# UNIVERSIDAD COMPLUTENSE DE MADRID

**FACULTAD DE INFORMÁTICA**
**Departamento de Arquitectura de Computadores y Automática**



# ESTUDIO E IMPLEMENTACIÓN DE UN SIMULADOR PARA EVALUAR EL RENDIMIENTO DE MICROARQUITECTURAS ASÍNCRONAS

**MEMORIA PARA OPTAR AL GRADO DE DOCTOR**

**PRESENTADA POR**


**José Manuel Colmenar Verdugo**

Bajo la dirección de los doctores
Juan Lanchares Dávila, Antonio Oscar Garnica Alcázar y José Ignacio Hidalgo Pérez


**Madrid, 2008**


- **ISBN:**

UNIVERSIDAD COMPLUTENSE DE MADRID


Dpto. de Arquitectura de Computadores y Automática

# Estudio e Implementación de un Simulador para Evaluar el Rendimiento de Microarquitecturas Asíncronas

Memoria para optar al grado de doctor presentada por

**D. José Manuel Colmenar Verdugo**

Directores:

**Dr. D. Juan Lanchares Dávila**

**Dr. D. Antonio Óscar Garnica Alcázar**

**Dr. D. José Ignacio Hidalgo Pérez**

Madrid (España), 2008

*A mis padres, José Luis y María del Rosario*

*A Lorena*

# Agradecimientos

Mis únicas palabras en primera persona dentro de esta tesis son para mostrar mi agradecimiento a quienes han hecho posible que este trabajo vea la luz. En primer lugar, quiero dar las gracias a mis directores: Juan Lanchares, espíritu crítico donde los haya, Óscar Garnica, investigación en estado puro, y José Ignacio Hidalgo, necesario equilibrio entre los anteriores y mi primer mentor en el mundo de la investigación. Para mí, forman un equipo que se complementa a la perfección. Su trabajo y esfuerzo hacia este proyecto ha sido, es, y será, impagable. Para vosotros, mi más profunda admiración.

No podía faltar mi agradecimiento a Román Hermida, cuyo criterio, consejos y directrices han sido fundamentales en mi trabajo. Gracias también a Francisco Tirado, investigador principal de los proyectos en los que he participado durante mi doctorado. Mis compañeras Sonia López, Guadalupe Miñana y Noelia Morón, también merecen mi reconocimiento por su inestimable colaboración.

A lo largo de estos años de docencia en el C. E. S. Felipe II he compartido mi tiempo con muchos y buenos compañeros. Algunos ya pasaron por esto antes que yo, y otros lo harán en un futuro próximo, por lo que *la tesis* siempre ha sido un tema recurrente en nuestras conversaciones. De todos mis compañeros, quisiera agradecer especialmente su ayuda, consejos y críticas a Nuria, Miguel Ángel, Alberto, Josele, Alfredo e Iván. Amigos, ya he vuelto.

También es el momento de recordar que nada de esto sería posible sin la educación, el respeto y la honestidad que me han inculcado mis padres, José Luis y María del Rosario, y la complicidad de mi hermano, Javi. Espero que, aunque esta tesis os suene más a chino que a cristiano, estéis orgullosos de ella.

Como suele ocurrir en la mayoría de las ocasiones, el agradecimiento más importante se deja siempre para el final, aunque realmente éste sea el más merecido. Sin la total comprensión y apoyo de Lorena este trabajo nunca se habría, siquiera, empezado. Sabes que, aunque a veces he estado ausente, nunca me he ido. Gracias, de corazón.

A todos aquellos que me habéis ayudado, mil gracias.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

Este capítulo inicial de la tesis comienza poniendo de manifiesto los problemas que acarrean los circuitos síncronos en los actuales niveles de integración. Seguidamente se detallan las soluciones que ofrecen los sistemas asíncronos para esos problemas, comentando además las conocidas desventajas de los circuitos carentes de señal global de reloj. A continuación se ofrece una visión general del tipo de circuitos concreto en que se centra esta tesis, los procesadores asíncronos de propósito general. En ese repaso se muestran distintos ejemplos de productos comerciales surgidos en los últimos años, mostrando así el creciente interés de la industria por este tipo de sistemas. Para finalizar, se describe el objetivo principal de esta tesis, dividido a su vez en varios objetivos secundarios.

## 1.1. Actualidad en el diseño de circuitos síncronos

Hoy en día, finalizando ya la primera década del siglo XXI, los avances en la tecnología y proceso de fabricación permiten que los circuitos síncronos puedan funcionar a elevadas frecuencias, en algunos casos superiores a los tres gigahertzios.

Este aumento de la frecuencia, conseguido gracias a la reducción en el tamaño del transistor CMOS, agrava los tradicionales problemas relacionados con la señal de reloj que los diseñadores de circuitos síncronos deben afrontar.

Los principales problemas a resolver por los diseñadores de circuitos síncronos se pueden resumir en los siguientes puntos:

- Desfases en la señal de reloj: los circuitos síncronos deben integrar mecanismos contra el *clock skew* puesto que la red de distribución de la señal de reloj resulta cada vez más extensa y compleja. La penalización debida al *clock skew* puede alcanzar el 15 % del retardo de propagación del reloj en la red de distribución, llegando a representar el 20 % del tiempo de ciclo en diseños de grandes procesadores como el *Itanium 2 9000 series* de Intel [Fet06].

- Diferencia entre los retardos de las conexiones y los retardos de las puertas lógicas: los retardos de las conexiones cobran cada vez mayor importancia con respecto a los retardos de las puertas lógicas debido a la reducción del tamaño del transistor. La longitud definitiva de las conexiones no se conoce hasta las etapas finales de ubicación y rutado, lo que puede suponer una dificultad añadida a la hora de estimar el camino crítico de un circuito síncrono, y por tanto su rendimiento.

- Tolerancia a fallos: es necesario construir sistemas tolerantes a fallos cuya funcionalidad no se vea alterada por variaciones en voltaje, temperatura, proceso de fabricación o envejecimiento de los componentes del circuito.

- Elevados consumos de potencia debidos a la red de distribución de la señal de reloj: un porcentaje significativo de la potencia total consumida por los circuitos síncronos se dedica a la distribución de la señal de reloj a todos sus componentes secuenciales. Esta penalización se puede solucionar en cierta medida a través de la implementación de técnicas como el *clock gating* [QPX06], cada vez más extendidas en los diseños síncronos actuales.

- Compatibilidad electromagnética (*EMC*) [EB99]: la actividad de un circuito síncrono es máxima en el instante posterior al flanco de reloj, disminuyendo gradualmente hasta alcanzar una situación estática anterior al siguiente ciclo de reloj. Estos picos periódicos de alta actividad, realizados a elevadas frecuencias, se convierten en una fuente de microondas emitiendo pulsos de potencia que pueden interferir en el funcionamiento de dispositivos ubicados en el entorno del circuito. Además, la cada vez mayor capacidad de integración permite incluir en un *chip* múltiples componentes realizando tareas en

paralelo, por lo que la compatibilidad electromagnética es determinante en los sistemas actuales.

Los circuitos asíncronos representan una alternativa plausible al diseño síncrono que podría paliar los inconvenientes que se acaban de describir. A continuación se analizan las ventajas e inconvenientes de los diseños asíncronos, destacando la problemática de la evaluación del rendimiento de esta clase de sistemas.

## 1.2. Ventajas e inconvenientes de los circuitos asíncronos

Los circuitos asíncronos presentan una diferencia esencial con respecto a los circuitos síncronos: no existe una señal global de reloj compartida por todos los componentes secuenciales del sistema. En su lugar, la señal de reloj se sustituye por un conjunto de señales actuando bajo un protocolo que se encarga de la sincronización, comunicación y secuenciación de las operaciones del circuito. Esta característica diferencial aporta a los circuitos asíncronos una serie de propiedades inherentes que se detallan a continuación:

- *Alto rendimiento* [OB06, CZ05, SRG+01]: en un circuito asíncrono el cómputo de un dato puede comenzar inmediatamente después de que el cómputo anterior haya terminado porque no hay necesidad de esperar a la transición de la señal de reloj. Esta situación conduce, potencialmente, a una ventaja en rendimiento asociada a la variabilidad en los tiempos de cómputo de cada dato, aunque la penalización introducida por el protocolo de comunicación puede llevar a compensar parte de esa ventaja. Sin embargo, al comparar el rendimiento de un circuito síncrono con respecto al equivalente asíncrono, una vez completado el proceso de fabricación, es necesario considerar ciertos detalles sobre el diseño síncrono. Por ejemplo, un circuito síncrono se diseña para funcionar a una determinada frecuencia en un determinado rango de temperaturas y voltajes de alimentación. De hecho, dada la importante diferencia entre el mejor y el peor caso en la actual tecnología CMOS (*best-case*

*and worst-case process corners*), las variaciones en el proceso de fabricación son objeto de una atención específica en recientes diseños [Fet06], y el margen de seguridad (*derating factor*) tiene un peso cada vez mayor. Esto significa que, en condiciones típicas de voltaje y temperatura, muchos circuitos síncronos podrían funcionar a una frecuencia mayor que la que realmente utilizan.

- *Bajo consumo de potencia* [BY06, CG06, KY02, Nie97, vBBK+94]: en general se puede afirmar que los circuitos asíncronos consiguen una reducción en el consumo de potencia frente a circuitos síncronos equivalentes. Las principales causas son las siguientes:

  - La red de distribución de reloj, cuyo porcentaje en el total del consumo de potencia de los circuitos síncronos es muy significativo, desaparece en los circuitos asíncronos.
  - La ausencia de señal de reloj implica que aquellos componentes del circuito asíncrono que no hayan recibido ningún dato de entrada permanezcan en estado inactivo, consumiendo exclusivamente la potencia debida a las corrientes de fuga (*leakage*), o potencia en estático. En los circuitos síncronos, todo componente que reciba la señal de reloj computará, independientemente de haber recibido nuevos datos a procesar.

  Cabe destacar también, en este apartado, que los circuitos asíncronos se adaptan automáticamente a técnicas de reducción de consumo de potencia como las basadas en el escalado dinámico del voltaje [NNSvB94], mientras que los circuitos síncronos deben modificar la frecuencia de la señal de reloj en función de los cambios en el voltaje de alimentación.

- *Alta compatibilidad electromagnética* [PDF+98, vBBK+94]: debido a la ausencia de señal de reloj, el cómputo de los circuitos asíncronos se distribuye en el tiempo siguiendo un patrón aleatorio dependiente de los tiempos de cómputo de los datos y de los retardos debidos al protocolo de comunicación. La energía generada se distribuye de modo más uniforme que en los circuitos síncronos por lo que los circuitos asíncronos son susceptibles de presentar menor ruido y mayor compatibilidad electromagnética.

- *Robustez frente a variaciones de parámetros físicos* [NNSvB94, MBL+89]: las variaciones en el proceso de fabricación, temperatura de trabajo o voltaje de alimentación no afectan al correcto funcionamiento de los circuitos asíncronos. El motivo es que la funcionalidad de este tipo de circuitos se diseña de manera independiente al comportamiento temporal de sus componentes. El aumento en el retardo de los componentes de un circuito síncrono puede desembocar en que el circuito no funcione correctamente. Por el contrario, un circuito asíncrono computa tan rápido como le sea posible bajo las circunstancias de su entorno. Si la temperatura sube o el voltaje cae, el rendimiento disminuirá, pero en condiciones típicas su rendimiento será el esperado, sin ningún margen de seguridad adicional.

- *Modularidad en el diseño* [FB06, MN06, Sut89, Mar86]: la funcionalidad de cada componente de un sistema asíncrono se diseña para trabajar de un modo independiente al comportamiento temporal del circuito. Su comunicación con otros módulos del sistema se realiza a través de un *interfaz* correspondiente al protocolo de comunicación. En ese sentido, cualquiera de los módulos de un sistema asíncrono se puede sustituir por otro que realice la misma tarea e implemente un interfaz para el mismo protocolo.

- *Ausencia de problemas en la distribución de la señal de reloj*: los circuitos asíncronos no necesitan distribuir una señal global que deba llegar a todos los puntos del circuito de manera simultánea, por lo que problemas como el *skew* de la señal de reloj son ajenos al paradigma asíncrono.

Sin embargo, y como era de esperar, no todo son ventajas a la hora de diseñar circuitos asíncronos. Es justo mencionar que los circuitos asíncronos presentan ciertos inconvenientes que obstaculizan su popularización y los convierten en interesantes líneas de trabajo para la comunidad investigadora. Estos inconvenientes, en resumen, son los siguientes:

- Necesidad de controlar el aumento en el área de silicio, tiempo de cómputo y consumo de potencia debido a la lógica de control que implementa el protocolo entre los distintos componentes del circuito asíncrono: la elección del modelo de retardos, *i.e.*, el conjunto de suposiciones acerca del valor

máximo de los retardos de los componentes del circuito, tiene una influencia determinante en los resultados de área, retardo y consumo de potencia del mismo. A menor número de restricciones sobre el retardo de conexiones y componentes del circuito según el modelo de retardos elegido, mayor cantidad de *hardware* será necesario para llevar a cabo la comunicación entre componentes del circuito, afectando negativamente a las características antes mencionadas.

- Falta de herramientas CAD que ayuden en el proceso de diseño: a pesar de que existe una razonable oferta de herramientas y algoritmos de síntesis de circuitos asíncronos como los publicados recientemente en [CKLS06, CCCGV06, SB06], las herramientas de diseño, verificación y generación de tests para circuitos asíncronos están lejos aún de la oferta disponible para los sistemas síncronos.

- Falta de herramientas de simulación de circuitos asíncronos complejos que permitan evaluar las ventajas e inconvenientes derivadas de la aplicación de nuevas técnicas o protocolos asíncronos.

Este último inconveniente es el que atrae la atención del presente trabajo: existe una carencia importante de herramientas de modelado y simulación de circuitos asíncronos complejos. Más concretamente, existe un pequeño número de herramientas de modelado y simulación de procesadores asíncronos de propósito general. En esta tesis se analizan las causas de esta carencia de herramientas, las prestaciones de algunas de las existentes, y se plantean soluciones a los problemas relacionados con el modelado y simulación de procesadores asíncronos de propósito general.

Con el ánimo de ilustrar la notoriedad de los procesadores asíncronos, en el siguiente apartado se describen algunos ejemplos de diseños de procesadores asíncronos recientes, mostrando así el creciente apoyo de la industria y la comunidad universitaria a este tipo de circuitos.

## 1.3. Procesadores asíncronos

La investigación sobre circuitos asíncronos comenzó a mediados de los años cincuenta del siglo XX [MB57, Mul63], pero no fue hasta la última década del siglo cuando la industria y el mundo académico apreciaron las excelentes ventajas de los diseños asíncronos, reflejando este hecho en forma de nuevos artículos y productos en el mercado, como los referenciados en el apartado anterior.

Dado que, como se verá más adelante, el objetivo principal de esta tesis está íntimamente relacionado con el área de los procesadores asíncronos de propósito general, en este apartado se muestran algunos ejemplos de procesadores asíncronos recientes, presentados siguiendo el orden cronológico de sus publicaciones relacionadas.

### MiniMIPS

En el año 1997, el grupo *Asynchronous VLSI* de *Caltech* publica el diseño del MiniMIPS [MLM+97], una versión asíncrona del procesador MIPS R3000. Consiste en un sencillo procesador RISC de 32 bits sin especulación ni predicción de saltos. Este procesador consiguió un rendimiento hasta dos veces y media mejor que procesadores contemporáneos síncronos como el StrongARM [WM96]. En la Figura 1.1 se muestra el circuito después de su fabricación.

### Familia AMULET

El diseño asíncrono comercial más conocido en el ámbito de los procesadores de propósito general quizá sea la familia de procesadores AMULET, desarrollada por el *Advanced Processor Technologies Group* en la Universidad de Manchester. Esta familia está formada por una saga de procesadores asíncronos compatibles con el repertorio de instrucciones de los procesadores síncronos ARM. El principal objetivo de los procesadores AMULET fue, desde un principio, minimizar el consumo de potencia.

El último de los procesadores de esta familia que vio la luz, el AMULET3 [FGG98, GFC99], constituye una importante evolución sobre los modelos anteriores de la

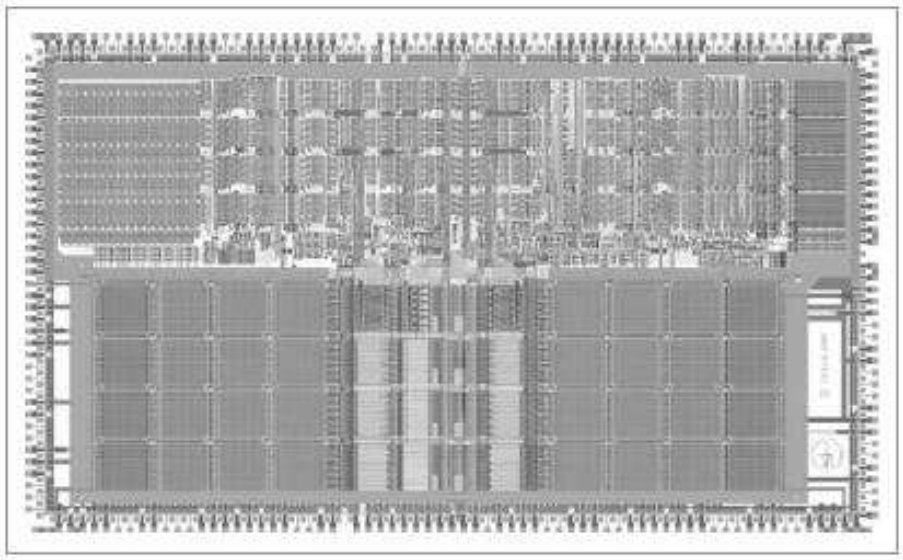

Figura 1.1: *Layout* del procesador MiniMIPS (1997) [MLM+97].

saga. Se trata de un procesador RISC de 32 bits con *pipeline* de cinco etapas, ejecución fuera de orden con predicción de saltos y *reorder buffer*. Es compatible con el repertorio de instrucciones ARM9 y sus prestaciones son muy similares al modelo síncrono equivalente de ARM, el ARM9TDMI, con la ventaja de su mayor compatibilidad electromagnética y su robustez ante variaciones de parámetros físicos.

El destino del AMULET3 era convertirse en un núcleo empotrable en sistemas más complejos. En el año 2000, el grupo de Manchester publica el diseño AMULET3i [GBB+00], un *SoC* (*System-on-Chip*) que incluía un procesador AMULET3, una memoria RAM de 8 Kbytes, una ROM de 16 Kbytes, un bus asíncrono interno MARBLE [BF98] y un interfaz para dispositivos síncronos.

## Alternativas asíncronas al microcontrolador 8051

El grupo de *Caltech* realiza en 2003 un nuevo diseño asíncrono de relevancia: Lutonium [MNP+03], una versión asíncrona del conocido microcontrolador 8051. Su diseño prima el alto rendimiento, obteniendo una eficiencia energética mayor que los diseños síncronos del controlador 8051 fabricados por Phillips ó Dallas.

En 2006, el grupo de circuitos integrados de la *Nanyang Technological University*

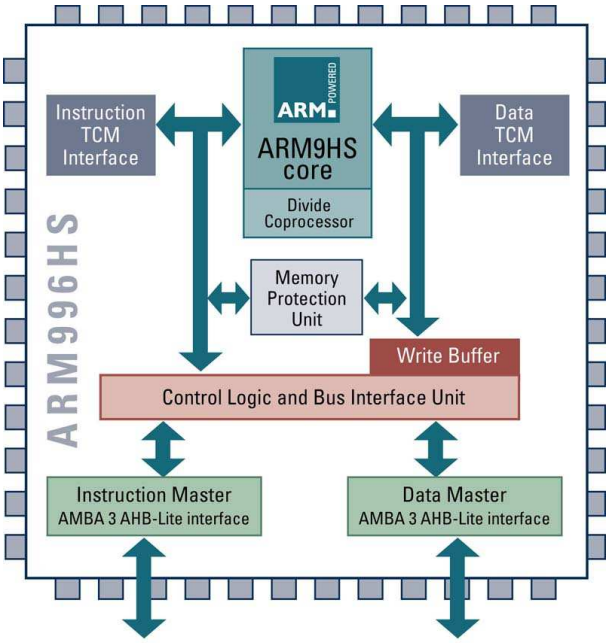


Figura 1.2: Estructura del sistema ARM996HS™(2006) [BY06].

presenta una nueva versión asíncrona del microcontrolador 8051, el A8051 [CG06], diseñado para obtener un consumo de potencia extremadamente bajo. Presenta como novedad la división de la arquitectura del controlador 8051 en un *pipeline* de dos etapas con el objetivo de minimizar la actividad del sistema.

## ARM996HS™

Un diseño aún más actual, basado también en el repertorio de instrucciones ARM, es el ARM996HS™ [BY06, BY07]. Desarrollado dentro de la alianza entre *Handshake Solutions* y *ARM*, se trata de un microchip destinado a abastecer a campos como la industria automovilística, la electrónica de consumo, la conectividad *wireless*, los implantes médicos y las conocidas *smartcards*.

Las características principales del ARM996HS™ son, en parte, similares a las del AMULET, aunque mejoradas en prestaciones. Se trata de nuevo de un procesador RISC de 32 bits con *pipeline* de cinco etapas, ejecución fuera de orden con predicción de saltos y *reorder buffer*. El diseño incluye un núcleo ARM9HS totalmente asíncrono que se comunica con el resto del chip por un protocolo *handshake* de

| *Procesador* | *Voltaje (V)* | *Año* | *Tecnología* | *Rdto. (MIPS)* | *Cons. Pot. (mW)* |
|---|---|---|---|---|---|
| MiniMIPS | 2,0 | 1997 | HP 0,6 µm | 100 | 850 |
| Amulet3 | 3,3 | 1999 | 0,35 µm | 120 | 155 |
| Lutonium 8051 | 1,8 | 2003 | TSMC 0,18 µm | 200 | 100 |
| A8051 | 1,1 | 2006 | 0,18 µm | 5 | 0,114 |
| ARM996HS™ | 1,2 | 2006 | TSMC Sage-X 0,13 µm | 83 | 0,045 |

Tabla 1.1: Resumen de características de varios procesadores asíncronos recientemente publicados.

cuatro fases. Además implementa un bus AMBA™ para comunicaciones con entornos síncronos. En la Figura 1.2 se muestra el esquema simplificado del sistema ARM996HS™.

## Comparativa

En la Tabla 1.1 se muestra un resumen de las características tecnológicas, de rendimiento y consumo de potencia de los procesadores descritos en este apartado.

Cabe destacar que todos los procesadores asíncronos mencionados son microarquitecturas de 32 bits que, como se verá más adelante, han sido desarrolladas, o bien a partir de entornos o lenguajes de programación creados especialmente para ellos, o bien a partir de lenguajes de descripción de *hardware*.

El interés de la industria y de la comunidad universitaria por los procesadores asíncronos de propósito general queda demostrado, así como la potencial demanda de herramientas y entornos de simulación parametrizables destinados al estudio de nuevas técnicas relacionadas con los procesadores asíncronos. Estos aspectos forman parte de las motivaciones fundamentales de la presente tesis, que se explican en el siguiente apartado.

## 1.4. Motivaciones y objetivos de la tesis

Como se ha visto, los circuitos asíncronos forman un campo en creciente auge que ofrece soluciones a los problemas que sufren los circuitos síncronos a causa de los avances tecnológicos. Sin embargo, existen aún inconvenientes relacionados con su desarrollo e implementación que resultan muy interesantes como líneas de investigación.

La notable carencia de herramientas de modelado y simulación de sistemas asíncronos complejos es uno de los principales inconvenientes relacionados con este tipo de circuitos. En la creación de estas herramientas, una de las mayores dificultades reside en modelar, de una manera computacionalmente eficiente, la variabilidad de los tiempos de cómputo debida a la capacidad de *computar lo antes posible* que tienen los circuitos asíncronos. Plantear una solución a este problema es una de las motivaciones de la presente tesis.

En el ámbito de los procesadores asíncronos de propósito general, la carencia de herramientas se acentúa. El desarrollo de este tipo de procesadores se suele realizar a partir de entornos construidos a medida. Estos entornos, difícilmente parametrizables, en ocasiones se apoyan en lenguajes de descripción de *hardware*. El resultado es un lento y pesado camino hasta la implementación definitiva. De hecho, no existen herramientas parametrizables de simulación de microarquitecturas asíncronas, capaces de modelar el comportamiento asíncrono, que permitan obtener información acerca del rendimiento del sistema simulado. Estas herramientas, comunes en el paradigma síncrono, permitirían evaluar las ventajas e inconvenientes derivados de la aplicación de nuevas técnicas o protocolos asíncronos a través de la recopilación de estadísticas detalladas sobre la ejecución de bancos de pruebas estándar. Aportar una herramienta de estas características para una microarquitectura asíncrona es otra de las motivaciones de este trabajo.

Por tanto, el principal objetivo de esta tesis es *el estudio e implementación de una herramienta capaz de evaluar el rendimiento de un procesador superescalar asíncrono de propósito general a través de la simulación, a nivel arquitectónico, del comportamiento dinámico de una microarquitectura donde los tiempos de cómputo de sus componentes sean variables.*

Para alcanzar este objetivo principal se desarrollarán los siguientes objetivos se-

cundarios:

1. Estudio de una metodología de modelado del tiempo de cómputo de un circuito asíncrono que, siendo computacionalmente eficiente, represente fielmente la variabilidad de la latencia de cada módulo del circuito.

    - La caracterización del tiempo de cómputo se realizará a través de funciones de distribución de la probabilidad.

2. Modelado, a través de un simulador arquitectónico, de un procesador superescalar asíncrono que disponga de características avanzadas como predicción de saltos y ejecución especulativa de instrucciones.

    - El procesador debe ser capaz de manejar repertorios de instrucciones compatibles con algún conjunto de bancos de pruebas conocido y extendido en la comunidad científica.

3. Estudio de caracterizaciones genéricas para el tiempo de cómputo que permitan al simulador modelar el comportamiento asíncrono en la microarquitectura simulada.

    - La caracterización del tiempo de cómputo debe ser independiente para cada uno de los módulos que componen el procesador.
    - La descripción microarquitectónica y la caracterización del tiempo de cómputo se deben mantener separadas en el simulador. Así será posible trabajar con distintas caracterizaciones de tiempo sin afectar a la funcionalidad del procesador.

4. Integración en el simulador de la caracterización del tiempo de cómputo de cada uno de los módulos asíncronos que componen la microarquitectura del procesador.

    - Es necesario que la integración incluya un interfaz amigable que permita la parametrización de las distintas funciones de distribución asociadas a cada módulo, así como la propia asociación entre módulo y función de caracterización.

- Se utilizará el lenguaje de marcado XML en la descripción de las funciones de distribución.

5. Integración en el simulador del modelado de varios protocolos de comunicación, configurables individualmente para cada una de las comunicaciones entre módulos de la microarquitectura.

   - En un diseño complejo como el procesador asíncrono tratado, es deseable disponer de distintos protocolos de comunicación para la trasferencia de información entre distintos componentes. En una primera aproximación se considerarán los protocolos de comunicación de dos y cuatro fases.
   - La selección y configuración de los protocolos de comunicación entre módulos se llevará a cabo utilizando el lenguaje de marcado XML.

6. Estudio y generación de medidas acerca del rendimiento de la microarquitectura asíncrona, adicionales a las estadísticas de ejecución comunes en los simuladores arquitectónicos (*i.e.*, número de instrucciones ejecutadas, número de saltos, estadísticas sobre el predictor de saltos, etc...):

   - Evaluación de tiempo de cómputo total para cada simulación, en unidades de tiempo.
   - Estadísticas sobre las instrucciones procesadas: valores máximos, mínimos y promedio sobre la latencia de las instrucciones dentro de la microarquitectura, rendimiento en términos de número de instrucciones por unidad de tiempo.
   - Organización de las estadísticas anteriores por cada uno de los módulos de la microarquitectura.
   - Tasa de actividad de cada módulo de la microarquitectura.

7. Estudio del modelado de *sistemas parcialmente síncronos*.

   - Los *GALS* (*Globally-Asynchronous, Locally-Synchronous*), sistemas globalmente asíncronos, localmente síncronos [Cha84], también suscitan interés en la comunidad investigadora. Desde el punto de vista de la

descripción temporal del circuito, un GALS se puede definir como un diseño asíncrono donde el tiempo de cómputo de algunos de sus componentes se ha fijado a una frecuencia determinada.

- Los *PAMs* (*Partially Asynchronous Microprocessors*), microprocesadores parcialmente asíncronos [MABB02], y los LAGS (*Locally-Asynchronous, Globally-Synchronous*) son también posibles objetivos de la herramienta propuesta.

8. Estudio de otras aplicaciones del simulador relacionadas con la variabilidad en los tiempos de cómputo.

   - La variación en el tiempo de cómputo de un circuito asíncrono se puede alterar debido a factores distintos al dato computado. El envejecimiento del circuito o las variaciones en voltaje y temperatura de funcionamiento pueden modificar su comportamiento. La herramienta que se presenta en esta tesis podría permitir el estudio del rendimiento de la microarquitectura bajo múltiples condiciones ambientales.
   - La selección, en tiempo de ejecución, de distintas funciones de distribución para caracterizar el tiempo de cómputo de un módulo asíncrono permite simular algoritmos o técnicas adaptativas, como por ejemplo, el escalado dinámico de voltaje.

Las principales aportaciones de este trabajo, relacionadas con los objetivos que se acaban de presentar, se explican con detalle en el Capítulo 6.

## 1.5. Estructura de la tesis

Esta tesis se estructura en seis capítulos. Los dos primeros capítulos se dedican a exponer el estado del arte y los conocimientos previos que fundamentan el resto de la tesis. Los siguientes cuatro capítulos muestran las aportaciones realizadas y se basan en los conceptos que se presentan en los dos primeros capítulos.

En el Capítulo 2 se presenta el estado del arte de las herramientas y técnicas de modelado de sistemas asíncronos complejos. En él se muestran, por un lado,

técnicas de alto nivel de abstracción como los métodos formales, herramientas asociadas a métodos formales y lenguajes de descripción de procesos concurrentes. Por otro lado, se describen algunos simuladores de microarquitecturas asíncronas, comparando sus características con las que se enuncian en las motivaciones de este trabajo.

El Capítulo 3 muestra las primeras aportaciones de esta tesis. En él se estudia la caracterización genérica del tiempo de cómputo de un circuito, para distintas configuraciones de temporización. En este estudio se valoran las posibles causas de la variabilidad en la latencia y se propone la utilización de funciones de distribución de la probabilidad para caracterizar esta variabilidad. Además se propone un método de caracterización del tiempo de cómputo que, utilizando fundamentos estadísticos, garantiza una caracterización fiable.

El resto de aportaciones de esta tesis se describen en los capítulos 4 y 5. En el Capítulo 4, tras indicar las diferencias entre el modelado y simulación de sistemas síncronos y asíncronos, se concluye la necesidad de crear un simulador que separe la descripción funcional del circuito del modelado de su temporización. En consecuencia se propone, por un lado, una nueva microarquitectura superescalar con ejecución especulativa de instrucciones y predicción de saltos capaz de procesar el repertorio de instrucciones del procesador Alpha 21264. Por otro lado, se proponen distintos modelos para caracterizar la temporización de la microarquitectura. Todo ello se integra en *Sim-async*, el nuevo simulador resultado de este trabajo.

El Capítulo 5 describe el proceso de validación del simulador y dos estudios arquitectónicos donde se utiliza la herramienta para evaluar el rendimiento, bajo distintas configuraciones asíncronas, de la microarquitectura modelada en comparación con su equivalente síncrona.

En el Capítulo 6 se presentan las conclusiones de este trabajo, sus principales aportaciones y se indican las líneas de investigación que quedan abiertas.

Finalmente, la tesis termina con dos apéndices. En el Apéndice A se describen conceptos fundamentales en el diseño de circuitos asíncronos. El Apéndice B muestra información adicional sobre el lenguaje de marcado XML, profundizando en las gramáticas definidas para la configuración del simulador propuesto.

# Capítulo 2

# Estado del arte

Las herramientas de modelado y simulación de sistemas son esenciales en el flujo de diseño de circuitos puesto que, generalmente, permiten medir el rendimiento del producto en desarrollo y verificar tanto su correcta funcionalidad como el cumplimiento de los requisitos de temporización. De hecho, en el paradigma de los circuitos síncronos existen multitud de herramientas de modelado sin las cuales el desarrollo de circuitos complejos, con el actual nivel de integración VLSI[1], tendría una elevada dificultad.

El reciente interés por el diseño de circuitos asíncronos ha alimentado una intensa actividad investigadora cuyo objetivo es desarrollar técnicas apropiadas para el modelado y simulación de sistemas asíncronos. En este tipo de sistemas, el papel de la simulación es más importante, si cabe, que en los circuitos síncronos. Esto se debe tanto a su carácter concurrente como a la variabilidad en los tiempos de cómputo, dependientes de los datos, que convierten cualquier intento de comprobar la corrección de un circuito asíncrono o estimar su rendimiento en una tarea muy complicada de llevar a cabo.

En este capítulo se realiza un repaso global a las herramientas y técnicas utilizadas para el modelado y simulación de circuitos asíncronos, tanto en el ámbito académico como en el industrial. Así, el conjunto de herramientas y técnicas descritas se divide en tres categorías principales:

- Métodos formales y herramientas derivadas
- Lenguajes de descripción de procesos concurrentes

[1] 45 nm, año 2008

- Simuladores de microarquitecturas asíncronas

A continuación se presentan las características de los representantes más relevantes en cada una de estas categorías, evaluando después las ventajas e inconvenientes de utilizarlas para el modelado y simulación de la microarquitectura asíncrona objetivo de esta tesis: un procesador superescalar asíncrono de propósito general donde los tiempos de cómputo de sus componentes es variable.

## 2.1. Métodos formales y herramientas derivadas

Los métodos ó metodologías formales para el modelado de circuitos permiten al diseñador de sistemas componer una descripción de alto nivel de la funcionalidad del circuito, lejos de los detalles de su implementación. De este modo, la comprobación de la corrección del circuito modelado se basa en el cumplimiento de las reglas que determina la metodología utilizada.

Generalmente, los métodos formales para el modelado de circuitos asíncronos que se encuentran en la literatura ofrecen soluciones para la descripción del circuito, pero no para su simulación. Son las herramientas derivadas de estos métodos las que aunan la capacidad de describir un sistema asíncrono con la simulación y la verificación de sus funcionalidades.

En este apartado se establece una clasificación en dos tipos de metodologías dentro de los métodos formales para el modelado de circuitos asíncronos: por un lado, las metodologías basadas en máquinas de estados finitas asíncronas ($AFSM$) y, por otro lado, las metodologías derivadas de redes de Petri. Para ambas categorías se incluyen también referencias a herramientas relacionadas.

### 2.1.1. Máquinas de estado finitas asíncronas

Las maquinas de estados finitas asíncronas son un método tradicional de modelado de circuitos asíncronos [Hol82, Chu92]. Esta metodología consiste en describir los cambios de estado del circuito en función de los cambios en los valores de entrada y del estado actual. Así, el resultado de modelar un sistema asíncrono a

través de su *AFSM* es una matriz donde se hacen corresponder estados y valores de entrada con los siguientes estados en la ejecución del sistema y los valores de salida. Herramientas basadas en las *AFSM* son, por ejemplo, MINIMALIST [FNT+99] y 3D [YDN92].

La descripción a partir de *AFSM* es viable para circuitos asíncronos sencillos, con un número de entradas y salidas contenido que genere un conjunto de estados pequeño, pero no para el modelado y simulación de un procesador asíncrono de propósito general puesto que el número de estados posibles en este tipo circuitos es muy elevado.

### 2.1.2. Redes de Petri

Una red de Petri es una representación matemática de un sistema distribuido discreto [Pet66, Mur89]. Desde un punto de vista semántico, se trata de una generalización de la teoría de autómatas que, a partir de unas reglas que definen lugares, transiciones y arcos, permite modelar la ejecución de un sistema de procesos concurrentes.

Han sido muchas las metodologías y herramientas surgidas a partir de las redes de Petri para la descripción de distintos tipos de sistemas. Además de la conocida herramienta Petrify [CKK+97], algunas metodologías derivadas son:

- Los grafos de señales (*signal graphs*) [RY85].
- Las redes de interfaces (*i-Nets*) [MF83].
- Los grafos de transición de señal (*STGs*) [Chu87] y herramientas derivadas como CASCADE [BEW00].

En todos los casos mencionados se trata de metodologías y herramientas de descripción y verificación de circuitos descritos a alto nivel, disponiendo algunas de ellas de la posibilidad de generar descripciones de circuitos a nivel de puertas. Sin embargo, ninguna de estas herramientas ofrece un entorno de simulación y modelado parametrizable capaz de trabajar con una microarquitectura como la propuesta en esta tesis. Es por ello que ninguno de los métodos formales basados en *AFSM* ó redes de Petri, ni sus herramientas derivadas, se han considerado a la hora de completar el presente trabajo.

## 2.2. Lenguajes de descripción de procesos concurrentes

Los circuitos asíncronos están formados por componentes que envían y reciben datos a través de un protocolo de comunicación, realizando su tarea de un modo autónomo independiente del resto del circuito. Este comportamiento es análogo al que presentarían un conjunto de procesos ejecutando de manera concurrente en un sistema, estableciendo comunicaciones entre ellos con una periodicidad indeterminada.

La especificación algebraica *CSP* (*Communicating Sequential Processes*) acerca de la computación y comunicación de procesos paralelos desarrollada por Hoare [Hoa78], se ajusta perfectamente al modelo de cómputo de los sistemas asíncronos:

- *CSP* soporta un modelo de cómputo concurrente, asíncrono, no determinista, basado en procesos, similar al funcionamiento de un circuito asíncrono.
- En *CSP*, las comunicaciones entre procesos se realizan a través de canales ó conexiones punto a punto sincronizadas y sin almacenamiento temporal. Es decir, la comunicación sólo se realiza entre dos procesos teniendo en cuenta que el primer proceso en llegar a un canal, bien para leer, bien para escribir, queda bloqueado en espera del otro interviniente en la comunicación. Este comportamiento refleja la interacción entre dos módulos asíncronos que, utilizando un protocolo de comunicación, esperan a que ambos estén preparados antes de la transmisión de los datos.
- La generación de circuitos autosincronizados[2] a partir de un conjunto de procesos concurrentes descritos en *CSP* es sencilla y puede automatizarse a través de mecanismos guiados por el análisis sintáctico del código *CSP* [BM88].

Gracias a esta similitud en el modelo de cómputo y transmisión de información, así como a la relativa sencillez en la generación de circuitos, se han utilizado multitud de sistemas y lenguajes basados en *CSP* para el desarrollo y simulación

[2] *Self-timed circuits.*

de diseños asíncronos. A continuación se muestran varios ejemplos de lenguajes basados en *CSP* utilizados en el desarrollo de procesadores y microcontroladores asíncronos, ordenados cronológicamente según su publicación.

### 2.2.1. Occam

Occam [Lim84] es una de las primeras implementaciones ejecutables de la especificación algebraica *CSP*. Este lenguaje de programación concurrente fue creado por la compañía *INMOS* (hoy *SGS-Thomson Microelectronics*) para el desarrollo del *transputer* [WS85], el primer microprocesador síncrono desarrollado específicamente para el cómputo de procesos concurrentes. A pesar de que este procesador no tomó la relevancia que sus creadores esperaban, el lenguaje que lo ayudó a nacer sirvió de base para el modelado y desarrollo de multitud de circuitos asíncronos.

Como ejemplo destacable de trabajo sobre procesadores asíncronos con Occam es necesario mencionar la colaboración entre el *Advanced Processor Technologies Group* de la Universidad de Manchester y la Universidad de Birmingham donde

- Se establecen las bases para considerar a Occam como un lenguaje realmente válido para la descripción de circuitos asíncronos [TTW97].
- Se demuestra la utilidad de Occam en el desarrollo de sistemas asíncronos complejos con la creación de un entorno de modelado para el procesador AMULET1 [The03].

### 2.2.2. Tangram

Tangram [vBKR+91] es un lenguaje de programación imperativo con extensiones para ejecución de procesos concurrentes basadas en *CSP*. Tangram es similar a Occam en su capacidad para la descripción de procesos, puesto que ambos derivan de *CSP*. Sin embargo, ambos lenguajes son distintos entre sí, principalmente, en cuanto a su capacidad para definir detalles de implementación de las estructuras que pueden describir. Fundamentalmente estos lenguajes se distinguen porque:

- Tangram permite definir canales de comunicación ”uno a muchos” mientras que Occam sólo permite comunicaciones ”uno a uno”.
- Tangram proporciona funciones compartidas por los procesos mientras que Occam permite asignación de procesos y canales a elementos de *hardware*.

Las diferencias entre Tangram y Occam provienen de los distintos objetivos para los que fueron creados cada uno de estos lenguajes. Mientras Occam se diseñó originalmente para modelar sistemas de procesos concurrentes a ejecutar en arquitecturas *transputer*, Tangram es un lenguaje dedicado a la especificación y síntesis de *hardware*, por lo que su objetivo es definir procesos concurrentes que serán transformados a circuitos VLSI en tiempo de compilación. Tangram se utilizó, por ejemplo, para el desarrollo de una versión asíncrona de bajo consumo de potencia del microcontrolador 8051 [vGBvB+98].

A partir de Tangram se han desarrollado también entornos integrados de trabajo o *frameworks*, como el presentado en [KP01], cuyo flujo de diseño se muestra en la Figura 2.1. De nuevo, la intención de entornos como el que se ilustra en la figura es partir de una especificación de alto nivel del circuito para llegar a una descripción a nivel de puertas.

### 2.2.3. *LARD*

Un ejemplo de lenguaje específico para descripción de circuitos asíncronos es *LARD* [EF98], acrónimo de *Language for Asynchronous Research and Development*, o lenguaje para investigación y desarrollo asíncrono. *LARD* es un lenguaje de descripción *hardware* cuya creación se vincula directamente con el microprocesador asíncrono AMULET3 [FGG98, GFC99]. En el desarrollo de anteriores versiones de este procesador, el *Advanced Processor Technologies Group* de la Universidad de Manchester concluyó que el tiempo empleado en desarrollar un modelo del procesador a alto nivel para evaluar su funcionalidad arquitectónica resultaba excesivo, convirtiéndose en el principal cuello de botella del proyecto. Por ello, decidieron crear *LARD* como lenguaje a utilizar para la descripción del AMULET3, con el ánimo de reducir el tiempo empleado en la obtención de modelos funcionales del procesador.

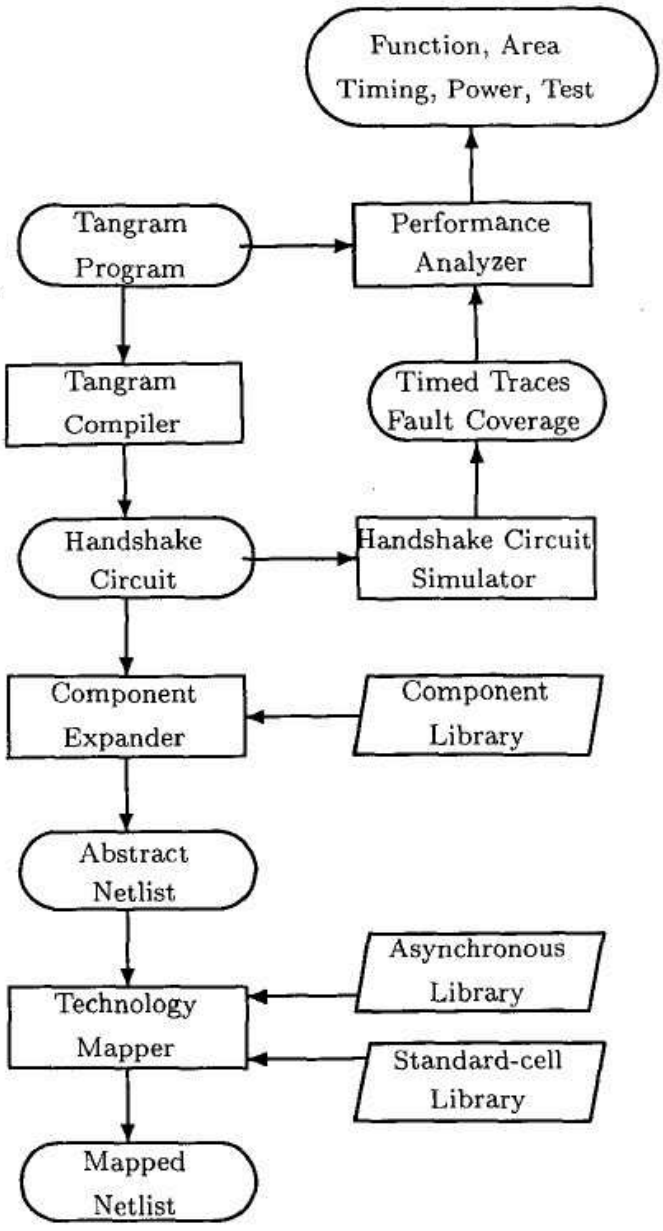


Figura 2.1: Flujo de diseño en un entorno Tangram (obtenido de [KP01]). Los rectángulos corresponden a herramientas y los óvalos a representaciones del diseño.

*LARD* proporciona soporte para comunicaciones atómicas entre procesos así como la posibilidad de definir tanto procesos concurrentes como secuenciales. Cumple, por tanto, los requisitos asociados a cualquier lenguaje derivado de *CSP*, y se sitúa en un nivel de abstracción intermedio entre Occam, lenguaje de más alto nivel, y Tangram, lenguaje más cercano a la implementación *hardware*.

### 2.2.4. Balsa

Balsa es el nombre que recibe tanto un entorno de síntesis de *hardware* asíncrono [EB00] como el lenguaje para describir este tipo de sistemas a nivel de transferencia de registros (RTL) [BE97]. Este lenguaje utiliza estructuras basadas en *CSP* para expresar tanto comunicaciones atómicas en forma de canales de transferencia de datos, como descripciones de concurrencia de procesos.

En la Figura 2.2 se muestra el flujo de diseño del entorno Balsa, donde este lenguaje sólo se utiliza en la simulación del sistema a alto nivel (*behavioral*). De hecho, para la simulación de partes del sistema cuya descripción *hardware* necesita de mayor detalle, Balsa se apoya en el lenguaje *LARD*, integrando en el interfaz del usuario la posibilidad de importar ficheros en este lenguaje, como se aprecia en la Figura 2.3.

El entorno Balsa se ha utilizado en el desarrollo del *SoC* AMULET3i [GBB+00], así como en el desarrollo del procesador asíncrono SAMIPS [ZT04], y en el desarrollo del reciente A8051 [CG06], una versión empotrable asíncrona del microcontrolador 8051.

### 2.2.5. Conclusiones acerca de los lenguajes *CSP*

Todos los lenguajes y entornos descritos en este apartado son capaces de realizar el modelado y simulación funcional de un procesador asíncrono, cubriendo así parte del objetivo planteado para este trabajo: el correcto modelado de un sistema asíncrono.

Sin embargo, puesto que el objetivo de todas las metodologías mencionadas es obtener una descripción a bajo nivel del circuito, ninguna de ellas proporciona

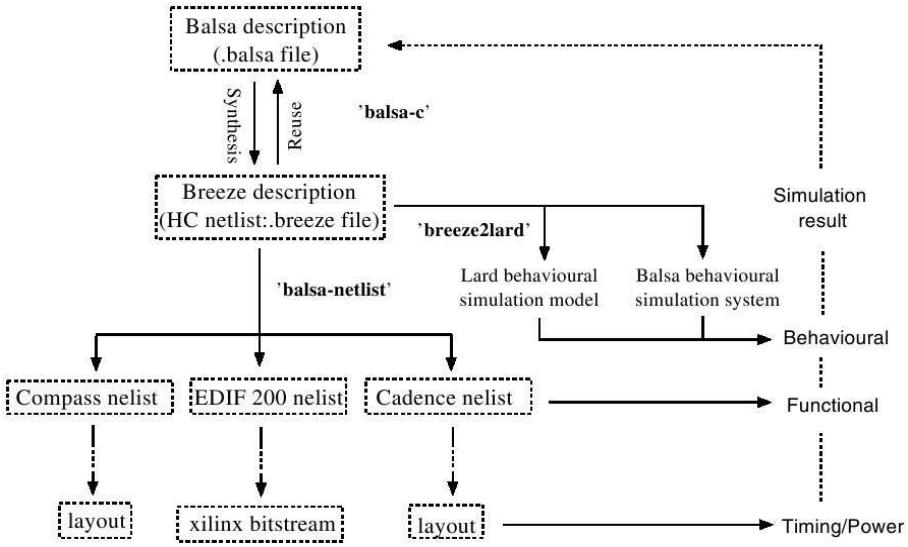


Figura 2.2: Flujo de diseño en un entorno Balsa (obtenido de [ZT04]). Sólo la descripción del sistema a alto nivel (*behavioral*) se realiza con Balsa, el resto de niveles utiliza herramientas CAD convencionales.

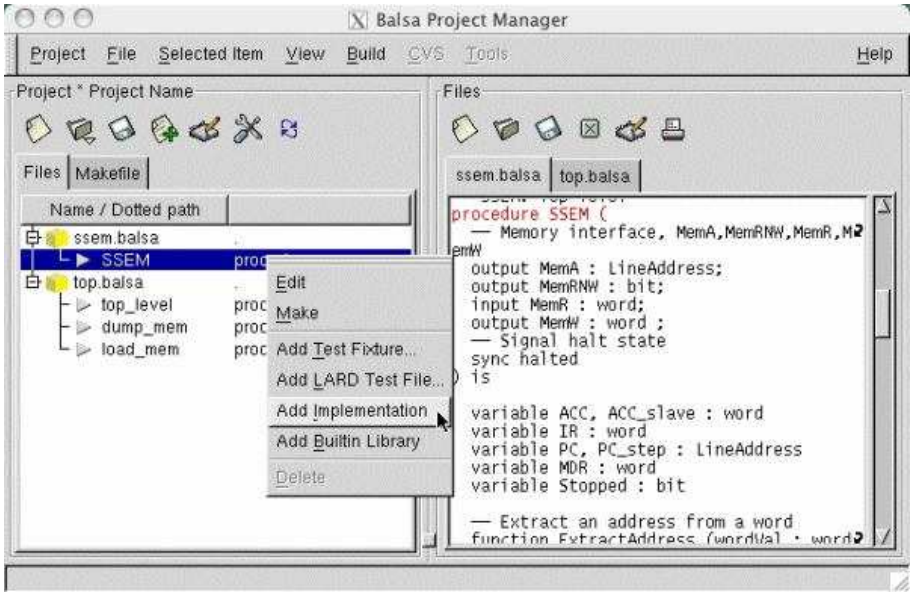


Figura 2.3: Captura de pantalla del gestor de proyectos de Balsa. Permite integrar descripciones a más bajo nivel utilizando *LARD*.

un entorno de simulación arquitectónica parametrizable similar al propuesto en esta tesis.

En resumen, utilizando cualquiera de los entornos o lenguajes basados en *CSP* que se detallan en este apartado:

- No es posible realizar simulaciones arquitectónicas que representen fielmente las variaciones en el tiempo de cómputo.
- No es posible estimar el valor de medidas acerca del rendimiento del procesador modelado, como la latencia de las instrucciones ó el número de instrucciones finalizadas por unidad de tiempo.
- No es posible ejecutar bancos de prueba estándar sobre ninguna de las representaciones de circuito que se muestran en este apartado sin emplear cantidades ingentes de tiempo.

En consecuencia, se ha descartado la utilización en este trabajo de lenguajes o entornos construidos en base a las directivas planteadas por *CSP*.

## 2.3. Simuladores de microarquitecturas asíncronas

La simulación de microarquitecturas asíncronas representa, al igual que ocurre en el paradigma síncrono, una área en continua evolución. Su motivación principal es la evaluación, a través de la ejecución de bancos de pruebas estándar, de distintas medidas acerca de la microarquitectura bajo estudio tanto en términos de rendimiento como en consumo de potencia. Las simulaciones, realizadas utilizando una herramienta parametrizable y suficientemente flexible, permiten estimar, entre otras posibilidades, el grado de satisfacción que pueden ofrecer distintas implementaciones de una misma funcionalidad del procesador con respecto a las restricciones u objetivos del diseño final.

En la bibliografía sobre procesadores asíncronos no existen muchos trabajos acerca de simuladores específicos construidos para la evaluación del rendimiento de

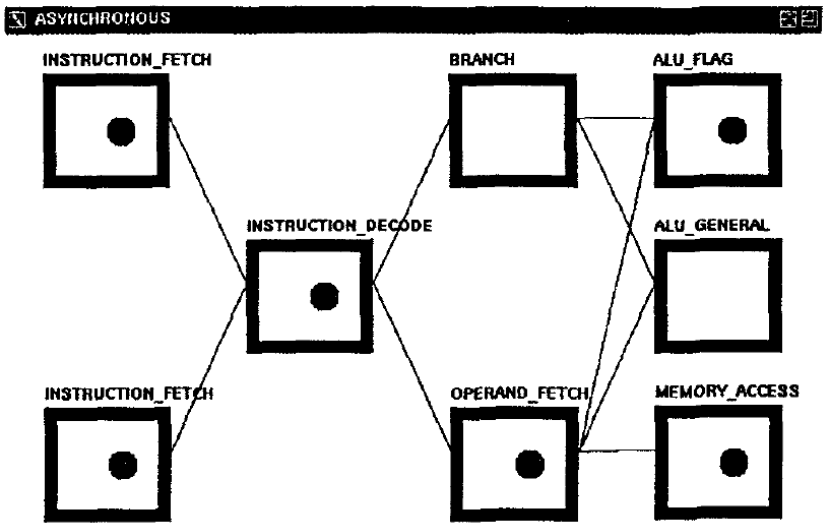


Figura 2.4: Captura de pantalla del interfaz gráfico de usuario de ARAS, tomada de [CFPP95].

microarquitecturas asíncronas. Como se mencionó en apartados anteriores, la mayoría de las herramientas se vinculan al proceso de obtención y verificación de una descripción de bajo nivel del sistema asíncrono en desarrollo.

A continuación se analizan, de nuevo según su orden cronológico de publicación, tres ejemplos de entornos de simulación de microarquitecturas asíncronas.

### 2.3.1. ARAS

Presentada en [CFPP95], ARAS es una herramienta gráfica para simulación de procesadores RISC asíncronos. En la Figura 2.4 se muestra una captura de pantalla del interfaz gráfico de usuario, tomada de [CFPP95]. En este simulador se representa cada una de las etapas del *pipeline* con un rectángulo posicionado en pantalla en una determinada columna. Aquellos rectángulos situados en la misma columna representan operaciones que se ejecutan en paralelo. La comunicación de datos entre etapas se representa a través de conexiones entre los rectángulos, y el flujo de instrucciones en el *pipeline* queda representado por círculos que se trasladan de un rectángulo a otro.

El simulador ofrece la posibilidad de diseñar distintos tipos de pipeline modificando el número de etapas que ejecuta en paralelo o el orden en que lo hacen.

También ofrece la posibilidad de especificar algunos parámetros como el tiempo empleado por el protocolo de comunicación entre etapas. Sin embargo, no incluye una descripción a nivel arquitectónico de las etapas ni la consideración de distintos tiempos de cómputo para datos diferentes, salvo para el módulo del sumador, donde se dispone de varias alternativas. El artículo no explica claramente ninguna consideración de tiempo variable de cómputo en los datos procesados, necesario para describir el comportamiento variable de un circuito asíncrono. No obstante sí se indica, a través de unas gráficas, la diferencia existente en los tiempos de cómputo de los sumadores elegibles en la aplicación.

La principal limitación de ARAS es su incapacidad para simular la ejecución de un código compilado. Por contra, ARAS simula una traza generada previamente utilizando un computador SPARC, por lo que la planificación de las instrucciones no se genera dinámicamente con respecto a la situación del sistema modelado en cada instante de la simulación.

### 2.3.2. PEPSÉ

El enfoque de PEPSÉ [Reb97] es distinto al de ARAS. PEPSÉ implementa la simulación guiada por eventos del modelo RTL de una microarquitectura, precisando detalles de temporización a través de retardos tomados de simulaciones SPICE de la propia microarquitectura.

La aplicación se codificó en Occam2, y su ejecución se llevaba a cabo en una red de *transputers*. PEPSÉ no se ha mencionado en el apartado dedicado a los lenguajes *CSP* porque, a pesar haber sido codificada en un lenguaje de este tipo, pretendía ser una herramienta de evaluación de rendimiento de microarquitecturas asíncronas en general, no una parte del desarrollo de ningún procesador en particular.

Aunque su objetivo es ser considerada una herramienta genérica, PEPSÉ simula una microarquitectura que ejecuta un conjunto de instrucciones propio, incompatible con cualquiera de los bancos de pruebas estándar que habitualmente se utilizan en el área de arquitectura de computadores. PEPSÉ no implementa funcionalidades actuales como el lanzamiento simultáneo de varias instrucciones (ejecución superescalar) o la predicción de saltos. Además, el simulador reutiliza

valores obtenidos de simulaciones de bajo nivel como retardos para la producción de los eventos manejados. Tampoco se tratan explícitamente tanto la variabilidad en los tiempos de ejecución como el protocolo de comunicación.

### 2.3.3. *simCore*

Una herramienta que se acerca más a la propuesta de esta tesis es *simCore* [JKKC00]. Su principal semejanza con el simulador propuesto en esta tesis es la utilización de eventos en el proceso de simulación, técnica que permite el modelado de componentes cuyas operaciones se ejecutan concurrentemente.

Sin embargo, *simCore* no describe una microarquitectura compleja parametrizable, sino que ofrece un entorno de simulación dividido en tres módulos: CPU, memoria y "sistema". En el primero de los módulos el usuario debe definir, programando en lenguaje C++, la microarquitectura a simular. En el módulo de memoria se cargan las trazas con los accesos a la memoria simulada, lo que correspondería al programa a ejecutar en la simulación. Por último, es el módulo "sistema" quien realiza la generación y gestión de eventos asociada a las operaciones de la CPU en función de lo indicado en el módulo de memoria.

La herramienta, bajo el punto de vista de la evaluación del rendimiento de una arquitectura asíncrona, apenas ofrece el núcleo de simulación, puesto que los detalles de la microarquitectura han de ser extendidos y detallados por el usuario a través de código escrito en lenguaje C++.

### 2.3.4. Otras alternativas

Dentro del conjunto de referencias sobre simuladores de sistemas asíncronos se encuentran algunos entornos de trabajo que, pese a no haber sido concebidos para el estudio de microarquitecturas asíncronas concretas, permitirían esa funcionalidad tras un proceso de extensión.

Ejemplos destacables de este tipo de aplicaciones son [Gho01] y [NKB04], donde el modelado del comportamiento asíncrono, de nuevo, se gestiona a través de eventos.

## 2.4. Resumen

Una vez planteadas las opciones encontradas en la bibliografía del área, en este apartado se analizan las ventajas e inconvenientes de utilizar en este trabajo cada una de esas alternativas examinadas previamente.

En resumen, el objetivo de esta tesis es el desarrollo e implementación de una herramienta de simulación para la evaluación del rendimiento de una microarquitectura asíncrona superescalar donde los tiempos de cómputo sean variables. Para lograr ese objetivo es necesario cubrir las siguientes funcionalidades:

1. La herramienta debe modelar correctamente un sistema asíncrono, donde cada módulo independiente pueda trabajar de manera concurrente, y la transmisión de información entre módulos se realice a través de un protocolo de comunicación.

2. La herramienta debe considerar tiempos de cómputo variables dato a dato, de manera individual en cada módulo.

3. Es necesario que la herramienta permita ejecutar bancos de pruebas estándar para poder generar medidas representativas sobre la microarquitectura bajo estudio.

4. Como herramienta de evaluación de rendimiento, el simulador debe ser altamente configurable y parametrizable.

Para completar las funcionalidades descritas, se han demostrado ineficaces las metodologías y herramientas basadas en máquinas de estados finitas asíncronas ó en redes de Petri puesto que:

- Su objetivo principal es la descripción a alto nivel y verificación funcional de sistemas asíncronos, por lo que se ignoran los detalles arquitectónicos.

- La descripción del conjunto de posibles estados de un procesador superescalar asíncrono es demasiado extensa para ser manejada de manera eficiente por este tipo de herramientas.

Los lenguajes de descripción de procesos concurrentes basados en *CSP* se han utilizado en el desarrollo de importantes proyectos sobre sistemas asíncronos. Sin embargo, ningún lenguaje o herramienta derivada se ha concebido para la evaluación, a nivel arquitectónico, de sistemas donde los tiempos de cómputo dependan de los datos que se procesan en cada momento.

Por último, ninguno de los simuladores que se encuentran en la literatura ofrece las funcionalidades descritas al inicio de este apartado.

En consecuencia, la propuesta de esta tesis consiste en el estudio e implementación de una nueva herramienta, altamente configurable, que permita la ejecución de bancos de pruebas estándar para la evaluación del rendimiento de un procesador asíncrono superescalar cuyos componentes empleen tiempos de cómputo variables para cada uno de los datos procesados.

## Capítulo 3

# Caracterización de tiempos de cómputo variables

En este capítulo se analizan las causas de la variabilidad en el tiempo de cómputo, tanto para circuitos síncronos como para circuitos asíncronos, y se propone la utilización de funciones de distribución de probabilidad como método de caracterización del tiempo de cómputo para la simulación arquitectónica de circuitos asíncronos.

Antes de abordar la caracterización de tiempos de cómputo variables, se presentan a continuación las definiciones de dos conceptos ampliamente utilizados a partir de este punto. Se trata de las definiciones de tiempo de cómputo y camino crítico de un circuito.

**Definición (Tiempo de cómputo)**: el *tiempo de cómputo* de un dato en un circuito se define como el tiempo transcurrido desde el instante en que el circuito recibe en sus entradas el conjunto de valores correspondientes a un dato, hasta el momento en que los resultados del cómputo de esos valores de entrada aparecen en las salidas del circuito.

**Definición (Camino crítico)**: el *camino crítico* de un circuito es la ruta con mayor retardo [Syn07]. La velocidad de un circuito síncrono depende del retardo más lento registro a registro. En otras palabras, el periodo de reloj no puede ser menor que el retardo del camino crítico puesto que las señales no llegarían a tiempo de ser capturadas en el siguiente registro.

## 3.1. Variabilidad en el tiempo de cómputo

La variabilidad en el tiempo de cómputo se define como la posibilidad de obtener distintos valores de retardo al trabajar con un mismo circuito. Existen varios factores que afectan a la variabilidad en el tiempo de cómputo tanto en circuitos asíncronos como en circuitos síncronos. Estos factores se pueden agrupar en dos clases. Por un lado se encuentran los factores extrínsecos al circuito como la temperatura, el voltaje de alimentación y el proceso de fabricación. Por otro lado aparecen factores intrínsecos al circuito como es la dependencia de los datos. En este sentido, se debe considerar qué datos de entrada distintos pueden recorrer caminos diferentes del circuito, dando lugar a tiempos de cómputo distintos. Por ello, la dependencia de los datos de entrada puede ser un factor determinante para el rendimiento de los circuitos asíncronos.

### 3.1.1. Influencia de factores extrínsecos al circuito

Tanto para los circuitos síncronos como para los circuitos asíncronos, los factores físicos que en mayor medida provocan variabilidad en el tiempo de cómputo son, por orden de relevancia, la temperatura, el voltaje de alimentación y el proceso de fabricación [LWAM07, UTB+06]. Estos factores se denotan de manera conjunta por la sigla *PVT* (*Process, Voltage and Temperature*). A continuación se describe, brevemente, la influencia de cada uno de ellos.

- **Temperatura**: todos los circuitos se diseñan para trabajar correctamente en un determinado rango de temperaturas. Sin embargo, este parámetro puede sufrir cambios importantes a causa de condiciones ambientales, provocando variaciones en el rendimiento del circuito. La causa más habitual de la degradación del rendimiento de un circuito es el aumento de temperatura, puesto que afecta al comportamiento de los componentes del circuito ante un mismo voltaje de alimentación. De hecho, se ha demostrado que el aumento de la temperatura provoca un incremento en el retardo de los componentes CMOS y en su consumo de potencia [GK03].
- **Voltaje de alimentación**: al igual que ocurre con la temperatura, los circuitos se diseñan para funcionar a una determinada tensión nominal de

trabajo, por supuesto, dentro de unas tolerancias. Cualquier variación en la tensión de alimentación provocará una aceleración ó una ralentización de la velocidad de operación de los dispositivos físicos del circuito, alterando el tiempo de cómputo de los datos.

En circuitos CMOS, la velocidad de operación de los transistores es inversamente proporcional a la corriente del drenador, $I_D$. Según [Pie94], aplicando la *ley cuadrática*, se obtiene la siguiente expresión

$$I_D = \frac{Z\bar{\mu_n}C_0}{L}\left[(V_G - V_T)V_D - \frac{{V_D}^2}{2}\right] \left( \begin{array}{c} 0 \leq V_D \leq V_{Dsat} \\ V_G \geq V_T \end{array} \right) \tag{3.1}$$

donde $Z$ y $L$ corresponden a la anchura y longitud del canal respectivamente, $\bar{\mu_n}$ a la carga electrónica promedio por $\mathrm{cm}^2$, $C_0$ a la capacidad del óxido por unidad de área, y $V_D$, $V_{Dsat}$, $V_G$, y $V_T$ a los voltajes de drenador, drenador en saturación, puerta y umbral respectivamente. El voltaje de alimentación corresponde a $V_D$ . Cuanto mayor sea este valor, menor será el valor de $I_D$ y viceversa.

- **Proceso de fabricación**: durante el proceso de fabricación de los circuitos integrados resulta prácticamente imposible que las impurezas y el dopado del silicio de dos circuitos que implementan la misma funcionalidad se distribuyan exactamente de la misma manera [UTB+06]. El resultado es la obtención de circuitos con unas propiedades físicas similares, pero no idénticas. Así, la velocidad de operación de los dispositivos físicos de un circuito se verá afectada por el proceso de fabricación, aunque en menor medida que en el caso de los valores de temperatura y voltaje de alimentación. La consecuencia, de nuevo, es la variabilidad en los tiempos de cómputo.

  Considerando de nuevo la expresión 3.1, las variables que se verían afectadas por el proceso de fabricación son $\bar{\mu_n}$ , carga electrónica promedio por $\mathrm{cm}^2$, y $C_0$ , capacidad del óxido por unidad de área. La influencia de estos valores no es determinante, aunque también influyen en la velocidad del dispositivo.

  De hecho, existen variaciones de proceso de fabricación que se consideran dinámicas puesto que se deben al envejecimiento del circuito, influyendo también en su rendimiento [UTB+06]. Este efecto se denomina literalmente

envejecimiento ó *aging*[1].

Por tanto, las variaciones en $PVT$ afectan al tiempo de cómputo tanto de los circuitos síncronos como de los circuitos asíncronos, aunque la solución que se aplica sobre estas variaciones es distinta en ambos paradigmas.

En los circuitos síncronos, el periodo de reloj se establece tras analizar distintas combinaciones de valores $PVT$. Estas combinaciones ó extremos (*corners*), suelen dar lugar a distintos tiempos de cómputo para el mismo dato de entrada. Esta circunstancia conduce a la necesidad de asegurar la correcta funcionalidad del circuito bajo cualquier combinación de factores $PVT$. En consecuencia, se aplican importantes márgenes de seguridad sobre el tiempo de cómputo, para así paliar los efectos de las variaciones en $PVT$. Como se verá más adelante, esta prevención afecta negativamente al rendimiento de los circuitos síncronos. Sin embargo, como se ha explicado, resulta inevitable para conseguir que el circuito sea tolerante a cualquier variación $PVT$ dentro del margen especificado.

Los circuitos asíncronos, en cambio, se diseñan de modo que su funcionalidad sea independiente de los requisitos temporales, por lo que se adaptan automáticamente a las variaciones en $PVT$. En consecuencia, el cómputo se realiza tan rápido como sea posible bajo valores concretos de $PVT$ para el circuito.

### 3.1.2. Dependencia de los datos de entrada

Bajo las mismas condiciones de $PVT$, el tiempo de cómputo de un circuito puede ser muy distinto para diferentes datos de entrada. Expresado de otro modo, dado un valor o valores a la entrada del circuito, el camino formado por las puertas que efectivamente computan puede ser distinto para cada uno de los datos de entrada y, muy probablemente, distinto del camino crítico. Por ello, se dice que el tiempo de cómputo depende de los datos de entrada y que la variación en el tiempo de cómputo debida a los datos depende tanto del propio diseño del circuito como de las restricciones aplicadas en el proceso de síntesis. Como ejemplo de esta variación se puede tomar la caracterización que se explica más adelante, en el

[1]En algunas publicaciones, el efecto de *aging* se distingue del proceso de fabricación, y se equipara al resto de factores extrínsecos, por lo que se maneja la sigla *PVTA* (*Process, Voltage, Temperature, Aging*).

Apartado 3.4, donde se obtienen tiempos de cómputo distintos para diferentes datos de entrada en un sumador de 64 bits.

En el diseño de circuitos síncronos no se aprovecha la variación del tiempo de cómputo debida a los datos para mejorar el rendimiento. De hecho, en este tipo de circuitos se utiliza el retardo del camino crítico para establecer el periodo de la señal de reloj, fijando así un tiempo de cómputo igual para todos los datos de entrada del circuito. Éste es el motivo por el que los circuitos síncronos ofrecen siempre un rendimiento del *caso peor* [Kea99], puesto que, independientemente de los datos de entrada, el tiempo de cómputo no varía, y es igual al tiempo empleado en el cómputo más lento.

En el diseño de circuitos síncronos se utilizan métodos como el *STA* (*Static Timing Analysis*, *análisis estático de tiempos*), para averiguar si el circuito diseñado cumple con las restricciones de temporización de la señal de reloj a utilizar. El *STA* es un método de búsqueda del camino crítico de un circuito basado en el análisis topológico del mismo, donde se tienen en cuenta los retardos especificados para cada uno de los componentes y conexiones del circuito [LMS06]. Esta búsqueda del camino crítico es independiente de los posibles datos de entrada del circuito, de ahí que se denomine a este método como estático.

Los métodos estáticos como el *STA* no tienen en cuenta que el camino crítico del circuito puede corresponder a una ruta que nunca será activada por un dato o conjunto de datos de entrada. En esta situación, el rendimiento de los circuitos síncronos se puede ver afectado puesto que el caso peor en el retardo es demasiado pesimista.

Por otro lado, en los circuitos asíncronos se puede aprovechar la circunstancia de que el tiempo de cómputo de cada uno de los datos procesados puede ser distinto a los demás. Este hecho se justifica por la ausencia de una señal de reloj que determine el instante en que un resultado ha de ser capturado. Al computar tan rápido como le es posible bajo cualquier circunstancia $PVT$, la lógica del circuito genera el resultado con un retardo, en la mayoría de los casos, menor que el retardo del camino crítico.

Existe, por tanto, una variabilidad en el tiempo de cómputo debida a los datos de entrada que conduce a que los circuitos asíncronos ofrezcan un rendimiento del *caso promedio* [Kea99].

## 3.2. Modelado con funciones de distribución

Como ya se ha explicado, los métodos de análisis estático como el *STA* son incapaces de caracterizar la variación del tiempo de cómputo de un circuito debida a los datos de entrada. Por tanto, uno de los objetivos del presente trabajo consiste en encontrar un método que caracterice de forma fidedigna la variabilidad de los tiempos de cómputo. Con una caracterización de este tipo sería posible estudiar el rendimiento de circuitos asíncronos complejos teniendo en cuenta la variación del tiempo de cómputo debida a los datos. Estos estudios requieren que la caracterización se integre en un simulador arquitectónico, de modo que se deberá tener en cuenta este aspecto a la hora de decidir la complejidad de la caracterización.

### 3.2.1. Problemas del modelado de tiempos de cómputo variables

El estudio del rendimiento de un circuito asíncrono complejo requiere de simulaciones a nivel arquitectónico[2]. Este tipo de simulaciones permiten obtener datos relevantes acerca del rendimiento como, por ejemplo, el tiempo de cómputo promedio global. Las simulaciones a nivel arquitectónico resultan complejas puesto que los tiempos de cómputo de cada módulo del circuito varían en función del dato procesado.

Una solución trivial para modelar los tiempos de cómputo variables constaría de los siguientes pasos:

1. Obtener, a través de simulaciones del circuito descrito a nivel de puertas, los tiempos de cómputo de todos los posibles datos de entrada para cada módulo dentro del circuito.

2. Almacenar los tiempos obtenidos en una estructura convenientemente indexada para hacer corresponder a cada dato su valor de tiempo asociado.

[2]El siguiente capítulo ofrece mayor detalle sobre tipos de simulaciones en el estudio del rendimiento de un circuito.

3. Utilizar en las simulaciones del circuito asíncrono la estructura que almacena los tiempos de cómputo, haciendo que cada dato tome como retardo su valor real de tiempo de cómputo.

Esta solución no es viable por los siguientes motivos:

- Largos tiempos de simulación para las descripciones a bajo nivel: las simulaciones de circuitos descritos a nivel de puertas son, generalmente, muy lentas. Esta lentitud se debe a la necesidad de incluir y considerar en las simulaciones información sobre los retardos de todos los componentes descritos.

- Elevado número de simulaciones de bajo nivel a realizar: el número de simulaciones a ejecutar sobre el circuito descrito a bajo nivel puede ser muy elevado, tanto como posibles combinaciones de bits haya en la entrada del circuito.

- Excesivo tamaño en estructuras de datos: serán necesarias enormes estructuras donde almacenar los retardos de cada dato de entrada en cada módulo. Como ejemplo se puede considerar un sencillo sumador asíncrono de dos operandos de 16 bits. Este sumador admite 4.294.967.296 pares de valores distintos ($2^{16} * 2^{16}$ pares de valores), cuyos tiempos de cómputo se representarían con números en punto flotante. Según el estándar de IEEE, los números en punto flotante de precisión simple ocupan 4 bytes, por lo que almacenar todos los posibles retardos de este sumador necesitaría de una estructura de datos que ocuparía, como mínimo, 16 Gbytes.

- Largos tiempos de simulación para el modelado arquitectónico: la utilización de los retardos en las simulaciones arquitectónicas del circuito bajo estudio probablemente no cumplirá los requisitos de tiempo de simulación. Dada la gran cantidad de retardos que posiblemente se utilizarán y las enormes estructuras de datos que se necesitan para contenerlos, obtener el retardo que cada dato empleó en cada módulo llevará un tiempo nada despreciable. Sumar todos estos tiempos de búsqueda supondrá una excesiva ralentización de las simulaciones del sistema.

Por tanto, es necesaria una alternativa eficiente para la caracterización del tiempo de cómputo de los componentes de un circuito asíncrono, capaz de integrarse en un simulador arquitectónico. En esta tesis se propone, en primer lugar, la utilización de funciones de distribución de probabilidad como solución a este problema. En segundo lugar se propone su integración en el simulador arquitectónico bajo estudio.

### 3.2.2. Antecedentes sobre funciones de distribución

Las funciones de distribución de probabilidad (FDPs) son potentes herramientas estadísticas capaces de describir la probabilidad que tiene una determinada variable de tomar diferentes valores. Esta propiedad fundamental, por tanto, encaja perfectamente en la descripción del problema de la caracterización de las variaciones del tiempo de cómputo.

Las FDPs se han utilizado profusamente en diversos campos de la ciencia. De hecho, muchos fenómenos de la naturaleza se pueden describir a través de FDPs. Por ejemplo, en relación al estado energético de una partícula, se pueden utilizar tres funciones distintas [Bla92]: la distribución de Maxwell-Boltzmann, la distribución de Bose-Einstein y la distribución de Fermi-Dirac. Otras áreas de la ciencia también confían en las FDPs para el modelado de distintos parámetros que ofrecen variaciones. Sirvan como ejemplo los siguientes trabajos: fluctuaciones en canales de agua [BWZ92], estudio de variaciones en nubes [YCK+94], simulaciones de construcciones [FKS99] y validación de satélites con sistemas sensores remotos [LL06]. En el campo de la Informática, las FDPs se han utilizado, por ejemplo, para modelar el comportamiento del tráfico de paquetes en redes TCP/IP [GW00, CHAG05, FCFW05].

El análisis del rendimiento de sistemas asíncronos también se ha beneficiado de la utilización de técnicas probabilísticas. A continuación se analizan brevemente, en orden cronológico, algunos antecedentes relacionados con el presente trabajo.

- Xie y Beerel estudiaron técnicas de análisis simbólico del rendimiento de sistemas asíncronos en [XB00, XKB99, XB97]. En estos artículos, los autores trabajan con representaciones de alto nivel de abstracción de los circuitos

analizados. Estas representaciones van desde máquinas de estados finitas probabilistas hasta lo que los autores denominan *redes de Petri estocásticas* (*STPN*, *STochastic Petri Nets*). Estas *STPN* son redes de Petri donde las transiciones incluyen valores de retardo y probabilidades asociadas. Las descripciones de los circuitos que utilizan son de muy alto nivel. A estas descripciones se les asocian expresiones simbólicas para las FDPs, de modo que es necesario obtener estas expresiones para cada nodo del circuito. Este requisito es la principal desventaja de la técnica propuesta, puesto que representar las FDPs con fórmulas se puede convertir en un problema considerable en circuitos complejos.

- Chakraborty y Anglish presentan en [CA02] un análisis probabilista del retardo para sistemas asíncronos. En este análisis se utilizan grafos marcados (*marked graphs*) para la descripción de los circuitos a analizar. El análisis consiste en aplicar simulaciones de tipo Montecarlo a los valores de media y varianza que caracterizan el comportamiento de cada elemento de un sistema. El resultado obtenido es un intervalo de posibilidades para la media y varianza del circuito en conjunto. El objetivo de este análisis probabilista, por tanto, no es determinar el rendimiento sino verificar la temporización de los circuitos. Como principal inconveniente se destaca la necesidad de realizar gran número de simulaciones debido a la utilización del método de Montecarlo.

- En [MNCJ05], McGee *et al.* describen un método de análisis del rendimiento asintótico de sistemas asíncronos modelados a través de grafos marcados. En este caso, los grafos marcados corresponden a una subclase de redes de Petri que capturan la concurrencia y las relaciones de dependencia entre componentes en sistemas libres de decisiones (*decision-free systems*) [CHEP71]. Las variaciones en los tiempos de llegada de los datos de entrada y en los tiempos de cómputo de los componentes del sistema se modelan utilizando FDPs. Este método de análisis, sin embargo, tiene dos restricciones. En primer lugar, sólo permite la utilización de distribuciones exponenciales para los retardos, inadecuadas para describir el comportamiento de multitud de circuitos asíncronos. En segundo lugar, el método se ciñe a los sistemas asíncronos libres de decisiones, lo que dificulta su extensión a microarqui-

tecturas complejas, según se explica en el propio trabajo.

Resumiendo, las FDPs se han utilizado fundamentalmente en descripciones de alto nivel de circuitos asíncronos, no en estudios del rendimiento de circuitos complejos. Además, habitualmente se ha utilizado la expresión simbólica que describe la propia FDP para caracterizar el tiempo de cómputo variable. Este requisito impide la particularización de las FDPs, por ejemplo, con funciones discretas.

### 3.2.3. Resumen de la propuesta

Una vez analizado el problema, se intuye que la utilización de FDPs en la caracterización del tiempo de cómputo de los circuitos asíncronos también puede ser una solución eficiente y viable en simuladores a nivel arquitectónico. La propuesta realizada en este capítulo consiste, por tanto, en caracterizar el tiempo de cómputo de los módulos de un circuito asíncrono utilizando FDPs. Las premisas de esta propuesta son las siguientes:

- El tiempo de cómputo de cada módulo del circuito se caracterizará con una FDP.
- Las FDPs se describirán como listas de retardos con una probabilidad asociada. No se utilizarán fórmulas o expresiones simbólicas para las FDPs.
- Para cada dato procesado, el simulador arquitectónico propondrá un tiempo de cómputo en base a la FDP asociada al módulo.
- El retardo que el simulador propondrá para cada dato probablemente no se corresponderá con su tiempo de cómputo real. Sin embargo, los tiempos de cómputo con mayor probabilidad se asignarán a más datos que aquellos tiempos de cómputo con menor probabilidad, y viceversa.

Como resultado de esta propuesta, el comportamiento de cada módulo seguirá el patrón marcado por la FDP a lo largo de la ejecución de la simulación. Este resultado, por tanto, será análogo al obtenido con la solución trivial descrita anteriormente.

## 3.3. Método de caracterización

En este apartado se explica el método de caracterización basado en FDPs que se propone en esta memoria. En primer lugar se describen los conceptos estadísticos relacionados, para reflexionar después sobre una medida acerca de la precisión de la caracterización. En segundo lugar, se presenta el método de caracterización describiendo los cuatro pasos en los que se divide.

### 3.3.1. Conceptos estadísticos relacionados

Los conceptos estadísticos relacionados con el método propuesto se muestran a continuación de manera incremental, siguiendo los pasos que conformarán la caracterización del tiempo de cómputo.

- ***Muestra*** (en este contexto). Subconjunto del conjunto de todos los posibles valores para una variable aleatoria dada. A partir de la muestra de una variable aleatoria es posible obtener la FDP que la caracteriza.

  Con objeto de ilustrar los conceptos aquí presentados, se utilizará un pequeño ejemplo a lo largo de todo este apartado. En este ejemplo se considera el tiempo de cómputo de un módulo asíncrono como la variable a caracterizar, y se presentan sus valores utilizando una medida de tiempo genérica llamada *unidades de tiempo*. El siguiente conjunto de valores $D$ corresponde a una posible muestra de retardos asociados al módulo asíncrono del ejemplo.

$$\begin{aligned} D \;=\; & \{\,0{,}22\,;\,0{,}75\,;\,1{,}12\,;\,1{,}3\,;\,1{,}51\,;\,1{,}55\,;\,1{,}56\,; \\ & 1{,}6\,;\,1{,}74\,;\,1{,}85\,;\,2{,}01\,;\,2{,}3\,;\,2{,}41\,;\,2{,}45\,;\,2{,}8\,\} \end{aligned}$$

- ***Histograma***. Gráfico donde aparecen clases de datos representadas por columnas, de modo que cada clase de datos se refiere a un valor o rango de valores de la variable aleatoria. La altura de cada columna se corresponde con el número de valores de la clase, es decir, el número de valores de la variable que pertenecen a la clase representada por la columna. Este dato también se denomina frecuencia absoluta de la clase.

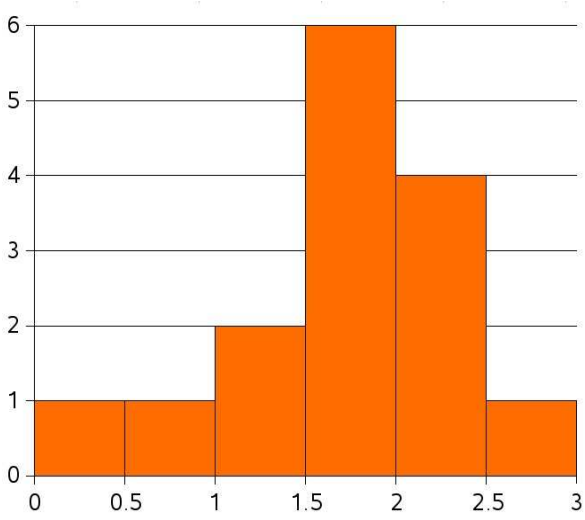


Figura 3.1: Histograma para la muestra $D$, considerando seis clases (columnas) de 0,5 unidades de tiempo de anchura.

Suponiendo que el conjunto de ejemplo $D$ sea una muestra de retardos obtenida de los tiempos de cómputo de un módulo asíncrono, es posible construir un histograma a partir de él. Para ello, se considerará, por ejemplo, que las clases (columnas) del histograma representan intervalos de 0,5 unidades de tiempo. De esta manera, la primera clase representará a los retardos de 0 a 0,5 unidades de tiempo, la segunda representará a los retardos de 0,5 a 1 unidades de tiempo, y así sucesivamente. El histograma resultante para el ejemplo anterior se muestra en la Figura 3.1.

La anchura y el número de clases del histograma corresponden a lo que se denominará *granularidad de la caracterización*. Cuanto mayor sea el número de clases, mayor precisión habrá en la caracterización del tiempo de cómputo. Si el número de clases es alto, el intervalo al que se refiere cada clase tendrá una anchura pequeña, por lo que se estará utilizando una granularidad fina. Por el contrario, un número pequeño de clases corresponde a una menor precisión debido a la mayor anchura de los intervalos a los que se refiere cada clase. Por tanto, se estará utilizando una granularidad gruesa.

- ***Función de Distribución de Probabilidad (FDP)***. La FDP de una va-

riable aleatoria discreta $X$ es una función $p$ tal que devuelve la probabilidad de que la variable $X$ sea igual a $x_i$. Esto es, para cada valor $x_i$ ,

$$p(x_i) = P(X = x_i)$$

La función de distribución de probabilidad se llama también función de masa, y satisface las siguientes condiciones:

$$\forall i,\ 0 \leq p(x_i) \leq 1$$

$$\Sigma_{i=0}^{\infty}\ p(x_i) = 1$$

Para una variable aleatoria discreta, la FDP es una lista de probabilidades asociadas a cada uno de los posibles valores de la variable. Esta definición correspondería a una caracterización con la granularidad más fina posible, donde se considera que cada valor de la variable forma una clase individual, con una probabilidad asociada. Sin embargo, de manera análoga a la descripción de los histogramas, es posible utilizar caracterizaciones de granularidad más gruesa. En estas caracterizaciones la FDP equivale a una lista de probabilidades asociadas a intervalos de valores de la variable, es decir, clases similares a las definidas en los histogramas. Para cada clase se elige un valor representante al que se asocia la probabilidad de la clase.

Partiendo de un histograma donde se representan las frecuencias absolutas de un conjunto de clases de una variable aleatoria, es posible calcular las frecuencias relativas de cada clase respecto del total de valores de la muestra. De este modo se obtienen las distintas probabilidades de que el tiempo de cómputo del módulo corresponda a cada uno de los intervalos de tiempo representados por cada clase.

Siguiendo con el ejemplo anterior donde se han definido clases al construir el histograma, una suposición conservadora asignaría como valor representante de cada clase el valor de su mayor retardo. De esta manera, la clase que representa los retardos desde 0 unidades de tiempo a 0,5 tomaría el 0,5 como valor de clase; la clase de 0,5 hasta 1 tomaría el 1, y así sucesivamente. Calculando las frecuencias relativas para cada clase en el ejemplo se obtiene la FDP que se representa en la Figura 3.2.

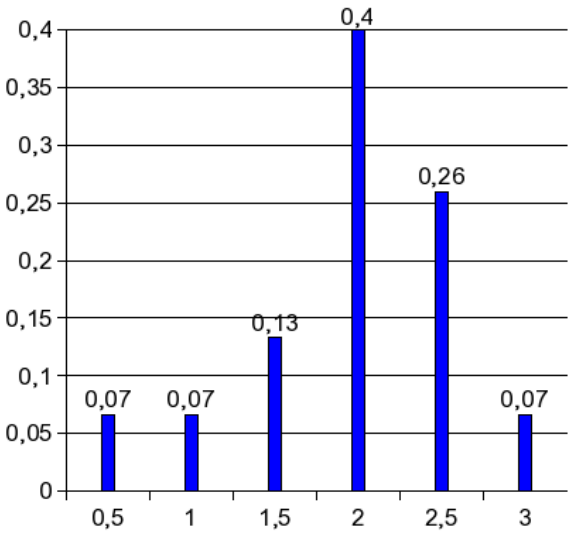


Figura 3.2: Función de distribución de probabilidad construida a partir del histograma de la Figura 3.1.

De este punto en adelante, todas las FDPs utilizadas en el método de caracterización se referirán siempre al tiempo de cómputo como una variable aleatoria discreta.

### 3.3.2. Medida de la calidad de la muestra

La muestra tomada para la caracterización de la variable aleatoria bajo estudio tiene una importancia crucial, puesto que representará a toda una población de valores. En consecuencia, es necesario verificar el grado de representatividad de la muestra. Por tanto, el objetivo de este apartado es obtener una expresión para medir la calidad de la muestra. Esta medida permitirá verificar matemáticamente el grado de representatividad o similitud entre una muestra de valores de un tamaño determinado y la población de la que procede.

En la caracterización del tiempo de cómputo como variable aleatoria, se utiliza esta medida para determinar la similitud entre los valores de retardo de la muestra y el comportamiento real del circuito del que fue tomada. Evidentemente, cuanto mayor sea el tamaño de la muestra, *i.e.*, el número de valores de retardo considerados, mayor será la similitud de la FDP resultante con el comportamiento real del circuito caracterizado. Sin embargo, no es deseable considerar muestras

demasiado grandes puesto que, dependiendo de la granularidad escogida, podrían dar lugar a FDPs de gran tamaño que ralentizarían la ejecución del simulador arquitectónico que las utilizara.

La obtención de una expresión para la medida que permite verificar si una muestra representa fielmente la variable aleatoria caracterizada, en este caso, el tiempo de cómputo, se plantea como un problema estadístico. A continuación se presenta la resolución de este problema.

**Enunciado:** se desea caracterizar una variable aleatoria dada, para lo cual es necesario estimar los parámetros principales de su población: la media, $\mu$, y la varianza, $\sigma^2$. Estos parámetros se estiman a partir de una muestra aleatoria, $M$, de tamaño $m$. Los datos para este problema son los siguientes:

- *Variable bajo estudio*, $T$: tiempo de cómputo de un circuito asíncrono dado.
- *Población*, $\Omega$: conjunto de todos los posibles valores de la variable $T$, *i.e.*, todos los posibles tiempos de cómputo para el circuito asíncrono dado.
- *Muestra*, $M$: subconjunto de $m$ valores tomados aleatoriamente de $\Omega$.

$$M = \{T_1, T_2, \ldots, T_m\}$$

La estimación de $\mu$ se puede realizar utilizando la media muestral, $E$, que es un estimador no sesgado para la media de la población. La media muestral de una población de $m$ elementos se define como

$$E = \frac{1}{m} \sum_{i=1}^{m} T_i$$

La verificación de que el tamaño de la muestra es suficientemente grande como para representar fielmente a la población de la variable caracterizada se realiza comprobando si la diferencia entre la media muestral y la media de la población es menor que un determinado valor $\varepsilon$, con una probabilidad lo más alta posible. En general, una probabilidad mayor que 0,95 sería aceptable, de modo que en los cálculos realizados se ha fijado una probabilidad aún mayor, del 0,97. Esta condición se representa matemáticamente de la siguiente manera:

$$P(|E-\mu| < \varepsilon) = 0,97 \tag{3.2}$$

Por otro lado, de acuerdo con el *Teorema Central del Límite* [MFJ92], es posible afirmar que la función de distribución de la media muestral para poblaciones grandes (más de cien elementos), se aproxima con una distribución Normal estándar. Este enunciado se expresa simbólicamente como:

$$\frac{E-\mu}{\frac{S}{\sqrt{m}}} \simeq N(0,1) = Z$$

donde $N(0,1)$ representa la distribución Normal estándar (con media cero y varianza uno), llamada también $Z$ ; y $\frac{S}{\sqrt{m}}$ representa el *error estándar de la media*[3]. La cuasi-desviación típica muestral, $S$ , se obtiene a partir de la cuasi-varianza muestral, $S^2$ , que a su vez se define como

$$S^2 = \frac{1}{m-1}\sum_{i=1}^{m}(T_i - E)^2$$

Así, dividiendo ambos términos de la probabilidad que se muestra en la fórmula (3.2) por el error estándar de la media, el resultado es el siguiente:

$$P\left(\frac{|E-\mu|}{\frac{S}{\sqrt{m}}} < \frac{\varepsilon}{\frac{S}{\sqrt{m}}}\right) = 0,97 \tag{3.3}$$

Puesto que el *Teorema Central del Límite* permite utilizar en este caso la distribución Normal, se transforma la fórmula (3.3), en la siguiente expresión:

$$P(|Z| < c) = 0,97 \text{ donde } c = \frac{\varepsilon}{\frac{S}{\sqrt{m}}}$$

Utilizando las tablas tipificadas para la función de distribución Normal, se comprueba que el valor requerido para que se verifique la expresión anterior es $c = 2,17$. Por tanto,

[3]Al suponer poblaciones grandes, el *Teorema Central del Límite* permite sustituir la desviación típica ($\sigma$) por la cuasi-desviación típica muestral ($S$) en el error estándar de la media $\left(\frac{\sigma}{\sqrt{m}}\right)$ [MFJ92].

$$\frac{\frac{\varepsilon}{S}}{\frac{1}{\sqrt{m}}} = 2,17$$

En consecuencia, despejando $m$, el tamaño mínimo de una muestra donde la diferencia entre la media de la población y la media muestral, en valor absoluto, es menor que $\varepsilon$ con una probabilidad de 0,97 , se obtiene de la siguiente fórmula:

$$m = \left(\frac{S * 2{,}17}{\varepsilon}\right)^2 \tag{3.4}$$

donde $S$ se calcula a partir de los valores de la muestra, y $\varepsilon$ corresponde al margen de error deseado, en las mismas unidades que los valores de la muestra.

La medida definida por la fórmula (3.4) permite comprobar, verificando el tamaño de la muestra, si ésta representa fielmente a la población de la que se obtuvo.

### 3.3.3. Descripción del método propuesto

El método de caracterización del tiempo de cómputo de un circuito utilizando FDPs seguirá una secuencia de pasos que se extraen de lo explicado en el Apartado 3.3.1.

Así, considerando el tiempo de cómputo como una variable aleatoria discreta, los pasos a seguir para obtener la FDP son los siguientes:

1. Obtención de la muestra de retardos del circuito asíncrono a caracterizar, verificando que el tamaño de la muestra es mayor o igual que el indicado en la fórmula (3.4).

2. Establecimiento de la granularidad del histograma, cuyo valor condiciona la precisión de la FDP.

3. Generación del histograma a partir del conjunto de retardos de la muestra.

4. Construcción de la FDP calculando las frecuencias relativas para cada clase del histograma.

Una vez generada, la FDP ofrece toda la información necesaria acerca del comportamiento del módulo, puesto que asocia un valor de probabilidad a cada clase o intervalo de retardos.

El método aquí presentado es muy flexible. Como se ha mostrado, permite describir la variabilidad del tiempo de cómputo de un circuito asíncrono con una precisión dependiente de la granularidad escogida. Además, las FDPs también permiten caracterizar el tiempo de cómputo de un circuito síncrono. Esto se consigue llevando la granularidad al caso más grueso posible, donde la FDP tendría una sola clase a la que asociar un valor de probabilidad. La clase tomaría el valor de probabilidad uno porque todos los datos se asociarían al retardo indicado por el representante de clase. El valor de retardo representante de la clase debería ser el mayor retardo posible en el circuito, es decir, el retardo del camino crítico. Esta caracterización de grano grueso reflejaría, como se ha indicado, el comportamiento de un circuito síncrono, donde todos los datos se procesan utilizando el mismo tiempo de cómputo, correspondiente al peor caso posible de rendimiento.

El siguiente problema a resolver es la integración de las FDPs en un simulador arquitectónico de manera que la información ofrecida por cada función se pueda aprovechar eficientemente en las simulaciones. La integración de esta caracterización en un simulador arquitectónico se explica detalladamente en el siguiente capítulo.

## 3.4. Caracterización de un sumador asíncrono

Con objeto de ilustrar la utilización del método propuesto en el apartado anterior, a continuación se describe su aplicación a un caso práctico. Este caso práctico consiste en caracterizar el tiempo de cómputo de un sumador asíncrono desarrollando todos los pasos del método expuestos anteriormente.

Más concretamente, se caracterizará el tiempo de cómputo de un sumador Kogge-Stone [KS73] de 64 bits. En la Figura 3.3 se muestra el esquema del árbol de cálculo de acarreos de un sumador Kogge-Stone de 16 bits, análogo a la versión de 64 bits, pero incluyendo dos etapas menos. Los círculos del diagrama representan la lógica que implementa las operaciones de generación y propagación de acarreos.

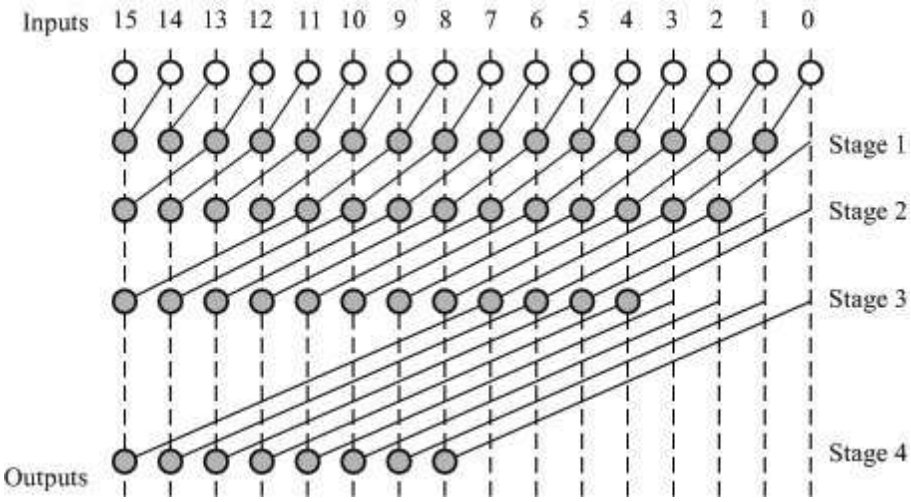


Figura 3.3: Esquema de un sumador Kogge-Stone de 16 bits. Este sumador tiene una profundidad lógica mínima, con el mínimo *fan-out*. Los círculos del diagrama representan las típicas operaciones de generación y propagación de acarreos.

| ***Netlist*** | ***Área (cell units)*** | ***Max. Retardo (ns)*** |
|---|---|---|
| *Dm* | 81811,29 | 2,961 |
| *Am* | 25261,89 | 18,692 |

Tabla 3.1: Datos principales sobre los *netlists* obtenidos a partir de la síntesis de la descripción RTL de un sumador Kogge-Stone de 64 bits bajo la restricción de mínimo retardo, *Dm*, y de mínimo área, *Am*.

Para realizar la caracterización, previamente se sintetizó la descripción a nivel de transferencia de registros de este sumador utilizando Synopsys Design Compiler® y una librería de celdas estándar[4] sobre tecnología CMOS TSMC de 0,25 $\mu$m y 2,5 V. Se realizaron dos síntesis distintas, siguiendo dos restricciones diferentes: mínimo retardo, *Dm*, y mínimo área, *Am*. En la Tabla 3.1 se resumen los datos obtenidos para ambos *netlists*[5] del sumador.

Para cada una de las implementaciones se ha aplicado el mismo método de caracterización, obteniendo como resultado dos FDPs distintas. En los siguientes apartados se detallan los pasos seguidos en la caracterización de ambas síntesis del circuito sumador.

[4]Se utilizó la librería VTVT, desarrollada por *Virginia Tech VLSI for Telecommunications*.

[5]Se utilizará el término *netlist* para denotar el resultado de la síntesis de un circuito, es decir, su descripción a nivel de puertas.

### 3.4.1. Obtención de la muestra

El primer paso en la caracterización del tiempo de cómputo de un circuito es la obtención de una muestra de tiempos. En el caso presentado como ejemplo, se ejecutaron simulaciones de ambos *netlists* del sumador, con anotaciones de retardos (*back-annotation*), en uno de los extremos (*corner*) críticos en el rendimiento del circuito: proceso lento, elevados valores de parásitos y 125ºC de temperatura. Este es uno de los extremos críticos que es preciso comprobar para validar que la temporización del circuito se sigue satisfaciendo incluso en las peores condiciones posibles.

Las simulaciones ejecutadas se realizaron tomando un conjunto representativo de vectores de entrada para ambos circuitos. Así, los tiempos de cómputo obtenidos para estos datos de entrada forman las muestras de tiempos para *Dm* y *Am*. Teniendo en cuenta que el tamaño de la población de los datos de entrada de un sumador de 64 bits es de $2^{128}$ pares de operandos, inicialmente se obtuvo el tiempo de cómputo para un millón de pares de datos de entrada en cada *netlist* del sumador. Esta cantidad corresponde a una pequeña fracción de la población, por lo que es necesario determinar si este tamaño de muestra es suficientemente representativo o no lo es. Para ello, se aplicó la medida descrita en el Apartado 3.3.2 a las dos muestras obtenidas.

Para aplicar esta medida se debe fijar, en primer lugar, el valor absoluto de la diferencia máxima entre la media de la población y la media muestral. Este valor, denotado en la medida como $\varepsilon$, se fijó en este caso a 0,01 nanosegundos (ns). Este retardo es menor que el retardo de un inversor en la tecnología utilizada[6], por lo que $\varepsilon$ se establece a un valor inferior a una puerta lógica. En segundo lugar se determina la probabilidad con la que la diferencia entre las medias se encontrará por debajo de $\varepsilon$ . Esta probabilidad se fija, como se hizo en la anterior descripción de la medida, a 0,97. Una vez establecidos estos valores, es necesario obtener los valores de la media y varianza muestrales.

En la Tabla 3.2 se presentan los valores obtenidos con *Matlab* [The07] para la media y varianza muestrales de las dos muestras. El dato del mínimo tamaño que

[6]Un inversor en tecnología TSMC 0,25 $\mu$m con una carga mínima de una puerta emplea, al menos, 0,038 ns [KAAK05]

| *Netlist* | $E$ | $S^2$ | $\varepsilon$ | $m$ |
|---|---|---|---|---|
| *Dm* | 1,7342 | 0,0332 | 0,01 | ~ 1564 |
| *Am* | 2,4698 | 0,3581 | 0,01 | ~ 16863 |

Tabla 3.2: Estadísticas para las muestras de los circuitos *Dm* y *Am*. Los valores de media muestral ($E$) y varianza ($S^2$) se representan en nanosegundos para ambas muestras. El dato del tamaño mínimo para que cada muestra sea representativa ($m$) se calcula sobre un factor $\varepsilon$ de 0,01 nanosegundos utilizando la fórmula (3.4).

debe tener cada muestra para ser representativa, denotado como $m$, e incluido también en la tabla, se ha calculado utilizando la fórmula (3.4), proveniente del Apartado 3.3.2:

$$m = \left( \frac{S * 2{,}17}{\varepsilon} \right)^2$$

Según esta medida, una muestra de al menos 1.564 tiempos es capaz de representar fielmente el comportamiento del circuito *Dm*, con una probabilidad de 0,97 de que la diferencia entre la media muestral y la media de la población esté por debajo de los 0,01 ns. Del mismo modo, para el circuito *Am* la muestra deberá tener un tamaño de, al menos, 16.863 tiempos de cómputo. Por tanto, y según la medida utilizada, las muestras obtenidas para ambos circuitos son suficientemente representativas, puesto que su tamaño es de un millón de valores de tiempo cada una.

### 3.4.2. Granularidad del histograma

El segundo paso en el método de caracterización consiste en decidir la granularidad del histograma, es decir, establecer el número de clases en que se dividirá y, por tanto, cuál será la anchura de los intervalos de retardos. Cuanto mayor sea el número de clases, más precisa será la caracterización, puesto que habrá un mayor número de retardos con probabilidad asociada en la FDP. Es importante recalcar que la FDP resultante de la aplicación de este método tendrá como destino su integración en un simulador, por lo que su tamaño debe ser moderado.

En el ejemplo del sumador que se desarrolla en este apartado, se optó por una

granularidad fina en la caracterización para obtener una mayor precisión en la caracterización. Para ello, se tomó como referencia el retardo de la puerta lógica más pequeña en la tecnología escogida para la síntesis. Así, se estableció que la anchura de las clases fuese de 0,035 ns, cantidad que corresponde, aproximadamente, al retardo intrínseco de un inversor de 0,25 $\mu$m sin ninguna carga asociada. Para cada muestra el número de clases es distinto, puesto que éste depende del máximo valor obtenido en cada una de ellas. De esta manera, al dividir los valores de máximo retardo mostrados en la Tabla 3.1 entre 0,035, resulta que el número de clases del histograma de *Dm* es 85, mientras que el número de clases del histograma de *Am* es 535.

### 3.4.3. Generación del histograma

El tercer paso en la caracterización consiste en la generación del histograma a partir de los retardos de la muestra. Se puede ver en las figuras 3.4 y 3.5 los histogramas para las muestras de los circuitos *Dm* y *Am*, sintetizados a partir del diseño RTL del sumador.

El histograma para *Dm* muestra que el circuito genera los resultados rápidamente para la mayoría de los datos de entrada. Concretamente, el 90,4 % de los datos emplean tiempos de cómputo por debajo de 1,995 ns, mientras que la media muestral es 1,734 ns y el máximo retardo es 2,961 ns.

La diferencia entre la media muestral y el retardo máximo es mayor aún en el circuito *Am*, sintetizado bajo la restricción de mínimo área. En este caso, el 90,6 % de los datos emplean tiempos de cómputo por debajo de 3,255 ns, con una media muestral de 2,469 ns y un caso peor de 18,692 ns.

Como era de esperar, ambos histogramas, pero especialmente el del circuito *Am*, muestran que el tiempo de cómputo del sumador depende fuertemente de los datos.

### 3.4.4. Construcción de la función de distribución

El cuarto y último paso del método consiste en generar la FDP partiendo de las frecuencias relativas calculadas para cada una de las clases del histograma. Estas

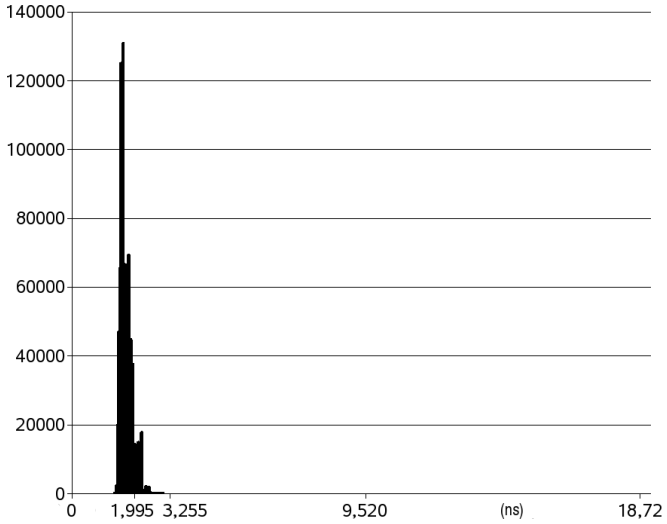


Figura 3.4: Histograma para la muestra de tiempos de cómputo para un sumador Kogge-Stone de 64 bits sintetizado bajo la restricción de mínimo retardo ($Dm$). El eje de las ordenadas muestra frecuencias absolutas para cada clase, mientras que en el eje de las abscisas se muestra, en ns, los retardos representantes de cada clase. El retardo del camino de crítico para $Dm$ es de 2,961ns.

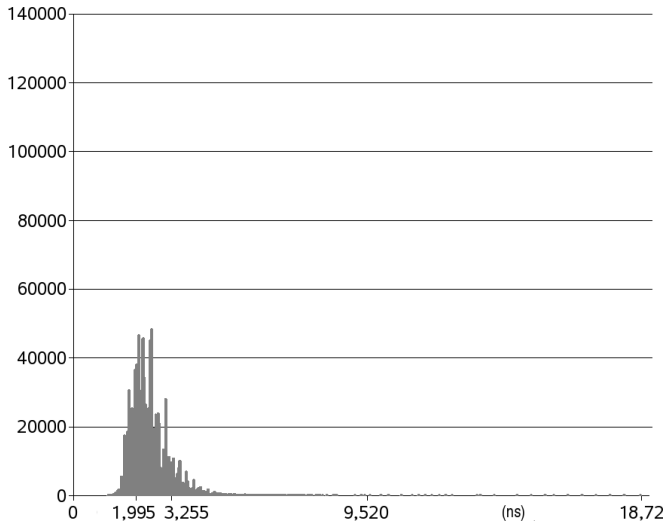


Figura 3.5: Histograma para la muestra de tiempos de cómputo para un sumador Kogge-Stone de 64 bits sintetizado bajo la restricción de mínimo área ($Am$). El eje de las ordenadas muestra frecuencias absolutas para cada clase, mientras que en el eje de las abscisas se muestra, en ns, los retardos representantes de cada clase. El retardo del camino de crítico para $Am$ es de 18,692ns.

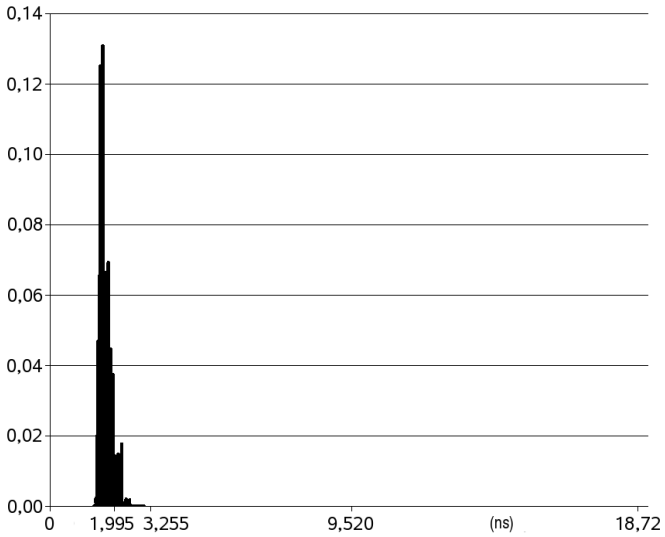


Figura 3.6: FDP para el histograma del sumador Kogge-Stone de 64 bits sintetizado bajo la restricción de mínimo retardo ($Dm$). El eje de las ordenadas muestra la probabilidad para cada clase, mientras que en el eje de las abscisas se muestra, en ns, los retardos representantes de cada clase definida para la función.

frecuencias representan la probabilidad de que un dato determinado emplee un cierto retardo de los que pertenecen a la clase.

Las figuras 3.6 y 3.7 muestran las FDPs obtenidas para los dos histogramas generados a partir de las dos muestras de tiempos de cómputo del sumador Kogge-Stone de 64 bits, correspondientes a las síntesis bajo las restricciones de mínimo retardo y mínimo área, respectivamente.

Finalmente, se validó la corrección de la metodología verificando que tanto los histogramas como las FDPs eran matemáticamente equivalentes[7] a aquellos generados a partir de muestras aleatorias de tiempos de cómputo de tamaños 1.564 y 16.863, tomadas de simulaciones de bajo nivel de entradas generadas aleatoriamente para los circuitos $Dm$ y $Am$ respectivamente.

[7]Para el mismo rango de valores e iguales valores máximo y mínimo, las funciones presentaban la misma media y desviación típica.

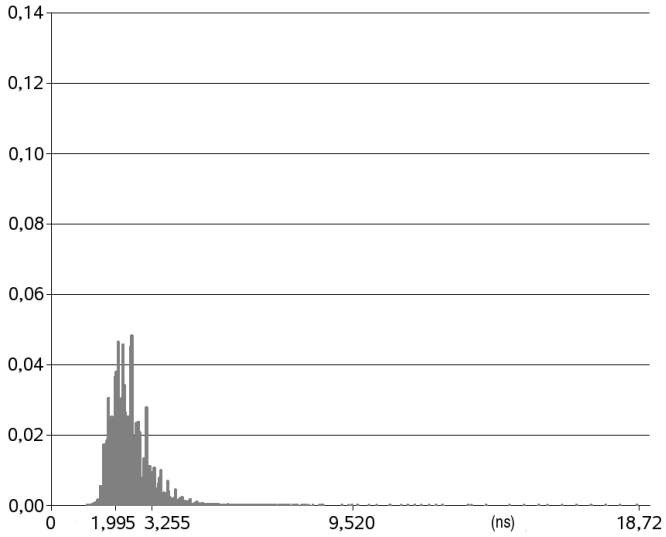


Figura 3.7: FDP para el histograma del sumador Kogge-Stone de 64 bits sintetizado bajo la restricción de mínimo área ($Am$). El eje de las ordenadas muestra la probabilidad para cada clase, mientras que en el eje de las abscisas se muestra, en ns, los retardos representantes de cada clase definida para la función.

### 3.4.5. Resumen y análisis del caso práctico

En resumen, como resultado de la caracterización se han obtenido dos FDPs que forman representaciones fiables del comportamiento de los dos circuitos que implementan el sumador. Las FDPs están compuestas por pares retardo-probabilidad que se pueden utilizar en posteriores simulaciones de sistemas asíncronos que incluyan un sumador Kogge-Stone de 64 bits como el descrito en este ejemplo.

La caracterización obtenida en este ejemplo en particular verifica la determinante influencia de las restricciones impuestas durante la síntesis del circuito. Como se puede apreciar en las FDPs, ambas implementaciones tienen distintos retardos máximos pero similar rendimiento promedio. Concretamente, $Am$ es alrededor de un 631 % más lento que $Dm$ en su caso peor, mientras que para el caso medio, $Am$ es sólo un 42,2 % más lento que $Dm$. Además, como se indica en la Tabla 3.1, existe una gran diferencia en la cantidad de área ocupada entre ambos *netlists* del sumador. Esta diferencia corresponde a una cantidad de área para $Dm$ aproximadamente 3,23 veces mayor que para $Am$.

Este resultado en la caracterización puede llevar al siguiente análisis arquitectónico: en un circuito asíncrono donde exista un sumador Kogge-Stone de 64 bits sintetizado bajo la restricción de mínimo retardo, es posible incrementar el rendimiento del sistema reemplazando un sumador *Dm* por tres *Am*. Esta sustitución acarrearía una ligera penalización de área debida a la lógica necesaria para implementar tres posibles caminos en los datos. El rendimiento del circuito completo se podría estudiar con un simulador arquitectónico donde el tiempo de cómputo de sus componentes se caracterizase a través de FDPs.

## 3.5. Resumen del método de caracterización

Los pasos en los que se divide el método de caracterización presentado se pueden resumir con el diagrama de flujo de la Figura 3.8.

En primer lugar, para obtener la muestra de retardos, o bien se realizan simulaciones de bajo nivel del circuito bajo estudio, o bien se estima su comportamiento con valores sintéticos de tiempos. Con esa información, la primera fase del método genera una muestra de retardos.

A partir de la muestra y considerando los objetivos de la caracterización, se determina la granularidad del histograma. Por ejemplo, si el objetivo principal es obtener una caracterización muy precisa, la granularidad debe ser fina. Por el contrario, si se prima la velocidad de ejecución del simulador que utilizará la caracterización, la granularidad del histograma debe ser gruesa.

Una vez determinada la granularidad y con ayuda de una aplicación que soporte funciones estadísticas, como por ejemplo *Matlab* [The07], se obtiene el histograma a partir de la muestra anterior.

Finalmente, de nuevo con ayuda de *Matlab* o de alguna otra herramienta capaz de obtener valores de probabilidad, se obtiene la FDP resultado de la caracterización.

## 3.6. Otras aplicaciones de las FDPs

A lo largo de este capítulo se ha detallado un método capaz de caracterizar la variación del tiempo de cómputo de un circuito asíncrono, dependiente de los

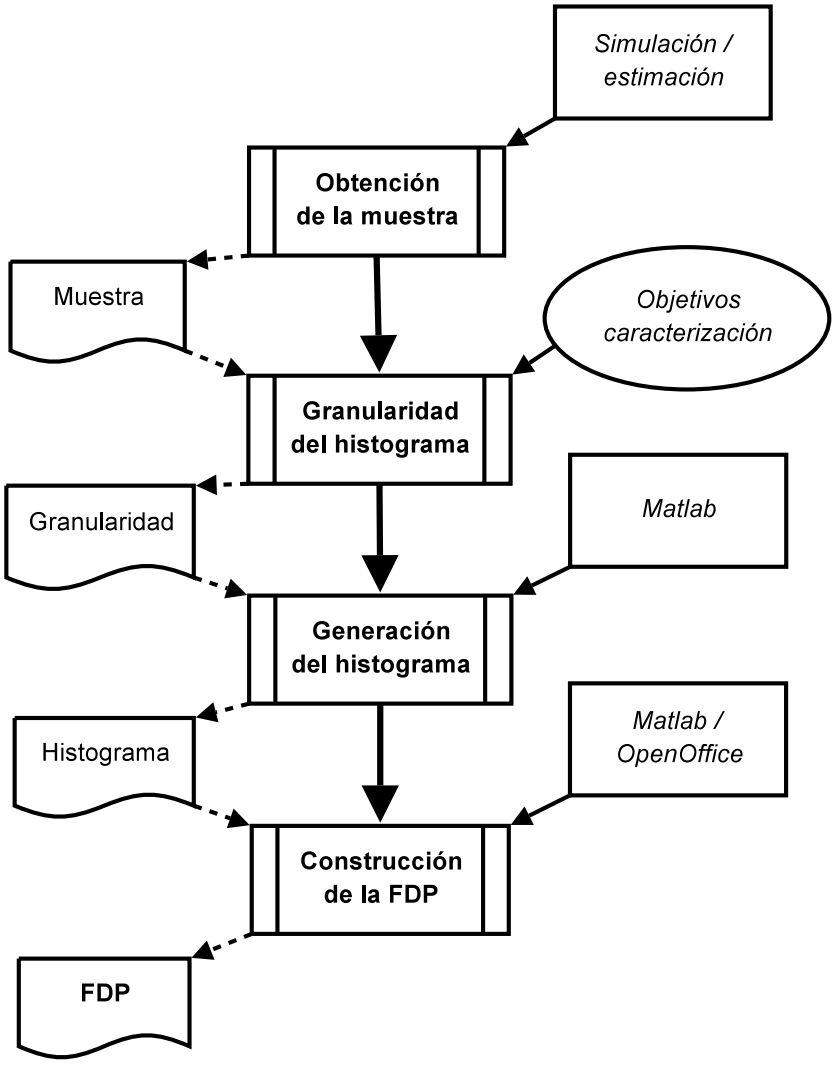


Figura 3.8: Diagrama de flujo para el método de caracterización de tiempo de cómputo basado en FDPs. Los cuatro pasos del método se representan en la columna central como rectángulos con doble marco. A la izquierda se representan las entradas y salidas de cada fase del método. En la derecha se muestran tanto herramientas (rectángulos) como decisiones (elipse) adicionales utilizadas en cada paso del método.

datos, utilizando FDPs. Sin embargo, como se explica en el Apartado 3.1, existen otros factores que influyen sobre la variabilidad del tiempo de cómputo. Estos factores son, principalmente, el proceso de fabricación, el voltaje de alimentación y la temperatura de funcionamiento.

Aún estando fuera de los objetivos de esta tesis, futuras aplicaciones de la metodología de caracterización se podrían relacionar con estos factores. Algunas de esas aplicaciones podrían ser las siguientes:

1. Obtención de muestras de retardos a partir de simulaciones a distintos voltajes y temperaturas de funcionamiento para un circuito asíncrono. De este modo se dispondría de un conjunto de FDPs para cada circuito, asociadas a distintas condiciones de funcionamiento ($PVT$).

2. Modelado de algoritmos adaptativos en un simulador arquitectónico. Utilizando conjuntos de FDPs como los que se describen en el punto anterior se podría ampliar la funcionalidad de un simulador arquitectónico:

   *a*) Variaciones de voltaje de alimentación: el simulador modelaría el funcionamiento de algoritmos adaptativos de ahorro de energía, estimando el rendimiento global del sistema cuando varios de sus módulos utilizan un voltaje de alimentación, por ejemplo, menor que el resto. El simulador debería utilizar, para cada módulo asíncrono, la FDP correspondiente a las condiciones de voltaje particulares de cada uno.

   *b*) Variaciones en la temperatura: de manera análoga se podría estimar el rendimiento de un sistema donde ciertos módulos varían su temperatura, afectando a su rendimiento individual, a través de la caracterización con las correspondientes FDPs.

3. Detección de puntos calientes (*hotspots*): una opción también interesante sería aquella en la que el propio simulador, dinámicamente, detectara el aumento en la utilización de un recurso o módulo asíncrono, y simulase un aumento en su temperatura. Para ello sólo tendría que sustituir la FDP que caracteriza su tiempo de cómputo por una más apropiada en esa circunstancia. Del mismo modo, se podrían detectar los módulos menos utilizados

y modelar una bajada en el voltaje de alimentación con un cambio análogo en su FDP.

Todas estas alternativas serán consideradas como trabajo futuro, puesto que representan una interesante línea en el campo de la estimación del rendimiento de circuitos asíncronos.

## Capítulo 4

# Simulación arquitectónica de sistemas asíncronos

El modelado de un procesador asíncrono de propósito general es una tarea compleja. Por un lado, es necesario modelar la ejecución de instrucciones en la microarquitectura, es decir, la funcionalidad del procesador. Por otro lado, un procesador asíncrono tiene un comportamiento que no depende de una señal global de reloj, sino que se guía por otros mecanismos de control y comunicación cuyos retardos varían en función de la carga de trabajo. Esta parte, correspondiente al modelado de la temporización del sistema, resulta tan importante como el modelado de la propia microarquitectura.

Existen diferentes estrategias de simulación tanto para la microarquitectura como para su temporización, por lo que se hace necesario decidir cuáles son las características más adecuadas para el simulador que implemente estos modelos. Estas características son, fundamentalmente, tres: el alcance del simulador, el tipo de entradas que recibirá y el nivel de detalle en las simulaciones.

En este capítulo se estudian, en primer lugar, las características de la simulación propuesta. Después se muestra el modelado de la microarquitectura y, a continuación, se describe la temporización que permite modelar el comportamiento asíncrono del sistema. Para finalizar, se describen algunos detalles sobre la implementación del simulador, así como la integración de la caracterización de la latencia variable utilizando funciones de distribución, según el método que se ha propuesto y validado en el Capítulo 3.

## 4.1. Propiedades de la simulación

A la hora de plantear la simulación de un sistema asíncrono complejo, surge la disyuntiva entre crear un nuevo simulador o bien adaptar uno de los simuladores ya existentes para sistemas síncronos. Para tomar esta decisión es necesario comparar las técnicas de simulación de sistemas síncronos con las necesidades que plantea la simulación de un sistema asíncrono complejo. La evaluación de estas necesidades guiará la elección entre distintas características o tipos de simulador.

Dentro de la amplia variedad de simuladores, existen una idea generalizada acerca de cuáles son las principales técnicas de simulación de microarquitecturas [CLSL02]. De esta idea se puede extraer una clasificación de simuladores en función de tres factores distintos: su alcance, el tipo de entradas que recibe y el nivel de abstracción del simulador. A continuación se presentan estas categorías, indicando el factor que se toma en consideración en cada clasificación.

Considerando el alcance o ámbito (*scope*) del simulador, se habla de:

- Simuladores de conjuntos de instrucciones: sólo modelan el repertorio de instrucciones de un procesador.
- Simuladores de sistemas completos: modelan procesador, memoria y parte del contexto de estos recursos. Pueden incluso simular la interacción con el sistema operativo.

Considerando el tipo de entrada que recibe el simulador, se distingue entre:

- Simuladores basados en traza (*trace-driven*): utilizan como entrada flujos de ejecución de instrucciones previamente definidos, donde los valores de los datos de entrada son invariables.
- Simuladores basados en ejecución (*execution-driven*): permiten la ejecución de distintos bloques de instrucciones, donde los datos de entrada pueden cambiar a voluntad del usuario.

Considerando el nivel de abstracción elegido para describir el sistema modelado, los simuladores se dividen en:

- Simuladores funcionales (*behavioral*): describen exclusivamente la funcionalidad del sistema modelado.
- Simuladores arquitectónicos (*architectural*): describen tanto la funcionalidad como los detalles de implementación y temporización de los componentes del sistema modelado.

A la hora de decidir las características del simulador a implementar en este trabajo, es necesario plantear en cuáles de las anteriores clasificaciones se encajará el simulador. Para tomar esta decisión, conviene revisar de nuevo uno de los objetivos que se plantean en la introducción de esta tesis. Este objetivo propone el *estudio y generación de medidas acerca del rendimiento de una microarquitectura asíncrona compleja: un procesador superescalar asíncrono de propósito general donde los tiempos de cómputo de sus componentes son variables.* Algunas de las medidas que se pretenden obtener acerca del rendimiento de esta microarquitectura son las siguientes:

- Evaluación de tiempo de cómputo total para cada simulación, en unidades de tiempo.
- Estadísticas sobre las instrucciones procesadas: valores máximos, mínimos y promedio sobre la latencia de las instrucciones dentro de la microarquitectura, rendimiento en términos de número de instrucciones por unidad de tiempo.
- Tasa de actividad de cada módulo de la microarquitectura.

Para llevar a cabo este objetivo, estrechamente relacionado con el estudio del rendimiento de la microarquitectura, el simulador se deberá incluir en las siguientes categorías:

- Simulador de sistema completo.
- Simulador basado en ejecución.
- Simulador arquitectónico.

La evaluación de cada una de estas categorías, así como sus ventajas e inconvenientes asociados, se discuten en los próximos apartados.

### 4.1.1. Simulación de sistema completo

Los simuladores de conjuntos de instrucciones, conocidos también como simuladores de arquitectura, simulan únicamente el repertorio de instrucciones de un procesador o microcontrolador. Este tipo de simuladores se suelen codificar utilizando lenguajes de alto nivel, de modo que imitan la funcionalidad del procesador manteniendo los valores de los registros del procesador en variables internas. La ejecución de simulaciones consiste en ir leyendo las instrucciones de entrada y realizando cambios en las variables internas según la instrucción leída. Las aplicaciones principales de estos simuladores son, por ejemplo, la comprobación puramente funcional de un conjunto de instrucciones o la verificación de compatibilidad de un procesador con versiones anteriores.

La mayor ventaja de los simuladores de conjuntos de instrucciones es su velocidad de ejecución, puesto que suelen tener en cuenta, principalmente, la funcionalidad de cada instrucción del repertorio simulado, no los detalles del *hardware* que están modelando. Estos simuladores, sin embargo, no modelan partes importantes de un sistema como son la memoria, los buses de comunicación o las llamadas al sistema operativo. Los resultados de las simulaciones, por tanto, no aportan información acerca de la influencia de estos factores en el rendimiento de la microarquitectura.

En los simuladores de sistemas completos, por el contrario, se incluye el modelado de los elementos principales del sistema. Estos elementos son, generalmente, el procesador, la memoria y el soporte necesario para procesar llamadas al sistema operativo. Este tipo de simuladores se suelen utilizar en el estudio tanto de la funcionalidad como del rendimiento de procesadores, puesto que su implementación modela con mayor detalle el *hardware* bajo estudio. Este modelo detallado permite la monitorización de los elementos del sistema y la generación de estadísticas acerca del rendimiento del mismo.

Como inconveniente se debe señalar que, dado que simulan un mayor número de elementos con mayor nivel de detalle, el tiempo de ejecución en las simulaciones también es mayor. Por tanto, la optimización del código de este tipo de simuladores es un punto fundamental a tener en cuenta.

Puesto que uno de los objetivos de la tesis es obtener estadísticas y medidas detalladas acerca de la microarquitectura y su entorno próximo, el simulador del

procesador superescalar asíncrono deberá modelar el sistema completo, entendiendo que en este ámbito se encuentran el procesador, la memoria y el soporte de llamadas al sistema operativo. Con este alcance, el simulador podrá generar un mayor número y variedad de estadísticas, así como reflejar mejor el comportamiento de la microarquitectura en un entorno completo. En consecuencia, se ha escogido la simulación del sistema completo como la variante más adecuada para este trabajo.

### 4.1.2. Simulación basada en ejecución

Según el tipo de valores de entrada de las simulaciones, las principales alternativas en la simulación de sistemas de cómputo son las simulaciones basadas en trazas (*trace-driven*) y las simulaciones basadas en ejecución (*execution-driven*). Las diferencias entre ambos tipos de simulación son muy significativas, y se ilustran a continuación.

Las simulaciones basadas en traza toman como valores de entrada un flujo de instrucciones previamente ejecutado en la misma microarquitectura, de manera que los valores de entrada y el comportamiento de la simulación son invariables en todas las simulaciones de la misma traza. La traza que se toma como entrada suele contener toda la información relacionada con la ejecución, desde los valores leídos de la memoria o el banco de registros, hasta el marcado de las instrucciones que serán desestimadas en la ejecución especulativa.

El mayor atractivo de este tipo de simulaciones está relacionado con la gran cantidad de información que las trazas almacenan. Gracias a ella, la utilización de trazas permite simplificar la ejecución de las simulaciones, por ejemplo, omitiendo la simulación de las instrucciones que se van a desestimar, o eliminando el modelado de estructuras internas de la microarquitectura como buses o registros. Estas peculiaridades han motivado que, tradicionalmente, las simulaciones basadas en traza se hayan utilizado en la evaluación del rendimiento de memorias cache, puesto que en este tipo de pruebas no es tan importante el código simulado como los accesos a memoria realizados.

Las simulaciones basadas en traza dependen en gran medida tanto del modelo utilizado para la microarquitectura, como de la información contenida en la traza

utilizada como entrada. De hecho, para conseguir simulaciones más rápidas, es habitual reducir la información proporcionada por la traza eliminando referencias al sistema operativo e incluso tomando partes aisladas de la traza total de una ejecución. Estas reducciones sobre la traza completa introducen un sesgo en los resultados de las simulaciones que pocas veces se estudia o se toma en consideración a la hora de presentar resultados [CFKM98, FNAT96]. Por tanto, el principal inconveniente de las simulaciones basadas en traza estriba en la obtención de trazas representativas, y en el elevado tamaño que pueden llegar a tener las trazas completas provenientes de la ejecución de bancos de pruebas estándar, llegando a veces a ocupar varios gigabytes de disco.

Como alternativa a este enfoque se encuentran las simulaciones basadas en ejecución. En este tipo de simulaciones se reproduce, con mayor o menor detalle, el funcionamiento interno de los componentes del sistema. Esta técnica permite tanto el acceso a los datos procesados a lo largo de la simulación, como la monitorización de las estructuras modeladas. Así, es posible generar distintos tipos de estadísticas y medidas acerca de cualquier parámetro de rendimiento del sistema simulado. La reproducción del funcionamiento de los componentes del sistema permite la ejecución completa de cualquier programa de prueba, sin necesidad de producir y almacenar enormes archivos de trazas.

Las simulaciones basadas en ejecución requieren el modelado tanto de la microarquitectura como del repertorio de instrucciones simulado. La precisión de las estadísticas obtenidas, así como la monitorización de estructuras, depende del nivel de detalle con el que se modela la microarquitectura. Cuanto mayor sea el nivel de detalle, mayor potencial se tendrá para monitorizar y obtener estadísticas, a costa de una menor velocidad de ejecución en las simulaciones.

Dado que el objetivo del simulador presentado es obtener estadísticas sobre el rendimiento de la microarquitectura asíncrona sobre cualquier programa de prueba, la elección más acertada es implementar un simulador basado en ejecución. El detalle con el que se modela cada uno de los componentes de la microarquitectura se examina en el siguiente apartado.

### 4.1.3. Simulación arquitectónica

Es muy importante decidir cuál es el nivel de abstracción más adecuado para simular un sistema de cómputo, entendiendo nivel de abstracción como la proximidad del simulador a las características de implementación del circuito simulado. Así, cuanto más bajo sea el nivel de abstracción, mayor detalle se tendrá acerca de la implementación del circuito y de su temporización; mientras que cuanto más alto sea el nivel de abstracción, menor detalle se utilizará. El nivel de abstracción se denomina también granularidad del simulador.

Como se ha explicado anteriormente, el nivel de abstracción del simulador es uno de los criterios que se siguen para diferenciar los simuladores de sistemas computadores en dos categorías: simuladores funcionales y simuladores arquitectónicos. Para ver claramente las diferencias entre ambos tipos de simuladores, es necesario analizar sus características por separado.

Las características principales de los simuladores funcionales, o simuladores de alto nivel de abstracción son las siguientes:

- Granularidad gruesa: la especificación detallada de los componentes *hardware* se sustituye por una descripción de la funcionalidad del circuito.
- Alta velocidad de las simulaciones: la simplificación en el detalle descriptivo lleva asociada, habitualmente, una mayor velocidad de simulación.
- Verificación: las simulaciones funcionales son muy adecuadas para realizar comprobaciones acerca del correcto funcionamiento del circuito debido, principalmente, a su elevada velocidad de ejecución.
- Bajo nivel de detalle: no es posible obtener información precisa acerca de retardos, latencias o cualquier otro parámetro que dependa de la implementación concreta de un circuito, puesto que estos detalles de implementación se obvian en el simulador.

Los puntos fuertes de este tipo de simulaciones son, por tanto, la alta velocidad de las simulaciones y la posibilidad de omitir detalles de implementación que, o bien no se conocen, o bien no se han decidido en el momento de verificar la

funcionalidad de un diseño. Sin embargo, estas ventajas, derivadas de las características anteriormente enumeradas, son insuficientes para cubrir los objetivos del simulador bajo estudio. Las medidas descritas en el capítulo introductorio, así como futuras propuestas relacionadas con ellas, no se podrían obtener con un simulador de este tipo.

Por otro lado, los simuladores arquitectónicos, o simuladores de bajo nivel de abstracción, presentan las siguientes características principales:

- Granularidad fina: se modela con gran detalle el comportamiento de los componentes *hardware* del sistema simulado.
- Múltiples medidas y estadísticas: gracias a la detallada descripción del sistema, los simuladores arquitectónicos suelen ser capaces de emular la ejecución real del *hardware* modelado. Así, estos simuladores permiten obtener gran cantidad de información acerca del cómputo ocurrido en cada uno de los componentes modelados.
- Velocidad de ejecución de las simulaciones: la principal desventaja de los simuladores arquitectónicos consiste en que cuanto menor es el nivel de abstracción, más lenta resulta la ejecución de las simulaciones. De hecho, las simulaciones de muy bajo nivel de abstracción (nivel de puertas lógicas) no se suelen aplicar a circuitos complejos, puesto que su ejecución tiende a dilatarse demasiado en el tiempo.

Un simulador arquitectónico permite, por tanto, obtener medidas de rendimiento sobre el sistema modelado. Así, tanto las medidas ya definidas como cualquier otra futura propuesta se podrán obtener siempre que el simulador modele el sistema con suficiente detalle.

El simulador propuesto modelará un procesador asíncrono superescalar de propósito general donde el nivel de abstracción escogido será lo suficientemente bajo como para conseguir simulaciones arquitectónicas, de manera que la ejecución de los componentes del procesador se pueda modelar fielmente. Como se explicará más adelante, el nivel descriptivo llegará hasta el punto de modelar individualmente, además de la memoria o el predictor de saltos, el banco de registros, las unidades funcionales del procesador y cada una del resto de etapas en la ejecución de las instrucciones.

## 4.2. Modelado de un procesador superescalar asíncrono

A lo largo de este apartado se detallan las cuestiones principales relacionadas con el modelado y simulación de un procesador superescalar asíncrono. En primer lugar se ponen de manifiesto las particularidades que diferencian el modelado de un procesador asíncrono frente a un procesador síncrono, haciendo hincapié en la necesidad de separar su descripción funcional de la verificación de su temporización. Seguidamente se describe el modelado de la funcionalidad, donde se consideran los componentes más relevantes de la microarquitectura. Por último, se presenta el modelo de la temporización, utilizado para caracterizar la evolución en el tiempo de los cómputos que realiza el procesador.

### 4.2.1. Modelado de procesadores asíncronos vs. síncronos

La simulación detallada de un procesador síncrono, lejos de ser sencilla, se puede simplificar en cierta medida aprovechando la propia naturaleza síncrona del procesador. La utilización de una señal de reloj global en el circuito implica que *todos los componentes del sistema síncrono computan una vez por cada ciclo de reloj.* Por tanto, siguiendo esta idea, el simulador puede implementar un bucle principal donde la funcionalidad de cada una de las etapas del procesador se ejecute de manera secuencial una vez por cada iteración.

Este mecanismo es el empleado por la mayoría de los simuladores de sistemas síncronos. Sirva como ejemplo *sim-outorder*, parte del conocido SimpleScalar [ALE02]. En la Figura 4.1 se muestra un extracto del bucle principal de este simulador, donde se aprecia cómo se invoca secuencialmente a las funciones que modelan cada una de las etapas (*commit, writeback, issue, dispatch, fetch*), justo en el orden inverso al recorrido por las instrucciones. Si el simulador siguiese el mismo orden que las instrucciones la información que se envía de una etapa a la siguiente debería esperar al fin del bucle para su actualización, porque podría sobreescribir los datos anteriores. Sin embargo, con el enfoque de ejecución de funciones a la inversa se evita ese problema. Como se aprecia en la figura,

```
/* main simulator loop, NOTE: the pipe stages are traverse
   in reverse order to eliminate this/next state synchronization
   and relaxation problems */
for (;;) {
   ...
   /* commit entries from RUU/LSQ to architected register file */
   ruu_commit();
   ...
   /* service result completions, also readies dependent ops */
   ruu_writeback();
   ...
   /* decode and dispatch new operations */
   ruu_dispatch();
   ...
   /* issue operations ready to execute from a previous cycle */
   /* <== drains ready queue <-- ready ops commence execution */
   ruu_issue();
   ...
   /* call instruction fetch unit if it is not blocked */
   if (!ruu_fetch_issue_delay) ruu_fetch();
   else ruu_fetch_issue_delay--;
   ...
   /* go to next cycle */
   sim_cycle++;
   ...
}  // for
```

Figura 4.1: Extracto del bucle principal en *sim-outorder*, simulador arquitectónico de *SimpleScalar*. Las etapas modeladas (marcadas en rojo) se simulan en orden inverso a la ejecución real.

al final del bucle se pasa a simular el siguiente ciclo incrementando el contador denominado *sim_ cycle*.

Un procesador asíncrono, por el contrario, no se rige por una señal global de reloj, sino que transmite datos entre sus componentes utilizando un protocolo de comunicación determinado. De este modo, cada componente del procesador trabaja de manera independiente. Al no existir un control centralizado, no se puede garantizar que cada componente del sistema ejecute tantas veces como lo hacen el resto. Más aún, el tiempo de cómputo de cada etapa puede ser distinto en función de los datos de entrada que reciba y, por lo tanto, distinto a la latencia de otros componentes del procesador. Una consecuencia directa de este hecho es la posibilidad de que una etapa finalice el cómputo de un dato e intente transmitir el resultado a otra etapa que no esté preparada para recibirlo. Si el receptor estuviera procesando una entrada previa, el dato a transmitir debería esperar, lo que significa que la etapa emisora tendría que retrasar la entrega de su salida.

Por tanto, los instantes de tiempo en que cada componente del procesador inicia o finaliza un cómputo son desconocidos *a priori* y dependen tanto de los datos de entrada como del estado actual del sistema. Esta indeterminación en la temporización sumada a la variación en el tiempo de cómputo de cada etapa, hace que la simulación de un procesador asíncrono no se pueda tratar de una manera análoga a la de su equivalente síncrono.

En resumen, la simulación de un procesador asíncrono debe modelar correctamente la ejecución concurrente de los componentes del sistema, así como la comunicación que ocurre entre ellos. Por tanto, el simulador debe considerar los siguientes aspectos principales:

- La temporización de cada elemento del procesador debe ser correctamente gestionada por el simulador. Los instantes de inicio y fin de cómputo deben ser determinados correctamente. En otras palabras, el simulador debe calcular dinámicamente las intervenciones de cada uno de los componentes del procesador a lo largo de la ejecución de las simulaciones teniendo en cuenta el estado del sistema.

- La comunicación entre los elementos del sistema debe ser correctamente

modelada teniendo en consideración la ejecución concurrente de los componentes del circuito.

- La simulación debe respetar la *ley de causa-efecto*, de manera que una operación sólo empezará cuando haya terminado la operación que la provocó.

Debido a su especial naturaleza, los simuladores de procesadores síncronos no tienen en cuenta ninguno de estos aspectos, por lo que se hace necesaria la creación de un nuevo simulador que cumpla con estos requisitos. En ese sentido, se propone un modelo para la simulación de un procesador superescalar asíncrono de 64 bits donde se desacopla el modelado de la funcionalidad del modelado de la temporización. En otras palabras, se distingue por un lado la ejecución de instrucciones en la microarquitectura y, por otro lado, se controlan los instantes de tiempo en que cada uno de sus componentes actúa. Esta separación permite definir distintos tipos de temporización para una misma microarquitectura, lo cual amplía significativamente las aplicaciones del simulador. Más adelante, en el apartado 4.4, se profundiza en los detalles de implementación.

### 4.2.2. Modelado de la microarquitectura

El simulador modela un procesador superescalar de propósito general. Se trata de una microarquitectura de 64 bits con ejecución especulativa fuera de orden y predicción de saltos. Las instrucciones procesadas evolucionan a lo largo de cinco etapas:

- ***Fetch*** : lectura de instrucciones de la memoria.
- ***Issue*** : decodificación y comprobación de dependencias.
- ***Exec*** : ejecución en unidades funcionales.
- ***Write-back*** : difusión de resultados y resolución de dependencias.
- ***Commit*** : finalización de instrucciones.

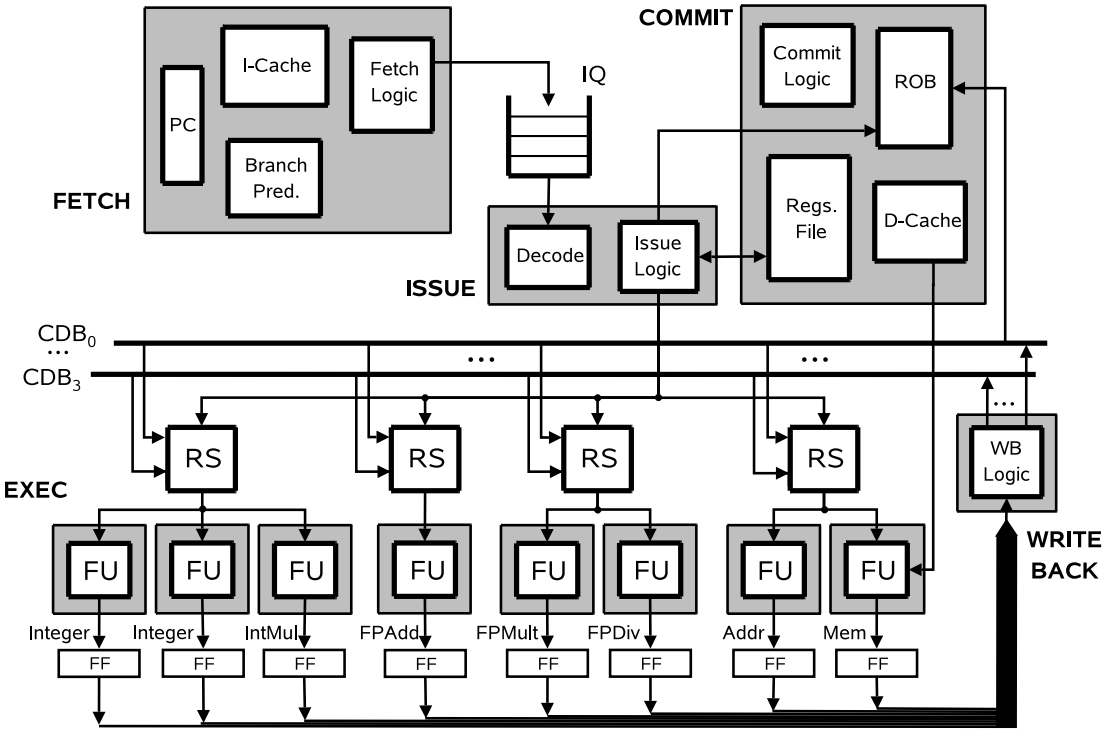


Figura 4.2: Esquema de la microarquitectura bajo estudio. Se trata de un microprocesador superescalar con ejecución fuera de orden y predicción de saltos. Las etapas por las que avanzan las instrucciones son cinco: *Fetch*, *Issue*, *Exec*, *Write-back* y *Commit*. Las zonas en gris representan los dominios independientes que se modelan en las simulaciones bajo temporización asíncrona.

En la Figura 4.2 se muestra el esquema general de la microarquitectura, indicando con etiquetas las zonas correspondientes a cada una de las etapas del procesador y utilizando flechas para señalar el flujo de datos entre cada una de ellas. En una arquitectura de tipo *pipeline* lineal, cada etapa en la ejecución de las instrucciones envía datos, principalmente, a la etapa siguiente. Como se puede observar, esta arquitectura no sigue el patrón de un *pipeline* lineal, puesto que existen comunicaciones entre etapas no consecutivas como la que se produce entre las etapas *Issue* y *Commit*, y también realimentaciones entre etapas como la que realiza *Write-back* hacia *Exec*.

Este planteamiento complica el modelado de la funcionalidad del procesador, más aún si se tiene en cuenta tanto su naturaleza superescalar como su funcionamiento bajo una temporización asíncrona. Como se verá más adelante, para modelar el funcionamiento asíncrono del procesador se divide la microarquitectura en dominios independientes, representados como rectángulos de color gris en la Figura 4.2. Las comunicaciones entre cada dominio se podrán producir en cualquier momento, dependiendo de los tiempos de cómputo de cada dato y del estado global del sistema, lo que provocará que la evolución temporal en la ejecución de un programa no sea predecible hasta el mismo momento de su simulación. Todos los detalles relacionados con el modelado se estudian a partir del Apartado 4.2.3. En los siguientes apartados se describe la estructura y funcionalidad de cada una de las etapas de la microarquitectura, comentando además las interacciones entre ellas.

### Etapa *Fetch*

La etapa de *Fetch* se encarga de la lectura de instrucciones de la memoria y de la predicción de saltos. En la Figura 4.3 se muestran las estructuras afectadas por la lógica de esta etapa (*Fetch Logic*): contador de programa ($PC$), memoria de instrucciones (*I-Cache*) y predictor de saltos (*Branch Pred.*). El trabajo realizado por la etapa de *Fetch* se puede dividir en las siguientes tareas:

- Lectura de $f$ instrucciones de *I-Cache*. El número $f$, denominado también *anchura de la etapa*, es uno de los parámetros que se pueden configurar en el simulador. La lectura de memoria se produce en función del valor del $PC$.

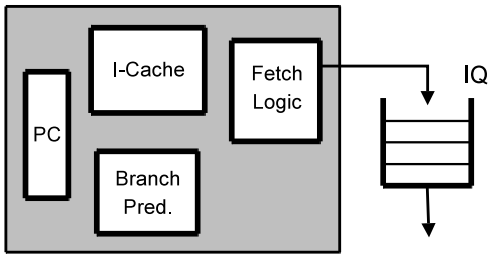


Figura 4.3: Detalle esquemático de la etapa *Fetch*. Las estructuras involucradas en esta etapa son: *Fetch Logic*, lógica de esta etapa; *PC*, contador de programa; *I-Cache*, memoria de instrucciones; *Branch Pred.*, predictor de saltos. Al terminar la etapa de *Fetch*, las instrucciones se escriben en la cola de instrucciones, *IQ*.

- *Predecodificación*. Sobre cada una de las instrucciones leídas se realiza una predecodificación con objeto de detectar alguna instrucción de salto. En caso de encontrar un salto condicional, la predecodificación se detiene, pasando a obtener la predicción sobre la próxima instrucción a ejecutar.

- Predicción de saltos. Como se verá más adelante, el predictor de saltos es totalmente configurable en el simulador. Es posible elegir predicción estática (saltos siempre tomados o no), predictor bimodal, predictor de dos niveles de historia, o bien predictor híbrido. La predicción decidirá cuál es la siguiente instrucción, escribiendo su dirección de memoria en el *PC*.

- Escritura en la cola de instrucciones ($IQ$). Las instrucciones que se procesan en esta etapa se escriben ordenadas en una cola FIFO denominada $IQ$. Como se ha explicado anteriormente, en caso de encontrar una instrucción de salto condicional, la predecodificación de instrucciones se detiene. Esto significa que, en esa situación, la última instrucción que se escribirá en $IQ$ será, precisamente, la instrucción de salto.

Según la funcionalidad descrita, el ancho ideal de la etapa, denotado como $f$ en el primero de los puntos anteriores, se puede reducir el tiempo de ejecución debido a la presencia de instrucciones de salto condicional. Dependiendo de la implementación real de la etapa, este hecho puede dar lugar a variaciones en su tiempo de cómputo. En el caso de una implementación donde las operaciones sobre

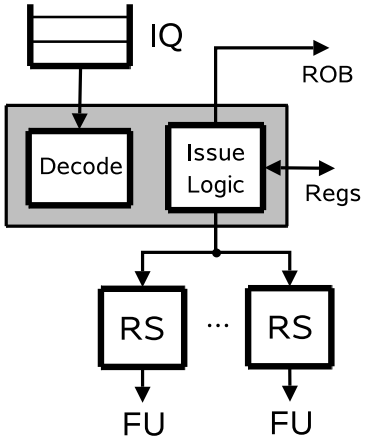


Figura 4.4: Detalle esquemático de la etapa *Issue*. Las estructuras involucradas en esta etapa son: *Decode*, decodificador de instrucciones e *Issue Logic*, lógica de lanzamiento encargada de leer operandos del banco de registros y aplicar el algoritmo de tratamiento de dependencias. Las instrucciones llegan a la etapa de *Fetch* ordenadas desde la *IQ*. El destino de las instrucciones son las estaciones de reserva (*RSs*), donde esperan la resolución de dependencias para poder ejecutar en las unidades funcionales (*FUs*).

las instrucciones leídas de memoria sea muy paralela, la variación en el tiempo de cómputo no será muy alta, puesto que sólo dependerá de los datos procesados. Por el contrario, una implementación de *Fetch* donde las instrucciones se procesan en serie reduciría el *hardware* empleado, presentando una mayor variabilidad en la latencia que dependerá, fundamentalmente, del momento en que se procese una instrucción de salto. En cualquier caso, el simulador es capaz de modelar todas estas especificaciones puesto que las funciones de distribución que modelan la latencia pueden representar cualquiera de estos tipos de comportamiento.

### Etapa *Issue*

Pese a que la Figura 4.4 muestra un esquema aparentemente simple para esta etapa, realmente su trabajo es complejo. Concretamente, la etapa *Issue* realiza las operaciones correspondientes a la decodificación de instrucciones, lectura de operandos del banco de registros y renombramiento de registros necesario para la ejecución especulativa.

En la microarquitectura bajo estudio, la especulación de instrucciones se modela a través de un algoritmo de planificación dinámica con *buffer* de reordenamiento (*reorder buffer*, *ROB*). Este algoritmo utiliza la técnica de *instruction shelving* ("estanterías de instrucciones"), descrita en [Sim97], para el control de dependencias entre instrucciones.

La técnica de *shelving* evita los bloqueos en el lanzamiento debidos a las dependencias entre instrucciones. Para lograr este objetivo, es necesario disponer de estructuras de almacenamiento asociadas a cada una de las unidades funcionales (*functional units*, *FUs*) del procesador. Estas estructuras, denominadas estaciones de reserva (*reservation station*, *RS*) ó *shelving buffers*, alojan las instrucciones que esperan a ser ejecutadas en las *FUs*.

Siguiendo este esquema, la lógica de *Issue* lee las instrucciones ordenadas de la *IQ*, las decodifica y procede a la lectura de sus operandos del banco de registros. A continuación, cada instrucción se almacena en algún hueco libre de la *RS* que le corresponda, según la *FU* en la que se deba ejecutar. A partir de ese instante, las instrucciones se ejecutan desordenadas, por lo que se necesita recordar su orden original. Para ello, las instrucciones se escriben en el *ROB* siguiendo el orden en que fueron leídas de la *IQ*. La escritura en el *ROB* se realiza de manera simultánea a la escritura en la *RS*. Todas estas operaciones se llevan a cabo en la etapa de *Issue* para cada instrucción siempre y cuando exista algún hueco libre en el *ROB* y en la *RS* que corresponda a la instrucción que encabeza *IQ*.

La gestión de dependencias a través del renombramiento de registros se realiza mezclando la técnica de *shelving* con una variante del algoritmo de Tomasulo [HP07, Tom67]. En esta implementación se utilizan etiquetas para identificar a cada una de las instrucciones que llegan a las *RS*. Concretamente, a cada instrucción se le asigna como etiqueta el índice de la *RS* donde se almacenó. De este modo, la existencia de dependencias de datos entre instrucciones se indica con la presencia de etiquetas en lugar de valores. Por ejemplo, si el destino de una instrucción es uno de los registros, se anota en ese registro la etiqueta de la instrucción que producirá el resultado, en lugar de su valor. Si al decodificar una instrucción posterior cuya fuente es el mismo registro se obtiene una etiqueta en lugar de un valor, entonces existe una dependencia de datos. Una dependencia como ésta provocará que la instrucción que la sufre detenga momentáneamente

su ejecución en la *RS*. Sin embargo, *Issue* podrá seguir decodificando y lanzando otras instrucciones. Una vez que la instrucción origen de la dependencia termina, su resultado se propaga a todas las *RS* y registros de modo que aquellas instrucciones que esperaban por ella (las que contienen su etiqueta en algún operando) obtienen el dato que resuelve su dependencia. En consecuencia, la decodificación y el lanzamiento de instrucciones son independientes de la comprobación de dependencias de datos. De hecho, al implementar el algoritmo de *shelving*, el lanzamiento de instrucciones sólo se detiene cuando no existe espacio libre en las *RS* [Sim97], o bien cuando se ha alcanzado la anchura de la etapa.

Al igual que ocurre en la etapa *Fetch*, el parámetro de anchura para *Issue* determina un comportamiento ideal, limitando el número máximo de instrucciones que se permite lanzar. De nuevo, la implementación real de la etapa determinará las variaciones en el tiempo de cómputo, que serían mayores en una implementación donde la decodificación y lanzamiento de las instrucciones se realiza en serie, mientras que en una implementación paralela el tiempo de cómputo sería más constante. En cualquier caso, el modelado que se realiza en el simulador soporta todo tipo de comportamientos y variaciones en la latencia de la etapa.

En este punto se hace necesario realizar una aclaración importante acerca de la expresión "lanzamiento de instrucciones" utilizada a lo largo del apartado. En los manuales técnicos escritos en inglés, el lanzamiento de instrucciones se puede encontrar expresado bien como *instruction issue* o bien como *instruction dispatch*. En la técnica de *shelving*, el lanzamiento de instrucciones corresponde a la etapa *Issue*, fase donde se produce la decodificación de instrucciones y su colocación en los huecos libres de las *RS*. Por otro lado, el *dispatch* corresponde al inicio de la etapa de ejecución, donde las instrucciones libres de dependencias de datos comienzan a utilizar las unidades funcionales correspondientes [Sim97].

#### Etapa *Exec*

En esta etapa se implementa la fase de ejecución de instrucciones del procesador. Los componentes de esta etapa son, principalmente, las unidades funcionales del procesador (*functional units*, *FUs*) y la lógica de *dispatch*, o lógica de selección y lanzamiento de instrucciones libres de dependencias. El repertorio de instruc-

ciones elegido es el del Alpha 21264 [Cor99], implementado a través de los siete tipos de *FUs* siguientes:

- *Integer* : operaciones de suma y resta de enteros. En la arquitectura se incluyen dos instancias de este tipo de *FU*.
- *IntMul* : multiplicación de enteros.
- *FPAdd* : suma y resta de números en punto flotante.
- *FPMult* : multiplicación de números en punto flotante.
- *FPDiv* : operaciones de división y raíz cuadrada de números en punto flotante.
- *Addr* : cálculo de direcciones para las operaciones sobre la memoria de datos.
- *Mem* : operación de lectura de la memoria de datos (*load*).

La Figura 4.5 muestra el esquema de la etapa *Exec* de manera genérica, donde se puede apreciar el flujo de los datos desde las *RS*. En primer lugar, la lógica de *dispatch* de la etapa selecciona una instrucción para ejecutar de entre aquellas que están libres de dependencias en la *RS*. A continuación, la instrucción seleccionada se envía a la *FU* para su ejecución. Una vez lanzada la instrucción y trascurrido el tiempo de cómputo correspondiente, el resultado obtenido se escribe en el registro (*FF*[1]) que se encuentra a la salida de cada una de las *FUs*. Aunque varias *FUs* pueden compartir la misma *RS*, cada *FU* tiene su propio *FF* a la salida (ver Figura 4.2).

Como se explicó anteriormente, las instrucciones que han superado la etapa *Issue* esperan en las *RSs* a que todos sus operandos queden libres de dependencias. De esta manera se produce la ejecución fuera de orden. En este sentido, cabe destacar que la etapa *Exec* no dispone de un parámetro de anchura, sino que se lanza como máximo una instrucción por *FU* de entre aquellas instrucciones que estén libres de dependencias. Así, en el caso más optimista se ejecutarían simultáneamente tantas instrucciones como *FUs* distintas convivan en la microarquitectura.

[1]Los registros que se colocan a la salida de las *FUs* se denotan con *FF*, de *flip-flop*. No se utiliza directamente el término registro para evitar confusiones con los componentes del banco de registros.

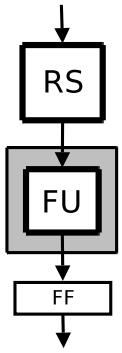


Figura 4.5: Detalle esquemático de la etapa *Exec*. La lógica de *dispatch* de la etapa selecciona una de las instrucciones libres de dependencias en la *RS*. Los operandos de la instrucción se ejecutan en la *FU* y, transcurrido el tiempo de cómputo correspondiente, el resultado se escribe en el registro (*FF*) conectado a la salida.

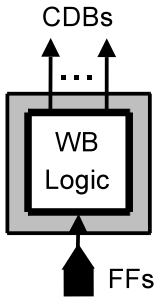


Figura 4.6: Detalle esquemático de la etapa *Write-back*. La lógica de esta fase lee los datos almacenados en los *FF* y los difunde a las *RS* y al *ROB* utilizando el *CDB*. De este modo se resuelven las dependencias de datos de aquellas instrucciones que esperan el valor que se difunde.

### Etapa *Write-back*

Una vez finalizado el cómputo en la *FU* y almacenado el resultado en el *FF*, es necesario difundir el valor obtenido para resolver las posibles dependencias de datos. La difusión de resultados a las *RSs* y su escritura en el *ROB* la realiza la etapa *Write-back*. La Figura 4.6 muestra un esquema muy simplificado para la etapa. En el esquema se puede apreciar que los valores obtenidos de los *FF* se difunden a través del bus común de datos (*common data bus*, *CDB*) que conecta con las *RSs* y con el *ROB*.

La anchura de *Write-back* es determinante puesto que fija el número de valores

que se difunden simultáneamente por el *CDB* permitiendo que las instrucciones que esperan puedan resolver sus dependencias. Sin embargo, ocurre que para que en la microarquitectura se realice la difusión de varios resultados en paralelo, es necesario replicar el *CDB* y la lógica asociada. En la Figura 4.2 se muestra cómo el *CDB* se conecta a todas las *RS* y al *ROB*. De hecho, se indican cuatro instancias de *CDB* ($CDB_0$ a $CDB_3$) para denotar que la anchura por defecto de *Write-back* en las simulaciones es de cuatro instrucciones.

### Etapa *Commit*

La etapa *Commit* realiza las tareas de finalización de instrucciones y resolución de la predicción de saltos. En esta fase se produce la reordenación de las instrucciones que se han ejecutado fuera de orden. Este proceso no implica ningún algoritmo complejo, puesto que en el *ROB* se almacenan las instrucciones que aún no han finalizado, ordenadas según se leyeron en la fase de *Issue*.

La especulación de instrucciones es posible porque hasta llegar a esta etapa no se realiza ninguna escritura en los datos almacenados en estructuras como la memoria de datos (*D-Cache*) o el banco de registros (*Regs File*). Por este motivo se han incluido estas estructuras dentro de la etapa *Commit*, como se muestra en la Figura 4.7.

De nuevo, la anchura de la etapa corresponde al número de instrucciones que se procesan cada vez que *Commit* ejecuta. Sin embargo, hay dos formas de tratar a las instrucciones que llegan a *Commit*, dependiendo de si son instrucciones de salto condicional o no.

Las instrucciones de salto condicional sufren un proceso algo más complejo que el resto. Al encontrar una instrucción de salto, *Commit* comprueba si la predicción realizada en la etapa *Fetch* fue correcta o no. En caso afirmativo, se elimina la instrucción del *ROB* y se pasa a la siguiente. En caso de que la predicción sea incorrecta, las instrucciones posteriores al salto deben ser eliminadas. Esta acción, denotada como *flush* ó vaciado de la microarquitectura, se compone de las siguientes acciones:

- Eliminación de datos e instrucciones en las siguientes estructuras: *ROB*, *RSs*, *IQ* y *FFs*.

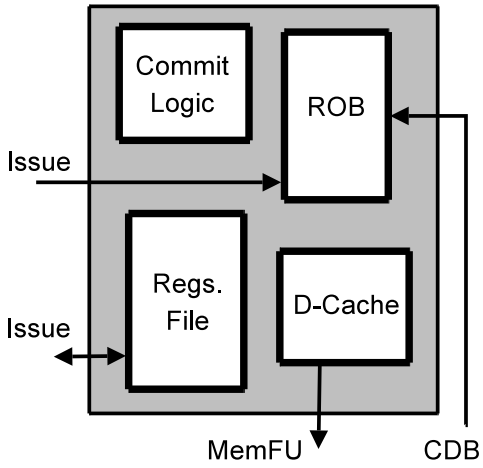


Figura 4.7: Detalle esquemático de la etapa *Commit*. Las estructuras involucradas en esta etapa son: *Commit Logic*, lógica de esta etapa; *ROB*, *reorder buffer*; *Regs File*, banco de registros; *D-Cache*, memoria de datos. La etapa *Commit* finaliza las instrucciones ordenadamente según se encuentran almacenadas en el *ROB*. La resolución de los saltos condicionales se produce en esta etapa.

- Cancelación de la ejecución del resto de etapas ó *reseteo*.
- Restauración de *PC* utilizando la información que la instrucción de salto procesada guarda en el *ROB*.
- Reinicio de *Fetch* con el nuevo *PC*.

Para aquellas instrucciones que no son saltos condicionales, *Commit* las finaliza según su funcionalidad. En caso de que se trate de instrucciones de escritura en registros o en memoria (*store*), su resultado se escribe en la estructura correspondiente. En cualquier caso, la instrucción se elimina del *ROB*.

Al igual que ocurre en las etapas *Fetch* e *Issue*, la anchura de *Commit* determina el comportamiento ideal, estableciendo el máximo número de instrucciones a finalizar en cada ejecución de la etapa. Sin embargo, existen dos motivos para que *Commit* no cumpla con la anchura definida. En primer motivo es la presencia de instrucciones de salto cuya predicción sea incorrecta. El otro motivo está directamente relacionado con la semántica del programa: *Commit* sólo finaliza aquellas instrucciones que hayan completado las fases *Exec* y *Write-back*. Por

tanto, la ejecución de *Commit* se debe detener si encuentra, siguiendo el orden marcado por el *ROB*, una instrucción que aún no ha terminado su ejecución. En ambos casos, la variación en el tiempo de cómputo de la etapa depende de la implementación real escogida, encontrando mayor variación en implementaciones donde las instrucciones se finalizan en serie, y menor variación en implementaciones más paralelas. Como se mostrará en los siguientes apartados, el modelo propuesto para la temporización de la microarquitectura es capaz de representar todas estas variaciones de manera individual para cada una de las etapas y unidades funcionales del procesador.

### 4.2.3. Modelado de la temporización

En el apartado anterior se ha expuesto una descripción puramente arquitectónica del procesador bajo estudio. En otras palabras, la implementación o simulación de esta microarquitectura podría corresponder tanto a un procesador síncrono como a un procesador asíncrono. De hecho, el modelado y simulación de un sistema asíncrono requiere una correspondencia clara con su posible implementación real. Sin embargo, para conseguir esta correspondencia no es necesario llegar a implementar el propio sistema asíncrono, pero sí resulta imprescindible especificar claramente las restricciones de temporización bajo las que funciona el circuito.

El objetivo del simulador que aquí se define es modelar fielmente la microarquitectura descrita en el apartado anterior funcionando bajo un modelo de temporización asíncrono. Sin embargo, como se explica en el Apéndice A, existen distintos tipos de circuitos asíncronos, clasificados según su modelo de retardos. Cada uno de esos tipos de circuito requiere, bien codificaciones especiales para las señales, bien protocolos de comunicación, bien combinaciones de ambos. A partir de estas necesidades y con objeto de conseguir un modelo que permita caracterizar la mayor cantidad posible de comportamientos para circuitos sin señal global de reloj, se han estudiado dos tipos de circuitos asíncronos concretos:

- Circuitos con retardos acotados
- Circuitos insensibles a retardos

Los circuitos asíncronos con retardos acotados se corresponden con la categoría que engloba a todos los circuitos diseñados bajo el modelo de retardos acotados. En estos circuitos se dispone de información acerca del retardo máximo asociado tanto a componentes como a conexiones, aunque siempre en ausencia de señal global de reloj. Pese a que estos circuitos se dividen en varias subcategorías, se ha estudiado un modelo general para circuitos con retardos acotados, puesto que las subcategorías no dependen del propio modelo de retardos sino del modo en que se introducen datos en el circuito (ver Apéndice A).

Por otro lado, los circuitos asíncronos insensibles a retardos son una subcategoría dentro de los circuitos construidos bajo el modelo de retardos no acotados (ver Apéndice A). Se trata de circuitos donde se asume que los retardos de componentes y conexiones no se conoce, por lo que se separa estrictamente su funcionalidad de su temporización. Este tipo de circuitos representa el comportamiento más puramente asíncrono, mostrando gran robustez y presentando variabilidad en sus tiempos de cómputo.

En resumen, en esta tesis se propone un modelo genérico de temporización que permite la simulación de un procesador superescalar asíncrono bajo cualquiera de las siguientes temporizaciones:

- Síncrona
- Globalmente asíncrona, localmente síncrona (GALS)
- Asíncrona con retardos acotados
- Asíncrona insensible a retardos

A continuación se explica el modelo genérico de temporización, describiendo con más detalle cada una de sus posibles particularizaciones para dar cabida a la simulación de distintas temporizaciones.

### Módulos genéricos y dominios de sincronización

En este trabajo se propone un modelo que describe de manera genérica tanto los componentes como las interacciones que ocurren en un sistema de cómputo. Este

modelo permite la simulación de un sistema utilizando distintas temporizaciones, tanto síncronas como asíncronas. Para abordar este problema se manejarán dos conceptos relacionados cuyas definiciones se proponen a continuación.

**Definición** ***(Dominio de sincronización):*** conjunto de lógica cuyas señales de salida se capturan en registros ó *flip-flops* que obedecen a una misma señal de control.

El concepto de dominio de sincronización es ampliamente conocido en el ámbito de los circuitos asíncronos. Sin embargo, en esta propuesta se concreta su definición con objeto de generar un modelo para la latencia del circuito. En ese sentido, un dominio de sincronización se divide en dos partes cuyo tiempo de cómputo es distinto e independiente entre sí. Estas dos partes, modeladas individualmente en el simulador, son las siguientes:

- *Lógica de cómputo*: conjunto de lógica encargada de implementar la funcionalidad del dominio de sincronización.
- *Lógica de temporización*: conjunto de lógica encargada de determinar el instante en que se capturan los datos en la entrada del módulo. Por extensión, esta lógica implementa el mecanismo de envío y recepción de datos desde otros dominios en el caso de modelar una temporización asíncrona, indicando los instantes de captura de datos.

La Figura 4.8 muestra el esquema de un dominio de sincronización *i*, comunicado con un dominio adyacente *i+1* y separados ambos por una línea punteada. Se puede apreciar en la figura que la lógica de temporización es quien decide el instante de captura de nuevos datos de entrada en el registro.

**Definición** (***Módulo genérico***): conjunto de elementos independientes que computan de manera concurrente y transmiten sus resultados utilizando un determinado mecanismo de comunicación. La descripción de un módulo genérico es independiente de la temporización concreta que se implemente en el módulo.

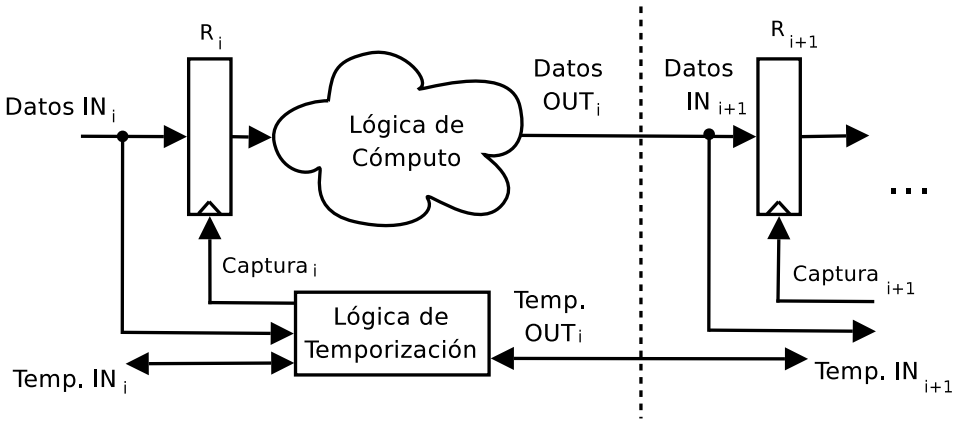


Figura 4.8: Esquema de un dominio de sincronización $i$ que se comunica con un dominio adyacente *i+1*. La línea punteada marca la frontera entre ambos. El dominio $i$ se compone de un registro ($R_i$) que captura los datos de entrada, la lógica de cómputo y la lógica dedicada a la temporización y comunicación de datos entre dominios.

La definición de módulo genérico extiende la definición de dominio de sincronización añadiendo la jerarquía que existe en el diseño de un circuito. En ese sentido, el módulo genérico más básico contiene un sólo dominio de sincronización mientras que, como se muestra en la Figura 4.9, un módulo se puede componer de otros módulos más pequeños.

Aplicando el concepto de módulo al diseño del procesador, se ha dividido la microarquitectura en distintos módulos, indicados en la Figura 4.2 como rectángulos de fondo gris. Cada módulo forma un dominio de sincronización distinto, de manera que el simulador modelará de manera independiente la temporización de cada uno de ellos. Por tanto, se necesitará manejar alguna expresión que caracterice la latencia de los datos que se procesan.

### Latencia de datos en módulos genéricos

Considerando que los módulos en que se divide el procesador se componen de un sólo dominio de sincronización cada uno, en este subapartado se define una expresión genérica para su temporización.

A la hora de determinar la latencia ($L$) para un dato que llega al registro de la entrada del módulo ($R_i$ en la Figura 4.8), se consideran varios valores de tem-

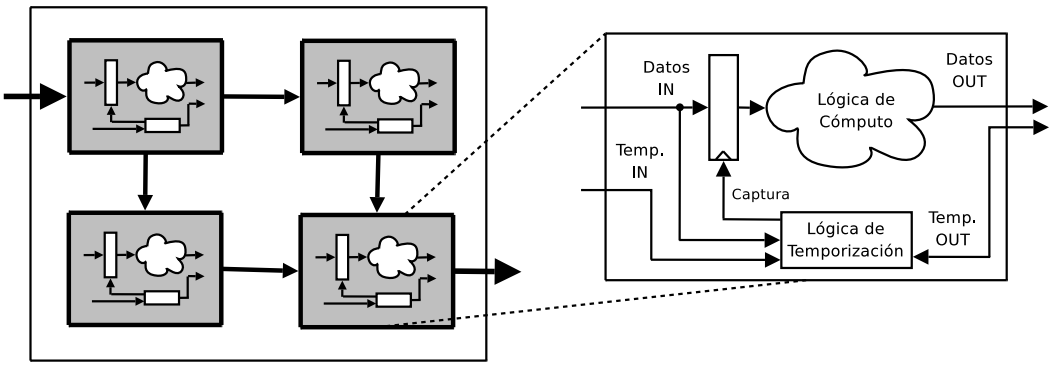


Figura 4.9: Vista esquemática de un circuito (izquierda) compuesto por cuatro módulos independientes, indicados con fondo gris. Las flechas indican transmisión de información entre ellos. El propio circuito en conjunto se puede describir como un módulo en sí, mientras que, como muestra el detalle de la derecha, los módulos más pequeños pueden contener un único dominio de sincronización.

porización. El primero de ellos es el *tiempo de cómputo* ($t_c$) ó retardo empleado por la lógica de cómputo en generar un resultado. Tras este intervalo de tiempo puede ocurrir que el receptor no se encuentre disponible para recibir el resultado, probablemente por estar operando sobre un dato anterior. El periodo de tiempo que transcurre hasta que el receptor se encuentra disponible se denomina tiempo de *espera por receptor* ($e_r$). Este tiempo no se conoce *a priori* puesto que depende del estado del sistema, pero siempre se debe respetar antes de iniciar la comunicación entre emisor y receptor. Por último, el protocolo de comunicación entre el emisor y el receptor del dato también consumirá un determinado tiempo, denominado *tiempo de protocolo* ($t_p$). Por lo tanto, los tres datos mencionados se deben sumar a la hora de obtener la latencia del módulo genérico:

$$L = t_c + e_r + t_p \tag{4.1}$$

Mientras que el valor de $t_c$ puede ser variable, dependiente de los datos de entrada, el retardo $t_p$ depende de cuestiones como la codificación de las señales o el tipo de protocolo empleados. Sin embargo, $e_r$ depende del estado del procesador en ese instante de tiempo. De hecho, sólo es posible determinar los instantes de tiempo en que una etapa o unidad funcional del procesador está ocupada si se aplica una simulación arquitectónica que tenga en cuenta las estructuras simuladas. Más

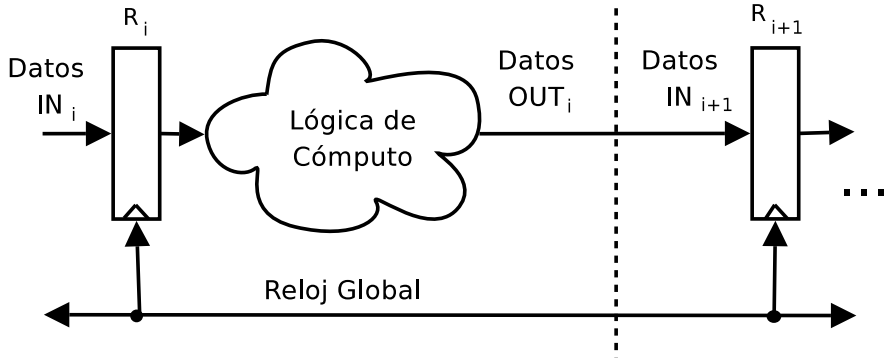


Figura 4.10: Esquema de un módulo síncrono $i$ que se comunica con un módulo adyacente *i+1*. El reloj global alimenta los registros de todos los módulos del sistema, separados por una línea punteada en la figura.

concretamente, en este tipo de simulación $e_r$ no es un valor numérico calculado, sino un intervalo de tiempo que finaliza cuando el simulador detecta el instante en que el receptor del dato está disponible.

El módulo que se acaba de definir es genérico en el sentido de que el modelado de su latencia se puede adaptar con el objeto de representar el comportamiento de otros tipos de circuito. A continuación se muestran en detalle las diferentes opciones y propuestas de modelado.

### Módulos síncronos

A partir de un módulo sencillo con un sólo dominio de sincronización, la adaptación a temporización síncrona del módulo genérico es sencilla. Como se aprecia en la Figura 4.10, desaparece el bloque para la lógica de temporización puesto que la señal de control global es el reloj.

El modelado de la latencia para este caso también se simplifica. En primer lugar, el tiempo de cómputo ($t_c$) se transforma en tiempo de ciclo ($t_{ciclo}$), correspondiente al peor caso de latencia para el módulo más lento del sistema en el peor caso de temperatura, voltaje y posibles variaciones en el proceso de fabricación, contando además con los tiempos de *setup* y *hold* del registro y los márgenes de seguridad habituales en este tipo de diseños. En segundo lugar, tanto el retardo debido al protocolo de comunicación como el referido a la espera por el receptor

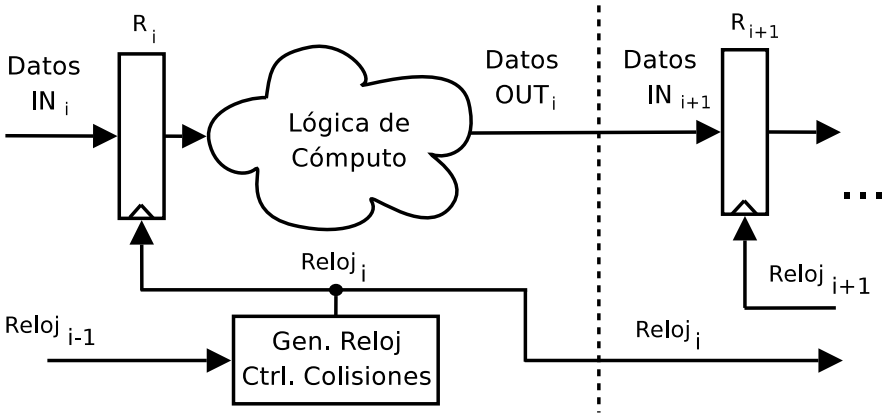


Figura 4.11: Esquema de un módulo GALS (globalmente asíncrono, localmente síncrono). El bloque generador de la señal de reloj local del módulo $i$ ($Reloj_i$) debe controlar las posibles colisiones con el reloj del módulo anterior ($Reloj_{i-1}$) para evitar metaestabilidades. Esta misma comprobación se produce en el siguiente módulo, separado por una línea punteada en la figura.

desaparecen, al carecer de sentido en este modelo. Así, transformando la Fórmula (4.1), la latencia de este módulo corresponde a la siguiente expresión:

$$L_{Synch} = t_{ciclo} \tag{4.2}$$

### Módulos GALS

Los circuitos Globalmente Asíncronos, Localmente Síncronos disponen, en cada uno de sus módulos, de su propia generación de la señal de reloj que determina el momento en que los datos se capturan a la entrada de cada módulo o dominio de reloj. En la Figura 4.11 se muestra un diagrama con la adaptación del módulo genérico a módulo GALS donde se incluye el bloque generador de reloj. En todo dominio GALS se debe evitar la generación de un pulso en el reloj local en aquellos instantes próximos al flanco de reloj del módulo emisor de los datos. En caso de coincidir ambos pulsos se produciría una *colisión* entre relojes que podría dar lugar a una metaestabilidad en las entradas.

El modelado de la latencia para este caso se debe particularizar para cada módulo puesto que cada uno de ellos puede emplear un tiempo de ciclo distinto. De nuevo,

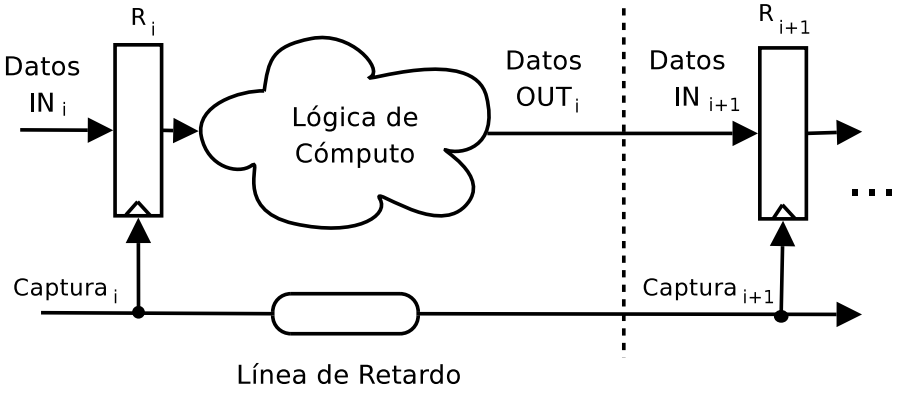


Figura 4.12: Esquema de un módulo asíncrono con retardos acotados. La línea de retardo retrasa la llegada de la señal de captura durante un intervalo de tiempo mayor al caso peor de cómputo del dominio de sincronización. En la figura se separan dos módulos consecutivos, *i* e *i+1*, utilizando una línea punteada.

el retardo debido al protocolo de comunicación carece de sentido en este modelo. Por otro lado, es el propio simulador quien vigila las posibles colisiones entre módulos aplicando el retraso pertinente, correspondiente a un ciclo del reloj local de cada módulo. Este retraso ó espera debida al receptor se denota como $e_{rGALS}$, y es particular para cada módulo. En consecuencia, la latencia de un módulo $i$ se determinaría por la sustitución en la expresión (4.1):

$$L_{GALS_i} = t_{ciclo_i} + e_{rGALS_i} \tag{4.3}$$

### Módulos asíncronos con retardos acotados

El tiempo de cómputo para circuitos asíncronos bajo el modelo de retardos acotados está, como su propio nombre indica, delimitado. En otras palabras, se garantiza que todo cómputo que se lleve a cabo en el módulo empleará un tiempo menor ó igual a una cantidad conocida (cota). En ese sentido, este tipo de circuitos son muy similares a los síncronos, con la salvedad de que el retardo de cada módulo no tiene por qué ser igual que los demás, mientras que en los circuitos síncronos el periodo de reloj es común. La Figura 4.12 muestra un esquema de módulo bajo el modelo de retardos acotados.

La latencia de este tipo de módulos, al igual que en el modelo GALS, se debe particularizar por módulo. Sin embargo, a diferencia de los GALS, no hay control de colisiones ni metaestabilidades por lo que las líneas de retardo suelen aplicar márgenes de seguridad importantes. De nuevo, simplificando la expresión (4.1), el valor de latencia de un módulo $i$ se obtendría a partir de la siguiente fórmula, donde $t_{delay_i}$ representa el tiempo que la línea de retardo retrasa la señal de captura:

$$L_{RA_i} = t_{delay_i} \tag{4.4}$$

### Módulos asíncronos insensibles a retardos

Como se explica en el Apéndice A, el modelo de retardos no acotados engloba múltiples alternativas en la implementación de circuitos asíncronos bajo la misma suposición: los retardos de conexiones y componentes del circuito no se conocen. De entre las alternativas posibles dentro de este modelo, se han elegido los circuitos insensibles a retardos (IR) como opción principal a modelar en el simulador. La elección de este tipo de circuitos se debe, principalmente, a que se trata del modelo más robusto y completo. Además, cualquier otra variante dentro del modelo de retardos acotados consiste en relajar alguna de las condiciones de los circuitos IR (ver Apéndice A), por lo que la implementación en el simulador de este modelo permite cubrir todas las combinaciones anteriormente mencionadas.

En la Figura 4.13 se muestra el esquema del módulo IR que se modela en el simulador. Los circuitos IR requieren de un bloque de detección de fin de cómputo (CD). Por tanto, como se aprecia en la figura, cada línea de datos tiene conectada el correspondiente bloque CD. Además es necesario establecer comunicación entre módulos emisor y receptor, por lo que se requiere que el bloque para la lógica de temporización se transforme en el control de la comunicación y *reset* de la etapa.

En cuanto al cálculo de la latencia de este tipo de módulos, se requiere una particularización por módulo debido a varios motivos. En primer lugar, el tiempo de cómputo puede ser variable, dependiendo de los datos de entrada. En los circuitos IR sí se aprovecha esta variación en la latencia puesto que no se necesita respetar el retardo del caso peor para pasar al siguiente dato. Para indicar que el tiempo

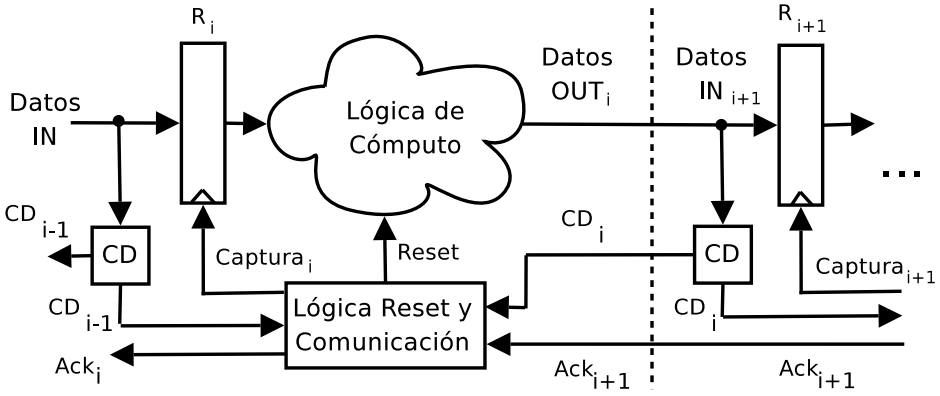


Figura 4.13: Esquema de un módulo asíncrono insensible a retardos, denotado como *i*, y parte del siguiente módulo, *i+1*, separados por una línea de discontinua. La lógica de reset y comunicación determina la captura del dato de entrada en función de la señal de reconocimiento del módulo receptor ($Ack_{i+1}$) y de la señal de detección de fin de cómputo ($CD_i$).

de cómputo es variable se denotará como $t_{cv_i}$. En segundo lugar, este modelo sí debe tener en cuenta la disponibilidad del módulo receptor de los resultados producidos, de modo que ese retardo también se tiene en cuenta. Por último, el retardo atribuido al protocolo de comunicación también es particular para cada módulo ya que distintos módulos podrían utilizar distintos protocolos. Se denota este retardo como $t_{p_i}$. En consecuencia, la expresión (4.1) se transforma en la siguiente fórmula:

$$L_{IR_i} = t_{cv_i} + e_r + t_{p_i} \tag{4.5}$$

El modelo para circuitos IR se ha precisado aún más incluyendo el protocolo de comunicación en la caracterización. Concretamente se ha optado por caracterizar los retardos de los protocolos *handshake* de cuatro fases y *handshake* de dos fases bajo codificación de doble raíl. A continuación se explican los detalles de esta caracterización.

### Protocolo *handshake* de cuatro fases en doble raíl.

El protocolo *handshake* de cuatro fases en doble raíl utiliza dos conexiones o señales por cada uno de los bits de información a transmitir. Una de las cuatro

combinaciones de valores binarios, habitualmente $(0, 0)$, se denomina espaciador (*spacer*), o sincronismo. Este valor no se considera un dato válido en el protocolo, sino que se utiliza como separador entre dos valores consecutivos en la transmisión. De esta manera, no es necesario utilizar una señal de solicitud (*request*) en el protocolo, sino que está implícita en el momento que las líneas pasan de contener el valor de sincronismo a transmitir un dato válido.

Este protocolo se puede describir como un diálogo entre dos módulos, emisor y receptor, que alternan distintas fases. Así, la comunicación de los datos se define, utilizando una notación sobre procesos concurrentes[2], con las siguientes expresiones:

$$Emisor: \quad [\neg ack]; \overbrace{\mathcal{X}\uparrow}^{t_c}; [ack]; \overbrace{\mathcal{X}\downarrow}^{t_{sync}} \tag{4.6}$$

$$Receptor: \quad \overbrace{[v(\mathcal{X})]}^{t_{fv}}; \overbrace{ack\uparrow}^{t_{ack}}; \overbrace{[n(\mathcal{X})]}^{t_{fn}}; \overbrace{ack\downarrow}^{t_{ack}} \tag{4.7}$$

donde $\mathcal{X}\uparrow$ significa la generación de un nuevo dato válido, $\mathcal{X}\downarrow$ corresponde a la generación de un sincronismo, $v(\mathcal{X})$ indica que $\mathcal{X}$ es un dato válido, y $n(\mathcal{X})$ que $\mathcal{X}$ es el sincronismo.

Las llaves que aparecen sobre la descripción anterior se refieren a la temporización del protocolo. Su caracterización se define asociando un retardo diferente para cada una de las cuatro fases:

- Fase 1: el emisor espera la caída en la señal de reconocimiento (*ack*). Esta espera tendrá un tiempo dependiente del receptor, por lo que no se indica en el emisor. Después computa los datos de entrada en un tiempo $t_c$ y espera a que el receptor levante *ack*, durante un tiempo también dependiente del receptor.

- Fase 2: el bloque de detección de fin de cómputo (CD) detecta un dato válido en un tiempo $t_{fv}$ , y provoca que el receptor levante la señal *ack* empleando

[2] *Notación*: el punto y coma (;) separa los pasos que necesariamente se deben ejecutar en serie; los corchetes ([]) implican espera hasta el establecimiento de una señal al valor indicado (0 si aparece el signo ¬, ó 1 en caso contrario); las llaves ({ }) indican espera hasta que la señal cambia su valor; las flechas verticales indican establecimiento (↑), reset (↓) ó cambio de valor (↕) en una señal.

un retardo $t_{ack}$ . A continuación, si el receptor está libre de datos anteriores, captura el nuevo dato en su registro de entrada y comienza el cómputo.

- Fase 3: tras detectar el establecimiento de la señal *ack*, el emisor genera el espaciador empleando un retardo $t_{sync}$ , y espera a que el receptor baje la señal *ack*.

- Fase 4: el bloque CD detecta la llegada de un sincronismo al receptor, de manera que la señal *ack* se baja. Estos dos pasos se realizan en un tiempo $t_{fn}$ , dando lugar al estado de *reset* en el emisor. Al finalizar esta fase, el emisor está disponible para empezar un nuevo cómputo.

En el peor de los casos, las cuatro fases de este protocolo ocurrirían estrictamente en serie, por lo que el retardo del protocolo, extrayendo el tiempo debido al cómputo ($t_c$), quedaría como sigue:

$$t_{4f} = t_{fv} + t_{ack} + t_{sync} + t_{fn} + t_{ack} \tag{4.8}$$

Por tanto, la latencia de un módulo insensible a retardos que utilizase el protocolo handshake de cuatro fases en doble raíl se caracterizaría utilizando la siguiente expresión, resultado de sustituir $t_{p_i}$ en (4.5) por el retardo anterior:

$$L_{IR_i} = t_{cv_i} + e_r + t_{4f} \tag{4.9}$$

**Protocolo *handshake* de dos fases en doble raíl.**

En esta versión de la comunicación se eliminan los pasos finales del protocolo de cuatro fases. En concreto, el protocolo *handshake* de dos fases no necesita un dato especial de sincronismo. En su lugar, el receptor interpreta el dato válido de una comunicación como el dato inicial o sincronismo para la siguiente comunicación. De nuevo, la comunicación entre emisor y receptor se puede describir utilizando notación de alto nivel:

$$Emisor: \quad \{ack\}; \overbrace{\mathcal{X} \uparrow}^{t_c} \tag{4.10}$$

$$Receptor: \quad \overbrace{[v(\mathcal{X})]}^{t_{fv}}; \overbrace{ack \updownarrow}^{t_{ack}} \tag{4.11}$$

donde $\mathcal{X} \uparrow$ significa la generación de un nuevo dato válido, $v(\mathcal{X})$ indica que $\mathcal{X}$ es un dato válido, y $ack \updownarrow$ denota que $ack$ cambia su valor.

En este protocolo la sincronización viene dada por la interpretación de datos válidos no consecutivos a la salida del emisor. En otras palabras, las fases del protocolo se puede describir de la siguiente manera:

- Fase 1: el emisor espera un flanco en la señal $ack$ ($\{ack\}$) y, a continuación, genera un nuevo dato válido. Durante la generación de dicho dato y hasta que éste se encuentre disponible, en la salida del emisor se cumple que $\neg v(\mathcal{X})$. Por tanto, el receptor no encontrará datos válidos en su entrada y esperará.
- Fase 2: el receptor reconoce un dato válido en su entrada empleando un retardo $t_{fv}$ . A continuación almacena el valor en su registro y cambia el valor de la señal $ack$ en un tiempo $t_{ack}$. El receptor sólo provocará ese flanco si antes del dato $\mathcal{X}$ que verifica $v(\mathcal{X})$ se recibió algún valor $\mathcal{Y}$ que verificase que $\neg v(\mathcal{Y})$.

De nuevo, extrayendo de la caracterización el tiempo de cómputo ($t_c$), la latencia del protocolo *handshake* de dos fases vendría dada por la siguiente fórmula:

$$t_{2f} = t_{fv} + t_{ack} \tag{4.12}$$

Y, en consecuencia, la latencia de un módulo insensible a retardos que utilizase el protocolo handshake de dos fases en doble raíl se caracterizaría sustituyendo el valor de $t_{p_i}$ en la expresión (4.5) por el retardo de la fórmula anterior:

$$L_{IR_i} = t_{cv_i} + e_r + t_{2f} \tag{4.13}$$

**Resumen**

A lo largo de los apartados anteriores se han expuesto todos los modelos necesarios para caracterizar fielmente la ejecución del procesador superescalar bajo distintas temporizaciones, empezando por la síncrona y terminando por la asíncrona insensible a retardos. También se ha detallado un modelo genérico para módulos asíncronos y su particularización en cada uno de los distintos tipos de circuito, además de los modelos para los protocolos de comunicación handshake de cuatro y dos fases.

Tanto el modelo asíncrono genérico como los modelos de protocolo se han implementado con éxito en el simulador, de modo que es capaz de modelar todas las combinaciones anteriores gracias a sus parámetros de configuración. Por un lado, el simulador recibe los valores de tiempo de cómputo para cada módulo del procesador. Estos valores pueden ser fijos ($t_c$) o variables ($t_{cv}$), utilizando funciones de distribución en este último caso. Por otro lado, el simulador también recibe como parámetros los retardos de caracterización de los protocolos de dos y cuatro fases: $t_{fv}$ ,$t_{ack}$ ,$t_{sync}$ y $t_{fn}$ . Como se verá en el Capítulo 5, esta capacidad permite modelar distintas implementaciones de estos protocolos variando los valores de cada uno de estos parámetros.

En el siguiente apartado se aborda uno de los aspectos fundamentales, común a todos los modelos asíncronos: la integración de los tiempos de cómputo variables en el simulador.

## 4.3. Latencia variable en el simulador arquitectónico

Como se ha explicado al inicio de este capítulo, un simulador arquitectónico se sitúa en un nivel de abstracción intermedio. Este nivel, suficientemente alejado de los detalles de implementación del sistema simulado, considera particularidades del circuito que los simuladores de alto nivel obvian. Una herramienta que pretenda caracterizar la ejecución de un procesador asíncrono requerirá, al menos, de una caracterización arquitectónica. El motivo fundamental de este requisito

es la necesidad de conocer el instante en que los datos de entrada llegan a cada módulo del procesador, de manera que le sea posible asignar un valor de tiempo de cómputo individual a cada dato procesado.

En el Capítulo 3 se muestra cómo las funciones de distribución de probabilidad (FDPs) permiten la caracterización de la latencia variable en circuitos. Precisamente ésa es la opción elegida en el simulador: la caracterización del tiempo de cómputo variable para cada uno de los módulos del procesador se realiza utilizando FDPs.

Por tanto, la integración de las FDPs queda circunscrita exclusivamente al paso en el que se produce la asignación del tiempo de cómputo para cada dato. En ese sentido, ocurrirá que los retardos que tengan una mayor probabilidad asociada en la función de distribución se asignarán en más ocasiones que aquellos retardos con menor probabilidad. Como resultado, se obtiene una latencia variable según el patrón que marca la FDP.

El problema consiste entonces en aplicar un algoritmo eficiente que sea capaz de seleccionar aleatoriamente un retardo de la función de distribución teniendo en cuenta las probabilidades que ésta asocia a cada retardo. Para solucionar este problema se ha utilizado una variante del *algoritmo de selección proporcional de la ruleta* [DeJ75], proveniente del campo de los algoritmos genéticos, y cuya aplicación en el simulador se describe a continuación.

La idea consiste en asignar a cada uno de los retardos considerados por la función de distribución una *porción* de una ruleta, siendo el tamaño de la porción proporcional a la probabilidad asignada a cada retardo. Del mismo modo que la suma de las probabilidades de la función de distribución siempre se supone normalizada a 1, la suma del tamaño de todas las porciones de la ruleta será igual al *tamaño de la ruleta*, $T$.

Para determinar el valor de $T$ se utiliza, en este simulador, el número de posiciones decimales de los valores de probabilidad de la función de distribución, denotado como $d$. El tamaño de la ruleta será, entonces:

$$T = 10^d \tag{4.14}$$

De esta manera, si las probabilidades tienen dos cifras decimales, $T$ tomará el valor

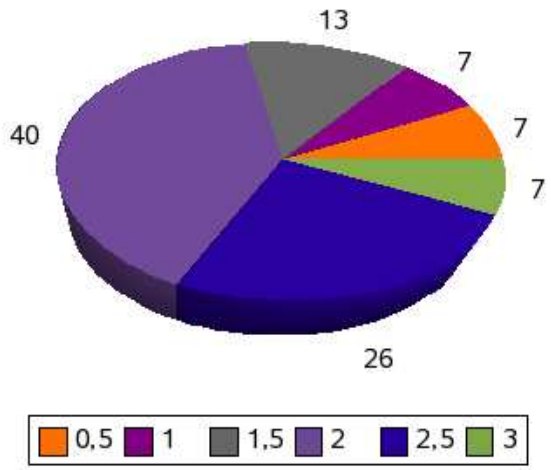


Figura 4.14: Construcción de la ruleta para el algoritmo de selección a partir de una función de distribución sencilla, que se muestra en la Figura 3.2. Los valores de probabilidad tienen dos decimales, por lo que el tamaño de esta ruleta es 100. Cada porción representa a uno de los retardos de la función de distribución, mostrando un tamaño igual al valor de probabilidad del retardo multiplicado por 100. En la parte inferior, la leyenda muestra los retardos, en unidades de tiempo, asignados a cada porción de la ruleta.

100; si las probabilidades tienen tres cifras decimales, $T$ tomará el valor 1000, y así sucesivamente. Esto hace que, incluso tomando valores de probabilidad de cinco decimales, el tamaño de la ruleta siempre sea muy pequeño en comparación con el tamaño de una estructura que almacenase todos los retardos posibles junto a su valor de probabilidad.

El tamaño de cada porción de la ruleta será igual al producto del valor de probabilidad del retardo que representa, multiplicado por $T$. En la Figura 4.14 se muestra un ejemplo de construcción de la ruleta a partir de la función de distribución de la Figura 3.2. En este ejemplo los valores de probabilidad tienen dos decimales, por lo que el tamaño de la ruleta es 100, y cada porción tiene un tamaño igual al valor de probabilidad que representa multiplicado por 100.

Para asignar un retardo a un dato, el simulador generará un número aleatorio entero entre 1 y $T$, que corresponderá a una posición de la ruleta. Esta posición siempre pertenecerá a una porción, asignada a su vez a un retardo. El algoritmo de selección, por tanto, empleará un tiempo de búsqueda acotado y pequeño. Además, gracias a la transformación hecha al construir la ruleta, la generación

de números aleatorios se simplifica, puesto que se realiza sobre valores enteros.

La construcción de la ruleta se realizaría en uno de los pasos iniciales de la ejecución del simulador. Cada ruleta ocupa un espacio contenido, y considera tantos retardos como indique la función de distribución, que son los mismos que se determinó al fijar la granularidad del histograma del que proviene (ver Capítulo 3).

Esta integración, por tanto, resulta eficiente en términos de complejidad computacional del algoritmo de selección utilizado, puesto que es constante. También resulta eficiente en términos de espacio extra necesario para alojar las estructuras de datos utilizadas, dependiente de la precisión decimal de las probabilidades.

## 4.4. Estructura del simulador

Una vez expuestas todas las funcionalidades requeridas para el modelado y simulación arquitectónica de un procesador superescalar asíncrono, en este apartado se tratan las cuestiones relacionadas con la estructura e implementación del propio simulador.

### 4.4.1. Presentación y arquitectura *software*

Al principio de este capítulo se justifica la necesidad de trabajar con simulaciones basadas en ejecución, dado que se pretende obtener medidas de rendimiento de la microarquitectura para cualquier tipo de programa, sin utilizar trazas de ejecución. Por ello, al tratarse de un simulador basado en ejecución, es necesario disponer de varios elementos básicos. Estos elementos son, fundamentalmente, tres: el cargador de código binario, el modelo de memoria de instrucciones y el control de fallos de página. Además se necesita un soporte para llamadas al sistema de manera que las operaciones de entrada y salida que realicen los programas de prueba sobre el disco ó con el terminal se interpreten correctamente en el sistema operativo anfitrión. Todas estas características deben ser suficientemente robustas y estar correctamente verificadas.

En ese sentido, el conocido SimpleScalar [ALE02], ofrece soluciones contrastadas para todas esas cuestiones. Como se aprecia en la Figura 4.15, SimpleScalar se

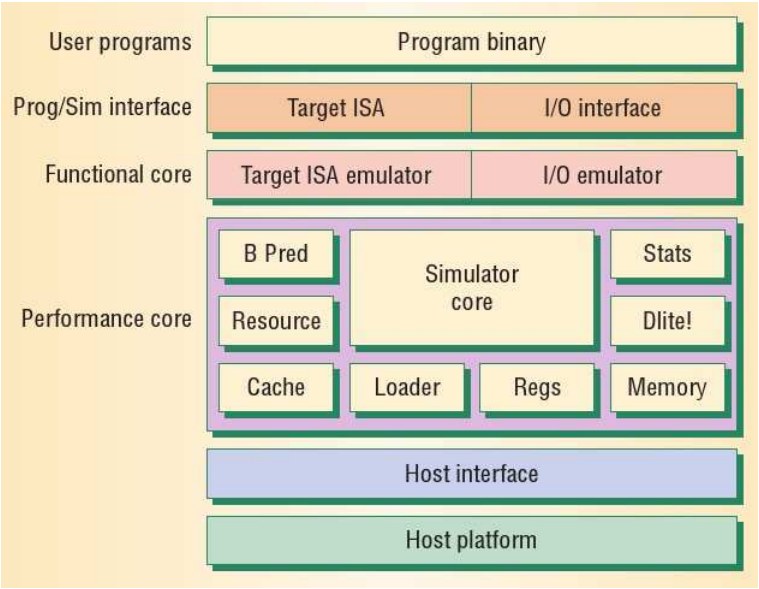


Figura 4.15: Estructura de SimpleScalar, tomada de [ALE02], donde se puede ver cómo el diseño del simulador es muy modular. Esta modularidad permite, entre otras alternativas, su extensión a distintos repertorios de instrucciones cambiando el *Functional core* (núcleo funcional), así como el modelado de distintas microarquitecturas, modificando las partes correspondientes del *Performance core* (núcleo de rendimiento).

ha diseñado en base a una estructura muy modular que lo hace muy extensible y portable. Más concretamente, la parte denotada como *Functional core* (núcleo funcional) se encarga de la simulación de la funcionalidad del repertorio de instrucciones modelado, mientras que el *Performance core* (núcleo de rendimiento) modela las operaciones correspondientes a la microarquitectura como son, por ejemplo, aquellas que afectan al contenido de la memoria y registros. En esta parte también se sitúan la configuración y operaciones sobre el predictor de saltos.

Las implementaciones actuales de SimpleScalar admiten varios conjuntos de instrucciones como binarios de entrada. Los más utilizados son el repertorio de instrucciones de los procesadores Alpha y un repertorio propio llamado PISA. En concreto, debido a la modularidad del código fuente, la extensión de SimpleScalar a otros repertorios de instrucciones se restringe a trabajar sobre un conjunto pequeño de archivos, independientes del resto del simulador salvo por los interfaces que deben respetar.

SimpleScalar también ofrece una infraestructura muy flexible y extensible para la generación de estadísticas relacionadas con la ejecución de simulaciones. El código fuente provee de un conjunto de funciones predefinidas que hacen que la creación y manejo de contadores para establecer medidas sea muy cómodo para el programador.

Por tanto, SimpleScalar es un excelente punto de partida para realizar simulaciones arquitectónicas del procesador descrito en apartados anteriores, así como para tomar estadísticas sobre cualquier programa de prueba. Además, SimpleScalar está codificado en lenguaje C y su código fuente se encuentra disponible en internet de manera gratuita en *http://www.simplescalar.com/* .

Ahora bien, como ya se ha explicado en este capítulo, SimpleScalar se diseñó para simular la ejecución de un procesador superescalar síncrono. Esto significa que el modelado de la ejecución de una microarquitectura asíncrona en SimpleScalar requiere una serie de modificaciones importantes del código fuente. Estas modificaciones se relacionan con los siguientes dos aspectos del simulador:

1. Modelado del paso del tiempo: la ejecución asíncrona de un circuito no se basa en ciclos de reloj, sino en instantes de tiempo. SimpleScalar no permite

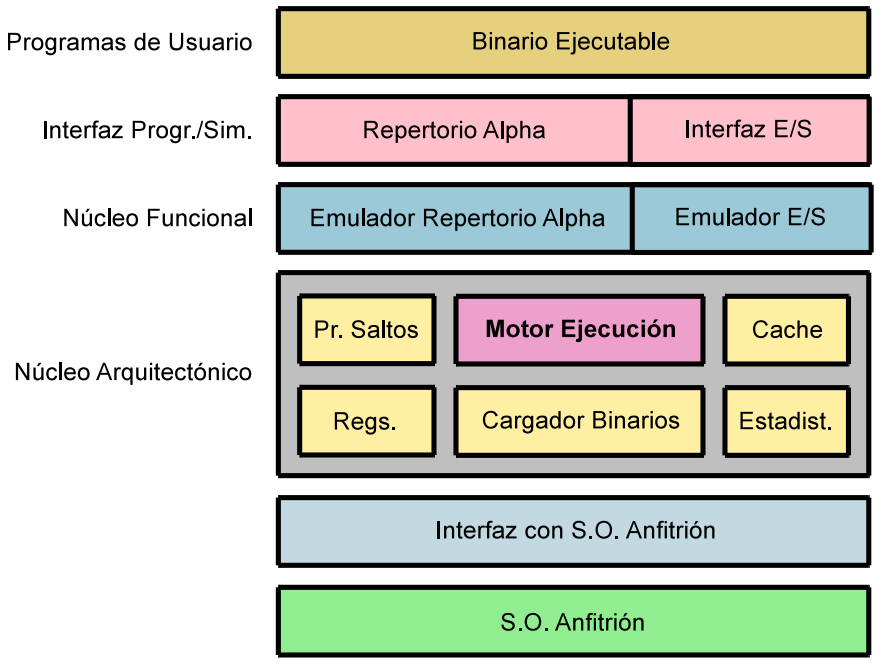


Figura 4.16: Estructura de *Sim-async*. El diseño modular es análogo al de SimpleScalar, aunque presenta las siguientes diferencias fundamentales: (1) El "Motor de Ejecución" se modifica para poder modelar una evolución temporal no ceñida a ciclos de reloj; (2) La arquitectura que se modela en el "Núcleo Arquitectónico" cambia, definiendo ahora distintos dominios asíncronos; (3) En la "Interfaz Programa/Simulador" se fija el repertorio de instrucciones Alpha.

que las etapas del procesador puedan iniciar ó terminar su ejecución en cualquier momento, sino que se ciñe a los flancos de reloj. La alternativa consiste en modelar la temporización del circuito de una manera más lineal.

2. Modelado de una microarquitectura asíncrona: las etapas de las instrucciones y los elementos microarquitectónicos definidos en SimpleScalar presentan relaciones muy complejas entre ellos debido a que se modela un procesador totalmente síncrono. Este diseño dificulta la división del modelo en dominios asíncronos independientes. La solución reside en crear un nuevo modelo de microarquitectura donde se definan dominios asíncronos concretos.

A partir de estas modificaciones sobre SimpleScalar nace ***Sim-async***, el simula-

dor propuesto en esta tesis. La estructura *software* de *Sim-async*, presentada en la Figura 4.16, es análoga a la de SimpleScalar, aunque presenta algunas diferencias resultado de aplicar distintas soluciones a los aspectos anteriormente expuestos. Estas soluciones son las siguientes:

1. Los instantes de tiempo en que se inician o finalizan acciones en la microarquitectura se representan utilizando eventos. Como se explicará más adelante, cada evento dispone de un valor ó marca de tiempo. Los eventos se ordenan según su marca de tiempo y se procesan siguiendo ese orden. La parte del simulador encargada de gestionar los eventos es el "Motor de Ejecución" que se indica en la Figura 4.16.

2. Cada uno de los dominios asíncronos de la microarquitectura que se modela en *Sim-async* (descrita en el apartado 4.2.2) se define en el "Núcleo Funcional" a través de funciones en lenguaje C. Se han creado además nuevas estructuras lógicas para elementos *hardware* modelados como el *reorder buffer* o las estaciones de reserva, y se ha ajustado la implementación del banco de registros a la nueva microarquitectura. Estas modificaciones sobre estructuras afectan fundamentalmente a la parte del "Núcleo Arquitectónico", aunque debido al nivel de detalle del simulador, existen múltiples interacciones entre la parte funcional y la parte arquitectónica (en las lecturas de valores del banco de registros, accesos a estaciones de reserva, *reorder buffer* o memoria, etc.).

3. Por último, se ha adaptado el repertorio de instrucciones Alpha a la nueva microarquitectura. Gracias a la similitud entre la nueva microarquitectura y las especificaciones del procesador Alpha 21264, utilizado como referencia, esta adaptación resultó bastante directa. La consecuencia más importante es la posibilidad de ejecutar código para Alpha sobre la nueva microarquitectura bajo cualquiera de las temporizaciones descritas en el Apartado 4.2.3.

En el resto de este apartado se profundiza aún más en los detalles de implementación. Inicialmente se abordan aspectos relacionados con el modelado del tiempo, para pasar después a explicar detenidamente el funcionamiento del "Motor de Ejecución" del simulador y su interfaz de usuario.

### 4.4.2. Simulación guiada por eventos

La microarquitectura modelada en *Sim-async* está compuesta por distintos módulos que ejecutan concurrentemente y se transmiten datos entre sí utilizando un protocolo de comunicación. La ejecución asíncrona de esta microarquitectura no se rige por ninguna señal de reloj, por lo que los instantes de tiempo en que cada componente del sistema inicia o finaliza un cómputo son desconocidos *a priori*. La indeterminación en la temporización, sumada a la variación en el tiempo de cómputo que se aplica en los modelos asíncronos, hacen que la simulación de este procesador no se pueda reducir a un bucle que procese ciclos completos de ejecución, como en el caso de los sistemas síncronos.

Para resolver esta cuestión, *Sim-async* integra un mecanismo de gestión de eventos que permite modelar cualquier temporización asociada a la ejecución de la microarquitectura. Según este mecanismo, el inicio del cómputo de cada etapa o unidad funcional del procesador tiene asociado un evento particular o *evento de simulación* que, a su vez, puede generar eventos adicionales que modelen la temporización de las comunicaciones de datos. Los eventos se insertan en una cola de eventos pendientes, donde se almacenan ordenados y esperan hasta el momento en que el simulador los procesa. Cada uno de los eventos que se manejan en el simulador posee la siguiente información:

- ***Tiempo de inicio***: instante de tiempo (de la línea temporal simulada) en que el evento se genera e inserta por primera vez en la cola de operaciones pendientes.
- ***Etapa***: fase de la microarquitectura a la que corresponde la operación. Dependiendo de la etapa, los valores pueden ser distintos:
  - *Fetch*: los eventos relacionados con esta etapa se identifican como *G_FETCH*.
  - *Issue*: el modelado de esta etapa se divide en dos fases: inicio de *Issue*, denotado como *G_INIISSUE*, y fin de *Issue*, *G_ENDISSUE*. En la primera fase el simulador comprueba las condiciones que se deben cumplir para que la etapa inicie su ejecución. Estas condiciones son

dos: (1) existen instrucciones en la cola donde escribe *Fetch*, *IQ*; (2) tanto el *ROB* como las *RSs* tienen entradas disponibles. En caso de cumplirse ambas condiciones, se simula el comienzo en la ejecución de la etapa *Issue*. Para ello se obtiene la latencia de la etapa de la correspondiente función de distribución y se marcan como "ocupadas" todas sus estructuras. Sin embargo, no se envía ningún resultado a los receptores de esta etapa hasta que no transcurra esa latencia. El fin de la ejecución de la etapa lo determina el evento *G_ENDISSUE*, que se genera con un tiempo igual a la suma de la latencia de la etapa más el tiempo del evento de inicio, y se inserta ordenadamente en la cola de eventos. Es *G_ENDISSUE* el evento que se encarga de activar el envío de resultados y de marcar la etapa *Issue* como disponible para realizar el siguiente cómputo.

- *Exec*: para esta etapa también se utiliza un modelado dividido en dos eventos. El inicio de la etapa *Exec* supone localizar una entrada de las *RSs* asociadas a la correspondiente *FU* donde los operandos estén disponibles para iniciar el cómputo. En caso de encontrarlo, se inicia la simulación de la ejecución de etapa, que consiste en marcar la correspondiente *FU* como ocupada y obtener su tiempo de cómputo según la función de distribución. Todo este proceso inicial se asocia al evento *G_INIEXEC*. El final de la etapa consiste en enviar los resultados al registro que se sitúa tras la *FU*, liberando los recursos ocupados. Estas acciones se asocian al evento *G_ENDEXEC*. Para especificar cuál es la *FU* afectada, los eventos incluyen un campo llamado *estructura*, que se explica más adelante.

- *Write-back*: la ejecución de esta etapa también se modela a través de dos eventos, *G_INIWB* y *G_ENDWB* en este caso. Se encargan, una vez más, de marcar la frontera temporal donde el *hardware* asociado a esta etapa está ocupado. Solo la fase final, representada por el evento *G_ENDWB* aplicará modificaciones sobre las estructuras en que escribe esta etapa.

- *Commit*: de nuevo, los principales eventos asociados a esta etapa son dos: *G_INICOMMIT* y *G_ENDCOMMIT*. Al igual que en *Write-*

| ***Etapa*** | ***Eventos Asociados*** |
|---|---|
| *Fetch* | *G_FETCH* |
| *Issue* | *G_INIISSUE, G_ENDISSUE* |
| *Exec* | *G_INIEXEC, G_ENDEXEC* |
| *Write-back* | *G_INIWB, G_ENDWB* |
| *Commit* | *G_INICOMMIT, G_ENDCOMMIT G_FLUSH* |

Tabla 4.1: Eventos de simulación asociados a cada una de las etapas del procesador.

*back*, estos eventos se encargan de delimitar el tiempo durante el cual la etapa está ocupada. *G_ENDCOMMIT* marca el momento en que se deben aplicar los cambios causados por las instrucciones finalizadas. La etapa *Commit* también genera un evento adicional, asociado indirectamente a todas las etapas del procesador. Se trata del evento denominado *G_FLUSH*, correspondiente al vaciado (*flush*) de todas las estructuras y etapas del procesador. Este evento se genera cuando *Commit* detecta que la predicción para una instrucción de salto fue incorrecta, caso en que se necesita retomar la ejecución por el camino que no se eligió en la predicción.

- ***Estructura***: campo aplicable a los eventos asociados a la etapa *Exec*, donde es necesario reflejar a cuál de las unidades funcionales se refiere el evento, o bien si se trata de una instrucción de acceso a memoria.
- ***Índice en el ROB***: los eventos no sólo se asocian a la etapa y/o estructura a la que corresponden, sino que además se refieren a una instrucción concreta de las que se ejecutan. Este campo realiza esa asociación, indicando el índice ó entrada del *ROB* que ocupa la instrucción relacionada con el evento.
- ***PC***: el contador de programa (*PC*) también se almacena en los eventos. La utilidad fundamental de este campo es controlar el destino y origen de los saltos en la ejecución de las fases *Fetch* y *Commit*.

Los eventos de simulación, por tanto, acotan los instantes en que cada una de las etapas computan y ayudan a modelar su ejecución. En la Tabla 4.1 se muestra un resumen donde se incluyen los eventos de simulación asociados a cada etapa.

### 4.4.3. Motor de ejecución del simulador

Los eventos que se describen en el apartado anterior recopilan toda la información que el simulador necesita para modelar la temporización del procesador. Sin embargo, es necesario añadir una estructura que se encargue de mantener el orden que indica la marca de tiempo de cada evento. Esa estructura es la *cola de eventos*.

La gestión de los eventos encolados no es trivial puesto que es necesario garantizar la ley de *causa-efecto*, consiguiendo que las operaciones de la microarquitectura se modelen en el momento adecuado. La Figura 4.17 presenta el flujo que sigue el motor de ejecución de *Sim-async* en el tratamiento de los eventos. Como se aprecia en la figura, mientras haya eventos en la cola, el motor continuará procesando el primero de los eventos almacenados, según su marca de tiempo.

Una vez obtenido el evento, las acciones a realizar son distintas según su tipo. Como ya se adelantó en el anterior apartado, existen dos tipos de eventos: aquellos relacionados con el inicio de ejecución de una etapa, denotados como *Init*, y aquellos eventos correspondientes al fin de la ejecución de la etapa, denotados como *End*.

#### Eventos *Init*

Los eventos *Init* sirven para comprobar si la etapa a la que corresponde el evento puede ejecutar. Como se verá en el siguiente apartado, cada etapa necesita que ciertas condiciones ó dependencias se cumplan para poder comenzar el cómputo. Si alguna de esas condiciones no se cumple, la etapa tiene que esperar, aunque esta espera es distinta según el modo de simulación (ver la rama izquierda de la Figura 4.17).

En caso de modelar el procesador en modo síncrono (rama *Sync* en la figura), el tiempo de espera antes de volver a intentar ejecutar corresponde a un ciclo, por lo que se genera un *evento de reintento* cuya marca de tiempo es igual al tiempo actual más un ciclo, y cuyo contenido es similar al del evento *Init*. De esta manera se modela el comportamiento de un procesador síncrono, donde la operación detenida no se intentaría hasta el ciclo siguiente.

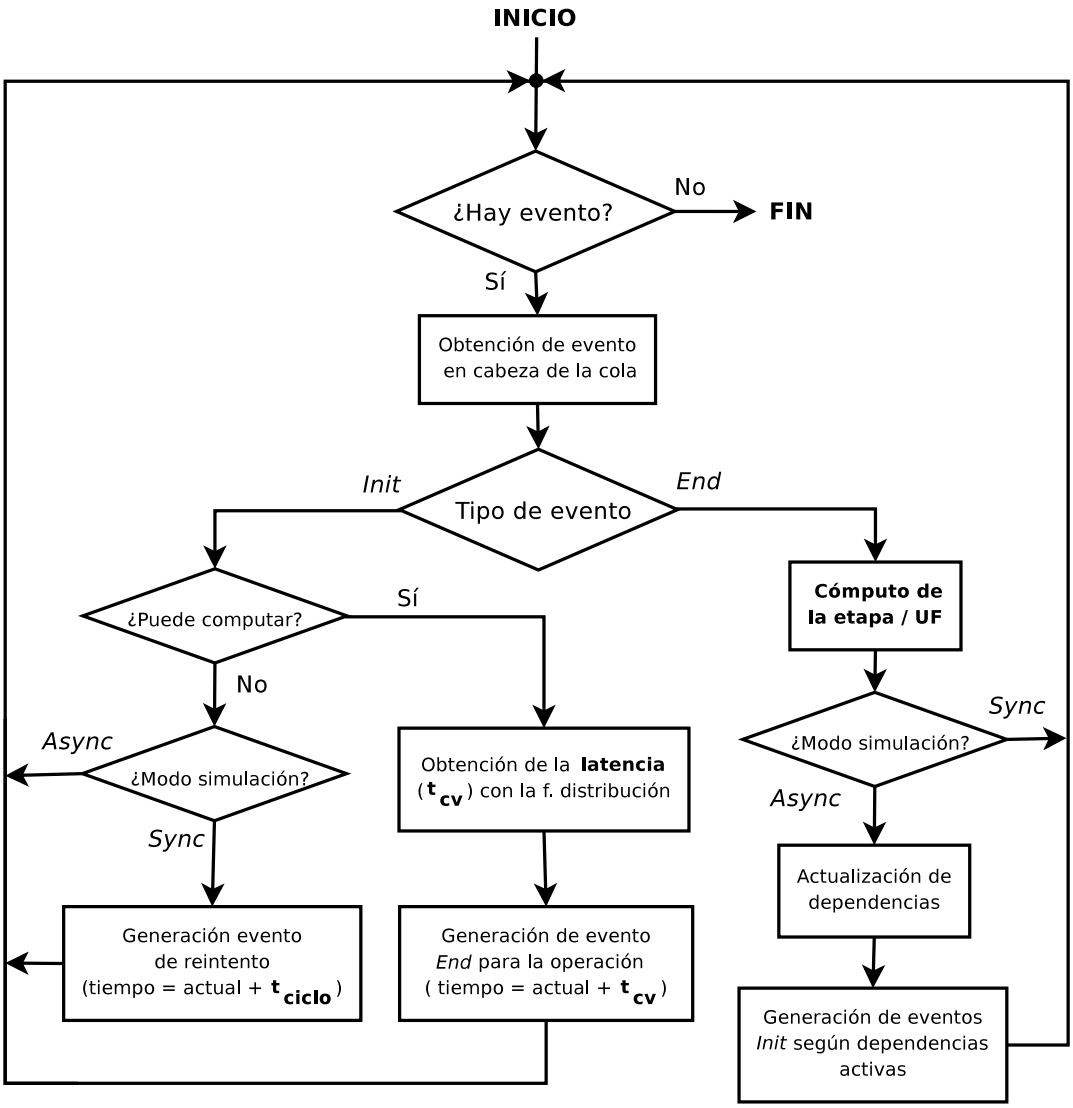


Figura 4.17: Diagrama de flujo del motor de ejecución de *Sim-async* en el proceso de eventos.

Por el contrario, cuando se modela el procesador bajo temporización asíncrona (rama *Async* en la figura), no se tiene certeza alguna acerca del instante de tiempo en que la etapa quedará libre para computar, por lo que el tiempo de espera es indeterminado. Esta espera corresponde precisamente al valor de tiempo denotado como $e_r$ (espera por el receptor) en el apartado 4.2.3, valor que se desconoce *a priori* puesto que depende del estado actual de la microarquitectura y de la evolución del resto de instrucciones, cuyos tiempos de cómputo son variables en cada etapa. En una primera aproximación, el modelado de esta situación se podría resolver volviendo a insertar el evento *Init* de la etapa con una marca de tiempo posterior pero muy próxima, de modo que se realizaría un muestreo sobre la condición de ejecución. Si al volver a procesar el evento la etapa está libre, se ejecuta, pero si no lo está, el evento se reinserta de nuevo. Sin embargo, esta estrategia de *fuerza bruta* implica la generación de una gran cantidad de eventos, lo que ralentiza la ejecución del simulador. Para evitar esta circunstancia se ha implementado en *Sim-async* el concepto de *dependencia entre eventos*.

### Dependencias entre eventos

Las dependencias entre eventos evitan la utilización de fuerza bruta en la resolución de conflictos y permiten conseguir una considerable reducción del número de eventos que se gestionan en el motor de ejecución del simulador. Esta estrategia se aproxima a la secuencia real de operaciones que ocurren en un sistema asíncrono, donde una etapa detenida se reactiva tras la resolución de la condición que provocó su detención gracias al protocolo de comunicación entre ellas.

En la Tabla 4.2 se muestran todas las dependencias definidas en *Sim-async* utilizando una sencilla nomenclatura[3]. La descripción de cada dependencia y su relación con cada condición de parada se describe a continuación, agrupando las dependencias por etapas:

- *Fetch*: si no existe hueco disponible en *IQ*, la etapa *Fetch* permanece detenida. El evento *G_FETCH* que activa la simulación de esta etapa se deberá

[3]Cada dependencia se denota como $dXn$, donde $d$ indica dependencia, $X$ es la inicial de la etapa a la que se asocia, y $n$ es el número de dependencia en esa etapa, expresado en hexadecimal.

| ***Etapa*** | ***Id.*** | ***Tipo de Dependencia*** |
|---|---|---|
| *Fetch* | *dF1* | No hay hueco en *IQ* |
| *Issue* | *dI1* | No hay entradas en el *ROB* |
| | *dI2* | No hay hueco en *RS* de acceso a memoria ó cálculo de dirección |
| | *dI3* | No hay hueco en *RS* de suma/resta de punto flotante |
| | *dI4* | No hay hueco en *RS* de multiplicación/división de punto flotante |
| | *dI5* | No hay hueco en *RS* de enteros |
| *Exec* | *dE1* | *FU* de acceso a memoria ocupada |
| | *dE2* | *FUs* de suma/resta de enteros ocupadas |
| | *dE3* | *FU* de cálculo de direcciones ocupada |
| | *dE4* | *FU* de división en punto flotante ocupada |
| | *dE5* | *FU* de multiplicación/división de enteros ocupada |
| | *dE6* | *FU* de suma/resta en punto flotante ocupada |
| | *dE7* | *FU* de multiplicación en *PF* sin espacio disponible |
| | *dE8* | *Store* previo a *Load* |
| | *dE9* | Op. en *R*S de suma/resta pto. en flotante espera operando |
| | *dEA* | Op. en *RS* de multipl./div. pto. en flotante espera operando |
| | *dEB* | Op. en *RS* de memoria ó cálculo de direcciones espera operando |
| | *dEC* | Op. en *RS* de enteros espera operando |
| | *dED* | Registro de salida de *FU* ocupado. |
| *Commit* | *dC1* | Instrucción en la primera entrada de *ROB* no puede finalizar |

Tabla 4.2: Dependencias entre eventos agrupadas por etapa. Cada dependencia se denota como $dXn$, donde $d$ indica dependencia, $X$ es la inicial de la etapa a la que se asocia, y $n$ es el número de dependencia en esa etapa, expresado en hexadecimal. El motor de simulación utiliza las dependencias para optimizar la gestión de eventos modelando más fielmente el funcionamiento de una microarquitectura asíncrona.

insertar cuando haya algún hueco en *IQ*. Esta situación sólo ocurrirá al terminar una ejecución de la etapa *Issue* (*G_ENDISSUE*). Esta dependencia se denota como *dF1*.

- *Issue*: si el *ROB* no dispone de entradas libres para que la instrucción a lanzar en *Issue* se almacene, se produce una dependencia en *Issue*, denotada en el simulador como *dI1*. Esta dependencia se resuelve al finalizar una ejecución de *Commit*, puesto que esta etapa liberará una entrada del *ROB* por cada instrucción cuya ejecución finalice.

  Otra razón por la que se detiene *Issue* es que la *RS* requerida por la instrucción a lanzar no disponga de un hueco libre. Dado que el simulador utiliza cuatro conjuntos de *RSs* distintos, las dependencias que surgen de esta situación son también cuatro. Si la *RS* requerida es para cálculo de direcciones o acceso a memoria la dependencia se llama *dI2*. Si la *RS* corresponde a la suma (ó resta) en punto flotante, la dependencia es *dI3*. En caso de que la *RS* sea la asignada a las operaciones de multiplicación (ó división) en punto flotante, la dependencia es *dI4*. Por último, si la RS requerida se refiere a operaciones sobre enteros, la dependencia es *dI5*.

  Todas estas dependencias se resuelven al final de la ejecución de la correspondiente *FU* en la etapa *Exec*, puesto que se libera la entrada de la *RS* ocupada por la instrucción que se ejecutaba en la *FU*. Por tanto, al finalizar *Exec* se inserta un nuevo evento *G_INIISSUE* según la dependencia resuelta.

- *Exec*: si existen instrucciones listas para comenzar esta etapa pero la *FU* requerida está ocupada, se genera una dependencia que sólo se resolverá al terminar la ejecución de la *FU*. Como se ve en la Tabla 4.2, las dependencias *dE1* a *dE7* representan esta condición para cada una de las siete *FUs* del procesador.

  Las instrucciones de carga de memoria (*Load*), no se pueden ejecutar si existe alguna instrucción de escritura en memoria (*Store*) previa sin resolver, puesto que si ejecutara *Load* antes que *Store* se podría obtener un valor incorrecto (situación correspondiente a una dependencia LDE[4]). Esta

[4]Lectura después de escritura.

condición se controla en el motor de simulación utilizando la dependencia denotada como *dE8*.

Si se cumplen los requisitos anteriores pero alguno de los datos fuente no está disponible, entonces existe una dependencia sobre la *RS* a la que se asocia la *FU*. La instrucción no podrá ejecutar hasta que en la entrada que ocupa en la *RS* no aparezcan todos los valores de sus operandos fuente. Estas dependencias, diferenciadas según la *RS*, toman los nombres *dE9* a *dEC*, como se aprecia en la Tabla 4.2.

Una vez los datos fuente están disponibles y todos las anteriores condiciones se cumplen, la *FU* comienza a ejecutar. Al terminar su cómputo debe almacenar el resultado en el registro conectado a su salida, pero puede que éste aún contenga el resultado del cómputo anterior. En ese caso se genera una dependencia entre la *FU* y el registro, denotada en el simulador como *dED*. Esta dependencia es la única que no se asocia con el inicio de un cómputo, sino con su finalización. El motivo de que esta dependencia se procese al final de la operación es evitar que una *FU* retrase el inicio de su cómputo hasta el momento en que su registro esté vacío. Así se permite que el registro se pueda vaciar mientras la unidad está computando.

- *Write-back*: no tiene dependencias puesto que si un evento *G_INIWB* se inserta en la cola ocurrirá siempre que alguno de los registros contenga valores a difundir por el *CDB*.

- *Commit*: la finalización de las instrucciones ejecutadas se debe realizar en el mismo orden en que se leyeron de la memoria. Puede ocurrir, sin embargo, que la primera de las instrucciones dispuesta para finalizar, es decir, aquella que ocupa la primera posición en el *ROB*, no haya terminado su ejecución. En este caso se produce una dependencia que el simulador utiliza para gestionar la espera al fin de la ejecución. Esta dependencia se denota como *dC1*, y se resolverá en la fase *Write-back* de la instrucción[5].

En este enfoque microarquitectónico, basta con que una única dependencia o condición no se cumpla en una etapa para impedir su ejecución. En consecuencia,

[5]Existe un caso especial para las instrucciones de llamada al sistema (tipo *PALL_CALL_SYS*). Estas instrucciones finalizan tras ejecutar *Issue*, puesto que no tienen funcionalidad relacionada con el procesador, sino con el sistema operativo anfitrión.

*Sim-async* sólo mantiene información de una dependencia activa por etapa. Dado que siempre que se inicia un cómputo en una etapa se comprueban todas sus dependencias asociadas, esta simplificación es correcta. El caso de la etapa *Exec* es especial porque aunque alguna de las *FUs* no pueda ejecutar, es posible que en el resto de *FUs* sí se pueda iniciar el cómputo. Por tanto, el simulador considera de manera independiente las dependencias de cada una de las *FUs* que componen la etapa *Exec*.

### Eventos *End*

Siguiendo el flujo expuesto en la Figura 4.17, si las condiciones para la ejecución de una etapa se cumplen, el siguiente paso es calcular la latencia para ese cómputo. Como se explicó en el Apartado 4.3, la integración de las funciones de distribución y su utilización no supone una carga excesiva en el simulador. Una vez obtenida la latencia, se genera el evento *End* que llevará a cabo las operaciones de escritura asociadas a la ejecución de la etapa. La marca de tiempo correspondiente a este evento tomará como valor la suma del tiempo actual más el tiempo de cómputo obtenido para la etapa. Según esta marca de tiempo, el evento *End* se inserta ordenadamente en la cola de eventos.

Al insertar nuevos eventos en la cola puede ocurrir que varios de ellos tengan la misma marca de tiempo ó, lo que es lo mismo, los eventos se planificaron para ejecutar en el mismo instante. En este caso se decide su ordenación aplicando la misma estrategia que en SimpleScalar: simular las etapas en orden inverso al que siguen las instrucciones. Esta idea, además, obliga a planificar los eventos *End* antes que los *Init*, puesto que aquellos liberan las dependencias necesarias para que las etapas detenidas ejecuten. En la Tabla 4.3, se muestra el criterio de ordenación en la planificación de eventos para el caso de contener la misma marca de tiempo.

Finalmente, en el momento en que el motor de ejecución procesa un evento de tipo *End*, el simulador hace efectivo el cómputo de la etapa a la que se refiera el evento, escribiendo los cambios correspondientes en las estructuras afectadas. Como indica el diagrama de la Figura 4.17, tras ese paso hay dos caminos posibles en función del modo de simulación. Si se está modelando una temporización síncrona (rama *Sync*), se vuelve al inicio, procesando el siguiente evento de la cola. Por

| ***Eventos Ordenados*** | |
|---|---|
| *GFLUSH* | + Precedencia |
| *G_ENDCOMMIT* | |
| *G_ENDWB* | |
| *G_ENDEXEC* | |
| *G_ENDISSUE* | |
| *G_INICOMMIT* | |
| *G_INIWB* | |
| *G_INIEXEC* | |
| *G_INIISSUE* | - Precedencia |
| *G_FETCH* | |

Tabla 4.3: Criterio de ordenación (de arriba hacia abajo) para los eventos de simulación en *Sim-async* en caso de presentar la misma marca de tiempo.

el contrario, las simulaciones de temporizaciones asíncronas (rama *Async*) manejan dependencias entre eventos, por lo que es necesario actualizar las condiciones que la etapa recién ejecutada podría haber modificado, generando después los correspondientes eventos *Init* para las etapas detenidas. Un sencillo ejemplo de esta situación podría ser aquel en el que la etapa *Issue* se detiene porque no hay hueco en la *RS* que necesita ocupar la siguiente instrucción a lanzar. En el momento que una instrucción alojada en esa misma *RS* finaliza la etapa *Write-back*, el hueco que esta instrucción ocupa en la *RS* se libera, por lo que *Issue* podría continuar su ejecución. El simulador, al procesar el evento *End* asociado al fin de *Write-back*, revisa las dependencias relacionadas con la liberación de huecos en *RS* (actualiza dependencias). En este caso, se detectaría una dependencia sobre *Issue* (*dIx*) que se eliminaría, insertando a continuación el correspondiente evento *Init* para *Issue*, de modo que la ejecución de esta etapa se reanudara.

El motor de ejecución de *Sim-async* es, por tanto, la parte fundamental del simulador. El conjunto de eventos que maneja, así como la gestión que realiza sobre ellos, permiten simular el modelo de procesador bajo cualquier temporización, tanto síncrona como asíncrona. Esta virtud se debe, fundamentalmente, a la separación existente entre la gestión de eventos ligada a la temporización, y el modelado de la microarquitectura.

### 4.4.4. Interfaz de usuario

Dado que *Sim-async* se desarrolló partiendo de *SimpleScalar*, su manejo básico es similar al de éste último. Sin embargo, todos los aspectos relacionados con la configuración y modelado de la microarquitectura bajo configuración asíncrona son nuevos, y se presentan por primera vez en este simulador. Para mostrar todas estas características se divide este apartado en dos partes. En primer lugar se expondrán las cuestiones relacionadas con la configuración de *Sim-async*, resaltando la parte relacionada con la temporización. En segundo lugar se detallarán los datos y estadísticas que el simulador produce al finalizar su ejecución.

#### Configuración de *Sim-async*

La configuración del simulador se consigue añadiendo una serie de parámetros en el lanzamiento de las simulaciones. Esta acción, al igual que en SimpleScalar, se puede realizar desde la línea de comandos. El prototipo de comando para lanzar el simulador es el siguiente:

```
sim-async [-config ficheroParametros] [-configxml ficheroXML] binario
```

Los argumentos son opcionales, puesto que en el simulador se establecen valores por defecto para todos los parámetros. Sin embargo, como se indica en el prototipo anterior, existen dos parámetros que permiten agrupar valores de configuración. Estos parámetros son *config* y *configxml*, y se describen a continuación:

##### -config ficheroParametros

Este parámetro, idéntico al utilizado en SimpleScalar, permite agrupar en un fichero de texto (*ficheroParametros*) los parámetros de configuración relativos a:

- Memoria
- Predictor de saltos

- Número máximo de instrucciones a ejecutar en el simulador
- Prioridad del proceso que ejecuta el simulador

Todos estos argumentos se pueden incluir directamente en la llamada al simulador, aunque dado que el fichero de texto admite comentarios, es posible crear configuraciones donde se describe la intención de cada uno de los valores utilizados. La sintaxis y posibles opciones de cada parámetro son las mismas que se encuentran en SimpleScalar.

**-configxml ficheroXML**

La temporización del procesador modelado se especifica a través de este parámetro. Más concretamente, es el contenido del fichero XML el que permite configurar la temporización. Los parámetros a especificar en el fichero son los siguientes:

- Modo de simulación (*mode*). Es necesario especificar el modo de simulación porque el flujo de gestión de eventos es distinto dependiendo del tipo de simulación (ver apartado anterior).
- Protocolo de comunicación por defecto (*protocol*). Esta especificación se aplica en aquellas etapas donde no se indique un protocolo en particular. Es necesario determinar tanto el tipo de protocolo, a elegir entre *handshake* de dos ó de cuatro fases, como los correspondientes valores para los retardos $t_{fv}$ , $t_{ack}$ , $t_{fn}$ y $t_{sync}$ (ver Apartado 4.2.3).
- Configuración de etapas (*stages*). El simulador permite la configuración individual de las etapas del procesador. Cada una de las configuraciones individuales (*stage*) consta de los siguientes elementos:
  - Nombre de la etapa (*name*): indica cuál de las etapas recibe la configuración. Los posibles valores para este parámetro son los siguientes: *fetch*, *issue*, *exec*, *wb* y *commit*.
  - Anchura de la etapa (*width*): número máximo de instrucciones a tratar en la etapa. El significado particular de la anchura depende de la etapa (ver Apartado 4.2.2). El rango de valores no está acotado en el simulador, aunque siempre debe ser mayor que cero.

- Temporización (*delay*): configura el tiempo de cómputo de la etapa. Para indicar latencia invariable se incluye el valor de retardo en el elemento *fixed*. Si la latencia depende de una función de distribución, se debe indicar en el elemento *distrib* la ruta al fichero XML que la contenga.
- Protocolo de comunicación (*protocol*): determina el tipo de protocolo de comunicación a aplicar en la etapa. Este argumento permite utilizar diferentes protocolos de comunicación en distintas etapas. Los valores especificados prevalecen sobre el protocolo por defecto, y su configuración es similar al elemento *protocol* global.
- Sub-etapas (*substages*): permite configurar componentes individuales dentro de una etapa. En la versión actual del simulador se utiliza para configurar las unidades funcionales dentro de la etapa *Exec*. Este elemento tiene los mismos componentes que el elemento *stage*, así como los rangos de valores a utilizar, con la salvedad de que los valores permitidos para su elemento *name* son distintos. En este caso, al tratarse de unidades funcionales dentro de *Exec*, los valores permitidos son: *intUFx*, *intMulSegUF*, *fpAddSegUF*, *fpMulSegUF*, *fpDivUF*, *dirsUF* y *memUF*, correspondientes a las unidades de suma y multiplicación de enteros, suma, multiplicación y división de punto flotante, cálculo de direcciones y acceso a memoria respectivamente.

En la Figura 4.18 se muestra un ejemplo de fichero de configuración XML para la temporización del sistema. En esta configuración se define una temporización asíncrona que utiliza el protocolo de cuatro fases para la comunicación entre etapas. *Fetch*, *Issue*, *Write-back* y *Commit* se definen con tiempo de cómputo estático, aunque diferente entre sí. Por su parte, las unidades funcionales que componen *Exec* se configuran de manera individual (en la figura no se muestran todas), indicando latencia variable para *intUFx* a través de una FDP.

En el Apéndice B se describen en detalle las gramáticas XML que permiten validar las configuraciones del simulador. Estas gramáticas son, por un lado, la que define la estructura de una función de distribución. Por otro lado se encuentra la gramática general que determina la configuración de las simulaciones en cuanto

```
<configuration name="tesis01">
   <mode value="Async" />
   <protocol name="protoGlobal">
      <drail4ph tfv="10" tfn="10" tack="5" tsync="5" />
   </protocol>
   <stages>
      <stage name="fetch">
         <width value="4"/>
         <delay> <fixed value="1000"/> </delay>
      </stage>
      <stage name="issue">
         <width value="4"/>
         <delay> <fixed value="2000"/> </delay>
      </stage>
      <stage name="exec">
         <substage name = "intUFx">
            <delay> <distrib file="mc_distrib_norm.xml"/> </delay>
            <protocol name="fo4_4ph_proto">
               <drail4ph tfv="20" tfn="10" tack="5" tsync="5" />
            </protocol>
         </substage>
         <substage name = "dirsUF" >
            <delay> <fixed value="1000"/> </delay>
         </substage>
         <substage name = "intMulUF" >
            <delay> <fixed value="4000"/> </delay>
         </substage>
         ...
      </stage>
      <stage name="wb">
         <width value="4"/>
         <delay> <fixed value="1000"/> </delay>
      </stage>
      <stage name="commit">
         <width value="4"/>
         <delay> <fixed value="2000"/> </delay>
      </stage>
   </stages>
</configuration>
```

Figura 4.18: Fichero XML de temporización para *Sim-async*. Muestra una configuración asíncrona con protocolo de cuatro fases donde las etapas *Fetch*, *Issue*, *Write-back* y *Commit* tienen tiempo de cómputo estático. Las unidades funcionales de la etapa *Exec* (no se muestran todas) se describen de manera individual, indicando latencia variable para la unidad *intUFx*.

a parámetros globales sobre etapas y protocolos de comunicación se refiere, así como también determina la configuración de cada etapa en particular.

## Datos de salida y estadísticas

Una vez lanzada la simulación, el programa que se ejecuta puede interactuar con el sistema operativo como si el simulador no se encontrase entre ambos. Por ello, es habitual que los programas de prueba muestren mensajes en el terminal, de modo que se pueda determinar si su ejecución es correcta o si presentan algún problema. En este sentido, el soporte de llamadas al sistema de *Sim-async* es similar al de SimpleScalar, por lo que su presencia se hace transparente al binario que se está simulando, permitiendo una interacción con el terminal análoga a la que proporciona SimpleScalar.

Por otro lado, dada su condición de simulador de un procesador asíncrono, *Sim-async* incluye estadísticas y medidas relacionadas con la naturaleza de este tipo de temporización y simulaciones. En resumen, las medidas definidas se agrupan en cinco categorías:

- Medidas sobre cantidad de instrucciones: tienen el prefijo *sim_num_* y cuentan el número de instrucciones que cumplen alguna condición. Por ejemplo, número de instrucciones que ejecuta el simulador (*sim_num_insn*), número de instrucciones que pasan por cada una de las etapas y unidades funcionales (*sim_num_insn_fetch*, *sim_sum_insn_issue*, ...), número de saltos mal predichos (*sim_num_misp_brchs*), etc.

- Medidas sobre tiempos totales: con el prefijo *sim_lat_*, acumulan la cantidad de tiempo total que las instrucciones emplean en alguna operación.

- Medidas sobre tiempos promedio: tienen el sufijo *_avg*, y se calculan dividiendo el número de instrucciones que cumplen cada condición entre el tiempo total empleado en ella.

- Medidas sobre número de ejecuciones: con el sufijo *_times*, indican el número de veces que una etapa o unidad funcional del procesador ha ejecutado. Estas medidas sólo tienen sentido en las simulaciones asíncronas, puesto que

se considera que en las simulaciones síncronas las etapas ejecutan siempre, en cada tic de reloj.

- Otras medidas: normalmente se refieren a instantes de tiempo en que ocurre alguna situación. El ejemplo principal es *ult_commit*.

La Tabla 4.4 muestra las estadísticas que el simulador genera al finalizar cada simulación.

En el próximo capítulo se muestran los resultados de algunos experimentos donde se ejemplifica la utilización de varias de las estadísticas que se presentan en la tabla anteriormente referenciada.

| ***Estadísticas*** | ***Descripción*** |
|---|---|
| *sim_num_insn_commited* | Instrucciones finalizadas |
| *sim_num_insn* | Instrucciones ejecutadas (incluyendo especuladas) |
| *ult_commit* | Instante de fin del último *Commit* |
| *sim_num_comm_branches* | Núm. de saltos finalizados |
| *sim_num_misp_brchs* | Núm. de saltos mal predichos |
| *sim_num_refs* | Núm. de instrucciones *Load* finalizadas |
| *sim_elapsed_time* | Tiempo total de simulación (segs.) |
| *sim_inst_rate* | Velocidad de simulación (insts/seg.) |
| *sim_insn_lat_acu* | Latencia acumuladas de instrucciones finalizadas |
| *sim_insn_lat_avg* | Latencia media de instrucciones finalizadas |
| *sim_fetch_times* | Ejecuciones de etapa *Fetch* -sólo en simulaciones asíncronas- |
| *sim_lat_IQ* | Latencia total utilizando IQ |
| *sim_lat_IQ_avg* | Latencia media de instrucciones en la IQ |
| *sim_issue_times* | Ejecuciones de etapa Issue -sólo en simulaciones asíncronas- |
| *sim_num_insn_issue* | Instrucciones lanzadas en la etapa *Issue* |
| *sim_lat_RS* | Tiempo de utilización de las RS |
| *sim_lat_RS_avg* | Latencia media de instrucciones en las RS |
| *sim_Int_FU_times* | Ejecuciones en *FUs* de suma/resta de enteros |
| *sim_IntMul_FU_times* | Ejecuciones en *FUs* de multiplicación/división de enteros |
| *sim_FPAdd_FU_times* | Ejecuciones en *FU* de suma/resta en pto. flotante |
| *sim_FPMul_FU_times* | Ejecuciones en *FU* de multiplicación en pto. flotante |
| *sim_FPDiv_FU_times* | Ejecuciones en *FU* división en pto. flotante |
| *sim_Addr_FU_times* | Ejecuciones en *FU* de cálculo de direcciones |
| *sim_Mem_FU_times* | Núm. de accesos a memoria de datos |
| *sim_num_insn_exec* | Núm. de instrucciones que ejecutaron la etapa *Exec* |
| *sim_lat_FU* | Tiempo de utilización de las *FUs* |
| *sim_lat_FU_avg* | Latencia media de las instrucciones en *FUs* |
| *sim_wb_times* | Ejecuciones de etapa *Write-back* -sólo en simulaciones asínc.- |
| *sim_num_insn_wb* | Núm. de instrucciones que ejecutaron la etapa *Write-back* |
| *sim_lat_RegFU* | Tiempo de utilización de los registros salida de las *FUs* |
| *sim_lat_RegFU_avg* | Latencia media de utilización de los registros de las *FUs* |
| *sim_lat_waiting_Reg* | Tiempo de espera por registro libre a la salida de *FUs* |
| *sim_lat_waiting_Reg_avg* | Latencia media de espera por registro libre tras *FUs* |
| *sim_commit_times* | Ejecuciones de etapa *Commit* -sólo en simulaciones asínc.- |
| *sim_lat_ROB* | Tiempo de utilización del *ROB* |
| *sim_lat_ROB_avg* | Latencia media de instrucciones en el *ROB* |
| *sim_num_event* | Núm. de eventos procesados |

Tabla 4.4: Estadísticas generadas por *Sim-async* tras ejecutar cada una de las simulaciones.

# Capítulo 5

# Resultados Experimentales

Llegado este punto, y tras completar el proceso de integración de la microarquitectura asíncrona y del motor de gestión de eventos dentro de *Sim-async*, se ha obtenido el siguiente conjunto de funcionalidades:

- El simulador es capaz de modelar la ejecución del procesador objetivo bajo distintos modelos temporales. Estos modelos, enumerados en el Apartado 4.2.3, se simulan aplicando distintas configuraciones en los parámetros de *Sim-async*. Estas configuraciones van desde la totalmente síncrona, pasando por el modelo GALS y, finalmente, configuraciones para cualquier suposición asíncrona desde la que aplica retardos acotados hasta la que considera el modelo insensible a retardos.
- *Sim-async* permite simular configuraciones asíncronas bajo dos protocolos de comunicación distintos, los protocolos *handshake* de dos y cuatro fases.
- Se ha introducido variabilidad en el tiempo de cómputo de los módulos del procesador aplicando a los retardos de cada módulo caracterizaciones basadas en funciones de distribución de probabilidad.

El paso siguiente a la incorporación de todas estas funcionalidades es la validación del simulador, que se aborda en el siguiente apartado. Tras la validación se exponen dos estudios arquitectónicos de utilización de *Sim-async*:

- El primer caso muestra una comparativa de rendimiento del procesador entre dos configuraciones de temporización opuestas: síncrona y asíncrona.

- El segundo estudio ilustra la utilización de *Sim-async* como herramienta para evaluar el rendimiento de nuevas propuestas microarquitectónicas. En esta ocasión se trata de una propuesta *PAM* (*Partially Asynchronous Microprocessor*), microprocesador parcialmente asíncrono [MABB02], que permite aprovechar la variabilidad en las latencias de las caches de datos.

## 5.1. Validación de *Sim-async*

Dado que *Sim-async* se basa en el código fuente de SimpleScalar y, al igual que éste, procesa el repertorio de instrucciones del procesador Alpha 21264 [Cor99], se plantea la validación de *Sim-async* como primer experimento a realizar comparando ambos simuladores. En el siguiente apartado se describe el método de validación elegido, para pasar después a explicar en profundidad el conjunto de experimentos realizados en la validación.

### 5.1.1. Método de validación

La validación del simulador consiste en comparar, para los mismos bancos de pruebas, los resultados de la ejecución de las simulaciones de *Sim-async* con los resultados que produce SimpleScalar. A continuación se explica cómo se realiza esta comparación de resultados, después se enumeran los conjuntos de simulaciones estudiados y, para finalizar este apartado, se describen los bancos de pruebas utilizados en la validación.

#### Comparación de resultados

Los resultados de una simulación se componen tanto de las salidas que produce el binario simulado como de las estadísticas que proporciona el simulador. Ambos conjuntos de datos se utilizan en la validación de *Sim-async*.

Por un lado, las salidas generadas, formadas tanto por mensajes en pantalla como por ficheros de resultados, indican la funcionalidad del binario en ejecución. En consecuencia, si las salidas que se producen en *Sim-async* son iguales a las

que se producen en SimpleScalar, el binario simulado habrá obtenido los mismos resultados, por lo que el modelado de la arquitectura será correcto.

Por otro lado, se debe utilizar algún mecanismo que permita contrastar el hecho anterior, puesto que podría ocurrir que sólo se simulasen las instrucciones que se encargan de generar la salida, sin realizar una ejecución completa del binario. Para corroborar que se produjo una ejecución completa, se utiliza la estadística del simulador que indica el número de instrucciones finalizadas. En este sentido, al recibir el mismo binario con los mismos valores de entrada, el número de instrucciones finalizadas debería ser similar en ambos simuladores. Sin embargo, este número no tiene por qué ser necesariamente idéntico, puesto que SimpleScalar y *Sim-async* modelan dos microarquitecturas distintas, aunque ambas ejecutan el mismo repertorio de instrucciones. Por tanto se hace necesario establecer una frontera de similitud. En esta misma línea, los desarrolladores de SimpleScalar ofrecen una justificación aplicada al desarrollo de su propio simulador arquitectónico, *sim-outorder*, en *http://www.simplescalar.com/faq.html#Q8*[1]:

> ***Q8.- How rigorously has SIM-OUTORDER's performance been verified? What kind of verification experiments have been done?***
>
> *[...] when SIM-OUTORDER was configured comparable to a dynamically scheduled stage processor, we found comparable results, within 5 % for SPEC92, we've also compared to other published results, but this has been less productive, since SIM-OUTORDER is more detailed than many of the other dynamically scheduled processor simulators on which we have published numbers.*

En consecuencia, se estima que diferencias inferiores al 5 % son aceptables siempre que se obtengan salidas correctas, por lo que en esta validación se tomará ese límite como frontera de similitud para el número de instrucciones finalizadas.

[1] El texto publicado en internet se reproduce aquí puesto que el enlace puede cambiar con el tiempo.

#### Simulaciones

La validación de *Sim-async* se basa en la comparación de tres conjuntos de simulaciones distintos:

- Simulaciones funcionales con SimpleScalar. Forman la base a partir de la cual se realizarán las comparaciones.
- Simulaciones arquitectónicas con *Sim-async* en modo síncrono. Permitirán la validación de una temporización similar a la original en SimpleScalar.
- Simulaciones arquitectónicas con *Sim-async* en modo asíncrono. Permitirá la validación de una temporización asíncrona. Estas simulaciones también servirán para comprobar que la microarquitectura modelada se comporta como un sistema asíncrono.

#### Bancos de prueba

Para comparar SimpleScalar y *Sim-async* se han utilizado los *benchmarks* SPEC2000 [Hen00]. Este conjunto de binarios permitirá verificar el correcto funcionamiento de *Sim-async*, haciendo posible además la comparación de los resultados de sus simulaciones con cualquier otro simulador arquitectónico que procese el mismo repertorio de instrucciones.

En pruebas previas a esta validación se obtuvieron tiempos de simulación muy elevados en simulaciones arquitectónicas con SimpleScalar (*sim-outorder*). Estos tiempos se debían a la carga de cómputo introducida por los datos de entrada que acompañan por defecto a los *benchmarks* SPEC2000. Por tanto, para prevenir este comportamiento en *Sim-async*, se decidió utilizar en todas sus simulaciones un conjunto de entradas reducido pero suficientemente representativo. Este conjunto de entradas se conoce como MinneSPEC [KL02], y fue reconocido por SPEC y distribuido con los *benchmarks* SPEC2000 a partir de su versión 1.2.

### 5.1.2. Experimentos para la validación

En primer lugar se presenta la validación del simulador para un sólo *benchmark*, para después extender la validación al resto de binarios. El *benchmark* escogido

es *gzip*, uno de los binarios pertenecientes a CINT2000, el conjunto de bancos de pruebas orientados al cómputo con números enteros en SPEC2000. El programa *gzip* realiza la compresión y descompresión de un fichero de entrada, tareas sobre las que aplica diversas comprobaciones sobre el número de bytes que obtiene como resultado de cada uno de sus pasos. Se ha elegido *gzip* por mostrar una salida muy representativa de su ejecución, incluyendo valores numéricos fácilmente comparables entre distintas ejecuciones del mismo binario.

La simulación funcional para *gzip* se lanzó utilizando *sim-safe*, uno de los simuladores básicos de SimpleScalar, obteniendo la salida que muestra la Figura 5.1. Como se aprecia en la figura, *gzip* imprime por pantalla diversos mensajes relacionados con la compresión y descompresión de un fichero de entrada. Al terminar la simulación, *sim-safe* imprime diversas estadísticas sobre su ejecución, entre las que se destaca el número de instrucciones que se han ejecutado, denotado aquí como *sim_num_insn*.

Por otro lado, en las simulaciones con *Sim-async* las posibilidades de configuración tanto de la microarquitectura como de su temporización son muy variadas. Por ello se hace necesario establecer una configuración básica común a todas las simulaciones de estas pruebas de validación. La configuración básica se compone de los valores asociados al predictor de saltos y a las estructuras por las que viajan las instrucciones en tiempo de ejecución: *IQ*, *RS*, bancos de registros y *ROB* (ver Apartado 4.2.2). Los valores de todos estos parámetros se resumen en la Tabla 5.1, y se introducen en el simulador utilizando un fichero de texto similar al que maneja SimpleScalar. Esta configuración, sin embargo, no se aplicó a las simulaciones con *sim-safe* porque éste es un simulador puramente funcional, utilizado como base para comprobar la correcta ejecución de los SPEC2000.

Una vez establecida la configuración microarquitectónica, se lanzaron dos simulaciones distintas para *gzip*. Por un lado, se configuró el simulador de modo que el procesador funcionase bajo temporización síncrona y, por otro lado, se estableció una temporización asíncrona aplicando suposiciones conservadoras a su configuración.

La temporización síncrona requiere que el tiempo de cómputo para cada una de las etapas y unidades funcionales (*FUs*) del procesador sea constante y se defina en función del tiempo de ciclo del reloj. Sin embargo, como se explicó

```
sim: ** starting functional simulation **

spec_init
Loading Input Data Duplicating 262144 bytes
Duplicating 524288 bytes
Input data 1048576 bytes in length
Compressing Input Data, level 1
Compressed data 108074 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 3
Compressed data 97831 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 5
Compressed data 83382 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 7
Compressed data 76606 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 9
Compressed data 73189 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Tested 1MB buffer: OK!
warning: partially supported sigprocmask() call...

sim: ** simulation statistics
sim_num_insn              601857009 # total number of instructions
executed
sim_num_refs              186176936 # total number of loads and stores
executed
sim_elapsed_time                 95 # total simulation time in seconds
...
```

Figura 5.1: Salida que produce SimpleScalar (*sim-safe*) al simular el binario *gzip*. Se incluyen, al final de la salida, algunas estadísticas de simulación generadas por *sim-safe*.

| ***Configuración*** | ***Valor*** |
|---|---|
| Predictor de saltos: | PAg 2 niveles |
| Nivel 1 | 1024 entr., hist. 10 |
| Nivel 2 | 1024 entr. |
| BTB | 4096 conj., 2-vías |
| Cola instrucciones (IQ) | 100 entradas |
| RS enteros | 6 entradas |
| RS suma P.F. | 3 entradas |
| RS mult., div. y raíz P.F. | 2 entradas |
| RS acceso a memoria | 5 entradas |
| Banco registros Enteros / P. F. | 32 / 32 |
| ROB | 100 entradas |

Tabla 5.1: Configuración básica de la microarquitectura en las simulaciones de *Sim-async*.

en el Capítulo 4, *Sim-async* no considera ciclos de reloj, sino que mantiene una línea de tiempo cuya medida es una unidad genérica llamada *unidades de tiempo* (u.t.). Así, cualquier valor de retardo se indica en u.t.. En el caso concreto de las simulaciones síncronas, se determinó que un ciclo de reloj equivaldría a 1000 u.t., y que el número de instrucciones a procesar simultáneamente en cada etapa (anchura de la etapa, *width*) sería de cuatro instrucciones. Para especificar esta configuración se utilizó el esquema XML que se muestra en la Figura 5.2.

Dado que el conjunto de instrucciones que se modela proviene del procesador Alpha 21264, se ha tratado de hacer corresponder los valores de tiempo de cómputo de cada módulo del procesador con sus valores reales (en ciclos) en el Alpha [Cor99]. Estos valores se muestran en la Tabla 5.2. Para conseguir esa analogía, *Sim-async* obtiene el tiempo de cómputo multiplicando el tiempo de ciclo por el valor correspondiente de la Tabla 5.2. De esta manera, el retardo asociado a cada dato se escala según el número de ciclos de la etapa análoga del Alpha.

Tras lanzar la simulación en *Sim-async* bajo la configuración síncrona descrita, se obtuvo una salida de *gzip* idéntica a la que generó ejecutando sobre *sim-safe*. En la Figura 5.3 se muestra la salida obtenida con *Sim-async* ejecutando en modo síncrono. Como se aprecia en la figura, los mensajes de *gzip* son idénticos a los mostrados en las simulaciones con *sim-safe* (ver Figura 5.1). Por otro lado, el número de instrucciones finalizadas en *Sim-async* (dato *sim_num_insn_commited*) está

```
<configuration name="todo_sync">
   <mode value="Sync" />
   <protocol name="Protocolo general">
      <clk t_cycle="1000" />
   </protocol>
   <stages>
      <stage name="fetch"> <width value="4"/> </stage>
      <stage name="issue"> <width value="4"/> </stage>
      <stage name="wb"> <width value="4"/> </stage>
      <stage name="commit"> <width value="4"/> </stage>
   </stages>
</configuration>
```

Figura 5.2: Esquema XML que define la configuración de las etapas en las simulaciones síncronas de *Sim-async*. El tiempo de ciclo se establece a 1000 u.t., mientras que la anchura para todas las etapas es de cuatro instrucciones.

| ***Etapa / FU*** | ***Num. ciclos*** |
|---|---|
| *Fetch, Issue, intUFx, Write-back, Commit* | 1 |
| *intMul* | 7 |
| *memUF, FPAdd, FPMul* | 4 |
| *FPDiv/Sqrt* | 30 |

Tabla 5.2: Número de ciclos de reloj asociados a etapas y *FU* en *Sim-async*. Se utilizan los mismos valores que en el procesador Alpha 21264, cuyo repertorio de instrucciones modela *Sim-async*.

muy cerca del obtenido en las simulaciones con *sim-safe* (dato *sim_ num_ insn* en la Figura 5.1). Concretamente, la diferencia entre ambos valores es de 110 instrucciones, un 0,00001 % del total, valor muy inferior al 5 % que se especificó en el apartado anterior como frontera de similitud.

En consecuencia, es posible afirmar que *Sim-async* modela una microarquitectura que ejecuta correctamente el conjunto de instrucciones Alpha para *gzip*, puesto que su simulación ha generado la misma salida y un número de instrucciones finalizadas similar a los obtenidos con *sim-safe*.

La configuración de *Sim-async* para la simulación bajo temporización asíncrona es algo más compleja que la anterior. En este escenario se utilizan los mismos valores para los parámetros arquitectónicos que en la simulación síncrona (ver Tabla 5.1). Además, la anchura asociada a cada una de las etapas es la misma que se aplicó a la simulación síncrona anterior (cuatro instrucciones). Sin embargo, al modelar una temporización asíncrona es necesario incluir un protocolo de comunicación entre etapas. En esta configuración se utilizó el protocolo de comunicación *handshake* de cuatro fases (ver Apartado 4.2.3 y Apéndice A) para todas las comunicaciones entre etapas y *FUs*. Todos estos parámetros se muestran en la Figura 5.4, donde se presenta el esquema XML utilizado para configurar *Sim-async* en estas simulaciones bajo temporización asíncrona. En este esquema también se indica el tiempo de cómputo a utilizar por cada una de las etapas de la microarquitectura.

El tiempo de cómputo asociado a los módulos del procesador en las simulaciones asíncronas se describe, como indica el esquema referido anteriormente, utilizando una función de distribución que se define en el fichero *mc_ distrib.xml*. Esta función se aplica a todas las etapas y *FUs* de la microarquitectura. La Figura 5.5 muestra el esquema XML contenido en dicho fichero, el cual establece que el tiempo de cómputo de los módulos sea variable. En ese sentido, considerando como tiempos de cómputo mínimo y máximo a los valores 0 u.t. y 1000 u.t., la función de distribución determina que los retardos con mayor probabilidad de ser elegidos sean aquellos cercanos al tiempo de cómputo medio, 500 u.t., como se aprecia en la Figura 5.6. Al igual que ocurre en las simulaciones bajo temporización síncrona, los valores de tiempo de cómputo de las etapas multiciclo se escalan siguiendo los tiempos de ciclo indicados en la Tabla 5.2.

```
sim: ** starting architectural simulation **
# You are running "SYNCHRONOUS-like" implementation.

spec_init
Loading Input Data Duplicating 262144 bytes
Duplicating 524288 bytes
Input data 1048576 bytes in length
Compressing Input Data, level 1
Compressed data 108074 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 3
Compressed data 97831 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 5
Compressed data 83382 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 7
Compressed data 76606 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 9
Compressed data 73189 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Tested 1MB buffer: OK!
warning: partially supported sigprocmask() call...

sim: ** simulation statistics **
sim_num_insn_commited     601857119 # total number of instructions
finalized (commited)
sim_num_insn             1023788989 # total number of instructions
executed (also speculative)
ult_commit             1203432862000 # finish time of last commit
...
```

Figura 5.3: Salida que produce el binario *gzip* al simular con *Sim-async* bajo temporización síncrona. Se incluyen, al final de la salida, algunas estadísticas de simulación generadas por *Sim-async*.

```
<configuration name="tesis_obj1">
   <mode value="Async" />
   <protocol name="Protocolo general">
      <drail4ph tfv="2" tfn="2" tack="3" tsync="2" />
   </protocol>
   <stages>
      <stage name="fetch">
         <width value="4"/>
         <delay> <distrib file="mc_distrib.xml"/> </delay>
      </stage>
      <stage name="issue">
         <width value="4"/>
         <delay> <distrib file="mc_distrib.xml"/> </delay>
      </stage>
      <stage name="exec">
         <substage name = "memUF" >
            <delay> <distrib file="mc_distrib.xml"/> </delay>
         </substage>
           ...
         <substage name = "fpAddSegUF" >
           <delay> <distrib file="mc_distrib.xml"/> </delay>
         </substage>
      </stage>
      <stage name="wb">
         <width value="4"/>
         <delay> <distrib file="mc_distrib.xml"/> </delay>
      </stage>
      <stage name="commit">
         <width value="4"/>
         <delay> <distrib file="mc_distrib.xml"/> </delay>
      </stage>
      </stages>
</configuration>
```

Figura 5.4: Esquema XML con la configuración de las etapas en las simulaciones asíncronas de *Sim-async*. Se establece un protocolo de comunicación *handshake* de cuatro fases común a todos los módulos, un tiempo de cómputo determinado por la distribución definida en el archivo *mc_ distrib.xml*, y una anchura de cuatro instrucciones para todas las etapas. Se omite la configuración de algunas *FUs* dentro de *Exec*, dado que son similares al resto.

```
<distrib>
   <name value="mc_distrib"/>
   <items prec="6">
       <item delay="10"  prob="0.000052"/>
       <item delay="460" prob="0.000004"/>
       <item delay="470" prob="0.000084"/>
       <item delay="480" prob="0.000364"/>
       <item delay="490" prob="0.004108"/>
       <item delay="500" prob="0.018720"/>
       <item delay="520" prob="0.041212"/>
       <item delay="530" prob="0.058880"/>
       <item delay="540" prob="0.074264"/>
       <item delay="550" prob="0.088496"/>
       <item delay="560" prob="0.102132"/>
       <item delay="570" prob="0.121548"/>
       <item delay="580" prob="0.046344"/>
       <item delay="590" prob="0.038820"/>
       <item delay="600" prob="0.067940"/>
       <item delay="610" prob="0.046056"/>
       <item delay="620" prob="0.037340"/>
       <item delay="630" prob="0.042292"/>
       <item delay="640" prob="0.040804"/>
       <item delay="650" prob="0.032368"/>
       <item delay="660" prob="0.003880"/>
       <item delay="670" prob="0.026584"/>
       <item delay="680" prob="0.021276"/>
       <item delay="690" prob="0.014480"/>
       <item delay="700" prob="0.006272"/>
       <item delay="710" prob="0.006856"/>
       <item delay="720" prob="0.008032"/>
       <item delay="730" prob="0.011336"/>
       <item delay="740" prob="0.007740"/>
       <item delay="750" prob="0.001716"/>
       <item delay="760" prob="0.017388"/>
       <item delay="770" prob="0.004336"/>
       <item delay="780" prob="0.000184"/>
       <item delay="790" prob="0.000760"/>
       <item delay="800" prob="0.000052"/>
       <item delay="810" prob="0.002052"/>
       <item delay="820" prob="0.000720"/>
       <item delay="830" prob="0.001508"/>
       <item delay="840" prob="0.000792"/>
       <item delay="850" prob="0.001896"/>
       <item delay="860" prob="0.000096"/>
       <item delay="880" prob="0.000036"/>
       <item delay="900" prob="0.000020"/>
       <item delay="930" prob="0.000028"/>
       <item delay="940" prob="0.000020"/>
       <item delay="950" prob="0.000024"/>
       <item delay="960" prob="0.000016"/>
       <item delay="970" prob="0.000012"/>
       <item delay="980" prob="0.000016"/>
       <item delay="990" prob="0.000020"/>
       <item delay="1000" prob="0.000025"/>
   </items>
</distrib>
```

Figura 5.5: Esquema XML contenido en el fichero *mc_ distrib.xml*. La función de distribución que define el esquema determina que los tiempos de cómputo con mayor probabilidad sean los cercanos al caso medio (500 u.t.).

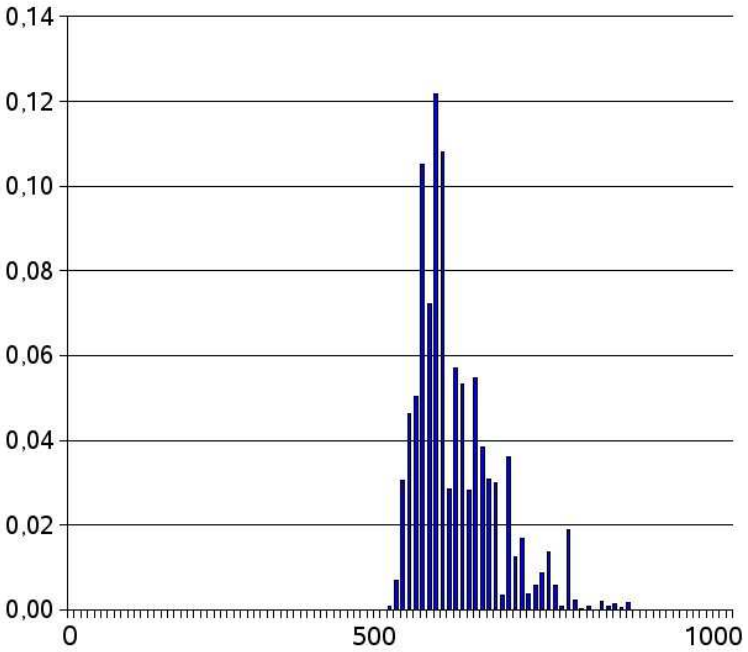


Figura 5.6: Función de distribución que define el esquema XML que se muestra en la Figura 5.5.

Bajo la configuración asíncrona descrita anteriormente, la simulación de *gzip* en *Sim-async* generó la salida que se muestra en la Figura 5.7. De nuevo, los mensajes que emite *gzip* son idénticos a los que se encuentran tanto en la simulación con *sim-safe* como en la simulación síncrona con *Sim-async*. Del mismo modo, el número de instrucciones finalizadas en esta simulación con *Sim-async* es prácticamente igual al obtenido con *sim-safe*. La diferencia entre ambos datos es de 152 instrucciones, representando un 0,00001 % de diferencia, de nuevo claramente inferior a la cota del 5 % explicada en el Apartado 5.1.1.

En este caso se han obtenido salidas idénticas para el binario, y un número de instrucciones finalizadas similar. Por tanto, al igual que para el caso síncrono, es posible afirmar que *Sim-async* modela una microarquitectura que ejecuta correctamente el conjunto de instrucciones Alpha para *gzip*. Esta afirmación se aplica ahora a la configuración asíncrona con latencias variables, donde las comunicaciones se realizan a través de un protocolo *handshake* de cuatro fases.

Las configuraciones descritas tanto para *sim-safe* como para *Sim-async*, éste último bajo temporizaciones síncrona y asíncrona, se aplicaron de manera análoga

```
sim: ** starting architectural simulation **
# You are running "ASYNCHRONOUS-like" implementation.

spec_init
Loading Input Data Duplicating 262144 bytes
Duplicating 524288 bytes
Input data 1048576 bytes in length
Compressing Input Data, level 1
Compressed data 108074 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 3
Compressed data 97831 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 5
Compressed data 83382 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 7
Compressed data 76606 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Compressing Input Data, level 9
Compressed data 73189 bytes in length
Uncompressing Data
Uncompressed data 1048576 bytes in length
Uncompressed data compared correctly
Tested 1MB buffer: OK!
warning: partially supported sigprocmask() call...

sim: ** simulation statistics **
sim_num_insn_commited     601857161 # total number of instructions
finalized (commited)
sim_num_insn              910685127 # total number of instructions
executed (also speculative)
ult_commit             598032093779 # finish time of last commit
...
```

Figura 5.7: Salida que produce el binario *gzip* al simular con *Sim-async* bajo temporización asíncrona. Se incluyen, al final de la salida, algunas estadísticas de simulación generadas por *Sim-async*.

| ***CINT2000*** | |
|---|---|
| ***SPEC*** | ***Num. Instr. Ejecutadas*** |
| *bzip* | *3253843516* |
| *crafty* | *168886264* |
| *gap* | *130742699* |
| *gcc* | *3728937372* |
| *gzip* | *1104801824* |
| *parser* | *530639204* |
| *perlbmk* | *322038396* |
| *vortex* | *760115* |

| ***CFP2000*** | |
|---|---|
| ***SPEC*** | ***Num. Instr. Ejecutadas*** |
| *ammp* | *84246765* |
| *apsi* | *244240882* |
| *galgel* | *199880850* |
| *lucas* | *23362025* |
| *mesa* | *2417544756* |
| *sixtrack* | *14910424* |
| *swim* | *32331540* |

Tabla 5.3: Número de instrucciones ejecutadas en las simulaciones bajo temporización asíncrona de *Sim-async* (estadística *sin_ num_ insn*) de varios *benchmarks* SPEC2000, tanto del conjunto de enteros, CINT2000, como de punto flotante, CFP2000.

al resto de *benchmarks*. De esta manera se aplicó el método de validación de *Sim-async* ya explicado sobre el resto de binarios del conjunto de SPEC2000, tanto para enteros como para punto flotante. En la Tabla 5.3 se muestra el número total de instrucciones simuladas para los *benchmarks* simulados. En total se ejecutaron 45 simulaciones de binarios SPEC2000 con *Sim-async*, lo que supone la simulación de más de $2,8 \cdot 10^{10}$ instrucciones (incluyendo las especuladas). Los resultados obtenidos fueron los siguientes:

1. Todos los SPEC simulados con *Sim-async*, tanto bajo temporización síncrona como asíncrona, generaron salidas idénticas a las producidas por el simulador funcional de SimpleScalar, *sim-safe*.

2. En todos los SPEC las estadísticas de número de instrucciones finalizadas obtenidas con *Sim-async* son similares a los números de instrucciones simuladas con *sim-safe*. En la Tabla 5.4 se muestran las diferencias obtenidas entre las ejecuciones de *sim-safe* y las ejecuciones de *Sim-async* bajo temporización asíncrona. Se puede apreciar que, porcentualmente, las diferencias son muy pequeñas, y siempre se encuentran muy por debajo del 5 % establecido como frontera de similitud por los desarrolladores de SimpleScalar.

En consecuencia, es posible afirmar que *Sim-async* modela una microarquitectura superescalar que ejecuta correctamente el repertorio de instrucciones del

| ***SPEC*** | ***sim-safe*** | ***Sim-async (asinc.)*** | ***Dif. ( %)*** |
|---|---|---|---|
| *ammp* | 45812883 | 45810845 | -0,004 |
| *apsi* | 197579651 | 197612776 | 0,017 |
| *bzip* | 1819780172 | 1819780267 | 0,000 |
| *crafty* | 94419973 | 94420229 | 0,000 |
| *galgel* | 139306245 | 139310055 | 0,003 |
| *gap* | 82873902 | 82874407 | 0,001 |
| *gcc* | 2016139124 | 2016204817 | 0,003 |
| *gzip* | 601857009 | 601857161 | 0,000 |
| *lucas* | 19239488 | 19242782 | 0,017 |
| *mesa* | 1608605448 | 1608410610 | -0,012 |
| *parser* | 268979662 | 269006191 | 0,010 |
| *perlbmk* | 205853718 | 205914747 | 0,030 |
| *sixtrack* | 11699655 | 11724227 | 0,210 |
| *swim* | 23557475 | 23562358 | 0,021 |
| *vortex* | 453666 | 454534 | 0,191 |

Tabla 5.4: Número de instrucciones finalizadas en *sim-safe* (estadística *sim_num_insn*) y en las simulaciones bajo temporización asíncrona de *Sim-async* (estadística *sin_num_insn_commited*) de varios *benchmarks* SPEC2000. Las diferencias se muestran porcentualmente.

| ***Estadística*** | ***Descripción*** |
|---|---|
| *sim_fetch_times* | Ejecuciones de etapa *Fetch* |
| *sim_issue_times* | Ejecuciones de etapa *Issue* |
| *sim_Int_FU_times* | Ejecuciones en FU de suma/resta de enteros |
| *sim_IntMul_FU_times* | Ejecuciones en FU de multiplicación/división de enteros |
| *sim_FPAdd_FU_times* | Ejecuciones en FU de suma/resta en pto. flotante |
| *sim_FPMul_FU_times* | Ejecuciones en FU de multiplicación en pto. flotante |
| *sim_FPDiv_FU_times* | Ejecuciones en FU división en pto. flotante |
| *sim_Addr_FU_times* | Ejecuciones en FU de cálculo de direcciones |
| *sim_Mem_FU_times* | Núm. de accesos a memoria de datos |
| *sim_wb_times* | Ejecuciones de etapa *Write-back* |
| *sim_commit_times* | Ejecuciones de etapa *Commit* |

Tabla 5.5: Estadísticas sobre el número de ocasiones en que se utiliza cada etapa ó *FU* del procesador en *Sim-async*. Esta tabla es un extracto de la Tabla 4.4.

procesador Alpha 21264 tanto en configuración síncrona como en configuración asíncrona.

Por otro lado, una de las principales diferencias en el modelado del procesador bajo temporización síncrona frente a la temporización asíncrona es la tasa de actividad de cada etapa ó *FU*. Suponiendo que no se implementen técnicas de *clock gating* ó similares ejecutando en configuración síncrona, todos los módulos del procesador estarán activos en cada ciclo de reloj. Sin embargo, la temporización asíncrona provoca que una etapa ó *FU* sólo trabaje si le llegan datos que procesar.

Para verificar el correcto modelado de la temporización asíncrona en *Sim-async*, se comprobaron los valores que registraron los contadores que se muestran en la Tabla 5.5. Estos contadores reflejan la tasa de actividad de cada etapa ó *FU* del procesador ejecutando bajo temporización asíncrona. La Tabla 5.6 muestra la reducción promedio en el número de ocasiones que ejecuta cada etapa en las simulaciones de los SPEC2000 en el caso asíncrono frente al síncrono. Estas reducciones van del 42,076 % de la etapa *Fetch*, cuya utilización es continua en ambos tipos de simulación, al 99,921 % de la *FU* para división en punto flotante, que permanece ociosa la mayor parte del tiempo en la simulaciones asíncronas.

Según los resultados de las simulaciones, se puede afirmar que la temporización asíncrona se modela correctamente puesto que las salidas obtenidas para los *ben-*

| ***Asinc vs. Sinc.*** | ***Dif. prom. ( %)*** |
|---|---|
| *Fetch* | -42,076 |
| *Issue* | -61,993 |
| *FU enteros* | -66,062 |
| *FU Mult. ent.* | -99,894 |
| *FU Suma P.F.* | -95,665 |
| *FU Mult. P.F.* | -97,733 |
| *FU Div. P.F.* | -99,921 |
| *FU Direc. Mem.* | -79,314 |
| *FU Mem.* | -81,116 |
| *Write-back* | -52,324 |
| *Commit* | -73,852 |

Tabla 5.6: Diferencia promedio en la tasa de actividad para cada etapa y *FU* entre las simulaciones bajo temporización síncrona y asíncrona ejecutadas en *Sim-async*.

*chmarks* son iguales, y los números de instrucciones finalizadas son similares. Además, dado que la tasa de actividad de cada uno de los módulos del procesador se ajusta a la demanda de cómputo que recibe en cada momento, se puede igualmente asegurar que la microarquitectura simulada se comporta de manera asíncrona. Si no fuera así, los valores de los contadores en las simulaciones asíncronas serían iguales al número de ciclos que cada binario emplea en ejecutar. En conclusión, se puede afirmar que *Sim-async* modela correctamente el comportamiento asíncrono. Por tanto, la validación del simulador concluye satisfactoriamente.

## 5.2. Estudios arquitectónicos con *Sim-async*

Tras confirmar la correcta ejecución de *Sim-async* bajo distintos tipos de configuración, se plantea la utilización del simulador para evaluar el rendimiento del procesador superescalar de 64 bits bajo distintas configuraciones de temporización.

En términos generales, el rendimiento de un procesador habitualmente se mide calculando valores de *CPI* (ciclos por instrucción). Sin embargo, en lugar de ciclos, *Sim-async* considera una línea de tiempo en la simulación, por lo que no

es posible utilizar como medida los *CPI*. Como alternativa, se podrían utilizar las *MIPS* (millones de instrucciones por segundo), pero *Sim-async* no mide el tiempo en segundos, sino en unidades genéricas de tiempo (u.t.), por lo que las *MIPS* tampoco representan una medida adecuada.

*Sim-async* mide el rendimiento de una configuración calculando el tiempo que la microarquitectura que modela emplea en ejecutar un programa. Este retardo se contabiliza utilizando el contador *ult_ commit*, que almacena el instante de tiempo en que se ejecutó por última vez la etapa *Commit* en cada binario simulado. En consecuencia, siempre que se mencione en ese apartado el rendimiento de una configuración se estará haciendo referencia al retardo que el procesador modelado emplea en ejecutar un determinado programa.

### 5.2.1. Procesador síncrono vs. procesador asíncrono

Dado que *Sim-async* permite modelar la ejecución de un procesador superescalar de 64 bits bajo configuraciones tanto síncronas como asíncronas, una aplicación inmediata del simulador es, precisamente, comparar el rendimiento que obtiene la microarquitectura en esas dos configuraciones.

En ese sentido, la idea que motiva esta comparación es evaluar si existe pérdida ó ganancia de rendimiento al ejecutar *benchmarks* estándar en una microarquitectura superescalar asíncrona en comparación con su equivalente síncrona. El resultado es incierto *a priori* puesto que existen puntos a favor y en contra de la temporización asíncrona. Por un lado, al considerar tiempos de cómputo variables en las etapas y *FUs* del procesador, se producirá un aumento teórico del rendimiento porque la probabilidad de trabajar con el retardo más lento es pequeña, mientras que en temporización síncrona siempre se trabaja en el caso peor. Por otro lado, el protocolo de comunicación y la lógica de control que necesita la implementación asíncrona introducen una penalización que en la versión síncrona no existe. Por tanto, para evaluar las diferencias en rendimiento se necesita un simulador arquitectónico como *Sim-async*, capaz de aplicar distintas configuraciones temporales a un mismo procesador.

### Configuración síncrona

Las simulaciones bajo temporización síncrona de este experimento fueron las mismas que se lanzaron en la validación del simulador (véase el apartado anterior). Por tanto, la configuración básica de la microarquitectura, los tiempos de cómputo y las anchuras de las etapas son las mismas que se definieron anteriormente (ver tablas 5.1 y 5.2). A pesar de que todos estos valores de configuración ya se han explicado en este mismo capítulo, a continuación se añaden algunos comentarios adicionales sobre los retardos considerados en esta configuración.

Como se explicó en el Apartado 4.2.3, al modelar una temporización síncrona *Sim-async* interpreta que el tiempo de cómputo de una etapa ó *FU* ($t_c$) corresponde al tiempo de ciclo ($t_{ciclo}$). Sin embargo, según se ilustra en [HBJ+02], el ciclo de reloj de un procesador no se compone únicamente del tiempo de cómputo, sino que se define según la siguiente ecuación:

$$t_{ciclo} = t_{logic} + t_{latch} + t_{skew} + t_{jitter} \tag{5.1}$$

donde $t_{logic}$ es el retardo empleado en el cómputo de la lógica de la etapa, definido en el simulador como $t_c$; $t_{latch}$ corresponde al margen de seguridad necesario para que las señales se estabilicen a la entrada de los registros (tiempos de *setup* y *hold*); y $t_{skew}$ y $t_{jitter}$ corresponden a las penalizaciones debidas a la señal de reloj. Según esta ecuación, el tiempo dedicado de manera efectiva al cómputo es $t_{logic}$, mientras que el resto de retardos se deben a la naturaleza síncrona del procesador. Por tanto, en un sistema asíncrono el retardo máximo para el tiempo de cómputo debería ser $t_{logic}$, tiempo al que se agregaría el retardo empleado por el protocolo de comunicación. En consecuencia, el tiempo debido a $t_{latch}$, $t_{skew}$ y $t_{jitter}$ en el caso síncrono solaparía con parte de la penalización por protocolo del caso asíncrono. Este hecho permitiría realizar suposiciones de temporización que favorecerían al caso asíncrono. En cambio, se decidió no aplicar esta equivalencia a las simulaciones de este experimento para no proporcionar ventaja a la configuración asíncrona. En ese sentido, las simulaciones bajo temporización síncrona asumen, como se explicó en el Apartado 4.2.3, que

$$t_{ciclo} = t_{logic} = t_c \tag{5.2}$$

En resumen, con esta especificación se considera una microarquitectura síncrona cuya temporización se ha simplificado sin discriminar el tiempo de cómputo efectivo, por lo que su rendimiento no sufre ninguna penalización frente al caso asíncrono (más bien se favorece con respecto al caso síncrono).

### Configuración asíncrona

La temporización de la configuración asíncrona se corresponde con una microarquitectura donde los tiempos de cómputo de los datos son variables. En ese sentido, la funcionalidad del circuito se encuentra totalmente desacoplada de su temporización, por lo que se trata de un modelo temporal radicalmente opuesto al síncrono. Este desacoplamiento obliga a utilizar codificaciones especiales y protocolos de comunicación para que la lógica de control gobierne el cómputo del circuito (ver Apéndice A).

La configuración asíncrona utilizada en esta comparación se podría implementar bajo el modelo insensible a retardos (IR, véase el Apartado 4.2.3) utilizando codificación en doble raíl e incluyendo un protocolo *handshake* de cuatro fases (ver Apéndice A). Para modelar la latencia de este tipo de circuitos se combinaron dos expresiones ya descritas en el Apartado 4.2.3: la ecuación que define los retardos del protocolo de cuatro fases (Ecuación (4.8)) y la expresión que define la latencia de un módulo asíncrono bajo el modelo IR (Ecuación (4.9)). Al sustituir en la segunda los valores de la primera se obtiene como resultado la expresión siguiente, que define la latencia de un módulo IR bajo el protocolo de cuatro fases en doble raíl:

$$L_{IR_{4f}} = t_{cv} + e_r + t_{fv} + t_{ack} + t_{sync} + t_{fn} + t_{ack} \tag{5.3}$$

En esta expresión, $t_{cv}$ es la latencia de la lógica de cómputo y $e_r$ es el tiempo de espera por el receptor. El resto de los argumentos son retardos referidos al protocolo que se analizarán en detalle más adelante.

Las simulaciones requieren que todos los factores de la expresión anterior tengan un valor de tiempo (en u.t.). Todos salvo $e_r$, que es el único factor de la expresión para el que no hay valor explícito de retardo puesto que éste depende del estado

del sistema en cada operación. La simulación que realiza *Sim-async* tiene en cuenta todos los tiempos de espera de tipo $e_r$, que se conocen gracias al minucioso modelado de la microarquitectura y del protocolo que se realiza. A continuación se presenta la estimación de los valores para los retardos anteriores (excepto $e_r$) con objeto de definir una configuración asíncrona equiparable a la configuración síncrona con la que se va a comparar.

Un circuito asíncrono bajo el modelo IR puede emplear un retardo distinto en el cálculo de diferentes datos de entrada. Como se explicó en el Capítulo 3, es posible caracterizar el comportamiento variable de un circuito utilizando funciones de distribución de probabilidad. En consecuencia, los valores de $t_c$ para esta configuración asíncrona se podrían caracterizar aplicando una función de distribución en cada etapa.

Sin embargo, la variabilidad en el tiempo de cómputo de cada etapa depende de la funcionalidad que implementa. Por tanto, se ha analizado la funcionalidad de cada una de las etapas, así como la de todas las *FUs* del procesador, con objeto de determinar si se caracterizan aplicando distribuciones de tiempos ó bien se les asigna un retardo constante. Este análisis no pretende obtener descripciones exactas del comportamiento temporal de cada módulo de procesador. Por el contrario, su motivación fundamental es evaluar, en función de una posible implementación real, el grado de variabilidad en el tiempo de cómputo que cada etapa ó *FU* podría presentar. El análisis, detallado por etapas, es el siguiente:

- *Fetch*: la latencia de esta etapa depende, fundamentalmente, de dos factores. Por un lado, del tiempo que se tarda en leer de la cache de instrucciones, y por otro lado, del tiempo empleado por el predictor de saltos. En estas simulaciones no se ha realizado ningún estudio sobre las posibles variaciones en la latencia debidas a los diferentes datos de entrada en ambos componentes de la etapa. En consecuencia, se ha supuesto que el tiempo de cómputo de *Fetch* estará cerca del caso peor síncrono, que es de un ciclo. Por tanto, se ha establecido que el retardo de esta etapa sea constante, empleando 1000 u.t., valor similar al retardo de un ciclo en el caso síncrono. Para evaluar esta decisión sobre el tiempo de cómputo de *Fetch* se ha tomado como referencia la etapa análoga del procesador Alpha 21264. La implementación se puede

considerar equivalente puesto que también realiza la lectura de instrucciones y la predicción de saltos en cada ejecución de la etapa, procesando el mismo repertorio de instrucciones.

- *Issue*: la mayor influencia sobre el retardo de esta etapa la ejerce el tiempo de acceso al banco de registros. Consultando de nuevo los resultados de [HBJ+02], se indica que la latencia del banco de registros es de 0,39 ns. Por tanto, considerando el retardo descrito para *Fetch* es posible suponer, utilizando una estimación conservadora, que es factible decodificar y lanzar dos instrucciones en paralelo en la mitad del tiempo que emplea *Fetch* en ejecutar. Por tanto, la latencia de *Issue* será constante, empleando 500 u.t. en cada ejecución.

- *Exec*: en términos generales, las *FUs* que componen esta etapa podrían presentar una variabilidad en el tiempo de cómputo mayor que la que se aprecia en otras etapas. Esta intuición se confirma en la caracterización del sumador de 64 bits que se presenta en el Apartado 3.4, donde se aprecia que la mayoría de los datos emplean un tiempo de cómputo menor que el camino crítico del circuito. Ese estudio de los tiempos de cómputo justifica la utilización de tiempos variables en las *FUs* del procesador bajo configuración asíncrona.

  Cada una de las *FUs* podría presentar un comportamiento distinto que sólo se averiguará aplicando a los siete tipos de *FU* un estudio similar al realizado para el sumador. Sin embargo, estas simulaciones arquitectónicas no pretenden comparar datos reales. Al contrario, se pretende justificar de manera conservadora un comportamiento asíncrono para el procesador y comparar su rendimiento con la configuración asíncrona. Por tanto, se decidió uniformizar el comportamiento de todas las *FUs* utilizando la misma función de distribución, denominada *SC* (*slow case*, caso lento). Esta función se crea a partir de la distribución *Dm* obtenida en la caracterización descrita en el Apartado 3.4. El aspecto final de la distribución *SC* se muestra en la Figura 5.8. Para las *FUs* multiciclo se ha escalado el valor máximo de esta normalización según el número de ciclos que cada una tarda en computar. En consecuencia, estas *FUs* no se consideran segmentadas, aunque su variación es mayor que aquellas ceñidas a un sólo ciclo de reloj.

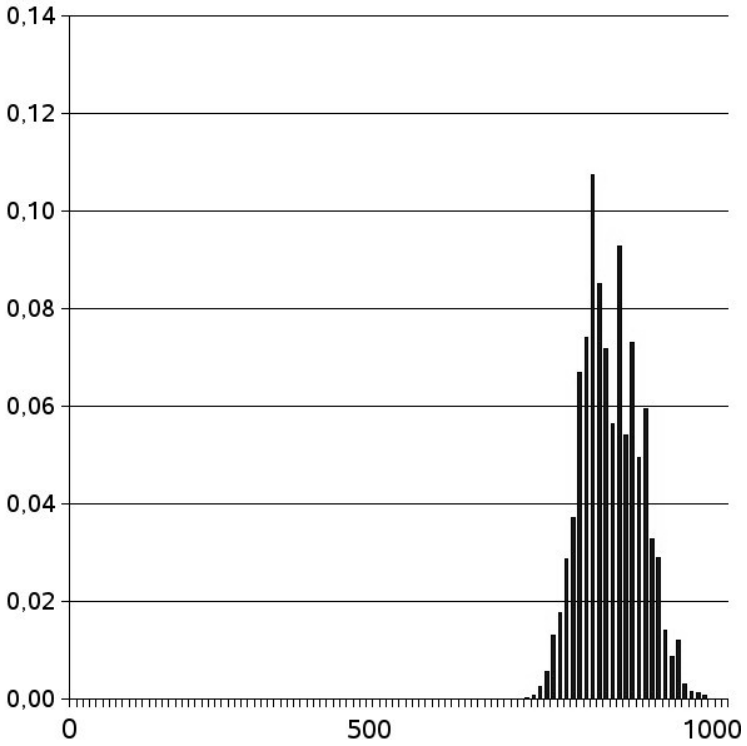


Figura 5.8: Función de distribución *SC*. Muestra un grado de variabilidad moderado del tiempo de cómputo, presentando un caso promedio cercano al camino crítico. El eje de las abscisas muestra, en u.t., los tiempos de cómputo, mientras que el eje de las ordenadas muestra la probabilidad para cada tiempo.

- *Write-back*: el tiempo de cómputo de esta etapa depende de dos factores. Por un lado, depende de los retardos empleados por la lógica de selección en elegir, de entre todos los *FFs*, los datos que viajarán por el *CDB*. Por otro lado, depende del tiempo que emplea la lógica de *wake-up* en activar las instrucciones que contienen las *RS* al otro extremo del *CDB*. Una vez más, se requiere una justificación adecuada para estos retardos.

  Según el trabajo de [PJS97], el retardo de la lógica de selección depende del número de árbitros implementados en el circuito. En esta etapa, el número de árbitros a implementar es bajo, puesto que sólo tendrían que evaluar la disponibilidad de datos en los ocho *FFs* que se encuentran a la salida de las *FUs*. Suponiendo, como se hizo en la etapa *Issue*, que la implementación utiliza tecnología de 180 nm, los resultados de [PJS97] indican que el retardo de la lógica de selección sería de 0,4 ns.

  Por otro lado, el retardo de la lógica de *wake-up* depende del tamaño de las estructuras en que se van a escribir los datos. En esta microarquitectura, la difusión de resultados se puede realizar en paralelo para todas las *RS* y el *ROB*. Por tanto, el mayor retardo corresponde al *ROB*, que es la estructura destino de mayor tamaño, con una longitud de 100 entradas (ver Tabla 5.1). Según [PJS97], el *wake-up* de dos instrucciones (*two-ways*) para una ventana de tamaño 100 emplea un retardo de 0,2 ns en 180 nm.

  Así, al igual que en *Issue*, el trabajo de *Write-back* encaja en medio ciclo, aunque procesando la mitad de instrucciones que el caso síncrono y utilizando dos instancias del *CDB* en lugar de cuatro. De nuevo, no se estudia la posible variabilidad en el tiempo de cómputo, por tanto la configuración asíncrona de la etapa *Write-back* tendrá una anchura de dos instrucciones, empleando un retardo constante de 500 u.t.. Una solución alternativa podría utilizar cuatro instancias del *CDB* y mantener una anchura de cuatro instrucciones en 1000 u.t., similar al caso síncrono. Sin embargo, dado que las *FUs* tienen tiempos de ejecución variables, sus resultados podrán estar disponibles antes del caso peor de 1000 u.t., por lo que si *Write-back* tuviese un retardo de 1000 u.t. no podría aprovechar esa variación en los tiempos.

- *Commit*: el análisis de tiempo para esta etapa es análogo al que se presenta para la etapa *Fetch*. En este caso también se podrían procesar un máximo de

| | ***Síncrona*** | | ***Asíncrona*** | |
|---|---|---|---|---|
| ***Etapa*** | $t_c$***(u.t.)*** | ***Anchura*** | $t_c$ ***(u.t.)*** | ***Anchura*** |
| *Fetch* | 1000 | 4 | 1000 | 4 |
| *Issue* | 1000 | 4 | 500 | 2 |
| *Exec* | depende de *FU* | - | función distr. | - |
| *Write-back* | 1000 | 4 | 500 | 2 |
| *Commit* | 1000 | 4 | 1000 | 4 |

Tabla 5.7: Tiempos de cómputo de las etapas del procesador en configuraciones síncrona y asíncrona. Se indica también la anchura, es decir, el máximo número de instrucciones que cada etapa procesa.

cuatro instrucciones, siempre que todas hayan terminado sus fases anteriores y no se encuentren saltos mal predichos. La variabilidad tampoco se ha considerado en este caso, de modo que el tiempo de cómputo de esta etapa será también constante, empleando 1000 u.t..

La Tabla 5.7 resume ambas configuraciones temporales para el procesador. La configuración síncrona presenta tiempos de cómputo y anchuras similares en todas las etapas. Sin embargo, en la configuración asíncrona existe una heterogeneidad propia de los circuitos insensibles a retardos. En esta configuración se ha establecido que las etapas *Fetch* y *Commit* se comporten igual que en el caso síncrono. Las etapas *Issue* y *Write-back* también mantienen un tiempo de cómputo constante, sólo que corresponde a la mitad del tiempo de ciclo síncrono, por lo que utilizan la mitad de anchura que en la configuración síncrona. En consecuencia, el rendimiento teórico es equivalente a la configuración síncrona.

Todas las suposiciones sobre los tiempos de cómputo de estas etapas se fundamentan en el análisis de los retardos basado en la literatura que se ha expuesto en los párrafos anteriores. La única variación en el tiempo de cómputo se aplica en las *FUs*, y se basa en la caracterización descrita en el Apartado 3.4.

Los valores asociados a los retardos de la configuración asíncrona son conservadores y sufren una penalización con respecto a la configuración síncrona. Para medir esta penalización es necesario retomar el estudio que se presenta en [HBJ+02], donde se muestra información sobre los retardos del procesador de referencia, el Alpha 21264 en su versión de 180 nm, con una frecuencia de reloj de 800 MHz. Según el artículo, de los 1,25 ns del tiempo de ciclo del Alpha, 1,13 ns corresponden

a la lógica de cómputo ($t_{logic}$ en la Ecuación (5.1); $t_c$ en el simulador), mientras que los restantes 0,12 ns corresponden a la suma del resto de factores debidos a la señal de reloj ($t_{latch} + t_{skew} + t_{jitter}$ en la Ecuación (5.1)). Por tanto, el camino crítico de la lógica de cómputo representa el 90,4 % del tiempo de ciclo. Teniendo en cuenta que la configuración asíncrona no dispone de señal de reloj, su tiempo de cómputo máximo debería emplear el mismo porcentaje sobre las 1000 u.t., es decir, cada ciclo correspondería a 904 u.t. más el retardo asociado al protocolo de comunicación. Sin embargo, se ha elegido utilizar el tiempo completo de un ciclo, 1000 u.t., para no sesgar esta comparación hacia la versión asíncrona del procesador. En consecuencia, los tiempos de cómputo de las etapas de la configuración asíncrona sufren una penalización del 9,6 % sobre el caso peor.

Además, como ya se ha mencionado, la configuración asíncrona debe considerar los retardos asociados al protocolo de comunicación entre etapas. En este experimento se ha configurado *Sim-async* para modelar el protocolo *handshake* de cuatro fases bajo codificación doble raíl. Sus valores de retardo, incluidos como parámetros globales del simulador ($t_{fv}$, $t_{fn}$, $t_{sync}$ y $t_{ack}$, ver Apartado 4.2.3), se aplican a las comunicaciones tanto de las etapas con tiempo de cómputo variable como a las de latencia constante.

Para establecer el valor de los parámetros del simulador correspondientes al protocolo se ha considerado [Che98]. En este artículo se implementa un circuito de detección y sincronización (*reset completion-detection*) de lineas de datos utilizando el protocolo *handshake* de cuatro fases bajo codificación doble raíl en tecnología CMOS de 2 $\mu$m. Cheng obtiene un valor promedio de 0,28 ns para el circuito de detección de fin de cómputo, cuyo valor se aplicaría a $t_{fv}$. También obtiene un retardo de 0,71 ns. para la sincronización ó *reset*, correspondiente al parámetro $t_{sync}$ del simulador. Sin embargo, estos valores no se pueden aplicar directamente al simulador porque se trata de valores para una tecnología distinta a la que se está utilizando como referencia, que son los 180 nm del Alpha 21264 a 800 MHz.

La solución a este inconveniente la proporciona la métrica *FO4* (*fan-out-of-four*) [HMH01]. Esta métrica, ampliamente utilizada [HBJ+02, SHG+05, CV08], define unidades de medida de tiempo independientes de la tecnología. En concreto, un *FO4* representa el retardo de un inversor cuya salida alimenta a cuatro copias de

| ***Param.*** | ***Retardo (u.t.)*** |
|---|---|
| $t_{fv}$ | 29 |
| $t_{sync}$ | 74 |
| $t_{fn}$ | 29 |
| $t_{ack}$ | 29 |

Tabla 5.8: Valores para la caracterización del protocolo *handshake* de cuatro fases de la configuración asíncrona.

sí mismo. De esta manera, es posible trasladar un valor en *FO4* a un valor de tiempo de cualquier tecnología objetivo. Esta traducción se realiza dividiendo el valor en *FO4* entre el retardo de un *FO4* en la tecnología objetivo [Ho05].

Aplicando la equivalencia en *FO4*[2], los retardos obtenidos por Cheng se convierten en 0,20 *FO4* y 0,*71* FO4 para $t_{fv}$ y $t_{sync}$ respectivamente.

Por otro lado, según [HMH01], el tiempo de ciclo de un procesador Alpha 21264 a 800 MHz tiene un retardo de 19,2 *FO4*. Como se ha utilizando este referente en las simulaciones para establecer el número de u.t. máximo por ciclo, ocurre entonces la siguiente equivalencia:

$$1000\,u.t. = 19,2\,FO4 \tag{5.4}$$

En consecuencia, los valores de retardo asociados al protocolo se pueden transformar de *FO4* a u.t. utilizando la equivalencia de la expresión 5.4. Así, los valores para $t_{fv}$ y $t_{sync}$ se transforman en 14,54 u.t. y 36,97 u.t. respectivamente. Una vez más, se ha utilizado un enfoque conservador para tratar de no favorecer a la versión asíncrona del procesador. Por tanto, en estas simulaciones se ha duplicado el valor de ambos retardos, y se ha equiparado el tiempo de $t_{ack}$ y de $t_{fn}$ al de $t_{fv}$. Los valores finales de retardo para los parámetros asociados al protocolo se resumen en la Tabla 5.8.

### Simulaciones

Las simulaciones de los bancos de pruebas SPEC2000 para ambas configuraciones muestran mayor rendimiento para la configuración asíncrona. Como se aprecia en

[2]En tecnología de 2 $\mu$m cada *FO4* corresponde a 1 ns.

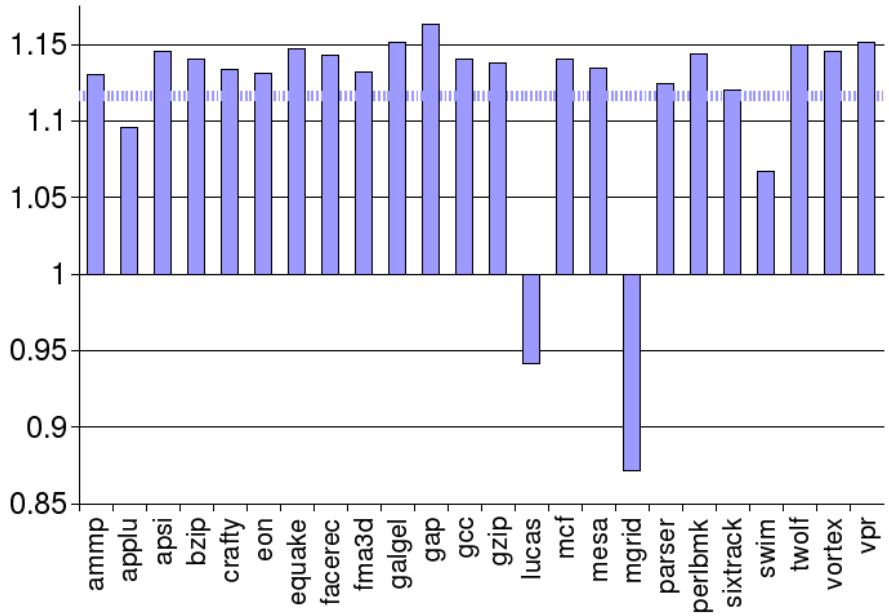


Figura 5.9: *Speedup* de la configuración asíncrona en comparación con la síncrona ejecutando los SPEC2000. La línea punteada indica el *speedup* promedio.

la Figura 5.9, la ganancia (*speedup*) promedio es de 1,12 con respecto a la configuración síncrona. Teniendo en cuenta que la configuración asíncrona presenta poca variabilidad y que las estimaciones tanto en los tiempos de cómputo como en los retardos del protocolo son muy conservadoras, este resultado muestra un importante potencial de mejora para este tipo de microarquitecturas.

Los binarios donde peores resultados se obtienen son *applu*, *lucas*, *mgrid* y *swim*, todos ellos con un *speedup* por debajo de la media. Los cuatro pertenecen al grupo de CFP2000, que engloba a los binarios con mayor componente de operaciones de coma flotante, lo que llevaría pensar que la configuración asíncrona penaliza a este tipo de *benchmarks*. Sin embargo, otros binarios como *apsi*, *equake* ó *galgel*, también pertenecientes al conjunto CFP2000, obtienen un buen rendimiento, por lo que se contradice esa hipótesis.

De los cuatro binarios comentados, los peores resultados los obtienen *lucas* y *mgrid*, donde el rendimiento de la configuración asíncrona es inferior a la síncrona. El binario *lucas* ejecuta comprobaciones de números primos, mientras que *mgrid* es un resolutor de ecuaciones 3D. Para intentar encontrar la causa del ba-

jo rendimiento de la microarquitectura en estos dos casos se utiliza otra de las medidas propuestas en el simulador. Esta medida es la latencia promedio de las instrucciones dentro de la microarquitectura.

La Figura 5.10 muestra cómo estos dos binarios son los únicos en los que la latencia promedio de las instrucciones es mayor que en el caso síncrono. Sin embargo, al examinar las estadísticas sobre latencia por etapas que proporciona *Sim-async*, se encuentra una coincidencia en ambos binarios que no ocurre en el resto. Esta coincidencia consiste en que la latencia promedio que las instrucciones pasan en la *IQ* (cola de instrucciones, ver Apartado 4.2.2), indicada por el contador *sim_lat_IQ_avg*, es mayor en el caso asíncrono que en el síncrono, mientras que en el resto de SPEC la latencia es menor en el caso asíncrono para todas las etapas. Concretamente, en *lucas* la diferencia es del 7,3 % frente a la configuración síncrona, mientras que en *mgrid* es del 19,4 %. Esta estadística indica que las instrucciones de ambos binarios, en promedio, esperaron más tiempo para iniciar *Issue* que en el caso síncrono. Esta espera habitualmente se debe a la ausencia de huecos en las *RSs*, que sólo se liberan en el momento en que la correspondiente *FU* termina de ejecutar instrucciones.

Por tanto, una teoría consistente con todos los indicios anteriores es la presencia de ráfagas de instrucciones de punto flotante sobre una misma *FU*. Si las ráfagas son suficientemente largas, y dado que la latencia de las *FUs* de punto flotante es mayor que el ritmo de la etapa *Issue*, podría suceder que las *RSs* asociadas a la *FU* afectada se colapsasen. Este colapso impediría que nuevas instrucciones del mismo tipo se pudieran decodificar, deteniendo el lanzamiento de instrucciones en la microarquitectura y aumentando así la latencia en la *IQ*.

A pesar de *lucas* y *mgrid*, la Figura 5.10 demuestra que la configuración asíncrona obtiene mejor rendimiento que la síncrona porque consigue reducir el tiempo promedio que las instrucciones pasan en el procesador. De hecho, la menor latencia de las instrucciones provoca que las dependencias se resuelvan antes en la configuración asíncrona. Esto se traduce en una liberación de recursos más rápida que en el caso síncrono. En consecuencia, el lanzamiento de instrucciones puede ser más rápido, incrementando el número total de instrucciones ejecutadas, incluyendo las especulativas. Este hecho también se constata gracias a las estadísticas que recopila *Sim-async*.

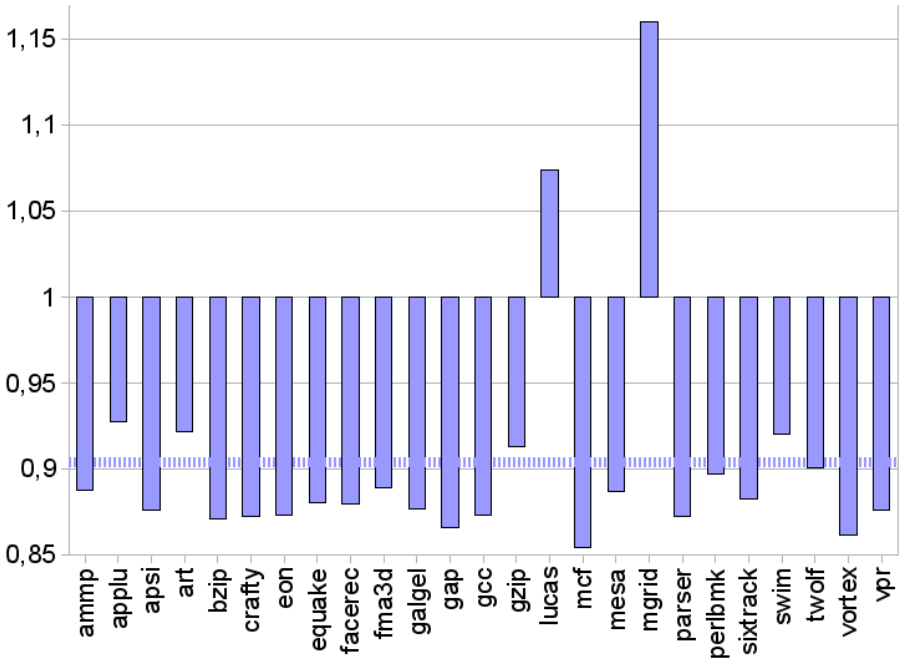


Figura 5.10: Latencia promedio de instrucciones finalizadas para las simulaciones asíncronas de SPEC2000. Valores normalizados con respecto a las simulaciones síncronas. La línea punteada indica el valor promedio.

La Figura 5.11 muestra el número total de instrucciones ejecutadas para los SPEC. Todos los binarios, salvo *parser* y *sixtrack*, ejecutan más instrucciones en la configuración asíncrona. Como se puede apreciar por la escala del eje de ordenadas, la diferencia no es demasiado representativa en ninguno de los binarios. Sin embargo, en la mayoría de los casos se consigue un incremento de *speedup* ejecutando un mayor número total de instrucciones, por lo que se reafirma el buen rendimiento de la configuración asíncrona.

En resumen, en este primer caso práctico se ha presentado una comparación entre el rendimiento de dos configuraciones para el procesador que modela *Sim-async*. La primera configuración corresponde a un enfoque totalmente síncrono, basado en el procesador Alpha 21264. La configuración asíncrona, por su parte, representa un enfoque conservador, puesto que todas las etapas salvo *Exec* presentan tiempos de cómputo constantes, mientras que es en las *FUs* donde se introduce variación. Además, el protocolo de comunicación de cuatro fases añade una penalización adicional a esta configuración. Pese a todo, los resultados muestran un *speedup* promedio de 1,12 a favor de la microarquitectura asíncrona, lo que indica un

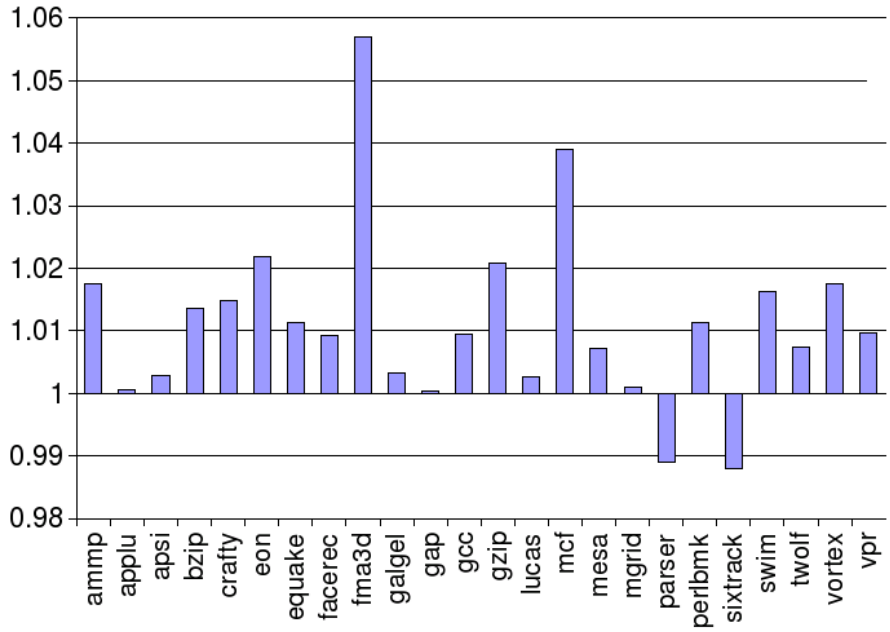


Figura 5.11: Número de instrucciones ejecutadas (incluyendo las especuladas) para las simulaciones de los SPEC2000 en la configuración asíncrona. Los valores están normalizados con respecto a las simulaciones síncronas.

importante potencial de mejora.

### 5.2.2. *PAM* con cache de latencia variable

El grado de abstracción del modelo que aplica *Sim-async*, así como el desacoplamiento que mantiene entre la descripción funcional del procesador y su temporización hacen de este simulador una herramienta muy flexible y aplicable a diversos tipos de simulaciones.

Este segundo caso práctico pretende ilustrar cómo utilizar *Sim-async* para obtener el rendimiento de una microarquitectura donde se ha aplicado una modificación arquitectónica. Esta modificación consiste en transformar el procesador en un *PAM* (*Partially Asynchronous Microprocessor*, microprocesador parcialmente asíncrono) [MABB02]. Concretamente se trabaja bajo un enfoque LAGS (Localmente Asíncrono, Globalmente Síncrono), donde los accesos a la cache de datos de nivel uno (*L1 D-Cache*) se gestionan a través de un recubrimiento (*wrapper*) asíncrono, mientras que el resto del procesador funciona en modo síncrono.

La idea a implementar consiste en capturar los resultados de las operaciones de lectura de memoria lo antes posible. Para ello, el recubrimiento que engloba a la *FU* de lectura de memoria detectará la llegada de los datos desde la memoria de manera asíncrona. En consecuencia, no será necesario esperar al caso peor, correspondiente a la versión síncrona del procesador, sino que la transmisión del dato recibido se podrá adelantar uno ó varios ciclos de reloj. De esta manera se tratará de aprovechar la variabilidad del tiempo de acceso de la cache sin necesidad de utilizar enfoques segmentados.

La variabilidad en la latencia de las memorias cache es un hecho probado en la literatura. Hay dos motivos fundamentales por los que se produce este comportamiento:

- Relación entre el retardo y la topología de conexiones: el acoplamiento capacitivo afecta enormemente al retardo de las conexiones largas que viajan paralelas. De hecho, a mayor tasa de conmutación, mayor perjuicio sobre el retardo de este tipo de conexiones, habituales en las memorias cache [GI05].
- Distancia a los datos: el retardo en el acceso a los datos más cercanos al puerto de la memoria es menor que el retardo en el acceso a los datos físicamente más alejados. En consecuencia, los tiempos de acceso de la cache se convierten en un continuo de latencias, más que en un único valor de latencia [KBK02].

La Figura 5.12 presenta un esquema con la implementación de esta técnica. En la figura se muestra el recubrimiento asíncrono que engloba a la *FU* de lectura de memoria y a la cache de datos, denotada como *D-Cache*. Este módulo se comunica con el entorno síncrono a través de dos canales: *I/F in* e *I/F out*. El canal denominado *I/F in* se encarga de enviar peticiones desde el entorno síncrono a la *FU* de lectura de memoria. *I/F out*, por su parte, envía los resultados recibidos de la memoria al *FF* que se encuentra a la salida de la *FU*.

Para modelar una microarquitectura LAGS, se configuró globalmente *Sim-async* bajo temporización síncrona, mientras que la *FU* de acceso a memoria se configuró particularmente como asíncrona. En consecuencia, esta configuración requiere una caracterización de tiempo de cómputo para la *FU*, además de la descripción temporal del protocolo de comunicación con el entorno síncrono.

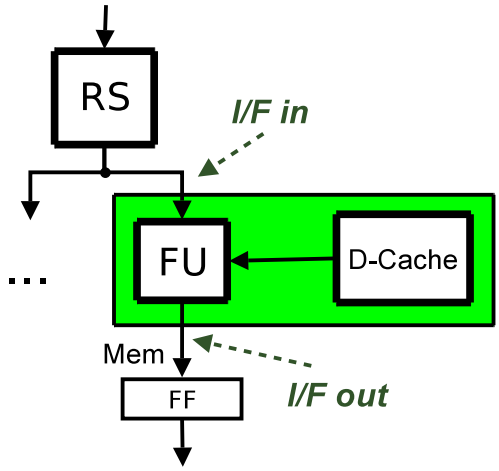


Figura 5.12: Recubrimiento asíncrono para el acceso desde la *FU* de lectura de memoria a la cache de datos (*D-Cache*). La comunicación entre el recubrimiento asíncrono y el entorno síncrono se realiza a través de dos canales: *I/F in* para la entrada e *I/F out* para la salida.

### Caracterización de la *FU* de acceso a memoria

La caracterización de la latencia variable de la cache se obtuvo a partir del trabajo publicado en [OMCK+07]. El objetivo de este artículo es presentar una memoria segmentada asíncrona que se aproveche de la variabilidad en la latencia de la memoria. Para ello modela el comportamiento de una cache de 32 Kb de cuatro vías, 64 bytes por bloque y una latencia máxima de cuatro ciclos. Ese estudio muestra, para todos los SPEC2000, la distribución de latencias que presenta la cache. En otras palabras, se indica el porcentaje de ocasiones en que se emplea uno, dos, tres ó cuatro ciclos en el acceso a la cache para cada uno de los SPEC. Como resultado se obtienen distintas distribuciones de latencias, dividiendo cada una de ellas en cuatro clases, una por cada número de ciclos. La Figura 5.13 muestra esas distribuciones para cada uno de los binarios SPEC2000.

En consecuencia, la *FU* de acceso a memoria se caracteriza en *Sim-async* aplicando, en la simulación de cada binario, la función de distribución que determina su comportamiento conforme a la Figura 5.13.

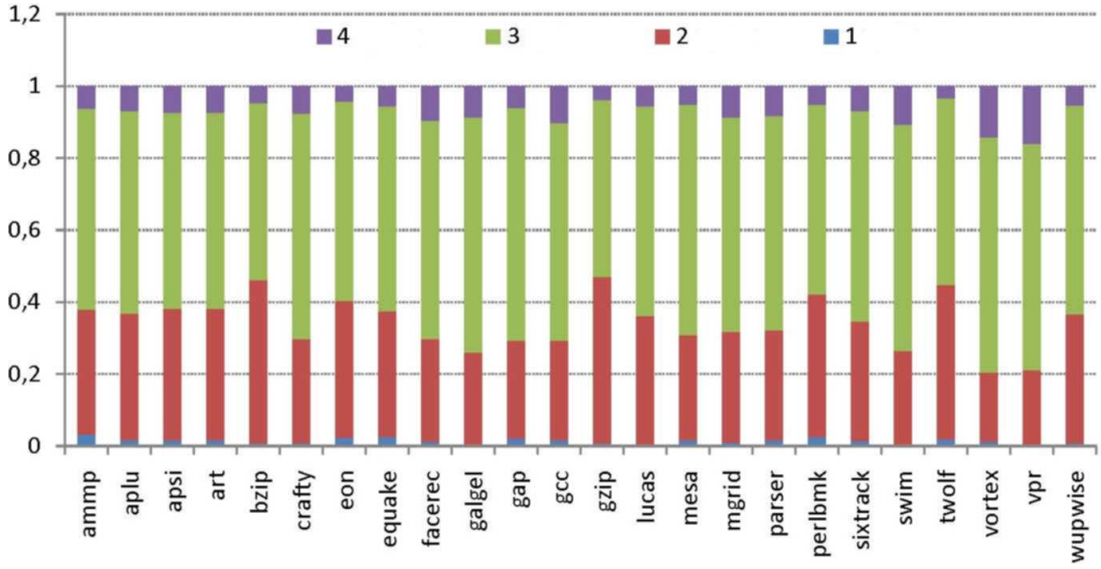


Figura 5.13: Distribución de la latencia en el acceso a la cache de datos (*D-Cache*) para cada uno de los SPEC2000 evaluado en [OMCK+07]. Los distintos valores de latencia (4, 3, 2 ó 1 ciclo) se representan en colores diferentes.

### Protocolo de comunicación

La comunicación entre el recubrimiento de la *FU* de lectura de memoria y el entorno síncrono se realiza a través de los canales de comunicación ya mencionados, *I/F in* para la entrada e *I/F out* para la salida.

El canal *I/F in* se ha modelado en base a dos casos posibles: caso síncrono y caso asíncrono. En el primero, la *FU* comenzaría la operación de memoria en el siguiente ciclo de reloj, por lo que no es necesario aplicar ningún protocolo en este canal. En el segundo caso, la comunicación sería asíncrona, de modo que se podría comenzar la operación en cualquier instante de tiempo, sin necesidad de esperar al siguiente flanco de reloj. En consecuencia sí sería necesario aplicar el protocolo en el canal *I/F in*. Concretamente se ha optado por aplicar un protocolo *handshake* de cuatro fases. La diferencia entre ambas alternativas se estudia en las simulaciones que se describen más adelante.

*I/F out*, por su parte, hace de interfaz entre un emisor asíncrono y un receptor síncrono. Por tanto podría haber colisiones entre datos ó metaestabilidades en la recepción. Para evitarlo, el simulador modela la solución propuesta en [SM97],

añadiendo un ciclo de penalización en el dominio del receptor cuando la distancia entre la llegada del dato y flanco de reloj del receptor esté por debajo del tiempo de *setup* del registro.

Los valores de tiempo correspondientes a los parámetros del protocolo de comunicación son los mismos que se han descrito en el apartado anterior, y aparecen resumidos en la Tabla 5.8.

### Simulaciones

Los resultados experimentales se componen de dos conjuntos de simulaciones que, como en los anteriores casos presentados, comparten la configuración arquitectónica que se ha indicado en la Tabla 5.1. Por un lado se lanzaron simulaciones considerando que el canal de comunicación *I/F in* se modela como asíncrono. Por otro lado, se ejecutaron simulaciones modelando el canal *I/F in* como síncrono. Para evaluar el rendimiento de ambas configuraciones se ha calculado el *speedup* con respecto a la configuración totalmente síncrona descrita en el apartado anterior. Como muestra la Figura 5.14, la mayoría de los SPEC2000 obtienen un rendimiento mejor que el caso totalmente síncrono para ambas configuraciones de *I/F in*.

En concreto, las simulaciones con *I/F in* síncrono obtienen un *speedup* promedio de 1,18, mientras que las simulaciones con *I/F in* asíncrono alcanzan una ganancia promedio de 1,22. Estos resultados confirman que acelerar el acceso a memoria en esta microarquitectura permite obtener una mejora importante en el rendimiento global del procesador. De hecho, pese a la inclusión de penalizaciones debidas al protocolo de comunicación, se verifica que el mejor rendimiento lo ofrece la configuración donde el canal *I/F in* se modela como asíncrono. En este modelo, las operaciones sobre la memoria se realizan tan pronto como sea posible, sin necesidad de esperar al flanco de reloj.

Los peores resultados en ambos casos los obtienen los binarios *applu*, *lucas* y *swim*, cuyo rendimiento queda por debajo del obtenido en el caso totalmente síncrono. Estos tres *benchmarks* se engloban dentro del conjunto CFP2000 de punto flotante. Sin embargo, al igual que en el experimento presentado en el Apartado 5.2.1, esta característica no es el motivo de su bajo *speedup*. De hecho,

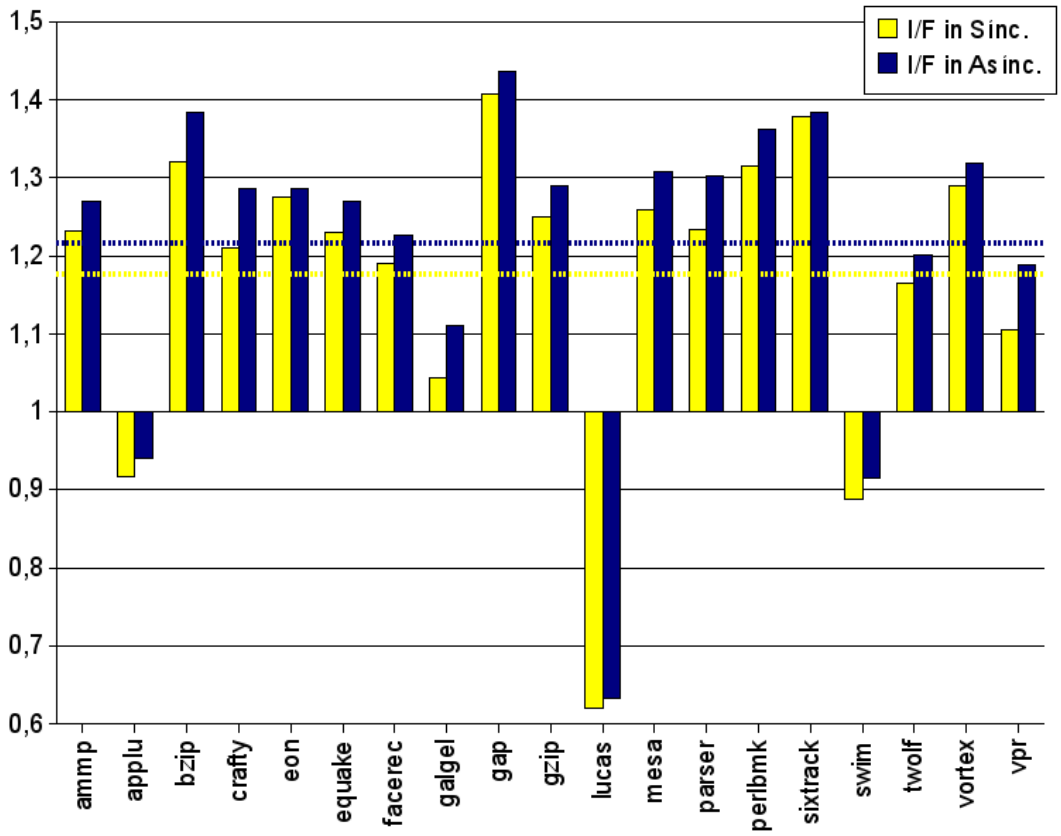


Figura 5.14: *Speedup* de las configuraciones *PAM* en comparación con la totalmente síncrona ejecutando los SPEC2000. En ambos casos se indica el *speedup* promedio con una línea punteada.

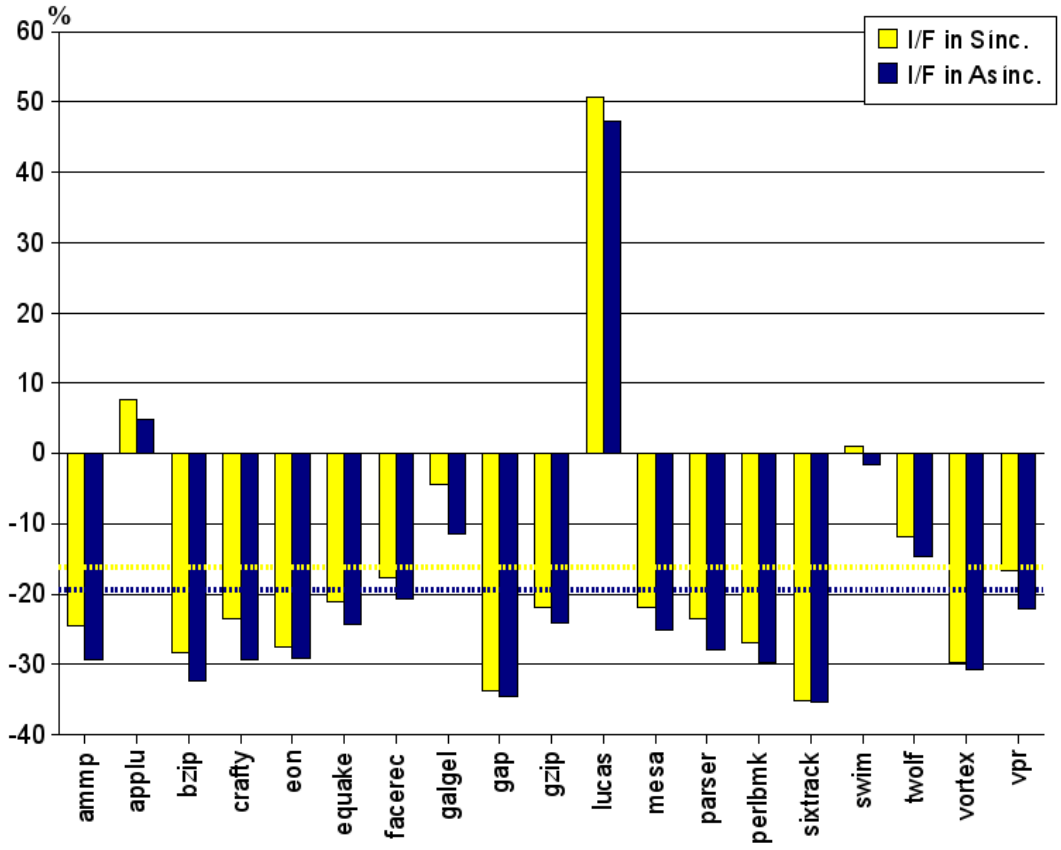


Figura 5.15: Diferencia porcentual entre la latencia promedio de las instrucciones en las simulaciones de SPEC2000 en las dos configuraciones *PAM* y el caso totalmente síncrono. En ambos casos se muestra el valor promedio con una línea punteada.

otros binarios de ese mismo conjunto como *ammp*, *equake* ó *sixtrack* obtienen importantes mejoras en el rendimiento de las configuraciones *PAM*.

Para encontrar una respuesta a este fenómeno, el análisis se complementa con una comparación entre la latencia promedio de las instrucciones ejecutadas y los valores obtenidos para las simulaciones totalmente síncronas descritas en el apartado anterior. La Figura 5.15 muestra las estadísticas obtenidas para los SPEC2000 en los dos escenarios modelados para el canal *I/F in*.

El enfoque síncrono para *I/F in* obtiene una latencia que, en promedio para todos los SPEC, se reduce en un 16,26 % con respecto al caso totalmente síncrono. La mayor reducción la consiguen aplicaciones como *gap* ó *sixtrack*, con un 33,84 % y un 35,18 % respectivamente. Los mejores resultados, no obstante, los obtiene la

configuración en la que se modela el canal *I/F in* como asíncrono. En promedio, la latencia de las instrucciones se reduce en un 19,48 % con respecto al caso totalmente síncrono. De nuevo las aplicaciones *gap* y *sixtrack* obtienen las mayores diferencias, cifradas en 34,6 % y 35,25 % respectivamente.

Los binarios *applu*, *lucas* y *swim*, que obtuvieron los peores resultados de *speedup*, alcanzan valores de latencia promedio superiores al caso totalmente síncrono, como muestra la Figura 5.15. La excepción en estos tres binarios se presenta en *swim* modelando *I/F in* como asíncrono, que queda ligeramente por debajo de la latencia síncrona.

El desglose por etapas de las latencias de las instrucciones completa el análisis de estos resultados. En la Figura 5.16 se presenta, de manera porcentual, la diferencia entre la latencia promedio de las instrucciones en la etapa *Fetch* en comparación con el caso totalmente síncrono. Para obtener esta latencia, el simulador contabiliza el tiempo promedio que pasan las instrucciones en la *IQ* (dato *sim_lat_IQ_avg*). De esta manera, cuanto mayor sea este valor, más tiempo habrán pasado las instrucciones esperando a comenzar la etapa *Issue*. Como se aprecia en la gráfica, todos los SPEC salvo los ya mencionados *applu*, *lucas* y *swim* presentan una reducción en este dato, siguiendo un patrón parecido al de la gráfica de *speedup* (Figura 5.14), aunque a la inversa. Por tanto, se puede concluir que el ritmo de lanzamiento de instrucciones de las configuraciones *PAM* es superior, en promedio, al caso totalmente síncrono y análogo a la ganancia en rendimiento de los *benchmarks*. El análisis del resto de etapas permitirá explicar el motivo de este comportamiento.

La Figura 5.17 muestra, porcentualmente, la diferencia entre la latencia promedio de las instrucciones en la etapa *Issue* en comparación con el caso totalmente síncrono. El simulador obtiene este dato de latencia contabilizando el tiempo que pasa desde que las instrucciones llegan a las *RSs*, al final de la etapa *Issue*, hasta que se inicia la etapa *Exec* (dato *sim_lat_ER_avg*). Las estadísticas muestran cómo la reducción, en ambos casos *PAM*, es similar en todos los SPEC salvo *lucas*. En promedio, la reducción se cifra en un 34,89 % para el caso donde *I/F in* se modela como síncrono, y en un 35,14 % para *I/F in* asíncrono. Esto indica que, aunque la etapa *Issue* comience antes que el caso síncrono (según los resultados de *Fetch*), el tiempo que las instrucciones pasan en las *RSs* es menor, lo que significa

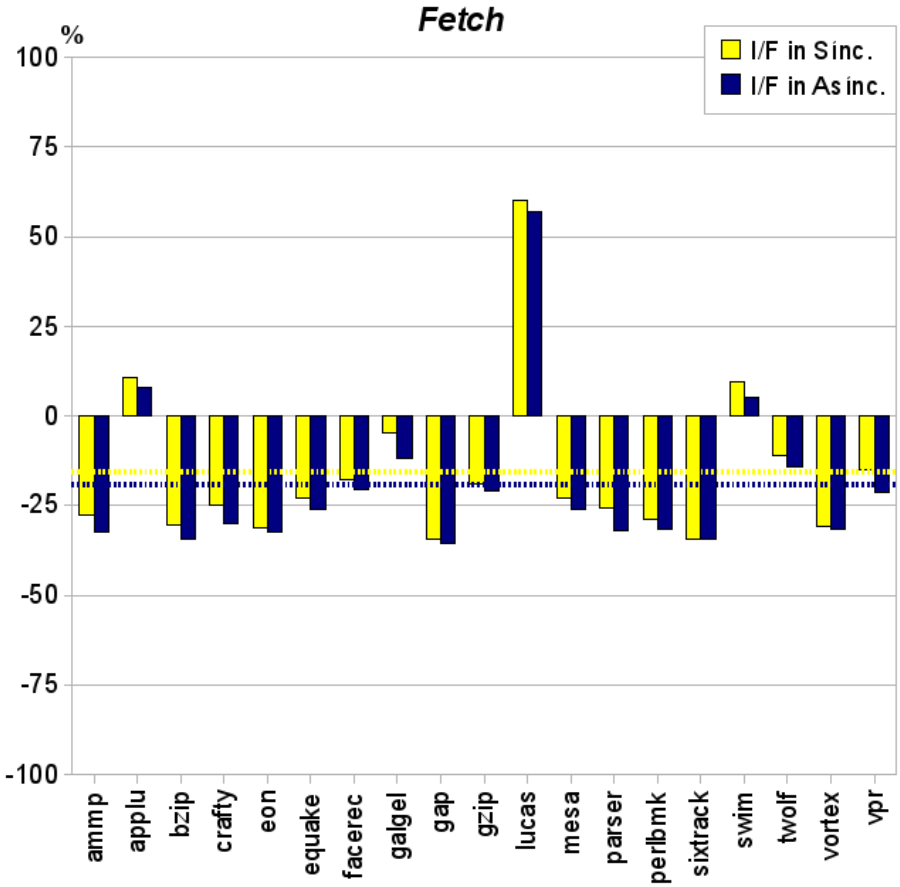


Figura 5.16: Diferencia porcentual, en las simulaciones de SPEC2000, entre la latencia promedio de las instrucciones en las dos configuraciones *PAM* y el caso totalmente síncrono para la etapa *Fetch*. Los valores promedio se muestran con líneas punteadas.

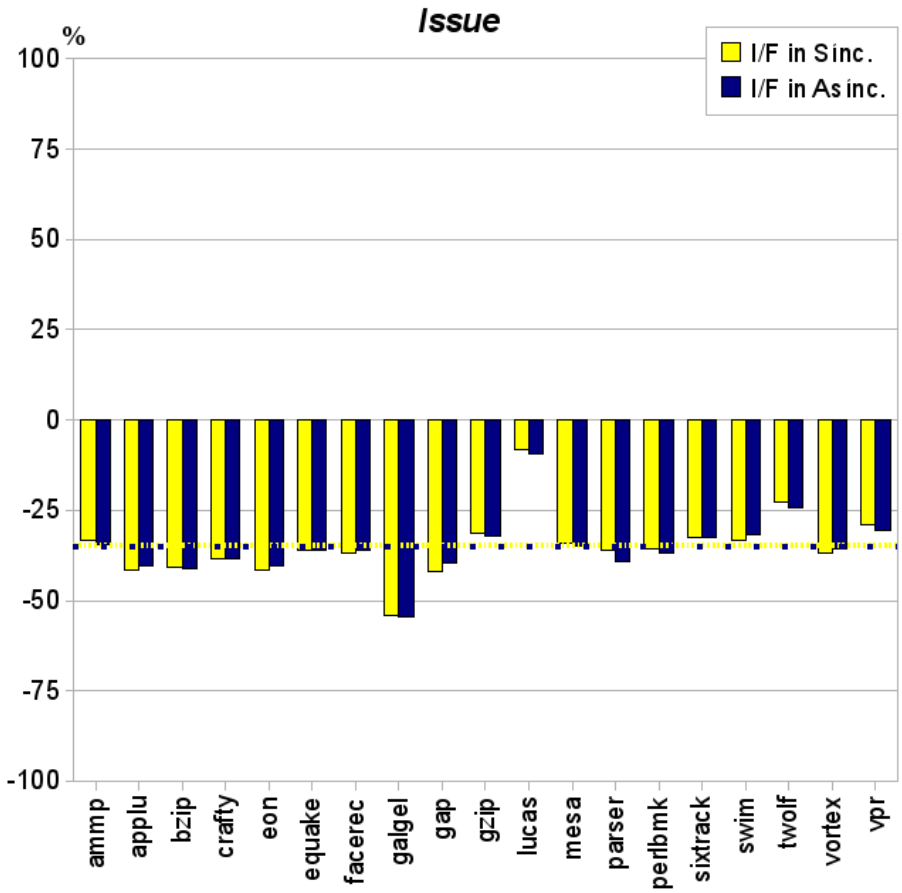


Figura 5.17: Diferencia porcentual, en las simulaciones de SPEC2000, entre la latencia promedio de las instrucciones en las dos configuraciones *PAM* y el caso totalmente síncrono para la etapa *Issue*. Los valores promedio se muestran con líneas punteadas.

que la etapa *Exec* adelanta su ejecución en los casos *PAM*. Este comportamiento tiene dos causas fundamentales. Por un lado, el tiempo que se emplea en la etapa *Exec* es menor, lo que en las configuraciones *PAM* sólo ocurre para las operaciones de lectura de memoria. Por otro lado, al reducir el tiempo de estas instrucciones, las dependencias de datos que producen se resuelven antes, provocando que las instrucciones pendientes esperen menos tiempo para entrar en *Exec*. Esta última causa parece la más acertada para explicar esta uniformidad en el comportamiento de todos los binarios en *Issue*.

En la Figura 5.18 se muestra de nuevo la comparación de latencias promedio, en este caso para la etapa *Exec*. Para obtener esta estadística, el simulador mide el

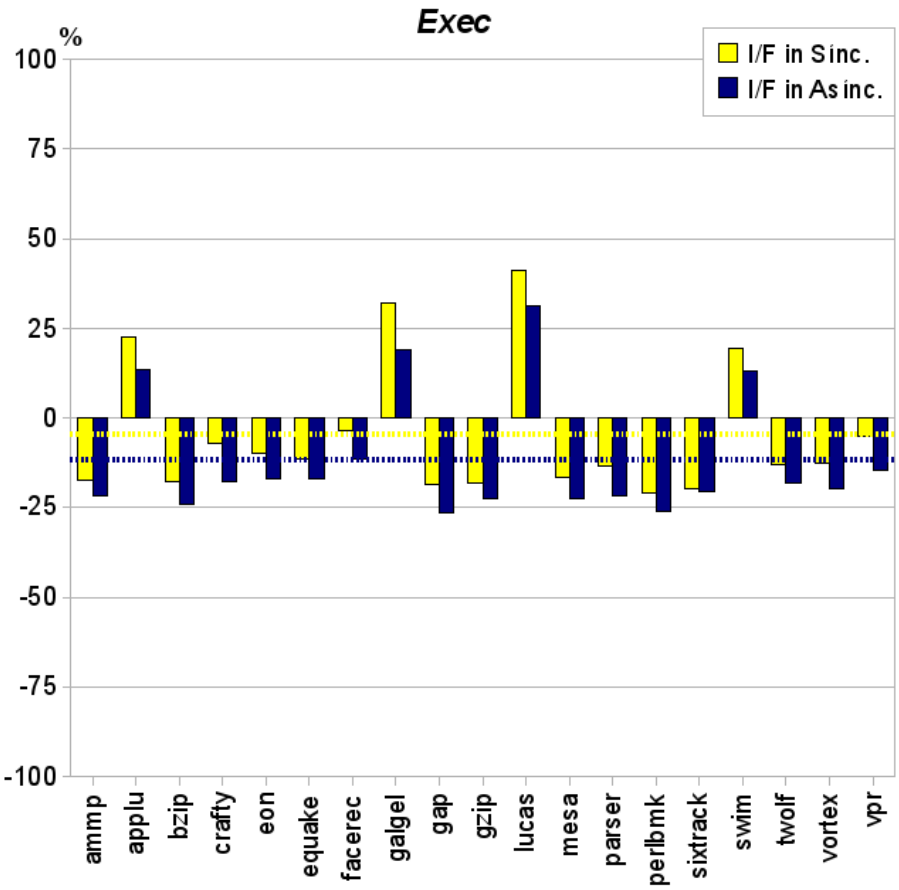


Figura 5.18: Diferencia porcentual, en las simulaciones de SPEC2000, entre la latencia promedio de las instrucciones en las dos configuraciones *PAM* y el caso totalmente síncrono para la etapa *Exec*. Los valores promedio se muestran con líneas punteadas.

tiempo que las instrucciones pasan en las *FUs* (dato *sim_lat_FU_avg*). Como se aprecia en la figura, la reducción en la latencia no es tan uniforme en esta etapa. El motivo fundamental es que todas las *FUs* salvo la de acceso a memoria tienen la misma latencia que el caso síncrono. Por tanto, se confirma que el motivo principal por el que se produce la reducción en *Issue* es la resolución temprana de dependencias. Por otro lado, los binarios que peor comportamiento muestran en *Exec* son *applu*, *lucas* y *swim*, ya comentados anteriormente, y *galgel*, cuyos resultados de *speedup* no son demasiado favorables.

Una estadística similar se contabiliza para *Write-back*, calculando el tiempo que las instrucciones pasan en los *FFs* a la espera de la difusión de resultados por el

*CDB* (dato *sim_lat_RegFU_avg*). En este caso, *Write-back* difiere de la tendencia del resto de etapas. Según se aprecia en la Figura 5.19, en todos los binarios ocurre, para ambos casos *PAM*, que la latencia de las instrucciones en esta etapa es mayor que en el caso totalmente síncrono. La razón más plausible para este hecho es el aumento en la presión sobre esta etapa. Dicho de otro modo, al resolver las dependencias entre instrucciones con más antelación, aumenta el número de instrucciones que puede ejecutar la etapa *Exec* simultáneamente. En consecuencia, habrá más resultados en los *FFs* al mismo tiempo, lo que provoca que, dada la limitación de anchura de la etapa *Write-back*, las instrucciones esperen más tiempo en los *FFs*. Una vez más, los peores resultados se encuentran en *applu*, *lucas* y *swim*. Como posibles soluciones arquitectónicas se plantean, por un lado, mantener la temporización síncrona de la etapa replicando el número de *CDBs* y la lógica asociada. Por otro lado, se podría implementar una estrategia asíncrona para esta etapa, realizando sus operaciones bajo demanda, sin necesidad de esperar por el flanco de reloj. Esta última propuesta formará parte de futuros trabajos relacionados con el simulador.

Finalmente, la estadística para la etapa *Commit* se presenta en la Figura 5.20. El simulador obtiene esta información midiendo el tiempo que transcurren las instrucciones en el *ROB* desde que finaliza la etapa *Write-back* hasta el fin de *Commit* (dato *sim_lat_ROB_avg*). En promedio, la reducción de la latencia en esta etapa se sitúa por encima del 50 % para ambas configuraciones *PAM*, aunque los binarios no muestran un comportamiento uniforme. La justificación para la reducción en la latencia de esta etapa se apoya de nuevo en la resolución temprana de dependencias. Una instrucción reduce su tiempo en el *ROB* si las instrucciones que la preceden no se detienen a causa de alguna dependencia. Por tanto, para la mayoría de SPEC la influencia de la reducción de la latencia de la *FU* de acceso a memoria permite agilizar la etapa *Commit*. Es destacable el caso de *mesa*, binario perteneciente al grupo CFP2000, donde la latencia promedio en *Commit* sufre un aumento superior al 100 % en las configuraciones *PAM*, pese a que obtiene un *speedup* particular de 1,26 y 1,31 para las configuraciones con *I/F in* síncrono e *I/F in* asíncrono respectivamente.

Tras este análisis por etapas, se retoma el análisis de los binarios *applu*, *lucas* y *swim*, cuyos valores de *speedup* resultaron inferiores al caso síncrono. Las figuras

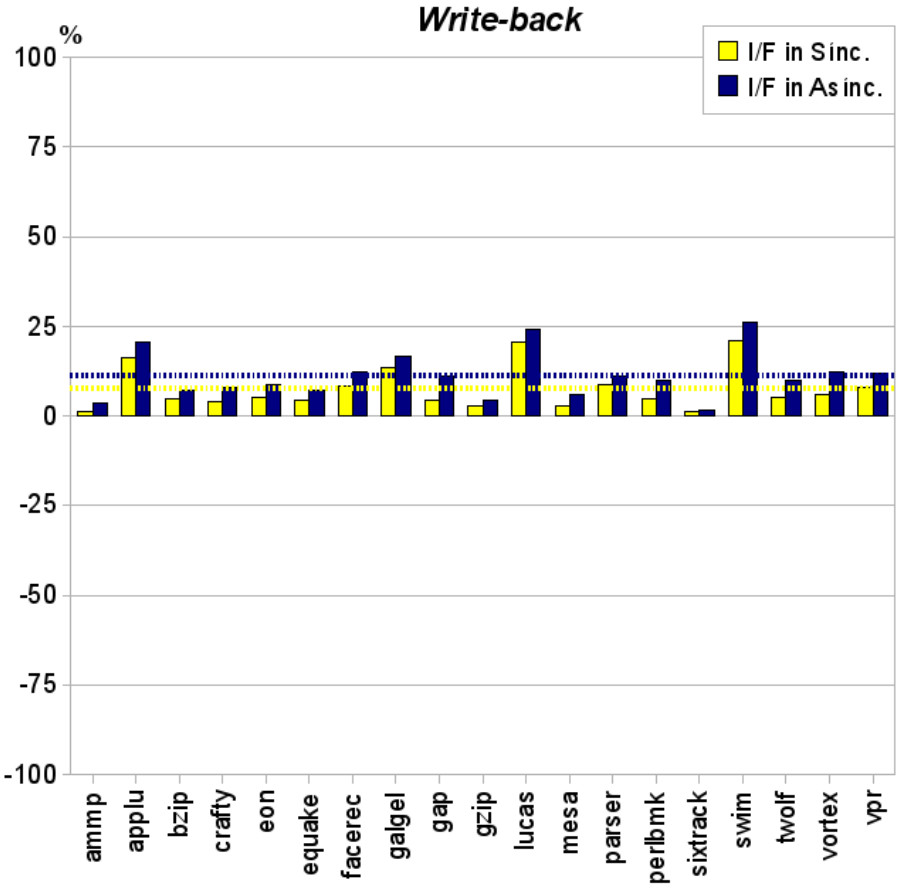


Figura 5.19: Diferencia porcentual, en las simulaciones de SPEC2000, entre la latencia promedio de las instrucciones en las dos configuraciones *PAM* y el caso totalmente síncrono para la etapa *Write-back*. Los valores promedio se muestran con líneas punteadas.

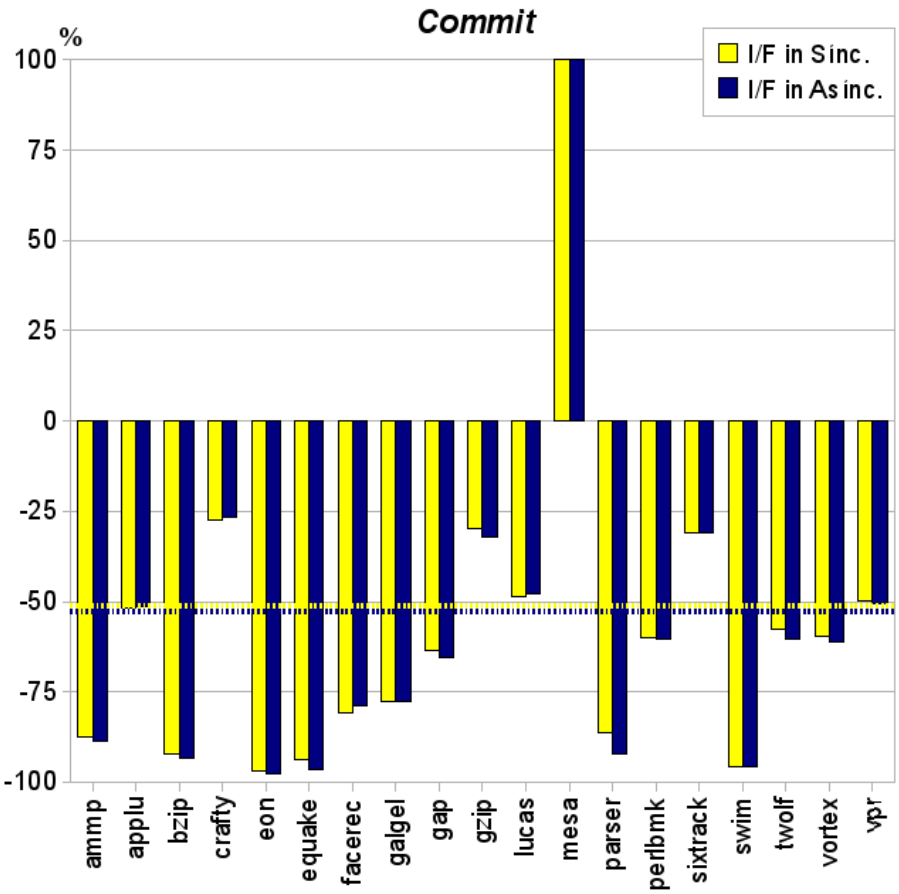


Figura 5.20: Diferencia porcentual, en las simulaciones de SPEC2000, entre la latencia promedio de las instrucciones en las dos configuraciones *PAM* y el caso totalmente síncrono para la etapa *Commit*. Los valores promedio se muestran con líneas punteadas.

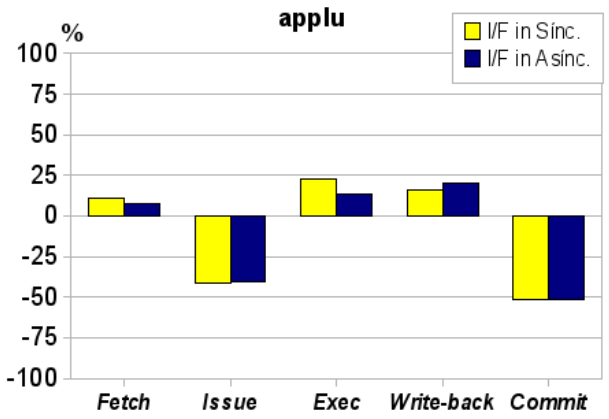


Figura 5.21: Diferencia porcentual, por etapas, entre la latencia promedio de *applu* en las dos configuraciones *PAM* y el caso totalmente síncrono.

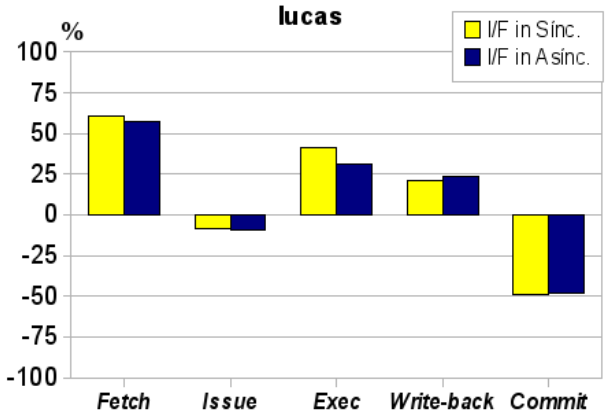


Figura 5.22: Diferencia porcentual, por etapas, entre la latencia promedio de *lucas* en las dos configuraciones *PAM* y el caso totalmente síncrono.

5.21, 5.22 y 5.23 muestran las latencias promedio de las instrucciones para *applu*, *lucas* y *swim* respectivamente, desglosadas por etapas. Pese a que en los tres binarios se producen reducciones de latencia importantes para *Issue* y *Commit*, y vistos los datos obtenidos por el resto de SPEC, se puede concluir que la mayor influencia en la latencia global la tiene la resolución temprana de dependencias, cuyo efecto se aprecia fundamentalmente en la latencia promedio de la etapa *Fetch*. En concreto, *applu*, *lucas* y *swim* son los únicos tres binarios en los que se produce un incremento de la latencia en *Fetch*.

La resolución temprana de dependencias también se menciona en el caso práctico examinado en el Apartado 5.2.1. En él se apreciaba cómo al reducir la laten-

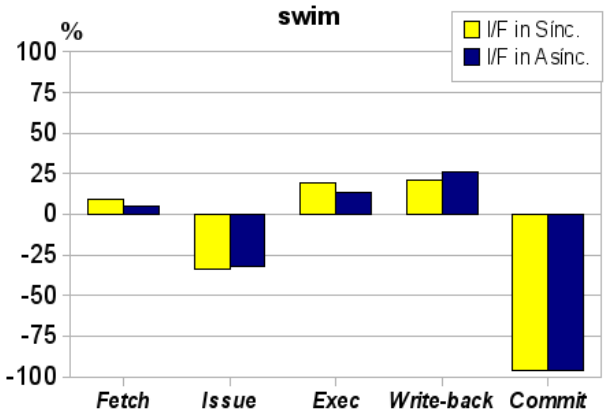


Figura 5.23: Diferencia porcentual, por etapas, entre la latencia promedio de *swim* en las dos configuraciones *PAM* y el caso totalmente síncrono.

cia promedio de las instrucciones la microarquitectura resolvía las dependencias antes que en el caso síncrono, aumentando así el número de instrucciones ejecutadas, incluyendo las especuladas. En la microarquitectura *PAM* de nuevo se verifica este comportamiento. La latencia promedio de las instrucciones se reduce considerablemente gracias a la reducción de latencia de la *FU* de acceso a memoria. Como consecuencia, las dependencias sobre las instrucciones de lectura de memoria se resuelven antes que el caso totalmente síncrono, provocando que la profundidad de la instrucciones especuladas sea mayor. Como muestra la Figura 5.24, en la mayoría de los SPEC se cumple este comportamiento.

Dado que la mayoría de los SPEC consiguen aumentar el número de instrucciones ejecutadas, llama la atención el caso del binario *sixtrack*, que ejecuta un 5 % menos de instrucciones que en el caso síncrono pese a conseguir un buen dato de *speedup* (superior a 1,35 en ambas configuraciones *PAM*). Si el rendimiento ha aumentado y la latencia promedio es menor, como atestigua la Figura 5.15, las dependencias entre instrucciones se habrán resuelto antes que en el caso síncrono. Por tanto, si el número de instrucciones ejecutadas no ha aumentado se debe a que no se han podido lanzar en la etapa *Issue*, situación que se produce al aumentar la presión sobre una misma *RS*. En consecuencia, cabe pensar que, debido a la reducción de la latencia de las instrucciones en las configuraciones *PAM*, los caminos especulativos donde la predicción no es correcta producen presión en alguna *RS*, provocando la parada de *Issue* y reduciendo así el número total de

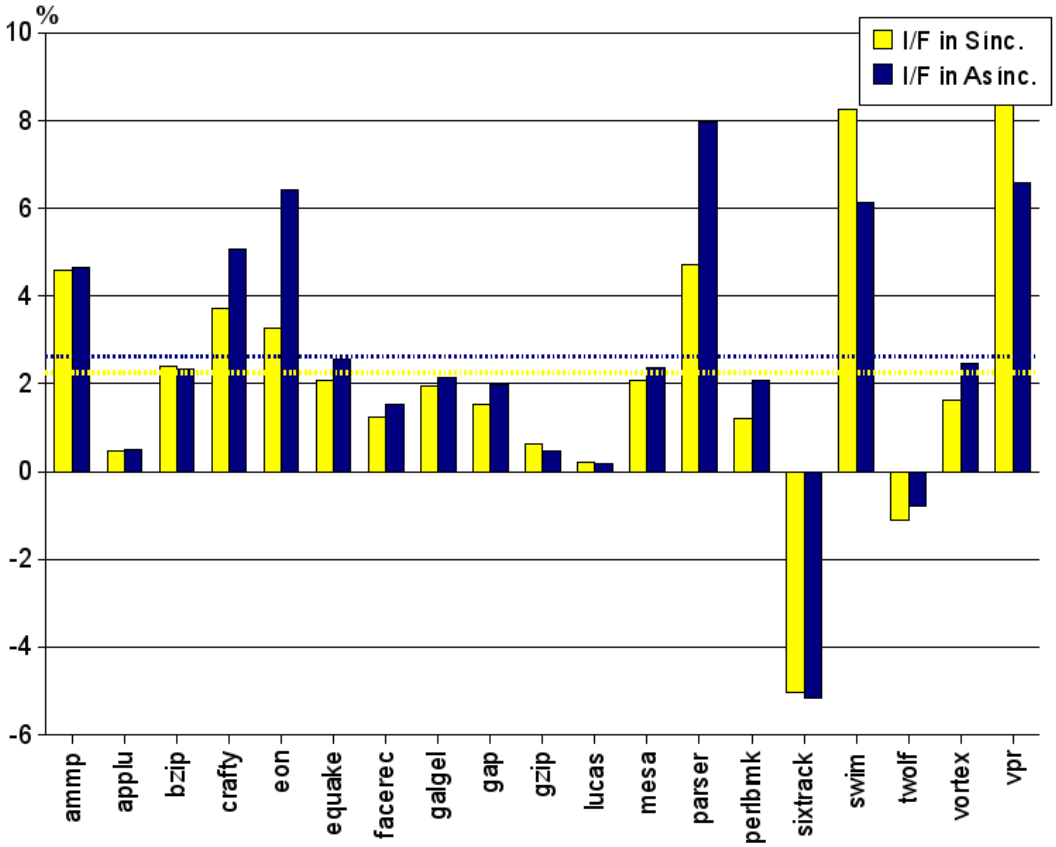


Figura 5.24: Diferencia porcentual entre el número de instrucciones ejecutadas (incluyendo las especuladas) en las simulaciones de SPEC2000 en las dos caracterizaciones *PAM* y el caso totalmente síncrono. En ambos casos se muestra el valor promedio con una línea punteada.

instrucciones ejecutadas.

En resumen, se ha presentado un caso práctico de utilización de *Sim-async* donde se compara la configuración totalmente síncrona del procesador con una configuración *PAM* donde la única parte asíncrona engloba al acceso a la cache de datos. Los resultados de las simulaciones demuestran que esta propuesta es capaz de hacer frente al cuello de botella que suele representar el acceso a la cache de datos. Este enfoque, basado en un recubrimiento asíncrono, es capaz de aprovechar la latencia variable que presenta la memoria ocultando a la vez las penalizaciones debidas al protocolo de comunicación entre la parte asíncrona y el entorno síncrono. Además, esta propuesta es independiente del tipo de memoria, y no requiere de segmentación alguna en la cache. Los resultados, pese a estar basados en caracterizaciones de memoria de grano muy grueso (cuatro clases ó latencias posibles en cada distribución), ofrecen una mejora apreciable frente al caso síncrono debida, tanto a la reducción de la latencia en el acceso a memoria, como a la resolución temprana de las dependencias sobre ese tipo instrucciones. Resultados tan prometedores motivan el seguir trabajando en esta línea.

## 5.3. Resumen

Los resultados experimentales que se presentan en este capítulo verifican dos importantes objetivos perseguidos en esta tesis. En primero de ellos es la validación de *Sim-async*, confirmada gracias a las simulaciones que se analizan en el Apartado 5.1. En ese apartado se verifica que el simulador modela una microarquitectura que ejecuta correctamente el repertorio de instrucciones del procesador Alpha, utilizando configuraciones de temporización tanto síncrona como asíncrona. Esta característica se alcanza gracias a que *Sim-async* modela de manera independiente la funcionalidad de la microarquitectura y su temporización.

El segundo de los objetivos es la aplicación del simulador al modelado de nuevas técnicas arquitectónicas. Para ello se han presentado dos experimentos diferentes. En el Apartado 5.2.1 se ha presentado una comparación entre el rendimiento de una configuración síncrona basada en el procesador Alpha 21264 y una configuración asíncrona con protocolo de comunicación de cuatro fases donde todas las etapas salvo *Exec* presentan tiempos de cómputo constantes, mientras que es

en las *FUs* donde se introduce variación. Los resultados indican un rendimiento superior para la configuración asíncrona, cifrado en un *speedup* promedio de 1,12.

Por otro lado, en el Apartado 5.2.2 se ilustra cómo *Sim-async* permite que la configuración del tiempo de cómputo y del protocolo de comunicación de sus etapas se realice de manera individualizada. Para ello, se evalúa el rendimiento de una configuración parcialmente asíncrona donde la *FU* de lectura de memoria cache se encapsula en un recubrimiento asíncrono. De esta manera es posible aprovechar la latencia variable que obtienen los distintos accesos a memoria, reduciendo el tiempo de cómputo de esta unidad y consiguiendo, en consecuencia, un aumento general en el rendimiento de la microarquitectura. Las simulaciones muestran valores de *speedup* promedio de 1,18 y 1,22 para las dos opciones de configuración estudiadas para el recubrimiento asíncrono.

En todos los experimentos *Sim-async* se configuró sin necesidad de recompilar el código fuente. En total se analizan en este capítulo más de cien simulaciones de binarios pertenecientes al conjunto SPEC2000. En promedio, se modela arquitectónicamente la ejecución de 550 millones de instrucciones en cada *benchmark*, incluyendo las instrucciones especuladas.

# Capítulo 6

# Conclusiones

En el primer capítulo se definió como objetivo principal de esta tesis el *estudio e implementación de una herramienta capaz de evaluar el rendimiento de un procesador superescalar asíncrono de propósito general a través de la simulación, a nivel arquitectónico, del comportamiento dinámico de una microarquitectura donde los tiempos de cómputo de sus componentes sean variables.*

Para conseguir este objetivo principal, se enunciaron los siguientes objetivos secundarios, descritos en la introducción de este documento:

1. Estudio de una metodología de modelado del tiempo de cómputo de un circuito asíncrono que, siendo computacionalmente eficiente, represente fielmente la variabilidad de la latencia de cada módulo del circuito.

2. Modelado, a través de un simulador arquitectónico, de un procesador superescalar asíncrono que disponga de características avanzadas como predicción de saltos y ejecución especulativa de instrucciones.

3. Estudio de caracterizaciones genéricas para el tiempo de cómputo que permitan al simulador modelar el comportamiento asíncrono en la microarquitectura simulada.

4. Integración en el simulador de la caracterización del tiempo de cómputo de cada uno de los módulos asíncronos que componen la microarquitectura del procesador.

5. Integración en el simulador del modelado de varios protocolos de comunicación, configurables individualmente para cada una de las comunicaciones entre módulos de la microarquitectura.

6. Estudio y generación de medidas acerca del rendimiento de la microarquitectura asíncrona, adicionales a las estadísticas de ejecución comunes en los simuladores arquitectónicos (*i.e.*, número de instrucciones ejecutadas, número de saltos, estadísticas sobre el predictor de saltos, etc...).

7. Estudio del modelado de *sistemas parcialmente síncronos*.

8. Estudio de otras aplicaciones del simulador relacionadas con la variabilidad en los tiempos de cómputo.

Todos estos objetivos se han alcanzado satisfactoriamente, salvo el último, que forma parte del trabajo futuro. A continuación se detallan las conclusiones y aportaciones de la tesis que corroboran estos objetivos. Seguidamente se expondrán las ideas que guían la investigación actual y el trabajo futuro. Para finalizar, se incluyen las referencias a las publicaciones relacionadas con la tesis, así como los proyectos que han financiado este trabajo.

## 6.1. Principales aportaciones del trabajo

Las conclusiones y aportaciones de esta tesis se presentan agrupadas en cuatro tópicos distintos:

- Modelado del tiempo de cómputo.
- Estudio de microarquitecturas asíncronas.
- Estudio e implementación de un simulador arquitectónico para microarquitecturas asíncronas.
- Estudio del rendimiento de configuraciones asíncronas.

### Modelado del tiempo de cómputo

- El modelado de la variabilidad del tiempo de cómputo a través del almacenamiento de todas las posibles latencias de un circuito es una técnica que consume demasiados recursos. Por un lado, puede demandar gran cantidad de espacio de almacenamiento. Por otro lado, los algoritmos de asignación de tiempos a datos de entrada pueden tener una complejidad elevada.

- Las funciones de distribución de la probabilidad (FDPs) permiten asociar un valor de probabilidad a un retardo, por lo que son capaces de describir las variaciones en el tiempo de cómputo de un circuito.

- En este trabajo se ha propuesto y verificado una metodología de caracterización de tiempos de cómputo variables basada en la utilización de FDPs. Esta metodología se divide en cuatro pasos:

  - Obtención de la muestra de retardos del circuito asíncrono a caracterizar, validando la calidad (similitud) de la muestra verificando que su tamaño sea mayor o igual que el indicado en la Fórmula (3.4).
  - Establecimiento de la granularidad del histograma, que condiciona la precisión de la FDP.
  - Generación del histograma a partir del conjunto de retardos de la muestra.
  - Construcción de la FDP calculando las frecuencias relativas para cada clase del histograma.

- El método de caracterización garantiza, gracias a la Fórmula (3.4), que la FDP que describe el tiempo de cómputo del circuito representa fielmente el comportamiento real del circuito según el margen de error elegido.

- La caracterización a través de FDP de la temporización de un circuito síncrono es un caso particular de la caracterización de un circuito asíncrono.

- La caracterización del tiempo de cómputo a través de FDP permite trabajar con jerarquías, por lo que se puede aplicar tanto a circuitos completos como a partes ó módulos de un circuito complejo.

- Las FDP permiten describir las variaciones en el tiempo de cómputo, pero para modelar la latencia de un circuito asíncrono es necesario modelar también el resto de retardos que la componen.

- En esta tesis se ha creado un modelo genérico para caracterizar la latencia de un circuito ($L$). Según el modelo, la latencia se obtiene como resultado de la agregación de tres factores, como se indica en la siguiente ecuación:

$$L = t_c + e_r + t_p \tag{6.1}$$

  - $t_c$ es el *tiempo de cómputo* o retardo empleado por la lógica combinacional en generar un resultado.
  - $e_r$ es el tiempo de *espera por receptor*, correspondiente al periodo de tiempo que transcurre hasta que el receptor se encuentra disponible. Dada la variabilidad en los tiempos de cómputo de un sistema asíncrono, $e_r$ no se conoce *a priori*, puesto que depende del estado del sistema en cada instante de tiempo. Por tanto, para hallar este retardo se requiere una simulación detallada del sistema.
  - $t_p$ es el *tiempo de protocolo*, correspondiente al retardo asociado al protocolo de comunicación entre el emisor y el receptor del dato.

- A partir del modelo genérico de latencia se han estudiado modelos particulares de latencia que permiten caracterizar las siguientes temporizaciones:

  - Síncrona
  - GALS (globalmente asíncrona, localmente síncrona)
  - Asíncrona bajo el modelo de retardos acotados
  - Asíncrona bajo el modelo insensible a retardos

**Estudio de microarquitecturas asíncronas**

- Se ha propuesto una microarquitectura superescalar con predicción de saltos y ejecución dinámica de instrucciones. La microarquitectura, detallada en el Apartado 4.2.2, se divide en doce dominios de sincronización, que se agrupan

en cinco etapas: *Fetch*, *Issue*, *Exec*, *Write-back* y *Commit*. A su vez, la etapa *Exec* se divide en siete tipos de unidades funcionales (*FUs*), formando cada una de ellas su propio dominio de sincronización independiente:

  - *Integer* : operaciones de suma y resta de enteros. En la arquitectura se incluyen dos instancias de este tipo de *FU*.
  - *IntMul* : multiplicación de enteros.
  - *FPAdd* : suma y resta de números en punto flotante.
  - *FPMult* : multiplicación de números en punto flotante.
  - *FPDiv* : operaciones de división y raíz cuadrada de números en punto flotante.
  - *Addr* : cálculo de direcciones para las operaciones sobre la memoria de datos.
  - *Mem* : operación de lectura de la memoria de datos (*load*).

- La microarquitectura ejecuta el repertorio de instrucciones del procesador Alpha 21264, lo que permite la ejecución de *benchmarks* estándar para la evaluación de su rendimiento.

**Estudio e implementación de un simulador arquitectónico para microarquitecturas asíncronas**

- La variabilidad en los tiempos de cómputo de un sistema asíncrono, así como las penalizaciones debidas al protocolo, dependen del estado de la microarquitectura en cada instante de tiempo. Esta información, dependiente del estado del sistema en cada momento, sólo se obtiene al realizar un modelado arquitectónico.

- Se ha desarrollado *Sim-async*, un nuevo simulador arquitectónico basado en SimpleScalar que modela la microarquitectura superescalar descrita en el apartado anterior. La validación de *Sim-async* se presenta en el Apartado 5.1.

- En el simulador se separa el modelado de la funcionalidad de la descripción temporal del circuito. Esta separación permite realizar simulaciones de la misma microarquitectura bajo diversas suposiciones de temporización tanto síncronas como asíncronas.

- El modelado de la funcionalidad del procesador se realiza a nivel arquitectónico, describiendo individualmente las etapas y *FUs* del procesador.

- La latencia de cada módulo se modela según la caracterización genérica para la latencia descrita anteriormente.

    - El tiempo de cómputo de cada etapa y *FU* se caracteriza a través de su propia función de distribución.
    - Se ha aplicado una variante del *algoritmo de selección proporcional de la ruleta* para obtener el tiempo de cómputo de cada dato a partir de la función de distribución correspondiente.
    - El simulador modela dos protocolos de comunicación, incluyendo sus retardos asociados: el protocolo *handshake* de cuatro fases y el protocolo *handshake* de dos fases. Su descripción se muestra en el Apartado 4.2.3.

- El simulador calcula dinámicamente las intervenciones de cada uno de los componentes del procesador a lo largo de la ejecución de las simulaciones teniendo en cuenta el estado del sistema. Para ello, se ha implementado en *Sim-async* un motor de gestión de eventos capaz de modelar la línea temporal en la ejecución de las simulaciones.

    - Los eventos se asocian a las distintas etapas y *FUs* de la microarquitectura, almacenando toda la información relacionada con la operación a la que corresponden, además del instante de tiempo en que cada uno se debe procesar.
    - En caso de encontrar eventos con igual valor de tiempo, el motor de ejecución los procesa según la estrategia de SimpleScalar, comenzando por los relacionados con las etapas finales del procesador y terminando por los correspondientes a las etapas iniciales.

  - En consecuencia, la simulación respeta la *ley de causa-efecto*, de manera que una operación sólo empezará cuando haya terminado la operación que la provocó.

- Todos los parámetros de configuración del simulador se describen a través de esquemas XML, desde las funciones de distribución hasta el tipo y configuración del protocolo de comunicación. Esta característica permite validar automáticamente la corrección de una configuración dada.

- Se han incorporado a *Sim-async* un total de treinta y siete estadísticas distintas, que se muestran en la Tabla 4.4. Estas medidas se agrupan en cinco categorías:

  - Medidas sobre cantidad de instrucciones: tienen el prefijo *sim_num_* y cuentan el número de instrucciones que cumplen alguna condición.
  - Medidas sobre tiempos totales: con el prefijo *sim_lat_*, acumulan la cantidad de tiempo total que las instrucciones emplean en alguna operación.
  - Medidas sobre tiempos promedio: tienen el sufijo *_avg*, y se calculan dividiendo el número de instrucciones que cumplen cada condición, entre el tiempo total empleado en ella.
  - Medidas sobre número de ejecuciones: con el sufijo *_times*, indican el número de veces que una etapa ó *FU* del procesador ha ejecutado.
  - Otras medidas: normalmente se refieren a instantes de tiempo en que ocurre alguna situación. El ejemplo principal es *ult_commit*.

**Estudio del rendimiento de configuraciones asíncronas**

- En el Apartado 5.2.1 se describe la comparación del rendimiento entre dos configuraciones de la microarquitectura descrita en *Sim-async*. La primera configuración corresponde a un enfoque totalmente síncrono, mientras que la segunda configuración representa un enfoque asíncrono conservador. En este última configuración todas las etapas salvo *Exec* presentan tiempos de cómputo constantes, mientras que es en las *FUs* donde se introduce

variación. Además, el protocolo de comunicación de cuatro fases añade una penalización adicional a esta configuración.

  - Los resultados muestran un *speedup* promedio de 1,12 a favor de la microarquitectura asíncrona, lo que indica un importante potencial de mejora si se continua aplicando técnicas asíncronas al resto de etapas ó *FUs*.

- En el Apartado 5.2.2 se ha presentado un experimento donde se utiliza *Sim-async* en la evaluación del rendimiento de un *PAM* (*Partially Asynchronous Microprocessors*, microprocesador parcialmente asíncrono) a partir de la microarquitectura propuesta. Concretamente se trata del modelo de un sistema tipo LAGS (Localmente Asíncrono, Globalmente Síncrono), donde el acceso a la cache de datos se realiza utilizando un recubrimiento asíncrono.

  - Los resultados obtenidos con *Sim-async* muestran un *speedup* promedio de 1,18 y de 1,22 para la ejecución de los SPEC2000. Estos valores de *speedup* corresponden, respectivamente, a la configuración donde la lectura de memoria empieza de manera síncrona, con el flanco de reloj, y a la configuración donde la lectura comienza de manera asíncrona, en cuanto la memoria está libre. En consecuencia, se obtiene una significativa mejora en el rendimiento del procesador que implementa esta mejora.
  - La utilización de un recubrimiento asíncrono en el acceso a memoria no requiere segmentación, y es independiente del tipo y características de latencia de la memoria cache.
  - Los resultados muestran que, para los SPEC2000, la reducción en la latencia del acceso a memoria incrementa el rendimiento de la microarquitectura, aumentando además el número de instrucciones ejecutadas (incluyendo las especuladas) en el procesador.
  - El análisis de las latencias promedio de las instrucciones en cada una de las etapas lleva a concluir que al reducir la latencia de las operaciones de lectura de memoria no sólo se reduce el tiempo de la etapa *Exec*,

sino que se consigue resolver las dependencias de datos con mayor antelación. La consecuencia principal es un aumento de la presión tanto en la etapa de *Write-back* como en *Issue*.

## 6.2. Trabajo futuro

En esta sección se muestran las líneas de investigación que componen tanto el trabajo actual como el futuro.

- Optimización en la ejecución de simulaciones.
  - Uno de los objetivos actuales consiste en reducir el tiempo de simulación de *Sim-async*. Para ello se está trabajando tanto en la optimización del código fuente, como en los algoritmos de procesamiento de eventos del motor de ejecución.
- Estudio del modelado de sistemas parcialmente síncronos.
  - Arquitecturas tipo GALS (globalmente asíncronas, localmente síncronas) serán evaluadas en futuros trabajos con *Sim-async*.
  - La mejora en rendimiento que se muestra en la configuración *PAM* descrita en el Apartado 5.2.2 sirve de motivación para futuros estudios de este tipo de procesadores. En posteriores experimentos se aplicarán estas técnicas a otras etapas, e incluso a *FUs* como las de enteros, cuya tasa de actividad suele ser muy alta.
- Estudio de otras aplicaciones del simulador relacionadas con la variabilidad en los tiempos de cómputo.
  - En futuros trabajos se estudiarán los cambios en el comportamiento de un circuito debidos a su envejecimiento ó a las variaciones en voltaje y temperatura. Estos estudios permitirán utilizar *Sim-async* para evaluar el rendimiento de una misma microarquitectura en distintas situaciones ambientales.

- Se extenderá la funcionalidad de *Sim-async* de modo que sea capaz de simular algoritmos o técnicas adaptativas como el escalado dinámico de voltaje. Esta técnica se podría simular a través de la selección de distintas funciones de distribución en tiempo de simulación.

## 6.3. Publicaciones

Los trabajos realizados durante el desarrollo de esta tesis se han recogido en distintas publicaciones científicas [CMG+08, CGL+06b, CGL+06a, CGL+04, CGL03]. El estudio donde se presenta el modelado con funciones de distribución y se compara la microarquitectura asíncrona con su equivalente síncrona se recoge en el artículo:

- J. M. Colmenar, N. Morón, O. Garnica, J. Lanchares, J. I. Hidalgo. *Modelling asynchronous systems using probability distribution functions*. Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008). Pags. 3-11. ISSN: 1066-6192. IEEE Computer Society. 13-15 de febrero de 2008. Toulouse (Francia). Artículo premiado dentro del *Best papers track*.

Las primeras versiones de *Sim-async*, una vez finalizado su estudio y desarrollo, dieron lugar a las siguientes dos publicaciones:

- J. M. Colmenar, O. Garnica, J. Lanchares, J. I. Hidalgo, G. Miñana, S. López. *Sim-async: an architectural simulator for asynchronous processor modeling using distribution functions*. Proceedings of the 12th International Euro-Par Conference (Euro-Par 2006 Parallel Processing), Lecture Notes in Computer Science, Volumen 4128/2006. Pags. 495-505. ISSN: 0302-9743 (impreso), 1611-3349 (on-line). Springer Berlin / Heidelberg. Septiembre de 2006. Dresden (Alemania).

- J. M. Colmenar, O. Garnica, J. Lanchares, J. I. Hidalgo, G. Miñana, S. López. *Comparing the performance of a 64-bit fully-asynchronous superscalar processor versus its synchronous counterpart*. Proceedings of the 9th

Euromicro Conference on Digital System Design (DSD 2006). Pags. 423-432. ISBN: 0-7695-2609-8. IEEE Computer Society. Septiembre de 2006. Dubrovnik (Croacia).

Los estudios previos acerca del comportamiento de la latencia de circuitos asíncronos complejos se publicaron en:

- J. M. Colmenar, O. Garnica, S. López, J. I. Hidalgo, J. Lanchares, R. Hermida. *Empirical Characterization of the Latency of Long Asynchronous Pipelines with Data-Dependent Module Delays*. Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2004). Pags. 311-321. ISSN: 1066-6192. IEEE Computer Society. Febrero de 2004. La Coruña (España).
- J. M. Colmenar, O. Garnica, J. I. Hidalgo, J. Lanchares. *Técnica de estimación del rendimiento de pipelines asíncronos*. XIV Jornadas de Paralelismo. Pags. 397-402. ISBN: 84-89315-34-5. Septiembre de 2003. Leganés (España).

De manera colateral, la experiencia tanto en el área de circuitos asíncronos, como en el estudio y simulación de procesadores de propósito general, ha permitido la colaboración en diversas publicaciones. Entre ellas se destacan las siguientes [MHL+07, MHG+06, LGH+05]:

- G. Miñana, J. I. Hidalgo, J. Lanchares, J. M. Colmenar, O. Garnica, S. López. *Reducing power of functional units in high-performance processors by checking instruction codes and resizing adders.* IET Computers & Digital Techniques. Volumen 1, Issue 2. Pags. 113-119. ISSN: 1751-8601. Marzo de 2007.
- G. Miñana, J. I. Hidalgo, O. Garnica, J. Lanchares, J. M. Colmenar, S. López. *A technique to reduce static and dynamic power of functional units in high-performance processor*. Proceedings of 33th International Workshop on Power and Timing Modelling, Optimization and Simulation (PATMOS 06), Lecture Notes in Computer Science. Volumen 4148/2006. Pags. 514-523. ISSN: 0302-9743 (impreso) 1611-3349 (on-line). Springer Berlin / Heidelberg. Septiembre 2006. Montpellier (Francia).

- S. López O. Garnica, J. I. Hidalgo, J. Lanchares, J. M. Colmenar, G. Miñana. *Study of the communication channels in a globally asynchronous locally synchronous simultaneous multithreading architecture*. Proceedings of the 2nd International Conference on Automation, Control and Instrumentation (IADAT 2005). ISBN: 84-933971-8-0. 5-7 de julio de 2005, Valencia (España).

## 6.4. Proyectos de investigación

Esta tesis se ha hecho realidad gracias a la financiación de distintos proyectos de investigación:

- Proyecto CSD00C-07-20811 del Ministerio de Educación y Ciencia, Consolider Ingenio 2010, titulado *Supercomputación y e-Ciencia*. Investigador Principal: Francisco Tirado Fernández. 2008-2012.
- Proyecto TIN2005-5619 de la Comisión Interministerial de Ciencia y Tecnología (CICYT), titulado *Arquitectura HW/SW para sistemas de alto rendimiento*. Investigador Principal: Francisco Tirado Fernández. Diciembre de 2005 a diciembre de 2008.
- Proyecto TIC2002/0750 de la Comisión Interministerial de Ciencia y Tecnología (CICYT), titulado *Tecnologías HW/SW para sistemas de alto rendimiento*. Investigador Principal: Francisco Tirado Fernández. Diciembre de 2002 a noviembre de 2005.

# Apéndice A

# Diseño de circuitos asíncronos

El diseño de circuitos asíncronos debe tener en cuenta diversos aspectos fundamentales. El primero de ellos es el *modelo de retardos* bajo el que funciona el circuito. A partir del modelo de retardos se realiza una descripción funcional del circuito que llevará a una implementación concreta. Una vez escogido el modelo de retardos y decidido el tipo de circuito asíncrono a construir, el siguiente paso consiste en evaluar la necesidad de contar con algún tipo de codificación de datos o *protocolo de comunicación* entre elementos del circuito.

## A.1. Modelos de retardos

Al tratar de definir el comportamiento de un circuito síncrono, es posible, gracias a la señal de reloj, simplificar su modo de funcionamiento como una sucesión de cómputos de igual retardo. Este retardo es el mismo que la duración del periodo de reloj, por lo que, en cierta medida, se puede obviar la temporización y modelar directamente la funcionalidad del circuito síncrono.

En los circuitos asíncronos no es posible obviar la temporización, puesto que la ausencia de señal de reloj implica que el funcionamiento del circuito a lo largo del tiempo no esté ceñido a un patrón concreto. En consecuencia, es necesario disponer de un *modelo de retardos* o conjunto de suposiciones acerca de su temporización.

En la actualidad existen multitud de diseños asíncronos que difieren entre sí por las suposiciones que aplican tanto a la temporización como al funcionamiento del

circuito. De hecho, aunque todo circuito tiene un retardo inherente, la interpretación de su retardo es distinta si se trata de circuitos síncronos o de circuitos asíncronos.

El modelo de retardos, por tanto, define la dinámica de funcionamiento de un circuito a nivel de puertas lógicas, es decir, modela su comportamiento a lo largo del tiempo para cualquier valor de entrada.

Existen dos modelos de retardos principales que describen los retardos de un circuito a nivel de puertas lógicas [DN97, Hau95]. Estos son el modelo de retardos acotados y el modelo de retardos no acotados, cuyos detalles se explican en los siguientes apartados.

### A.1.1. Modelo de retardos acotados (*bounded delays model*)

Este modelo asume que los retardos de los componentes y de las conexiones del circuito son conocidos o, al menos, existe una cota superior conocida. La ventaja principal de los circuitos construidos según este modelo está en la posibilidad de utilizar herramientas y metodologías relacionadas con circuitos síncronos.

Los inconvenientes de los circuitos construidos bajo el modelo de retardos acotados son, en cierta medida, similares a los inconvenientes de los circuitos síncronos. La utilización de cotas o límites en los retardos implica que el diseño dependa de las suposiciones de trabajo en el caso peor. Al igual que ocurre en los circuitos síncronos, es necesario incluir márgenes de seguridad para proteger la correcta funcionalidad del circuito, con la consiguiente pérdida de rendimiento. Como consecuencia, disminuye la robustez frente a cambios en temperatura ó voltaje de alimentación y frente a variaciones en el proceso de fabricación.

### A.1.2. Modelo de retardos no acotados (*unbounded delays model*):

Este modelo asume que los retardos de los componentes y de las conexiones del circuito no se conocen. Por tanto, el circuito se debe diseñar para computar correctamente ante cualquier situación de temporización.

El modelo de retardos no acotados define una situación típicamente asíncrona donde no existe suposición alguna acerca de los tiempos de cómputo o propagación de datos en el circuito. Así, la principal ventaja de este tipo de circuitos es la robustez ante variaciones en temperatura, voltaje de alimentación o proceso de fabricación.

Sin embargo, conseguir que el funcionamiento del circuito sea independiente de la temporización complica el diseño del circuito. Por ello, no siempre se considera que los retardos son desconocidos. De esta manera, dependiendo de la rigidez de las suposiciones aplicadas, será necesario contar con la ayuda de lógica para la detección de fin de cómputo, codificación en las señales y protocolos de comunicación de datos.

Los circuitos diseñados bajo este modelo funcionan siempre bajo el *modo entrada/salida*. En este modo de trabajo no existen suposiciones acerca del comportamiento del entorno del circuito, por lo que las señales de entrada pueden cambiar su valor en cualquier instante de tiempo.

## A.2. Clasificación de circuitos asíncronos

Existen diversas taxonomías de circuitos asíncronos atendiendo a distintos factores. Entre ellos, quizá el más importante es el modelo de retardos bajo el que se construye el circuito. A continuación se muestra una clasificación según esta característica.

### A.2.1. Tipos de circuitos asíncronos bajo el modelo de retardos acotados

Los circuitos construidos bajo el modelo de retardos acotados se suelen excluir de las principales clasificaciones que aparecen en la literatura [Spa01, DN97, Hau95]. El motivo de esta exclusión proviene de la similitud de estos diseños con los circuitos síncronos. Sin embargo, la ausencia de señal global de reloj mantiene su estatus asíncrono por lo que estos circuitos deben ser considerados en un repaso como el que se ofrece en esta sección.

Dentro del modelo de retardos acotados se pueden considerar distintos tipos de circuitos asíncronos, diferenciados por su modo de funcionamiento:

- Circuitos de modo fundamental o circuitos Huffman: desarrollados por D. A. Huffman en la década de los años cincuenta del siglo XX [Huf54], se basan en un modo de funcionamiento llamado *modo fundamental*. El modo fundamental supone que el circuito recorre estados donde todas sus entradas, salidas y señales internas permanecen estables. En un estado estable como el definido, el entorno que rodea al circuito sólo tiene permitido cambiar una de las señales de entrada. A continuación, el entorno debe esperar a que el circuito cambie de estado y todas las señales se estabilicen de nuevo antes de computar el siguiente dato. Puesto que las señales internas son desconocidas para el entorno, se debe calcular el retardo más largo que el circuito emplea en cambiar de estado y estabilizar sus señales. Así, el entorno del circuito está obligado a mantener inalteradas las señales de entrada durante ese intervalo de tiempo. El esquema general de un circuito Huffman se presenta en la Figura A.1.

  El diseño de circuitos Huffman resulta complejo, puesto que es habitual que varias señales de entrada cambien simultáneamente en los circuitos actuales, situación que viola el modo fundamental de trabajo. Además, el rendimiento de este tipo de circuitos se ve perjudicado por la restricción de cambiar los valores de las señales de uno en uno que debe respetar el entorno del circuito.

- Circuitos de modo ráfaga (*burst-mode circuits*): este tipo de circuitos tratan de generalizar el modo fundamental utilizado por los circuitos Huffman, acercándose aún más al diseño de circuitos síncronos. Los circuitos de modo ráfaga [Now93] se definen utilizando máquinas de estados finitos estándar donde las transiciones entre estados se llevan a cabo debido a cambios en un conjunto no vacío de entradas (ráfaga de entrada, *input burst*). Cada transición lleva asociado un conjunto de valores de salida (ráfaga de salida, *output burst*).

  Al igual que ocurre en los circuitos síncronos, para modificar el estado sólo deben ocurrir cambios en aquellas ráfagas asociadas a las transiciones que parten del estado actual. El orden de los cambios en las entradas de una

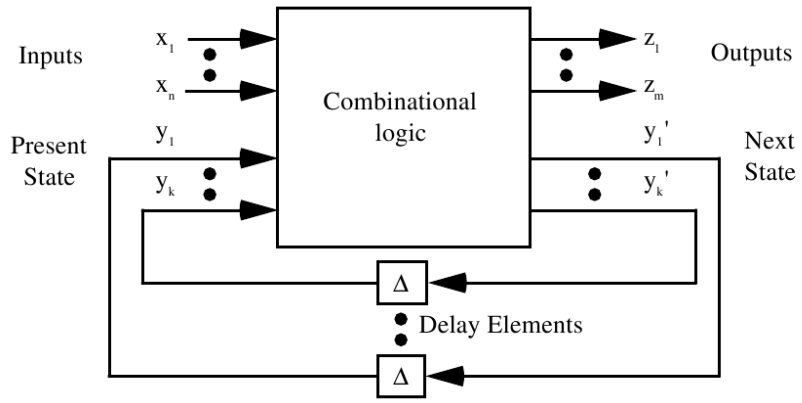


Figura A.1: Esquema general de un circuito secuencial tipo Huffman. Las líneas de retardo aseguran que la salida del circuito alcanza un valor estable antes de que el nuevo valor de las señales de estado siguiente realimente al circuito.

ráfaga no es relevante. Sin embargo sí existe la restricción de que el cambio de estado no se produzca hasta que todas las entradas de la ráfaga asociada a la transición hayan tomado los valores indicados. En ese instante, el circuito establece los nuevos valores de las señales de salida y realiza el cambio de estado. No se permiten cambios en las entradas hasta que el circuito haya reaccionado ante la ráfaga de entrada previa.

En consecuencia, los circuitos de modo ráfaga también utilizan las suposiciones del modo fundamental, pero sólo entre diferentes ráfagas de entrada. Otra de las restricciones de este tipo de circuitos es que una ráfaga de entrada no puede ser subconjunto de ninguna ráfaga de entrada en aquellas transiciones que parten de un mismo estado. Así, la máquina de estados puede reaccionar sin ambigüedades. En la Figura A.2 se muestra un ejemplo de máquina de estados finita que describe el comportamiento de un circuito de modo ráfaga.

La implementación de este tipo de circuitos incluye un reloj local para cada una de las máquinas de estados definidas. La señal de reloj local controla los registros que guardan la información sobre el estado actual, mientras que los valores de salida son generados por lógica combinacional, directamente conectada a las señales de entrada. La utilización de señales de reloj permite utilizar técnicas de diseño síncronas, pero existen riesgos asociados tanto a

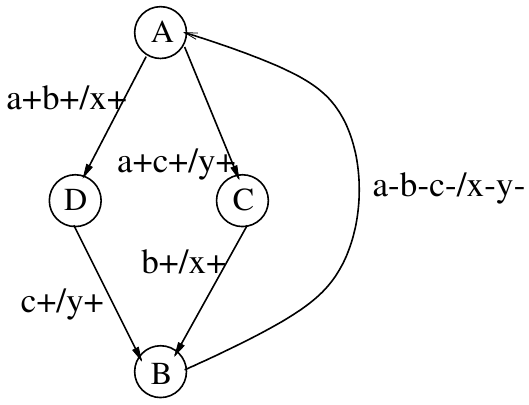


Figura A.2: Máquina de estados finita que especifica el comportamiento de un circuito de modo ráfaga. Las etiquetas de cada transición entre estados tienen el formato *ráfaga de entrada / ráfaga de salida* . El signo "+" indica que la señal toma valor "1", mientras que el signo "-" indica valor "0".

la generación de salidas como a la transición al estado siguiente [ND92].

### A.2.2. Tipos de circuitos asíncronos bajo el modelo de retardos no acotados

En la literatura es habitual encontrar clasificaciones de circuitos asíncronos donde sólo se incluyen circuitos construidos bajo el modelo de retardos no acotados [Spa01, DN97]. Estas clasificaciones son similares a la que se presenta a continuación, donde los tipos de circuitos se detallan de mayor a menor nivel de incertidumbre sobre los retardos:

- Circuitos insensibles a retardos, IR (*delay-insensitive circuits*): esta categoría engloba a los circuitos diseñados para funcionar correctamente independientemente de los retardos de sus componentes y conexiones.

  Los circuitos IR asumen íntegramente el modelo de retardos no acotados. Por tanto, no existe la total garantía de que, transcurrido un cierto tiempo, el circuito haya acabado el cómputo para los valores presentes en las entradas. Para asegurar que esto haya ocurrido se necesita incluir cierta lógica que permita comprobar que el circuito ha finalizado el cómputo.

En los circuitos IR ocurre que un único bit de información no puede ser enviado por una única conexión, puesto que no se podría distinguir entre las siguientes dos situaciones: dos datos iguales son enviados de manera consecutiva, o bien un único dato se mantiene estable durante un espacio de tiempo prolongado. Esto motiva la necesidad de una codificación más compleja que la lógica binaria. Más adelante se mostrarán distintas alternativas de codificación.

Una vez que ha finalizado el cómputo de un circuito IR, es necesario indicar al entorno del circuito esta situación. Por un lado, se debe indicar al entorno que provee de datos de entrada cuándo enviar nuevos valores. Por otro lado, se debe informar al entorno que recoge los valores de salida del momento en que estos se han generado. Este intercambio de información se debe controlar a través de un protocolo de comunicación. Las principales alternativas en los protocolos de comunicación se evalúan más adelante.

Las principales ventajas de este tipo de circuitos son dos. En primer lugar, su robustez frente a variaciones en la temperatura, voltaje de alimentación o proceso de fabricación, puesto que la funcionalidad del circuito es totalmente independiente de la temporización. En segundo lugar, esta separación entre la temporización y el funcionamiento del circuito permite que el diseño sea modular, lo que redunda en una mayor reutilización y posibilidad de optimización de los circuitos.

- Circuitos cuasi-insensibles a retardos (*quasi-delay-insensitive circuits*): se trata circuitos insensibles a retardos con la restricción de que todas las conexiones que se bifurquen deben dar lugar a rutas lógicas isocrónicas. Una bifurcación isocrónica (*isochronic fork*) es aquella conexión que se divide en varias ramas donde todos los caminos lógicos que parten de ella tienen exactamente el mismo retardo.

  Los circuitos cuasi-insensibles a retardos utilizan componentes menos complejos que los circuitos insensibles a retardos. Sin embargo, conseguir que todas las bifurcaciones de un circuito sean isocrónicas es una tarea difícil de llevar a cabo, y suele requerir de la inclusión de líneas de retardo[1].

[1]Una línea de retardo es un conjunto elementos lógicos que no afectan a la funcionalidad del circuito, pero sí a su retardo. Habitualmente se trata un número par de inversores conectados

- Circuitos independientes de la velocidad (*speed-independent circuits*): en este tipo de circuitos, presentados por Muller en 1.965 [Mil65], se relaja la condición acerca de las conexiones suponiendo que su retardo, o bien es cero, o bien es despreciable. No se conoce, sin embargo, el valor máximo del retardo de ningún componente del circuito. A efectos de diseño se verifica que el circuito funcione correctamente independientemente del retardo de las puertas lógicas.

  Existen potentes esquemas matemáticos para describir el comportamiento de este tipo de circuitos, como las redes de Petri [Pet66, Mur89] (ver 2.1.2), lo que permite su generación automática a partir de descripciones de alto nivel [BM91]. Además, la implementación de este tipo de circuitos es más sencilla que, por ejemplo, la de los circuitos insensibles a retardos.

  Con la disminución del tamaño del transistor, el retardo de las conexiones tiene un peso cada vez más importante. Por tanto, el principal inconveniente de este tipo de circuitos reside en conseguir, en la práctica, diseños donde el retardo de las conexiones sea despreciable.

- Circuitos autotemporizados[2] (*self-timed circuits*): esta clase de circuitos, descritos por Seitz en [MC80], contienen un grupo de elementos independientes del resto en cuanto a temporización se refiere. Cada uno de estos elementos está contenido en una "región equipotencial", donde el retardo de las conexiones, o bien es despreciable, o bien es conocido. A su vez, cada una de estas regiones puede ser un circuito independiente de la velocidad, o un circuito cuyo funcionamiento dependa de suposiciones acerca de los retardos de sus propios componentes. Sin embargo, no existen suposiciones acerca del retardo en las comunicaciones entre regiones. En otras palabras, se supone que la comunicación entre regiones es insensible a retardos.

La Figura A.3 muestra, resumida, la clasificación de los circuitos asíncronos en función de su modelo de retardos.

---

en serie.

[2]Traducción literal del término *self-timed*.

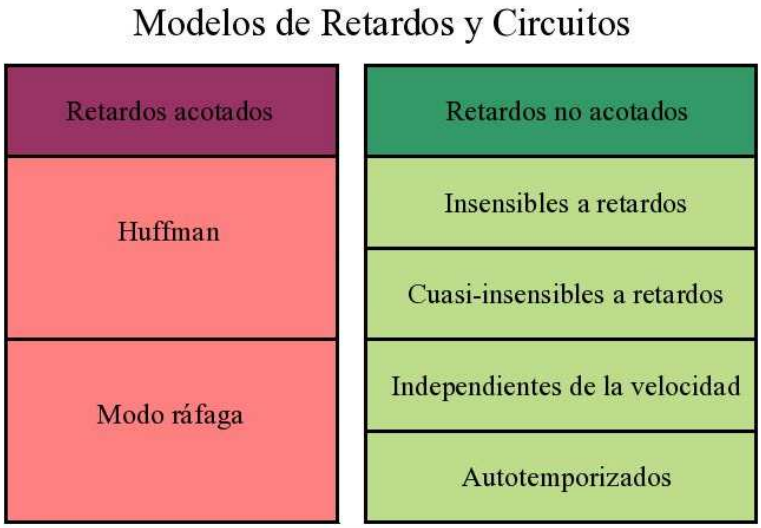


Figura A.3: Clasificación de circuitos asíncronos en función del modelo de retardos.

## A.3. Codificación de señales

Como ya se ha explicado, la lógica binaria se muestra insuficiente para algunos tipos de circuitos asíncronos, puesto que una conexión sólo es capaz de transmitir dos valores distintos.

Por ejemplo, en [Mar92], A. J. Martin mostró que los circuitos IR deben manejar dos clases de datos distintos. Por un lado, debe existir una clase de datos de cómputo, *d*. A esta clase pertenecen los datos involucrados en el cómputo (los datos 0 y 1 del álgebra de *Boole*). Por otro lado, debe existir la clase de datos de sincronismo, *s*. A esta clase pertenecen todos los datos utilizados para distinguir entre dos datos de clase *d* consecutivos. Por lo tanto, por cada conexión se deben transmitir, al menos, tres posibles valores distintos: los dos valores de cómputo de la clase *d* y, al menos, un valor de la clase *s*.

La generación de más de dos valores por conexión se puede conseguir utilizando distintos tipos de codificación o de lógicas de cómputo. Por tanto, la codificación elegida para las señales en un circuito asíncrono depende fundamentalmente del tipo de circuito a diseñar. Además, la codificación está estrechamente relacionada con la existencia de un protocolo de comunicación en el circuito, y con el propio

| $d_1$ | $d_0$ | ***Significado*** |
|---|---|---|
| 0 | 0 | Color 0 |
| 0 | 1 | Valor lógico 0 |
| 1 | 0 | Valor lógico 1 |
| 1 | 1 | Color 1 |

Tabla A.1: Significado de las combinaciones de valores binarios en una codificación doble raíl.

tipo de protocolo. En los siguientes apartados se describen las codificaciones más ampliamente utilizadas.

### A.3.1. Codificación en doble raíl[3] (*dual rail*)

En este tipo de codificación, cada bit de información se representa mediante dos conexiones ($d_1$, $d_0$) [NPI06, SS98]. Estas dos conexiones o señales permiten transmitir cuatro valores binarios. Como se muestra en la Tabla A.1, los valores lógicos 0 y 1 del álgebra de Boole se codifican activando únicamente los bits de menor y mayor peso respectivamente. Los valores donde ambas señales toman el mismo valor se denominan "colores", y se utilizan, por ejemplo, como separadores de datos en protocolos de comunicación.

La detección de fin de cómputo en este tipo de codificación consiste en detectar un valor distinto de alguno de los colores para todas las señales del dato que se está produciendo.

### A.3.2. Lógica de cuatro estados (*four-state logic, FSL*)

Extensión de la codificación doble raíl [McA92]. La idea principal de FSL es utilizar dos conjuntos de codificaciones disjuntos. Estos conjuntos, denotados como $\varphi_0$ y $\varphi_1$, representan el estado lógico de una señal, denotados como $LO$ para el 0 y $HI$ para el 1. De esta manera, además de indicar un valor lógico, representan la fase en que se encuentra el valor. Por ejemplo, para conexiones de dos bits se

[3]Derivado de la expresión inglesa *dual rail*, que se podría traducir literalmente como "doble carril".

| ***Estado lógico*** | ***Fase*** $\varphi_0$ | ***Fase*** $\varphi_1$ |
|---|---|---|
| *LO* | ($d_1$, $d_0$) = (0,0) | (0,1) |
| *HI* | ($d_1$, $d_0$) = (1,1) | (1,0) |

Tabla A.2: Esquema de codificación FSL para una conexión de dos bits ($d_1$, $d_0$).

dispondría de dos valores lógicos en dos fases distintas, como muestra la Tabla A.2.

En este tipo de codificación, dos datos transmitidos de manera consecutiva deben tener distinta fase, con objeto de poder distinguir dos valores consecutivos que podrían corresponder al mismo estado lógico. Además, toda transición de un valor en una fase a cualquier valor de la otra fase requiere la conmutación de una sola señal.

La detección de fin de cómputo en este esquema de codificación consiste en comprobar que todos los bits de un dato se encuentran en la misma fase. Considerando que una puerta XOR devolverá el valor 1 cuando un bit está en fase $\varphi_1$, la AND de todas las salidas de XOR se podría implementar como circuito de detección de fin de cómputo.

### A.3.3. Codificación de datos agrupados (*bundled data*)

La codificación de datos agrupados [KB97, SK96] normalmente se refiere a la utilización de señales binarias para codificar los datos en señales distintas a las señales implicadas para el protocolo de comunicación. Como se muestra en la Figura A.4, las señales de datos aparecen separadas de las señales de comunicación (*Req* y *Ack*). Este tipo de codificación se utiliza habitualmente en circuitos bajo el modelo de retardos acotados.

### A.3.4. Codificación "uno activo" ("*one-hot*")

La codificación *one-hot* consiste en disponer de tantas líneas de bits como valores distintos se desean transmitir. Sin embargo, en un instante dado sólo se activa una señal, aquella que corresponde al valor que se desea transmitir. Esta codificación también se denomina "1 de N" ("*1-of-N*") debido a su peculiar funcionamiento.

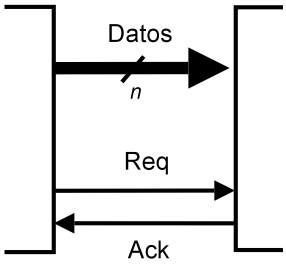


Figura A.4: Esquema de comunicación en una codificación *bundled data*. Las señales de datos aparecen separadas de las señales involucradas en el protocolo de comunicación: *Req* y *Ack*.

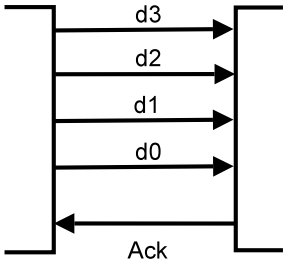


Figura A.5: Esquema de comunicación en una codificación *one-hot* para la transmisión de cuatro valores distintos. Sólo se activa la señal correspondiente al dato a transmitir. Las señales de datos, *d3* a *d0* aparecen separadas de la señal de reconocimiento para el protocolo de comunicación, *Ack*.

La Figura A.5 muestra el esquema de comunicación para la transmisión de cuatro valores distintos. En lógica binaria se podría utilizar dos bits de conexión para comunicar cuatro valores, sin embargo, *one-hot* requiere disponer de un bit por cada valor a transmitir. En el ejemplo de la figura se utilizan cuatro conexiones para datos, *d3* a *d0*, y una para el reconocimiento de comunicación, *Ack*.

## A.4. Protocolos de comunicación

Gran parte de los circuitos asíncronos necesitan utilizar protocolos de comunicación, bien para intercambiar datos con su entorno, bien para transmitir informa-

ción entre distintas partes del propio circuito.

En toda comunicación se distingue un emisor, origen de la información a transmitir, y un receptor, destino de esa información. Entre emisor y receptor se establece, por tanto, un canal de comunicación. Los canales de comunicación se dividen en dos tipos, identificados según el interlocutor que inicia la comunicación [Spa01]:

- Canales tipo *push*: es el emisor quien inicia la comunicación, "empujando" el dato hacia el receptor. La señal de solicitud de inicio de comunicación (ó *request*), *Req*, va desde el emisor al receptor. La señal de reconocimiento (ó *acknowledgement*), *Ack*, parte del receptor para llegar al emisor.
- Canales tipo *pull*: es el receptor quien "estira" del dato desde el emisor hacia sí mismo, iniciando la comunicación. De manera análoga al anterior, la señal *Req* va desde el receptor al emisor mientras que la señal *Ack* parte del emisor para llegar al receptor.

Existen multitud de variantes e implementaciones de mecanismos de comunicación. Sin embargo, los principales protocolos de comunicación a partir de los que surgen la mayoría de esas implementaciones son dos: el *protocolo handshake de cuatro fases* y el *protocolo handshake de dos fases*. A continuación se describen ambos protocolos utilizando canales tipo *push*, donde el emisor es quien solicita la comunicación.

### A.4.1. *Handshake* de cuatro fases

El protocolo *handshake* de cuatro fases recibe este nombre debido a que permite la comunicación de información entre un emisor y un receptor transcurridas cuatro etapas. En términos generales, el protocolo se inicia con una solicitud de transmisión. Tras el reconocimiento de esta solicitud, el dato se transmite de emisor a receptor. Por último, se reconoce el fin de la transmisión, terminando así la comunicación y el protocolo en sí. Estas cuatro fases se identifican gracias a los valores que van tomando las señales implicadas en el protocolo de comunicación.

Tanto el número de señales implicadas en el protocolo como los valores que cada una de ellas van transfiriendo dependen del esquema de codificación elegido (doble

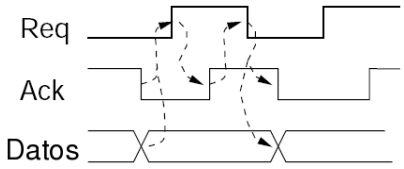


Figura A.6: Diagrama de evolución del protocolo *handshake* de cuatro fases entre dos circuitos bajo codificación *bundled data* en un canal tipo *push*.

raíl, *bundled-data*, ...). La Figura A.6 muestra un diagrama de comunicación bajo codificación *bundled data* entre dos circuitos, análoga al esquema de la Figura A.4. Las señales implicadas en el control del protocolo son dos: la señal de solicitud de comunicación, *Req*; y la señal de confirmación, *Ack*. La señal *Datos* se utiliza para transmitir la información.

En la Figura A.6 se ilustra la comunicación utilizando un canal tipo *push*. Partiendo del instante donde el valor de las señales *Req* y *Ack* es 0, las fases de la comunicación que se aprecian en dicha figura son las siguientes:

1. Solicitud de transmisión de información: el emisor coloca el valor a transmitir en la señal *Datos*. A continuación, el emisor establece la señal *Req* a 1, indicando la solicitud de transmisión o *request*.

2. Reconocimiento de la solicitud y comunicación del dato: en el momento que el receptor tiene constancia del valor 1 en *Req*, lee la información de la señal *Datos*. Una vez recogida la información, el receptor confirma la llegada de los datos estableciendo la señal *Ack* a 1.

3. Fin de solicitud de transmisión: el emisor, al recibir la confirmación de la transmisión, finaliza la solicitud poniendo el valor 0 en la señal *Req*.

4. Reconocimiento del fin de la comunicación por parte del receptor: el fin de la solicitud indica al receptor que el emisor ha recibido el *Ack*. En ese instante, el receptor indica el fin de la comunicación y su disposición para recibir nuevos datos estableciendo la señal *Ack* a 0.

Como muestra el ejemplo anterior, en el protocolo *handshake* de cuatro fases el nivel lógico de las señales involucradas en la comunicación indica su estado

(activado o no activado). Por ello, también se denomina protocolo RTZ (*return to zero*, vuelta a cero) ó protocolo por niveles.

### A.4.2. *Handshake* de dos fases

De nuevo, el nombre del protocolo hace alusión al número de etapas de la comunicación. En el caso del protocolo *handshake* de dos fases, la comunicación se produce en dos etapas. En la primera de las etapas se produce la transmisión del dato, mientras que en la segunda se confirma la correcta transmisión de la información.

Este protocolo consigue simplificar la comunicación descrita para el protocolo de cuatro fases. Para lograrlo, el protocolo considera las transiciones de las señales de control, no sus niveles lógicos. Este protocolo también se conoce como protocolo NRZ (*non return to zero*, sin vuelta a cero) ó protocolo por transiciones.

En la Figura A.7 se muestra una comunicación entre dos circuitos a través de un canal tipo *push*. Inicialmente, el valor de la señal *Req* es 0, mientras que *Ack* toma el valor 1. A partir de ese momento, las fases de la comunicación son las siguientes:

1. Transmisión de información: en esta fase se realizan dos acciones. En primer lugar, el emisor coloca el valor a transmitir en la señal *Datos*. A continuación, el emisor cambia el valor de la señal *Req* (en el ejemplo de la figura establecido a 1) para indicar que la transmisión está en curso.

2. Confirmación del dato: al detectar la transición en *Req*, el receptor lee el dato. Una vez leído, el receptor confirma la recepción modificando el valor de la señal *Ack* (en este caso estableciendo su valor a 0). Esta transición de la señal *Ack* indica el fin de la comunicación y provoca que el emisor disponga un nuevo dato para su posterior transmisión.

Al necesitar menos etapas, el protocolo de dos fases es más rápido que el de cuatro. Sin embargo, su implementación es más costosa en área puesto que la lógica necesaria para detectar las transiciones en las señales es más compleja que la lógica que permite identificar distintos niveles.

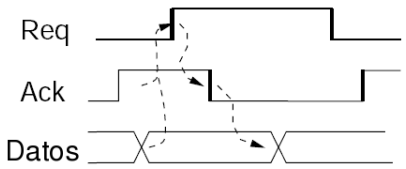


Figura A.7: Diagrama de evolución del protocolo *handshake* de dos fases entre dos circuitos bajo codificación *bundled data* en un canal tipo *push*.

# Apéndice B

# Gramáticas XML

Parte de la configuración de *Sim-async* se realiza utilizando archivos XML como parámetros de entrada del simulador. Como se ha explicado a lo largo de este trabajo, los archivos XML se refieren, bien a funciones de distribución (ver Capítulo 3), bien a la configuración de las simulaciones (ver Apartado 4.4.4 y Capítulo 5). Sin embargo, ciertos aspectos relacionados con XML no se han cubierto hasta este punto. En consecuencia, este apéndice pretende completar los detalles sobre la relación entre el simulador y XML.

## B.1. Conceptos básicos

*Sim-async* recibe la mayoría de sus parámetros de configuración a través de archivos XML. Cada uno de estos archivos, utilizando terminología de lenguajes de marcado, forma un *esquema*. Los esquemas son conjuntos de datos organizados en forma de árbol que se construyen respetando un conjunto de reglas dado. El conjunto de reglas que debe verificar un esquema XML se denomina *gramática* [Mor00].

Gracias a la relación entre esquemas y gramáticas es posible determinar si un esquema se ha construido correctamente sin más que verificar que cumple las reglas definidas en la gramática que le corresponde. Esa verificación no es un proceso complejo, más bien al contrario. La comprobación de que un esquema cumple las reglas dadas en una gramática se realiza a través de un proceso de análisis sintáctico (*parsing*). Este proceso se guía por las reglas definidas en cada gramática en particular, aunque el algoritmo de análisis es genérico.

*Sim-async* utiliza dos tipos de esquema XML distintos, uno para funciones de distribución y el otro para la configuración de la temporización de las simulaciones. Por tanto, es necesario definir dos gramáticas distintas, una por cada esquema. En los siguientes apartados se muestran las gramáticas definidas, comentando los detalles más relevantes de cada una de ellas.

## B.2. Gramática para funciones de distribución

Las funciones de distribución se construyen siguiendo la gramática definida en el archivo *distrib_ schema.xsd* del simulador, cuyo contenido se muestra en la Figura B.1. En esta gramática se define como raíz del esquema el elemento *distrib*. Este elemento se forma, a su vez, con las siguientes propiedades:

- *name*: elemento de inclusión obligatoria. Almacena una cadena de caracteres que servirá como nombre para la configuración.
- *items*: permite definir una función de distribución a través de un conjunto de retardos, asociando cada retardo a un valor de probabilidad. Este elemento *items* no se puede utilizar en el mismo esquema que el elemento *definition* que se explica más adelante. Los elementos dependientes de *items* son los dos siguientes:
  - *prec*: indica el número de decimales que se utiliza en la descripción de la función de distribución. Este número se denomina precisión porque, dado que los valores de probabilidad están normalizados, el número de decimales marcará el máximo número de valores de retardos distintos, Este valor de precisión sirve para acotar el máximo número de valores de retardo distintos, que serán $10^{prec}$. El elemento *prec* aparece una vez dentro de *items*.
  - *item*: asocia un valor de latencia (*delay*) a una probabilidad (*prob*). Este elemento puede aparecer más de una vez en un elemento *items*, pero se debe cumplir que la suma de las probabilidades de todos los elementos *item* sea 1, puesto que el simulador trabaja con valores de probabilidad normalizados. Esta comprobación se realiza en el proceso de análisis del esquema.

- *definition*: permite definir una función de distribución utilizando sus valores de media (*mean*) y desviación típica (*stdev*). Este elemento no es compatible con *items*.

En la Figura B.2 se muestra el contenido de un fichero XML que describe la función de distribución correspondiente al ejemplo de la Figura 3.2 (Capítulo 3). En el esquema se define la función de distribución asociando valores de probabilidad a los distintos valores de retardo. Para ello, el elemento *items* contiene tantos elementos *item* como retardos se definen, seis distintos para este ejemplo concreto. Cada *item* se componen de un valor de retardo (*delay*) y su probabilidad asociada (*prob*). La precisión (*prec*) en este ejemplo es 2, equivalente al número de decimales utilizados en los valores de probabilidad.

## B.3. Gramática para temporización

El esquema que permite configurar la temporización de las simulaciones de *Simasync* se construye siguiendo la gramática definida en el archivo *config_ schema.xsd*. En esta gramática se define como raíz del esquema el elemento *configuration*. Este elemento se forma, a su vez, con los siguientes componentes:

- Modo de simulación (*mode*). Indica simulación síncrona (*Sync*) ó asíncrona (*Async*). Determina el funcionamiento básico del motor de eventos del simulador.

- Protocolo de comunicación por defecto (*protocol*). Esta especificación se aplica en aquellas etapas donde no se indique un protocolo en particular. Es necesario determinar tanto el tipo de protocolo, a elegir entre *handshake* de dos ó de cuatro fases, como los correspondientes valores para los retardos $t_{fv}$ , $t_{ack}$ , $t_{fn}$ y $t_{sync}$ (ver Apartado 4.2.3).

- Configuración de etapas (*stages*). El simulador permite la configuración individual de las etapas del procesador. Cada una de las configuraciones individuales (*stage*) consta de los siguientes elementos:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="items_type">
      <xs:sequence>
         <xs:element name="item" type="item_type" minOccurs="1"
             maxOccurs="50"/>
      </xs:sequence>
      <xs:attribute name="prec" type="xs:integer" use="required"/>
   </xs:complexType>

   <xs:complexType name="item_type">
      <xs:attribute name="delay" type="xs:double"/>
      <xs:attribute name="prob" type="xs:double"/>
   </xs:complexType>

   <xs:complexType name="name_type">
      <xs:attribute name="value" type="xs:string"/>
   </xs:complexType>

   <xs:complexType name="definition_type">
      <xs:all>
         <xs:element name="mean" type="xs:string" minOccurs="1"
             maxOccurs="1"/>
         <xs:element name="stdev" type="xs:string" minOccurs="1"
             maxOccurs="1"/>
      </xs:all>
   </xs:complexType>

   <xs:element name="distrib">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="name" type="name_type"  minOccurs="1"
                maxOccurs="1"/>
            <xs:choice>
               <xs:element name="items" type="items_type"  minOccurs="1"
                   maxOccurs="1"/>
               <xs:element name="definition" type="definition_type"
                   minOccurs="1" maxOccurs="1"/>
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

</xs:schema>
```

Figura B.1: Archivo *distrib_ schema.xsd*. Contiene la gramática que deben verificar los esquemas correspondientes a las funciones de distribución. En *Sim-async*, las funciones de distribución se utilizan para modelar el tiempo de cómputo variable de las etapas y unidades funcionales del procesador.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
   <!-- Probability distribution function:
        (delay, probability) -->
   <distrib>
      <name value="distrib1" />
      <items prec="2">
         <item delay="0.5" prob="0.07" />
         <item delay="1.0" prob="0.07" />
         <item delay="1.5" prob="0.13" />
         <item delay="2.0" prob="0.40" />
         <item delay="2.5" prob="0.26" />
         <item delay="3.0" prob="0.07" />
      </items>
   </distrib>
```

Figura B.2: Esquema XML para la función de distribución que se muestra como ejemplo en la Figura 3.2 (Capítulo 3). La función se define utilizando pares formados por un valor de retardo (*delay*) y su probabilidad asociada (*prob*). La precisión (*prec*) en este ejemplo es 2, equivalente al número de decimales utilizados. La precisión acota el máximo número de valores de retardo distintos, que son 100 en este ejemplo ($10^{prec}$).

- Nombre de la etapa (*name*): indica cuál de las etapas recibe la configuración. Los posibles valores para este parámetro son los siguientes: *fetch*, *issue*, *exec*, *wb* y *commit*.
- Anchura de la etapa (*width*): número máximo de instrucciones a tratar en la etapa. El significado particular de la anchura depende de la etapa (ver Apartado 4.2.2). El rango de valores no está acotado en el simulador, aunque siempre debe ser mayor que cero.
- Temporización (*delay*): configura el tiempo de cómputo de la etapa. Para indicar latencia constante se incluye el valor de retardo en el elemento *fixed*. Si la latencia depende de una función de distribución, se debe indicar en el elemento *distrib* la ruta al fichero XML que la contenga.
- Protocolo de comunicación (*protocol*): determina el tipo de protocolo de comunicación a aplicar en la etapa. Este argumento permite utilizar diferentes protocolos de comunicación en distintas etapas. Los valores especificados prevalecen sobre el protocolo por defecto, y su configuración es similar al elemento *protocol* global.

- Sub-etapas (*substages*): permite configurar componentes individuales dentro de una etapa. En la versión actual del simulador se utiliza para configurar las unidades funcionales dentro de la etapa *Exec*. Este elemento tiene los mismos componentes que el elemento *stage*, así como los rangos de valores a utilizar, con la salvedad de que los valores permitidos para su elemento *name* son distintos. En este caso, al tratarse de unidades funcionales dentro de *Exec*, los valores permitidos son: *intUFx*, *intMulSegUF*, *fpAddSegUF*, *fpMulSegUF*, *fpDivUF*, *dirsUF* y *memUF*, correspondientes a las unidades de suma y multiplicación de enteros, suma, multiplicación y división de punto flotante, cálculo de direcciones y acceso a memoria respectivamente.

Debido a su longitud, el contenido del archivo *config_schema.xsd* se muestra en tres figuras separadas. En la Figura B.3 se presenta la parte inicial de la gramática, donde se definen los elementos más básicos del esquema. Estos elementos *mode*, *fixed*, *distrib*, etc., configurar las propiedades básicas para las simulaciones. En esta parte de la gramática se definen también los retardos asociados a cada tipo de protocolo posible: *handshake* de cuatro fases doble raíl (*drail_4ph_type*) y *handshake* de dos fases doble raíl (*drail_2ph_type*).

En la Figura B.4 se definen los componentes que configuran etapas y subetapas (*stage* y *substage*). Estos elementos conforman la parte intermedia en la jerarquía del esquema, y se agrupan a través de los elementos de nivel superior.

La tercera parte de la gramática, como se muestra en la Figura B.5, define los elementos del nivel superior en la jerarquía: *stages*, *protocol* y *configuration*. En ellos se configuran las etapas, el protocolo global y se define el elemento principal de la configuración.

Por último, en la Figura B.6 se presenta un ejemplo de fichero de configuración para el simulador. Este fichero define una configuración asíncrona con protocolo doble raíl de cuatro fases donde las etapas *Fetch*, *Write-back* y *Commit* tienen tiempo de cómputo constante, aunque distinto. Las unidades funcionales de la etapa *Exec* (no se muestran todas) se describen de manera individual, indicando latencia variable a través de una función de distribución para la unidad de enteros *intUFx* y para la etapa *Issue*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

      <xs:simpleType name="mode_name_type">
         <xs:restriction base="xs:string">
            <xs:enumeration value="Async"/> <xs:enumeration value="Sync"/>
         </xs:restriction>
      </xs:simpleType>

      <xs:complexType name="mode_type">
         <xs:attribute name="value" type="mode_name_type"/>
      </xs:complexType>

      <xs:complexType name="fixed_type">
         <xs:attribute name="value" type="xs:integer"/>
      </xs:complexType>

      <xs:complexType name="time_type">
         <xs:attribute name="value" type="xs:integer"/>
      </xs:complexType>

      <xs:complexType name="distrib_type">
         <xs:attribute name="file" type="xs:string"/>
      </xs:complexType>

      <xs:complexType name="width_type">
         <xs:attribute name="value" type="xs:integer"/>
      </xs:complexType>

      <xs:complexType name="sync_type">
         <xs:attribute name="t_cycle" type="xs:integer"/>
      </xs:complexType>

      <xs:complexType name="drail_4ph_type">
         <xs:attribute name="tfv" type="xs:integer"/>
         <xs:attribute name="tack" type="xs:integer"/>
         <xs:attribute name="tfn" type="xs:integer"/>
         <xs:attribute name="tsync" type="xs:integer"/>
      </xs:complexType>

      <xs:complexType name="drail_2ph_type">
         <xs:attribute name="tfv" type="xs:integer"/>
         <xs:attribute name="tack" type="xs:integer"/>
      </xs:complexType>

      <xs:complexType name="delay_type">
         <xs:choice>
            <xs:element name="fixed" type="fixed_type" minOccurs="1" maxOccurs="1"/>
            <xs:element name="distrib" type="distrib_type" minOccurs="1" maxOccurs="1"/>
         </xs:choice>
      </xs:complexType>
    ...
```

Figura B.3: Parte del archivo *config_ schema.xsd*. Este archivo contiene la gramática que deben verificar los esquemas que configuran la temporización de las simulaciones. En la figura se muestra la parte de esa gramática que define los tipos básicos para el modo, tiempo de cómputo fijo ó variable y distintos tipos de retardos.

```
...
      <xs:simpleType name="stage_name_type">
         <xs:restriction base="xs:string">
            <xs:enumeration value="fetch"/> <xs:enumeration value="issue"/>
            <xs:enumeration value="exec"/> <xs:enumeration value="wb"/>
            <xs:enumeration value="commit"/>
         </xs:restriction>
      </xs:simpleType>

      <xs:simpleType name="substage_name_type">
         <xs:restriction base="xs:string">
            <xs:enumeration value="memUF"/> <xs:enumeration value="dirsUF"/>
            <xs:enumeration value="intUFx"/> <xs:enumeration value="intMulSegUF"/>
            <xs:enumeration value="fpDivUF"/> <xs:enumeration value="fpMulSegUF"/>
            <xs:enumeration value="fpAddSegUF"/>
         </xs:restriction>
      </xs:simpleType>

      <xs:complexType name="stage_type">
         <xs:sequence>
            <!-- Anchura fija por defecto de 4 en todas las etapas-->
            <xs:element name="width" type="width_type" minOccurs="0" maxOccurs="1"/>
            <!-- Si no se indica ningún delay, se toma temporización fija a 100 ut-->
            <xs:element name="delay" type="delay_type" minOccurs="0" maxOccurs="1"/>
            <xs:choice>
               <xs:element name="protocol" type="protocol_type" minOccurs="0" maxOccurs="1"/>
               <xs:element name="substage" type="substage_type" minOccurs="1" maxOccurs="7"/>
            </xs:choice>
         </xs:sequence>
         <xs:attribute name="name" type="stage_name_type"/>
      </xs:complexType>

      <xs:complexType name="substage_type">
         <xs:sequence>
            <xs:element name="protocol" type="stage_protocol_type" minOccurs="0" maxOccurs="1"/>
            <xs:element name="delay" type="delay_type" minOccurs="0" maxOccurs="1"/>
         </xs:sequence>
         <xs:attribute name="name" type="substage_name_type"/>
      </xs:complexType>
...
```

Figura B.4: Segunda parte del archivo *config_ schema.xsd*. En la figura se muestran las definiciones de los elementos *stage* y *substage*.

```
...
      <xs:complexType name="stages_type">
         <xs:sequence>
            <xs:element name="stage" type="stage_type" minOccurs="0" maxOccurs="5"/>
         </xs:sequence>
         <xs:attribute name="name" type="stage_name_type"/>
      </xs:complexType>

      <xs:complexType name="stages_type">
         <xs:sequence>
            <xs:element name="stage" type="stage_type" minOccurs="0" maxOccurs="5"/>
         </xs:sequence>
         <xs:attribute name="name" type="stage_name_type"/>
      </xs:complexType>

      <xs:complexType name="protocol_type">
         <xs:choice>
            <!--Asíncrono doble raíl 2 fases-->
            <xs:element name="drail2ph" type="drail_2ph_type" minOccurs="1" maxOccurs="1"/>
            <!--Asíncrono doble raíl 4 fases-->
            <xs:element name="drail4ph" type="drail_4ph_type" minOccurs="1" maxOccurs="1"/>
            <!--Síncrono-->
            <xs:element name="clk" type="sync_type" minOccurs="1" maxOccurs="1"/>
         </xs:choice>
         <xs:attribute name="name" type="xs:string"/>
      </xs:complexType>

      <!--Impide que en una etapa se pueda seleccionar un reloj distinto.
          Si se requiere síncrona, establecerlo en el default.-->
      <xs:complexType name="stage_protocol_type">
         <xs:choice>
            <!--Asíncrono doble raíl 2 fases-->
            <xs:element name="drail2ph" type="drail_2ph_type" minOccurs="1" maxOccurs="1"/>
            <!--Asíncrono doble raíl 4 fases-->
            <xs:element name="drail4ph" type="drail_4ph_type" minOccurs="1" maxOccurs="1"/>
         </xs:choice>
         <xs:attribute name="name" type="xs:string"/>
      </xs:complexType>

      <xs:element name="configuration">
         <xs:complexType>
            <xs:sequence>
               <!--Modo de funcionamento del simulador-->
               <xs:element name="mode" type="mode_type"  minOccurs="1" maxOccurs="1"/>
               <!--Protocolo por defecto-->
               <xs:element name="protocol" type="protocol_type" minOccurs="1" maxOccurs="1"/>
               <!--Configuración específica para cada etapa-->
               <xs:element name="stages" type="stages_type" minOccurs="1" maxOccurs="1"/>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string"/>
       </xs:complexType>
    </xs:element>

</xs:schema>
```

Figura B.5: Tercera parte del archivo *config_ schema.xsd*. En la figura se muestran las definiciones de los elementos *stages*, *protocol* y *configuration*.

```
<configuration name="tesisEJ">
   <mode value="Async" />
   <protocol name="protoGlobal">
      <drail4ph tfv="20" tfn="20" tack="5" tsync="5" />
   </protocol>
   <stages>
      <stage name="fetch">
         <width value="4"/>
         <delay> <fixed value="1000"/> </delay>
      </stage>
      <stage name="issue">
         <width value="4"/>
         <delay> <distrib file="fc_distrib_norm.xml"/> </delay>
      </stage>
      <stage name="exec">
         <substage name = "intUFx">
            <delay> <distrib file="mc_distrib.xml"/> </delay>
            <protocol name="fo4_4ph_proto">
               <drail4ph tfv="20" tfn="10" tack="5" tsync="5" />
            </protocol>
         </substage>
         <substage name = "dirsUF" >
            <delay> <fixed value="1000"/> </delay>
         </substage>
         <substage name = "intMulUF" >
            <delay> <fixed value="4000"/> </delay>
         </substage>
         ...
      </stage>
      <stage name="wb">
         <width value="2"/>
         <delay> <fixed value="500"/> </delay>
      </stage>
      <stage name="commit">
         <width value="4"/>
         <delay> <fixed value="2000"/> </delay>
      </stage>
   </stages>
</configuration>
```

Figura B.6: Ejemplo de fichero XML de temporización para *Sim-async*. Muestra una configuración asíncrona con protocolo doble raíl de cuatro fases donde las etapas *Fetch*, *Write-back* y *Commit* tienen tiempo de cómputo constante. Las unidades funcionales de la etapa *Exec* (no se muestran todas) se describen de manera individual, indicando latencia variable para la unidad *intUFx* y para la etapa *Issue*, cada una descrita por una función de distribución distinta.

## B.4. Resumen

Los esquemas XML permiten configurar dos aspectos distintos de las simulaciones de *Sim-async*. Por un lado, gracias a la gramática definida en el archivo *distrib_schema.xsd*, es posible especificar la distribución de retardos asociados al tiempo de cómputo de una etapa o unidad funcional. Por otro lado, siguiendo la gramática definida en *config_schema.xsd*, se determina la configuración de las simulaciones en cuanto a parámetros globales sobre etapas y protocolos de comunicación se refiere, así como también se determina la configuración de cada etapa en particular.

La estrategia de parametrización a través de esquemas XML permite al simulador utilizar tipos de datos y funciones externas para el análisis y verificación de esquemas. Estas funciones y tipos de datos están disponibles en la librería para XML que utiliza el simulador, lo que le proporciona abstracción y, por tanto, extensibilidad.

# Bibliografía


[ALE02] T. M. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer Journal*, 35, 2:59–67, Feb 2002.

[BE97] A. Bardsley and D. Edwards. Compiling the language Balsa to delay insensitive hardware. In *CHDL'97: Proc. of the IFIP TC10 WG10.5 Int'l Conference on Hardware Description Languages and their Applications : Specification, Modelling, Verification and Synthesis of Microelectronic Systems*, pages 89–91, London, 1997. Chapman & Hall, Ltd.

[BEW00] J. Beister, G. Eckstein, and R. Wollowski. CASCADE: A tool kernel supporting a comprehensive design method for asynchronous controllers. In *Proc. of Int'l Conference on Application and Theory of Petri Nets*, pages 445–454, Jun 2000.

[BF98] W. J. Bainbridge and S. B. Furber. Asynchronous macrocell interconnect using MARBLE. In *Proc. of the 4th Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–132, 1998.

[Bla92] F. J. Blatt. *Modern Physics*. Mcgraw-Hill College, Jan 1992.

[BM88] S. M. Burns and A. J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In *Proc. of the 5th MIT conference on Advanced research in VLSI*, pages 35–50, Cambridge, MA, USA, 1988. MIT Press.

[BM91] P. A. Beerel and T. H.-Y. Meng. Testability of asynchronous timed control circuits with delay assumptions. In *Proc. of the 28th ACM/IEEE Design Automation Conference*, pages 446–451, 1991.

[BWZ92] B. M. Bara, D. J. Wilson, and B. W. Zelt. Concentration fluctuation profiles from a water channel simulation of a ground-level release. *Atmospheric environment. Part A, General topics*, 26:1053–1062, 1992.

[BY06] A. Bink and R York. ARM996HS: The first licensable, clockless 32-bit processor core. In *Hot Chips 18*, Aug 2006.

[BY07] A. Bink and R York. ARM996HS: The first licensable, clockless 32-bit processor core. *IEEE Micro*, 27(2):58–68, 2007.

[CA02] S. Chakraborty and R. Angrish. Probabilistic timing analysis of asynchronous systems with moments of delays. In *Proc. of the 8th Int'l Symposium on Asynchronus Circuits and Systems (ASYNC '02)*, pages 99–108, Washington, DC, USA, 2002. IEEE Computer Society.

[CCCGV06] J. Carmona, J. M. Colom, J. Cortadella, and F. Garcia-Valles. Synthesis of asynchronous controllers using integer linear programming. *Proc. of IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1637 – 1651, 2006.

[CFKM98] J. Casmira, J. Fraser, D. Kaeli, and W. Meleis. Operating system impact on trace-driven simulation. In *31st Annual Simulation Symposium, 1998*, pages 76 – 82, Apr 1998.

[CFPP95] C. Chien, M. A. Franklin, T. Pan, and P. Prabhu. ARAS: asynchronous RISC architecture simulator. *Proc. of the 2nd Working Conference on Asynchronous Design Methodologies (ASYNC'95)*, pages 210–219, 1995.

[CG06] K-L. Chang and B-H. Gwee. A low-energy low-voltage asynchronous 8051 microcontroller core. In *Proc. of 2006 IEEE Int'l Symposium*

*on Circuits and Systems, 2006. ISCAS 2006.*, pages 3181–3184, May 2006.

[CGL03] J. M. Colmenar, O. Garnica, and J. Lanchares. Técnica de estimación del rendimiento de pipelines asíncronos. In *XIV Jornadas de Paralelismo*, pages 397–402, Sep 2003.

[CGL+04] J. M. Colmenar, O. Garnica, S. Lopez, J. I. Hidalgo, J. Lanchares, and R. Hermida. Empirical characterization of the latency of long asynchronous pipelines with data-dependent module delays. *Proc. of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pages 311–321, 2004.

[CGL+06a] J. M. Colmenar, O. Garnica, J. Lanchares, J. I. Hidalgo, G. Miñana, and S. Lopez. Comparing the performance of a 64-bit fully-asynchronous superscalar processor versus its synchronous counterpart. In *Proc. of the 9th EUROMICRO Conference on Digital System Design*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[CGL+06b] J. M. Colmenar, O. Garnica, J. Lanchares, J. I. Hidalgo, G. Miñana, and S. Lopez. Sim-async: An architectural simulator for asynchronous processor modeling using distribution functions. *Proc. of the 12th Int'l Euro-Par Conference (Euro-Par 2006), Lecture Notes in Computer Science*, 4128/2006:495–505, 2006.

[Cha84] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, 1984.

[CHAG05] K. Cantillo, R. E. Haber, A. Alique, and R. Galán. Modeling of communication delays aiming at the design of networked supervisory and control systems. A first approach. *Lecture Notes in Computer Science*, 3516/2005:1056–1059, May 2005.

[Che98] F. Cheng. Practical design and performance evaluation of completion detection circuits. In IEEE Computer Society Press, editor, *Proc. of the Int'l Conference on Computer Design*, pages 354–359, 1998.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and Systems Science*, 5(5):511–523, 1971.

[Chu87] T-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, Jun 1987.

[Chu92] T-A Chu. Automatic synthesis and verification of hazard-free control circuits from asynchronous finite state machine specifications. In *Proc. of the 1991 IEEE Int'l Conference on Computer Design on VLSI in Computer & Processors (ICCD '92)*, pages 407–413, Washington, DC, USA, 1992. IEEE Computer Society.

[CKK+97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers (special issue on asynchronous circuit and system design). *IEICE Trans. on Information and Systems*, 80(3):315–325, 1997.

[CKLS06] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 25(10):1904 – 1921, 2006.

[CLSL02] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Proc. of the 5th. Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 13 – 22, Feb 2002.

[CMG+08] J. M. Colmenar, N. Morón, O. Garnica, J. Lanchares, and J. I. Hidalgo. Modelling asynchronous systems using probability distribution functions. In *Proc. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 3–11, Washington, DC, USA, 2008. IEEE Computer Society.

[Cor99] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*. 1999.

[CV08] Z. Chishti and T. N. Vijaykumar. Optimal power/performance pipeline depth for SMT in scaled technologies. *IEEE Trans. on Computers*, 57(1):69–81, 2008.

[CZ05] S. Chen and T. Zhang. Self-timed dynamically pipelined adaptive signal processing system: a case study of DLMS equalizer for read channel. *IEEE Trans. on Circuits and Systems I: Regular Papers*, 52(7):1338–1347, Jul 2005.

[DeJ75] K. A. DeJong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, Univ. Michigan, 1975.

[DN97] A. Davis and S. M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, Computer Science Department, University of Utah, Sep 1997.

[EB99] J. Ebergen and R. Berks. Response time properties of linear asynchronous pipelines. *Proc. of the IEEE*, 87(2):308–318, Feb 1999.

[EB00] D. A. Edwards and A. Bardsley. The Balsa asynchronous circuit synthesis system. In *Proc. 3rd Int'l Forum on Design Languages*, pages 37–44. Elsevier Science, Sep 2000.

[EF98] P. Endecott and S. Furber. Modelling and simulation of asynchronous systems using the LARD hardware description language. In *Proc. of the 12th European Simulation Multiconference on Simulation - Past, Present and Future*, pages 39–43. SCS Europe, 1998.

[FB06] M. Ferretti and P. A. Beerel. High performance asynchronous design using single-track full-buffer standard cells. *IEEE Journal of Solid-State Circuits*, 41(6):1444–1454, 2006.

[FCFW05] Wu-chang Feng, F. Chang, Wu-chi Feng, and J. Walpole. A traffic characterization of popular on-line games. *Networking, IEEE/ACM Trans. on*, 13:488– 500, Jun 2005.

[Fet06] E. S. Fetzer. Using adaptive circuits to mitigate process variations in a microprocessor design. *IEEE Design & Test of Computers*, 23(6):476–483, 2006.

[FGG98] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proc. of the Int'l Conference on Computer Design: VLSI in Computers and Processors, ICCD '98.*, pages 247–252, Oct 1998.

[FKS99] J. Fente, K. Knutson, and C. Schexnayder. Defining a beta distribution function for construction simulation. In *Simulation Conference Proc.*, pages 1010–1015, 1999.

[FNAT96] J. K. Flanagan, B. E. Nelson, J. K. Archibald, and G. Thompson. The inaccuracy of trace-driven simulation using incomplete mulit-programming trace data. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1996., MASCOTS '96*, pages 37–43, 1996.

[FNT+99] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, Jul 1999.

[GBB+00] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, J. Liu, D. W. Lloyd, S. Mohammadi, J. S. Pepper, O. Petlin, S. Temple, and J. V. Woods. AMULET3i - An asynchronous System-on-Chip. In IEEE Computer Society Press, editor, *Proc. of the 6th Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 162–175, Apr 2000.

[GFC99] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *Proc. of the 5th Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, Apr 1999.

[Gho01] S. Ghosh. P2EDAS: Asynchronous, distributed event driven simulation algorithm with inconsistent event preemption for accurate execution of VHDL descriptions on parallel processors. *IEEE Trans. on Computers*, 50(1):28–50, 2001.

[GI05] M. Ghoneima and Y. I. Ismail. Accurate decoupling of capacitively coupled buses. In *Proc. of the Int'l Symposium on Circuits and Systems, 2005. ISCAS 2005*, volume 4, pages 4146–4149. IEEE, 2005.

[GK03] A. Golda and A. Kos. Temperature influence on power consumption and time delay. In *DSD '03: Proc. of the Euromicro Symposium on Digital Systems Design*, pages 378–382, Washington, DC, USA, 2003. IEEE Computer Society.

[GW00] R. Geist and J. Westall. Correlational and distributional effects in network traffic models. In *IEEE Int'l Computer Performance and Dependability Symposium, 2000. IPDS 2000.*, pages 113 – 122, Mar 2000.

[Hau95] S. Hauck. Asynchronous design methodologies: An overview. *Proc. of the IEEE*, 83(1):69–93, Jan 1995.

[HBJ+02] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. *SIGARCH Comput. Archit. News*, 30(2):14–24, 2002.

[Hen00] J. L. Henning. SPEC CPU 2000: Measuring CPU performance in the new millenium. *IEEE Computer*, 33(7):28–35, Jul 2000.

[HMH01] R. Ho, K. W. Mai, and M. Horowitz. The future of wires. *Proc. of the IEEE*, 89(4):490–504, Apr 2001.

[Ho05] R. Ho. High-performance ULSI: the real limiter to interconnect scaling. In *Proc. of the Seventh Int'l Workshop on System-Level Interconnect Prediction (SLIP 05)*, pages 3–3. ACM, Apr 2005.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug 1978.

[Hol82] L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Trans. on Computers*, 31(12):1133–1141, Dec 1982.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecture (4th ed.). A quantitative approach.* Morgan Kaufmann Publishers Inc., 2007.

[Huf54] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257:161–190 (Mar),275–303 (Apr), 1954.

[JKKC00] Y. Jung, D. Kim, Y. Kim, and Y. Chiba. simCore: An event-driven simulation framework for performance evaluation of computer systems. *8th IEEE Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'00)*, 00:274–280, 2000.

[KAAK05] A. Kabbani, D. AlKhalili, and A. J. Al-Khalili. Technology portable analytical model for DSM CMOS inverter delay estimation. *IEE Proc. on Circuits, Devices and Systems*, 152(5):433–440, Oct 2005.

[KB97] D. Kearney and N. W. Bergmann. Bundled data asynchronous multipliers with data dependent computation times. In *ASYNC '97: Proc. of the 3rd Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 186–197, Washington, DC, USA, 1997. IEEE Computer Society.

[KBK02] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGOPS Operating Systems Review*, 36(5):211–222, 2002.

[Kea99] D. Kearney. Theoretical limits on the data dependent performance on asynchronous circuits. *Proc. of Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 201–207, 1999.

[KL02] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7–7, Jan 2002.

[KP01] J. Kessels and A. Peeters. The Tangram framework: Asynchronous circuits for low power. In *Proc. of the Asia and South Pacific Design Automation Conference*, pages 255–260, Feb 2001.

[KS73] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. on Computers*, 22:786–792, 1973.

[KY02] W. Kuang and J. S. Yuan. Low power operation using self-timed circuits and ultra-low supply voltage. *14th Int'l Conference on Microelectronics, (ICM)*, pages 185–188, Dec 2002.

[LGH+05] S. López, O. Garnica, J. I. Hidalgo, J. Lanchares, J. M. Colmenar, and G. Miñana. Study of the communication channels in a globally asynchronous locally synchronous simultaneous multithreading architecture. In *Proc. of the 2nd Int'l Conference on Automation, Control and Instrumentation (IADAT 2005)*, 2005.

[Lim84] INMOS Limited. *Occam Programming Manual*. Prentice Hall International, 1984.

[LL06] D. J. Lary and L. Lait. Using probability distribution functions for satellite validation. *IEEE Trans. on Geoscience and Remote Sensing*, 44:1359 – 1366, May 2006.

[LMS06] L. Lavagno, G. Martin, and L. Scheffer. *Electronic design automation for integrated circuits handbook*, volume II. Chapman & Hall / CRC Press / Kluwer, 2006. Cap. 8, "Static Timing Analysis". ISBN 0-8493-3096-3.

[LWAM07] B. Lasbouygues, R. Wilson, N. Azemard, and P. Maurine. Temperature- and voltage-aware timing analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 26(4):801–815, Apr 2007.

[MABB02] D. Marculescu, D. Albonesi, P. Bose, and A. Buyuktosunoglu. Partially asynchronous microprocessor design. Tutorial at 35th Annual Int'l Symposium on Microarchitecture (MICRO-35), 2002.

[Mar86] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[Mar92] A. J. Martin. Asynchronous datapaths and the design of an asynchornous adder. *Formal Methods in System Design*, 1(1):119–137, Jul 1992.

[MB57] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. *Proc. of an Int'l Symposium on the Theory of Switching, Cambridge*, pages 204–243, 1957.

[MBL+89] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The first asynchronous microprocessor: The test results. *Computer Architecture News*, 17(4):95–98, 1989.

[MC80] C. Mead and L. Conway. *Introduction to VLSI systems*, chapter 7: System timing. Seitz, C. L. Addison-Wesley, Reading, MA, 1980.

[McA92] A. J. McAuley. Four state asynchronous architectures. *IEEE Trans. on Computers*, 41(2):129–142, 1992.

[MF83] C. E. Molnar and T-P. Fang. Synthesis of reliable speed-independent circuit modules: I. General method for specification of module-environment interaction and derivation of a circuit realization. Technical Memorandum 297, Computer Systems Laboratory, Institute for Biomedical Computing, Washington Univ., St. Louis, MO, 1983.

[MFJ92] I. Miller, J. E. Freund, and R. A. Johnson. *Probabilidad y estadística para ingenieros*. Prentice-Hall Hispanoamericana, 1992.

[MHG+06] G. Miñana, J. I. Hidalgo, O. Garnica, J. Lanchares, J. M. Colmenar, and S. López. A technique to reduce static and dynamic power of functional units in high-performance processor. *Proc. of the 33th Int'l Workshop on Power and Timing Modelling, Optimization and Simulation (PATMOS 2006), Lecture Notes in Computer Science*, 4148/2006:514–523, 2006.

[MHL+07] G. Miñana, J. I. Hidalgo, J. Lanchares, J. M. Colmenar, O. Garnica, and S. López. Reducing power of functional units in high-performance processors by checking instruction codes and resizing adders. *IET Computers & Digital Techniques*, 1(2):113–119, 2007.

[Mil65] R. E. Miller. *Switching Theory. Volume II: Sequential Circuits and Machines*. New York, NY, 1965.

[MLM+97] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Proc. of the 17th Conference on Advanced Research in VLSI*, pages 164–181, Washington, DC, USA, 1997. IEEE Computer Society.

[MN06] A. J. Martin and M. Nystrom. Asynchronous techniques for system-on-chip design. *Proc. of the IEEE*, 94(6):1089–1120, 2006.

[MNCJ05] P. B. McGee, S. M. Nowick, and E. G. Coffman Jr. Efficient performance analysis of asynchronous systems based on periodicity. In *Proc. of the 3rd IEEE/ACM/IFIP Int'l conference on Hardware/-software codesign and system synthesis (CODES+ISSS '05)*, pages 225–230, New York, NY, USA, 2005. ACM Press.

[MNP+03] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P Prakash, C. G. Wong, J. Chang, K. S. Ko, B Lee, E. Ou, J. Pugh, E. V. Talvala, J. T. Tong, and A. Tura. The Lutonium: a sub-nanojoule asynchronous 8051 microcontroller. In *Proc. of 9th Int'l Symposium on Asynchronous Circuits and Systems*, pages 14–23, May 2003.

[Mor00] M. Morrison. *XML al descubierto*. Prentice-Hall, Madrid, 2000.

[Mul63] D. E. Muller. Asynchronous logics and application to information processing. *Proc. Symposium on Application of Switching Theory in Space Technology*, pages 289–297, 1963.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, Apr 1989.

[ND92] S. M. Nowick and D. L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *Int'l Conference on Computer-Aided Design (ICCAD)*, pages 626–630, 1992.

[Nie97] L. S. Nielsen. *Low-power Asynchronous VLSI Design*. PhD thesis, 1997.

[NKB04] D. Nellans, V. Krishna Kadaru, and E. Brunvand. ARCS: an architectural level communication driven simulator. In *Proc. of the 14th ACM Great Lakes Symposium on VLSI (GLSVLSI '04)*, pages 73–77, New York, NY, USA, 2004. ACM Press.

[NNSvB94] L. S. Nielsen, C. Niessen, J. Sparsø, and K. van Berkel. Low-power Operation Using Self-timed Circuits and Adaptive Scaling of the Supply Voltage. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2(4):391–397, Dec 1994.

[Now93] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[NPI06] E. Nigussie, J. Plosila, and J. Isoaho. Full-duplex link implementation using dual-rail encoding and multiple-valued current-mode logic. In *IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, pages 2217–2220, 2006.

[OB06] R. O. Ozdag and P. A. Beerel. An asynchronous low-power high-performance sequential decoder implemented with QDI templates. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 14(9):975–985, 2006.

[OMCK+07] S. Ozdemir, A. Mallik, J. Chunk Ku, G. Memik, and Y. Ismail. Variable latency caches for nanoscalar processor. In *Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, pages 10–16. ACM, Nov 2007.

[PDF+98] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP.

In *Int'l Symposium on Advanced Reseach in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[Pet66] C. A. Petri. Communication with automata. Technical Report RADC-TR-65-377, Reconnaissance-Intelligence Data Handling Branch, Rome Air Develop, Center, Griffin AFB, New York, Jan 1966.

[Pie94] R. F. Pierret. *Dispositivos de efecto campo*. Addison-Wesley Iberoamericana, 1994.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. of 24th Annual Int'l Symposium on Computer Achitecture*, pages 206–218, 1997.

[QPX06] W. Qing, M. Pedram, and W. Xunwei. An asynchronous low-power high-performance sequential decoder implemented with QDI templates. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 14(9):975–985, 2006.

[Reb97] V. Rebello. *On the Distribution of Control in Asynchronous Processor Architectures*. PhD thesis, 1997.

[RY85] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proc. of Int'l Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, Jul 1985. IEEE Computer Society.

[SB06] K. Sangyun and P. A. Beerel. Pipeline optimization for asynchronous circuits: complexity analysis and an efficient optimal algorithms. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):389 – 402, 2006.

[SHG+05] S. Sun, Y. Han, X. Guo, K. H. Chong, L. McMurchie, and C. Sechen. 409ps 4.7 FO4 64b adder based on output prediction logic in 0.18um CMOS. In *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI)*, pages 52–58. IEEE Computer Society, 2005.

[Sim97] D. Sima. Superscalar instruction issue. *IEEE Micro*, 17:28–39, Sep-Oct 1997.

[SK96] V. Schöber and T. Kiel. An asynchronous scan path concept for micropipelines using the bundled data convention. In *Proc. of the IEEE Int'l test conference on test and design validity*, pages 225–231, Washington, DC, USA, 1996. IEEE Computer Society.

[SM97] A. E. Sjogren and C. J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. In *17th Conference on Advanced Research in VLSI*, pages 47–61, 1997.

[Spa01] J. Sparsø. Asynchronous circuit design - a tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems perspective"*, pages 1–152. Kluwer Academic Publishers, Boston / Dordrecht / London, Dec 2001.

[SRG+01] K. S. Stevens, S. Rotem, R. Ginosar, P. A. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, Feb 2001.

[SS98] M. Storto and R. Saletti. Time-multiplexed dual-rail protocol for low-power delay-insensitive asynchronous communication. In Anne-Marie Trullemans-Anckaert and Jens Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 127–136, 1998.

[Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, Jun 1989.

[Syn07] Synopsys, Inc. *Design Compiler User Guide. Version A-2007.12.* Dec 2007.

[The03] G. Theodoropoulos. Modeling an Asynchronous Microprocessor. *SIMULATION*, 79(7):377–409, 2003.

[The07] The Math Works, Inc. *MATLAB User Guides.* 2007.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Reseach and Development*, pages 25–33, 1967.

[TTW97] G. K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods. Occam: an asynchronous hardware description language? In *Proc. of the 23rd EUROMICRO Conference: New Frontiers of Information Technology*, pages 249–256, Sep 1997.

[UTB+06] O. S. Unsal, J. W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. *IEEE Micro*, 26(6):30–39, 2006.

[vBBK+94] C. H. van Berkel, R. Burguess, J. Kessels, A. Peeters, M. Roneken, and F. Schalij. Asynchronous circuits for low power: a DCC error corrector. *IEEE Design and Test*, 11(2):22–32, 1994.

[vBKR+91] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. of the European Conference on Design Automation. (EDAC)*, pages 384–389, 1991.

[vGBvB+98] H. van Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80C51 microcontroller. In *Proc. of Fourth Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.

[WM96] R. Witek and J. Montanaro. StrongARM: a high-performance ARM processor. *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pages 188–191, Feb 1996.

[WS85] C. Whitby-Strevens. The transputer. *SIGARCH Comput. Archit. News*, 13(3):292–300, 1985.

[XB97] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In

*Proc. of the 3rd Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '97)*, pages 64–75, Washington, DC, USA, 1997. IEEE Computer Society.

[XB00] A. Xie and P. A. Beerel. Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 239–268. Kluwer Academic Publishers, 2000.

[XKB99] A. Xie, S. Kim, and P. A. Beerel. Bounding average time separations of events in stochastic timed Petri nets with choice. In *Proc. of the 5th Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '99)*, pages 94–107, Washington, DC, USA, 1999. IEEE Computer Society.

[YCK+94] E. Yee, R. Chan, P. R. Kosteniuk, G. M. Chandler, C. A. Biltoft, and J. F. Bowers. Concentration fluctuation measurements in clouds released from a quasi-instantaneous point-source in the atmospheric surface-layer. *Boundary - layer meteorology*, 71:341–373, 1994.

[YDN92] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. of Int'l Conference on Computer Design*, pages 346–350. ICSP, Oct 1992.

[ZT04] Q. Zhang and G. Theodoropoulos. Modelling SAMIPS: a synthesisable asynchronous MIPS processor. In *Proc. of 37th Annual Simulation Symposium*, pages 205 – 212, Apr 2004.