posible modificar un poco estos valores al hacer cambios directamente sobre el mapeo, empleando la herramienta *Floorplanner*.

```
Device utilization summary:
   Number of External IOBs              4 out of 448      1%
        Flops:                                  0
        Latches:                                0
   Number of CLBs                    2837 out of 3136     90%
        Total Latches:                  0 out of 6272      0%
        Total CLB Flops:              612 out of 6272      9%
        4 input LUTs:                5516 out of 6272     87%
        3 input LUTs:                1128 out of 3136     35%
   Number of BUFGLS:                    2 out of    8     25%
```

En el reporte *Logic Level Timing Report (map.twr)* podemos observar la estimación del desempeño de nuestro diseño durante el mapeo. Cuando éste ocurre se están considerando cada uno de los elementos utilizados dentro del FPGA, pero no los retrasos debido a las interconexiones entre ellos, ya que aún no se ha realizado el ruteo. Por lo tanto, esta estimación nos da el desempeño ideal de nuestro diseño:

```
Design statistics:
   Minimum period:  51.494ns (Maximum frequency:  19.420MHz)
   Maximum path delay from/to any node:  58.679ns
```

Es posible controlar algunos aspectos de la etapa *Place & Route*. En el reporte correspondiente *(*.par)* se muestran estos parámetros, entre los que se encuentra el nivel de esfuerzo realizado por el ruteador, que mencionábamos anteriormente.

```
Overall effort level (-ol):   5 (set by user)
Placer effort level (-pl):    5 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    5 (set by user)
Extra effort level (-xe):     1 (default)
```

[4] El símbolo * denota el nombre del proyecto actual

También se presentan, en forma general, los retrasos. El tiempo promedio de propagación para todas las señales del diseño es de aprox. 6ns. El promedio de las señales con mayor retraso es de casi 18ns. Y la línea que presenta el retraso máximo requiere 32ns:

```
The Number of signals not completely routed for this design is: 0
The Average Connection Delay for this design is:         5.585 ns
The Maximum Pin Delay is:                               32.062 ns
The Average Connection Delay on the 10 Worst Nets is:   17.996 ns
```

Considerando el intervalo entre los valores promedio y máximo se muestra el número de señales que tienen cierto retardo. Por ejemplo, 14157 señales (61.6%) requieren menos de 6ns para propagarse, 5122 señales (22.3%) requieren entre 6 y 12ns, y así sucesivamente, hasta llegar a 213 señales (0.9%) que tardan entre 24 y 33ns.

```
Listing Pin Delays by value: (ns)
 d < 6.00   < d < 12.00  < d < 18.00  < d < 24.00  < d < 33.00  d >= 33.00
---------   ---------   ---------   ---------   ---------   ---------
  14157        5122        2743        738         213          0
```

En este reporte también se presentan las restricciones de tiempo que se hayan incluido en el proyecto al editar el archivo *.ucf a través del *Constraints Editor*. En este caso, decidimos poner restricciones a los 2 relojes que sincronizan el *pipeline*, buscando mejorar el desempeño general del microprocesador. El contenido del archivo *.ucf es el siguiente:

```
NET CLK1 TNM_NET =              CLK1_grp ;
NET CLK2 TNM_NET =              CLK2_grp ;
NET "$U10/RD_A<*>" TNM_NET =  ESP1;

TIMESPEC TS_CLK1 =            PERIOD : CLK1_grp : 99 : LOW 40;
TIMESPEC TS_CLK2 =            PERIOD : CLK2_grp : 99 : LOW 85;
TIMESPEC TS_CLK1_2_CLK2 =  FROM : CLK1_grp : TO : CLK2_grp : 110 ;
TIMESPEC TS_CLK2_2_CLK1 =  FROM : CLK2_grp : TO : CLK1_grp : 6.5 ;
TIMESPEC TS_ESPECIF1 =        FROM: FFS : TO : ESP1: TS_CLK1*1;
```

En la primera restricción se ha creado un grupo, *"CLK1_grp"*, con todas las señales sincronizadas por CLK1:

NET CLK1 TNM_NET = CLK1_grp ;

Después se define para este grupo un *Timing Specification, TS_CLK1*, que propone que el período de esta señal sea 99ns, permaneciendo 40ns en el 0 lógico. La duración real es de 97.932ns. El valor propuesto se definió luego de varios intentos, hasta ajustarlo lo más posible al real, pero sin sobrepasarlo, pues en tal caso la restricción no se cumpliría y provocaría un error que detendría la implementación. Para ello nos auxiliamos del *Logic Level Timing Report*, que detalla cada restricción presentando el retardo de cada una de las señales involucradas.

TIMESPEC TS_CLK1 = PERIOD : CLK1_grp : 99 : LOW 40;

Otra restricción define el retardo de todas las señales sincronizadas por CLK1 que modifican a otro grupo de señales, sincronizadas por CLK2. Este *Timing Specification*, TS_CLK1_2_CLK2, propone un retardo de 110ns, cuando el real es 109.386ns:

TIMESPEC TS_CLK1_2_CLK2 = FROM : CLK1_grp : TO : CLK2_grp : 110 ;

Después de obtener una mejoría significativa en la velocidad, se analizaron las señales con mayor retardo en el *Asynchronous Delay Report*, para incluir nuevas restricciones. Al descubrir que el bit 0 de la señal RD_A[7:0] se encontraba en primer lugar entre las 20 señales más lentas, se hizo necesario aplicarle una restricción. Recordemos que esta señal es la dirección de un registro que será leído en la Etapa de Decodificación y Lectura de Operandos. Para identificarla, le dimos un nombre al conjunto de 8 bits de la señal RD_A, perteneciente al bloque de registros de propósito general (símbolo $U10).

NET "$U10/RD_A<*>" TNM_NET = ESP1;

La siguiente restricción indica que la comunicación desde cualquier *flip-flop* hasta el conjunto ESP1 debe tardar como máximo TS_CLK1*1, es decir, 99ns. Dentro del diseño, esto lo podemos interpretar como: la instrucción leída de memoria será descompuesta en tres direcciones al pasar por el Registro 1 del *pipeline*; una de estas direcciones es RD_A, que para ir del Registro 1 al Bloque de Registros debe tardar máximo 99ns.

TIMESPEC TS_ESPECIF1 = FROM: FFS : TO : ESP1: TS_CLK1*1;

De esta manera, en el reporte descubrimos que se logró reducir el retardo de RD_A[0] de 335.195ns a 12.803ns. Además este cambio repercutió favorablemente en el resto del diseño, ya que ahora el mayor retardo de una señal es de 32.062ns (y ya no corresponde a la señal RD_A[0]). Enseguida se presenta cada especificación de tiempo con: el tiempo sugerido en la primera columna, el tiempo real en la columna central y en la tercera columna el número de niveles lógicos que atraviesa dentro del FPGA el conjunto de señales.

```
Asterisk (*) preceding a constraint indicates it was not met.
------------------------------------------------------------------------------
  Constraint                                | Requested  | Actual     | Logic Levels
------------------------------------------------------------------------------
  TS_CLK1 = PERIOD TIMEGRP "CLK1_grp"       | 99.000ns   | 97.932ns   |    19
     99 nS LOW 40 nS                        |            |            |
------------------------------------------------------------------------------
  TS_CLK2 = PERIOD TIMEGRP "CLK2_grp"       |            |            |
     99 nS LOW 85 nS                        |            |            |
------------------------------------------------------------------------------
  TS_CLK1_2_CLK2 = MAXDELAY FROM TIMEGRP    | 110.000ns  | 109.386ns  |    23
 "CLK1_grp" TO TIMEGRP "CLK2_grp" 110 nS    |            |            |
------------------------------------------------------------------------------
  TS_CLK2_2_CLK1 = MAXDELAY FROM TIMEGRP    | 6.500ns    | 4.614ns    |     2
  "CLK2_grp" TO TIMEGRP "CLK1_grp" 6.500 nS |            |            |
------------------------------------------------------------------------------
  TS_ESPECIF1 = MAXDELAY FROM TIMEGRP       | 99.000ns   | 58.137ns   |    12
  "FFS" TO TIMEGRP "ESP1" TS_CLK1 * 1.000   |            |            |
------------------------------------------------------------------------------
All constraints were met.
Dumping design to file risca.ncd.
```

Durante el PAR, la herramienta Xilinx *Timing Analizer* genera un nuevo reporte: *Post Layout Timing Report (*.twr)*. En él se vuelven a analizar las restricciones de tiempo, e indica que cubren el 71.6% de las señales de todo el proyecto. Además se presenta el periodo mínimo requerido para ejecutar este diseño, la frecuencia de ejecución y el retraso máximo que ocurre para propagar una señal de un dispositivo a otro. Podemos considerar que el desempeño que se muestra en este reporte es el real, ya que considera los retardos.

```
Constraints cover 115543064 paths, 0 nets, and 11156 connections (71.6% coverage)

Design statistics:
   Minimum period:  95.303ns (Maximum frequency:  10.493MHz)
   Maximum path delay from/to any node: 106.764ns
```

Finalmente, cabe mencionar que es posible crear varias revisiones de la implementación de un diseño, para conservar los cambios en el diseño o en los parámetros de la implementación. En otra revisión no agregamos ningún tipo de restricción de tiempo, y su desempeño fue inferior, como lo vemos a continuación. A partir de estos datos confirmamos que las restricciones de tiempo aplicadas fueron de utilidad.

```
Design statistics:
   Minimum period: 407.536ns (Maximum frequency:   2.454MHz)
   Maximum net delay: 293.846ns
```

3.2.3 Análisis de la Simulación

En esta figura es posible ver la simulación de la ejecución de las primeras instrucciones, dentro de las localidades 0 - F de la memoria de instrucciones. Se presentan las señales involucradas con los dispositivos de la ***Etapa 1, Lectura de la Instrucción***. Ya hemos descrito la forma en que se ejecutan las instrucciones, y ahora podemos observarlo a través de la interfaz del Logic Simulator. Las localidades 0 y 1 contienen una palabra de 0's: B"00000000_00000000_00000000_00000000", lo que implica que la unidad de control sólo activará las líneas ENABLE_A e INCREMENT_A para la unidad de conteo. También podemos ver en la figura que la instrucción con dirección 2 (DIR= D"0002") se lee a los 600ns y que contiene la palabra[5] "0A031A03" (B"00001010_00000011_00011010_00000011").

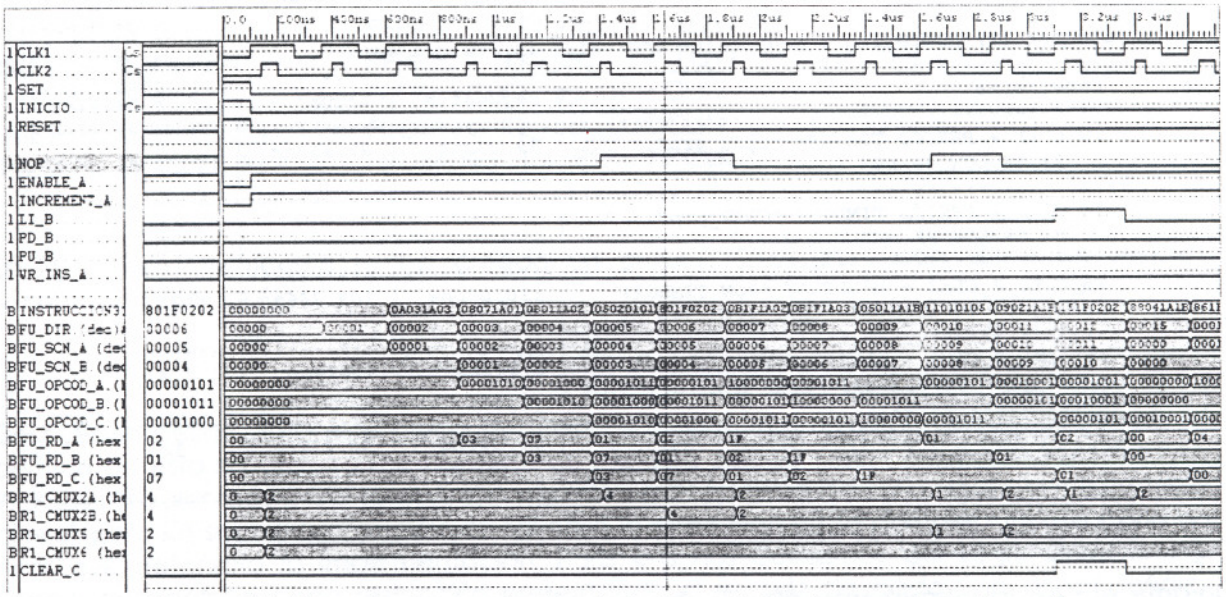


Figura 3.5 Ejemplo de simulación de la fase de Lectura de la Instrucción

Debido a la estructura en *pipeline*, en la misma subida de CLK1, se están leyendo FU_OPCOD_A[6] = B"00000000", FU_OPCOD_B= B"00000000" y FU_OPCOD_C = B"00000000",

[5] La notación de las señales es hexadecimal, a menos que se indique lo contrario, con B (binario)o D (decimal)al inicio.

[6] Al crear el script para la simulación se agregó a varias señales un prefijo: FU (porque pertenecen a la *Forward Unit*), R1 (porque pertenecen al Registro 1), R2 (porque pertenecen al Registro 2), etc.

además de las direcciones de los registros destino de las 3 instrucciones previas: RD_A, RD_B y RD_C, para que la *Forward Unit* realice las comparaciones correspondientes en la subida de CLK2.

Más adelante, cuando se lee la instrucción en la ***localidad 4 (en 1.1μs)***, observamos que forma una secuencia inversa ("0004", "0003", "0002") con las direcciones de las 2 instrucciones que anteriormente entraron al *pipeline* y que aún se encuentran en él. Los registros que se actualizarán con estas instrucciones son RD_A= "07" y RD_B= "03", respectivamente. Podemos corroborar esto observando el bus INSTRUCCIÓN. En el valor que toma a los 600ns, el primer par de números corresponde al código de operación "0A", el siguiente par es la dirección del registro destino "03". Cuando esta señal cambia tiene un nuevo código de operación, "08", y un nuevo registro destino "07".

Analicemos ahora la palabra leída de la posición ***DIR= D"0005"*** de la memoria de instrucciones: "05020101". El código de operación es "05"= B"00000101", que corresponde a la operación ***ANDI***. La *Forward Unit* verifica de qué tipo es la instrucción previa; al leer OPCOD_A= B"00001011" determina que su tipo es OPERACIÓN y realiza más comparaciones:

- El registro destino de la instrucción D"0004" (RD_A= "01") es igual al primer registro fuente actual ("01"). Esto ocasiona una ***dependencia de datos*** con la instrucción actual, y por lo tanto NOP= '1' y CMUX2A[2:0]= B"100".
- El registro destino de la instrucción "0004" es igual al segundo registro fuente actual ("01"). Pero no modifica el valor de NOP ni de CMUX2B[2:0], ya que ANDI tiene como segundo operando a un valor inmediato, y por lo tanto "01" no es la dirección de un registro, sino el propio operando, que no puede dar lugar a una dependencia de datos.

Con la subida de CLK2 en ***1.640μs***, la *Forward Unit* detecta una nueva ***dependencia de datos***. Ahora se trata de las ***instrucciones "0005" y "0006"***. Esta última está marcada por una línea vertical, y por lo tanto también muestra sus valores en la columna central, de color blanco. Su código de operación es "80"= B"10000000"= ***LOEX***. Se trata de una instrucción de tipo LOAD, que concatenará el valor de dos registros y esto lo almacenará en un registro destino. La *Forward Unit* realiza las mismas comparaciones que en el ciclo anterior. Al detectar que el registro que actualizará ANDI coincide con los 2 registros requeridos por LOEX, activa NOP y asigna CMUX2A[2:0]= B"100" y CMUX2B[2:0]= B"100". En el siguiente ciclo ya no se detecta una dependencia de datos y la señal NOP ahora estará inactiva.

También en la figura 3.5 se muestra la lectura de una instrucción de ramificación, ***BEQI***. Se encuentra en la localidad ***D"0010"***. Cuando esta instrucción llegue a la Etapa de Ejecución saltará a la localidad D"0015". En el bus de direcciones, DIR, observamos que la secuencia es alterada: D"0010", D"0011", D"0012", D"0015". Mientras tanto, en este ciclo se verifica la existencia de una dependencia de datos. La instrucción es "11010105", y el segundo par de bits representa el registro fuente RD[7:0]= "01". Los siguientes bits, "01" y "05" son datos inmediatos, y por lo tanto no pueden causar dependencia. Pero sí puede ocasionarla la lectura de RD, y por ello en primer lugar se identifica el tipo de la instrucción D"0009", OPERACIÓN. Luego la *Forward Unit* realiza las siguientes comparaciones:

- RD_A[7:0] = RD[7:0] = "01" y la instrucción actual es de tipo RAMIFICACIÓN, entonces NOP= '1' y CMUX5[1:0]= "01".

Más adelante seguiremos el comportamiento de estas instrucciones, a lo largo de las etapas del *pipeline*.

Como ya hemos visto, durante la ***segunda etapa*** el *pipeline* del UAM - RISC II es de Decodificación de Instrucción y Lectura de Operandos (Decode). Por lo tanto ahora visualizamos otro grupo de señales en la columna de la izquierda (Figura 3.5). ER habilita la lectura de registros, los valores contenidos en los registros leídos: DR1S_A, DR2S_A, DRDS_A. Enseguida se encuentran las direcciones de los registros fuente o datos inmediatos, según sea el caso. Después se muestran las líneas de selección de los multiplexores 1A y 1B, seguidas por el dato que han elegido para pasar al Registro 2. La unidad de control lee el código de operación y genera las señales de control correspondientes. Finalmente el Registro 2 recibirá DATA_OUT (proveniente de la memoria de datos en Etapa 4), DATA4_C (desde el Registro 3
en Etapa 3), DATA5_D (desde el mux. 4 en Etapa 5). Estas 3 señales pueden servirnos para resolver una dependencia de datos, pero en la siguiente etapa del *pipeline*. En la figura 3.6 aparecen sólo porque serán leídas por el Registro 3.

Como ya mencionábamos, existe una ***dependencia de datos entre las instrucciones 4 y 5***. En esta sección de la simulación podemos observar que el primer operando de la instrucción 5 será seleccionado con CMUX2A_A= '4': en lugar de DR1_A se elige DATA4_C.

Los datos correspondientes a la instrucción en la ***localidad D"0005"*** se encuentran a partir de la subida de CLK1 que ocurre a los ***1.6μs***, después de la línea vertical. Podemos ver que el valor de los 3 registros leídos es "00". Ya que ***ANDI*** tiene un operando inmediato, CMUX1A= '0' y CMUX1B= '1'; y por tanto se seleccionan DR1_A= "00" y el valor inmediato DR2A= "01" como posibles operandos. Su código de operación es OPCOD_A= B"00000101" y equivale a la operación básica OPER_A= '2', es decir, equivale a un AND. Para realizar esta operación empleará la ALU, y por lo tanto se habilita la señal EU_A. La actualización del registro destino requerirá la escritura de un registro y por ello WB_A está activa.

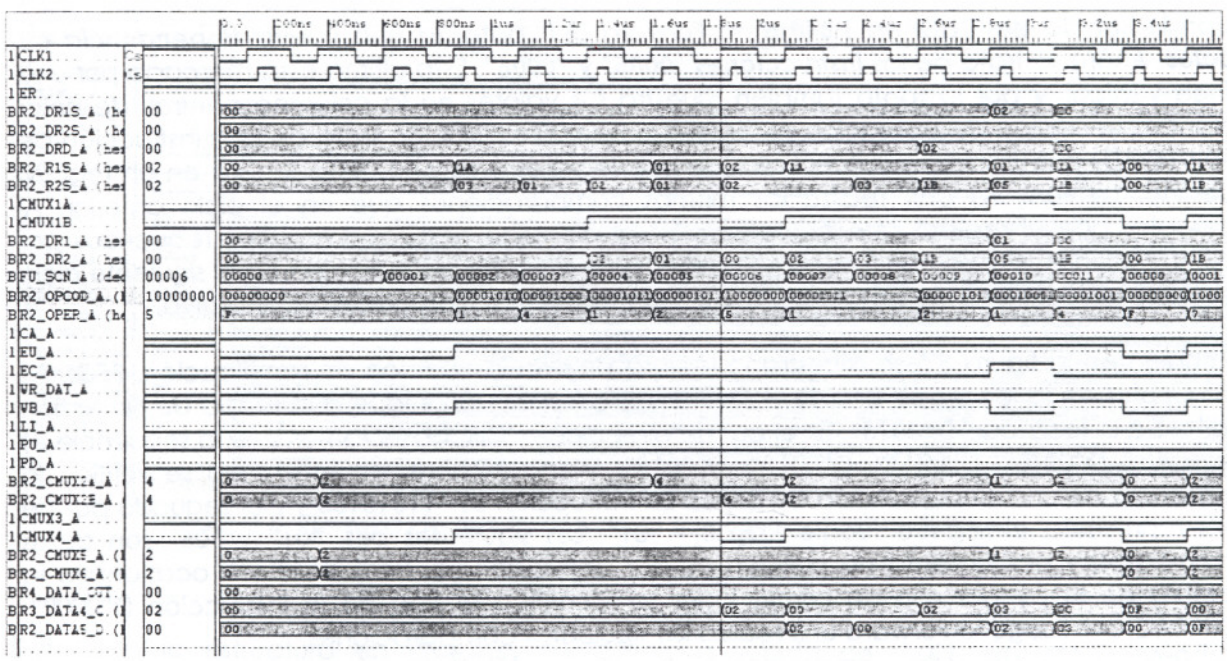


Figura 3.6 Ejemplo de simulación de la Etapa de Decodificación

De igual forma en esta figura visualizamos la Etapa de Decodificación de la instrucción en la ***localidad D"0006"***, en aproximadamente ***1.85μs***. Su código de operación es B"10000000" ***(LOEX)*** y por lo tanto debe leer el contenido de dos registros. Pero ya que tiene una ***dependencia de datos*** con la instrucción previa, realiza la lectura de datos no válidos. Estos

datos son DR1S_A y DR2S_A, y pasan a través de los multiplexores 1A y 1B, para convertirse en DR1_A y DR2_A. El bus OPER_A nos indica que la operación básica equivalente al código de operación es '5' (concatenación). Más abajo encontramos los buses correspondientes a CMUX2A_A y CMUX2B_A, ambos con el valor '4' para resolver en el próximo ciclo del *pipeline* la dependencia de datos, con el valor de DATA4_C.

En la etapa anterior también revisamos la ***instrucción BEQI, D"0010".*** Ahora, en la Etapa de Decodificación la detectamos a los ***2.85μs***. Podemos ver las direcciones de sus registros fuente en los buses R1S_A y R2S_A, "01" y "05", respectivamente. Ya que las líneas CMUX1A y CMUX1B están activas, estos valores constantes pasan a las señales DR1_A y DR2_A. El código de operación es "11"= B"00010001"; le corresponde la operación básica '1'= ADD, ya que posteriormente sumará un offset a la dirección de la memoria de instrucciones. La unidad de control debe identificar BEQI como de tipo RAMIFICACIÓN y activar la línea EC_A que irá al comparador. También desactiva la línea WB_A, pues no se actualizará ningún registro. Las señales CMUX2A y CMUX2B, en la figura 3.6, han propagado su valor a CMUX2A_A= '1' y CMUX2B_A= '0', respectivamente. Con las señales CMUX5 y CMUX6 ocurre lo mismo, ahora CMUX5_A= '1' y CMUX6_A= '0'. El resto de las señales de control, generadas en esta etapa, pasarán al Registro 3.

Continuando con este análisis etapa por etapa de algunas instrucciones, ahora presentamos un fragmento de la simulación ***durante la Ejecución***. Esta etapa puede considerarse como la más compleja dentro del *pipeline* del UAM - RISC II, ya que en este ciclo se resuelve finalmente la dependencia de datos, y además es posible realizar de forma simultánea una operación con la ALU y una comparación.

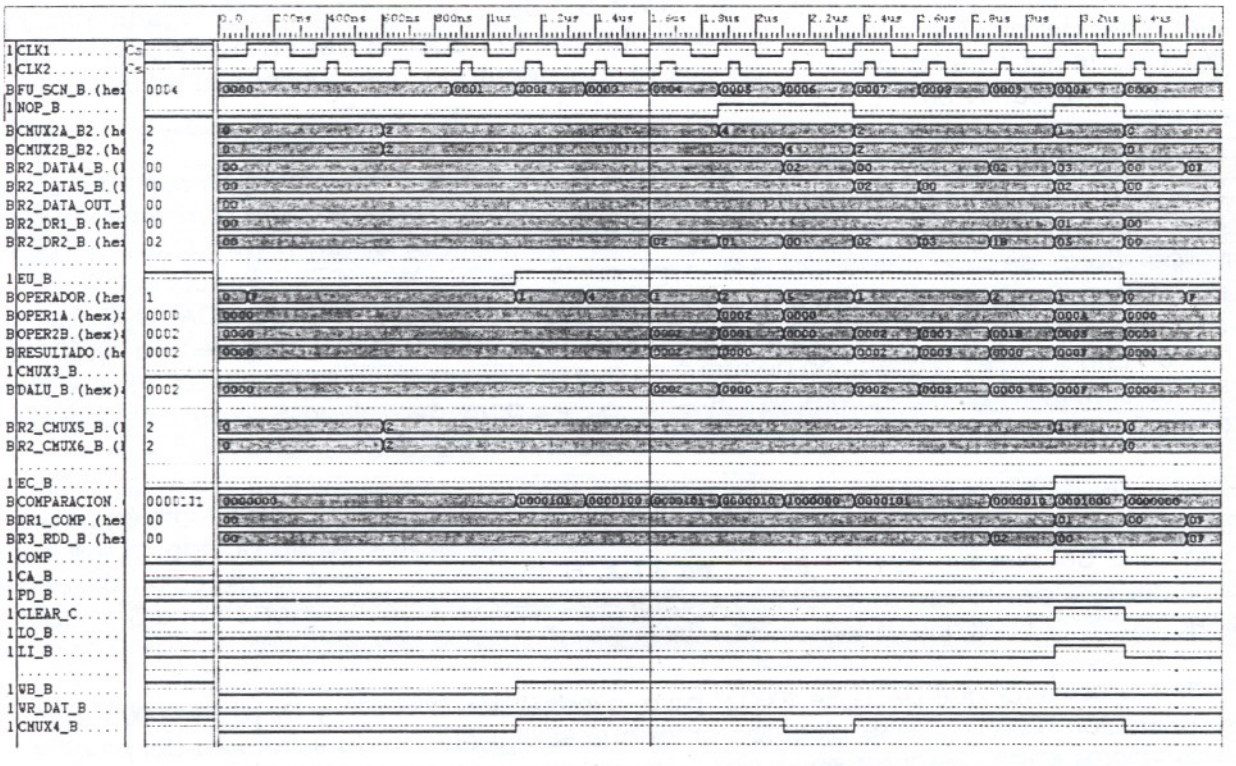


Figura 3.7 Ejemplo de simulación de la Etapa de Ejecución (parte 1)

En esta pantalla del Logic Simulator (Fig. 3.7), encontramos en primer lugar a la dirección de las instrucciones en etapa 3 (SCN_B), seguida por la línea NOP_B, que influirá en la selección

de los multiplexores 2A y 2B cuando lean las señales: DATA4_B, DATA5_B, DATA_OUT_B, SCN_B, DR1_B, DR2_B, y DATA4_C, DATA5_D, DATA_OUT (ver Fig. 3.6). Después encontramos a la señal que habilita la ALU y al operador básico que se aplicará a OPER1A y OPER2B. El resultado entrará al multiplexor 3 junto con dos de las direcciones contenidas en la instrucción. El bus elegido por CMUX3_B pasará a DALU_B. En el siguiente bloque de señales se encuentran las involucradas con el comparador. En la parte inferior de la figura están las señales que posteriormente controlarán la escritura del resultado calculado en esta etapa.

La instrucción con ***dirección "0005", ANDI,*** entra a la Etapa de Ejecución con la subida de reloj que ocurre aprox. a los ***1.85μs***. Desde la detección de ***dependencia de datos*** se activó NOP, que luego se convirtió en NOP_A y en esta etapa es NOP_B. Esta señal será leída por el par de multiplexores que generarán los operandos de la ALU. Las líneas de selección conservan su valor desde la primera etapa: "4" y "2". Cuando el multiplexor 2A lee CMUX2A_B= "4" debe elegir entre DATA4_B y DATA4_C. El valor NOP_B= '1', indica que la dependencia es con la instrucción inmediata anterior ("0004") y por lo tanto se elige DATA4_C, recién calculado: "02" (ver Fig. 3.7, a la derecha de la línea vertical), y se convierte en OPER1A. Para obtener el segundo operando (OPER2B) el multiplexor 2B selecciona a la señal DR2_B, que contiene el valor constante "0001". El resultado de la operación AND es "0000". El mux. 3 tiene su línea de control inactiva y por lo tanto permite el paso de este valor a DALU_B. Posteriormente esta señal actualizará al registro destino, por lo que WB_B y CMUX4_B están activas. WR_DAT_B= '0' indica que no habrá escritura en memoria de datos.

En la etapa anterior, la ***instrucción "0006", LOEX,*** aparecía en 1.85ns. Ahora aparece un ciclo después, ***en 2.1μs***. En el análisis realizado en las etapas previas ya mencionábamos que esta instrucción también tiene una dependencia de datos. Podríamos pensar que la ejecución se retrasaría al tener 2 instrucciones consecutivas con dependencia. Pero esto no ocurre así, la ejecución del *pipeline* continúa normalmente. Esto lo podemos ver en el bus SCN_B, que lleva la dirección de las instrucciones que se van ejecutado, que contiene la secuencia: ..., D"0005", D"0006", D"007", ...

Las señales CMUX2A_B y CMUX2B_B con valor '4' indican que la instrucción con dirección SCN_B= ***D"0006" tiene dos dependencias de datos***; ambas se resolverán en el mismo ciclo de reloj. Considerando que NOP_B está activa, los multiplexores eligen DATA4_C como dato válido para ambos operandos. Al explicar la ejecución de la instrucción "0005" mencionamos que su resultado era "0000". Este pasó a DALU_B y después a DATA4_C. Por lo tanto es el valor que tomarán ambos operandos. EU_B habilita la ALU, que ejecuta "0000" & "0000". Esta suma es la dirección de memoria que se leerá en la siguiente etapa. WR_DAT_B= '0' para permitir dicha lectura. WB_B está activa para que el dato leído actualice el contenido del registro destino.

Sigamos ahora con la instrucción que proviene de la ***localidad D"0010". BEQI*** también activa NOP, y por lo tanto en la Figura 3.6 podemos observar cómo su valor ha *viajado*: al salir del Registro 2 del *pipeline* es NOP_B= '1', ***en 3.1μs***. El multiplexor 1 leerá la dirección de esta instrucción, y por ello su línea CMUX2A_B tiene el valor '1. El multiplexor 2 tomará el dato inmediato DR2_B, ya que así lo indica CMUX2B_B= '2'.

Podemos observar que DALU_B= "0002" para la instrucción D"0007", y DALU_B= "0003" para la instrucción "0008", que se ejecutan antes que BEQI. Debido a la estructura del *pipeline*, estos valores están presentes en la ejecución de esta nueva instrucción, pero ahora como DATA4_B y DATA5_B, respectivamente.

La ALU, habilitada por EU_B= '1', realiza la suma de ambos operandos, SCN_B + DR2_B= "000A" + "0005"= "000F". Esta será la nueva dirección que leerá la unidad de conteo.

También ***se habilita el comparador***, con EC_B= '1'. El bus OPERACIÓN contiene los 7 bits más significativos del código de operación. "0001000" corresponde a comprobar si las 2 cantidades son iguales, es decir, DR1_COMP = RDD_B. Al realizar esta comparación, resulta verdadera, y por lo tanto COMP= '1'.

Enseguida se realiza: (COMP) *OR* (LO_B = '1') OR '0' = '1'. Este valor se asigna a LI_B, y por lo tanto la unidad de conteo tomará la dirección recién calculada para continuar con la ejecución del microprocesador. También el *OR* entre CA_B, PD_B y COMP da como resultado '1', que activa CLEAR_C. En la siguiente subida de CLK1 esta señal ocasionará que se borre el contenido de los Registros 1 y 2. Esto lo podemos ver en las Figura 3.7 y 3.8: hay dos ciclos de reloj a partir de los 3.5µs, donde todas las señales provenientes de las Etapas 1 y 2 son 0's. Debemos subrayar que también el bus de dirección SCN_B tiene durante 2 ciclos el valor "0000" y seguidos por la nueva dirección de ejecución, resultado de este salto condicional (Fig. 3.8, en 3.85*µs*). Ahora la secuencia de instrucciones en el bus de dirección SCN_B es: ..., D"0009", D"0010", D"0015", D"00016", ...

En la parte inferior de la Fig. 3.7 podemos ver que la señal WB_B correspondiente a la dirección "000A" está desactivada, ya que no realizará la escritura en un registro de propósito general. Tampoco requiere la lectura de memoria de datos, por lo que WR_DAT_B = '0'.

Podemos ver otros 2 ejemplos de salto condicional en la Fig. 3.8. El bus de dirección de la Etapa de Ejecución, SCN_B, muestra un salto de D"00017" a D"00021", y después de D"00021" a D"00025". Al pasar de una instrucción a otra hay dos ciclos donde la mayoría de las señales son 0's, ya que CLEAR_C= '1' borró el contenido de los Registros 1 y 2. A partir de esto podemos concluir que cuando se cumple la condición de una instrucción de salto, al cambiar de dirección se pierden dos ciclos de reloj.

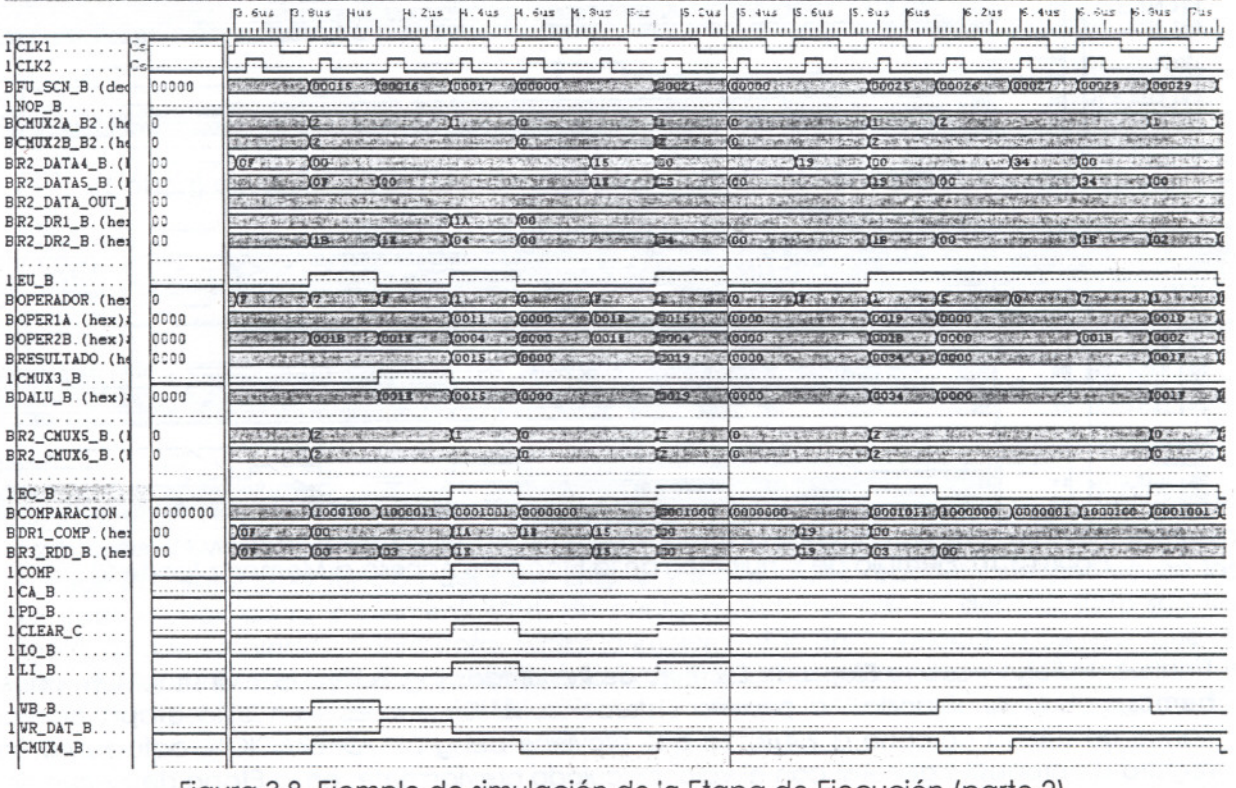


Figura 3.8 Ejemplo de simulación de la Etapa de Ejecución (parte 2)

Para continuar con la ejecución del *pipeline*, ahora analizaremos la ***Etapa 4, de Acceso a memoria***. Como ya se han resuelto las dependencias de datos, en esta etapa no hay evidencia de este tipo de situación. La instrucción número ***D"0005"*** llega a esta etapa aprox. ***a los 2.1μs***. Hemos visto que es ***ANDI***, por lo que en realidad no necesita la memoria de datos. La señal WR_DAT_C permanece en el '0' lógico, ya que no se actualizará ninguna localidad de esta memoria con RDD_C. Aunque tampoco se requiere, el valor leído de la dirección DATA4_C= "00" es DATA_OUT= "00". Por ser de tipo OPERACIÓN, la instrucción ANDI actualizará posteriormente el registro RD_C= "02" y por lo tanto aún conserva el valor WB_C= '1'.

Por su parte, la ***instrucción 6, LOEX,*** lee una localidad de memoria y almacena el resultado en un registro. Podríamos decir que en este momento, ***aprox. 2.4μs***, hemos llegado a la etapa del *pipeline* más importante para este tipo de instrucciones (LOAD). Con WR_DAT_C= '0' habilita la memoria de datos para lectura. El contenido de la localidad DATA4_C= "00" se ve reflejado en DATA_OUT= "00", y posteriormente se escribirá en el registro RD_C= "1F".
En la parte derecha de la Fig. 3.9 aparece la instrucción ***BEQI de la localidad D"0010"***, en el flanco de subida de CLK1 cercano ***a 3.4μs***, y su ejecución continúa en la Fig. 3.10. Después de haber ocasionado un salto de dirección, las señales que provienen de los Registros 1 y 2 son 0's durante dos ciclos del *pipeline*. En el segundo de estos ciclos observamos una señal que no tiene valor '0', es RDD_C= "0F". Esto ocurre ya que esta señal proviene del Registro 3, y por lo tanto no fue borrada. En el flanco de subida de CLK1 que ocurre a los 4.1μs llega a la etapa de Acceso a memoria la instrucción D"0015". Y podemos apreciar que se han vuelto a habilitar las señales WB_C y CMUX4_C, ya que se trata de SRL, un corrimiento lógico a la izquierda.

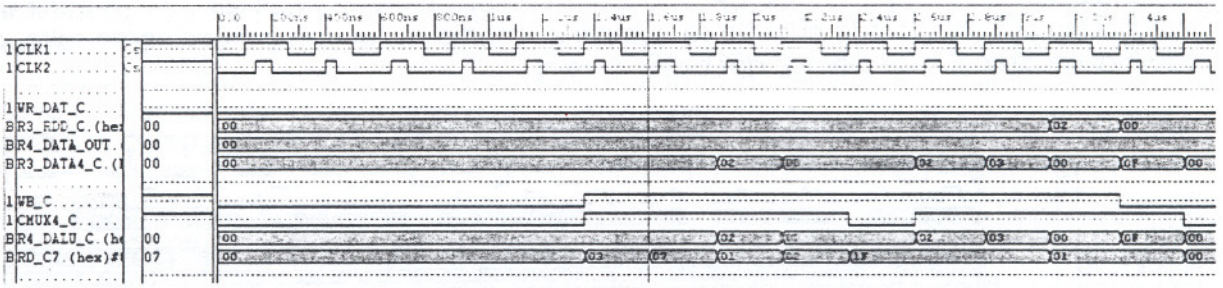


Figura 3.9 Ejemplo de simulación de la Etapa de Acceso a Memoria (parte 1)

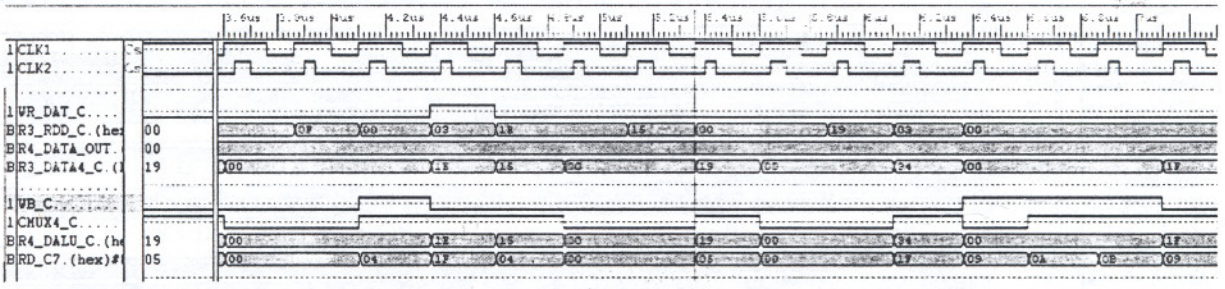


Figura 3.10 Ejemplo de simulación de la Etapa de Acceso a Memoria (parte 2)

Finalmente llegamos a la ***Etapa de Escritura de Resultados*** (Fig. 3.10)con las instrucciones que hemos venido analizando. La primera instrucción, ***ANDI***, aparece en esta etapa con el flanco de subida cercano a ***2.4μs***. Ya que proviene del grupo OPERACIÓN, actualizará un registro de propósito general con el valor calculado previamente, en la Etapa de Ejecución. La dirección de tal registro se encuentra en la señal RD_D= "02". El multiplexor 4 elige cuál

debe ser el dato a escribir, con CMUX4_D= '1'. En este caso es elegido DALU_D, que proviene del resultado del AND calculado por la ALU, y lo asigna a DATA5_D. La línea que habilita la escritura en un registro es WB_D= '1'.

En el siguiente ciclo encontramos a la instrucción que originalmente tenía la dirección ***D"0006" es LOAD***, y en esta etapa también se actualiza el contenido de un registro. Ya que CMUX4_D ha cambiado de valor, su elección será DATA_OUT_D= "00", el bus de salida de la memoria de datos. Ya que la línea WB_D está habilitada, el resultado del multiplexor 4, DATA5_D, será escrito en el registro RD_D= "1F".

La instrucción con dirección ***D"0010", BEQI,*** está en el último ciclo visible de la Figura 3.11. Su ejecución completa la podemos apreciar en la Figura 3.12, a partir de los ***3.6μs***. Como ya hemos dicho, BEQI es una instrucción de salto condicional y por lo tanto no actualizará ningún registro. Esto es evidente con WB_D= '0'. De todas maneras el multiplexor 4 debe elegir un dato, en este caso DALU_D= "0F" transmite su valor a DATA5_D.
Después de esta instrucción también hay dos ciclos con ceros, por las dos instrucciones que se cancelaron al cambiar la secuencia de ejecución. Enseguida aparece la nueva instrucción de tipo OPERACIÓN, con dirección D"0015", SRL. Por esta instrucción en el flanco de subida en 4.6μs el multiplexor 4 elige DALU_D= "1E" y lo asigna a DATA5_D. Con este valor actualiza el registro con dirección "1F".

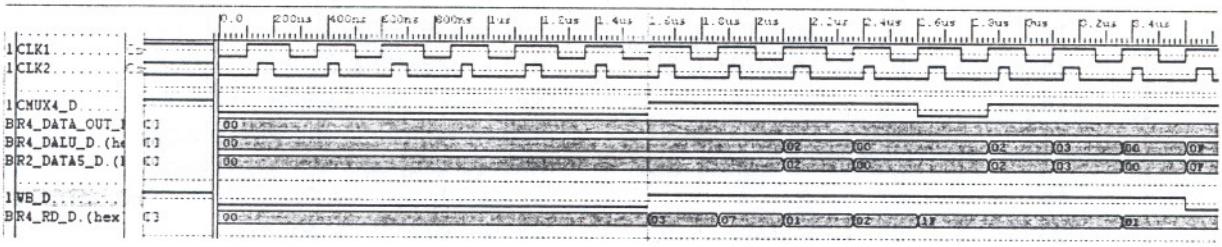
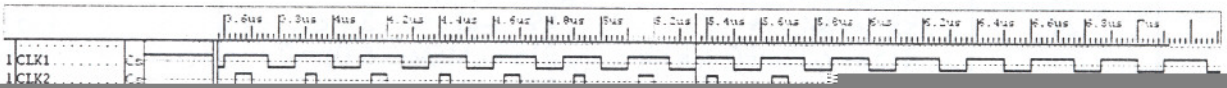


Figura 3.11 Ejemplo de simulación de la Etapa Escritura de Resultados (parte 1)



3.2.4 Validación de resultados

Se probaron 2 programas para validar la ejecución del UAM - RISC II.

Programa 1

No. de instrucción	Función	Resultado
0	-	
1	-	
2	R3 ← R26 + R3	R3 ← 0 + 0 = 0
3	R7 ← R26 xor R1	R7 ← 0 xor 0 = 0
4	R1 ← R26 + 2	R1 ← 0 + 2 = 2
5	R2 ← R1 + 1	R2 ← 2 + 1 = 3
6	R31 ← MEM[R0 & R2]	R31 ← MEM[0 & 3] = 0
7	R31 ← R20 + 2	R31 ← 0 + 2 = 2
8	R31 ← R1 or R3	R31 ← 2 or 0 = 2
9	R3 ← R26 + 1	R3 ← 0 + 1 = 1
10	MEM[R0 & R2] ← R3	MEM[0& 3] ← 1
11	R4 ← R2 or 1	R4 ← 3 or 1 = 3
12	MEM[0 & 30] ← R2	MEM[0 & 30] ← 3
13	R5 ← R4 sll 1	R5 ← 3 sll 1 = 6
14	R31 ← MEM[0 & 3]	R31 ← MEM[0 & 3] = 1
15	R7 ← MEM[R0 & 3]	R7 ← MEM[0 & 3] = 1
16	R6 ← R5 sll 2	R6 ← 6 sll 2 = 24
17	R30 ← MEM[0 & 30]	R30 ← MEM[0 & 30] = 3
18	MEM[R10 & R31] ← R4	MEM[0 & 1] ← 3
19	R7 ← R3 + 1	R7 ← 1 + 1 = 2
20	R9 ← MEM[0 & 30]	R9 ← MEM[0 & 30] = 3
21	R5 ← MEM[R0 & R30]	R5 ← MEM[0 & 3] = 1
22	R8 ← R6 + R7	R8 ← 24 + 2 = 26
23	MEM[R0 & R4] ← R5	MEM[0 & 3] ← 1
24	R9 ← R4 + R9	R9 ← 3 + 3 = 6
25	R15 ← MEM[R0 & R10]	R15 ← MEM[0 & 0] = 0
26	R31 ← R5 + R5	R31 ← 1 + 1 = 2
27	MEM[R0 & 16] ← R4	MEM[0 & 16] ← 3
28	R12 ← MEM[R0 & R30]	R12 ← MEM[0 & 3] = 1
29	R31 ← R12 xor R7	R31 ← 1 xor 2 = 3
30	R11 ← MEM[R0 & 5]	R11 ← MEM[0 & 5] = 0
31	R14 ← R6 + R7	R14 ← 24 + 2 = 26

Tabla 3.2 Descripción del programa 1

Tanto los registros como las localidades de la memoria de datos están inicializados en 0's. Conforme se realizan operaciones puede ir cambiando su valor. Si no hay ninguna operación que los modifique, su valor permanece en 0. A continuación se resumen los resultados obtenidos con el programa 1.

Registro no.	Valor inicial	Valor final
1	0	1
3	0	1
4	0	3
5	0	1
6	0	24
7	0	2
8	0	26
9	0	6
11	0	0
12	0	1
14	0	26
15	0	0
30	0	3
31	0	3

Localidad de mem.	Valor inicial	Valor final
1	0	3
3	0	6
16	0	3
30	0	3

Tabla 3.3 Resultados del programa 1: Registros y localidades modificados

El siguiente conjunto de tres figuras muestra la simulación del programa 1.

En el Simulador Lógico, las dos primeras señales corresponden a CLK1 y CLK2. Analicemos la instrucción leída de la localidad 4. La señal FU_DIR= 4 representa la dirección de la memoria de instrucciones que se lee en la etapa 1 del pipeline. FU_OPCOD= "00001011" es equivalente a una suma, correspondiente a esta instrucción, de acuerdo con la tabla 3.2.

FU_SCN_B contiene la misma dirección, pero dos ciclos de reloj después, es decir, en la etapa de ejecución. También en esta etapa podemos observar al código de operación FU_OPCOD_B y al OPERADOR= "0001", que indica a la ALU la operación básica suma. Sus operandos son OPER1A= 0 y OPER2B=2. En este caso RESULTADO = 0+2 = 2. Este nuevo valor es enviado a DALU_B. Y finalmente, en la etapa de escritura de resultados este valor lo tiene DATA5_D, para actualizar el registro destino.

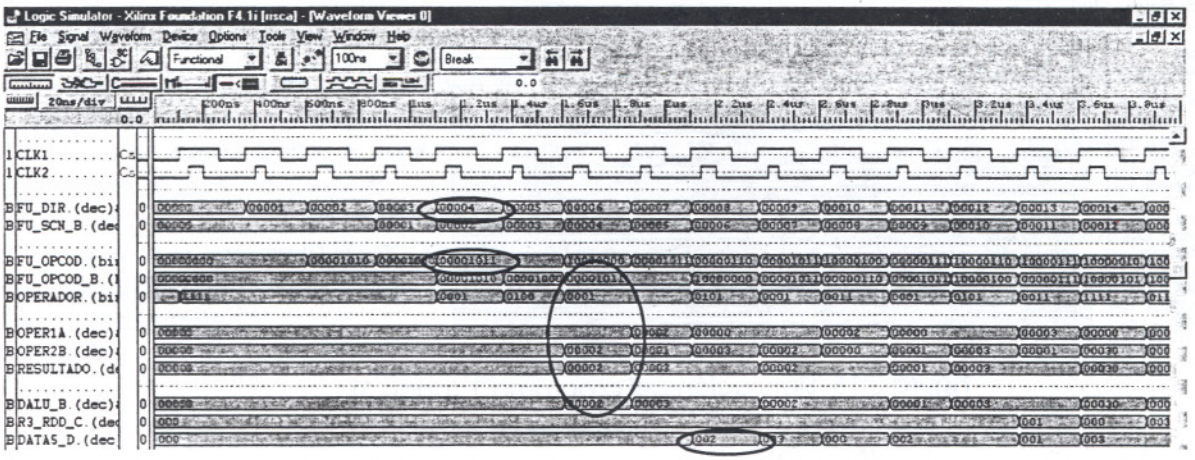


Figura 3.13 Simulación del programa 1 (parte 1)

En la tabla 3.2 la instrucción 23 corresponde a un acceso a la memoria de datos (STOEX). En esta pantalla podemos analizar cómo se ejecuta. De la etapa 3 observamos FU_OPCOD_B= "10000100" y OPERADOR= "0101", correspondiente a una concatenación. La señal RESULTADO = 0 & 3 = 3. Esto indica que se modificará la localidad 3. El nuevo valor para esta localidad es R3_RDD_C= 001, en la siguiente etapa del pipeline, de acceso a memoria.

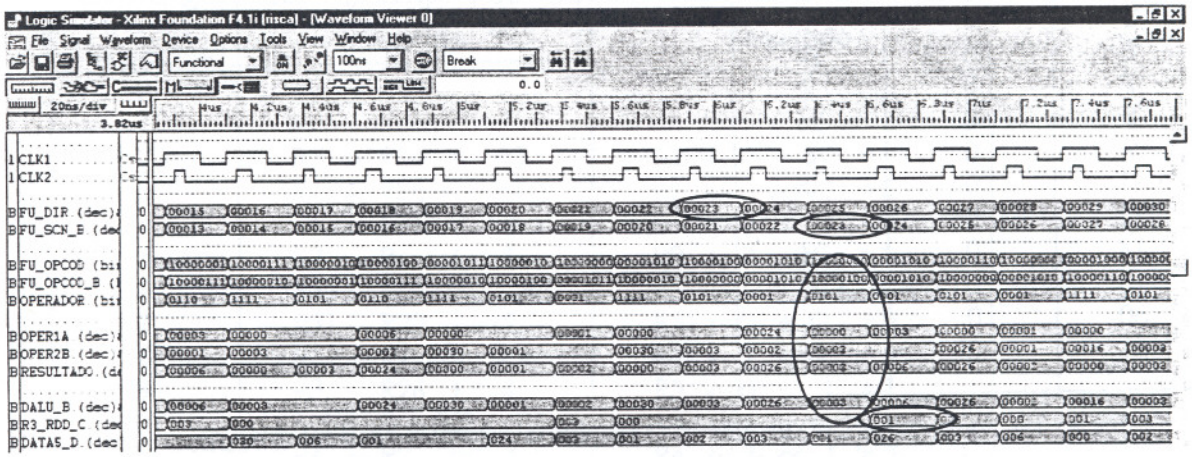


Figura 3.14 Simulación del programa 1 (parte 2)

En esta figura observamos la simulación de las últimas instrucciones del programa 1, destacando la ejecución de la instrucción 31. La ejecución del programa 1 comenzó en el ciclo 1, generando la **instrucción 0** (FU_DIR). El resultado de la instrucción 0 se obtuvo en el ciclo 5, al mismo tiempo que la instrucción 4 entró al pipeline; el resultado de la instrucción 1 se obtuvo en el ciclo 6, y así sucesivamente, hasta llegar a la instrucción 31, cuyo resultado está listo en el ciclo 36, cuando se lee la instrucción 35.

De esta forma probamos la eficiencia de nuestro diseño cuando no hay instrucciones de salto, ya que resolvió todas las dependencias de datos sin detener la ejecución. El pipeline no estuvo ocioso en ningún ciclo y la ejecución de 31 instrucciones terminó en el ciclo 36.

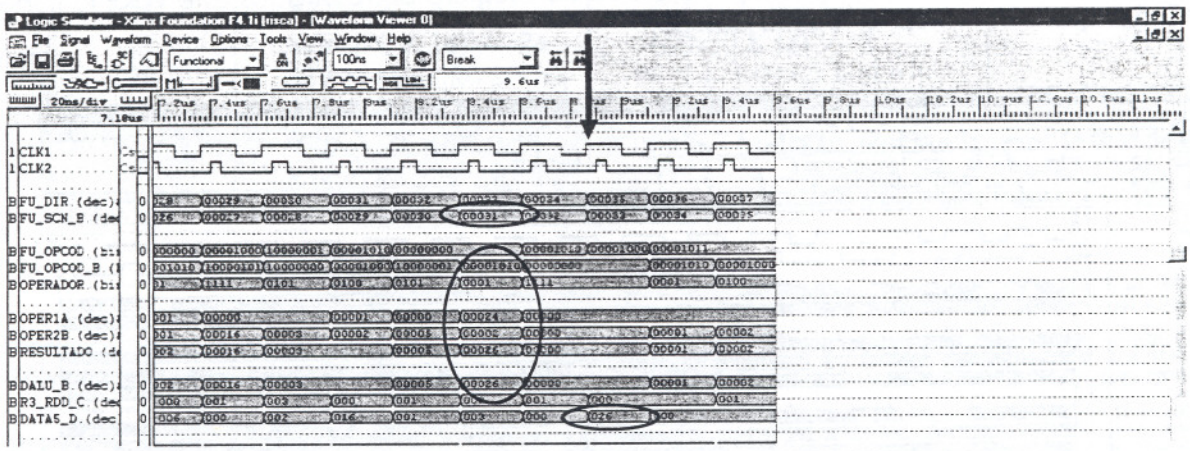


Figura 3.15 Simulación del programa 1 (parte 3)

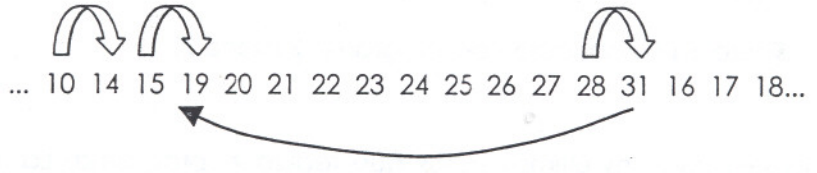
Programa 2

No. de instrucción	Función	Resultado
...		
9	R1 ← R26 and 27	R1 ← 0 and 27 = 0
10	DIR ← 10 + 4	DIR ← 14
11	R4 ← R3 xor 0	-
12	R4 ← R3 xor 0	-
13	R4 ← R3 xor 0	-
14	R4 ← R3 xor 0	R4 ← 0 xor 0 = 0
15	DIR ← 0 & 19; SC_n = 1	DIR ← 19; SC_n = 1
16	R4 ← R3 xor 0	-
...		-
19	R4 ← R3 xor 0	R4 ← 0 xor 0 = 0
20	R4 ← R3 xor 0	R4 ← 0 xor 0 = 0
...		
28	DIR ← 28 + 3	DIR ← 28 + 3
29	R4 ← R4 xor 0	-
30	R4 ← R4 xor 0	-
31	SC_n = 0	SC_n = 0

Tabla 3.4 Descripción del programa 2

Tanto los registros como las localidades de la memoria de datos están inicializados en 0's. Conforme se realizan operaciones puede ir cambiando su valor. Si no hay ninguna operación que los modifique, su valor permanece en 0.

Secuencia de instrucciones ejecutadas por el programa 2:



Enseguida se muestra la simulación del programa 2. En la primera figura vemos la instrucción 10 (JUMP), que en su etapa de ejecución calcula la nueva dirección a la que saltará. Los dos ciclos con ceros entre el 10 y el 14 reflejan que se han borrado las dos instrucciones que entraron al pipeline después de la instrucción 10, y que ya no serán ejecutadas.

La figura 3.17 exhibe la llamada a una subrutina. La instrucción con dirección 15 (CALL) pasa la ejecución a la dirección 19 (en DALU_B), dejando ocioso al pipeline por 2 ciclos, igual que en el caso anterior.

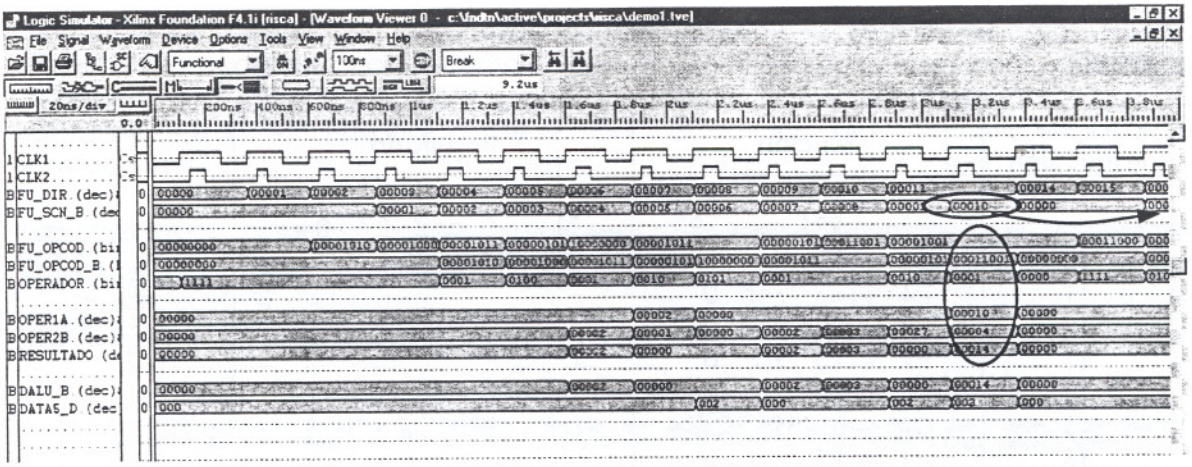


Figura 3.16 Simulación del programa 2 (parte 1)

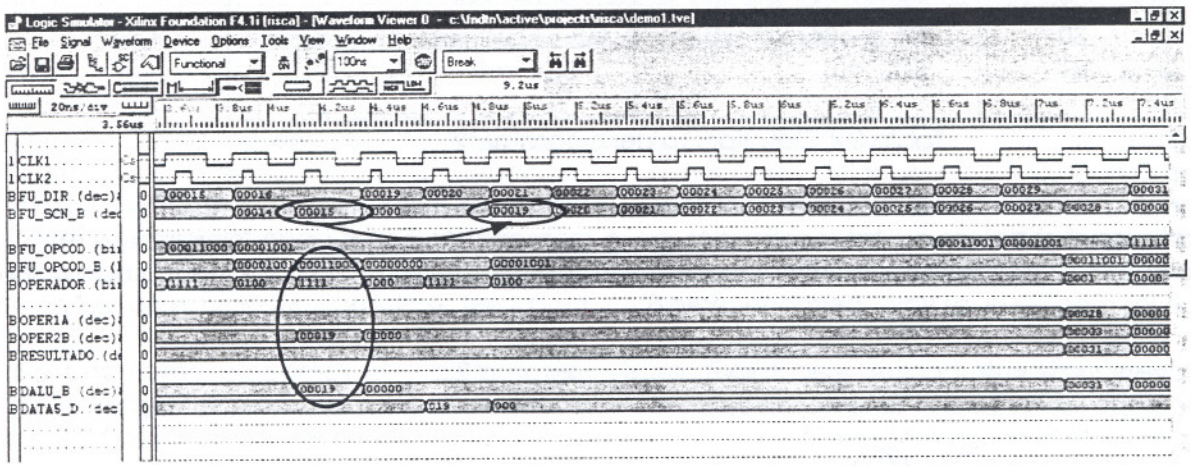


Figura 3.17 Simulación del programa 2 (parte 2)

En la figura 3.18 observamos los últimos saltos que realiza el programa. La instrucción 28 corresponde a JUMP. La nueva dirección se calcula sumando a la dirección actual un offset: 28+3= 31, lo que provoca que las instrucciones 29 y 30 no se ejecuten.

La primera vez que se ejecuta la instrucción 31 no se llega al término del programa, ya que se trata de un RETURN que lleva la ejecución a la dirección que sigue del CALL (ver figura 3.17). Esta nueva dirección es la 16. A partir de esta instrucción continuará la ejecución, hasta llegar nuevamente a la instrucción 31, que esta vez no tendrá ningún efecto sobre la secuencia del programa.

Con el programa 2 se redujo la eficiencia del pipeline, ya que estuvo ocioso en 8 ciclos (correspondientes a 4 saltos) y la ejecución de sólo 24 instrucciones terminó en el ciclo 36.

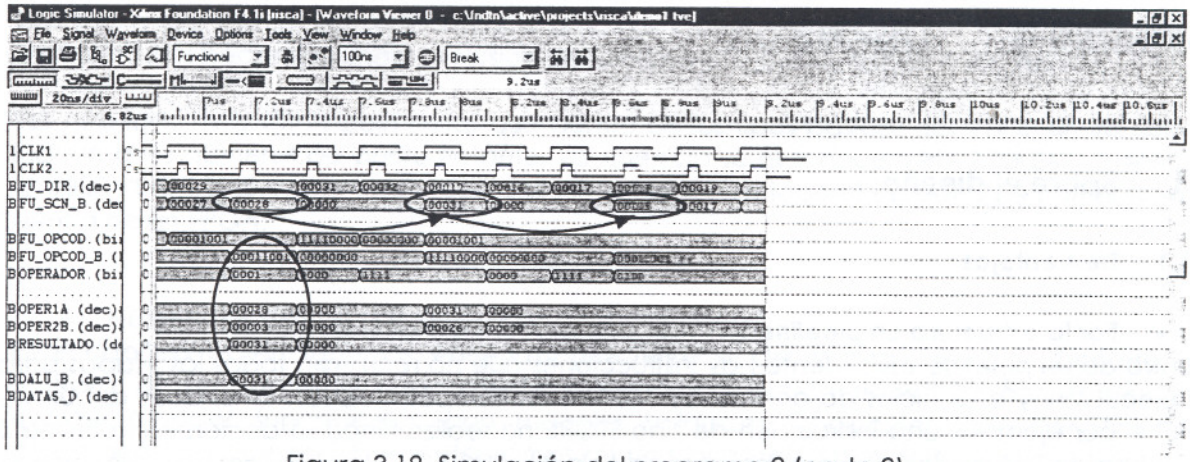


Figura 3.18 Simulación del programa 2 (parte 3)

Capítulo 4 Discusión

4.1 Sobre el diseño

4.1.1 Desventajas

- **Por la estructura en *pipeline***

Dentro de un *pipeline* usualmente hay instrucciones que no requieren todas las etapas para su ejecución, y esos ciclos de ocio se convierten en tiempos muertos. En el caso del UAM - RISC II, por ejemplo, una instrucción de tipo STORE no realiza ninguna acción en la Etapa 5 (Escritura de Resultados); las instrucciones de tipo RAMIFICACIÓN prácticamente terminan su ejecución con la Etapa 3 (Ejecución). Esto puede verse como una desventaja, ya que todas las instrucciones deben recorrer cada etapa del *pipeline*, y no permiten el paso a nuevas instrucciones hasta que hayan pasado por las 5 etapas. Sin embargo, consideramos que la ejecución en *pipeline* beneficia el performance, ya que este tipo de estructura busca minimizar la sub-utilización de los diferentes elementos del microprocesador y por lo tanto del FPGA, permitiendo el paralelismo para trabajar con varias instrucciones a la vez.

- **Por las instrucciones de tipo RAMIFICACIÓN**

En cuanto a los saltos condicionales o ramificaciones, en el UAM - RISC II no se desarrolló ninguna manera de predecirlos. Realizar estas mejoras de forma estática o dinámica podría tener resultados interesantes, con un efecto real sobre el performance. El propósito de la predicción es tener lista la nueva instrucción a la que se saltará, para evitar retrasos, lo que implicaría hardware adicional; por ejemplo: duplicar al menos los elementos pertenecientes a la Etapa de Lectura (excepto la memoria de instrucciones), así como de la Etapa de Decodificación y Lectura de Operandos, incluyendo a los Registros 1 y 2. Cuando es de forma estática se considera la probabilidad de que el salto sí se ejecute, de acuerdo con la frecuencia de aparición de este tipo de instrucciones en el listado del programa; este método no es necesariamente efectivo. Para realizar una predicción dinámica sería necesario contar con elementos de memoria auxiliares donde almacenar el comportamiento de cada una de las ramificaciones que va ejecutando el *pipeline*, para así "suponer" si en la próxima instrucción de este tipo efectivamente se realizará un salto.

4.1.2 Perfeccionando el diseño

- **ALU y comparador**

Durante la Etapa de Ejecución es de vital importancia la función de la ALU. La unidad funcional que programamos para el UAM - RISC II realiza únicamente operaciones de punto fijo y con números positivos. Esto se extiende a las señales manejadas por el comparador. Luego, podríamos considerar como una posible mejora el ampliar estas capacidades para manejar cantidades negativas. Para trabajar con estos últimos habría que agregar un bit para manejar el signo, lo que traería modificaciones obviamente en el tamaño de los buses, además de cambios en las operaciones o comparaciones a realizar entre ellos. Ya que el diseño no incluye una unidad de cálculo que realice operaciones complejas como multiplicación o división, no se hizo necesario el manejo de negativos.

Hasta el momento, todas las señales que el diseño utiliza son de tipo STD_LOGIC. Y únicamente se utiliza el complemento a 2 para realizar la sustracción, aunque esta operación está

limitada a dar como resultadonúmeros positivos y cero. En el caso de las operaciones de corrimiento es necesaria una conversión de tipos, de STD_LOGIC_VECTOR a BIT_VECTOR.

Por otro lado, hay que hacer notar que dentro de los tipos básicos del VHDL **sintetizable** no incluye a los números de punto flotante, ya que sólo se reconocen: BIT, BOOLEAN, CHARACTER, INTEGER y STD_LOGIC. Por lo tanto resulta difícil trabajar con todo el espectro de números reales dentro de un FPGA.

- **Número de unidades funcionales**

En el capítulo 1 mencionamos, entre las características del microprocesador diseñado, que se trata de un procesador escalar. Se podría modificar su estructura para obtener un microprocesador superescalar, con múltiples unidades funcionales. Esto beneficiaría el desempeño, ya que podría ejecutarse más de una instrucción por ciclo de reloj. Los cambios no serían únicamente en el número de unidades funcionales, sino en todo el *pipeline*, ya que esto afectaría a todas las etapas.

- **Número de señales de reloj**

Como ya hemos mencionado, nuestro diseño posee dos señales de reloj. CLK2 se hace necesario para identificar las dependencias de datos y para resolverlas. Sin embargo, el uso de dos señales de reloj también tiene contras:

- Reduce la portabilidad del diseño, pues podría llevarse solamente a otro modelo de FPGA con la capacidad para manejar más de una señal de reloj.
- El basar la correcta ejecución del procesador en el desfasamiento de ambas señales de reloj puede no ser totalmente confiable, ya que una vez llevado el diseño a un FPGA puede llegar a estar sujeto a interferencias o ruido del mundo real que alteren la frecuencia de alguna señal, incluyendo a los relojes.

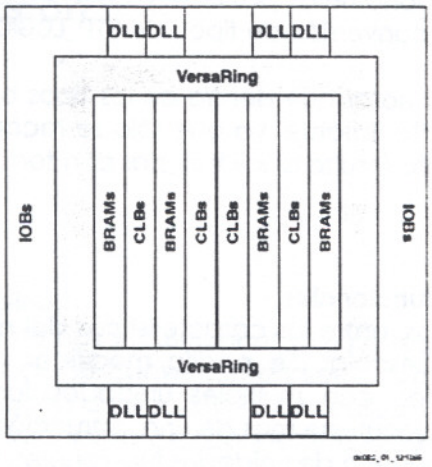
Una transformación del diseño podría enfocarse a crear retardos sobre la primera señal de reloj, CLK1, mediante elementos de hardware, tales como búfers. Sin embargo esto tiene una desventaja, ya que la segunda señal del reloj estaría sujeta al retraso específico del búfer en cuestión. Esto haría al diseño dependiente de la tecnología, lo que también limitaría su portabilidad.

También sería conveniente acelerar la lectura de la memoria de datos y de instrucciones que, como ya hemos visto (sección 2.2.1), son determinantes del desfasamiento de la segunda señal de reloj.

- **Memoria BRAM**

Una forma eficaz de acelerar el acceso a memoria es el uso de BRAM *(Block SelectRAM)*, que se encuentra disponible en los FPGA de la familia Virtex de Xilinx [26]. Se trata, como su nombre lo dice, de bloques de memoria RAM, su tamaño es 4096 bits y se encuentran distribuidos en varias columnas a lo largo del FPGA.

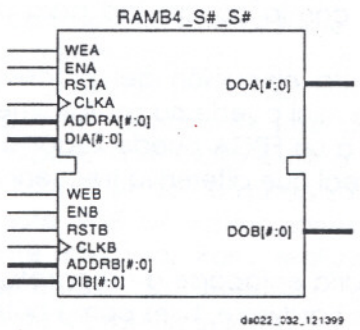
Este tipo de memoria puede realizar la lectura y escritura de datos en un solo ciclo de reloj. Existen además bloques de memoria dual, con dos puertos que pueden ser configurados de forma independiente y además con tamaños de bus diferentes.



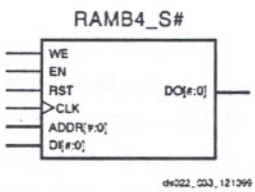
Virtex-E Architecture Overview

Figura 4.1 Distribución de BRAM dentro del FPGA [26]

Encontramos varios modelos de BRAM (o RAMB) dentro de las bibliotecas de Virtex. Por lo tanto, cuando se está creando un diseño en el Schematic Editor es posible llamar uno de estos módulos para incluirlo en nuestro proyecto.



Dual-Port Block SelectRAM+ Memory



Single-Port Block SelectRAM+ Memory

Figura 4.2 Modelos genéricos de RAMB con uno y dos puertos [26]

El emplear memoria de este tipo lógicamente sería benéfico para el desempeño de nuestro microprocesador, ya que podríamos accesar más rápidamente a las memorias de datos y de instrucciones. Además, se reduciría el número de CLBs ocupados por el diseño, ya que actualmente se utiliza gran número de ellos para implementar estas memorias. Por lo tanto quedarían más recursos disponibles dentro del FPGA, lo que podría permitirnos implementar funciones adicionales en nuestro diseño.

- **Manejo de Interrupciones**

En el UAM – RISC II se ha diseñado únicamente el núcleo de un microprocesador que, además del CPU, contiene una memoria de datos. En esta etapa del proyecto no se consideró el manejo de interrupciones, ya que sólo se buscó diseñar y probar la función de los dispositivos básicos del microprocesador, sin considerar su posible interacción con elementos externos al FPGA. Sin embargo, como parte de un proyecto futuro podrían agregarse líneas de comunicación con periféricos para el manejo de interrupciones, así como los controladores correspondientes.

Es importante resaltar que este núcleo de un microprocesador RISC puede ser visto como una función, una caja negra. Esto significa que posteriormente podría convertirse en la base de diferentes microcontroladores, al combinarse con otros dispositivos dentro de un FPGA de mayor tamaño. Así, podemos considerar que esta es la aportación más importante de este proyecto de tesis: contar con un microprocesador en VHDL que, con las modificaciones pertinentes, podría trabajar con otros elementos tales como temporizadores, puertos, sensores, etc. con el propósito de obtener un microcontrolador que pueda emplearse en el control de procesos, en un robot en línea de producción, en control de calidad, en comunicación de datos, etc.

4.2 Sobre los FPGAs

- Las limitaciones de tamaño del UAM - RISC II están dadas por el modelo de FPGA seleccionado, dentro de la familia XC400XL. El diseño es fácilmente portable a un modelo de otra familia. En tal caso tal vez sería posible implementar todo el espacio direccionable de la memoria de datos o de Instrucciones, o incrementar el número de Registros de Propósito General o de contadores de subrutinas.

  A continuación se muestra una comparación entre el modelo utilizado actualmente y uno mayor, de la familia Virtex.

Parámetro	XC4085XL	XCV1000
Compuertas lógicas	85,000	1,124,022
Celdas lógicas	7,448	27,648
Max. bits para RAM	100,352	393,216
No. de Flip Flops	7,168	
Max. no. de I/O del usuario	448	514
Matriz de CLBs	56 x 56	64 x 96
Total de CLBs	3,136	6,144

Tabla 4.1 Tabla comparativa de las familias XC400XL y Virtex [24]

- Para explicar el gran potencial que tiene un FPGA para incrementar drásticamente la velocidad de ejecución de algunos procesos, en [6] se comparan un procesador y el FPGA XC4085XL, fabricados con la misma tecnología (0.35 micron CMOS). La ALU del procesador tiene un desempeño máximo teórico de 128 operaciones de un bit en 2.3 segundos, lo que equivale a 55.7 operaciones por nseg. El FPGA tiene 3,136 CLBs y puede alcanzar un ciclo de reloj de 4.6ns. En términos simples, podríamos decir que cada CLBs equivale a una operación de 1 bit de una ALU. Entonces el FPGA tendría un poder de cómputo de 3,136 operaciones de un bit en 4.6ns, ó 682 operaciones de un bit por nseg. Esta comparación no menciona si todas las operaciones ejecutadas son útiles ni los factores que pueden reducir el desempeño del FPGA. Sin embargo, es

útil para ilustrar que un FPGA puede obtener más capacidad de cómputo de un chip de silicón que un procesador. Sin embargo, cuando se trata del diseño de un *pipeline* nos encontramos con un gran problema, la dificultad de adecuar el *pipeline* al FPGA para conservar dicho rendimiento. Esto se debe a que se originan grandes retrasos debido a las interconexiones dentro del FPGA.

- En el artículo "*PipeRench: A Reconfigurable Architecture and Compiler*" [8] se mencionan algunas características de los FPGAs, percibiéndolas como desventajas:
  - Están diseñados para aplicaciones lógicas, no para aplicaciones de multimedia. En este punto cabe mencionar que en el sitio web de Xilinx [21] se habla de aplicaciones de los FPGAs en nuevas ramas de la tecnología, tales como *Bluetooth*.
  - Para cambiar las características de un FPGA, es decir, para convertirlo en otro chip diferente, es necesario rediseñarlo o recompilarlo.
  - En un FPGA se puede sólo se pueden implementar *kernels* o núcleos de microprocesadores de tamaño fijo y relativamente pequeño.
  - Para la implementación del núcleo de un microprocesador, que comprende la fases de síntesis y PAR, se requiere un tiempo muy largo.
  - El tiempo necesario para configurar un chip puede ser de hasta cientos de milisegundos, lo que limita su aplicabilidad.

- Se ha buscado reducir el tiempo de configuración de los chips. Una de las propuestas es incluir en el FPGA un pequeño caché que almacene una o varias configuraciones extra, para que ya no se tenga que alimentar de forma externa al FPGA. El inconveniente que presenta esta solución es que ocupa un área del FPGA que podría destinarse a la lógica, en lugar de utilizarse sólo como memoria.

  Investigadores de Caltech proponen el uso de una memoria holográfica [18] que almacene diferentes configuraciones (incluso cientos de ellas), de forma que el dispositivo se reconfigure leyendo uno de los hologramas. Con esto se reduciría drásticamente el tiempo de configuración, a decenas de microsegundos. Un OPGA *(Optically Programmable Gate Array)* también contiene bloques de lógica e interconexiones, tal como los FPGAs. Este tipo de dispositivos es una versión mejorada de un FPGA, que utiliza una memoria holográfica. La configuración puede realizarse por un medio óptico, o incluso de forma electrónica, como en los FPGAs. Ya que éstos se han aplicado con éxito al procesamiento de imágenes y al reconocimiento de patrones, los OPGAs podrían ser útiles también en el procesamiento de video en tiempo real. Otra de sus aplicaciones sería realizar búsquedas en bases de datos.

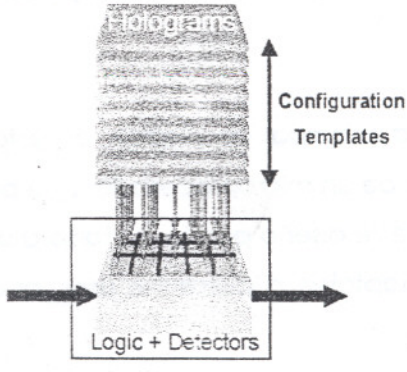


Figura 4.4 OPGA [18]

Conclusiones

Se desarrolló un RISC en VHDL y se implementó en un FPGA de Xilinx. Para llegar a ello fue necesario hablar de las características de los microprocesadores RISC, enfatizando en el paralelismo. Éste puede influir positivamente en la velocidad de procesamiento, de ahí su importancia. En este caso hablamos de paralelismo ya que se ha diseñado y programado un *pipeline* de 5 etapas, en el que a partir de la 5ª instrucción leída, permanentemente hay 5 instrucciones ejecutándose de forma simultánea, y por lo tanto se obtiene un nuevo resultado con cada ciclo de reloj.

También hablamos del cómputo reconfigurable, lo que conlleva el tema de los FPGAs. Un elemento de gran importancia en el desarrollo del cómputo reconfigurable ha sido el avance tecnológico. Los FPGAs han evolucionado en pocos años, para pasar de dispositivos con miles de compuertas a los Virtex actuales, con más de un millón de compuertas. Un FPGA es un dispositivo de arquitectura fija y al mismo tiempo programable, ya que su función no se establece al momento de ser fabricado. Posteriormente se le aplicará una configuración, gracias a los bloques de lógica y a las interconexiones que contiene. En nuestro caso lo configuramos como un RISC, por lo que mencionamos las ventajas de implementar un microprocesador en un FPGA: entre ellas destacan una mayor productividad al permitir fácilmente múltiples actualizaciones y mayor control sobre los datos, ya que durante la simulación se puede conocer en todo momento el valor de cualquier señal. Además hemos citado aplicaciones de los FPGAs en la actualidad, así como las diversas áreas en las que potencialmente serán de utilidad. Ser reconfigurable le da flexibilidad a un sistema, pues puede llegar a adaptarse a los requerimientos exactos, o incluso ajustarse a incesantes cambios. Esto hace atractivo el cómputo reconfigurable en el ámbito académico y especialmente en el industrial, ya que además puede ayudar a reducir costos y tiempos de diseño.

Podemos hablar aquí de la principal aportación de este proyecto. Con el UAM - RISC II contamos con un el núcleo de un microprocesador que puede tener diversas aplicaciones. Por estar descrito en VHDL, este diseño podría portarse a un FPGA más grande y convertirse en un microcontrolador adaptable a diferentes usos, al combinarse con otros módulos o dispositivos digitales.

Para programar o configurar un FPGA es necesario primero describir su comportamiento, ya sea de forma esquemática o con un código fuente en un lenguaje sintetizable. En este caso empleamos VHDL, un estándar de la IEEE. Durante todo el proyecto empleamos el ambiente de desarrollo de Xilinx Foundation 4.1, el cual incluye herramientas para diseño, síntesis, implementación y simulación.

En la tesis se incluye una descripción detallada de cada tipo de instrucción del microprocesador, así como de todos sus elementos. Cada dispositivo fue programado y su funcionamiento probado. Más tarde se procedió a la integración del *pipeline*, lo que nos llevó a una etapa de pruebas exhaustiva, para sincronizar todo el conjunto, incluyendo los 4 registros del *pipeline*. Cuando se logró ajustar los tiempos de todos los buses y líneas, se procedió a resolver el problema de la dependencia de datos, lo que nos llevó nuevamente a un sinnúmero de pruebas en el simulador de Xilinx. Finalmente se alcanzó la meta de eliminar los retrasos en todos los casos de dependencia de datos, siendo necesarios únicamente con instrucciones de bifurcación.

Se presentan diversas pantallas del simulador que muestran el funcionamiento del microprocesador, y se analiza minuciosamente la ejecución de diferentes instrucciones en cada una de las etapas del *pipeline*.

Como punto final hablamos del trabajo que potencialmente se puede desarrollar a partir de este proyecto. También hablamos del futuro de los FPGAs, que lógicamente influye en la evolución del cómputo reconfigurable.

Bibliografía

[1] Armstrong, James

"Introduction to VHDL", Video from The Distinguished Lecture Series.
Virginia Polytechnic Institute. IEEE/Design Automation Technical Committee (DATC) & ACM Special Interest Group on Design Automation (SIGDA).
USA, 1991.

[2] Bielby, Robert

Sr. Director, Strategic Solutions, Xilinx, Inc.
"Digital Convergence Demands Reprogrammability"
Xcell journal Issue 42,
USA, 2002

[3] Chang, Morris J. and Agun, Kagan S.

"On Design-For- Reusability in Hardware Description Languages"
Illinois Institute of Technology.
USA, 2000.

[4] Collins, Carlis

Managing Editor of Corporate Communications, Xilinx,
XCell Journal 31 - 1Q99
http://www.xilinx.com/xcell/xl31/xl31_4.pdf
USA, 1999

[5] Davidson, Jacob

"FPGA Implementation of a Reconfigurable Microprocessor"
IEEE 1993 Custom Integrated Circuits Conference

[6] De Hon, André

"The Density Advantage of Configurable Computing"
California Institute of Technology
Computer, vol. 33, issue 4, pp. 41-49.
USA, 2000

[7] De Luca A., Rojas L., Tellez V.

"Sistemas Digitales"
Universidad Autónoma Metropolitana,
México, 1993

[8] Goldstein S., Schmit H., Budiu M. et al.

"PipeRench: a Reconfigurable Architecture and Compiler"
Carnegie Mellon University
Computer, vol. 33, issue 4, pp. 70-77.
USA, 2000

[9] Guccione, Steve

"List of FPGA-based Computing Machines"
IOCOM Corporation
http://www.io.com/~guccione/HW_list.html

[10] Hwang, Kai

"Advanced Computer Architecture : Parallelism, Scalability, Programmability"
McGraw-Hill Higher Education
USA, 1992

[11] Jamonline, Inc.

"JamCam 3.0 Camera"
http://www.jamonline.com/products/jamcam_prod.asp

[12] Jürgen B., Andreas K., Frank–Michael R., Manfred G.

"Perspectives of Reconfigurable Computing in Research, Industry and Education"
Lecture Notes in Computer Science vol. 1482,
"Field-Programmable Logic and Applications", 8th International Workshop, 1998

[13] Khosravipour M., Grünbacheer H.

"VHDL-based Rapid Hardware Prototyping Using FPGA Technology"
Lecture Notes in Computer Science vol. 975, "Field-Programmable Logic and Applications", 5th International Workshop, FPL '95

[14] Kim, Austin and Chang, Morris

"Designing a Java microprocessor core using FPGA technology"
Computing & Control Engineering Journal,
June 2000

[15] Markovic, P. and Mujkovic, V.

"FPGA to ASIC Conversion Design Methodology with the Support for Fast Retargetting to Different CMOS Implementation Technologies"
Proc. 22nd International Conference on Microelectronics, 2000

[16] Meier, Russell

"Rapid Prototyping of a RISC Architecture for Implementation in FPGAs"
Department of Electrical Engineering and Computer Engineering
Iowa State University of Science and Technology,
USA, 1995 IEEE database

[17] Multi Video Designs

"Síntesis Lógica para Componentes Programables. VHDL, muy simple..." Versión 3.0a
Francia, 1998

[18] Mumbru J., Panotopoulos G., et al.

"Optically Programmable FPGA Systems"
Caltech Center for Neuromorphic Systems Engineering,
USA, 2002
http://www.cnsee.caltech.edu/Research02/reports/panotpoulos2full.html

[19] Sharp, Steve

"FPGAs Can Be an Effective Alternative to Mask Gate Arrays"
XCell Journal 30 article,
http://www.xilinx.com/xcell/xl30/xl30_6.pdf
USA, 1998

[20] Stansfield A., Page I.

"The Design of a New FPGA Architecture"
in Lecture Notes in Computer Science vol. 975,
"Field-Programmable Logic and Applications", 5th International Workshop, 1995

[21] Xilinx, Inc.

"Wireless Home Networks — DECT, Bluetooth, HomeRF, and Wireless LANs"
http://www.xilinx.com/publications/whitepapers/wp_pdf/wp135.pdf
USA, 2001

[22] Xilinx, Inc.

"Foundation Series 2.1i In-Depth Tutorials"
USA, 1999

[23] Xilinx, Inc.

"VHDL Reference Guide"
USA, 1999

[24] Xilinx, Inc.

"Xilinx Data Book 1999"
USA, 1999

[25] Xilinx, Inc.

"LogiBlox"
http://www.xilinx.com/products/logicore/lbloxdes.htm

[26] Xilinx, Inc.

"Virtex™-E 1.8 V Field Programmable Gate Arrays"
http://www.xilinx.com/partinfo/ds022-1.pdf
USA, 2002

[27] Xilinx, Inc.

"Xilinx Solutions for Industrial Applications. Emerging Standards & Protocols"
http://www.xilinx.com/esp/ind/collateral/Xilinx_Solutions_for_Industrial.pdf
USA, 2003

[28] Zhang Y. , Kai-Kuang M.

"An Embedded RISC Core Design for Variable Length Coding"
School of Electrical and Electronic Engineering, Nanyang Technological University
Singapur, 2000 IEEE database

[29] Zwolinski, Mark

"Digital System Design with VHDL"
Prentice Hall
USA, 1999

[30] ChipCenter

"Product Review"
http://www.chipcenter.com/pld/products_001-500/pldp394.htm
ChipCenter -QuestLink
USA, 2002