UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadoras y Automática

REDUCCIÓN DEL CONSUMO DE POTENCIA EN UNIDADES FUNCIONALES MEDIANTE COTEJO DE CÓDIGOS DE OPERACIÓN.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Guadalupe Miñana Ropero

Bajo la dirección de los doctores

José Ignacio Hidalgo Pérez
Juan Lanchares Dávila
Óscar Garnica Alcázar

Madrid, 2009

- **ISBN: 978-84-692-8422-3**

Reducción del Consumo de Potencia en Unidades Funcionales Mediante Cotejo de Códigos de Operación

Guadalupe Miñana Ropero

Tesis Doctoral

Universidad Complutense de Madrid
Departamento de Arquitectura de Computadores y Automática

Reducción del Consumo de Potencia en Unidades Funcionales Mediante Cotejo de Códigos de Operación

Memoria presentada por Guadalupe Miñana Ropero para optar al grado de Doctor por la Universidad Complutense de Madrid. Trabajo realizado en el Departamento de Arquitectura de Computadores y Automática de la Facultad de Informática de la Universidad Complutense de Madrid bajo la dirección de los doctores José Ignacio Hidalgo Pérez, Juan Lanchares Dávila, Oscar Garnica Alcázar.

Madrid Enero 2009

A Luismi

Esta tesis ha sido posible gracias a la financiación de la Comisión Interministerial de Ciencia y Tecnología, a través de los proyectos CICYT TIC 2002-00750 y CICYT TIC 2005-05619

Agradecimientos

Al concluir la investigación que he realizado durante estos años me gustaría agradecer a todas aquellas personas que han contribuido de alguna manera a este proyecto ayudándome en su realización o animandome a llevarlo a cabo.

El trabajo no habría sido posible sin la valiosa ayuda de mis directores, Jose Ignacio Hidalgo Pérez, Juán Lanchares Dávila y Oscar Garnica Alcazar. Han demostrado tener una paciencia y tesón superiores a lo que sería esperable.

Mi más sincero agradecimiento a Francisco Tirado Fernández y a Román Hermida Correa por la confianza que han depositado en mí desde que me incorporé al departamento.

Gracias a mis compañeros de despacho Fredy Rivera, Teresa Higuera y Sara Román por aguantarme en los momentos difíciles, y por los buenos momentos que hemos pasado.

A Marcos Sánchez-Élez, Juan Carlos Fabero, Silvia del Pino, Inmaculada Pardines, José Herrera, Katia, Elena Pérez, Sonia López, Nacho Gómez, Dani Chaver, Guillermo Botella, Miguel Peón, Pablo García, Jose Luis Vázquez ... gracias por ser tan buenos compañeros y por las risas que nos hemos echado.

Gracias a David Atienza, por su apoyo durante mi estancia en la Suiza.

Gracias a Enrique de la Torre por haber estado pendiente de que las má-

quinas funcionasen correctamente. Sin él no se podrían haber realizado las simulaciones de esta tesis.

Y, por supuesto, a Luismi y a mi familia, porque no han perdido la fe en mí y me han seguido apoyando en todo momento y sobre toda circunstancia.

Índice general

1. Introducción **1**

1.1. Procesadores de propósito general 6

1.2. Objetivos principales de este trabajo 9

1.3. Organización de este trabajo 11

2. Herramientas y metodologías de simulación para evaluar el consumo **13**

2.1. Fuentes de potencia en los diseños actuales 14

2.2. Métricas 16

2.3. Simuladores 20

2.3.1. Clasificación de los simuladores 21

2.3.2. Simuladores más utilizados 27

2.4. Conclusiones 38

3. Técnicas de bajo consumo **41**

3.1. Tendencias en el diseño de bajo consumo 42

3.2. Técnicas de diseño de bajo consumo a nivel de microarquitectura 45

3.3. Técnicas aplicadas a las unidades funcionales 65

3.4. Conclusiones 78

4. Adaptación de un simulador de potencia 81

4.1. Simulador *FU-Wattch* 83

4.1.1. Características de *Wattch* 84

4.1.2. Aspectos a mejorar en *Wattch* 88

4.1.3. Modelo de las UFs 90

4.1.4. Modificaciones hechas a *Wattch* 92

4.2. Validación del simulador *FU-Wattch* 94

4.3. Comparación de los simuladores *Wattch* y *FU-Wattch* 101

4.4. Conclusiones 103

5. Reducción del consumo en los procesadores superescalares 105

5.1. Estudio de las instrucciones del repertorio *Alpha* 107

5.1.1. Instrucciones del repertorio *Alpha* 107

5.1.2. Instrucciones del repertorio *Alpha* que usan un sumador 111

5.1.3. Número de ciclos en los que se requiere más de un sumador de 64/32-bits 124

5.1.4. Conclusiones 126

5.2. Propuestas para reducir el consumo en las UFs 127

5.3. Entorno de simulación 132

5.3.1. Modelo de las UFs 132

5.3.2. Estimación del consumo en los sumadores 133

5.3.3. Parámetros del procesador modelado 136

5.3.4. *Benchmarks* 139

5.3.5. Métrica 139

5.4. Resultados experimentales 140

5.5. Conclusiones 163

6. Conclusiones y trabajo futuro 167

6.1. Conclusiones 167

6.2. Trabajo Futuro 175

6.3. Publicaciones 180

Bibliografía 183

Índice de figuras 201

Índice de tablas 205

Capítulo 1

Introducción

Año tras año venimos asistiendo a la mejora, tanto en prestaciones como en capacidad, de los sistemas basados en microprocesador. Estas mejoras han venido impulsadas por la constante evolución de la tecnología. Uno de los principales avances es el aumento de la capacidad de integración debido a la reducción de los tamaños mínimos de fabricación . El tamaño mínimo de fabricación delimita la separación mínima que debe existir entre dos transistores dentro de un chip para que el funcionamiento siga siendo correcto. Esta separación se ha reducido en tres órdenes de magnitud desde los años 60, enfrentándonos actualmente con tamaños inferiores a los 65 nm. Es previsible que esta reducción se mantenga durante los próximos años, aunque a un ritmo más suave. Según las estimaciones del *International Technology Roadmap for Semiconductors* [ITR06], se esperan tecnologías (*technology nodes*) de 22 nm para el año 2016.

Esta reducción en los tamaños mínimos de fabricación deja un mayor espacio disponible en el chip del procesador, permitiendo integrar una mayor can-

tidad de transistores dentro de él. Así por ejemplo, la mayoría de procesadores actuales disponen de un primer nivel de memoria *cache*, para instrucciones y datos, de unos 64 KB cada una y de un segundo nivel de cache de al menos 128MB, integrados en el mismo chip, lo que conlleva una disminución de la penalización del acceso a memoria.

Esta tecnología ha sido llamada tecnología de integración, y sigue la tendencia de crecimiento predicha por la ley de (Gordon) Moore (cofundador de Intel Corporation) [Moo65]. Dicha ley nos dice que el número de transistores por chip se duplica aproximadamente cada 18 o 24 meses. De hecho, hoy en día ya se pueden integrar más de 400 millones de transistores en un único chip usando una tecnología de proceso de 130nm [INT04], y todos los informes tecnológicos actuales sobre semiconductores revelan que la evolución futura lleva camino de preservar las nuevas predicciones de Moore para los próximos años y tecnologías [EKRZ04].

Desde el punto de vista de la arquitectura de computadores, el mencionado incremento del número de transistores dentro del chip del procesador, ha permitido incorporar mecanismos cada vez más complejos para mejorar el rendimiento en los procesadores actuales. Ejemplos de estos son las técnicas de planificación dinámica, sistemas de especulación y predicción de saltos, réplica de las unidades funcionales, etc.

Sin embargo, todos estos avances tecnológicos traen asociados una serie de problemas que el diseñador debe tener presente. Aunque la problemática del diseño es diferente para los distintos tipos de procesadores, ya sean procesadores de propósito general como procesadores destinados a sistemas empotrados, es posible establecer una serie de problemas comunes a todos ellos, entre los

que destacan el consumo de potencia y los problemas derivados de la energía, objeto fundamental del trabajo presentado en esta tesis.

Las razones de la importancia de la potencia en el diseño de procesadores son muy variadas:

- **Incremento del consumo de potencia:** El tener cada vez mayor número de transistores por chip implica un mayor consumo de potencia, tanto dinámica como estática. En los procesadores de gama alta, el consumo total actual se encuentra en torno a los 100W, estimándose un aumento hasta los 300W en el año 2016 [ITR06].

- **Aumento del consumo de potencia dinámica:** La potencia dinámica es directamente proporcional a la frecuencia, por lo tanto el aumento de la frecuencia de trabajo de los procesadores implica un aumento del dicho consumo [GBJ98]. En la figura 1.1 se puede ver como cada año aumenta significativamente la frecuencia de trabajo de los procesadores, esto implica un significativo aumento del consumo potencia dinámico.

- **Aumento del consumo de potencia estática:** La reducción del tamaño de integración hace que se agraven los efectos de segundo orden de la tecnología CMOS, lo que implica que las corrientes de *leakage* sean cada vez mayores como se puede ver en la figura 1.2. Esto hace que el consumo estático represente cada vez una proporción mayor del consumo total. En concreto, en la figura 1.3 se puede ver que para tecnologías por debajo de los 70nm el porcentaje de consumo estático respecto al total puede superar el 50 %. [TSC07] [YAE06] [BS00]

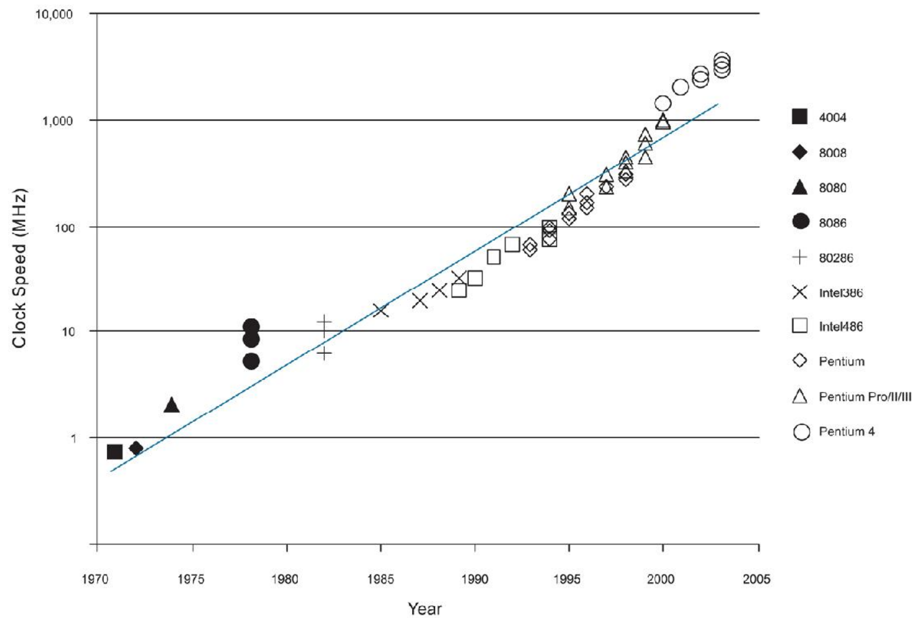


FIGURA 1.1: Frecuencia del reloj de los microprocesadores de INTEL. Fuente:[WH05].

- **Aparición de puntos calientes (*Hot-Spots*):** El aumento de la densidad de potencia (energía disipada por unidad de tiempo y unidad de superficie) alcanza actualmente valores próximos al centenar de W/cm2. Esto está dando lugar a la aparición de zonas en el procesador, que debido a la alta densidad de transistores y al gran número de accesos que en ellas se realizan, generan mayor cantidad de calor del que se puede disipar, aumentando considerablemante la temperatura en esa zona y dando lugar a los llamados puntos calientes (*Hot-Spot*) del procesador [GS04] [PSV05] [RSSS06] [HSS+04] [SSS+04].

- **Deterioro del rendimiento y la fiabilidad:** El aumento de temperatura centrado en una zona del procesador tiene un impacto significativo, por una lado, en la velocidad de los transistores ya que esta disminuye al subir la temperatura lo que produce un deterioro del rendimiento del

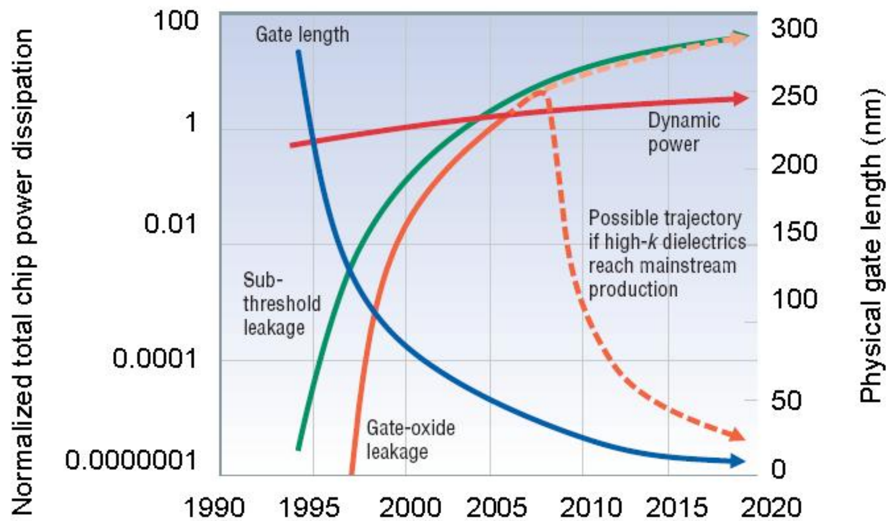


FIGURA 1.2: Tendencia del consumo dinámico y estático basados en *the International Roadmap for Semiconductors*[KAB+03].

procesador. Por otro lado, la elevación de la temperatura puede acarrear la degradación funcional del circuito. Además, las corrientes de *leakage* presentan una fuerte dependencia con la temperatura llegando a ser varios ordenes de magnitud mayor al incrementarse esta, lo que hace incrementar el consumo estático.

- **Aumento del coste**: Niveles altos de potencia hacen necesarios empaquetamientos de los circuitos integrados más resistentes y más sofisticados, y sistemas de refrigeración mas complejos, todo ello con el consiguiente incremento del coste de producción. [RSSS06]

Para establecer los objetivos de este trabajo, en la siguiente sección, se analizarán los problemas y las necesidades que tienen los procesadores de propósito general

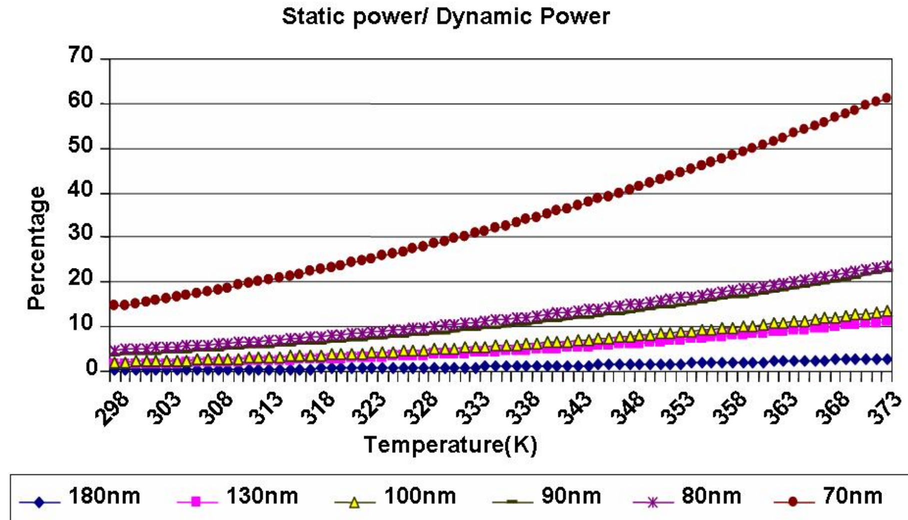


FIGURA 1.3: Tendencia del porcentaje entre el consumo dinámico y estático en función de la tecnología. Fuente: Skadron et al. Universidad de Virginia.

1.1. Procesadores de propósito general

En lo que respecta al diseño de los procesadores de propósito general (GPPs: *General Purpose Processors*) de hoy en día, una gran parte de ellos son procesadores segmentados, con una organización superescalar (capacidad de procesar más de una instrucción simultáneamente en cada una de las etapas), y varios cores o núcleos de procesamiento. La tendencia actual en el diseño de estos procesadores, es una incesante búsqueda de técnicas que permitan incrementar el paralelismo a nivel de instrucción (ILP: *Instruction Level Parallelism*), o lo que es lo mismo, permitir un mayor solapamiento en la ejecución de instrucciones. Para poder sacar partido a todas estas técnicas, los procesadores actuales disponen de mecanismos de planificación dinámica de instrucciones. Es decir, las instrucciones no se ejecutan en el orden en que aparecen en el programa sino que el *Hardware* decide en que orden se ejecutan para obtener un mayor rendimiento. Además, disponen de *caches* de datos e

instrucciones separadas, varios niveles de *caches*, lógica sofisticada de lanzamiento de instrucciones, de un buffer de reordenamiento, de múltiples unidades funcionales, lógica de predicción de salto, etc.

Todas estas mejoras introducidas en el modelo de ejecución fuera de orden han contribuido de manera significativa al incremento de la velocidad, hoy en día los microprocesadores operan a frecuencias muy altas, por ejemplo, el *Intel Xeon Dual Core* opera a una frecuencia de 3 GHz, pero sin lugar a dudas el aumento más espectacular será en número de transistores, que superará los mil millones en el 2012. Pero, como ya se ha comentado, este mayor número y densidad de componentes tiene sus efectos negativos, entre ellos se encuentra el aumento de consumo de potencia y sobre todo la disipación de calor.

Dada la importancia de la potencia en el diseño de los GPP, es necesario analizar que partes del procesador son las que más consumen para poder proponer soluciones. Tomaremos como ejemplo y base de nuestro trabajo la arquitectura *Alpha* [ACC]. Esta arquitectura recoge las técnicas mas avanzadas en los procesadores superescalares y supone una referencia bien conocida para el arquitecto de computadores [HP07]. Además, dispone del simulador mas usado para la investigación de procesadores de alto rendimiento, SimpleScalar [ALE02], que incluye un modelo validado de dicha arquitectura. La tabla 1.1 muestra el porcentaje de consumo que representa cada una de las partes de un *Alpha* 21264 respecto al total del procesador. En ella se puede ver que el reloj representa un tercio del consumo total, y tanto la lógica de lanzamiento, como las *caches* y las unidades funcionales (de enteros y punto flotante) representan cada una de ellas un quinto del consumo total. Además la última generación del *Alpha*, el *Alpha* 21464, aumentó el consumo total del procesador de 90W

a 150W con el consecuente aumento en las partes anteriormente mencionadas, [BTM00] [WM04].

TABLA 1.1: Porcentaje de potencia que consume cada parte del procesador.

Estructura *Hardware*	***Alpha* 21246**
Reloj	34.4 %
Cache	16.1 %
Lógica de lanzamiento	19.3 %
Unidad de ejecución de enteros	10.8 %
Unidad de ejecución de P.F	10.8 %

En la literatura se pueden encontrar multiples propuestas para reducir el consumo de potencia en los GPP. La mayoría de ellas están centradas en la memoria y la lógica de lanzamiento de las instrucciones. Sin embargo, aunque como acabamos de ver, la unidad de ejecución (unidades funcionales de enteros y punto flotante y los registros) es una de las estructuras que más consumen, representando hasta el 20 % del consumo total del procesador y situandose al nivel de las colas de lanzamiento, con su lógica de *wake-up* y las *caches*, el número de propuestas es menor. Además, las unidades funcionales (UFs) presentan una gran tendencia a aumentar el consumo estático debido al impacto de los avances de la tecnología [DKA+02] [YAE06] [TSC07], y son una de las partes del procesador donde pueden aparecer la mayor parte de los puntos calientes [WM04] [PSV05].

De todo esto se deduce que la unidad de ejecución es una de las partes del procesador con mayor interés a la hora de proponer técnicas de reducción del consumo en los GPP. Para evaluar el impacto y la bondad de las posibles propuestas será necesario adaptar los simuladores actuales, ya que estos no

proporcionan datos fiables sobre el consumo en estas estructuras.

1.2. Objetivos principales de este trabajo

Como se ha dicho en las secciones anteriores, en los últimos años se han producido importantes avances en los procesos de integración de transistores. Esto ha hecho aumentar enormemente las prestaciones de los procesadores actuales. Estos avances tecnológicos traen asociados una serie de problemas que el diseñador tiene que tener presentes. El consumo de potencia aparece, entre otros, como uno de los principales problemas que se debe tener en cuenta en todos los aspectos del diseño de los sistemas actuales y desde las primeras etapas. Por todo esto, el trabajo de investigación aquí presentado se centra en la optimización del consumo de potencia en los GPPs.

En la sección anterior se ha visto que la unidad de ejecución es una de las parte del procesador con mayor interés a la hora de proponer técnicas de reducción del consumo en los GPPs. Esto es debido principalmente a tres factores:

- Las UFs (de enteros y punto flotante) son una de las estructuras que más consumen, representando el 20 % del consumo total del procesador. Esto las sitúa al nivel de las las *caches* y las colas de lanzamiento, con su lógica de *wake-up*.

- Presentan una gran tendencia a aumentar el consumo con la aplicación de las nuevas tecnologías ya que son una de las partes del procesador donde mayor impacto tienen los efectos secundarios que aportan los avances de

la tecnología [DKA+02] [YAE06] [TSC07].

- Las UFs son de las partes del procesador donde se encuentran la mayor parte de los puntos calientes del procesador [WM04] [PSV05].

Por otro lado, las herramientas existentes no están preparadas para obtener medidas de consumo en la unidad de ejecución. Todo esto nos ha llevado a plantearnos los siguientes objetivos:

- Analizar las herramientas de simulación existentes que se utilizan para evaluar el consumo de potencia en los GPPs.

- Adaptar los simuladores para que incluyan un modelado correcto de la unidad de ejecución, modelando correctamente las diferentes unidades funcionales (multiplicador y sumador de enteros y punto flotante, unidad lógica, etc).

- Analizar los repertorios de instrucciones de la arquitecturas más habituales para poder obtener conclusiones que permitan proponer nuevas técnicas.

- Definir técnicas que reduzcan el consumo en las UFs y prevengan la aparición de puntos calientes en los GPPs.

- Analizar el impacto de las distintas propuestas mediante los simuladores adaptados.

1.3. Organización de este trabajo

El resto de esta tesis se ha organizado de la siguiente manera. En el capítulo 2 se hace una breve revisión de las herramientas de diseño automático usadas en la minimización el consumo de potencia tanto de procesadores de proposito general como de sistemas empotrados de altas prestaciones. En primer lugar se hace un análisis de las métricas utilizadas para evaluar la bondad de las optimizaciones y a continuación se estudian algunos modelos de estimación aplicados en el diseño, diferenciando entre los distintos niveles de abstracción. Finalmente se analizan más en profundidad las herramientas de simulación más utilizadas a la hora de diseñar nuevas arquitecturas (*Simplepower*, *Timer Power*, *Wattch*, etc..).

El capítulo 3 presenta una revisión bastante completa de las técnicas existentes para la optimización del consumo en los procesadores de propósito general. En esta revisión se verán técnicas aplicadas a las estructuras de los procesadores actuales que mayor consumo representan centrándose principalmente en las UFs para los GPPs.

En el capítulo 4 se presenta una versión del simulador *Wattch*, que llamaremos *FU-Wattch*, a la cual hemos añadido algunas modificaciones para estimar el consumo en las UFs con mayor precisión.

En el capítulo 5 se proponen técnicas a nivel *Hardware* para reducir el consumo en las UFs de en los GPPs. Se analiza el repertorio de instrucciones de la arquitectura *Alpha*, se explica el entorno de simulación y las adaptaciones realizadas, y se describen todos los detalles de las técnicas mostrando los resultados.

Por último en el capítulo 6 se presentan las conclusiones y trabajo futuro. En él se describirán y resumirán con claridad las aportaciones de esta tesis doctoral y se enumerarán las publicaciones derivadas de este trabajo.

Capítulo 2

Herramientas y metodologías de simulación para evaluar el consumo

Como acabamos de ver en el capítulo 1, el consumo energético, así como el calor disipado por un procesador, han ido creciendo en importancia como factor a la hora de diseñar un procesador con tecnologías CMOS. Es por lo tanto interesante tener una estimación de la potencia disipada por un procesador a la hora de diseñarlo, además de poder comparar las distintas opciones de diseño para elegir la más adecuada teniendo en cuenta el consumo. En este capítulo pretendemos dar una visión general de las ideas básicas en consumo de potencia y analizar las herramientas de simulación más utilizadas.

El resto del capítulo está organizado como sigue: la sección 2.1 repasa brevemente las fuentes de consumo en los diseños CMOS. La sección 2.2 describe las métricas más importantes. La sección 2.3 presenta un repaso de los

simuladores existentes para la estimación del consumo, centrándonos en los simuladores más utilizados como son los basados en *SimpleScalar*. Y por último, en la sección 2.4 se expondrán algunas conclusiones.

2.1. Fuentes de potencia en los diseños actuales

El consumo en un procesador proviene de dos fuentes, el consumo estático y el consumo dinámico.

- **El Consumo Estático** es el consumo que se produce debido a corrientes de fuga (*leakage*) existentes en los transistores. Siempre existe, incluso cuando el circuito está inactivo. Con el avance de la tecnología este componente de la potencia es cada vez más importante, especialmente para los diseños de altas prestaciones cuya capacidad de integración está dentro de la clasificación DSM (*Deep Sub-Micron*). El valor de este consumo depende de características de la tecnología que se emplea, el número de transistores y la temperatura de funcionamiento del circuito.

  La potencia estática $P_{estatica}$ se define como el producto del voltaje de la fuente de alimentación V_s por la corriente estática del circuito i_0. Todo esto viene recogido en la ecuación(2.1):

$$P_{estatica} = \sum_{1}^{n} i_0 * V_s \; ; \quad i_0 = i_s \left(e^{\frac{qV_{\text{diodo}}}{KT}} - 1 \right) \tag{2.1}$$

  Donde i_s es la corriente inversa de saturación, o corriente de fuga, de los diodos, V_{diodo} es el voltaje del diodo, q es la unidad de carga $(1{,}602 *$

$10^{-19}C$), K es la constante de Boltzmann ($1{,}38 * 10^{-23} J/K$) y T es la temperatura.

- **El Consumo Dinámico** se produce debido a la carga y descarga de la capacidad de los transistores y las conexiones, y depende de la actividad del circuito. Es decir, ocurre únicamente durante las transiciones, cuando las puertas están conmutando. Por lo tanto es proporcional a la frecuencia de conmutación y cuanto mayor sea el número de conmutaciones mayor será el consumo de potencia dinámico. La ecuación(2.2) representa a la potencia dinámica.

$$P_{dinamica} = a * C * f * V_s^2 \tag{2.2}$$

Donde a es la actividad de conmutación, C es la capacidad en cada nodo que conmuta, f es la frecuencia de reloj y V_s es el valor del potencial de alimentación

La potencia dinámica tiene dos componentes, como muestra la ecuación(2.3): la potencia de conmutación (*crowbar*) y la de carga (*load*). La primera es debida a las corrientes que van desde la fuente de alimentación a tierra cuando el transistor cambia de estado, mientras que la de carga se debe a la corriente necesaria para cargar las capacidades de los elementos conectados a la salida. Cuando se diseñan circuitos ASIC estas potencias se reagrupan en Potencia de celda (*cell*) y de carga (*load*)

$$P_{dinamica} = P_{conmutacin} + P_{carga} \tag{2.3}$$

2.2. Métricas

La comparación de los procesadores en términos de consumo no es una tarea fácil y además es necesario encontrar una métrica que realice una comparación honesta. Por ejemplo, la potencia sin más no es una buena medida ya que depende de la frecuencia del procesador. Podríamos por lo tanto reducir el consumo de potencia de un sistema sin más que reducir su frecuencia pero esto sin embargo no nos llevaría necesariamente a un mejor procesador.

Otra posible métrica es la Energía medida en Julios/instrucción o la inversa SPEC/W [GH96]. Aunque esta opción es mejor que la anterior sin embargo también tiene sus problemas. En este caso es proporcional a CV^2, por lo que se puede reducir la energía por instrucción simplemente reduciendo el voltaje de alimentación o la capacidad utilizando transistores de menor tamaño. Ambos cambios implican un aumento en el retardo de los circuitos, por lo que los circuitos con menor consumo pueden ser también los de menor rendimiento. En realidad lo que se busca siempre es el menor consumo de potencia a un retardo dado. Por ello una característica importante es el producto retardo potencia DP, que se expresa en julios y se define como el producto del valor medio del retardo de propagación multiplicado por el valor medio de la disipación como recoge la ecuaciòn(2.4).

$$DP = t_D * P_D \ ; \quad t_D = \frac{t_{PHL} + t_{PLH}}{2} \ ; \quad P_D = \langle P_{dinamica} + P_{estatica} \rangle \tag{2.4}$$

Donde t_D es el tiempo de propagación, P_D es la potencia disipada, y t_{PHL} y t_{PLH} son los tiempos de propagación al conmutar de 1 a 0 y viceversa.

Cuanto menor sea este producto para un circuito, más se acercarán sus características a las de un elemento lógico ideal [Hor96].

En [GH96] se presentó el producto Energía-Retardo (EDP) como una forma efectiva de medir la disipación de potencia teniendo una cierta restricción en el rendimiento. La unidad es Julios/SPEC o su inversa $SPEC^2$/W. De esta forma se podían comparar los sistemas de una manera más apropiada, ya que para mejorar el EDP es necesario o aumentar el rendimiento, o reducir la energía sin afectar al otro factor, o bien hacer ambas cosas. Muchas de las técnicas de diseño de bajo consumo no reducían este factor, sino que buscaban un equilibrio entre rendimiento y consumo. Por ejemplo reduciendo el voltaje de alimentación, se puede reducir la energía y el rendimiento en un orden de magnitud y el EDP se ve afectado mínimamente. Una forma efectiva de reducir el EDP es utilizar una tecnología más pequeña. Si el factor de escala es λ, bajo condiciones ideales el EDP escala con λ^4 [GM94], sin embargo la mayoría de las tecnologías no escalan idealmente porque el potencial de alimentación no lo hace [CSBP94]. Además el rendimiento total del sistema siempre está limitado por la memoria externa. De acuerdo a [GH96] el EDP escala en λ^2 y de esta forma se puede hacer una comparación más precisa. De esta forma la eficiencia energética de un procesador es tremendamente dependiente de la eficiencia de la tecnología con la que esté construido. Por ello es posible comparar distintas tecnologías sin tener en cuenta el factor de escala.

Cuando se propuso el EDP como medida de la potencia, la tecnología estaba en una fase de desarrollo muy diferente a la actual. En estos momentos la fabricación es DSM (*Deep SubMicron*) y aparecen otros efectos que hay que considerar a la hora de aplicar este tipo de métricas. Esto no significa que no sea

válida sino que se debe ser cuidadoso con la forma de obtenerla. Por ello Lee *et ál.*, en [LFDD03], plantean un par de preguntas antes de decidir la métrica, las preguntas son: ¿qué energía vamos a medir? y ¿es necesario utilizar *Hardware* adicional?. En algunas ocasiones puede ser nociva para el funcionamiento del resto del circuito la incorporación de elementos nuevos para la reducción del consumo de potencia. Por ejemplo, si se rediseña un disco con tecnología *Flash* que gana un 5 % en el consumo, esto significará aproximadamente un 0.5 % del consumo total del sistema, sin embargo optimizando los patrones de acceso se puede conseguir este mismo beneficio pero sin necesidad de rediseñar nuevo *Hardware*. Hay que analizar por tanto que optimizaciones tienen sentido.

Para saber cuando es efectivo un cambio en [LFDD03] se propone la ecuación(2.5) denominada Producto Energía Retardo Completo(CEDP):

$$[1 - R_{sys}(CPU) * R_{CPU}(u) * R_{saved}(u)](1 + \frac{\Delta D}{D}) \leq 1{,}0 \qquad (2.5)$$

donde $R_{sys}(CPU)$ es la energía disipada por la CPU con respecto al sistema total. $R_{CPU}(u)$ es la proporción de energía consumida por una unidad funcional u antes de cualquier optimización con respecto a la CPU y $R_{saved}(u)$ es la proporción de energía ahorrada por la unidad funcional u al realizar una modificación que le afecte. $\frac{\Delta D}{D}$ es el retardo que añade esa modificación si existiera. Obviamente si el retardo es menor y la energía también está formula no es necesaria.

Para esta métrica, se establece el valor del CEDP de referencia como 1. Si un diseño tiene un CEPD menor que 1, será eficiente en términos de energía. Podemos elegir un elemento que suponga el porcentaje $R_{CPU}(u)$ del consumo

total del sistema y calcular lo que se ahorra y el incremento del tiempo total de ejecución, y por lo tanto $\frac{\Delta D}{D}$. Si se cumple la ecuación(2.5), el diseño será eficiente en términos de reducción de energía. Esta formula se puede particularizar para el caso en el que haya una batería única para la CPU (que no suele ocurrir) y obtener la ecuación(2.6).

$$
\begin{aligned}
[1 - R_{CPU}(u) * R_{saved}(u)] \left(1 + \frac{\Delta D}{D}\right) &\leq 1{,}0 \\
\left(\frac{\Delta D}{D}\right) &\leq \left(\frac{R_{CPU}(u) * R_{saved}(u)}{1 - R_{CPU}(u) * R_{saved}(u)}\right)
\end{aligned}
\tag{2.6}
$$

También se puede obtener de forma gráfica y consultarla para los diseños que se realicen. Otro problema a tener en cuenta en cuanto a la energía ahorrada es el problema de las corrientes de *leakage*. Para tecnologías anteriores a la micra se podían desestimar, sin embargo con el avance de las tecnologías DSM, la disipación de energía estática debe ser una variable que deben incluir los modelos de estimación del consumo actuales. Para ello se puede estudiar la viabilidad de un diseño desde un punto de vista estrictamente energético reduciéndolo a cuatro variables para redefinir el producto energía retardo según la ecuación(2.7) [LFDD03] y evaluar así si merece la pena incluir nuevo *Hardware* en el sistema: la nueva actividad de conmutación (a_{new}), el cambio en el número de transistores (ΔT), el cambio en la frecuencia (ΔF) y el cambio en el retardo (ΔD)

$$
\frac{a_{ref}}{a_{new}} \geq \left(1 + \frac{\Delta T}{T} + \frac{\Delta F}{F}\right) * \left(1 + \frac{\Delta D}{D}^2\right) \tag{2.7}
$$

Esta ecuación se puede considerar valida si se cumple que $\Delta T << T$, $\Delta F << F$ y las corrientes de *leakage* despreciables. Aunque las dos primeras se cumplen casi siempre hay que tener cuidado con la tercera condición ya que para tecnologías DSM esto no siempre sucede y se pueden cometer errores de hasta un 15 % en las estimaciones. El estudio completo se puede seguir utilizando [WE93] para las ecuaciones de los transistores y de las corrientes parásitas y [LFDD03] para ampliar información y ver algunos ejemplos de situaciones conflictivas.

En conclusión a la hora de elegir una métrica y un modelo hay que tener en cuenta que el *Hardware* adicional puede consumir potencia por corrientes de *leakage* importantes, reduciendo de esta forma la vida media de las baterías incluso en estados de descanso o inactividad. Incluso cuando la actividad de conmutación no se vea afectada, el *Hardware* nuevo puede ser un problema para el sistema total y por lo tanto cualquier investigación de arquitecturas de bajo consumo debe tener en cuenta la energía estática a la hora de evaluar las posibles mejoras.

2.3. Simuladores

Existen numerosas herramientas de diseño automático para minimizar el consumo de potencia. Algunas están diseñadas específicamente para el campo de la potencia mientras que otras tienen un carácter más general y se usan sobre todo para otras optimizaciones. Además, existen distintos niveles de abstracción en los que se utilizan las herramientas de análisis. En esta sección se hace una revisión en primer lugar de los distintos tipos de simuladores que po-

demos encontrar para, a continuación, estudiar las herramientas de simulación más utilizadas a la hora de diseñar nuevas arquitecturas.

2.3.1. Clasificación de los simuladores

Dentro de la amplia variedad de simuladores, existe un convenio generalizado acerca de las principales técnicas de simulación de sistemas de cómputo [CLSL02]. De este convenio 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. En la tabla 2.1 se presenta a modo de resumen esta clasificación de los simuladores.

TABLA 2.1: Clasificación de los simuladores.

Factor considerado	Tipo de simulador
El alcance o ámbito	De conjunto de instrucciones De sistemas completos
El tipo de entrada	Basados en trazas Basados en ejecución
El nivel de abstracción	Funcionales Arquitectónicos

Observando la tabla 2.1 y centrándonos en el alcance o ámbito del simulador podemos ver que las herramientas de simulación pueden ser de dos tipos: simuladores de conjuntos de instrucciones y simuladores de sistemas completos.

- **Los simuladores de conjuntos de instrucciones**, conocidos también como simuladores de microarquitectura, simulan únicamente el repertorio de instrucciones de un procesador o microcontrolador. Este tipo de simuladores suelen ser codificados con lenguajes de alto nivel, de modo

que imitan la funcionalidad del procesador manteniendo los valores de sus registros en variables internas. En estos simuladores una ejecución consiste en ir leyendo las instrucciones de entrada y realizando cambios en las variables internas según la instrucción leida.

Utilización: las principales aplicaciones 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.

Ventaja: la principal ventaja es su velocidad de ejecución, puesto que que suelen tener en cuenta, principalmente, la funcionalidad de cada instrucción del repertorio simulado, no los detalles *Hardware* que están modelando.

Inconveniente: estos simuladores no modelan partes importantes de un sistema de cómputo como la memoria, los buses de comunicación o las llamadas al sistema operativo.

- **En los simuladores de sistemas completos**, como su nombre indica, se incluye el modelado de los elementos principales de un sistema de cómputo. Estos elementos son, generalmente, el procesador, la memoria y el soporte necesario para poder hacer llamadas al sistema operativo.

  Utilización: generalmente se utilizan para el estudio tanto de la funcionalidad como del rendimiento de sistemas de cómputo, puesto que la implementación de este tipo de simuladores suele modelar con mayor detalle el *Hardware* bajo estudio.

  Ventaja: estos simuladores permiten, por un lado la monitorización de

los elementos del sistema, debido a que modelan con mayor detalle el *Hardware*, y por otro la generación de estadísticas acerca del rendimiento del sistema.

Inconveniente: presentan mayor tiempo de ejecución en las simulaciones debido al mayor nivel de detalle y elementos modelados. Por tanto, es importante la optimización del propio código de este tipo de simuladores.

Si nos fijamos en el tipo de valores de entrada de las simulaciones los podemos clasificar en: simuladores basados en trazas y simuladores basados en ejecución.

- **Los simuladores basados en trazas** toman como valores de entrada un flujo de instrucciones previamente ejecutado en la misma arquitectura, 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 tomada 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.

  Utilización: estos simuladores se aplican, principalmente, 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.

  Ventaja: debido a la cantidad de información que contienen las trazas, estos simuladores pueden simplificar la ejecución de las simulaciones.

Por ejemplo, omitiendo la simulación de las instrucciones que van a ser desestimadas, o eliminando el modelado de estructuras internas de la microarquitectura, como los buses o registros

Inconveniente: el principal inconveniente de los simuladores basados en trazas es que dependen en gran medida de la información contenida en la traza utilizada como entrada. Las trazas completas provienen de la ejecución de bancos de pruebas estándar y pueden llegar a tener un elevado tamaño, llegando a veces a ocupar varios gigabytes de disco. Esto, además de toda la memoria que ocupa, relentizará la ejecución. 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 la ejecución. Estas reducciones sobre la traza completa introducen un sesgo en los resultados que pocas veces es estudiado o tomado en consideración a la hora de presentar resultados. Por todo esto, es importante la obtención de trazas representativas que tengan un compromiso entre el tiempo de ejecución y la cantidad de información que contienen.

- **Los simuladores basados en ejecución** reproducen con mayor o menor detalle el funcionamiento interno de los componentes del sistema. Estos simuladores requieren el modelado tanto de la microarquitectura como del repertorio de instrucciones simulado. La precisión de las estadísticas obtenidas, asi como la monitorización de estructuras, dependen del nivel de detalle con el que se modela la microarquitectura. Cuanto mayor sea este, mayor potencial se tendrá para monitorizar y obtener

estadísiticas, eso si, a costa de una mayor velocidad de ejecución de las instrucciones.

Utilización: debido a que reproducen, con mayor o menor detalle, el funcionamiento interno de los componentes del sistema, este tipo de simuladores se usa cuando se quiere obtener información acerca del rendimiento del sistema simulado.

Ventaja: estos simuladores permiten tanto el acceso a los datos procesados a lo largo de la simulación como la monitorización de las estructuras modeladas. Asi, es posible generar distintos tipos de estadísticas y métricas acerca de cualquier parámetro de rendimiento del sistema simulado. La reproducción del funcionamiento de los componentes del sistema modelado permite la ejecución completa de cualquier programa de prueba, sin necesidad de reproducir y almacenar enormes archivos de trazas.

Inconveniente: la principal desventaja de estos simuladores es la velocidad de simulación. Por tanto, es importante tener un compromiso entre el del nivel de detalle con el que se modela la microarquitectura y la velocidad de ejecución.

Y por último, mirando la tabla 2.1 podremos ver que los simuladores también se pueden clasificar atendiendo al nivel de abstracción, entendido 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 su temporización. El nivel de abstracción también se denomina granularidad del simulador. Dentro de esta clasificación podemos distinguir dos tipos de simuladores: simuladores

funcionales y simuladores arquitectónicos.

- **Los simuladores funcionales** son de granularidad gruesa. Es decir, la especificación del *Hardware* es sustituida por una descripción de la funcionalidad del circuito.

  Utilización: estos simuladores son muy adecuados para realizar comprobaciones acerca del correcto funcionamiento del circuito, debido principalmente a su alta velocidad.

  Ventaja: debido a que sus estructuras están descritas a nivel funcional, estos simuladores presentan una alta velocidad de ejecución, siendo esta una de sus mayores ventajas.

  Inconveniente: debido al bajo nivel de detalle no es posible obtener información precisa acerca de los retardos, latencias o cualquier otro parámetro que dependa de la implementación concreta de un circuito, puesto que estos detalles son obviados por el simulador.

- **Los simuladores arquitectónicos** son de granularidad fina. Es decir, se modela con gran detalle el comportamiento de los componentes *Hardware* del sistema simulado.

  Utilización: estos simuladores se usan cuando se quiere obtener medidas precisas de rendimiento, latencias o cualquier otro parámetro que dependa de la implementación concreta de un circuito.

  Ventaja: gracias a la detallada descripción del sistema, estos simuladores suelen ser capaces de emular la ejecución real del *Hardware* modelado. Así, este tipo de simuladores permite obtener gran cantidad de

información acerca del computo ocurrido en cada una de las estructuras modeladas.

Inconveniente: la principal desventaja de estos simuladores es la velocidad de simulación, ya que debido a su bajo nivel de abstracción la ejecución puede resultar muy lenta. De hecho, las simulaciones de muy baja granularidad (nivel de puertas lógicas) no suelen realizarse en circuitos completos, puesto que su ejecución puede resultar muy costosa en tiempo.

A continuación, haremos una revisión de los simuladores que existen en investigación para evaluar el consumo de un procesador, con el objetivo de encontrar el simulador más adecuado para evaluar las técnicas que se proponen en este trabajo.

2.3.2. Simuladores más utilizados

Actualmente existen distintos tipos de herramientas que permiten modelar el consumo de un procesador. Para evaluar nuevos diseños y realizar comparativas es necesario aplicar un modelo de estimación. Dicho de otra forma, debemos caracterizar cada una de las partes del sistema para la cual queramos estimar el consumo de potencia. Para realizar una taxonomía de los distintos modelos podemos hacerlo según el nivel de descripción que estén trabajando y clasificarlos como de bajo, medio y alto nivel.

- **Los modelos de bajo nivel** se basan en el estudio de las capacidades de difusión, de puerta y de las líneas a partir de las descripciones completas del circuito o del *layout*. Se realiza una simulación analógica

utilizando modelos muy detallados de los dispositivos y resolviendo una gran cantidad de ecuaciones para obtener una precisión muy elevada. Son modelos capaces de obtener valores tanto de la potencia estática como de la dinámica con un margen de error muy pequeño. Todo esto conlleva un coste computacional elevado y hace que sólo sean prácticos para diseños de hasta 100K transistores. Los modelos más utilizados son los de los simuladores *HSpice* y *PowerMill* (*Sypnosys*) que es unas diez veces más rápido que el primero.

La sintaxis para escribir los archivos *HSpice* es la misma que en PSPICE [PSp], la única diferencia es que se debe escribir a mano el archivo de netlist para la simulación y no se dispone del editor de esquemáticos y la sonda como en *Pspice*. Por su parte *PowerMill* [Syp] viene incorporado con el conjunto de herramientas de diseño de *Sypnosys* y es bastante similar. Para acelerar el cómputo utiliza un algoritmo de partición que divide en etapas el circuito original y calcula la corriente y el voltaje simultáneamente. También utiliza ficheros de tecnología precaracterizados y una herramienta de estimación basada en algoritmos genéticos que evalúa la potencia total. Soporta también las características de SPICE en algunos análisis. Los formatos de entrada incluyen Verilog y EDIF entre otros. *PowerMill* es un conjunto de aplicaciones entre la que destacamos Low Power Design. Esta aplicación permite poner límites de potencia, correr una simulación y comprobar si se ha producido una violación tanto en potencia estática como en dinámica.

- **Los modelos de nivel medio** trabajan a nivel RTL. Para ello definen

estructuras en VHDL o *Verilog* y calculan el consumo. Un ejemplo de este nivel de abstracción se puede encontrar en [AVGDA02]. En dicho artículo se explica una metodología simple para calcular el consumo dinámico. La idea fundamental es utilizar eventos en VHDL para detectar la actividad de los dispositivos lógicos. Durante una simulación un dispositivo lógico se puede estimular cambiando los niveles lógicos de sus puertos de entrada. Utilizando los procesos VHDL se puede recoger todas las transiciones del circuito y utilizar este dato y la ecuación 2.2 para establecer el consumo de potencia dinámico. El principal defecto de esta metodología es que no tiene en cuenta el consumo estático y como ya se ha dicho para las tecnologías actuales es impensable obviar el efecto de las corrientes de *leakage*.

Afortunadamente, muchos componentes de un procesador son estructuras regulares, por ejemplo, las memorias *cache*. Algunos simuladores aprovechan esta característica para poder reducir el tiempo de simulación. Este es el caso de CACTI [DSN], que calcula los principales parámetros de memorias *cache*, como son tiempos de acceso, tiempos de ciclo, área de silicio, y consumo, tanto estático como dinámico. La principal ventaja que ofrece CACTI es que en un tiempo corto, y solamente a partir de parámetros arquitectónicos de la memoria *cache* (tamaño, asociatividad, número de puertos, etc ...), es capaz de ofrecer resultados con márgenes de error dentro del 10 % del resultado que se obtendría con SPICE. Sin embargo, el hecho de que no modele un procesador completo, y no mida la actividad del procesador al ejecutar una carga de trabajo representativa, hace que no sea el simulador más adecuado para calcular

el consumo de procesadores.

- **Los modelos de alto nivel** más utilizados para el cálculo de consumo están basados en la herramienta de simulación *SimpleScalar* [BA97]. Este es un simulador arquitectónico que se ha convertido en la principal herramienta de simulación y modelado de sistemas de alto rendimiento, como dato cabe decir que en el año 2000 una tercera parte de los artículos científicos presentados en conferencias de arquitectura de computadores internacionales utilizaban *SimpleScalar* para evaluar los resultados obtenidos, y esta tendencia ha llegado prácticamente al 70 % en muchas de ellas en el año 2003.

  A continuación se describen algunos de estos simuladores:

  - **El modelo de *Cai-Lim***, basado en *SimpleScalar*, concretamente en el *out-of-order*. Para hacer la simulación divide el sistema en 17 estructuras *Hardware*, que a su vez se dividen hasta un total de 32 bloques funcionales [GG01] [CL99]. Hace una caracterización de cada bloque tanto en área como en densidad de potencia. La densidad de potencia la divide en cinco tramos, en función de qué parte sea la que la causa. Estas posibles fuentes son la potencia dinámica, la estática, potencia PLA, el consumo debido al reloj y las secciones de memoria. Dentro de cada bloque *Hardware* se buscan partes activas e inactivas.

    Las estimaciones de área se basan en diseños públicos completados y se complementan con área de reloj, conexiones y de la fuente de alimentación. Por su parte la densidad de potencia se basa en simu-

laciones de SPICE para procesos de 0.25um. También se considera que cuando una parte está inactiva tendrá un consumo aproximado del 10 % del total. Con los números de densidad de potencia y los informes de actividad del *SimpleScalar* se calcula el consumo total. Para ello contabiliza cómo se utiliza cada estructura *Hardware* y evalúa el número de accesos. Cabe destacar que distingue entre tres tipos diferentes de accesos. Por ejemplo la *cache* de primer nivel cuenta los accesos lógicos de la *cache* y los accesos a líneas de etiquetas que se utilizan cuando hay un acceso, un reemplazamiento o una invalidación de la *cache*.

- ***Wattch***, está basado en *SimpleScalar* 3.0, concretamente en la herramienta *sim-out-of-order* e implementa un *pipeline* de 5 etapas. Modela el consumo de potencia producido durante la ejecución para cuatro situaciones distintas de activación o funcionamiento [BTM00]. El primer modelo, supone que todas las partes del sistema están activas en todo momento. El segundo considera que se produce un consumo del 100 % cuando hay un acceso y 0 % cuando no lo hay. La tercera forma de considerar el sistema es estimando que no hay consumo cuando no hay acceso a una parte y un consumo lineal cuando si lo hay. Finalmente La cuarta forma de considerar el sistema es estimando un 10 % de consumo cuando no hay acceso a una parte y un consumo lineal cuando si lo hay.

  Para modelar el consumo de cada una de las estructuras del procesador simulado, *Wattch* las organiza en cuatro clases de bloques:

el primero son las estructuras en array que incluyen las *caches* de datos e instrucciones, los arrays de etiquetas en la *cache*, todos los registros, RAT (*Register Alias Tables*), predictores de saltos y una gran parte de la lógica de la ventana de instrucciones y de la cola de *load/store*. Las memorias totalmente asociativas direccionables por contenido constituyen el segundo grupo que incluye la lógica de *wake-up* de la ventana de instrucciones y del *buffer* de reordenamiento, los módulos de comprobación del orden de load y store, o los TLBs entre otros. El tercer tipo de bloque modelado es el referente a la lógica combinacional y cableado, donde estarían las unidades funcionales, la lógica de selección de la ventana de instrucciones, el comprobador de dependencias y los buses de resultado. Por último se modelan las señales de reloj (incluyendo *buffers*, líneas de reloj y cargas capacitivas).

Los factores de actividad para calcular el consumo de potencia se miden ejecutando cargas de trabajo en el simulador de la arquitectura (i.e. *SimpleScalar*). Para los subcircuitos con los que no se puede medir la actividad con el simulador, se supone una actividad de base de 0.5. Además el modelo de alto nivel selecciona estructuras del procesador para reducir el factor de actividad mediante *clock gating*.

- ***Estima*** es un estimador de potencia, área y latencia para bancos de registros segmentados y multi-puerto [BPN03]. Un ejemplo de este tipo de estructuras son las *caches* de datos y de instrucciones,

los arrays de etiquetas en las *caches*, los predictores de saltos, RATs, etc... Al ser estructuras bastante uniformes se prestan al modelado. *Estima* incluye operaciones segmentadas. Además, este simulador incluye el consumo de potencia debido a las líneas de reloj, lo cual es un factor importante, especialmente cuando se usa segmentación. *Estima* permite también incluir restricciones temporales seleccionadas por el usuario y utiliza una definición del tamaño de los dispositivos basado en la simulación a nivel circuital y que es independiente de la biblioteca de dispositivos. De esta forma incorpora las restricciones temporales definidas por el usuario y obtiene el tamaño de los transistores de paso, los dispositivos de precarga, etc... El modelo básico de consumo que utiliza es un modelo por acceso. Calcula la energía por acceso en operaciones de lectura/escritura en un único puerto. Este número junto con los factores de actividad y la frecuencia de reloj nos da el consumo de potencia separando para lecturas y escrituras. Los parámetros tecnológicos vienen definidos por las dimensiones físicas del array que estamos simulando y el tamaño de los dispositivos. También incluyen modelos para las dimensiones físicas (el área) y la latencia.

◦ ***Myrmigki*** es un simulador específico para microcontroladores y microprocesadores [SMH01]. Para sistemas empotrados se simula sobre la base del HITACHI SH3, aunque es extensible a otros sistemas por su diseño modular. Los módulos se encuadran en 3 grupos: procesador (CPU, *cache on chip* , periféricos *on chip*), Memoria *off chip* e *interface* de comunicaciones RS232. El sistema *Myrmig-*

ki permite establecer diversas configuraciones de reloj, de memoria *cache*, etc. Evidentemente cuanto mayor es el nivel de detalle mayor es el tiempo de simulación.

En la estimación del consumo no incorpora las transiciones de señal que se producen en la lógica de control o dentro de cada unidad funcional. Lo que hace es realizar recuento del número de transiciones para una determinada carga de trabajo. También realiza una estimación del consumo para cada tipo de instrucción y configuración del sistema. La estimación de potencia por instrucción proporciona una estimación de la corriente media que consume el procesador y el subsistema de memoria, cuando se ejecuta una instrucción específica. Para obtener los datos del modelo se lanza una ejecución de 100 instrucciones idénticas en una tarjeta de evaluación del procesador y se mide la corriente media en estas simulaciones. Todo esto se puede realizar con seis niveles de detalle de simulación distintos, que van desde solo simulación funcional (*Fast Functional*) hasta los que añaden un modelado de las instrucciones en el *pipe* y modelan la latencia de la ALU y la memoria.

- ***Simple Power*** [CIB01] es una herramienta de simulación basada en la ejecución de programas (*execution-driven*) y que suministra información del consumo en cada ciclo de reloj (*cycle-accurate*). Funciona a nivel de transferencia entre registros (RTL) y utiliza modelos de energía sensibles a las transiciones. Está basado en un *pipeline* de 5 etapas Tiene 5 componentes que permiten una si-

mulación ciclo a ciclo. Estos componentes son: (1) *Simple Power cores* (SP *cores*), (2) RTL *power estimation interface*, (3) Tablas de la capacidad dependientes de la tecnología, (4) simulador de *caches*/buses y (5) el *Loader*. En cada ciclo SP *cores* simula la ejecución de todas las instrucciones activas y llama al *interface* de estimación de potencia hasta que se encuentra la instruccion *halt*. Una vez encontrada esta instrucción finaliza lo que hubiera dentro del *pipeline*. Las tablas de capacidad permiten adaptarlo a las distintas tecnologías. El simulador de *cache* simula tanto la *cache* de instrucciones como la de datos con un error entorno al 2.4 %. Por su parte el simulador de bus permite simular el bus de direcciones de la *cache* de instrucciones y de la *cache* de datos, el bus de datos de la *cache*, obtener el número total de accesos y el número total de transacciones de los buses.

Simple Power, para modelar el consumo, clasifica las distintas partes del procesador en dos tipos: unidades funcionales dependientes del bit y unidades funcionales independientes del bit. Las unidades funcionales dependientes son aquellas en las que la conmutación de un bit afecta al resto de bits de la operación. Ejemplos de estas unidades son los sumadores, multiplicadores, decodificadores, multiplexores, etc. Para este tipo de unidades funcionales se utiliza una tabla en la que encontramos el valor previo, el valor actual y la capacidad resultante. Esta tabla tiene el inconveniente de que es difícil de comprimir. Si la unidad funcional es muy compleja se usa un modelo de simulación analítica independiente de las transiciones,

o si se puede, se divide la unidad funcional en submódulos.

Por otro lado, las unidades funcionales independientes son aquellas en los que la conmutación de un bit no afecta al resto de bits de la operación. ejemplos de estas unidades son los registros del *pipeline*, la unidad lógica de la ALU, etc. Para estas unidades funcionales se calcula la capacidad por bit.

- **El simulador *Step-Power*** hace una estimación del consumo de potencia en cada ciclo, de ahí su nombre [EEAS02]. Este simulador no cuenta el ruido de conmutación y utiliza el modelo de *Wattch* que supone un consumo lineal cuando está activa esa zona y un 10 % de consumo cuando no se utiliza. Además, cuando hay un estado de espera se evalúa un consumo del 20 % debido a la lógica combinacional. En el módelo de rejoj incluye los *buffers* globales o repetidores de señal, el tronco del árbol de reloj y el generador de reloj que no están incluidos en las versiones anteriores de *Wattch*.

  Step-Power nos da información sobre la potencia máxima, la media, la mínima en cada ciclo. Esta información la proporciona en forma de histogramas. También realiza una adaptación del modelo usado de *Wattch* en el que supone que la potencia de reloj se consume toda en cada ciclo. Este es un modelo que está en desarrollo y que debe incorporar la estimación del consumo debido a los fallos en la predicción y en las distintas variaciones de acceso a la memoria *cache*.

- ***Kim-Austin-Mudge*** [KAMG02] presentan un modelo de estima-

ción basado en *SimpleScalar* y diseñado en la Universidad de Michigan. El enfoque de este modelo consiste en trata de resolver los problemas de los simuladores por ciclos que omiten el modelado de los movimientos de datos, tanto en los buses externos como en los internos que conectan bloques de la micro-arquitectura. La metodología debe en primer lugar conocer o estimar los parámetros de la tecnología como los voltajes de alimentación, la capacidad por área por unidad de longitud y la resistencia de hoja de los materiales de interconexión. Para cada bloque de la arquitectura se debe especificar el estilo de diseño y la frecuencia de funcionamiento. A partir de aquí se construyen los modelos de potencia y aportando las estadísticas de acceso y transición de los datos se obtienen las estimaciones definitivas del consumo de potencia.

La disipación de potencia de cada uno de los bloques tiene tres componentes: la potencia de conmutación de las capacidades de carga, la potencia debida a la conmutación en las entradas de cada bloque y la potencia de *leakage* debida a las corrientes inversas y sub-umbral. El artículo [KAMG02] explica ampliamente como se realiza el modelado de cada uno de los bloques. Los autores han realizado una implementación basada en *SimpleScalar* y ellos mismos exponen sus dudas acerca del retardo que se introduce en la simulación y la precisión que se obtiene con cada estilo de diseño. Además hay que tener en cuenta que un simulador tan preciso se debe hacer de tal forma que se pueda parametrizar.

2.4. Conclusiones

Como acabamos de ver en la sección anterior, para evaluar nuevos diseños y realizar comparativas es necesario aplicar un modelo de estimación que permita calcular el consumo de un procesador. Es decir, debemos caracterizar cada una de las partes del sistema para la cual queramos estimar el consumo de potencia. También hemos visto que existen distintos modelos según el nivel de abstracción en que estén trabajando: nivel bajo, medio y alto.

Es importante conocer los modelos de niveles de abstracción inferiores porque, al realizar mejoras orientadas a reducir el consumo de potencia, siempre debemos tener en mente las consecuencias físicas que estas producen, y nos pueden orientar a la hora de hacer el diseño. Con estos modelos se consiguen resultados más precisos, pero presentan varios inconvenientes para ser utilizados en etapas tempranas de diseño. El primero de estos inconvenientes es que necesitan una descripción completa del procesador, incluyendo la implementación física del mismo (*layout*). El segundo de los inconvenientes es su elevado coste computacional, ya que para calcular con tal precisión los valores de potencia, es necesario resolver una gran cantidad de ecuaciones. Por estos dos motivos, parece inviable simular el consumo de un procesador con un simulador de este estilo, y mucho menos simular una carga de trabajo representativa como puede ser un juego de *Benchmarks* SPEC CPU2000 [SPE].

Nuestro interés se centra en los modelos de alto nivel que utilizan los simuladores, y dentro de estos, nos interesan los basados en *SimpleScalar* [BA97] ya que se ha convertido en la principal herramienta de simulación y modelado de sistemas de alto rendimiento. Como se ha visto en la sección 2.3.2, este

tipo de simuladores basados en *SimpleScalar* calculan el consumo en base a la ejecución de trazas de cargas representativas, midiendo la actividad. Dentro de este estilo de simuladores, el más popular es *Wattch* [BTM00].

Wattch [BTM00] es un simulador que incluye un modelo de alto nivel para el consumo del procesador y que se ha convertido en la herramienta más usada para calcular consumo en artículos presentados en conferencias internacionales sobre Arquitectura de Computadores.

Observando la clasificación de los simuladores que se presenta en la tabla 2.1 podemos decir que *Wattch*, respecto al alcance o ambito de la simulación, es un simulador de conjunto de instrucciones, es decir simula únicamente el repertorio de instrucciones de un procesador o microcontrolador y está codificado con lenguaje de alto nivel, de modo que imita la funcionalidad del procesador manteniendo los valores de sus registros en variables internas.

Si nos fijamos en el tipo de entrada, *Wattch* es un simulador basado en trazas, es decir, toma como valores de entrada un flujo de instrucciones previamente ejecutado en la misma arquitectura, de manera que los valores de entrada y el comportamiento de la simulación son invariables para todas las simulaciones de la misma traza. La traza tomada como entrada contiene 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.

Por último, respecto al nivel de abstracción, entendiendo por este la proximidad del simulador a las características de implementación del circuito simulado, este es un simulador funcional, es decir de granularidad gruesa donde la especificación del *Hardware* es sustituida por una descripción de la funciona-

lidad del circuito. Todas estas caracteristicas hacen de *Wattch* un simulador con una alta velocidad de ejecución y valido para la evaluación del consumo, puesto que en este tipo de pruebas no es tan importante el código simulado como los accesos a las estructuras.

Sin embargo, este simulador estima el consumo en las UFs de manera poco precisa. Por ejemplo tiene modeladas las UFs para calcular el consumo de distinta forma a como están modeladas funcionalmente. Es decir, si se está ejecutando una instrucción lógica (por ejemplo una AND), funcionalmente se realiza un acceso a la unidad lógica, sin embargo a la hora de calcular el consumo lo que se tiene en cuenta es un acceso a un sumador. Por ello, en el capítulo 4 se presenta una versión del simulador *Wattch*, que llamaremos *FU-Wattch*, a la cual hemos añadido algunas modificaciones para estimar el consumo en las UFs con mayor precisión. Con esto pretendemos preparar el simulador para poder disponer de una herramienta que sea capaz de evaluar de manera precisa las técnicas, para reducir el consumo en la unidad de ejecución, que proponemos en este trabajo.

Capítulo 3

Técnicas de bajo consumo

Como se ha mencionado en el capítulo 1, el consumo y la disipación de potencia constituyen uno de los principales retos tecnológicos en el diseño de los microprocesadores actuales y por lo tanto es fundamental tenerlo presente a la hora de evaluar nuevas propuestas. A lo largo de estos años, muchas técnicas han sido exploradas para reducir el consumo en los procesadores. Un trabajo interesante, donde se hace una revisión de los métodos y técnicas de optimización del consumo existentes hasta el 1999, es el realizado por Benini y de Micheli en el ISLPED del año 1999 [BM99a]. Por otro lado, en un trabajo algo más antiguo presentado por Frenkil en el DAC 1997 [Fre97], se hace una pequeña revisión a las fuentes de potencia y a las herramientas de diseño existentes en ese momento con una orientación fundamental al bajo consumo. Aunque en estos trabajos, algunos de los datos se han quedado obsoletos la mayoría de las motivaciones están vigentes.

En este capítulo se va a presentar un conjunto de técnicas de diseño de bajo consumo a nivel de arquitectura. Las técnicas que se van presentar son

fundamentalmente las propuestas en los últimos 8 años ya que, como se ha comentado antes, existe una revisión de los años anteriores hecha por Benini y de Micheli.

El resto del capítulo está organizado de la siguiente forma: la sección 3.1 muestra las tendencias en el diseño de bajo consumo. La sección 3.2 presenta un repaso general de las técnicas de bajo consumo aplicadas a nivel de arquitectura en los procesadores actuales. La sección 3.3 se centra en las técnicas para reducir el consumo en las unidades funcionales de los procesadores superescalares, ya que, además de ser una de las partes de mayor consumo, como ya se vio en el capítulo 1, es la parte del procesador en la que nos hemos centrado en este trabajo. Y por último, en la sección 3.4 se expondrán algunas conclusiones.

3.1. Tendencias en el diseño de bajo consumo

Una estrategia tradicional para reducir el consumo de potencia ha sido disminuir el voltaje de alimentación. Esta reducción consigue disminuir todas las componentes del consumo, especialmente la potencia dinámica, debido a su relación cuadrática con el voltaje, pero se hace necesario reducir la frecuencia, con lo que el rendimiento se degrada. Para seguir manteniendo velocidades de conmutación elevadas (alta frecuencia de reloj) es necesario escalar también el voltaje umbral del transistor, pero esto provoca un aumento en las corrientes de fuga, con el consiguiente perjuicio sobre la potencia estática. Muchos procesadores actuales incorporan mecanismos de control de frecuencia/voltaje que permiten hallar un adecuado compromiso entre consumo y rendimiento, y

que reciben la denominación genérica de escalado dinámico de voltaje (DVS: *Dynamic Voltage Scaling*) [IBM] [Int] [Tra]. No obstante, dado el nivel de consumo que se ha llegado a alcanzar, esta solución resulta insuficiente, y hoy día es necesario abordar el problema desde varios frentes adicionales: a nivel tecnológico, de microarquitectura y de *Software*.

- *A nivel tecnológico y de circuito*

  Una gran parte de la investigación desarrollada en este campo se ha centrado en reducir la componente dinámica del consumo, ya que históricamente suponía la principal fuente de disipación de energía en los circuitos CMOS. Generalmente, estas técnicas persiguen la disminución de la capacidad efectiva del circuito mediante modificaciones en su diseño. Sin embargo, como ya se dijo en el capítulo 1, con los tamaños de fabricación actuales, el peso de la componente estática ha ido cobrando cada vez mayor importancia por lo que algunas de las propuestas más recientes atacan el problema del consumo estático a nivel del diseño del propio transistor. Ejemplos representativos de esta tendencia son el *strained silicon*, los materiales *high-k*, los transistores *tri-gate*, etc. [CDD+04] [MAA+04].

- *A nivel de micromicroarquitectura*

  La mayor parte de propuestas en este nivel están enfocadas a la reducción del consumo dinámico, y se basan en la disminución de la actividad del circuito, entendiendo como tal la frecuencia de conmutación de los transistores. Los principales referentes en este contexto son las técnicas denominadas *gating* y *throttling*. Mientras que la primera técnica consiste

en evitar el acceso a ciertas estructuras o etapas del procesador (cuando no contribuyen de manera significativa a la mejora del rendimiento) [MKG98] [LBC+03] [HBS+04] [Kim07], la segunda se basa en ralentizar su funcionamiento [BM01] [AGS05] [KN05] [RV07]. Estas últimas deterioran el rendimineto del procesador por lo que, de manera complementaria, para mejorar la eficiencia del procesador se pueden aplicar soluciones alternativas como la división de estructuras en módulos para poder evitar el acceso a los que no sean necesarios, la organización en *clusters*, o la división de los dominios de reloj [Gho00] [Mar04] [SAD+02] [SMB+02] [CGG04].

- *A nivel Software*

  Por último, el compilador o incluso el sistema operativo pueden contribuir también a mejorar la gestión de consumo. Por una parte, teniendo en cuenta el consumo a la hora de tomar sus decisiones [HK03], y por otra parte, ayudando a la aplicación de algunas de las técnicas mencionadas anteriormente [HK05] [HKH01] [Kre05a] [XMM03].

En la siguiente sección nos vamos a centrar en el nivel de microarquitectura ya que en esta tesis se presentan técnicas de diseño de bajo consumo en dicho nivel.

3.2. Técnicas de diseño de bajo consumo a nivel de microarquitectura

La continua evolución en pos de un mayor rendimiento, unida a la cada vez mayor disponibilidad de transistores, ha provocado la incorporación de elementos cada vez más complejos dentro del procesador, muchos de los cuales sólo son beneficiosos bajo circunstancias muy particulares. Esta tendencia, ha hecho que para ciertos programas, una gran parte de los recursos sean innecesarios o estén infrautilizados, de manera que su uso comporta un desperdicio innecesario de energía, con los problemas que ello conlleva.

Para evitar este malgasto de energía, han surgido diversas técnicas a nivel de microarquitectura que, siguiendo las tendencias descritas en el apartado anterior, permiten ejecutar cada aplicación usando solamente el *Hardware* necesario, de manera que el *Hardware* que no se use reduzca su consumo o directamente no consuma. Es decir, con estas ténicas se consigue modificar dinámicamente la configuración del procesador para adaptarlo a las necesidades específicas del programa o fase de programa en curso. Dichas técnicas pueden agruparse bajo la denominación genérica de *Técnicas de Adaptación Dinámica de Estructuras del Procesador*.

En esta sección vamos a hacer un repaso de las técnicas de diseño de bajo consumo a nivel de microarquitectura aplicadas a procesadores superescalares en los últimos años. Debido a la gran cantidad de técnicas existentes en la literatura, para exponerlas hemos decidido agruparlas en función del elemento/etapa del procesador objeto de adaptación. Como se ha comentado al principio de esta sección, a las técnicas aplicadas en las unidades funcionales le

hemos dedicado una sección especial ya que, además de ser una de las partes de mayor consumo, como ya se vio en el capítulo 1, es la parte del procesador en la que nos hemos centrado en este trabajo.

Tablas de renombramiento de registros

En los microprocesadores modernos los registros de renombramiento se implementan usando RATs (*Register Alias Tables*). Estas tablas contienen el mapeado entre los registros lógicos y los físicos, de manera que cada instrucción puede identificar su registro fuente físico consultando en la RAT la dirección del registro fuente lógico. Esta información es necesaria para manejar la dependencia de datos. La energía disipada en una RAT proviene de los siguientes procesos: (1) la lectura de la dirección del registro físico para los operandos fuente, (2) la lectura de los registros destinos desde el ROB y (3) la escritura de los registros destino para la actualización de la RATs. Los accesos de lectura y escritura a la RAT y las acciones para chequear y actualizar la información consumen gran cantidad de energía (alrededor del 14 % del consumo total de un procesador). Su alto consumo y su área reducida hacen de las RATs estructuras con una alta densidad de potencia comparable a las *caches*.

Kucuk *et ál.*, en [KEPG03], proponen dos técnicas para reducir la potencia y la densidad de potencia de las RATs en los microprocesadores superescalares. Estas técnicas explotan la observación de que muchos de los valores de los registros fuentes de las instrucciones son proporcionados por instrucciones muy cercanas. La primera técnica consiste en desactivar la consulta de la RAT para un registro fuente, si dicho registro es el destino de una instrucción que ha sido lanzada en el mismo ciclo. Es decir usa las dependencias de datos entre las instrucciones lanzadas en un mismo ciclo (*intra-group dependency*).

Estas dependencias son detectadas usando un conjunto de comparadores como en las RATs convencionales. Cuando uno de los comparadores detecta la dependencia, la puerta NOR a la que están conectadas las salidas de todos los comparadores se pone a 0 impidiendo que se activen los *sense-amp* usados para leer dicho registro fuente de la RAT. Esta técnica no prolonga el tiempo de ciclo porque la salida de la puerta NOR, para activar las líneas de bit, llega a los *sense-amp* antes de que la línea de palabra complete su activación.

La segunda técnica elimina el acceso de lectura a la RAT cuando el valor del registro fuente de una instrucción es producido por una de las instrucciones que han sido lanzada en un ciclo anterior cercano. Para ello se añaden unos cuantos *latches*, externos a la RAT, donde se almacenan algunos de los valores producidos por las últimas instrucciones. Esta técnica tampoco prolonga el tiempo de ciclo porque la búsqueda asociativa es rápida y la detección de un acierto en los *latches*, para activar las líneas de bit, se completa antes de que la línea de palabra complete su activación.

Ventana de instrucciones

Otra de las piezas clave en los procesadores modernos de alto rendimiento es la ventana de instrucciones. Esta estructura se usa para poder realizar ejecución fuera de orden. En ella las instrucciones se almacenan después de haber pasado por las etapas de búsqueda, decodificación y renombramiento y allí esperan a que sus operandos estén listos y se hayan resuelto las dependencias. Cuando esto ocurre, las instrucciones están listas para ser lanzadas (*issued*) a la unidad funcional. Cuando una instrucción termina de ejecutarse la etiqueta de su registro destino se envía a todas las instrucciones que se encuentran en la ventana. Una vez hecho esto, cada entrada de la ventana compara la etiqueta

del bus con las etiquetas de sus operandos fuente. Si alguna de las comparaciones es positiva el valor del dato que hay en el bus se almacena en el operando fuente correspondiente y la instrucción puede estar lista para ser ejecutada si todos sus operandos están resueltos. Al envío de la etiqueta desde la unidad funcional a la ventana de instrucciones y a la comparación asociativa de las etiquetas se les denomina *wake-up* de instrucciones. Esta operación requiere la utilización de un cableado y múltiples comparadores lo que conlleva un gran consumo de potencia.

Existen diversas técnicas para reducir la energía consumida en esta etapa. Unos de los primeros en abordar este tipo de adaptación fueron Folegnani y González, que en [FG01] proponen una lógica de *wake-up* de bajo consumo. Estos autores observaron que, al igual que sucede con otras estructuras del procesador, ciertas entradas de la ventana de instrucciones (aquellas que están vacías o tienen sus operandos disponibles) no comtribuyen a mejorar el rendimiento y sin embargo consumen energía. Para hacer frente a este problema, propusieron deshabilitar (*gating*) la lógica de *wake-up* asociada a estas entradas. Es decir, que las entradas de la ventana de instrucciones que estén vacías o que no estén esperando algún operando no activen sus comparadores cuando llegue un dato de la unidad funcional. Además, para reducir en mayor medida el consumo de energía, plantearon ajustar su tamaño efectivo(máximo número de entradas activas), en función de la demanda del programa, de manera que se reduzca el área vacía de la ventana.

La decisión de cuándo y cómo adaptar el tamaño de la ventana de instrucciones(lo que se llama *mecanismo de control*) es el siguiente. Se registra para cada instrucción su posición en la ventana al comenzar su ejecución. Cuan-

do se observa que las instrucciones no provienen del final de dicha ventana, se infiere que esta es demasiado grande y se reduce su tamaño. Periódicamente, se restaura el tamaño original a fin de evitar pérdidas de rendimiento.

Posteriormente, en [AG03], Abella y González retomaron esta misma idea, pero basando las decisiones en el tiempo que las instrucciones permanecen en la ventana de instrucciones y en el *Buffer* de Reordenamiento (ROB: *Reorder Buffer*). Estos mismos autores han investigado también el control *Software* aplicado a este mismo esquema de adaptación [JOAG05].

En la misma línea que Folegnani y Gonzalez, Huang *et ál.*, en [HRT02], presentan una técnica basada en reducir el número de comparadores que se activan en la etapa de *wake-up*, pero sin ajustar el tamaño de la ventana. Lo que proponen en esta técnica es recordar la posición de la instrucción dependiente y mediante indexación activar sólo el comparador de esa instrucción. En el caso de que más de una instrucción tenga esa dependencia, se activarán un subconjunto de comparadores o todos ellos, dependiendo del caso.

Estas dos técnicas explotan el hecho de que la mayoría de las instrucciones tienen pocas instrucciones dependientes de ellas. Han observado que, en un código ordinario, alrededor de un 70 % de las instrucciones con registro destino (los saltos y los *stores* no tienen registro destino) tienen una o ninguna instrucción dependiente de ellas (60 % y 10 % respectivamente). Si sólo consideramos las instrucciones dependientes que se encuentran dentro de la ventana de instrucciones cuando la instrucción que produce el resultado finaliza (*dependencia cercana*), este porcentaje aumenta, (para una ventana de 96 entradas hay un 91,3 % de media de instrucciones que tienen una o ninguna dependencia cercana, el 40,5 % ninguna y 50,8 % una).

Por otro lado, en [Gho00] se proponen tres técnicas para reducir el consumo en la ventana de instrucciones desde otro enfoque. La primera técnica consiste en dividir el *Buffer* de la ventana de instrucciones *IWB, Instruction Window Buffer* en pequeñas ventanas especializadas en distintas unidades funcionales. Como consecuencia de esta división el número de puertos de lectura y escritura se reduce, reduciendo con ello el consumo sin pérdida en la ejecución. Además los árboles que se necesitan para implementar la búsqueda en las pequeñas IWB son más sencillos, por lo tanto consumen menos. En la segunda técnica también se divide la ventana IWB en pequeñas ventanas especializadas para los distintos tipos de unidades funcionales, pero ahora el formato de la entrada puede ser diferente dependiendo de las necesidades de cada tipo de instrucciones, como por ejemplo el número y tamaño de los operandos. Como tercera técnica propone reducir el número de transiciones en el bus y en los bits de línea, detectando dinámicamente el número de bits significativos en los operandos y almacenando dicho número codificado en un campo añadido a la entrada de IWB (por ejemplo, con un bit se pueden indicar si el dato es de 32 bits o de 64 bits).

Otras propuestas con este enfoque se pueden encontrar en los trabajos [BAS+01] [BABC02] [BSB+01]. En ellos la ventana de instrucciones se fragmenta en bancos que se activan y desactivan en función de su actividad (número de entradas válidas por intervalo de tiempo).

Predictor de saltos

Los procesadores de alto rendimiento incorporan predictores de saltos muy sofisticados, compuestos de grandes estructuras que se acceden en cada ciclo, con lo que en conjunto consumen una gran parte de la energía total del chip.

En muchos casos, esta puede superar el 10 % de la energía consumida en el procesador [PSZ+02] [CWS05]

Si bien esta elevada complejidad del predictor de saltos es necesaria en muchos casos, un análisis detallado del uso de los predictores durante la ejecución de aplicaciones reales demuestra que, en ocasiones, las estructuras que lo componen permanecen infrautilizadas durante largos períodos de tiempo, o simplemente no contribuyen en absoluto al resultado de la predicción, lo que supone un derroche importante de energía.

Como ejemplos tenemos la propuesta de Chaver *et ál.* Este grupo, en [MH03] propone estimar la demanda de la aplicación y ajustar dinámicamente los recursos de predicción de acuerdo con dicha estimación. Es decir, presentan una solución basada en *profiling*, para ajustar dinámicamente los recursos del predictor y de este modo reducir su consumo de energía. Esta propuesta nace de observar el uso que hacen las aplicaciones del *Hardware* de predicción. En este trabajo han demostrado experimentalmente que las necesidades varían ampliamente de un programa a otro, o incluso entre módulos de una misma aplicación. Por este motivo, mantener constante la configuración de los elementos que componen el predictor de saltos, resulta ineficiente desde el punto de vista del consumo. Para mejorar el uso de esta componente del procesador, plantean como alternativa un esquema que adapta la complejidad operacional a la demanda específica de cada sección de código, procurando no incrementar ni la complejidad de diseño ni el tiempo de ejecución.

***Buffer* de reordenamiento**

Tampoco se puede olvidar el *buffer* de reordenamiento (ROB). Esta es una estructura clave, usada en el diseño del camino de datos de los microprocesa-

dores superescalares, para la programación dinámica de instrucciones. Se usa para recuperar el estado exacto de los registros y las direcciones de memoria cuando ocurre un fallo en la predicción de un salto o una interrupción. La estructura ROB se implementan como una cola circular FIFO, con dos punteros, uno en la cabecera y otro en el final de la cola. Las instrucciones lanzadas conjuntamente se van almacenando al final de la cola, manteniendo el orden del programa y desde aquí, cuando se sabe que la predicción ha sido correcta, se realiza la fase de *commit*. En un estudio hecho por Folegnani y González [FG01] se estima que el 27 % del consumo total del microprocesador se disipa en los ROB.

Un ROB disipa energía, en primer lugar, en el proceso de introducir las instrucciones en la ROB, pues ello implica escribir la dirección de la instrucción (PC), el *flag* que indica el tipo de instrucción y el identificador del registro destino o la dirección de la predicción del salto (si la instrucción es de salto). En segundo lugar en el proceso de lectura de los registros físicos fuente de las instrucciones cuando se realiza la etapa de *dispatch*. Antes de leer dichos registros se ha tenido que realizar una búsqueda asociativa para identificarlos, siendo esta búsqueda una de las tareas que más consumen. También consume al escribir los resultados desde las unidades funcionales al ROB en el tiempo de *writeback* y en la fase de *commit*. Por último mencionamos el proceso de limpiar el ROB si hay un fallo de predicción, aunque esta es bastante pequeña comparada con las demás fuentes de consumo.

Para solucionar los problemas de consumo en el ROB se pueden usar varias técnicas. Por ejemplo se puede usar un ROB dinámicamente redimensionable, es decir, el ROB se implementa como un conjunto de particiones indepen-

dientes los cuales se activan o desactivan en función de la demanda de las aplicaciones [PKG01]. También se pueden usar comparadores de bajo consumo, que disipan energía en el caso de que la comparación de positiva, para la búsqueda asociativa de los registros físicos en vez de buscar en tablas de renombramiento (RAT). Otra opción consiste en usar un decodificador de bytes de ceros para detectar esos *bytes* y evitar su lectura y escritura, esto reduce el número de líneas de bits que se tienen que activar durante los procesos de *dispatch*, *writeback* y *commitment*. El uso de comparadores de bajo consumo y decodificadores de *bytes* de ceros ya ha sido propuesto para las colas de lanzamiento [KGPK01] y las *caches* [VZA00]. Ponomarev *et ál.*, en [PKG02], proponen combinar estos tres mecanismos consiguiendo una importante reducción del consumo.

Etapa de captación de instrucciones

Además de todas estas estructuras la etapa de captación de instrucciones también representa un elevado porcentaje del consumo de energía del procesador, alrededor del 25 % [CWS06]. El modelo de ejecución superescalar suele dividirse en dos partes: el *front-end*, cuya responsabilidad es suministrar el flujo dinámico de instrucciones, y el *back-end*, encargado de procesarlas. Si el primero no es capaz de proporcionar un flujo suficientemente abundante de instrucciones, el rendimiento del conjunto del procesador se ve seriamente afectado. Por este motivo, se ha dedicado un gran esfuerzo a la mejora de los mecanismos implicados en la captación de instrucciones. No obstante, la atención se ha centrado principalmente en el rendimiento, y son relativamente pocos los trabajos que han considerado su consumo de energía a pesar de ser éste muy elevado.

Las técnicas de adaptación dinámica aplicadas en esta etapa se dividen en dos clases: aquellas que actúan sobre la velocidad de búsqueda, y aquellas que modifican su comportamiento.

- Velocidad de búsqueda

  Como se ha probado en trabajos previos, por ejemplo en [MKG98], en muchos casos el suministro de instrucciones por parte del *front-end* es mayor que el necesario. Esto se debe a que su diseño está orientado a mantener el *back-end* a máximo rendimiento, y se buscan las instrucciones a la mayor velocidad posible. Sin embargo esta estrategia suele implicar un desperdicio de energía, ya que es posible que las instrucciones se traigan antes de lo estrictamente necesario y permanezcan más tiempo dentro del *pipeline*. Además, este problema se agrava en caso de fallos de predicción de saltos, ya que muchas instrucciones se lanzan especulativamente, de manera que cuando ocurre un fallo en la predicción del salto, todas esas instrucciones tienen que ser borradas.

  Una de las primeras soluciones que se ha estudiado consiste en realizar un control de la especulación en base a la estimación de *confianza* de los saltos. En [MKG98] se reduce el número de instrucciones especulativas entregadas al *pipe* cuando el nivel de confianza del predictor es bajo (PG, *Pipeline Gating*). Es decir, para aquellas predicciones que ofrecen poca fiabilidad, no se especula y se aguarda a que el salto se resuelva, congelando durante este tiempo la captación de instrucciones. De esta manera se evita el gasto de energía derivado de las instrucciones pertenecientes al camino incorrecto. Con posterioridad a este primer trabajo, Aragón

et ál., en [AGG03], mejoraron esta solución, empleando la estimación de confianza como criterio para ajustar la velocidad, no sólo de la etapa de búsqueda, sino también de las de decodificación y lanzamiento.

En la misma línea, Karkhanis *et ál.*, en [KSB02], proponen un método llamado *entrega de instrucciones justo a tiempo* (JITD, *Just-in-Time Instruction Delivery*). La idea es monitorizar y ajustar dinámicamente el máximo número de instrucciones entregadas al *pipe*. Cuando se alcanza dicho máximo la etapa de búsqueda (*fetch*) se bloquea, evitando que la etapa de decodificación y la cola de lanzamiento se completen. Este número de instrucciones entregadas al *pipe* no debe ser demasiado pequeño para que no penalice demasiado el tiempo de ejecución.

- Modificación del comportamiento

  Actuar sobre la velocidad del *front-end* permite, como acabamos de mencionar, disminuir el número de instrucciones en el *pipeline*, con la consiguiente reducción de energía. Pero el consumo del propio mecanismo de búsqueda de instrucciones no se ve prácticamente modificado. Para reducirlo, es necesario actuar sobre las estructuras que lo componen.

  Talpes y Marculescu en [TM01] proponen una nueva organización de la microarquitectura que permite reducir el consumo reutilizando trabajo hecho en la fase de *issue* y en la de renombramiento de registros. En esta nueva organización la *cache* de trazas (*cache* especializada en almacenar instrucciones ya decodificadas (trazas) listas para su uso [RBS96]) se coloca después de la fase *issue*, de manera que las instrucciones que han sido buscadas, decodificadas y sus registros han sido renombrados, se

almacenan en la *cache* de trazas en orden de lanzamiento y no en el orden de programa. De esta manera, la etapa de ejecución puede tomar las instrucciones de la ventana de instrucciones, durante la fase de la construcción de la traza, o de la *cache* de trazas, si ocurre un acierto en dicha *cache*. Con este nuevo diseño se reduce el camino crítico de ejecución y con ello el número de módulos que se usan para ejecutar una instrucción. Esto permite intervalos largos y predecibles durante los cuales algunos de los recursos no consumen.

En esta misma línea se encuentran los trabajos de Hu *et ál.* En un primer trabajo [HVKI02], estos autores evaluaron las ventajas de usar STC (*Sequential Trace Cache*) como solución para reducir el consumo de un mecanismo de captación de instrucciones basado en la *cache* de trazas tradicional. En el esquema original [RBS96], se accede en paralelo a la *cache* de instrucciones y a la *cache* de trazas, lo que supone un notable incremento del gasto de energía. Por el contrario, empleando un esquema STC, se accede a estas estructuras de manera secuencial: primero a la *cache* de trazas, y sólo si esta falla, se accede en el siguiente ciclo a la *cache* de instrucciones. De este modo se ahorra energía en caso de acierto en la *cache* de trazas, a costa de un ciclo de penalización en caso de fallo. En sus trabajos posteriores [HIVK02] [HVIK03] [HVIK07], estos mismos autores exploraron dos diseños alternativos para lograr un ahorro de energía similar al obtenido con una STC, pero suavizando el impacto sobre el rendimiento. El primero, denominado DPTC (*Dynamic direction Prediction based Trace Cache*), filtra los accesos a la *cache* de trazas cuando su confianza es baja. Dicha estimación de confianza se

lleva a cabo gracias a dos contadores saturados que se incorporan a cada una de las entradas de la BTB (*Branch Target Buffer*). Si bien este esquema mantiene el rendimiento, proporciona unos ahorros de energía muy inferiores. El segundo diseño propuesto por estos autores, denominado SlTC (*Selective Trace Cache*), representa una variante de la STC, en la que gracias a la ayuda del compilador se restringe el uso de la *cache* de trazas a aquellos casos en los que previsiblemente acertará, evitando así las penalizaciones que supone utilizarla en el resto de los casos.

Chaver *et ál.*, en [CRP+05], proponen hacer uso de técnicas de escalado dinámico y de *gating* para adaptar la configuración del *front-end* según las necesidades del módulo en ejecución. Concretamente, emplean escalado dinámico en la *cache* de trazas, y *gating* para seleccionar la política de captación, que puede variar entre cuatro políticas distintas (SEQ1, SEQ3, CTC, y STC), que son frecuentemente utilizadas en la literatura. No obstante, la metodología desarrollada tiene un carácter general. Además, el control de la adaptación se ha efectuado por mecanismos *Software*. En general, la técnica consigue una disminución del consumo de energía muy considerable, logrando incluso una ligera mejora del rendimiento debida a efectos laterales.

Por último, cabe mencionar el trabajo de Co, Weikle y Skadron [CWS06], en el que se ha evaluado, para diversas implementaciones de la *cache* de trazas (CTC, STC y BBTC), su eficiencia en términos de energía.

Memoria

El consumo de energía de las memorias *cache* representa una parte muy

significativa del consumo total del chip (debido tanto a su elevado número de accesos como a su área), y por este motivo se han dedicado importantes esfuerzos a reducirlo. Aunque no todas las propuestas, muchas de ellas hacen uso de técnicas de adaptación dinámicas. A continuación describiremos algunas de las más significativas. Podemos decir que existen dos principios del diseño de memorias de bajo consumo:

1. Especializar las *caches* para manejar cierto tipo de referencias. Si estas referencias son frecuentes esta especialización hace que se produzca una mejora en tiempo de acceso y en el consumo de la *cache*. Un ejemplo de *caches* especializadas son las *caches* de pilas (SSC, *Specialized Stack Cache*) diseñadas para manejar los accesos a pilas [LSNT01] [CYL99].

2. Romper o dividir la *cache* en pequeñas *subcaches*. Se sabe que las *caches* pequeñas consumen menos y son más rápidas, luego si rompemos la *cache* en pequeñas *subcaches* a las cuales se pueda acceder independientemente, conseguimos mejorar el producto energía-retardo. Además esta división de la *cache* en *subcaches* permite desactivar las partes que no sean necesarias. Una *cache* organizada de esta manera se denomina *cache* pseudo asociativa por conjuntos (PSAC, *Pseudo Set-Associative Cache*). Ejemplos que usan estructuras PSAC son: *Predicting Secuential Associative Cache* [CGE96] y *Way Predicting Set-Associative Cache* [IIM99].

Huang *et ál.*, en [HRYT01], proponen un método para reducir el producto energía retardo del primer nivel de *caches*, que combina las dos técnicas que

se acaban de describir. Para la primera técnica propone un nuevo diseño de *cache* de pilas en la que se van a almacenar todas las referencias a pila.

Para la segunda técnica propone un nuevo esquema de *caches* pseudos-asociativas por conjuntos, basado en el concepto *cache* por fases (*Phased Cache*) [HKY+95]. Estas son *caches* que cuando se accede a ellas, primero activan las etiquetas y sólo si hay acierto, se accede al array de datos. En estas *caches* cuando se va a producir un acceso, un mecanismo de predicción selecciona en cuál de las *subcaches* se va a buscar. Si se produce un acierto el acceso se lleva a cabo de manera rápida y con bajo consumo. Si se produce un fallo hay que seleccionar otra *subcache* y volver a probar, con la consecuente penalización en el tiempo y consumo.

Como ya se ha comentado en el apartado anterior, una forma de reducir el consumo consiste en reducir el tamaño del primer nivel de *cache*. Es decir, en dividir una *cache* en unidades más pequeñas, cada una de las cuales son a su vez pequeñas *caches*, denominadas *subcaches*. Una *cache* grande se divide en pequeñas *subcaches* para reducir el cableado y las capacidades de difusión de las líneas de bit y de las líneas de palabras usadas para activar las celdas de memoria. La reducción de la capacidades ayuda a reducir el consumo de energía dinámica cuando se accede a la *cache*. Además esta división de la *cache* en *subcaches* permite desactivar las partes que no sean necesarias.

Una forma de hacer la partición es usar políticas de emplazamiento de datos y de predicción de *subcaches* aprovechando la localidad espacial y temporal de los datos como proponen Kim *et ál.* en [KVK+01].

Respecto a desactivar las partes de la *cache* que no sean necesarias, la forma más sencilla de proceder sería basar la decisión en la dirección del acceso, y

activar sólo el sub-array que contiene el dato. No obstante, esta aproximación conlleva un incremento de la latencia, al tener que secuencializar las tareas de selección y acceso. Para evitar esta penalización, las soluciones más frecuentes toman la decisión empleando adaptación dinámica. En [Alb00], se presenta un esquema que permite variar dinámicamente la asociatividad, mediante la desactivación (*gating*) de los sub-arrays que forman la(s) vía(s). En este caso, la gestión de la adaptación se realiza con la ayuda del *Software*. Evidentemente, esta técnica sólo se puede aplicar si la *cache* es asociativa por conjuntos, no obstante esto no supone un gran inconveniente ya que se trata del caso habitual. Al igual que sucede con el resto de técnicas adaptativas, es necesario tener cautela al reducir la asociatividad, pues de lo contrario el rendimiento y el consumo pueden verse seriamente penalizados.

Si las vías a su vez se dividen en sub-arrays, cabe la posibilidad de aplicar esta misma idea para la desactivación de conjuntos, como han propuesto algunos autores [YPFV02]. Es obvio que esta propuesta implica un ligero sobrecoste *Hardware* respecto a la propuesta de Albonesi, sin embargo, permite una mayor granularidad en la adaptación de la *cache*. En [YPFV02] se han considerado también propuestas híbridas aplicando ambas técnicas.

Otra técnica es la denominada *way-prediction scheme*, propuesta en [IIM99]. En ella se usa un predictor con una entrada por cada conjunto de la *cache* asociativa. En dicha entrada hay un puntero que indica cual de los marcos de bloque del conjunto es al que se va a acceder, con esto se consigue ahorrar energía porque sólo se hace la comparación de etiqueta en el marco de bloque seleccionado y sólo se activa su correspondiente array de datos o de instrucciones. En el caso de una predicción incorrecta, hay que realizar de nuevo el

acceso, lo que conlleva un consumo extra de energía y tiempo. En esta misma línea está el trabajo de [PAV+01].

La desactivación de sub-arrays empleando *gating* permite reducir únicamente la energía dinámica. Para reducir también la componente estática es necesario, además, modificar el diseño de las celdas tal y como propusieron Yang *et ál.* en [YPF+01]. Esta solución ha sido adoptada recientemente por Intel en el diseño del procesador *Yonah* [Kre05b], para la adaptación de la asociatividad de la *cache*.

Para lograr que no se pierdan los datos y al mismo tiempo reducir casi completamente la energía estática cuando una estructura se encuentra en el estado desactivado, existen técnicas en las que la deshabilitación no apaga totalmente el conjunto sino que lo alimenta con una tensión menor. Un ejemplo de este tipo de *caches* son las *caches drowsy* [FKM+02]. Esta es una *cache* en la que los bloques pueden estar en dos tipos de estado. (1) El estado *somnoliento* (*drowsy*), en el cual el dato se conserva pero no se puede acceder a él y en el que se reducen las corrientes de *leakage*. (2) El estado *despierto* (*awake*) que se usa para acceder al dato. En las *caches drowsy* los bloques están periódicamente en estado de bajo consumo (*drowsy state*), por lo que para acceder a ellos se necesita un ciclo extra llamado *wake-up* time que los saque de ese estado. Este tipo de *caches* produce una indeterminación en la latencia de las instrucciones debido a que el bloque seleccionado puede estar en modo ahorro o no. Otros trabajos en esta línea son [KHM01] [KFBM02] [KFBM04] [SS04] [PJH07].

Kim *et ál.*, en [KVIJ03], proponen un método para las *caches drowsy* que reduce los efectos de la indeterminación de la latencia de las instrucciones *load*. La técnica que proponen para evitar esto, es un esquema de predicción

del conjunto al que se va a acceder (*early caches set resolution scheme*). La idea es la siguiente: si se puede saber con anterioridad a que conjunto de la *cache* es al que se va a acceder, no necesitamos esperar a que se genere la dirección para despertar (*wake-up*) los bloques de dicho conjunto, pudiendo usar el mismo ciclo de la fase de ejecución para hacerlo.

Por otro lado tenemos técnicas a nivel arquitectónico que no son adaptativas aunque si compatibles con ellas. Estas consisten en incorporar una pequeña *cache*, denominada *cache* de filtrado (*filter cache*), para reducir el número de accesos al primer nivel y de este modo reducir su energía dinámica. Ejemplos de estas técnicas se pueden ver en [KMMS97] [KGMS00] donde se propone una *cache* que actúe como un filtro, esto es una *cache* L0, pequeña y rápida, que en los aciertos tiene un consumo y una latencia menores que la *cache* L1. Sin embargo, debido a su pequeño tamaño tiene un porcentaje de fallos muy alto, esto hace que la latencia aumente, es decir este método en promedio reduce el consumo pero aumenta la latencia.

En este mismo sentido, se tiene a la *cache* de *víctimas* (*victim cache*)[BAM98]. La *cache* de víctimas es una *cache* más pequeña, totalmente asociativa y asociada a la memoria *cache*. Está situada entre el primer nivel de *cache* y el siguiente nivel en la jerarquía de memoria. Estas *caches* son pequeñas estructuras de bajo consumo que evitan el acceso a estructuras mayores y por lo tanto de mayor consumo. Cuando se produce un fallo en el primer nivel de la *cache* se accede a la *cache* de víctimas, si el dato se encuentra en dicha *cache*, entonces se lleva al primer nivel de *cache* y el bloque que se reemplaza se coloca en la *cache* de víctimas. Si el dato no se encuentra en la *cache* de víctimas, entonces se accede al siguiente nivel de memoria y se trae el bloque

de este nivel directamente al primer nivel de *cache*. El bloque reemplazado se lleva a la *cache* de víctimas. En esta configuración antes de acceder a la *cache* de víctimas se accede a la *cache* principal, por lo que se denomina *serial victim cache* (SVC).

Bahar *et ál.*, en [BCG99], proponen una nueva configuración para usar la *cache* de víctimas: *Parallel Victim Cache* (PVC). Por otra parte, Memik *et ál.*, en [MRMS03], pretenden mejorar el consumo de las *caches* de víctimas, para ello introducen funciones *bypass* (*bypass function*) que se usan para predecir los accesos fallidos a la *cache* de víctimas. Si de antemano sabemos que el acceso va a producir un fallo, se elimina dicho acceso. Es decir, estas funciones son en realidad predictores de fallos.

Otras técnicas para las *caches* de datos son las propuestas por Bodik y Gupta [BGS99] y Yang y Gupta [YG01] donde está presente el concepto *load redundancy*, es decir un significativo número de instrucciones *load* acceden al mismo dato en un corto intervalo de tiempo. La idea de estas técnicas es usar una unidad de reutilización de datos para instrucciones *load* y *store* como mecanismo de filtro para la *cache* de datos, de manera que se reduzca la actividad de la *cache* y con ello el consumo. Un *load* se puede evitar mediante reutilización de datos, si el valor que tiene que leer desde una determinada dirección ha sido cargado (almacenado) anteriormente desde (en) la misma dirección y además dicho valor no ha sufrido cambios. Por su parte, un *store* se puede evitar si el valor que tiene que escribir en una determinada dirección está ya en esa dirección. Estas técnicas reducen el consumo pero requieren un *Hardware* adicional importante.

La tarea de encontrar un mecanismo de reutilización que realmente reduzca

el consumo no es fácil, puesto que el propio mecanismo puede llegar a consumir bastante energía. La técnica presentada por Yang *et ál.* en [YG01] es una técnica no especulativa que ha sido cuidadosamente diseñada para conseguir dos objetivos: por un lado que sea capaz de capturar una gran fracción de los *load* y *stores* reutilizables y por otro lado, que gestione la energía de la manera más eficaz, es decir, que en los programas donde hay un gran nivel de reutilización se consiga un gran ahorro de energía y que en los que existen pocas posibilidades de reutilización el incremento de la energía sea el mínimo posible.

Molina *et ál.*, en [MAG+03], proponen un diseño para la *cache* de datos que reduce el área, el consumo y la latencia. La fuente principal para la reducción del consumo en este diseño viene de la reducción de área (en un chip de memoria la potencia de disipación dinámica está relacionada con el área y la potencia de disipación estática con el número de transistores). Esta técnica no está enfocada a reducir la tasa de fallos, mantiene la misma tasa que las *caches* tradicionales. A esta nueva *cache* la llaman *Non Redundant Cache* (NRC) y esta diseñada para el segundo nivel de *caches*, L2. La idea surge de explotar la repetición de datos observados en las *caches* tradicionales: se ha observado que el número de valores repetidos aumenta con el tamaño de la *cache*, sin embargo, es independiente del grado de asocitividad.

Por último, Nicolaescu *et ál.*, en [NVN03], proponen una técnica que reduce la energía consumida por la *cache* de datos y, a la vez, la latencia de las instrucciones *loads*. Es decir consigue un buen producto energía-retardo. Además su implementación usa recursos presentes en el procesador y requiere muy poco *Hardware* extra. La idea es la siguiente: el tamaño de las colas de

load/store (LSQ) está creciendo significativamente en los nuevos procesadores debido a la necesidad de explotar mejor el paralelismo a nivel de instrucción, sin embargo la ocupación media de dichas colas no es muy alta, esto deja disponible un espacio de almacenaje debido a las entradas no usadas de la LSQ. Esta idea explota el desuso temporal de algunas entradas de la LSQ para almacenar datos, permitiendo a las instrucciones *load* tomar los datos de esta zona evitando el acceso a la *cache*. La energía disipada en los accesos a LSQ es mucho menor que en los accesos a *cache* de datos y por otro lado el acceso a la LSQ es más rápido y el número de aciertos aumenta. Luego con esta técnica se consigue un ahorro de energía y un decremento de la latencia, y por lo tanto se reduce el producto energía-retardo. Este diseño explota la localidad de las instrucciones *load* y *store* que existe en los programas.

3.3. Técnicas aplicadas a las unidades funcionales

La mayoría de las técnicas aplicadas a las UFs, a nivel de microarquitectura, se centran en minimizar el número de transiciones en dichas estructuras ya que el consumo dinámico es proporcional al factor de actividad de sus entradas, es decir a la cantidad de cómputos realizados en sus entradas. Uno de los métodos más efectivos para reducir el número de transiciones en las unidades funcionales es incrementar la correlación de los datos de entrada. En esta línea se encuentran los trabajos presentados en [RJ94] [MC95] [RDJW96] [SC97] [KC97] [DRJW99].

Uno de los inconvenientes de estas técnicas, es la complejidad que implica la lógica que chequea y comprueba cuales son los datos para obtener la máxima correlación entre ellos. Esta lógica, hace que se introduzca una penalización en el rendimiento, además del consumo extra que representa. Haga *et ál.*, en [HRBM03] y [HRBM05], presentan diferentes maneras para optimizar esta lógica.

Como se ha dicho en el capítulo 1 , una de las características de los procesadores actuales es que tienen varias unidades funcionales. Como ejemplo tenemos el *Alpha* 21264 [Kes99] que presenta cuatro sumadores de enteros, cuatro unidades lógicas, etc. La figura 3.1 muestra los operandos de entrada en tres UFs identicas, que forman parte de la unidad de ejecucion de un procesador de 3-vias, durante dos ciclos consecutivos. En ella se pueden ver dos opciones posibles, la ruta por defecto y una ruta alternativa. En la ruta por defecto, la asignación de operandos a las UFs no tiene en cuenta la correlación de los datos. Sin embargo en la ruta alternativa, se puede ver como los operandos han sido asignados a la UF que tenía en sus entradas los valores más parecidos a ellos, consiguiendo una importante reducción en el consumo dinámico. Como podemos observar, en esta asignación está permitido hasta intercambiar la posición de los operandos, cosa que si la operación es una resta no es trivial. Como ya hemos comentado, el trabajo de Haga presenta una serie de opciones para optimizar la lógica de asignación de operandos a las UFs. La clave de esta lógica es que no compara todos los bits de los operandos, sino que se basa en que, por un lado existen muchos operandos que son ceros por lo que con un detector de ceros es suficiente y por otro lado, muchos de los operandos presentan tamaños pequeños (a estos operandos se les llama *valores*

narrows). Estos operandos de menor tamaño son sometidos a una extensión de signo antes de lanzarse a una UF, por lo que todos los del mismo signo tendrán una gran cantidad de bit iguales y por lo tanto se intentaran asignar a la misma UF. En la figura 3.2, se puede ver como la estación de reserva está conectada a una lógica de control de enrutamiento que es la encargada de asignar a qué UF se lanzan los operandos de una determinada instrucción.

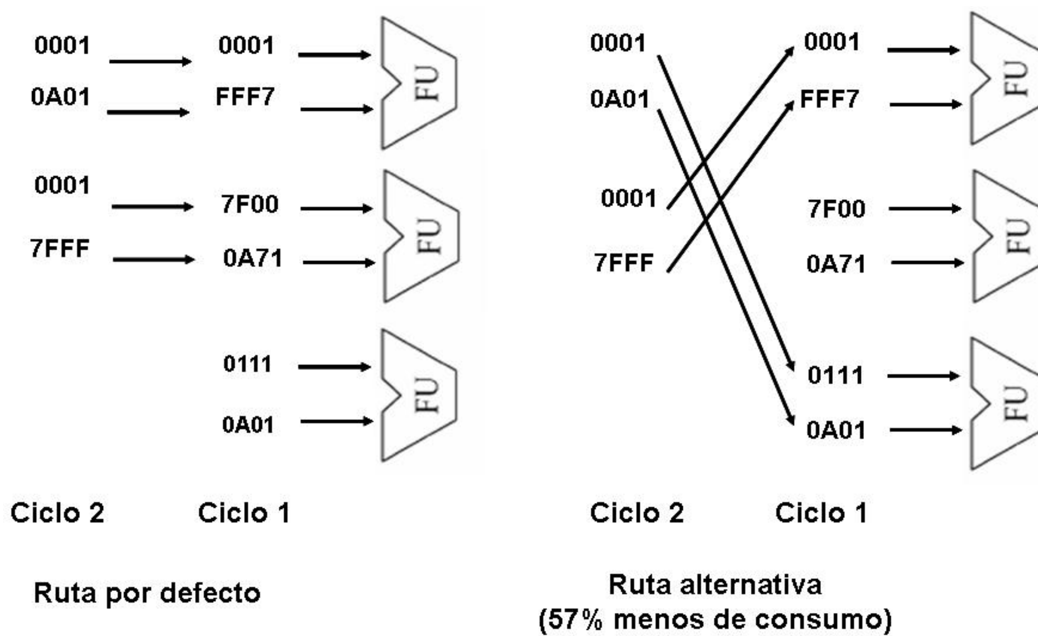


FIGURA 3.1: Rutas de datos alternativas en un procesador de 3-vias [HRBM05].

Otra manera de reducir el número de transiciones en la unidad de ejecución es aplicando la ya mencionada técnica de *gating*. Como se vio en la sección 3.1, aplicar *gating* es una de las principales tendencias en el diseño de bajo consumo de los procesadores actuales. Este mecanismo tiene diferentes grados de aplicación, desde no acceder a las estructuras hasta tenerlas completamente apagadas. Dependiendo del grado de *gating* que se aplique se conseguirá reducir el consumo total o solamente el consumo dinámico.

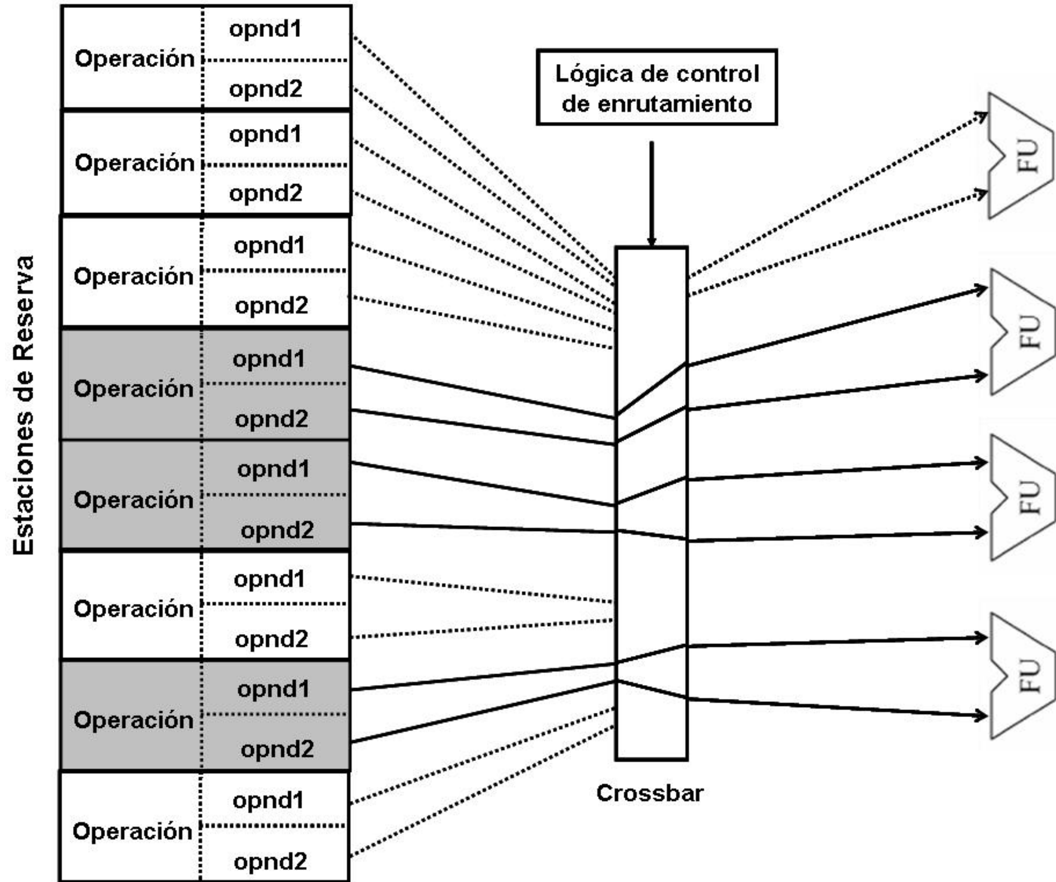


FIGURA 3.2: Implementación de la técnica propuesta en [HRBM05].

Centrándonos en técnicas que bloquean los accesos a las UFs tenemos la presentada por Brooks y Martonosi en [BM99b] [BM00]. Esta técnica detecta los operandos con valores pequeños (*valores narrows*) y los explota para reducir el consumo en la unidad de ejecución aplicando una forma de *clock gating*. Es decir, cuando se detectan operandos menores de 16 bits, los bits más significativos de los registros que contienen los operandos que entran en la UF, están bloqueados de manera que no se realizan transiciones innecesarias. Con esto se consigue reducir el número de transiciones y por lo tanto el consumo dinámico de la unidad de ejecución.

La figura 3.3 representada una de las posibles implementaciones para esta técnica. En ella podemos ver como los registros estan divididos en dos partes. La parte inferior, que contiene los 16-bits menos significativos del operando y que funciona de manera normal con la señal de reloj común al circuito, y la parte superior que contiene los bits mas significativos. Esta parte de los registros es manejada por una señal que proviene de un circuito combinacional cuyas entradas son, por un lado, la señal comun de reloj (*Clk*) y por otro unas señales encargadas de detectar si los 48 bits más significativos de los operandos son ceros o no (*Cero48-Bypass*, *Cero48-RegA* y *Cero48-RegB*). Con esta información este circuito combinacional produce una señal (*Gated Clk*) que es la encargada de hacer que la parte superior del registro esté bloqueada o no.

En el mismo trabajo, Brooks y Martonosi hacen un estudio donde demuestran que hay un alto porcentaje de operandos que como mucho ocupan 33 bits. Por lo tanto, la técnica expuesta se puede extender a los *valores narrows* de 33-bits. La implementación se muestra en la figura 3.4.

En esta nueva implementación los registro están divididos en tres partes, la parte más baja que contiene los 16 bits menos significativos del operando, la del medio que contiene del bit 16 al 33, ambos inclusive, y la parte superior con el resto de bits hasta 63. Al igual que antes, ni la parte superior ni la del medio están manejadas por la señal común de reloj. Para manejar estas partes del registro existen unas señales que proviene de circuitos combinacionales que detectan si hay que bloquearlas o no. Esta lógica de detección también se ha ampliado de manera que ahora no sólo detecta operandos positivos, es decir, detección de ceros en los bits que no forman parte del operando, (por ejemplo,

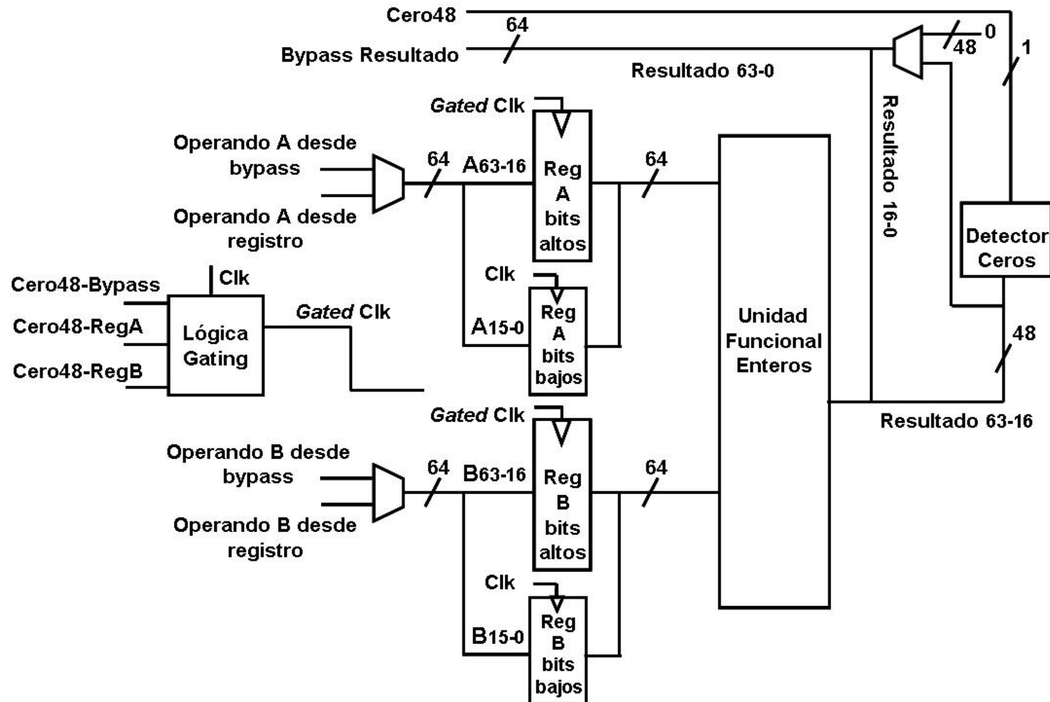


FIGURA 3.3: Primera implementación de la arquitectura de *clock gating* propuesta por Brooks y Martonosi en [BM00].

en caso de un operando positivo de 16-bits, los 48 bits más significativos son cero), sino que también detecta los valores negativos, (siguiendo con el ejemplo anterior en este caso tendría que comprobar si los 48 bits más significativos son unos). En la figura 3.4 se puede ver que el detector ya no es sólo de ceros sino que ahora detecta ceros y unos, al igual que las señales que entran en la lógica de bloqueo ya no son *Cero48-Bypass* ni *Cero48-Reg* sino que son *NarrowX-Bypass* y *NarrowX-Reg*. Con esta información, mas la señal común de reloj, el circuito de bloqueo produce dos señales *Gated Clk 63-34* y *Gated Clk 33-16* encargadas de bloquear o no las partes correspondientes de los registros.

Siguiendo con la idea de explotar los *operandos narrow* está la técnica propuesta por Choi *et ál.* en [CJC00]. En ella se explota el hecho de que, en

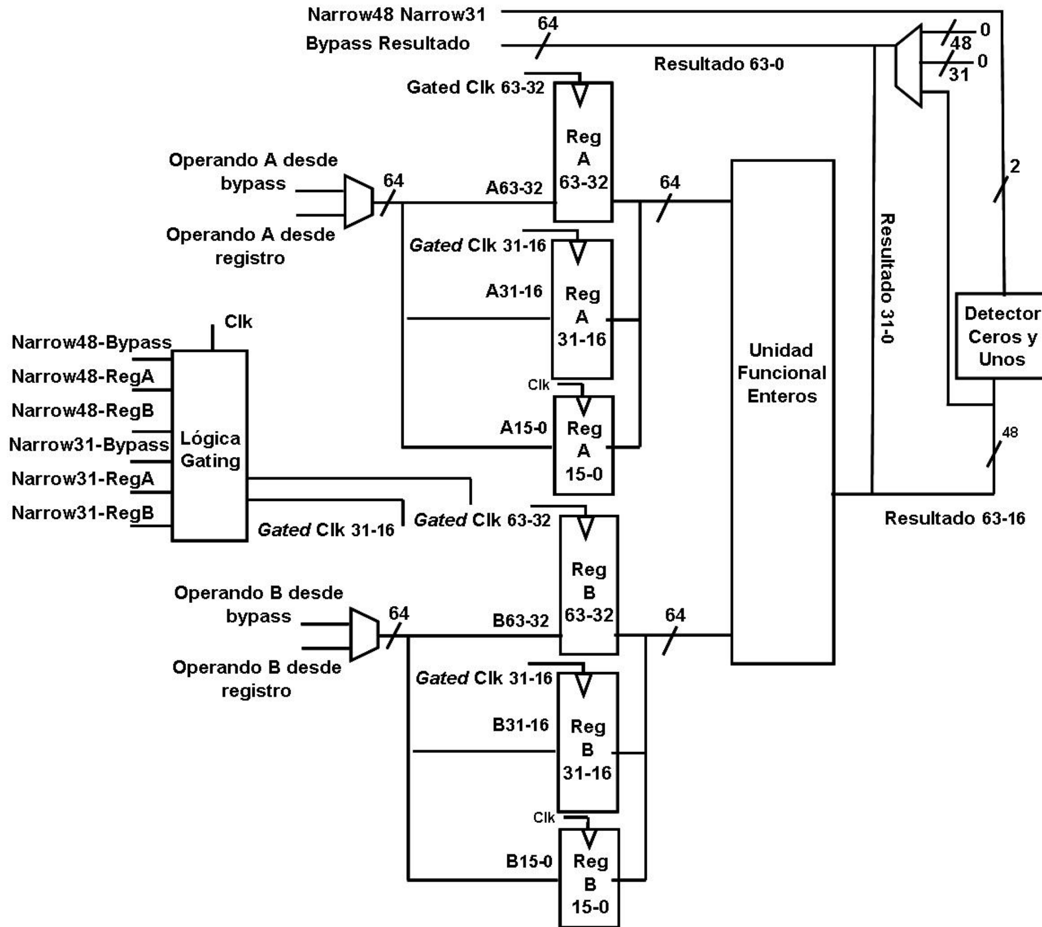


FIGURA 3.4: Segunda implementación de la arquitectura de *clock gating* propuesta por Brooks y Martonosi en [BM00].

la mayoría de los operandos, el número de bits que se necesitan para efectuar la operación es menor que el tamaño máximo (64 bits en los procesadores modernos). La técnica se llamada PGC (*Partially Guarded Computation*) y consiste en no acceder a una parte de la unidad funcional de acuerdo con el margen dinámico del dato de entrada. Como se puede ver en la figura 3.5, la idea es dividir la unidad funcional en dos partes; MSP (*Most Significant Part*) y LSP (*Least Significant Part*), de manera que si el rango del dato de entrada está dentro del rango de LSP, sólo se produzcan transiciones en

dicha parte de la unidad funcional. Para dividir la unidad funcional proponen un algoritmo que dinámicamente determina la posición de la frontera entre las dos partes, maximizando la reducción del consumo. También proponen un algoritmo para incrementar la correlación de los datos (*operand binding algorithm*), que maximiza la reducción del consumo obtenida al aplicar la técnica PGC. En la misma línea esta Cheng y su equipo, que en [CSW02] proponen sumadores y algoritmos para detectar dinámicamente el rango de los operandos y poder desactivar partes del circuito para que no se produzcan conmutaciones innecesarias.

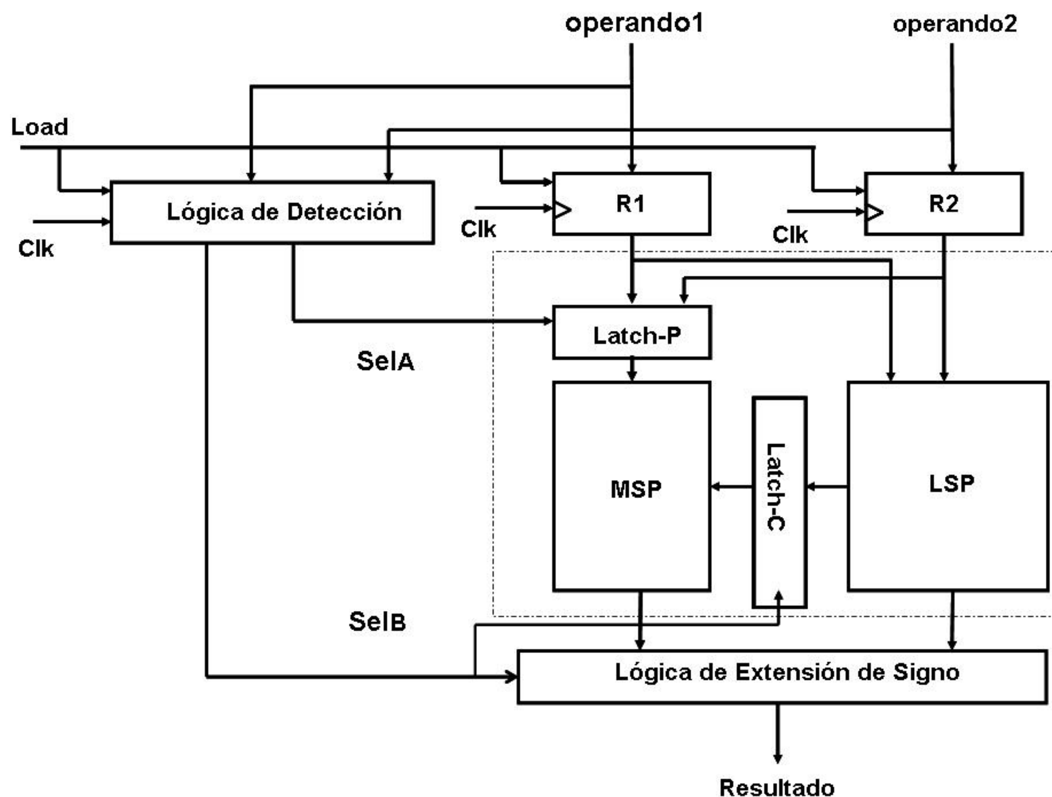


FIGURA 3.5: Implementación de la técnica propuesta en [CJC00].

En [GM03], Gandhi y su equipo presentan una comparación de las técnicas

que acabamos de describir hasta ahora. Además el mismo grupo en [GM05] siguiendo con la idea de los trabajos desarrollados por Choi [CJC00] y Brooks [BM00] proponen particionar dinamicamente las UFs no sólo en dos partes sino en más partes de manera que se pueda usar no sólo con los *operandos narrow*.

Kim propone en [Kim07] una técnica para reducir el consumo en las UFs basada en la detección dinámica de los operandos que son cero. Esta técnica explota el hecho de que existe un alto porcentaje de operandos que tiene valor cero. La propuesta consiste en lo siguiente: cuando se detecta que uno de los dos operandos de una instrucción suma, o el sustraendo de una instrucción resta, es cero, se evita el acceso al sumador y como resultado se toma directamente el valor del otro registro. De esta manera se ahorran cómputos innecesarios en las UFs y por lo tanto se reduce el consumo dinámico. Una implementación de esta propuesta se puede ver en la figura 3.6 en la cual la parte discontinua representa la lógica adicional que necesita esta propuesta.

Las técnicas que acabamos de presentar se han centrado solamente en la reducción del consumo dinámico, puesto que sólo evita cómputos en las UFs, dejando de lado el consumo estático. Como ya se ha dicho varias veces a lo largo de este trabajo, estamos en un momento en el que el consumo estático no se puede despreciar, llegando a tener una importancia casi del mismo orden que el dinámico, por lo que se hace imprescindible desarrollar mecanismos que sean capaces de reducir también la consumo estático.

Además, todas estas propuestas necesitan una lógica para detectar los *operandos narrow* u operandos con valor cero. Esta lógica no sólo tiene el inconveniente de añadir un consumo extra, sino que además introduce un retardo,

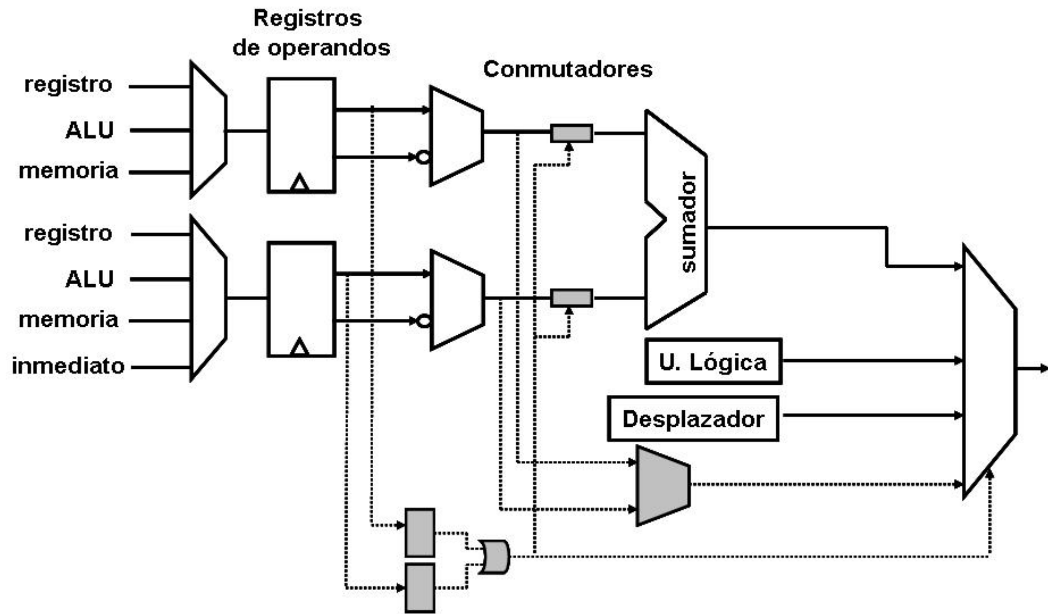


FIGURA 3.6: Implementación del detector dinámico de registros con valor cero para evitar cómputos en el sumador [Kim07].

lo cual no es interesante en los procesadores de alto rendimiento.

Existen dos técnicas bien conocidas para reducir el consumo estático en las estructuras que llevan largo tiempo sin utilizar. La primera, llamada *Power-Gating*, consiste en cortar el voltaje de alimentación de la estructura que no está siendo usada [Roy98] [PYF+00]. La segunda técnica, llamada *Dual Threshold Voltage Technology* [WV98], incrementa el voltaje de *threshold* de la estructura durante los periodos en los que esta no trabaja. Con estos mecanismos podemos conseguir grados de *gating* más fuertes y por lo tanto reducir el consumo estático.

Podemos encontrar varias propuestas en esta línea [DKA+02], [RPOG02], [HBS+04], [YAE06], [TSC07], y en todas ellas, cuando las UFs llevan un periodo de tiempo sin usarse, se envía una señal al circuito que gestiona el consumo,

que informa que dicha estructura no está trabajando. Esta manera de actuar se representa en la figura 3.7. En ella se puede ver que dentro de la unidad que gestiona el consumo existe un circuito combinacional que en función de la señal que recibe de las unidades funcionales, genera una señal que envía al mecanismo de control de corrientes de fuga para que este actue y ponga a las unidades funcionales en el modo durmiente.

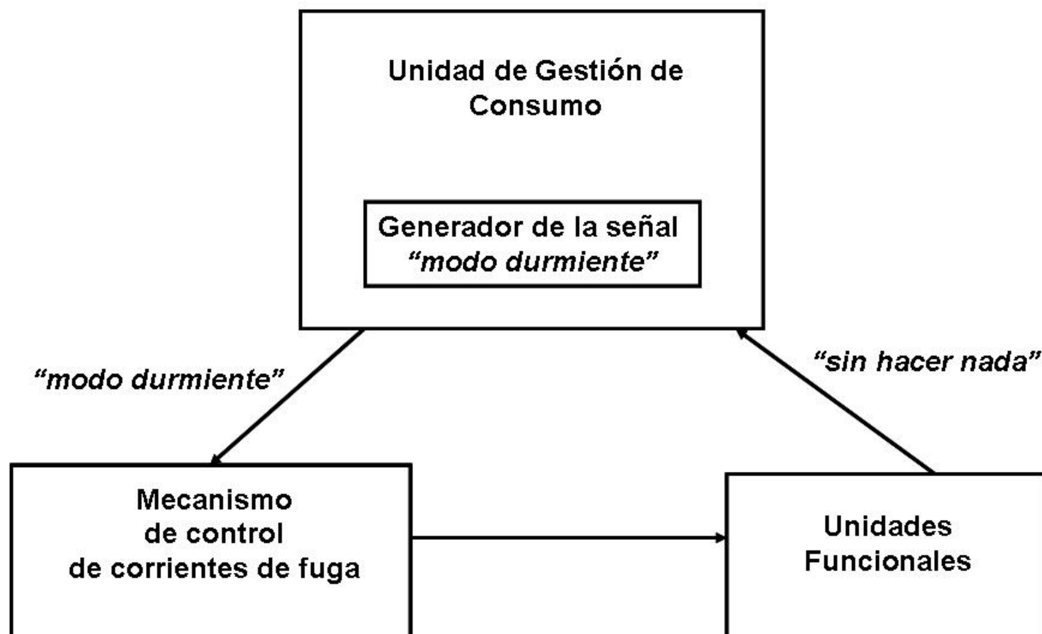


FIGURA 3.7: Control dinámico de las corrientes de fuga [YAE06].

Como acabamos de ver, estas técnicas requieren de una lógica adicional (mecanismo de control de corrientes de fuga, generador de la señal que permite entrar en modo durmiente, etc). Dicha lógica aporta un consumo extra e introduce un retardo debido a que hay que sacar a las UFs de su modo durmiente antes de volver a poder usarlas. Por lo tanto, el consumo ahorrado con estas técnicas dependerá del algoritmo usado para detectar cuando una estructura lleva tiempo sin usarse y del nivel de complejidad de los circuitos

que lo implementen.

En esta línea están los trabajos [RPOG02] y [TSC07], en los que usan el compilador para identificar largos periodos en los que las UFs estarán inactivas. Esto lo consiguen mirando los códigos de operación de las instrucciones. Esta información se comunica al *Hardware* usando directivas, de manera que durante la ejecución de un programa, el *Hardware* sepa en que periodos de tiempo las UFs no se van a usar y se pueden desconectar evitando el consumo estático. También se sabe en que momentos se vuelven a necesitar y por lo tanto hay que volver a activarlas.

Dropso en [DKA+02] estudia analítica y empíricamente distintas políticas de activación del modo durmiente para las UFs usando *dual-trheshold domino logic circuit*. En ese trabajo se presenta un esquema llamado *GradualSleep* que permite seleccionar diferentes modos durmientes de manera que se puedan usar incluso durante pequeños periodos de tiempo.

En [HBS+04], Hu propone varios métodos. Por un lado, para simplificar la lógica de detección y activación del modo durmiente, propone usar un simple contador para mantener a las UFs en estado durmiente durante un tiempo fijo cuando las UFs no estén trabajando. Este mecanismo es muy sencillo por lo que consume poco, pero en contrapartida penaliza mucho el rendimiento ya que las UFs tienen que estar un tiempo fijo en modo durmiente sin tener en cuenta si son requeridas o no. En [YAE06] se proponen algoritmos dinámicos, donde el tiempo del estado durmiente no es fijo y por lo tanto la penalización del rendimiento es menor.

Por último, hay algunas técnicas que en vez de bloquear parcial o completamente las UFs, lo que plantean es usar UFs de diferentes tipos. En esta

línea tenemos a Seng *et ál.* en [STT01] y a Yan *et ál.* en [YT06]. La diferencia entre estas técnicas es el criterio de asignación de UFs a las instrucciones. Asi tenemos que en [STT01] se propone usar UFs con distintas latencias y por lo tanto distintos consumos, de manera que a las instrucciones se les asignará una u otra UF en función de su criticidad. Es decir, las instrucciones predichas como críticas en su ejecución serán ejecutadas en la UF más rápida y por lo tanto de mayor consumo, mientras que las demás instrucciones se ejecutaran en la UF más lenta y de menor consumo. Por lo tanto se puede decir que esta técnica usa la información del camino crítico de las instrucciones para reducir el consumo en la etapa de ejecución.

Por otro lado, en [YT06] también se usan UFs con distintas latencias, pero el criterio de asignación de UFs a las instrucciones se basa en el tiempo que se tarda en usar el resultado que produce la UF. Es decir, si se dispone de dos UFs con diferentes latencias, por ejemplo de 1 y de 2 ciclos, las instrucciones cuyo resultado no se vaya a usar antes de dos ciclos serán asignadas a la UF lenta y por lo tanto de menos consumo, mientras que el resto de instrucciones serán lanzadas a la UF más rápida.

Estas dos últimas técnicas reducen tanto el consumo dinámico como el estático puesto que las UFs más lentas tienen menor consumo tanto dinámico como estático. El problema de estas propuestas es que implican una degradación en el rendimiento debido al retardo introducido por la lógica de asignación de UFs.

3.4. Conclusiones

En este capítulo se ha hecho un repaso de las técnicas a nivel de microarquitectura mas relevantes para reducir el consumo en las diferentes estructuras de los procesadores actuales. Dentro de estas, se ha dedicado una sección especial (sección 3.3) a las técnicas de bajo consumo aplicadas a las unidades funcionales ya que, además de ser una de las partes de mayor consumo, como ya se vio en el capítulo 1, es la parte del procesador en la que nos hemos centrado en este trabajo.

Como se ha visto en la sección 3.3, muchas de las propuestas aplicadas a unidades funcionales explotan el hecho de que en la ejecución de un programa existe un alto porcentaje de *operandos narrow*, [CJC00]) [BM00] [GM05] [Kim07]. El inconveniente principal de estas técnicas es que necesitan identificar que operandos son *narrows.* Esto implica añadir una lógica que chequée todos los operandos para identificar cúales son *narrow.* Esto conlleva un consumo extra, y una penalización en el rendimiento mientras se hacen estas detecciones. En este trabajo presentamos técnicas para reducir el consumo en las UFs que también explotan el alto porcentaje de *operandos narrow* que exiten en los programas, pero que a diferencia de estas propuestas, las nuestras, para la detección de estos valores, no necesitan chequear todos los bist de los operandos puesto que nos basamos en el código de operación de la instrucción, de manera que la lógica de detección es muy sencilla ya que puede reusar *Hardware* que ya existe. Esto hace que practicamente no se añada consumo extra.

Otra de las desventajas de las anteriores técnicas es que estas sólo reducen

el consumo dinámico. En nuestras propuestas usamos UFs de diferentes consumos, al igual que en [STT01] [YT06], por lo que no sólo se consigue reducir el consumo dinámico sino también el estático.

Por último, también hemos visto técnicas que reducen ambos consumos apagando la unidad de ejecución cuando las UFs llevan un periodo de tiempo sin usarse, como las propuestas en [DKA+02] [RPOG02] [HBS+04] [YAE06] y [TSC07]. Pero estos mecanismos requieren de una lógica adicional que detecte cuando una unidad funcional no se está utilizando y genere una señal que la permita entrar en modo-durmiente. Dicha lógica aporta un consumo extra e introduce un retardo debido a que hay que sacar a las UFs de su modo-durmiente antes de volver a poder usarlas. Nuestras propuestas, debido a la sencillez de la lógica de asignación de UFs y a que no necesitamos usar técnicas de *gating* para reducir el consumo estático, ya que esto lo hacemos usando diferentes modelos de UFs, no introducen practicamente penalización en el rendimiento. Por otro lado, son compatibles con estas técnicas de *gating* por lo que se podría incluso conseguir una mayor reducción del consumo.

Capítulo 4

Adaptación de un simulador de potencia para unidades funcionales en procesadores de alto rendimiento

Una optimización en el diseño, o cualquier modificación en la arquitectura de los procesadores, es aceptada por la comunidad científica si una herramienta de simulación de potencia indica una reducción en el consumo sin un incremento significativo en el tiempo de computación. Por ello, la precisión en las estructuras, la velocidad y la exactitud de la microarquitectura simulada son de importancia crítica para los investigadores. Como ya se ha dicho en el capítulo 2, *SimpleScalar* [ALE02] se ha convertido en la principal herramienta de simulación y modelado de sistemas de alto rendimiento. Existen varios simuladores que añaden a dicha herramienta un modelo para el cálculo de consumo

de las distintas estructuras del procesador. *Wattch* [BTM00] es uno de ellos y el más usado en los artículos científicos presentados en conferencias de arquitectura de computadores internacionales. Sin embargo, este simulador estima el consumo en las UFs de manera poco precisa. Por ejemplo, tiene modeladas las UFs para calcular el consumo de distinta forma a como están modeladas funcionalmente. Es decir, si se ejecuta una instrucción lógica (por ejemplo una AND), funcionalmente se realiza un acceso a la unidad lógica, y sin embargo a la hora de calcular el consumo lo que se tiene en cuenta es un acceso a un sumador de enteros.

En este capítulo se presenta una versión del simulador *Wattch*, que llamaremos *FU-Wattch*, a la cual le hemos añadido algunas modificaciones para estimar el consumo en las UFs con mayor precisión. Con esto pretendemos preparar el simulador para poder disponer de una herramienta que sea capaz de evaluar de manera precisa las técnicas, para reducir el consumo en la unidad de ejecución, que proponemos en este trabajo.

Los objetivos que pretendemos conseguir al modificar *Wattch* son los siguientes:

- Modificar el simulador *Wattch* para estimar el consumo en la unidad de ejecución con más exactitud.

- Mejorar del modelo de las Unidades Aritmético Lógicas, tanto de enteros como de punto flotante.

- Separar el estudio del consumo dinámico y estático de potencia

- Mejorar la fiabilidad cuando se realizan medidas del consumo, si el sis-

tema esta utilizando la técnica de clock gating en la unidad de ejecución

El resto del capítulo está organizado de la siguiente forma: la sección 4.1 presenta el simulador *FU-Wattch* y las incorporaciones hechas para estimar el consumo en las UFs con más precisión. La sección 4.2 muestra algunos resultados experimentales que validan el simulador *FU-Wattch*. La sección 4.3 presenta una comparación entre el consumo estimado por *Wattch* y el consumo estimado por *FU-Wattch*, donde se puede apreciar la mejora que se realiza con el nuevo simulador. Y por último, en la sección 4.4 se expondrán algunas conclusiones.

4.1. Simulador *FU-Wattch*

Como ya se ha comentado anteriormente, *FU-Wattch* está basado en *SimpleScalar* [ALE02]. *SimpleScalar* es un simulador que proporciona un modelo preciso de procesadores de alto rendimiento. Como modelo de consumo para las distintas estructuras del procesador, excepto las unidades funcionales, *FU-Wattch* usa el modelo de consumo de *Wattch* [BTM00] que, como ya se ha visto en el capítulo 2, es un simulador para analizar el consumo de los procesadores a nivel de arquitectura. Para modelar el consumo en las unidades funcionales se han incluído algunas mejoras que permiten obtener valores de consumo en la unidad de ejecución que son más próximos a los valores mencionados en la literatura que los que se obtienen con el *Wattch*.

Veamos primero un breve resumen con las principales características del simulador *Wattch* para después describir las modificaciones que hemos añadido, así como el modelo de consumo usado en las UFs.

4.1.1. Características de *Wattch*

Wattch es un simulador presentado en la conferencia ISCA 2000 [BTM00], basado en *SimpleScalar* y que añade un modelo de potencia/consumo. *Wattch* toma como entrada la descripción del procesador y una traza o ejecutable a simular. Tras la simulación de la traza, *Wattch* muestra la máxima potencia de pico de cada una de las unidades que componen el procesador, así como el consumo de potencia producido durante la ejecución para cuatro estilos diferentes de *Clock Gating*. El *Clock Gating* es una técnica que inhibe la señal de reloj a determinadas zonas de un circuito para que no haya transiciones en las señales durante los ciclos que esa zona está inactiva y de esta forma evitar consumo innecesario. Los cuatro estilos definidos en *Wattch* son los siguientes:

- **NCC:** no aplicar *Clock Gating*.

- **CC1:** *Clock Gating simple*. Es decir, una unidad o bien esta inactiva, con un consumo nulo, o bien su consumo es el 100 % del consumo máximo.

- **CC2:** *Clock Gating lineal ideal*. En esta opción, el consumo de una unidad es lineal. Es decir, cuando hay acceso a una estructura, el consumo se calcula en función de los recursos utilizados durante cada ciclo (por ejemplo, el número de puertos a los que se ha accedido en un banco de registros), mientras que si no hay acceso a la estructura el consumo es nulo.

- **CC3:** *Clock Gating lineal no ideal*. Este estilo es igual que el anterior cuando se accede a una estructura. Pero, en caso de que la unidad no

se utilice, el consumo de dicha estructura se estima como un 10 % de consumo total.

Para obtener estos resultados, *Wattch* calcula la potencia máxima de cada una de las estructuras antes de iniciar la simulación de la traza o del ejecutable. Luego, durante la simulación propiamente dicha, en cada ciclo cuenta el número de accesos a cada estructura, para al finalizar el ciclo de simulación calcular, utilizando los cuatro estilos diferentes de *Clock Gating*, que consumo se ha producido y acumularlo en unos contadores de consumo global de la simulación.

Para modelar el consumo de cada una de las estructuras del procesador simulado, *Wattch* las organiza en cuatro clases de bloques: unidades tipo RAM, unidades tipo CAM (lógica asociativa), unidades basadas en lógica combinacional y distribución de reloj.

- ***Unidades tipo RAM***

  Estas son las estructuras en array. En esta categoría se incluyen las *caches* de datos e instrucciones, los arrays de etiquetas en la *cache*, todos los registros, RAT (*Register Alias Tables*), predictores de saltos y una gran parte de la ventana de instrucciones y de la cola de *load/store*.

- ***Unidades tipo CAM***

  Aquí se incluyen todas las estructuras que sean memorias totalmente asociativas direccionables por contenido. Estas son, entre otras, la lógica de *wake-up* de la ventana de instrucciones y del *buffer* de reordenamiento, los módulos de comprobación del orden de *load* y *store*, y los TLBs.

- ***Unidades basadas en lógica combinacional***

  En este grupo estarían las unidades funcionales, la lógica de selección de la ventana de instrucciones, el comprobador de dependencias y los buses de resultado.

- ***Distribución de reloj***

  Este grupo incluye todas las señales de reloj (incluyendo *buffers*, líneas de reloj y cargas capacitivas).

En *Wattch*, las estructuras incluidas en los dos primeros tipos de bloques están modeladas de forma genérica mediante funciones que calculan la potencia de las distintas partes de una memoria RAM y de una memoria CAM en función del número de filas, columnas y puertos. Estas funciones, realizan llamadas a funciones de la librería de la herramienta CACTI [WJ96], para calcular la capacidad de los distintos componentes. Además se utiliza CACTI para obtener los parámetros de *subbanking* (partición de la memoria en bancos) de la memoria *cache*.

A continuación se describe como se modelan algunas de las estructuras más representativas que están implementadas en el simulador:

- ***Lógica de renombre, decodificación y predictor***

  Wattch calcula el consumo de la lógica de renombramiento sumando el consumo de la tabla de renombre RAT (*Register Alias Table*), la cual es modelada como una memoria RAM, más el consumo de la lógica de chequeo de dependencias (DCL) considerando esta lógica como un circuito combinacional. Además se añade la potencia de la lógica de

decodificación, que es considera una estructura de tipo RAM, y la de un predictor de saltos convencional, cuya consumo se modela como la suma de varias tablas RAM. No se tienen en cuenta las lecturas al predictor.

- ***Ventana de instrucciones y cola Load-Store***

  El consumo de la ventana de instrucciones se calcula como la suma de tres componentes: lógica de despertado (lógica de *wake-up*), lógica de selección y estaciones de reserva. La lógica de despertado se modela como una memoria CAM, la lógica de selección es considerada como un circuito combinacional y las estaciones de reserva como una memoria RAM.

  El consumo de la cola *Load-Store* se calcula como la suma de dos componente: una memoria CAM para la lógica de despertado más una memoria RAM para los registros. *Wattch* incluye además una función que calcula la potencia en el bus de resultados.

- ***Memorias cache***

  La potencia de las *caches*, tanto de primer nivel, de datos y de instrucciones, como de segundo nivel, mixta, se modela como una estructura de tipo RAM para las etiquetas más otra RAM para los datos. El resultado de este cálculo se multiplica por el número de puertos y por el número de sub-bancos. La herramienta CACTI es la encargada de encontrar la mejor opción de *subbanking*.

  El consumo del TLB, tanto de datos como de instrucciones, se modela como una estructura de tipo CAM más una de tipo RAM.

- ***Reloj***

  Wattch define una función para el cálculo de la potencia disipada por la red de distribución del reloj. A esta potencia contribuyen tres factores: la capacidad de las líneas de distribución de la señal, los *buffers* intermedios y la carga de los registros a los que llega la señal.

Existen ciertas estructuras, como por ejemplo el banco de registros, para las que *Wattch*, además de contar el número de accesos, se tiene en cuenta la actividad conmutación a nivel de bit, en los resultados. Estos factores de actividad se miden ejecutando *Benchmarks* en un simulador de la arquitectura (por ejempo *SimpleScalar*). Para los subcircuitos en los que no se puede medir la actividad con el simulador, se supone una actividad por defecto de 0.5. Esto permite calcular de modo más fino el consumo en estas estructuras.

4.1.2. Aspectos a mejorar en *Wattch*

Las características de *Wattch* hacen que sea una herramienta muy útil a la hora de realizar un estudio en arquitectura de computadores. Sin embargo, *Wattch* no está libre de errores. El primer problema con el que se encuentra un usuario de *Wattch* es la falta de documentación del simulador, ya que la única documentación disponible se limita al artículo presentado en el ISCA 2000 [BTM00]. Este artículo da pocos detalles de su funcionamiento interno y del modelo de procesador que utiliza a la hora de calcular el consumo.

Otra desventaja que surge al utilizar *Wattch* es la falta de flexibilidad a la hora de definir parámetros de consumo como la tecnología de fabricación y la frecuencia de reloj. Ambos parámetros son clave en el cálculo de la poten-

cia/consumo y no pueden ser modificados sin recompilar el simulador.

Entrando más a fondo en la implementación del simulador se pueden observar otros inconvenientes como, por ejemplo, modelar el consumo de un procesador basado en *Buffer* de reordenamiento más ventana de intrucciones [PJS97] usando un simulador basado en Estaciones de reserva.

Respecto al consumo, en los resultados que aporta *Wattch* no hay ninguna información sobre el consumo estático, siendo este interesante debido a que el consumo estático está creciendo en importancia con respecto al consumo dinámico según aumenta la escala de integración.

Y por último, el punto que más afecta para el desarrollo de este trabajo es que *Wattch* estima el consumo en las UFs de manera poco precisa. Es decir, tiene modeladas las UFs para calcular el consumo de distinta forma a como están modeladas funcionalmente. Por ejemplo, *Wattch* considera una UF para la multiplicación de enteros. Sin embargo a la hora de calcular el consumo asociado a una instrucción de multiplicación de enteros, *Wattch* supone que el consumo de esta operación es el mismo que el asociado al sumador de enteros. Lo mismo ocurre con otras UFs, como lo unidad lógica, etc.

Como ya hemos comentado en el capítulo 1, la unidad de ejecución es una de las partes del procesador que más consumen, por lo tanto es necesario que las herramientas, en este caso *Wattch*, nos proporcionen datos fiables sobre el consumo en estas estructuras, de manera que permitan evaluar el impacto y la bondad de las propuestas que son uno de los objetivos de este trabajo.

4.1.3. Modelo de las UFs

Se ha elegido un modelo de UFs similar a la organización clásica del *Alpha* 21264 [Kes99] pero no idéntico. La razón es poder simular procesadores más generales. La figura 4.1 muestra un esquema del modelo que se ha elegido. En ella se puede ver que la unidad de ejecución está compuesta por una unidad de enteros y otra de punto flotante. La unidad de enteros esta formada por dos *clusters*, cada uno de los cuales esta dividido en dos *subclusters*. Cada unos de estos *subclusters* está formado por un sumador de 64-bits (Sumador), una unidad lógica (U. Lógica) y una unidad de desplazamientos (U. Desplaza). Estos cuatro *subclusters* son simétricos salvo porque uno de ellos contiene además un multiplicador de enteros (Multiplicador). Por otro lado la unidad de punto flotante (FP, *Floating Point*) está formada por un *cluster* compuesto de un sumador (Sumador FP) y un multiplicador (Multiplicador FP) ambos en punto flotante.

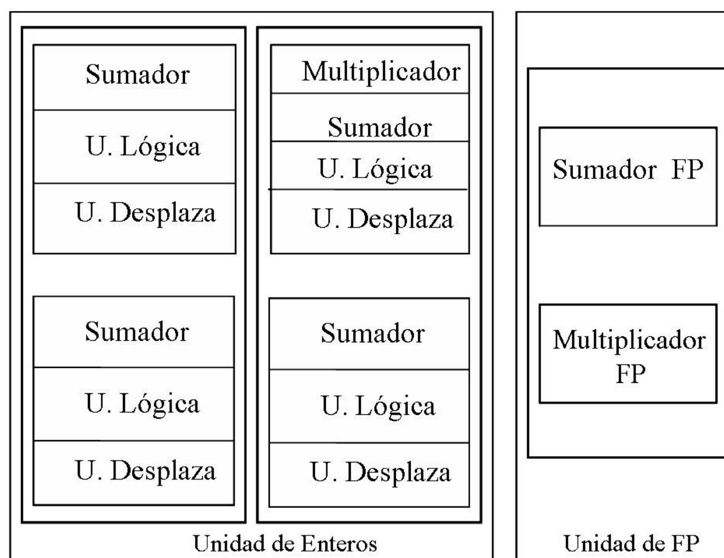


FIGURA 4.1: Modelo de la unidad de ejecución que se ha implementado.

Para estimar el consumo en las UFs con el modelo que se acaba de describir se han distribuido las instrucciones como muestra la tabla 4.1. En ella se puede ver que, por ejemplo, a efectos de cálculo de consumo, todas las instrucciones lógicas se procesan como si se ejecutaran en la unidad lógica. Esto significa que cuando se ejecuta una instrucción lógica sobre un *subcluster* sólo habrá consumo dinámico y estático en la unidad lógica mientras que el resto de componentes sólo aportarán consumo estático.

TABLA 4.1: Unidades funcionales y sus instrucciones asociadas.

UFs para enteros	**Tipo de Instrucción de enteros**
Desplazador	Instrucciones de desplazamiento Instrucciones de manipulación de Bytes
Unidad Lógica	Lógicas Integer Condicional Move Comparaciones lógicas
Sumador	Aritméticas de enteros Comparaciones aritméticas Cálculo de la dirección efectiva en instrucciones *Load/Store* De control
Multiplicador	Multiplicación de enteros
UFs para PF	**Tipo de Instrucción de PF**
Sumador	Aritméticas en PF (salvo multiplicación, división raiz cuadrada)
Multiplicador	Multiplicación en PF División en PF Raíces cuadradas

4.1.4. Modificaciones hechas a *Wattch*

Como ya se ha comentado anteriormente, *FU-Wattch* es una versión modificada del simulador *Wattch* donde hemos incluido algunas mejoras que permiten obtener valores de consumo en las UFs más próximos a los valores mencionados en la literatura que los que se obtienen con *Wattch*. A continuación se explican las modificaciones que hemos implementado y sus justificaciones:

1. En términos de consumo de potencia, *Wattch* modela la unidad de ejecución de enteros como un sumador. Para simular de manera más fidedigna el comportamiento funcional de dicha unidad le hemos incluido más componentes, de manera que en *FU-Wattch* la unidad de enteros esta compuesta por un sumador, una unidad lógica y una unidad de desplazamiento, como representa la figura 4.1.

2. Por otro lado, *Wattch* considera una UF para la multiplicación de enteros. Sin embargo a la hora de calcular el consumo asociado a una instrucción de multiplicación de enteros, *Wattch* supone que el consumo de esta operación es el mismo que el asociado al sumador de enteros. En *FU-Wattch* hemos incluido tanto consumo estático como dinámico de multiplicadores.

3. Lo mismo ocurre con las instrucciones de punto flotante. Todas ellas, sin importar que tipo de unidad funcional necesiten, son modeladas en *Wattch*, como operaciones de suma. En *FU-Wattch* hemos incluído en el modelo de consumo de las unidades funcionales un sumador y un multiplicador de punto flotante.

4. Por otro lado, *Wattch* muestra un solo valor de consumo. En dicho valor, sólo si se usa el modelo cc3 (*Linear Clk gating* w/ 10 %) [BTM00], esta incluído el consumo estático. En este modelo (explicado en la sección 4.1.1) cuando no se producen accesos a una estructura, se tiene en cuenta el consumo estático que viene representado por el 10 % del consumo total de dicha estructura. En el caso de las UFs también funciona así cuando no se accede a ninguna UF, pero cuando sólo hay alguna/s UFs sin usar, el consumo estático de esta/s no se tiene en cuenta. Esto hace que *Wattch* no estime bien el consumo estático en la unidad de ejecución. Este defecto lo hemos corregido en *FU-Wattch*, de manera que ahora se tiene en cuenta el consumo estático de las UFs en todos los casos.

5. Además en *FU-Wattch* el consumo total lo hemos separado en sus dos componentes: dinámica y estática. Como ya se ha comentado en el capítulo 1, la reducción del tamaño de integración hace que el consumo estático represente cada vez una proporción mayor del consumo total. En concreto las previsiones estiman que para tecnologías por debajo de los 65nm el porcentaje de consumo estático respecto al total podrá superar el 50 %. [TSC07] [YAE06] [BS00]. En *FU-Wattch* hemos tomado como porcentaje que representa el consumo estático el 25 % del consumo total, valor válido hasta tecnologías de 80nm.

6. Por último, *Wattch* tiene adaptada la técnica de *clock gating* a todas las estructuras salvo a la unidad de ejecución. En *FU-Wattch* hemos adaptado esta técnica también a las unidades funcionales de manera que sólo se tiene en cuenta el consumo dinámico en las unidades funcionales que

estén ejecutando alguna instrucción. Es decir, si sólo se está ejecutando una instrucción lógica, unicamente hay consumo dinámico en la unidad lógica del *subcluster* donde se esté ejecutando dicha instrucción, aportando, tanto el resto de UFs de dicho *subcluster* (sumador, unidad de desplazamiento y, si hay, multiplicador) como el resto de los *clusters*, sólo consumo estático.

4.2. Validación del simulador *FU-Wattch*

Para validar la precisión del modelo de consumo en las UFs del simulador *FU-Wattch*, se han ejecutado un subconjunto de *Benchmarks* del conjunto SPEC CPU2000 sobre dicho simulador y sobre el simulador *Sim-Alpha* [DBKA01] con el fin de comparar ambos resultados. El objetivo de esta sección es poder comparar los resultados estimados con los dos simuladores con valores reales de consumo de la arquitectura *Alpha* aportados en la literatura [BTM00]. Se ha elegido el *Sim-Alpha* para esta comparación porque aunque ambas herramientas simulan la arquitectura del *Alpha* 21264, la precisión del *Sim-Alpha* es mayor.

Sim-Alpha es un simulador basado en *SimpleScalar* [ALE02] [BAB96] pero que simula de manera más precisa que *SimpleScalar* las características del *Alpha* 21264, como se demuestra en [DBKA01]. De manera que, en términos de funcionalidad, simula perfectamente el modelo de UFs de la arquitectura *Alpha* 21264 [Kes99]. Para que este simulador haga estimaciones de consumo le hemos incorporado el mismo fichero para calcular el consumo que usa *FU-Wattch*, adaptandolo a las UFs de la arquitectura *Alpha* 21264.

La figura 4.2 representa la unidad de ejecución del *Alpha* 21264. En ella se puede ver que la unidad de ejecución está compuesta por una unidad de enteros y otra de punto flotante. La unidad de enteros esta formada por dos *clusters*, cada uno de los cuales esta dividido en dos *subclusters*. Estos cuatro *subclusters* son simétricos dos a dos, de manera que los *subclusters* inferiores (el 2 y 4) contienen un sumador de 64-bits (Sumador), una unidad lógica (U. Lógica) y un sumador para calcular las direcciones efectivas de las instrucciones *load* y *store* (Sumador L/S). Mientras que los *subclusters* superiores (el 1 y 3) estan formados por un sumador de 64-bits (Sumador), una unidad lógica (U. Lógica), una unidad de desplazamientos (U. Desplaza) y una unidad para calcular la dirección de los saltos (U. de Salto). Además existe un multiplicador de enteros (Multiplicador) en el *subcluster* 3 y una unidad para ejecutar instrucciones especiales como PERR, MINxxx (MVI/PLZ) en el 4. Por otro lado la unidad de punto flotante está formada por un *cluster* compuesto de dos *subclusters*. El *subcluster* inferior esta compuesto de un sumador (Sumador FP), un divisor (Divisor FP) y una unidad para calcular raices cuadradas (Raíz Cuadrada FP). Mientras que el superior contiene solamente un multiplicador (Multiplicador FP).

También se ha adaptado el modelo de UFs del *FU-Wattch* (que como se ha comentado anteriormente es similar pero no igual a la clásica organización del *Alpha* 21264) para que los dos simuladores tengan modeladas las mismas UFs. Para ello al modelo usado en *FU-Wattch* (ver figura 4.1) se le ha añadido una unidad para el cálculo de raíces cuadradas en punto flotante, una unidad de división en punto flotante, dos unidades para calcular las saltos condicionales, dos sumadores de enteros para generar la dirección efectiva de las instrucciones

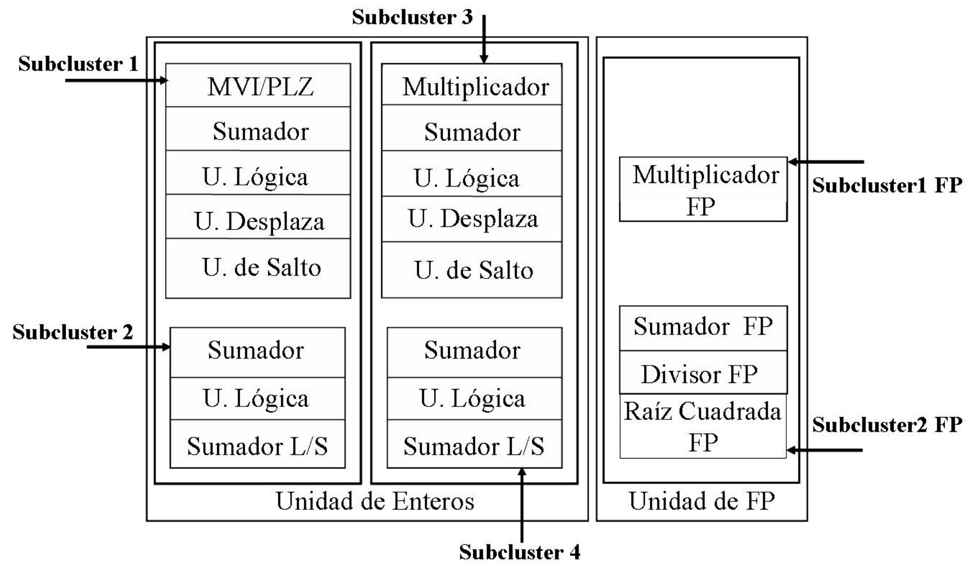


FIGURA 4.2: Modelo de la unidad de ejecución de la arquitectura *Alpha* 21264.

load y *store*, y una unidad para ejecutar instrucciones especiales como PERR, MINxxx.

Una vez adaptados los dos simuladores, *Sim-Alpha* con cálculo de potencia en las UFs y *FU-Wattch* con el modelo de UFs de la arquitectura del *Alpha* 21264, se ha ejecutado un subconjunto de *Benchmarks* del conjunto SPEC CPU2000 [SPE]. Se han seleccionado los *Benchmarks* de manera que se cubre todo el rango de comportamientos. Además, como la simulación completa de cada uno de ellos puede llevar semanas y no aporta más precisión en las medidas, se ha usado la herramienta *Sim-point* [PHC03] para seleccionar un conjunto de instrucciones que represente la ejecución total del programa. De esta forma, de cada *Benchmarks* se han simulado 100M de instrucciones elegidas con *Sim-point* [PHC03].

En la tabla 4.2 se pueden ver las caracteristicas de los *Benchmarks* seleccionados para las simulaciones. En la primera columna se muestra el nombre

de cada uno de los *Benchmarks*, en la segunda columna se muestran las entradas que se han usado para cada uno de ellos, como se puede observar la mayoría usan las entradas de referencia (ref), y en la última columna aparece la ventana de instrucciones elegida, mediante la herramienta *Sim-point*, para cada *Benchmark*.

TABLA 4.2: *Benchmarks* del conjunto SPEC CPU2000 que se han elegido para las simulaciones.

Benchmarks	**Datos de Entrada**	**Ventana de Simulación**
	Enteros	
bzip02	source 58	100M-200M
crafty	ref	1000M-1100M
cc1	166.i	2000M-2100M
gzip	source 60	100M-200M
mcf	ref	1000M-1100M
parser	ref	100M-200M
twolf	ref	1000M-1100M
vpr	ref	190M-290M
	Punto Flotante	
applu	ref	300M-400M
art	ref	300M-400M
equake	ref	100M-200M
galgel	ref	100M-200M
lucas	ref	100M-200M
mesa	ref	100M-200M
mgrid	ref	100M-200M
swim	ref	1000M-1100M

En las tablas 4.3, 4.4 y 4.5 se pueden ver las principales características del procesador que se ha simulado en *FU-Wattch*. En el simulador *Sim-Alpha* se han usado los mismos parámetros salvo las particularidades relacionadas con las diferencias entre sus algoritmos de planificación.

Tabla 4.3: Configuración del núcleo del procesador simulado.

Parámetros	Valores
RUU tamaño	80 instrucciones
LSQ (ld/store queue)	tamaño 32
Tamaño de la cola Fetch	4 instrucciones
Ancho Fetch	4 instrucciones/ciclo
Ancho Decode	4 instrucciones/ciclo
Ancho Issue	6 instrucciones/ciclo (fuera de orden)
	(4 de enteros, 2 de PF)
Ancho Commit	11 instrucciones/ciclo (en orden)
Unidades Funcionales	4 *Subclusters* de enteros
	2 de PF

Como se puede ver en la tabla 4.3, el procesador puede lanzar hasta seis instrucciones por ciclo y fuera de orden de la cola de lanzamiento (cuatro de operaciones con enteros y dos en PF). Esto es posible ya que en la unidad de ejecución existen cuatro *subclusters* de enteros y dos de punto flotante (ver figuras 4.1 y 4.2).

Tabla 4.4: Configuración del predictor del procesador simulado.

Parámetros	Valores
Bimodal	tamaño de la tabla 4K
De 2-niveles	1K cada nivel, 3 bits de historia
	tamaño de la meta-tabla 4K
BTB	512 conjuntos, asociatividad 4
Return-address stack	32 entradas
Penalización del fallo de predicción	7 ciclos

TABLA 4.5: Jerarquía de la memoria del procesador simulado.

Parámetros	Tamaño	Bloques	Asocitividad	Latencia
L1 datos	512K	64	2	3
L1 instrucciones	512K	64	2	2
L2 unificada	32M	64	1	6 (datos) 12 (instruc)
TLBs	**128 entradas completamente asociativa**			50

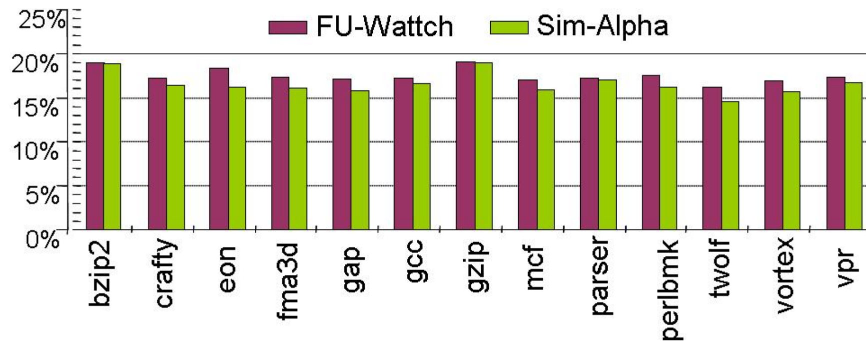
La tabla 4.6 resume el número, tipo y consumo de las distintas UFs que se ha usado en las simulaciones. Los valores de consumo se han obtenido de [BTM00] y el número y tipo de UFs han sido escogidos para asemejarse en todo lo posible a la microarquitectura del *Alpha* 21264 [Kes99].

TABLA 4.6: Consumo estimado de cada unidad funcional a 5V, 733MHz y 180nm [BTM00].

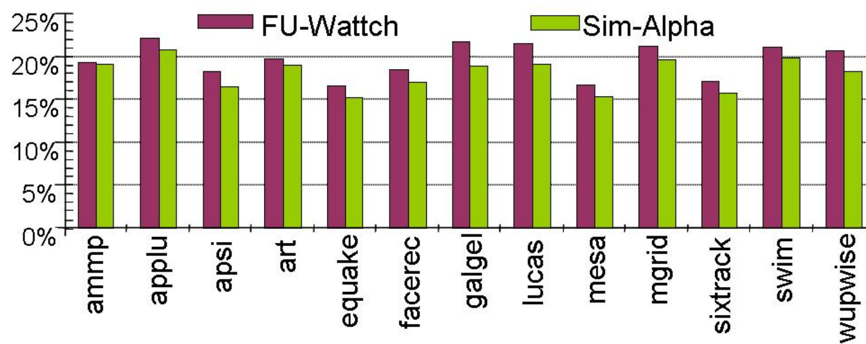
Tipo de UF	Consumo por unidad (w)	UFs disponibles
UFs de Enteros		
Sumador(64-bits)	1.16503	4
Unidad Lógica	0.104394	4
Unidad de Desplaz	0.505265	2
Multiplicador(16-bits)	0.897785	1
Unidad para L/S	1.16503	2
Unidad de Salto	1.7703	2
Unidad Especial	1	1
UFs de PF		
Sumador	3.57026	1
Multiplicador	3.57026	1
Divisor	3.57026	1
SQRT	3.57026	1

La métrica usada para obtener datos de consumo es la potencia media (*averaged power*). Este valor se obtiene sumando la potencia consumida en

cada ciclo sobre un número definido de ciclos de reloj.



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 4.3: Porcentaje del consumo de la unidad de ejecución respecto al consumo total del procesador obtenido con *Sim-Alpha* y *FU-Wattch*.(a) muestra los resultados para INTSPEC 2000, y (b) muestra los resultados para FPSPEC 2000.

Los resultados de las simulaciones se puede ver en la figura 4.3. En ella se muestra el porcentaje que representa el consumo total en la unidad de ejecución respecto al consumo total del procesador, obtenido con los simuladores *Sim-Alpha* y *FU-Wattch*. De ellas, la figura 4.3(a) muestra dicho consumo para todos los *Benchmarks* de enteros, mientras que la figura 4.3(b) muestra los resultados para todos los *Benchmarks* de punto flotante. En el caso del *Sim-Alpha* para calcular este porcentaje se ha usado el consumo total del procesador obtenido con *FU-Wattch* ya que *Sim-Alpha* solo tiene incorporado el consumo

en las UFs.

Como podemos observar en ambas figuras, los resultados obtenidos con el simulador *FU-Wattch* son muy cercanos a los obtenidos con *Sim-Alpha*. Las pequeñas diferencias en los resultados son debidas a que, aunque las dos herramientas simulan la arquitectura del *Alpha* 21264, presentan ligeras diferencias en el algoritmo implementado. Por otro lado, el porcentaje de consumo que representan las UFs respecto al resto del procesador en ambos casos es prácticamente el que se encuentra en la literatura del *Alpha* 21264 (alrededor del 20 %), [BTM00], [WM04]. Estos resultados validan las incorporaciones introducidas en *Wattch*.

4.3. Comparación de los simuladores *Wattch* y *FU-Wattch*

Con el objetivo de valorar la mejora introducida por *FU-Wattch* en esta sección se muestra una comparación entre los resultados obtenidos con el simulador *Wattch* y *FU-Wattch*. En estas simulaciones se han usado los mismos *Benchmarks* y la misma configuración del procesador que en la sección anterior. La única diferencia en la configuración es el modelo de consumo de las UFs de cada uno de los simuladores.

En *FU-Wattch* el modelo de UFs es el descrito en la sección 1 (ver figura 4.1) que como ya se ha comentado es similar a la clásica organización del *Alpha* 21264 [Kes99] pero no idéntico, ya que el objetivo es usar un modelo que simule procesadores más generales.

Por otro lado *Wattch* mantiene el modelo original, que, como ya se ha dicho anteriormente, consiste en modelar, en términos de consumo, las UFs como sumadores de enteros y de punto flotante. Es importante recalcar que esta diferencia en el modelo de las UFs afecta a la estimación del consumo ya que el comportamiento funcional de dichas unidades es idéntico.

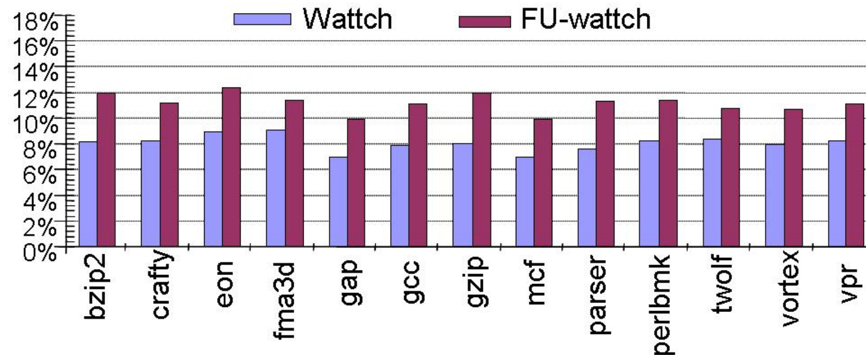
La tabla 4.7 resume el número, tipo y consumo de las distintas UFs para cada uno de los simuladores. Al igual que en la sección anterior, los valores de consumo se han obtenido de la literatura [BTM00].

TABLA 4.7: Consumo estimado de cada unidad funcional a 5V, 733MHz y 180nm [BTM00].

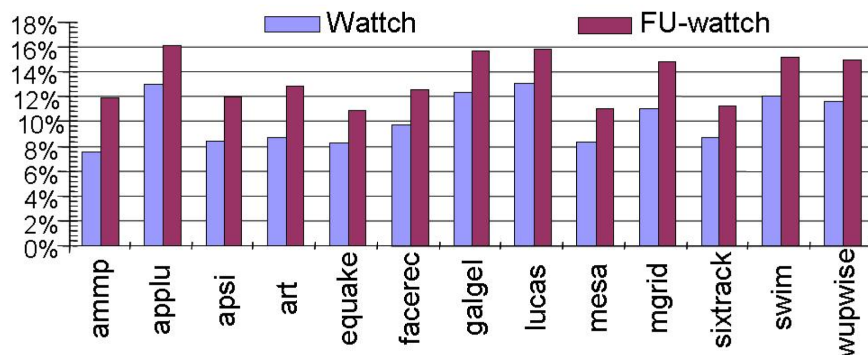
Tipo de UF	**Consumo por unidad** (w)	**UFs disponibles** ***FU-Wattch***	***Wattch***
UFs de Enteros			
Sumador(64-bits)	1.16503	4	4
Unidad Lógica	0.104394	4	0
Unidad de Desplaz	0.505265	4	0
Multiplicador(16-bits)	0.897785	1	0
UFs de PF			
Sumador	3.57026	1	1
Multiplicador	3.57026	1	0

En la figura 4.4 se puede ver el porcentaje que representa el consumo de las UFs respecto al consumo total del procesador, estimado con *Wattch* y con *FU-Wattch*, para todos los *Benchmarks*. De ellas, la figura 4.4(a) muestra los resultados para todos los *Benchmarks* de enteros, mientras que la figura 4.4(b) muestra los resultados para los *Benchmarks* de punto flotante. En estas gráficas se puede apreciar que hay una gran diferencia entre los valores obtenidos con cada uno de los simuladores. Con *Wattch* el porcentaje medio es un 8 % para los *Benchmarks* de enteros y un 10 % para los de punto flotante. Mientras que

con *FU-Wattch* estos porcentajes son el 11 % y 13,5 % respectivamente.



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 4.4: Porcentaje del consumo de la unidad de ejecución respecto al consumo total del procesador obtenido con *Wattch* y con *FU-Wattch*. La figura (a) muestra los resultados para INTSPEC 2000, la figura (b) muestra los resultados para FPSPEC 2000.

4.4. Conclusiones

En este capítulo hemos presentado una versión modificada de la herramienta de simulación de potencia *Wattch*. El objetivo de la modificación ha sido conseguir un modelo más preciso para estimar los consumos de potencia tanto estáticos como dinámicos en los *clusters* de las unidades funcionales de enteros y de punto flotante.

Los resultados experimentales demuestran que los resultados obtenidos con el simulador *FU-Wattch* son más reales, en términos de porcentajes de potencia total consumida por las unidades funcionales en el procesador, que los que se obtienen si se utiliza la configuración inicial de *Wattch*.

La motivación para mejorar este simulador ha sido obtener una herramienta más realista para examinar nuevas arquitecturas y organizaciones de las unidades funcionales como las que se propondrán en el capítulo 5.

Capítulo 5

Reducción del consumo en los procesadores superescalares mediante cotejo de códigos de operación

Como ya se dijo en el capítulo 1, la unidad de ejecución es una de las partes de los procesadores superescalares que más consumen. Por otro lado, en el capítulo 2, se ha hecho un repaso de las diferentes técnicas que existen para reducir el consumo en las UFs en dichos procesadores. En él se pudo ver que la mayoría de las técnicas están basadas en la identificación de *valores narrows* [HRBM03] [BM00] [STT01] [CJC00] [GM05] [HRBM05] [HBS+04] [HLR05] [TSC07]. En este capítulo presentamos una técnica que también usa la detección de dichos valores para reducir el consumo en las UFs, y que aporta algunas ventajas frente a las anteriores. A continuación destacaremos algunas

de las más relevantes:

- Las técnicas que explotan los valores *narrows* tienen que chequear los propios operandos para detectar si sus valores lo son o no. Esto implica añadir un complicado *Hardware* para la detección de dichos valores y, por lo tanto, un consumo extra y una penalización en el rendimiento mientras se hacen estas detecciones. Nosotros no necesitamos chequear completamente los operandos; simplemente analizando los códigos de operación, y en algún caso un número reducido de bits, podemos conocer si una instrucción implica una operación con valores *narrow*.

- La mayoría de las técnicas descritas sólo reducen el consumo dinámico mientras que esta técnica reduce el consumo tanto dinámico como estático porque reemplaza sumadores de alto consumo por otros de menor consumo.

- Esta técnica introduce cambios en el número y las características de los sumadores, sin introducir prácticamente penalización en el rendimiento del procesador.

El resto del capítulo está organizado de la siguiente forma: la sección 5.1 presenta un resumen del repertorio de instrucciones de la arquitectura *Alpha* 21264, para luego centrarse en las instrucciones que usan un sumador en algún momento de su ejecución y proporcionar un estudio que demuestra la existencia, de una gran cantidad de ellas que no requieren un sumador de 64-bits. La sección 5.2 presenta todos los detalles de la técnica propuesta en este trabajo para reducir el consumo en las UFs. La sección 5.3 describe el entorno de simulación usado. La sección 5.4 muestra los resultados obtenidos al aplicar dicha

técnica a un procesador de alto rendimiento. Y por último, en la sección 5.5 se expondrán algunas conclusiones.

5.1. Estudio de la instrucciones del repertorio *Alpha*

Esta sección proporciona un estudio donde se muestra que en los procesadores de 64 bits existe una gran cantidad de sumas que no requieren un sumador 64-bits. Para este estudio nos hemos centrado en el conjunto de instrucciones de la arquitectura *Alpha* [ACC].

5.1.1. Instrucciones del repertorio *Alpha*

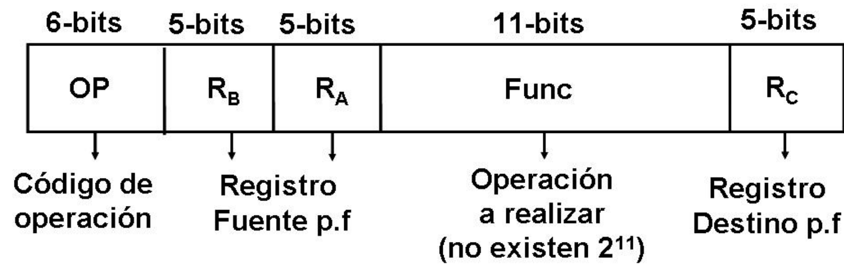
La arquitectura *Alpha* se caracteriza por seguir la filosofía RISC (Conjunto reducido de instrucciones). La mayoría de sus instrucciones operan sobre los registros, accediendo a la memoria sólo para instrucciones de carga y almacenamiento. La razón de esto es que se intenta minimizar los accesos a memoria, puesto que suponen el principal cuello de botella en los procesadores actuales.

Esta arquitectura contiene 32 registros de uso general para enteros y otros 32 registros, también de uso general, para punto flotante. La longitud de palabra de los registros es de 64 bits pero admite operaciones con diferentes tamaños. A continuación se presentan los tipos de datos que soporta la arquitectura *Alpha* 21264:

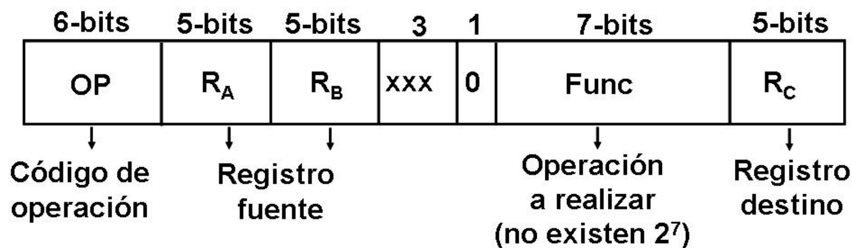
- Enteros:
  - Byte
  - Palabra (Word) 2 Bytes
  - Doble palabra (Longword) 4 Bytes
  - Cuádruple palabra (Quadword) 8 Bytes
- Punto Flotante:
  - IEEE Simple(32 bits),
  - IEEE Doble(64)
  - IEEE Extendido(128)
  - Formatos VAX: F-floating(32), G-Floating(64)

Su repertorio de instrucciones está constituido por cuatro tipos: de memoria, de salto, de operación y especificas. A continuación se hace un breve resumen de cada uno de los tipos:

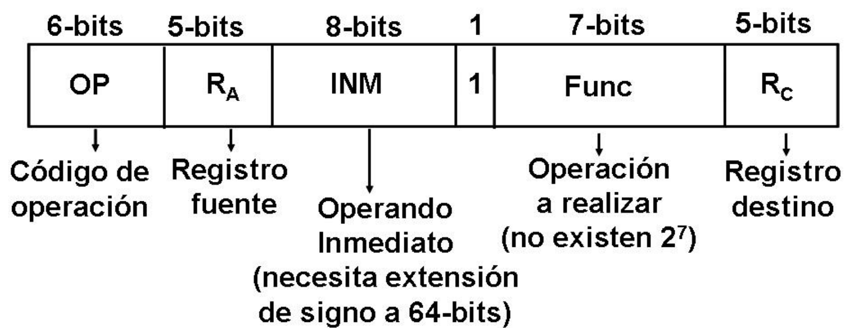
- **Instrucciones de operación:** pueden operar con enteros o con punto flotante. Las operaciones con punto flotante se realizan entre registros. Las operaciones con enteros presentan dos opciones, o se realizan entre registros o, entre registro y un inmediato de tamaño byte que está contenido en la propia instrucción. En la figura 5.1 se muestra el formato de las instrucciones de operación para cada uno de los casos.
- **Instrucciones de memoria:** trabajan entre registro y memoria. Son de dos tipos: *Load* que llevan datos de memoria a registro y *Store* que lo



(a) Operación entre Registros de Punto Flotante



(b) Operación entre Registros de Enteros



(c) Operación entre Registro de Enteros e Inmediato

FIGURA 5.1: Formato de las instrucciones de operación del repertorio de la arquitectura *Alpha* 21264. (a) muestra el caso de operación con punto flotante, (b) muestra el caso de operación con enteros entre registros y (c) muestra el caso de operación con enteros entre un registro y un inmediato.

hacen de registro a memoria. Ambas pueden mover *bytes*, *words*, *longwords* y *quadwords*. Para calcular la dirección efectiva, usan un registro indice y un desplazamiento que está contenido en la propia instrucción. La figura 5.2 muestra el formato de este tipo de instrucciones.

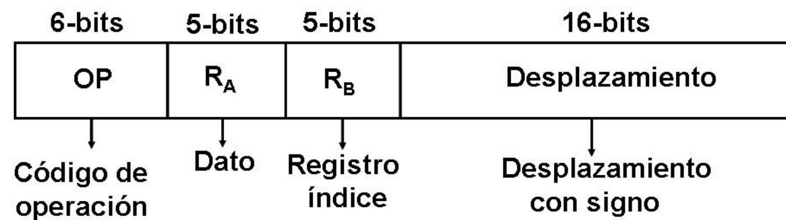


FIGURA 5.2: Formato para las instrucciones de memoria del repertorio de la arquitectura *Alpha* 21264.

- **Instrucciones de salto:** son de dos tipos, saltos condicionales y saltos incondicionales. Ambas tienen que calcular la dirección efectiva del salto, para ello usan el contador de programa (PC) y un desplazamiento que está contenido en la propia instrucción. Además, las de salto condicional tienen que chequear el operando contenido en el registro especificado en el campo de la instrucción llamado RA, para evaluar la condición. La figura 5.3 muestra el formato de las instrucciones de este tipo.

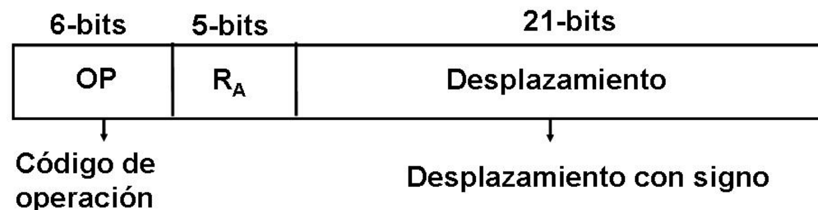


FIGURA 5.3: Formato para las instrucciones de salto del repertorio de la arquitectura *Alpha* 21264.

- **PALcode:** (biblioteca de arquitectura privilegiada). Subrutinas relacionadas con el sistema operativo: cambios de contexto, interrupciones, ges-

ción de memoria. La figura 5.4 muestra el formato de las instrucciones de este tipo.

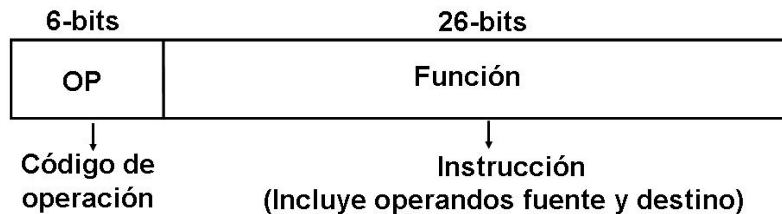


FIGURA 5.4: Formato para las instrucciones especiales (PALcode) del repertorio de la arquitectura *Alpha* 21264.

Para el trabajo que se presenta en este capítulo, sólo nos interesan las instrucciones que usan un sumador en algún momento de su ejecución, por lo que en las siguientes secciones vamos a mostrar un estudio detallado de dichas instrucciones y sacar conclusiones que nos servirán de base para desarrollar las técnicas que se proponen en esta tesis.

5.1.2. Instrucciones del repertorio *Alpha* que usan un sumador

Como hemos indicado las técnicas propuestas en este capítulo se centran en la sustitución de uno o varios de los sumadores de 64 bits incluidos en la unidad de ejecución por otros de menor tamaño. Para saber en qué condiciones podemos realizar estas modificaciones debemos primero asegurar que las instrucciones se ejecutan correctamente, es decir que no se produce ningún resultado erróneo tras la ejecución de las mismas. Como paso previo debemos por lo tanto analizar que sucede con las instrucciones que utilizan estos sumadores susceptibles de ser sustituidos y en qué casos debemos tener cuidado

con los operandos implicados en la ejecución. El repertorio de instrucciones de la arquitectura *Alpha* incluye siete instrucciones que utilizan sumadores de enteros. Dado que la UF de enteros original ofrece cuatro sumadores de 64 bits, todas estas instrucciones se ejecutan siempre sobre 64 bits. Ahora bien, debemos preguntarnos ¿Si dispusiéramos de sumadores de menor tamaño (en número de bits), podrían ejecutarse correctamente las instrucciones?, la respuesta como veremos a continuación es afirmativa. Las instrucciones que usan sumador en algún momento de su ejecución son las siguientes:

- BRA: instrucción de salto condicional. Tiene que sumar el contador de programa (PC) de 64-bits con el desplazamiento de 21-bits, que viene en un campo de la instrucción, para obtener la dirección efectiva a la cual se quiere saltar. Como se refleja en la figura 5.5 al desplazamiento se le aplica una extensión de signo (de 21 bits hasta 64 bits) para después sumarlo con el contenido del PC.

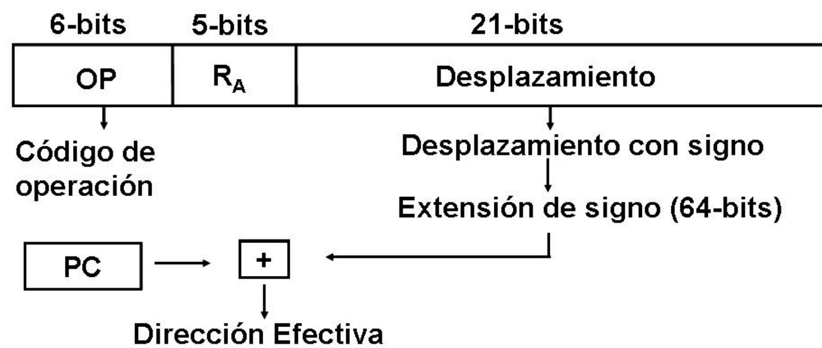


FIGURA 5.5: Formato para las instrucciones de salto del repertorio de la arquitectura *Alpha* 21264, donde se representa la operación que se realiza para calcular la dirección efectiva.

- JMP: instrucción de salto incondicional. Tiene que sumar el PC de 64-bits con el desplazamiento de 13-bits, que viene en un campo de la ins-

trucción, para obtener la dirección efectiva a la cual se quiere saltar. Este caso es el mismo que el mostrado en la figura 5.5, donde ahora el desplazamiento es de 13-bits.

- L/S y LDA: instrucciones de memoria. Tienen que sumar el operando Rb de 64-bits con el desplazamiento de 16-bits, que viene en un campo de la instrucción , para obtener la dirección efectiva de memoria donde se quiere almacenar (o de donde se quiere tomar) el dato. Como se refleja en la figura 5.6 al desplazamiento se le aplica una extensión de signo (de 16 hasta 64-bits) para después sumarlo con el contenido de la dirección de memoria indicada por el registro RA

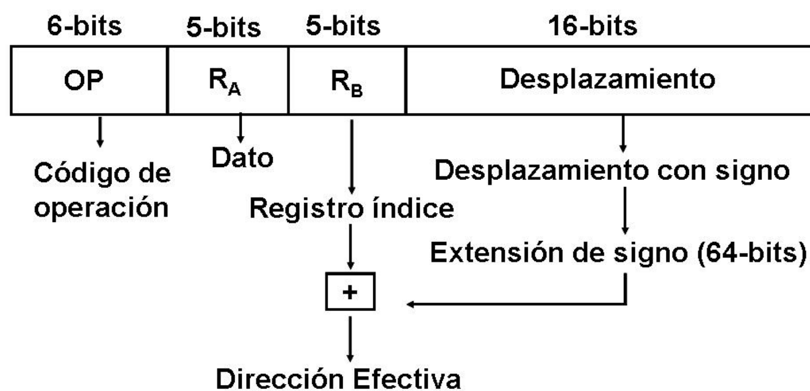


FIGURA 5.6: Formato para las instrucciones de memoria del repertorio de la arquitectura *Alpha* 21264, donde se representa la operación que se realiza para calcular la dirección efectiva.

- ARIT IMM: instrucción de operación aritmética. Tiene que sumar el operando Ra de 64-bits con un valor Inmediato de 8-bits, que viene en un campo de la instrucción. La figura 5.1(c) muestra el formato.

- ARIT LONG: instrucción de operación aritmética. Tiene que sumar dos operandos (Ra y Rb) de 32-bits cada uno. Las figuras 5.1(b) y 5.1(a)

muestran el formato.

- ARIT: instrucción de operación aritmética. Tiene que sumar dos operandos (Ra y Rb) de 64-bits cada uno. Las figuras 5.1(b) y 5.1(a) muestran el formato.

Las instrucciones que requieren utilizar el sumador para realizar su fase de ejecución están recogidas en la tabla 5.1. La primera columna nos dice el nombre de la instrucción, la segunda columna nos indica el tipo de instrucción, y la tercera columna nos muestra la operación que se realiza en la unidad de ejecución para cada una de las instrucción cuando se están ejecutando. La nomenclatura que se ha usado es la siguiente: *PC* es el registro Contador de Programa de longitud 64 bits; *Ri* representa un registro fuente que se encuentra en el banco de registros y tienen una longitud de 64 bits; *ExtSign* indica que se hace la extensión de signo del valor que tiene entre paréntesis; *Desplaz(xbits)* indica el tamaño del deplazamiento que viene en la instrucción y por último *Inmediato* indica el tamaño del valor inmediato que viene en la instrucción.

Para entender cómo se lee esta tabla tomemos la instrucción BRA. Esta es una instrucción de salto que, como ya hemos dicho, necesita calcular la dirección efectiva donde va a saltar. Esta se calcula sumando al contador de programa un desplazamiento de 21 bits que viene dado en la propia instrucción. Para ello, primero se extiende el signo del desplazamiento por tener este memos bits que el contador de programa. Esta operación la realiza un sumador de la unidad de ejecución y viene representada en la tercera columna por la siguiente espresión: EA=PC + ExtSig(Desplaza(21 bits))

TABLA 5.1: Instrucciones del procesador *Alpha* que necesitan sumador.

Tipo de Instrucción	**Formato de la instrucción**	**Operación realizada en sumador**
BRA	De salto	PC + ExtSig(Desplaz(21bits))
JMP	De salto	PC + ExtSig(Desplaz(13bits))
L/S	De memoria	Rb + ExtSig(Desplaz(16bits))
LDA	De memoria	Rb + ExtSig(Desplaz(16bits))
ARIT IMM	De operación	Ra + Inmediato(8bits))
ARIT LONG	De operación	Ra(32bits) + Rb(32bits))
ARIT	De operación	(Ra + Rb)

Analizando simplemente los operando que interviene en estas instrucciones podemos sacar varias conclusiones:

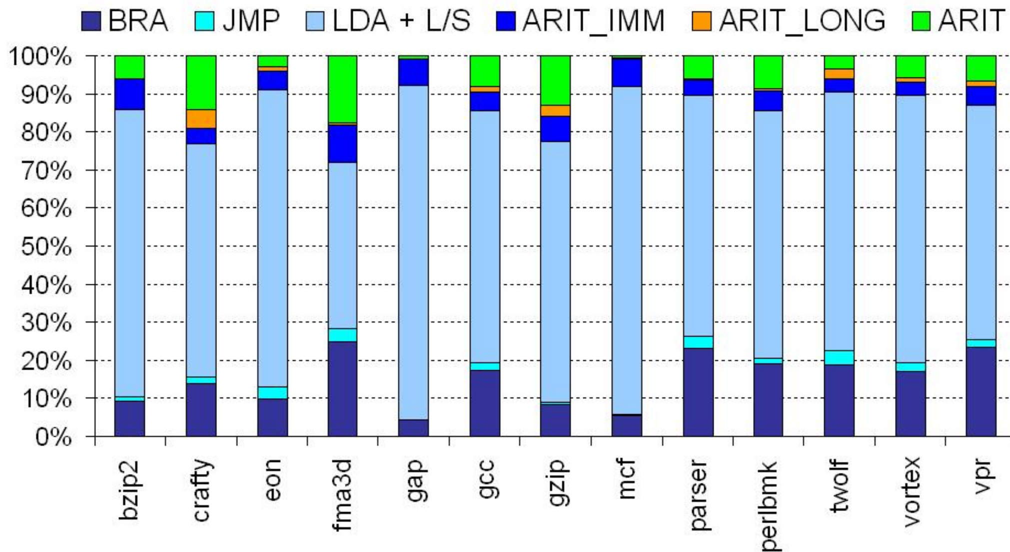
1. Las instrucciones ARIT LONG pueden dar un resultado correcto con un sumador de 32-bits en todos los casos ya que trabajan con tamaño de operando de 32-bits.

2. Las instrucciones del tipo BRA, JMP, L/S, LDA, ARIT IMM pueden ejecutarse en la mayoría de los casos en un sumador de 24-bits sin que se pierda ningún bit significativo. Si observamos la tabla 5.1 podemos pensar, en una primera aproximación, que requieren un sumador de 64 bits porque tanto el contador de programa (PC) como los registros fuentes tienen ese número de bits. Sin embargo, en todas ellas el segundo operando fuente es un valor de los llamados *narrow* (21 bits para las instrucciones BRA, 13 bits para las instrucciones JMP, etc..) por lo que es poco probable que necesiten el sumador de 64-bits.

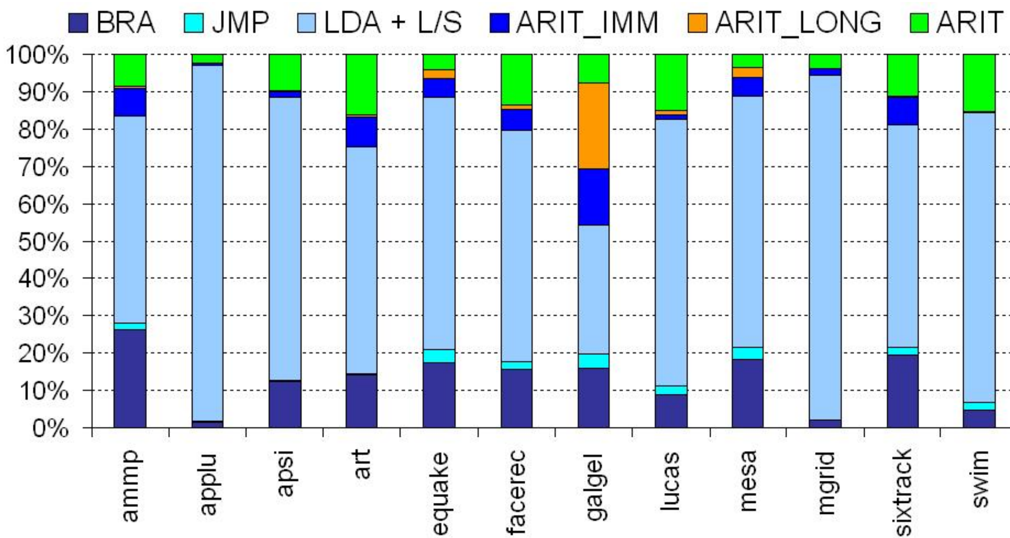
3. Las instrucciones ARIT necesitan siempre un sumador de 64-bits ya que trabajan con tamaño de operando de 64-bits.

Con el fin de validar la idea que se acaba de exponer, se han ejecutado en el simulador *SimpleScalar* [ALE02], un subconjunto de *Benchmarks* del conjunto SPEC CPU2000 (ver sección 5.3.4). El resultado de estas simulaciones se puede ver en la figura 5.7. En ella se muestra el porcentaje de instrucciones que hay de cada uno de los tipos de instrucción representados en la tabla 5.1 (instrucciones que necesitan un sumador para su ejecución). La figura 5.7(a) muestra los resultados para los *Benchmarks* de enteros y la 5.7(b) lo hace para los de punto flotante. Observando estos resultados podemos ver que las instrucciones ARIT, que son las que deberían ejecutarse en el sumador de 64-bits, representan un porcentaje muy pequeño (7.9 % de media) del total de instrucciones que requieren sumador. Lo mismo pasa con las instrucciones tipo ARIT_LONG, que son las que se pueden ejecutar en un sumador de 32-bits, donde el promedio es incluso menor (4,3 %). Además, en este caso existen algunos *Benchmarks*, como por ejemplo bzip2 y gap, en los que no existen este tipo de instrucciones. El resto de instrucciones, que son las que se podrían ejecutar en un sumador de 24-bits, representan el 87,8 %. Como conclusión podemos decir que, en media, el 65.4 % de las instrucciones de los *Benchmarks* necesitan un sumador para ejecutarse y de este 65.4 % el 92.1 % se podrían ejecutar en un sumador de 24/32-bits (87.8 % y 4.3 % respectivamente).

Ya hemos visto que todas estas instrucciones realizan una suma en algún momento de su ejecución y hemos sacado algunas conclusiones sobre ello. En la segunda de las conclusiones que hemos comentado anteriormente, hemos indicado que era poco probable que las instrucciones del tipo BRA, JMP, L/S, LDA, ARIT IMM necesiten un sumador de 64-bits. Ahora bien, analicemos en que casos particulares es indispensable que este tipo de instrucciones se



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 5.7: Porcentaje de instrucciones de cada tipo que hay en cada *Benchmarks*. (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

ejecuten en un sumador de 64-bits. Las instrucciones mencionadas no podrán utilizar un sumador de menor tamaño si se cumplen las siguientes condiciones simultaneamente:

1. Que exista acarreo al sumar el bit más significativo del operando *narrow* (desplazamiento o dato inmediato) con el correspondiente bit del otro operando fuente (PC, Ra o Rb).

   Por ejemplo, si nos fijamos en las instrucciones de salto tipo BRA de la tabla 5.1 podemos observar que para calcular la dirección efectiva se tiene que sumar el PC de 64-bits con el desplazamiento de 21-bits. Esta suma puede producir un acarreo en la posición 20 como se refleja en la figura 5.8.

2. Y que dicho acarreo se arrastre hasta el bit 24 o 32, (dependiendo del sumador que queramos usar).

   Siguiendo con el ejemplo anterior, podemos ver que cuando se produce acarreo en la posición 20 puede ocurrir que este desaparezca al seguir sumando el resto de posiciones, o por el contrario, puede que dicho acarreo se propage hasta la posición 24, lo cuál no nos permitirá usar un sumador de 24-bits, o hasta la posición 32, lo cuál tampoco nos permitirá usar un sumador de 32-bits. Esta situación ocurrirá siempre si, (1) el operando *narrow* es negativo, o si, (2) el operando *narrow* es positivo y los bits más significativos de la primera palabra del PC (o registros, dependiendo del codigo de operación) son 1s. El número de bit significativos depende del tipo de instrucción, por ejemplo, en el caso de las instrucciones BRA,

estos bits serán los situados entre el bit 21 y el 31, ambos inclusive. Estas dos situaciones se pueden ver en la figura 5.8.

La figura 5.8 muestran cuando las instrucciones BRA, L/S, LDA, JMP y ARIT_IMM necesitan un sumador de 64-bits, es decir, cuando sus operandos se encuentran en la situación que se acaba de describir. En particular, la figura 5.8(a) describe la situación en caso de desplazamiento o inmediato positivos y la figura 5.8(b) lo hace para los negativos.

Con objeto de comprobar el número de casos en que se producen estas suposiciones, se han realizado nuevas simulaciones detectando los casos descritos en la figuras 5.8(a) y 5.8(b). Dado que detectar todos los casos implica hacer de antemano una suma para ver si se produce acarreo, como indica la primera condición, se ha supuesto que dicho acarreo siempre existe. Esto nos hace contemplar más casos de los que en realidad serían, pero a cambio simplificamos mucho la lógica de detección de estos casos, y por lo tanto la lógica de selección del sumador, ya que solamente hay que chequear el bit más significativo del operando *narrow* (desplazamiento o dato inmediato) y sólo en caso de ser este positivo, los bits más significativos de la primera palabra del PC (o registros, dependiendo del codigo de operación) para ver si son 1.

Para que se entienda mejor como detectamos estos casos veamos un ejemplo con una instrucción tipo BRA. Como se ha comentado en el parrafo anterior, para evitar tener que hacer una suma de antemano, suponemos que se produce acarreo en el bit 20 y de esta manera nos aseguramos que se cumple la primera condición. Para ver si se cumplen la segunda condición tenemos que chequear el bit de signo del operando *narrow*, bit 20 del despalzamiento. Si este bit es

Tipo de Instruc	Caso a estudiar
BRA	63 32 31 21 20 0 X ....X | 11.........1 | X X ....X PC C 0 X.... ....X Desplazamiento
L/S y LDA	63 32 31 16 15 0 X ....X | 11.............1 | X X...........X Rb C 0 X...........X Desplazamiento
JMP	63 32 31 13 12 0 X ....X | 11...............1 | X X.........X PC C 0 X.........X Desplazamiento
ARIT_IMM	63 32 31 8 7 0 X ....X | 11..................1 | X X.....X Ra C 0 X...X Imm

(a)

Tipo de Instruc	Caso a estudiar
BRA	63 32 31 21 20 0 X ....X | X.........X | X X ....X PC C 1 X.... ....X Desplazamiento
L/S y LDA	63 32 31 16 15 0 X ....X | X.............X | X X.........X Rb C 1 X..........X Desplazamiento
JMP	63 32 31 13 12 0 X ....X | X...............X | X X.........X PC C 1 X......... X Desplazamiento
ARIT_IMM	63 32 31 8 7 0 X ....X | X..................X | X X....X Ra C 1 X... X Imm

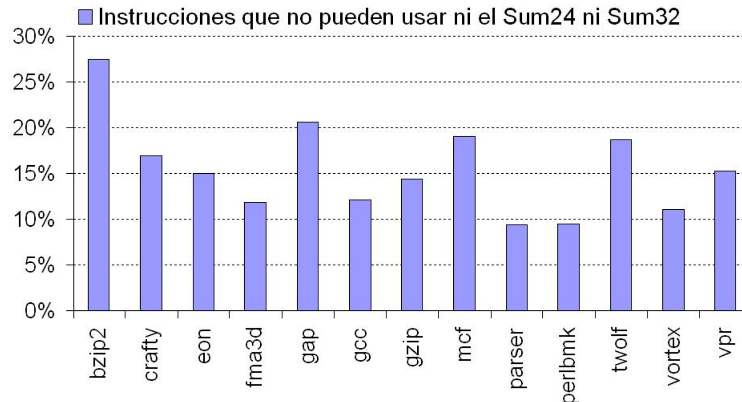
(b)

FIGURA 5.8: Casos en los que las instrucciones de la primera columna de la tabla 5.1 tienen que usar el sumador de 64 bits. (a) muestra los casos cuando el operando *narrow* es positivo, (b) muestra los casos cuando el operando *narrow* es negativo.

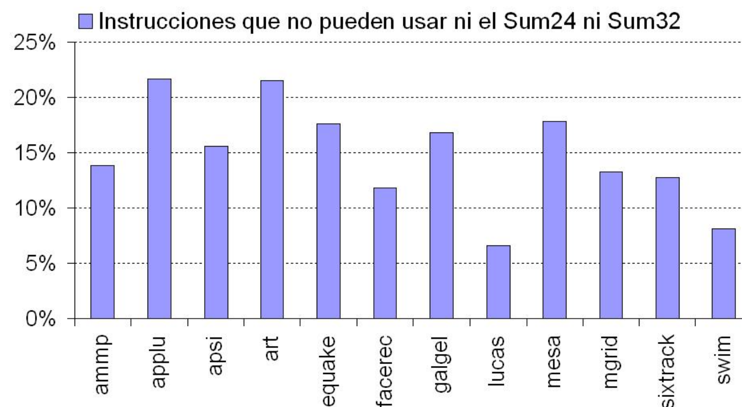
1, lo que implica que el operando es negativo, ya no hay que mirar más bits y podemos decir que estamos ante uno de los casos en los que la instrucción sólo se puede ejecutar en un sumador de 64-bits. Si por el contrario, dicho bit es 0, lo que implica que el desplazamiento es positivo, entonces hay que comprobar si los bit 20, 21 y 22 del PC son 1. Si lo son, la instrucción no se puede lanzar a un sumador de 24-bits, y entonces hay que comprobar si los bits del 23 al 31 del PC también son 1s. En este caso, a la instrucción tampoco se le puede asignar un sumador de 32-bits y la única posibilidad es que se ejecute en un sumador de 64-bits. Como se pudede ver, en el mejor de los casos solamente hay que mirar el bit de signo del desplazamiento y en el peor de los casos sólo hay que chequear 10 bits del PC.

La figura 5.9 muestra el porcentaje de instrucciones, respecto de las instrucciones que se podrían ejecutar en un sumador de 24/32-bits, que se encuentran en la situación descrita en la figuras 5.8(a) y 5.8(b). Observando estos resultados se puede ver que sólo hay cuatro *Benchmarks* que tienen más del 20 % de las instrucciones en estos casos. Podemos decir que del 92.1 % de las instrucciones que se podrían ejecutar en un sumador de 24/32-bits (ver figura 5.7) sólo el 14.7 %, en promedio, se encuentran en esta situación.

Teniendo en cuenta estos resultados la figura 5.10 muestra, que porcentaje de instrucciones, del total de instrucciones que necesitan un sumador para ejecutarse, requieren un sumador de 64-bits y que porcentaje de instrucciones podrían usar un sumador de 24/32 bits. La figura 5.10(a) muestra los resultados para los *Benchmarks* de enteros y la 5.10(b) lo hace para los de punto flotante. Observando dicha gráfica podemos ver que, en todos los *Benchmarks*, siguen siendo una gran mayoría las instrucciones que se podrían ejecutar en



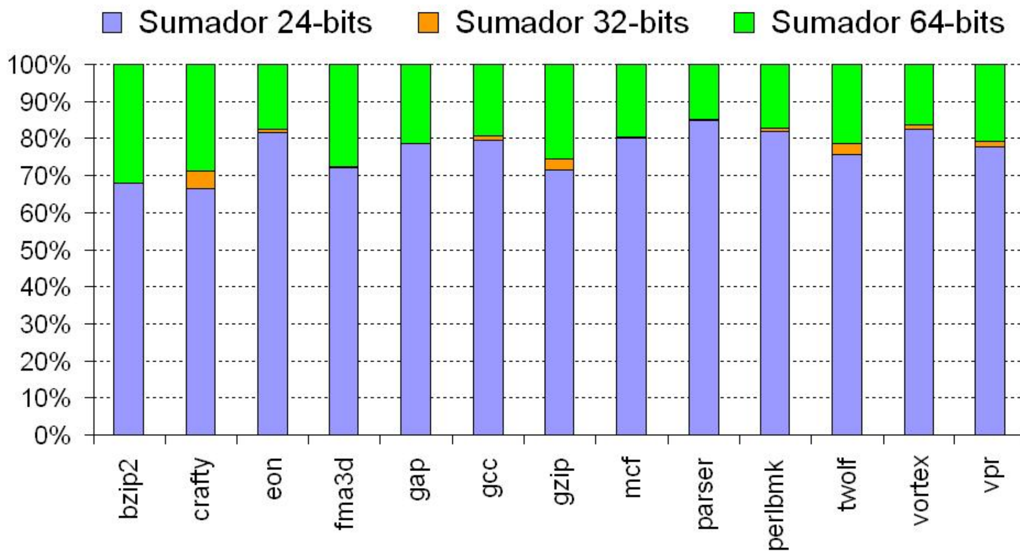
(a) INTSPEC 2000



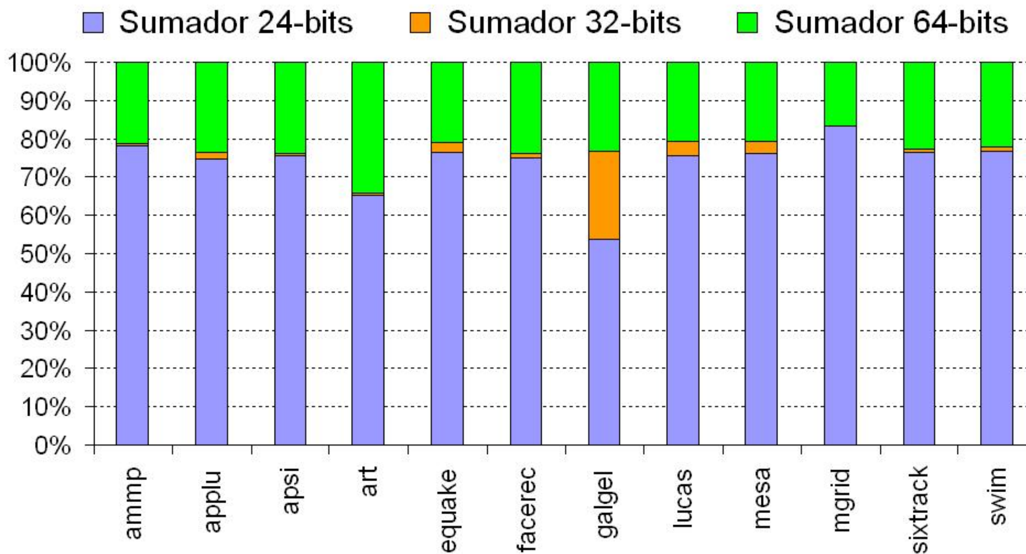
(b) FPSPEC 2000

FIGURA 5.9: Porcentaje de instrucciones que se encuentran en los casos descritos en la figura 5.8 y por lo tanto no pueden ejecutarse ni en el sumador de 24-bits (Sum24) ni en el de 32-bits (Sum32). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

sumadores de 24-bits (el 73.7 % en promedio) y algunas instrucciones se podrían lanzar al de 32-bits (4.7 % en promedio). Existen *Benchmarks*, como gap, bzip2, mgrid etc, que no tiene instrucciones para ejecutar en un sumador de 32-bits. También podemos observar que en todos los *Benchmarks* existen instrucciones, aunque son un porcentaje pequeño (21.6 % en promedio), que necesitan un sumador de 64-bits. Estos valores están medidos respecto al número de instrucciones que necesitan un sumador para ejecutarse.



(a) INTSPEC 2000



(b) FPSPEC 2000

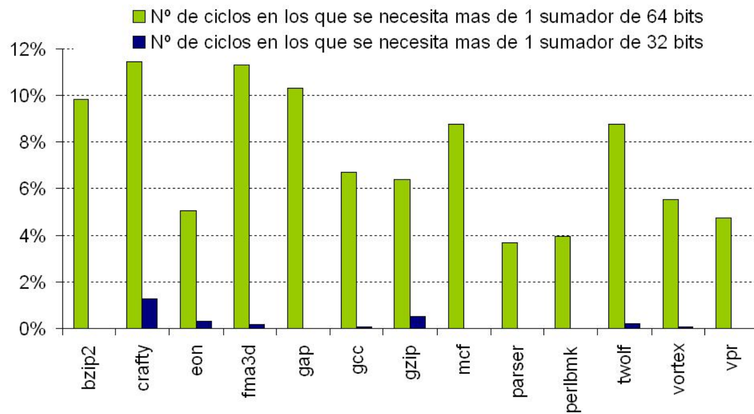
FIGURA 5.10: Porcentaje de instrucciones que podrían ejecutarse en cada sumador. (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

5.1.3. Número de ciclos en los que se requiere más de un sumador de 64/32-bits

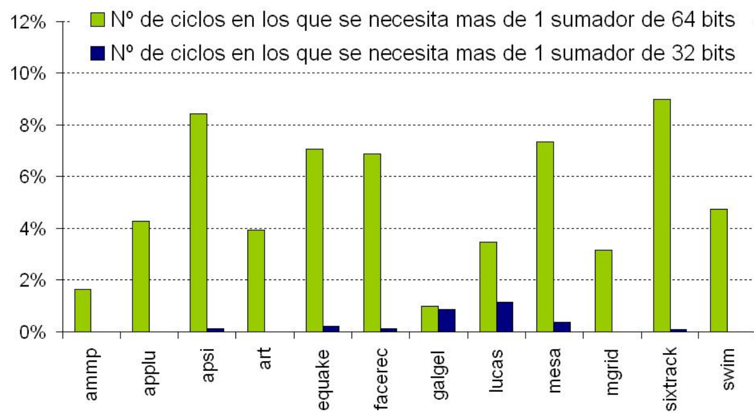
Otro aspecto importante a resaltar en este estudio es saber en cuantos ciclos se va a necesitar más de un sumador de 64-bits, ya que con ello podremos intuir como afecta al rendimiento del procesador el tener menos sumadores de 64-bits de los que actualmente existen en las arquitecturas de los procesadores de alto rendimiento. Dicho de otro modo, supongamos que se dispone de dos sumadores de 64-bits en vez de los cuatro de los que dispone la arquitectura *Alpha*, y el procesador lanza cuatro instrucciones por ciclo. Puede ocurrir que tres de ellas, (o las cuatro) necesiten ejecutarse en un sumador de 64-bits. Al haber solamente disponibles dos de estos sumadores, una (o dos) de las instrucciones tienen que esperar a que queden esos sumadores disponibles, repercutiendo este hecho en el rendimiento del procesador.

La figura 5.11 muestra el porcentaje de ciclos, respecto a los ciclos totales, en los que se necesitan más de un sumador de 64-bits. La figura 5.11(a) muestra los resultados para los *Benchmarks* de enteros y la 5.11(b) lo hace para los *Benchmarks* de punto flotante. En ellas se puede observar que el porcentaje de estos ciclos se encuentra entre el 1 % (para el *Benchmark* gagel) y el 11.3 % (para el *Benchmark* fma3d). De estos resultados podemos deducir que se podrían sustituir algunos de los sumadores de 64-bits por unos de 24 o 32-bits esperando no afectar demasiado al rendimiento.

Por otro lado, como no sólo se van a sustituir sumadores de 64-bits por sumadores de 32-bits, sino que también se pretende poner algunos de 24-bits, resulta interesante saber como afecta al rendimiento usar el mínimo número de



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 5.11: Ciclos en los que se necesita más de un sumador de 64-bits y más de una sumador de 32-bits. (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

sumadores de 32-bits. Por ello, en la figura 5.11 también están representados datos para el sumador de 32-bits. Si observamos estos resultados se puede ver que el porcentaje de ciclos en los que se necesita más de un sumador de 32-bits no supera el 2 %, existiendo muchos *Benchmarks* en los que el porcentaje es 0 %. Esto nos conduce a pensar que no son necesarios muchos sumadores de 32-bits. Todo esto concuerda con lo que ya se había observado en los resultados de las secciones anteriores donde se tenía que el porcentaje de instrucciones

que podrían usar el sumador de 32 bits es muy pequeño (el 4.7 %), existiendo *Benchmarks* en los que en los que el porcentaje de estas instrucciones es 0 %.

5.1.4. Conclusiones

De todo este estudio se puede concluir que:

1. De todas las instrucciones que requieren un sumador para ejecutarse, hay un porcentaje muy alto que se podrían ejecutar en un sumador de 32/24 bits (en promedio el 78.4 %):

    - El 73.7 % de las instrucciones pueden usar uno de 24-bits.
    - El 4.7 % de las instrucciones pueden usar uno de 32-bits.
    - El 21.6 % de las instrucciones pueden usar uno de 64-bits.

2. Se podría sustituir la mayoría de los sumadores de 64-bits por otros de 24/32-bits esperando no afectar al rendimiento.

    - Sólo en el 6.1 %, en promedio, de los ciclos se necesita más de un sumador de 64-bits.

3. Lo mismo se puede decir de los sumadores de 32-bits.

    - Las instrucciones ARITM_LONG sólo representan el 4 % de las instrucciones que necesitan un sumador, existiendo *Benchmarks* en los que no existen instrucciones que necesitan un sumador de 32-bits.

- Sólo el 0.4 % de las instrucciones que no pueden ser ejecutadas en un sumador de 24-bits, pueden usar uno de 32-bits.
- Sólo en el 1.5 %, en promedio, de los ciclos se necesita más de un sumador de 32-bits.

5.2. Propuestas para reducir el consumo en las UFs

En esta sección se presenta una técnica *Hardware*, que explota los resultados de la sección anterior, para reducir el consumo en las UFs de los procesadores superescalares actuales. La técnica consiste en sustituir algunos de los sumadores de enteros de 64-bits de los procesadores modernos, que están diseñados para ser rápidos lo que les hace tener un alto consumo de potencia, por sumadores de enteros de 32/24-bits, que no necesitan ser tan rápidos (ya que el máximo retardo lo marca el sumador de 64-bits) y por lo tanto su consumo tanto estático como dinámico es bastante menor. Esta técnica explota el hecho de que el 78,4 % de las instrucciones que requieren sumadores para su ejecución pueden usar un sumador de 32/24-bits en vez de uno de 64-bits, como se ha demostrado en la sección anterior. Con ella se consigue reducir tanto el consumo estático como el dinámico en las unidades funcionales sin afectar prácticamente al rendimiento.

El uso de tres tipos de sumadores implica una significativa complejidad en el *Hardware* que asigna UFs a las instrucciones, con el consecuente aumento de consumo. Para hacer la técnica más eficiente se ha diseñado el siguiente

protocolo para la asignación de UFs:

1. Es necesario saber si la instrucción requiere usar un sumador y de que tamaño lo necesita. Para ello existe un árbitro que chequea los bits de código de operación de la instrucción y en función de ellos, asigna un sumador a dicha instrucción. Este árbitro está situado en la cola de lanzamiento. El algoritmo que selecciona el tamaño del sumador se ha implementado para consumir lo mínimo e introducir el menor retardo. Este algoritmo funciona de la siguiente manera:

    - Si la instrucción es ARITH (tabla 5.1) le asigna un sumador de 64-bits.
    - Si la instrucción es ARITH_LONG le asigna un sumador de 32-bits. En estos dos casos basta con mirar en el código de operación, no hace falta chequear los operandos.
    - Para el resto de instrucciones se estudia la existencia de las situaciones que se muestran en las figuras 5.8(a) y 5.8(b). En caso de detectarse una de estas situaciones le asigna un sumador de 64-bits, en caso contrario uno de 32/24-bit. Para detectar dichas situaciones se ha supuesto que siempre existe acarreo al sumar el bit más significativo del operando *narrow* (desplazamiento o dato inmediato) con el correspondiente bit del otro operando fuente (PC, Ra , Rb), de manera que sólo hay que chequear el bit más significativo del operando *narrow* y, solamente en caso de que este sea positivo, se chequearán los bits necesarios del PC (o registros, dependiendo del codigo de operación) para ver si son 1s. El número de bits que se

deben chequear depende del tipo de instrucción. Por ejemplo, en el caso de las instrucciones L/S, estos bits serán los situados entre el bit 16 y el 23, ambos inclusive, para ver si puede lanzarse al sumador de 24-bits, y en caso de ser todo 1s se chequearan desde el bit 24 hasta el bit 31 para comprobar si puede usar el sumador de 32-bits (ver las figuras 5.8(a) y 5.8(b) para más detalles). La suposición de que siempre exista ese acarreo sobreestima la presencia de estos casos, pero aun así las probabilidades de que sucedan siguen siendo muy pequeñas y a cambio se hace más sencilla la detección de estos casos, simplificando la implementación y reduciendo el tiempo de ejecución.

2. Una vez se sabe si la instrucción necesita sumador y el tipo de sumador, hay que buscar un sumador disponible. Algoritmo 1 muestra el método seguido para hacer esta búsqueda. En él podemos ver que si una operación puede realizarse en un sumador de 24-bits siempre se va a mirar primero si hay algún sumador de este tipo disponible; si lo hay la instrucción se lanza a dicho sumador. Caso de estar todos ocupados se mira si hay algún sumador de 32-bits disponible, si es así se lanza a ese sumador, si no entonces se lanza al de 64-bits.

   Con esta política no reducimos al máximo el ahorro de consumo ya que algunas instrucciones que podrían ser ejecutadas en un sumador de menor consumo se ejecutan en uno de consumo mayor. A cambio conseguimos minimizar el retardo introducido por la lógica de selección.

Algorithm 1 Protocolo de asignación de sumadores

```
if (tipo instrucción==BRA or JMP or L/S or LDA or ARIT-INM) then
  if (Sumadores-24bits==Disponibles) and (UsarSumador-24bits==Si) then
    la instrucción se lanza a un sumador de 24-bits;
  else if (Sumadores-32bits==Disponibles) and (UsarSumador-32bits==Si) then
    la instrucción se lanza a un sumador de 32-bits;
  else if (Sumadores-64bits == Disponibles) then
    la instrucción se lanza a un sumador de 64-bits;
  else
    la instrucción espera en la cola;
  end if
else if (tipo instrucción == ARIT-LONG) then
  if (Sumadores-32bits == Disponibles) then
    la instrucción se lanza a un sumador de 32-bits;
  else if (Sumadores-64bits == Disponibles) then
    la instrucción se lanza a un sumador de 64-bits;
  else
    la instrucción espera en la cola;
  end if
else if (tipo instrucción == ARIT) then
  if Sumadores-64bits == Disponibles then
    la instrucción se lanza a un sumador de 64-bits;
  else
    la instrucción espera en la cola;
  end if
end if
```

La figura 5.12 presenta una posible implementación del algoritmo que acabamos de describir, para el caso particular de disponer de un sumador de 24-bits, un sumador de 32-bits y un sumador de 64-bits. En ella, *Código-Operación X* es una señal que cuando toma el valor 1 está indicando que la instrucción es del tipo X, *UsarSumX-bits* es una señal que vale 1 cuando los operandos de la instrucción no se encuentran en ninguna de las situaciones descritas en la figura 5.8 y por lo tanto se puede usar un sumador de X-bits para ejecutar la instrucción, *SumX-bits-Disponible* es una señal que indica si el sumador de X-bits se puede usar en ese momento y *EN-SumX* es la señal de capacitación de los sumadores.

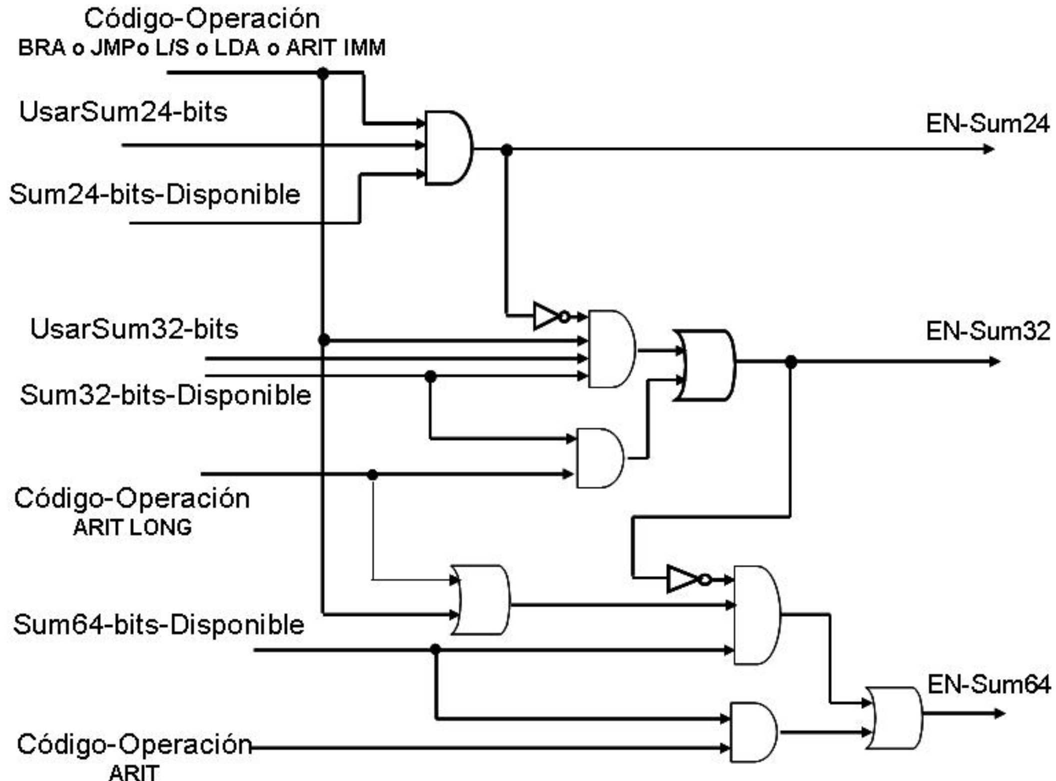


FIGURA 5.12: Posible implementación del algoritmo 1

En resumen, el protocolo de asignación de UFs se ha diseñado intentando que se ejecuten en el sumador de 24-bits el mayor número de instrucciones posibles pero teniendo en cuenta dos principios fundamentales:

1. Que el *Hardware* del árbitro consuma lo menos posible.

2. Que afecte lo mínimo posible al rendimiento del sistema.

5.3. Entorno de simulación

Las simulaciones para este trabajo han sido realizadas con el simulador *FU-Wattch* descrito en el capítulo 4. Este simulador, como ya se ha dicho, está basado en *SimpleScalar* [ALE02]. *SimpleScalar* es un simulador que proporciona un modelo preciso de un procesador de alto rendimiento. Además, tiene implementado un repertorio de instrucciones del conjunto de instrucciones de la arquitectura *Compaq Alpha*, que es el juego de instrucciones que acabamos de analizar en la sección 5.1.

Como modelo de consumo para las distintas estructuras del procesador, *FU-Wattch* usa el modelo de consumo del simulador *Wattch* [BTM00], que hemos mejorado para que proporcione datos fiables sobre el consumo en la unidad de ejecución del procesador.

5.3.1. Modelo de las UFs

En el capítulo 4 se ha descrito detalladamente el modelo de la unidad de ejecución que utiliza el simulador *FU-Wattch*. A modo de recordatorio en esta sección hacemos un pequeño resumen de dicho modelo. Como ya se ha dicho,

la unidad de ejecución está basada en el modelo de *clusters* del *Alpha* 21264 [Kes99]. Dicho modelo está compuesto por una unidad de enteros y otra de punto flotante como muestra la figura 4.1. La unidad de enteros está formada por dos *clusters*, cada uno de los cuales está dividido en dos *subclusters*. Estos *subclusters* contienen un sumador de 64-bits (ADD), una unidad lógica (LOGIC) y una unidad de desplazamiento. Además, en uno de los *subclusters* hay un multiplicador (MULT). Por otro lado, la unidad de punto flotante está formada por un cluster compuesto de un sumador (FP ADD) y un multiplicador (FP MULT) ambos en aritmética punto flotante.

5.3.2. Estimación del consumo en los sumadores

Como se ha comentado al principio de la sección 5.2, para poder aplicar nuestra propuesta se requieren distintas implementaciones de sumadores. Por un lado, sumadores de 64-bits con un diseño donde la prioridad es la rapidez a expensas de otros parámetros (en particular, área y consumo de potencia). Por otro lado, sumadores de 24 y 32-bits con un diseño cuyos principales requisitos son: 1) tener un camino crítico muy similar (aunque nunca mayor) al camino crítico del sumador de 64-bits y 2) optimizar el consumo.

Para modelar el consumo de los diferenten sumadores se ha asignado un valor de consumo al sumador de 64-bits y se ha reescalado dicho valor mediante un *coeficiente de proporcionalidad*, que llamaremos *cp*, para asignar el consumo a los sumadores de 24 y 32-bits. Es decir, el consumo del sumador de 32-bits se calcula en función del de 64-bits mediante la fórmula 5.1:

$$P_{S32} = cp_{S32} * P_{S64} \tag{5.1}$$

Y el consumo del de 24-bits, en función del de 32-bits, mediante la fórmula 5.2:

$$P_{S24} = cp_{S24} * P_{S32} \tag{5.2}$$

A continuación se presentan y justifican los valores que hemos usado para estos coeficientes, cp_{S32} y cp_{S24}:

- Por un lado los sumadores van a tener distinto número de bits (64-bits frente a 32 y 24-bits). Así, por ejemplo, el sumador de 32-bits respecto al de 64-bits:

  - Va a realizar la mitad de conmutaciones a la hora de calcular una operación, esto conlleva una reducción del consumo dinámico.
  - Va a tener la mitad de transistores, lo que implica una reducción del consumo estático.

  Luego para dos sumadores con la misma arquitectura, los sumadores de 32-bits consumen un 50 % menos que los sumadores de 64 bits, es decir, el consumo del sumador de 32-bits es 1/2 del consumo del sumador de 64-bits. El mismo razonamiento se puede aplicar para los de 24-bits, pudiendo decir que los sumadores de 24-bits consumen un 25 % menos que los sumadores de 32-bits, por lo tanto se puede decir que el consumo del sumador de 24-bits es 0.75 veces el consumo del sumador de 32-bits.

- Por otro lado, los sumadores de 32 y 24-bits son más rápidos que los de 64-bits (para el mismo tipo de sumador) [Mat03] [Vaz03] [Asl04] [YT06]. Sin embargo, estos no necesitan ser tan rápidos ya que el máximo retardo lo marca el sumador de 64-bits. Por lo tanto, a la hora de diseñarlos, los requisitos impuestos, respecto al retardo, a los sumadores de 32 y 24-bits son distintos a los impuestos al de 64-bits. Así, al sumador de 64-bits se le exige rapidez mientras que para los sumadores de 32 y 24-bits esa no es la prioridad ya que su camino crítico puede ser igual al del sumador de 64-bits. Esto implica que a la hora de diseñar los sumadores de 32 y 24-bits:

  - Se puede reducir el número de transistores, reduciendo con ello el consumo estático.
  - Se pueden aplicar técnicas de redimensionamiento del transistor y/o sustitución de puertas por otras con baja capacidad de carga de salida. Con ello se reduce el consumo dinámico en cada transición de la puerta [BOI96] [TPB98] [Bor99] [Nar05].
  - Se pueden aplicar técnicas para reducir el voltaje de *threshold* de los transistores, reduciendo el consumo estático [WCJ+98] [WCR+99] [KM06] [SSA06] [VMR06] [NMN07] .

Teniendo en cuenta todo esto, se puede elegir un sumador de 32-bits cuyo consumo sea hasta 1/3 del consumo del sumador de 64-bits. Resumiendo, hemos decidido usar los siguientes valores para el coeficiente de proporcionalidad: $cp_{S32} = P_{S32}/P_{S64} = 0{,}5$ *y* $0{,}33$ *y* $cp_{S24} = P_{S24}/P_{S32} = 0{,}75$

5.3.3. Parámetros del procesador modelado

Las tablas 5.2, 5.3 y 5.4 muestran las principales caracteristicas del procesador simulado. Como se puede ver en la tabla 5.2, el procesador puede lanzar hasta seis instrucciones por ciclo de la cola de lanzamiento, cuatro de operaciones con enteros y dos en PF. Esto es posible ya que en la unidad de ejecución existen cuatro *subclusters* de enteros y dos de punto flotante (ver figuras 4.1

TABLA 5.2: Configuración del núcleo del procesador simulado.

Parámetros	**Valores**
RUU tamaño	80 instrucciones
LSQ (ld/store queue)	tamaño 32
Tamaño de la cola Fetch	4 instrucciones
Ancho Fetch	4 instrucciones/ciclo
Ancho Decode	4 instrucciones/ciclo
Ancho Issue	6 instrucciones/ciclo (fuera de orden)
	(4 de enteros, 2 de PF)
Ancho Commit	11 instrucciones/ciclo (en orden)
Unidades Funcionales	4 *Subclusters* de enteros
	2 de PF

TABLA 5.3: Configuración del predictor del procesador simulado.

Parámetros	**Valores**
Bimodal	tamaño de la tabla 4K
De 2-niveles	1K cada nivel, 3 bits de historia
	tamaño de la meta-tabla 4K
BTB	512 conjuntos, asociatividad 4
Return-address stack	32 entradas
Penalización del fallo de predicción	7 ciclos

Tabla 5.4: Jerarquía de la memoria del procesador simulado.

Parámetros	**Tamaño**	**Bloques**	**Asocitividad**	**Latencia**
L1 cache de datos	512K	64	2	3
L1 cache de instrucciones	512K	64	2	2
L2 unificada	32M	64	1	6 (datos) 12 (instruc)
TLBs	**128 entradas completamente asociativa**			50

El modelo de unidad de ejecución que tiene el procesador se definió al principio de esta sección y se puede ver en la figura 4.1. La tabla 5.5 resume los valores tomados para el consumo de las componentes de la unidad de ejecución. Los valores de consumo se han obtenido de la literatura [BOI96] [BTM00] [ZF97]. En dicha tabla sólo se presenta el consumo del sumador de 64-bits porque el consumo de los sumadores de 32 y 24-bits se tomará en función del ratio aplicado, como se ha comentado en la sección anterior.

Tabla 5.5: Valores de consumo de las distintas UFs usadas, para una tecnología de 733MHz y 0.18μm

Sum de de enteros (64-bits)	**Unidad Lógica**	**Unidad desplaz**	**Mult de enteros (16-bits)**	**Sum de PF**	**Mult de PF**
1.16503W	0.104394W	0.505265W	0.897785W	3.57026W	3.57026W

Para la unidad de enteros, se han elegido cinco diseños diferentes con el objetivo de encontrar el número óptimo de sumadores de cada tipo que permita reducir el máximo de consumo con el mínimo de penalización en el rendimiento del procesador. Como Diseño-Base se ha tomado el descrito en la figura 4.1,

es decir, en este diseño tenemos 4 sumadores de enteros de 64-bits (uno en cada *subcluster*), en el resto de diseños sólo se han cambiado el número y el tipo de los sumadores de enteros. En los diseños que tienen cinco sumadores, como el número de *subclusters* es cuatro, uno de los cuatro *subclusters* tiene que tener un sumador más. A continuación se presentan los seis diseños que se han estudiado y el número total de sumadores que tiene cada uno:

- Diseño-Base: 4 sumadores de 64 bits. Total 4.
- Diseño-1: 1 sumador de 64-bits, 2 de 32-bits y 1 de 24-bits. Total 4.
- Diseño-2: 1 sumador de 64-bits, 1 de 32-bits y 2 de 24-bits. Total 4.
- Diseño-3: 1 sumador de 64-bits, 1 de 32-bits y 3 de 24-bits. Total 5.
- Diseño-4: 1 sumador de 64-bits, 3 de 24-bits. Total 4.
- Diseño-5: 1 sumador de 64-bits, 4 de 24-bits. Total 5.

Para elegir estas combinaciones se han tenido en cuenta dos cosas. Por un lado, que el número total de sumadores tiene que ser como mínimo 4, como en Diseño-Base, para penalizar lo mínimo el rendimiento y poder seguir lanzando cuatro instrucciones de enteros por ciclo. Por otro lado, que se podría dejar un sólo sumador de 64-bits, ya que el número de instrucciones que necesitan el sumador de 64-bits es bastante reducido (21.6 %) y el número de ciclos en los que se usa más de uno de estos sumadores representa solamente el 6.1 %, como se ha demostrado en la sección 5.1. Lo mismo se puede decir de los sumadores de 32-bits; en este caso sólo el 4.7 % de las instrucciones necesitan este sumador y el 5,1 % de los ciclos usan más de un sumador de 32-bits. Para este último

tipo de sumadores los porcentajes son tan bajos que incluso se podría llegar a prescindir de ellos, esto nos ha llevado a estudiar Diseño-4 y Diseño-5.

5.3.4. *Benchmarks*

Para las simulaciones se han elegido los *Benchmarks* del conjunto SPEC CPU2000 [SPE]. De nuevo, como la simulación completa de cada uno de ellos puede llevar semanas y no aporta más precisión en las medidas, se ha usado la herramienta Sim-point [PHC03] para seleccionar un conjunto de instrucciones que represente la ejecución total del programa. De esta forma, de cada *Benchmarks* se han simulado 100M de instrucciones elegidas con Sim-point. Como entradas para los *Benchmarks* se han tomado, las entradas de referencia.

5.3.5. Métrica

Se han usado tres tipos de magnitudes para medir el consumo de potencia:

- El valor estimado de la potencia media (*averaged power*). Este valor se obtiene sumando el consumo en cada ciclo sobre un número definido de ciclos de reloj.

- El cociente entre instrucciones por ciclo y el consumo de la estructura (IPC/consumo) [STT01]. Esta métrica es interesante para determinar que diseño obtiene el compromiso óptimo entre rendimiento y consumo para una técnica dada.

- El producto energía retardo Completo (CEDP) (ver la ecuación 2.5 del

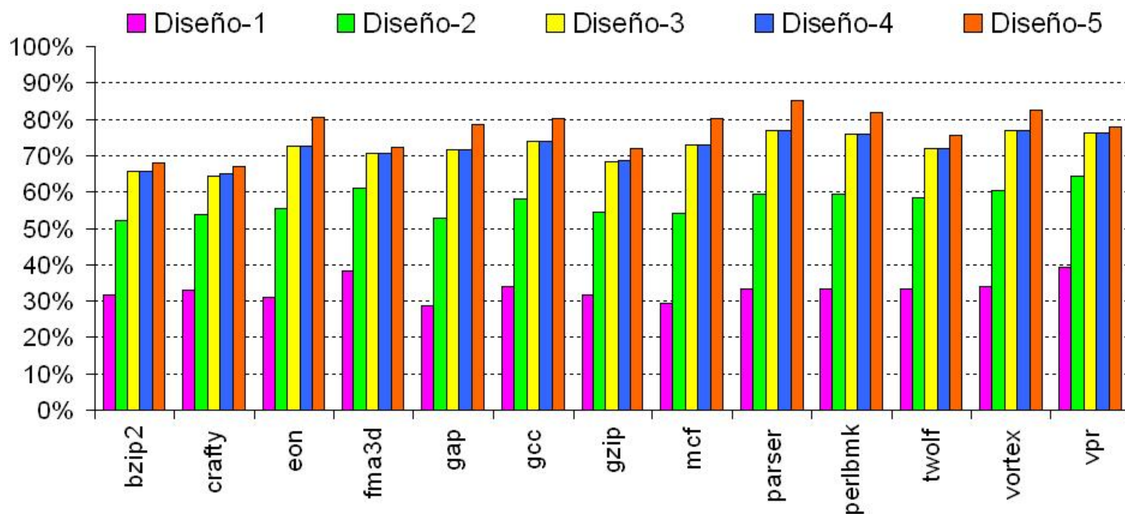
capítulo 2). Esta métrica es interesante para determinar cuando es efectivo un cambio.

5.4. Resultados experimentales

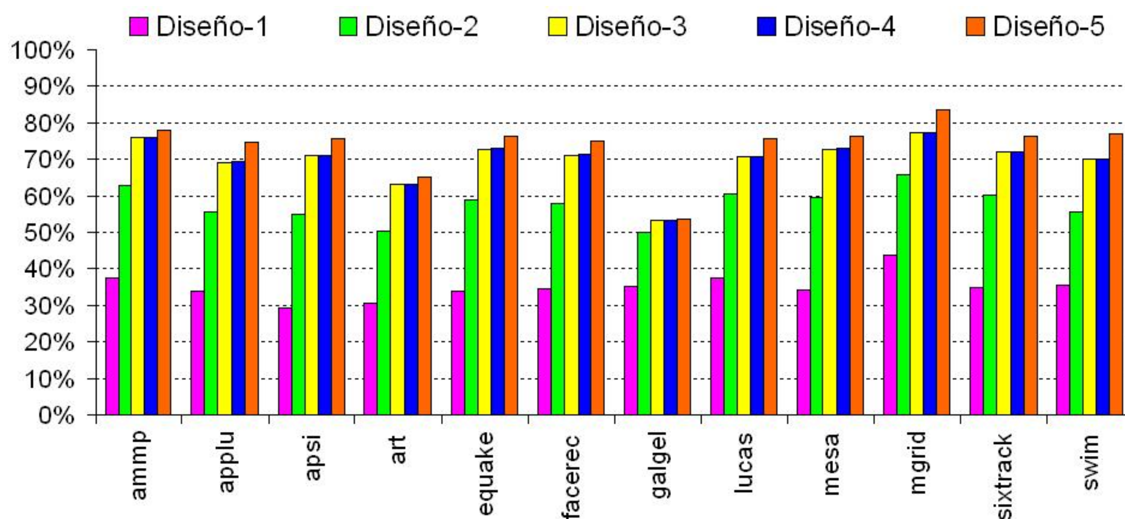
En esta sección se presentan los resultados obtenidos al aplicar la técnica descrita en la sección 5.2 en procesadores superescalares de alto rendimiento. Como ya se ha comentado anteriormente, se han elegido seis diseños diferentes para encontrar cual de ellos proporciona la máxima reducción de consumo en la unidades funcionales penalizando al mínimo el rendimiento del procesador. Como banco de pruebas se han usado el conjunto de *Benchmarks* de la sección 5.3.4 y las simulaciones se han realizado con el simulador *FU-Wattch* presentado en el capítulo 4.

La figura 5.13 muestra el porcentaje de instrucciones que se ejecutan en un sumador de 24-bits para cada uno de los diseños. La figura 5.13(a) muestra los resultados para los *Benchmarks* de enteros mientras que la 5.13(b) muestra los de PF. Como se puede observar, en esta figura *Diseño-Base* no aparece. Esto es debido a que este diseño sólo tiene sumadores de 64-bits, por lo tanto no se pueden ejecutar instrucciones en un sumador de 24-bits. Esto no ocurre en el resto de diseño, porque en ellos, además de un sumador de 64-bits, existen sumadores de 32/24-bits y por lo tanto hay instrucciones que pueden usar dichos sumadores.

Observando los resultados para Diseño-1 (S64-2S32-S24) y Diseño-2 (S64-S32-2S24) podemos ver que el porcentaje de instrucciones que se ejecutan en un sumador de 24-bits es bastante inferior al obtenido en la sección 5.1.2. En



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 5.13: Instrucciones que se ejecutan en un sumador de 24-bits para cada uno de los diseños (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

dicha sección vimos que alrededor del 80 % de las instrucciones de un *Benchmark* podrían usar un sumador de 24-bits, existiendo algunos *Benchmarks* que incluso lo superan (ver la figura 5.10). De todo esto se deduce que instrucciones que podrían haber usado un sumador de 24-bits terminan ejecutandose en uno de 32/64-bits por no encontrar disponible algún sumador de 24-bits. La política de asignación de sumadores está descrita en el Algoritmo 1 de la sección 5.2.

En Diseño-3 (S64-S32-3S24) y Diseño-4 (S64-3S24) se observa que el porcentaje de instrucciones que se ejecutan en un sumador de 24-bits ha aumentado considerablemente. Esto es debido a que en estos diseños existe un sumador más de 24-bits, de manera que, instrucciones que en Diseño-1 y Diseño-2 no se podían ejecutar en un sumador de 24-bits por no encontrar uno disponible, ahora si lo encuentran. Aunque en Diseño-3 y Diseño-4 el porcentaje de instrucciones que usan un sumador de 24-bits es mayor que en Diseño-1 y Diseño-2, sigue sin alcanzar el valor predicho en la sección 5.1.2. Esto nos lleva a la conclusión de que para poder atender a todas las instrucciones que pueden usar un sumador de 24-bits, es necesario disponer de más de tres sumadores de este tipo.

Diseño-5 (S64-4S24) tiene cuatro sumadores de 24-bits. Observando la figura 5.13 podemos ver que es en este diseño en el que el porcentaje de instrucciones que se ejecutan en el sumador de 24-bits alcanza los valores óptimos. Esto demuestra que el número de sumadores de este tipo que se necesitan para poder atender a todas las instrucciones que pueden usarlo, es cuatro.

En la tabla 5.6 se puede ver el porcentaje de instrucciones, promediados sobre todos los *Benchmarks*, que se ejecutan en cada uno de los tipos de suma-

dores para cada uno de los diseños. En ella se puede observar todo lo comentado de la figura anterior. Como ya se ha visto, en Diseño-1 y Diseño-2 el porcentaje de instrucciones que se ejecutan en un sumador de 24-bits no alcanza el 73.7 %, en promedio, predicho por el estudio realizado en la sección 5.1.2, quedandose en un 33.5 % ,en promedio, para el primero y un 56.2 % para el segundo. Esto es debido a que instrucciones que podrían haber usado el sumador de 24-bits terminan lanzándose a los sumadores de 32/64-bits, por lo que el porcentaje de instrucciones que se ejecutan en estos sumadores (39.9 % y 26.6 % ,respectivamente, para el primero y 16.5 % y 27.3 % para el segundo) es también diferente, en este caso superior, al obtenido por dicho estudio (4.7 % y 21.5 %).

TABLA 5.6: Porcentaje de instrucciones, promediados sobre todos las *Benchmarks*, que se ejecutan en cada uno de los tipos de sumadores para cada uno de los diseños.

Diseños	Sum 24-bits	Sum 32-bits	Sum 64-bits
Diseño-1 (S64-2S32-S24)	33.5 %	39.9 %	26.6 %
Diseño-2 (S64-S32-2S24)	56.2 %	16.5 %	27.3 %
Diseño-3 (S64-S32-3S24)	69.4 %	7.8 %	22.8 %
Diseño-4 (S64-3S24)	69.6 %	0 %	30.4 %
Diseño-5 (S64-4S24)	73.8 %	0 %	26.2 %

En Diseño-3 el porcentaje de instrucciones que se ejecutan en el sumador de 24-bits ha aumentado considerablemente (69.4 %), aunque sigue sin llegar a ser el óptimo, por lo que el porcentaje de instrucciones que se ejecutan en los sumadores de 32/64-bits disminuye (7.8 % y 22.8 % respectivamente). Esta dismunución se observa sobre todo en el número de instrucciones que usan el sumador de 32-bits, lo que qiere decir que las instrucciones que no

encontraban un sumador de 24-bits disponible en Diseño-1 y Diseño-2, eran ejecutadas principalmente en el sumador de 32-bits, y solamente se ejecutan en el sumador de 64-bits si no hay ninguno disponible de los anteriores.

En Diseño-4, aumenta el porcentaje de instrucciones que usan el sumador de 64-bits (30.4 %) debido a que no existen sumadores de 32-bits y por lo tanto todas las instrucciones que se ejecutaban en dicho sumador, en los anteriores diseños, ahora tienen que ejecutarse en uno de 64-bits.

Por último, en Diseño-5 el porcentaje de instrucciones que se ejecutan en cada tipo de sumador alcanza los valores predichos en la sección 5.1.2, 73.7 % de las instrucciones se ejecutan en los sumadores de 24-bits y 26,2 % en el sumador de 64-bits (4.7 % de los que podrían usar un sumador de 32-bits más 21.5 % de los que necesitan uno de 64-bits).

De todo esto se puede deducir que, cuatro es el número óptimo de sumadores de 24-bits que debe tener la unidad de enteros para explotar al máximo el hecho de que la mayoría de las instrucciones que requieren sumadores para su ejecución pueden usar un sumador de 24-bits en vez de uno de 64-bits, como se ha demostrado en la sección 5.1.2.

Hasta ahora sólo se han estudiado los diferentes diseños buscando el que más aprovecha los resultados del estudio realizado en la sección 5.4. Desde este punto de vista, el mejor es el diseño 5, donde todas las intrucciones que requieren un sumador de 24-bits siempre encuentran uno disponible ya que en este diseño hay cuatro sumadores de 24-bits y las instrucciones se lanzan de cuatro en cuatro. Pero el objetivo principal de nuestra técnica es reducir el consumo en las unidades funcionales, por lo que acontinuación vamos a analizar el consumo que implica cada uno de estos diseños.

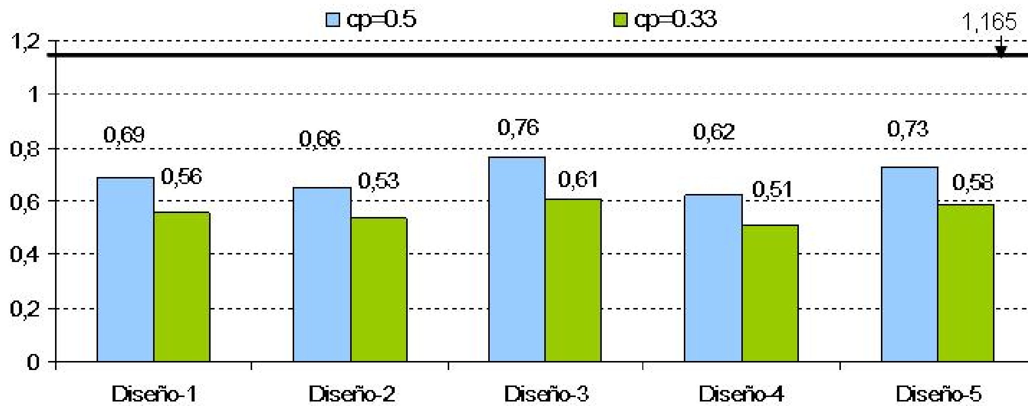


FIGURA 5.14: Consumo estático en los sumadores para cada uno de los diseños, y cada uno de los cp (cp_{S32} = 0,5 y 0,33 y cp_{S24} = 0,75); (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)).

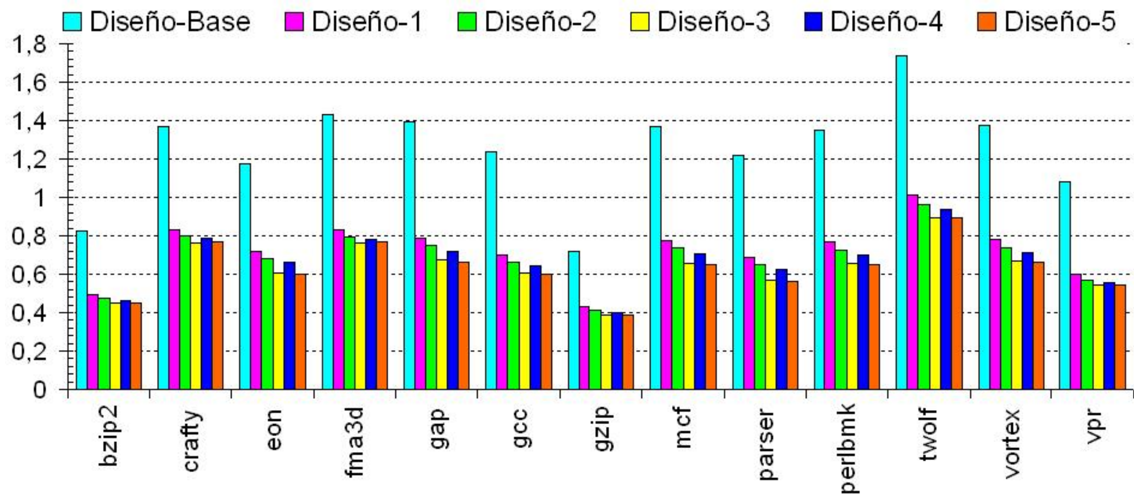
Sobre el consumo estático se puede decir que este no depende de las carga de trabajo, sólo depende del número y del tipo de sumadores que tenga cada diseño. La figura 5.14 muestra los valores del consumo estático en los sumadores para cada uno de ellos, usando diferentes cp para calcular el consumo de los sumadores de 24/32-bits ($cp_{S32} = 0{,}5$ y $0{,}33$ y $cp_{S24} = 0{,}75$). En ella se puede observar que todos los diseños tienen menos consumo estático que Diseño-Base (este tiene un consumo de 1.165W), incluso los diseños que tienen cinco sumadores, uno más que *Diseño-Base*. Hay dos razones para este comportamiento, por un lado que el número de sumadores de 64-bits se ha reducido a uno y por otro lado que los sumadores de 24/32-bits consumen bastante menos que los de 64-bits. Como se puede ver en dicha figura, en términos de consumo estático, el mejor diseño para todos los cp es, el diseño 4 ya que es el que tiene el mejor balance entre: el mínimo número de sumadores (cuatro) y el máximo número de ellos que son de menor consumo (tres de 24-bits).

El consumo dinámico si depende de los *Benchmarks*, ya que este consumo

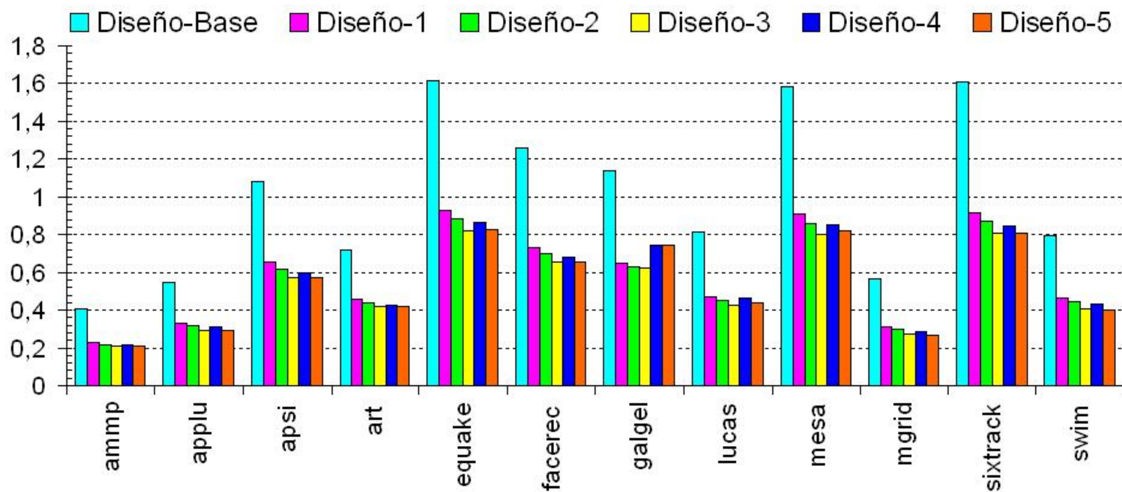
depende del número de instrucciones que se ejecuten en cada tipo de sumador. Las figuras 5.15 y 5.16 muestran el consumo dinámico en los sumadores para cada uno de los *Benchmarks* simulados, usando diferentes *cp* para calcular el consumo de los sumadores de 24/32-bits ($cp_{S32} = 0{,}5 \; y \; 0{,}33 \; y \; cp_{S24} = 0{,}75$). En todas ellas se puede ver que *Diseño-Base* es la que presenta mayor consumo dinámico. Esto es debido a que en él todas las instrucciones que requieren sumador se ejecutan en uno de 64-bits, cosa que no ocurre en el resto de diseños, ya que en estos hay algunas instrucciones que se ejecutan en sumadores de 24/32-bits. También podemos observar en dichas gráficas, que el resto de diseños tiene un consumo muy parecido.

Despues de Diseño-Base, Diseño-1 es el que más consumo dinámico presenta. Esto era de esperar porque en este diseño sólo existe un sumador de 24-bits lo que hace que el porcentaje de instrucciones que se ejecutan en dicho sumador, 33.5 % en promedio (ver tabla 5.6) esté muy por debajo del óptimo, 73.7 % en promedio (ver sección 5.1.2) y por lo tanto la reducción del consumo dinámico sea menor.

Si observamos Diseño-3 y Diseño-5, podemos decir que tienen practicamente el mismo consumo dinámico (salvo el caso del *Benchmark galgel* que ya comentaremos después). Entre estos dos diseños, la diferencia está en lo siguiente: Diseño-3 tiene instrucciones que se podrían haber ejecutado en el sumador de 24-bits y, por no encontrar uno disponible, se ejecutan en el sumador de 32-bits, es decir hay más instrucciones que usan el sumador de 32-bits de las necesarias, un 4,4 % más.

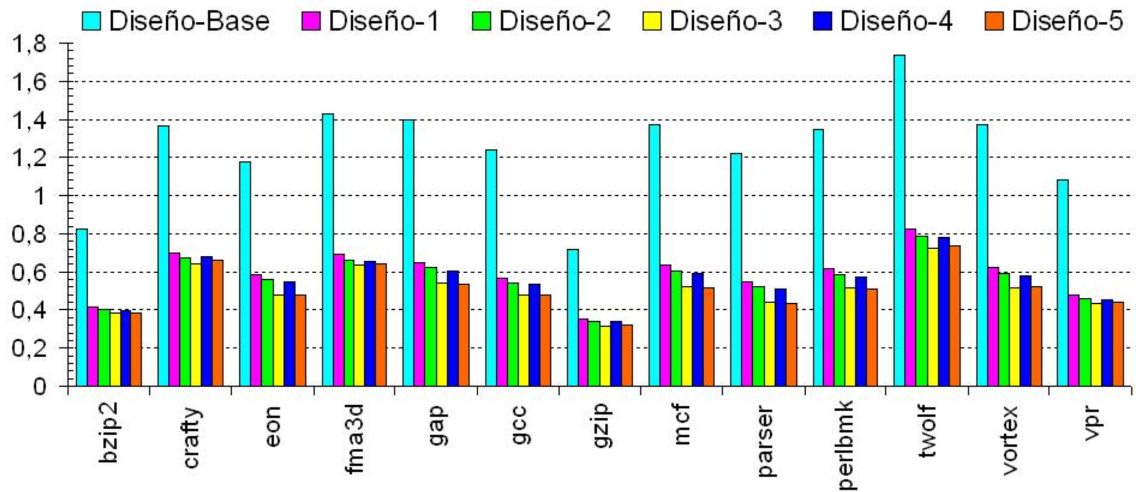


(a) INTSPEC 2000

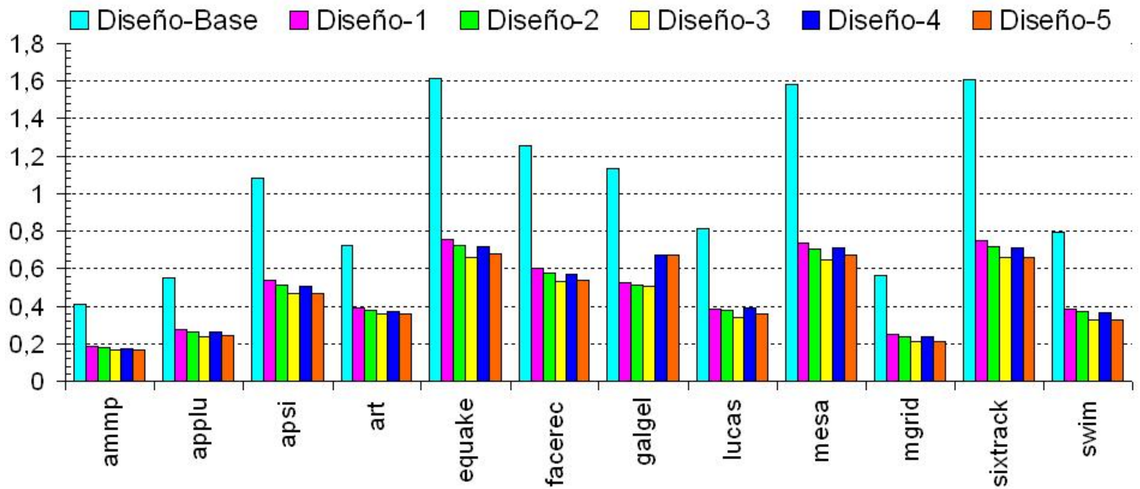


(b) FPSPEC 2000

FIGURA 5.15: Consumo dinámico en los sumadores para cada uno de los diseños, cuando $cp_{S32} = 0{,}5$; (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 5.16: Consumo dinámico en los sumadores para cada uno de los diseños, cuando cp_{S32} = 0,33; (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

En Diseño-5 esto ya no sucede, en él todas las instrucciones que se pueden ejecutar en los sumadores de 24-bits lo hacen, luego el porcentaje de instrucciones que usan dicho sumador ha aumentado el 4,4 % que tenía de menos Diseño-3, alcanzando los valores predichos en la sección 5.1.2.

Al consumir el sumador de 24-bits menos que el sumador de 32-bits y haber más instrucciones que usan el sumador de 24-bits, el consumo en Diseño-5 debería ser menor que en Diseño-3. Pero, como por otro lado, Diseño-5 no tiene sumadores de 32-bits, las instrucciones que necesitan dicho sumador se tienen que ejecutar en uno de 64-bits, por lo que el número de instrucciones que usan un sumador de 64-bits también ha aumentado, un 3,4 %, lo que implica un aumento de consumo. Luego podemos concluir que el consumo que se ha ahorrado con el aumento de instrucciones que se ejecutan en un sumador de 24-bits, se ha perdido al aumentar el número de instrucciones que usan el sumador de 64-bits. Esto es lo que hace que ambos diseños tengan un consumo muy parecido.

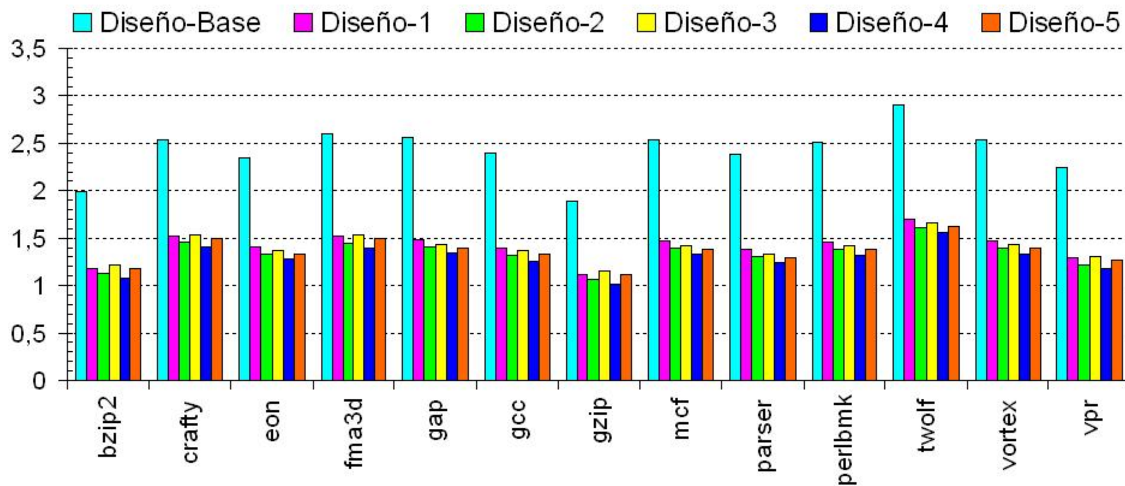
Como ya hemos comentado, *galgel* es el único *Benchmark* que no tiene Diseño-3 y Diseño-5 con un consumo muy parecido. En este programa el consumo Diseño-5 es claramente mayor que el de Diseño-3. Esto es debido a que en este *Benchmark* el número de instrucciones que se ejecutan en el sumador de 32-bits es muy alto(ver figura 5.10) ya que esta carga de trabajo contiene muchas instrucciones del tipo ARIT LONG, que son instrucciones que necesitan ejecutarse en un sumador de 32-bits (ver figura 5.7). Esto implica que, en los diseños que no tienen sumadores de 32-bits, estas instrucciones tienen que ejecutarse en un sumador de 64-bits, lo que hace que aumente el consumo en dichos diseños.

Respecto a Diseño-2 y Diseño-4, podemos decir se encuentran en una situación parecida a Diseño-3 y Diseño-5, es decir, estos diseños también tienen un consumo muy parecido. En este caso, el número de instrucciones que se ejecutan en el sumador de 24-bits es un 13 % más en Diseño-4 que en Diseño-2, mientras que el número de instrucciones que se ejecutan en el sumador de 64-bits es sólo un 3,1 % mayor. Aunque en este caso, el aumento del número instrucciones que se ejecutan en cada sumador (24 y 64-bits) es distinto, el consumo que se ha ahorrado con el aumento del número instrucciones que usan el sumador de 24-bits, se ha perdido al aumentar el número de instrucciones que se ejecutan en el sumador de 64-bits. Esto tiene sentido porque el sumador de 64-bits consume más del doble del de 24-bits. La conclusión que podemos sacar de todo esto es que el aumento del porcentaje de instrucciones que se ejecutan en el sumador de 24-bits sólo compensa si es más del 13 %.

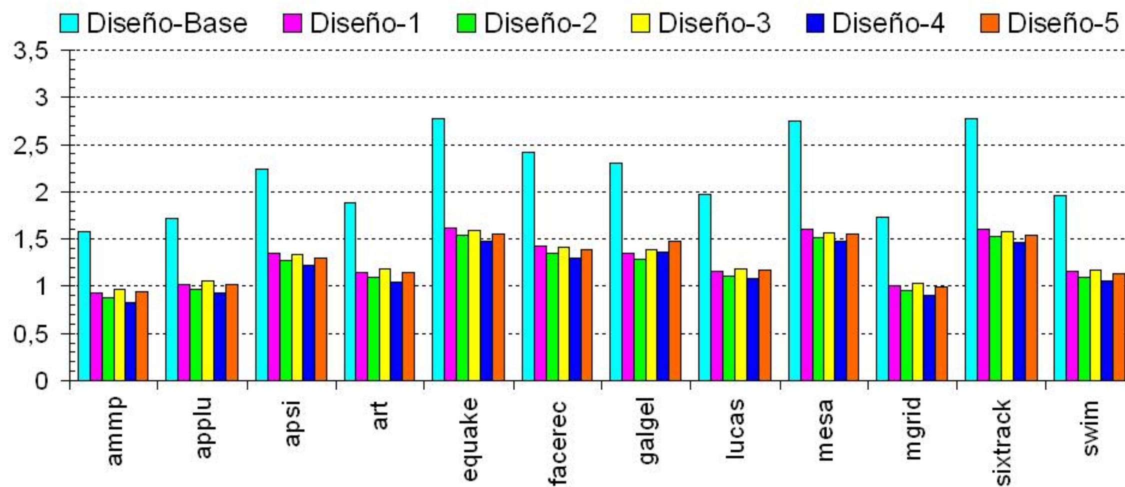
Como conclusión final podemos decir que los mejores diseños, en cuanto al consumo dinámico se refiere, pueden ser tanto Diseño-3 como Diseño-5. Aunque como hemos visto, si un programa tiene bastantes instrucciones tipo ARIT LONG (operan con datos de 32-bits), Diseño-5 no es el óptimo ya que no tiene sumadores de 32-bits y esto afectaría al rendimiento.

Las figuras 5.17 y 5.18 muestran el consumo total, tanto dinámico como estático, en los sumadores para cada uno de los *Benchmarks* simulados y para cada uno de los ratios usados.

Hasta ahora se ha visto que el mejor diseño no es el mismo en el ahorro de estático (Diseño-4) que en el dinámico (Diseño-3, Diseño-5). La diferencia entre estos diseños radica en que:

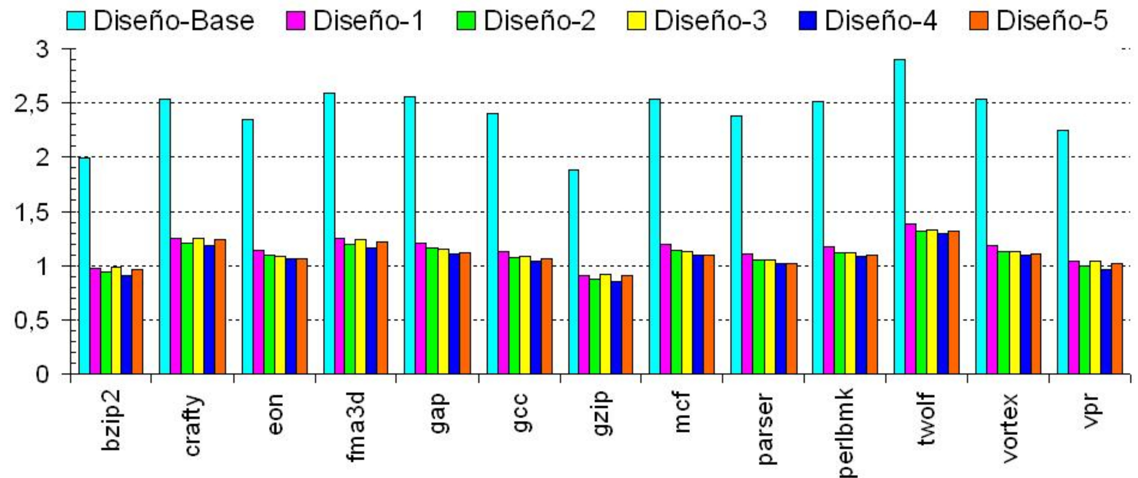


(a) INTSPEC 2000

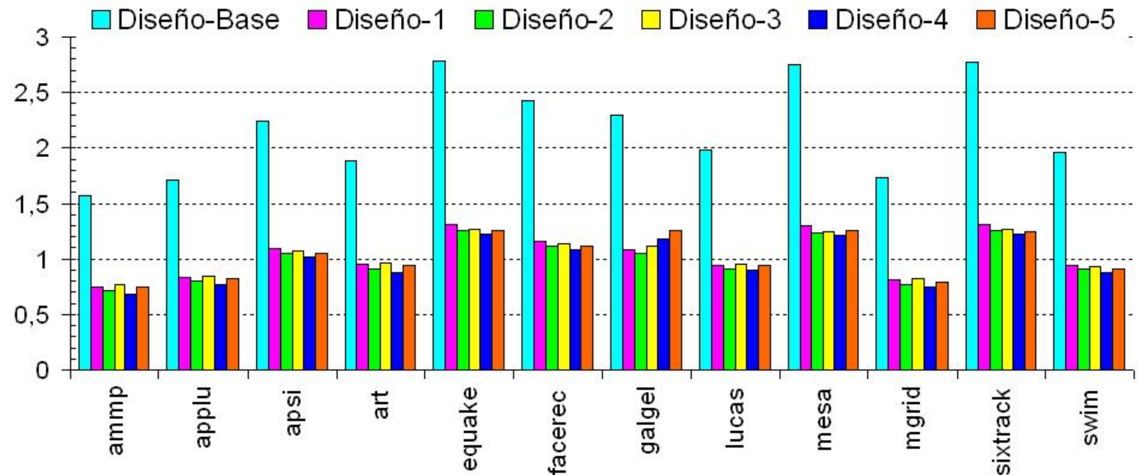


(b) FPSPEC 2000

FIGURA 5.17: Consumo total en los sumadores para cada una de los diseños, cuando $cp_{S32} = 0{,}5$; (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 5.18: Consumo total en los sumadores para cada uno de los diseños, cuando $cp_{S32} = 0{,}33$; (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

- Por un lado Diseño-3 y Diseño-5 tienen más consumo estático que Diseño-4 por tener 5 sumadores en vez de 4.

- Por otro, Diseño-4 tiene más consumo dinámico ya que, tiene más instrucciones que se ejecutan en el sumador de 64-bits que Diseño-3 y Diseño-5. Esto es debido a no tener sumador 32-bits y a que el número de sumadores de 24-bits no es el óptimo.

Si observamos el consumo total, se puede decir que el mejor diseño es Diseño-4. Este diseño no es el que más consumo dinámico ahorra, pero si el que más consumo estático. Esto nos viene a confirmar la importancia que está tomando el consumo estático frente al dinámico, por lo que es algo que, no se puede despreciar a la hora de evaluar cualquier optimización que se haga respecto al consumo.

TABLA 5.7: Ahorro de potencia, promediado sobre todos los *Benchmarks*, tanto en los sumadores como en toda la unidad de ejecución, para cada uno de los diseños.

	Sumadores		**Unidad de Ejecución**	
cp_{S32}	**0.5**	**0.33**	**0.5**	**0.33**
Diseño-1 (S64-2S32-S24)	41.22 %	52.31 %	17.24 %	21.8 %
Diseño-2 (S64-S32-2S24)	44.04 %	54.07 %	18.4 %	22.6 %
Diseño-3 (S64-S32-3S24)	41.06 %	52.8 %	17.2 %	22.1 %
Diseño-4 (S64-3S24)	46.16 %	55.09 %	19.5 %	23.2 %
Diseño-5 (S64-4S24)	42.56 %	53.36 %	18 %	22.5 %

La tabla 5.7 presenta un resumen del ahorro conseguido tanto en el consumo los sumadores y como en toda la unidad de ejecución al aplicar esta técnica. En ella se puede ver el ahorro, promediado sobre todos los *Benchmarks*, conseguido con cada diseño y para cada uno de los *cp* aplicados. Para calcular el consumo en toda la unidad de ejecución se han tenido en cuenta el resto de UFs descritas

en la sección 5.3.1 (ver figura 4.1) y el consumo del árbitro descrito en la sección 5.2. Como se puede observar en dicha tabla, se consigue un ahorro de potencia bastante importante con todos los cp usados y para todos los diseños, llegando a reducir el consumo en los sumadores hasta un 55.09 % y en la unidad de ejecución hasta un 23.2 % para el diseño 4 cuando se usa un $cp_{32} = 0{,}33$.

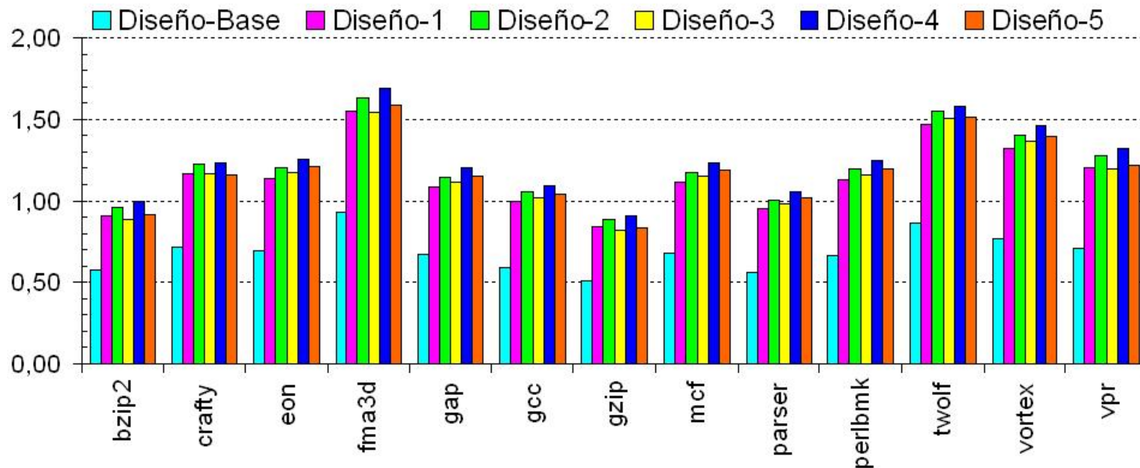
Con el objetivo de demostrar que esta técnica reduce el consumo sin prácticamente afectar al rendimiento, la tabla 5.8 muestra el valor de IPC (instrucciones por ciclo) para cada *Benchmarks*. En ella se puede observar que como mucho la pérdida de rendimiento del procesador es un 2 %, al aplicar Diseño-4 y Diseño-5.

TABLA 5.8: IPC para cada diseño, promediado sobre todos los *Benchmarks*

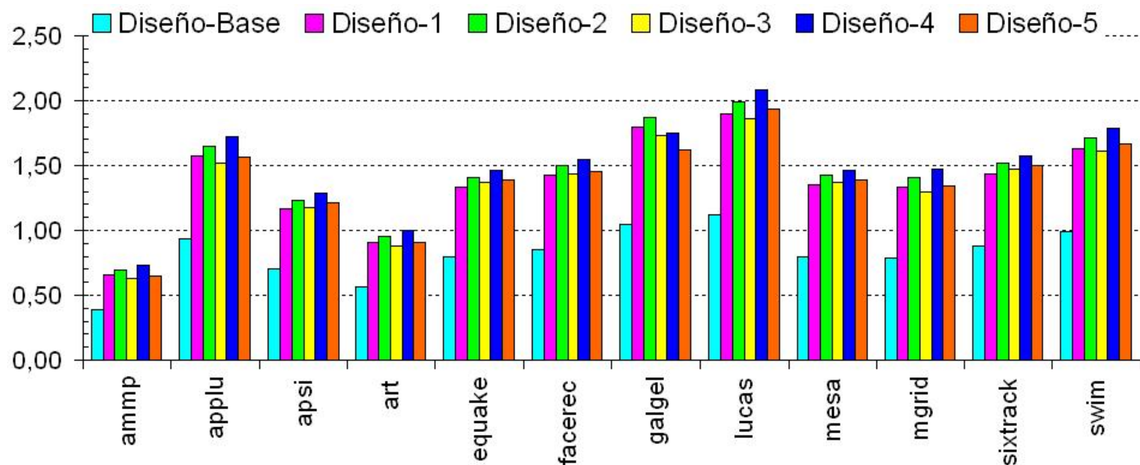
Diseños	IPC
Diseño-Base (4S64)	1.78
Diseño-1 (S64-2S32-S24)	1.74
Diseño-2 (S64-S32-2S24)	1.74
Diseño-3 (S64-S32-3S24)	1.74
Diseño-4 (S64-3S24)	1.71
Diseño-5 (S64-4S24)	1.71

Otra métrica interesante es el cociente entre IPC y el consumo en los sumadores. Esta métrica, como se explicó en la sección 5.3.5, nos sirve para obtener el diseño que presenta el mejor compromiso entre ahorro de potencia y rendimiento.

Las figuras 5.19 y 5.20 nos muestran el cociente entre IPC y el consumo en los sumadores para cada uno de los *Benchmarks*, usando diferentes cp ($cp_{S32} = 0{,}5\ y\ 0{,}33\ y\ cp_{S24} = 0{,}75$).

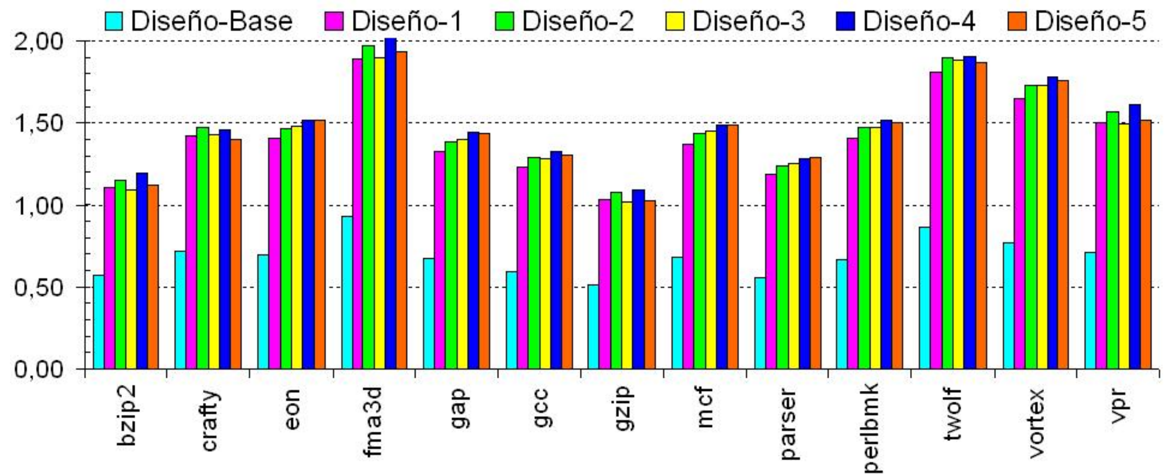


(a) INTSPEC 2000

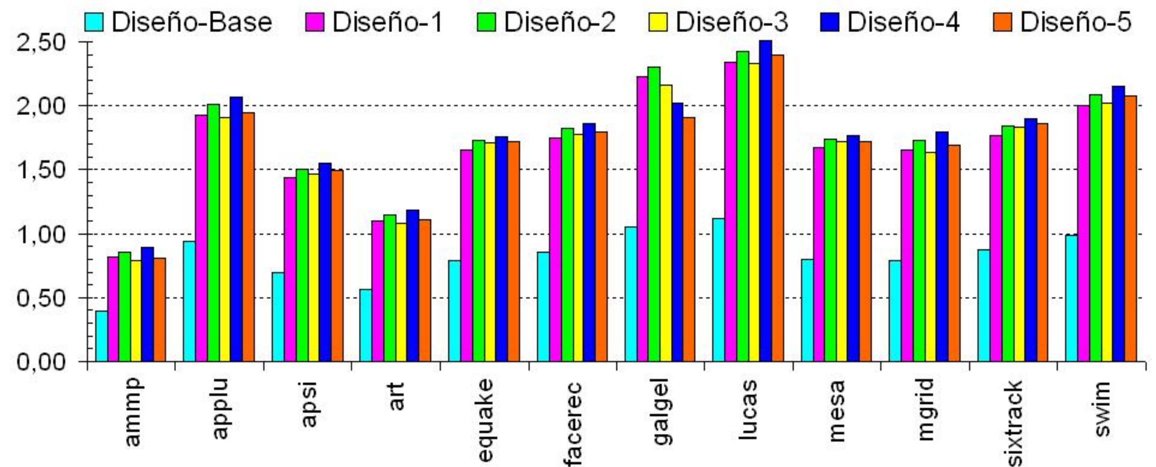


(b) FPSPEC 2000

FIGURA 5.19: IPC/(consumo en los sumadores) para cada uno de los diseños, cuando cp_{S32} = 0,5; Diseño-Base=(4S64); Diseño-1=(S64-S32-2S24); (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.



(a) INTSPEC 2000



(b) FPSPEC 2000

FIGURA 5.20: IPC/(consumo en los sumadores) para cada uno de los diseños, cuando $cp_{S32} = 0{,}33$; Diseño-Base=(4S64); Diseño-1=(S64-S32-2S24); (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)). (a) muestra los resultados para INTSPEC 2000, (b) muestra los resultados para FPSPEC 2000.

Observando dichas figuras podemos decir que Diseño-Base es, con bastante diferencia, el que presenta peores resultados. Para el resto de diseños el valor de IPC/consumo es mucho mayor que para Diseño-Base. Esto es debido a que el consumo de todos ellos es menor que el consumo de Diseño-Base (ver las figuras 5.17 y 5.18), mientras que el IPC es practicamente el mismo en todos los diseños (ver tabla 5.8). Como conclusión podemos decir que todos los diseños presentan un buen compromiso entre ahorro de potencia y rendimiento, siendo Diseño-4 el mejor de todos ellos.

Por último, para completar el estudio de todos los diseños propuestos, también se ha evaluado la métrica denominada Producto Energía Retardo Completo (CEDP) definida en el capítulo 2. En esta métrica, se establece el valor del CEDP de referencia como 1 y si un diseño tiene un CEPD menor que 1, se puede decir que dicho diseño compensa la perdida de rendimiento con la reducción del conumo.

Las figuras de la 5.21 a la 5.25 muestran el valor del CEDP, aplicado al ahorro de consumo obtenido en los sumadores, para cada uno de los diseños. En particular en las figuras 5.21, 5.22 y 5.23 podemos observar que en Diseño-1, Diseño-2 y Diseño-3 existen bastantes *Benchmarks* que presentan un valor del CEDP igual a 1. Para el resto de *Benchmarks* dicho valor no supera el 1.01. Por lo tanto podemos considerar que en estos diseños se compensan la penalización en el rendimiento con la reducción obtenida en el consumo de potencia. Para Diseño-4 y Diseño-5, figuras 5.24 y 5.25, el valor del CEDP es algo mayor que en los anteriores diseños, pero aun así no supera el 1.025.

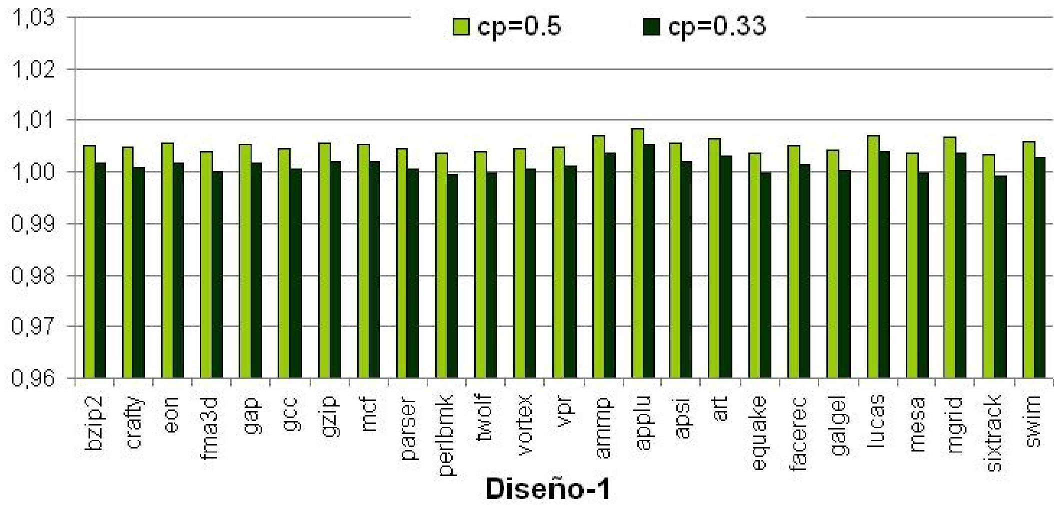


FIGURA 5.21: Producto Energía-Retardo aplicado en los sumadores para Diseño1=(S64-2S32-S24), en todos los *Benchmarks* y cada uno de los *cp*.

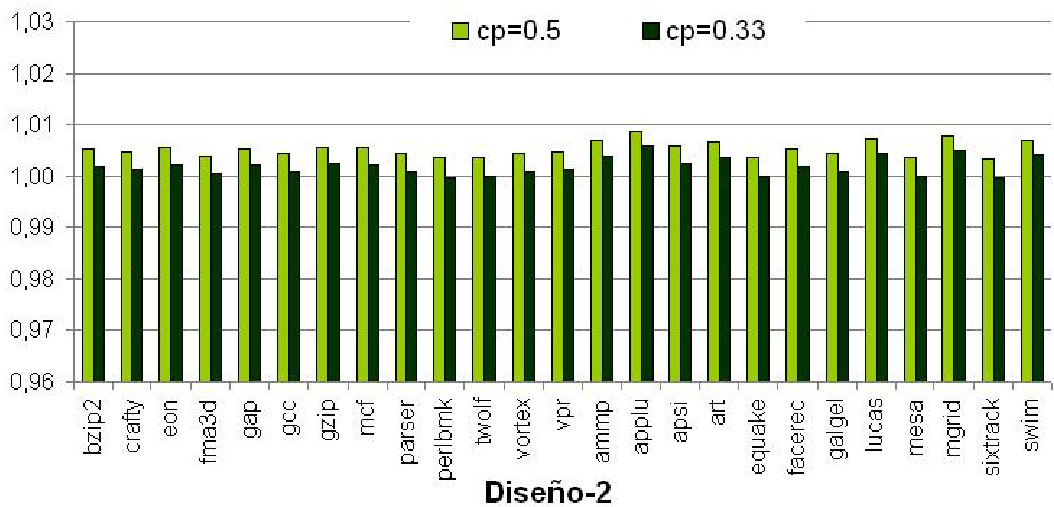


FIGURA 5.22: Producto Energía-Retardo aplicado en los sumadores para Diseño2=(S64-S32-2S24), en todos los *Benchmarks* y cada uno de los *cp*.

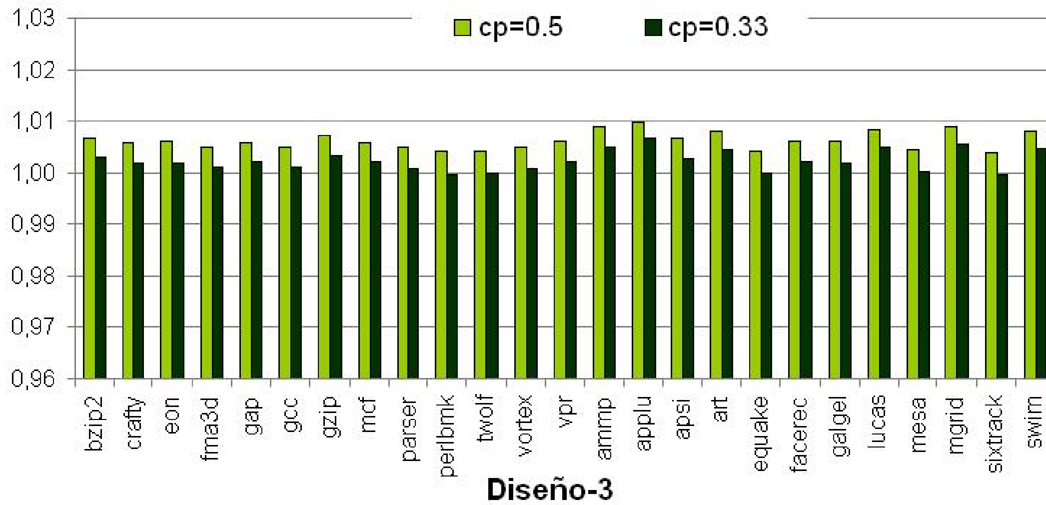


FIGURA 5.23: Producto Energía-Retardo aplicado en los sumadores para Diseño3=(S64-S32-3S24), en todos los *Benchmarks* y cada uno de los *cp*.

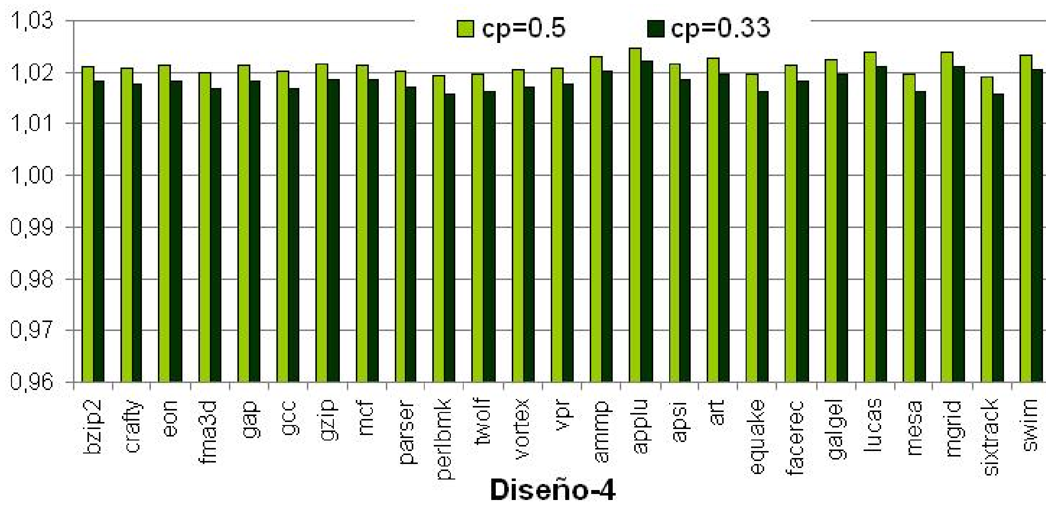


FIGURA 5.24: Producto Energía-Retardo aplicado en los sumadores para Diseño4=(S64-3S24), en todos los *Benchmarks* y cada uno de los *cp*.

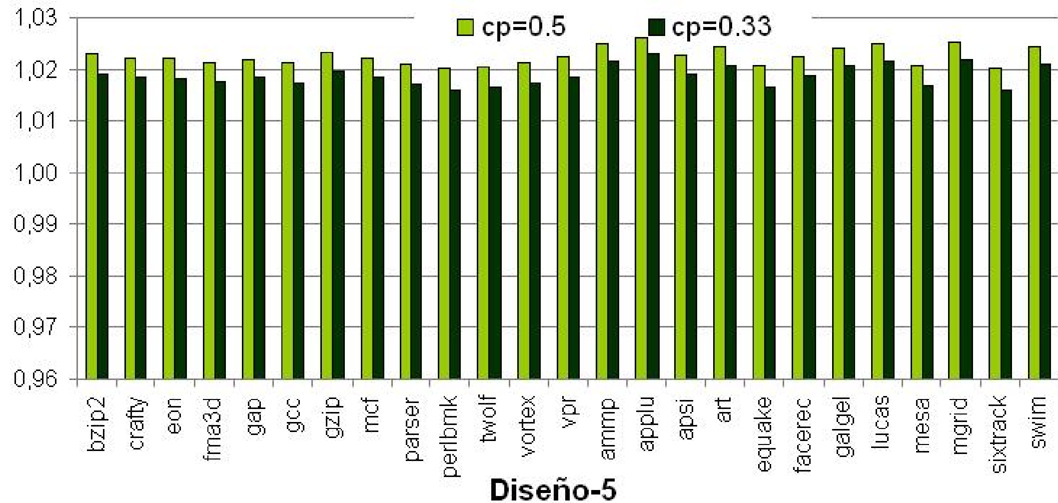


FIGURA 5.25: Producto Energía-Retardo aplicado en los sumadores para Diseño5=(S64-4S24), en todos los *Benchmarks* y cada uno de los *cp*.

Las figuras de la 5.26 a la 5.30 muestran el valor del CEDP, aplicado al ahorro de consumo obtenido en la unidad de ejecución para todos los diseños. Como se puede observar en dichas figuras, los valores del CPED obtenidos en la unidad de ejecución presentan el mismo comportamiento que los obtenidos en los sumadores.

Como se puede ver en las figuras 5.26, 5.27 y 5.28, Diseño-1, Diseño-2 y Diseño-3 presentan bastantes *Benchmarks* que tienen un valor del CEDP igual a 1 y en el resto de *Benchmarks* dicho valor no supera el 1.01. Para Diseño-4 y Diseño-5, figuras 5.29 y 5.30, el valor del CEDP es algo mayor que en los anteriores diseños, pero aun así no supera el 1.025. Esto nos reafirmar que podemos considerar que en estos diseños se compensan la penalización en el rendimiento con la reducción obtenida en el consumo de potencia.

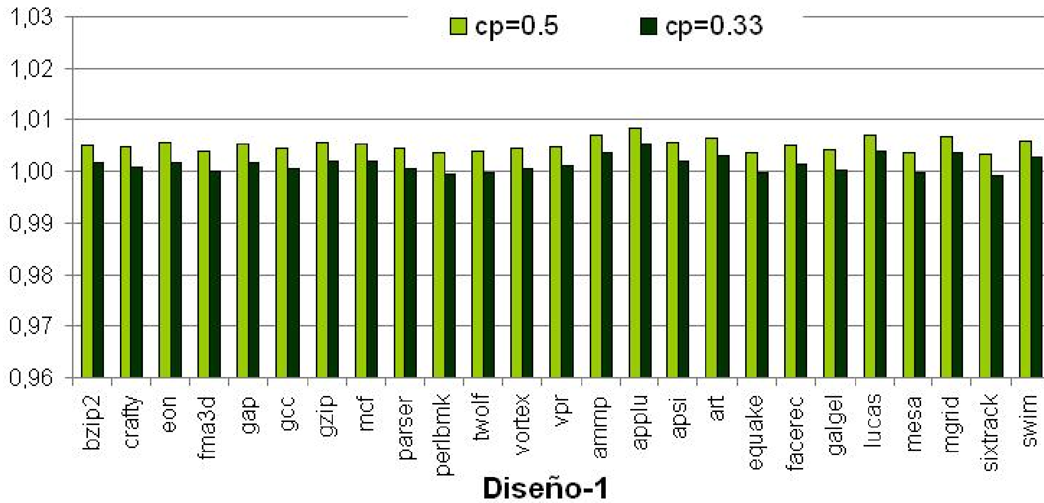


FIGURA 5.26: Producto Energía-Retardo aplicado en la unidad de ejecución para Diseño1=(S64-2S32-S24), en todos los *Benchmarks* y cada uno de los *cp*.

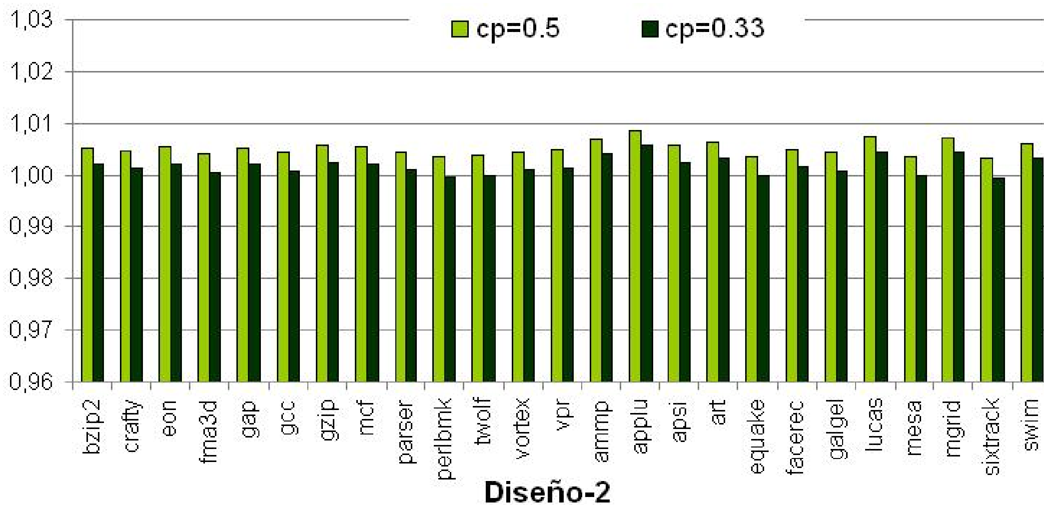


FIGURA 5.27: Producto Energía-Retardo aplicado en la unidad de ejecución para Diseño2=(S64-S32-2S24), en todos los *Benchmarks* y cada uno de los *cp*.

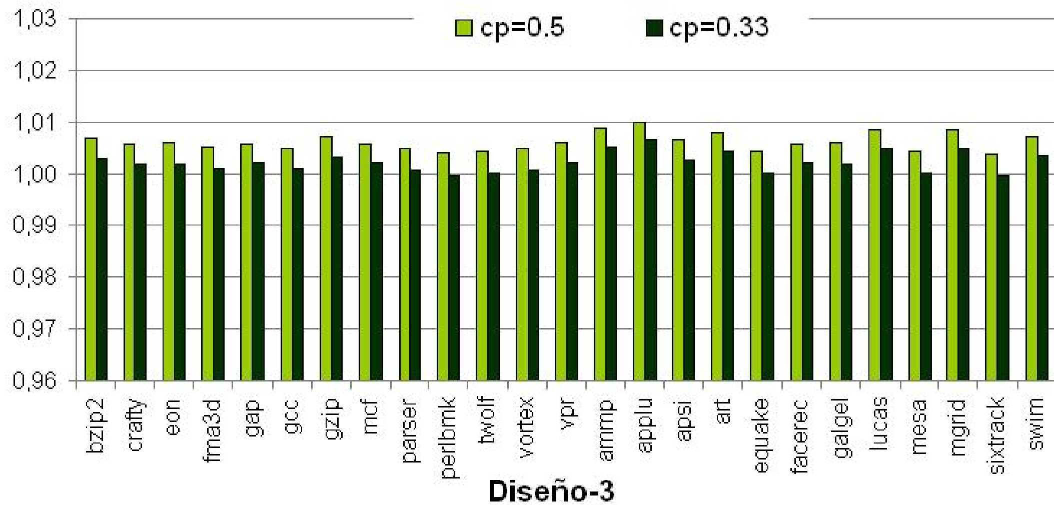


FIGURA 5.28: Producto Energía-Retardo aplicado en la unidad de ejecución para Diseño3=(S64-S32-3S24), en todos los *Benchmarks* y cada uno de los *cp*.

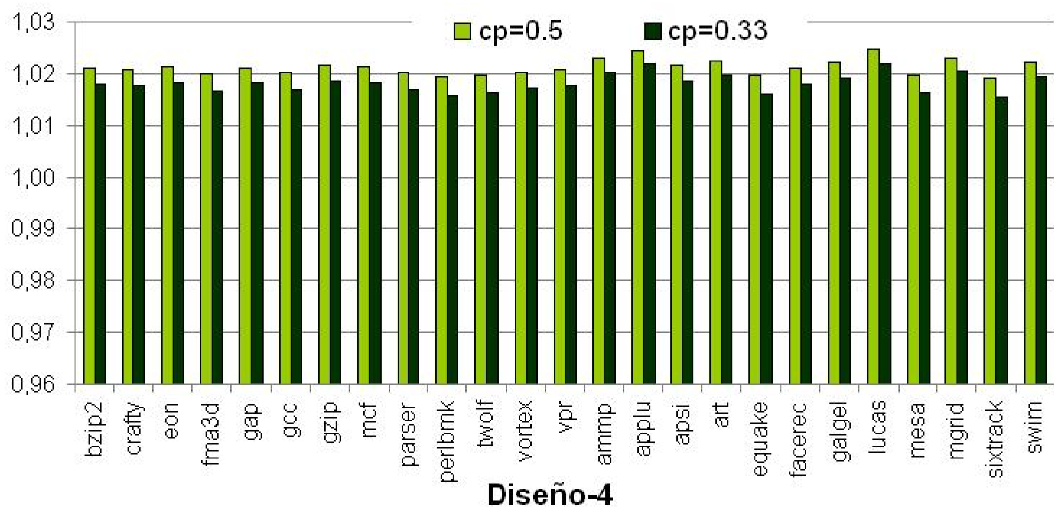


FIGURA 5.29: Producto Energía-Retardo aplicado en la unidad de ejecución para Diseño4=(S64-3S24), en todos los *Benchmarks* y cada uno de los *cp*.

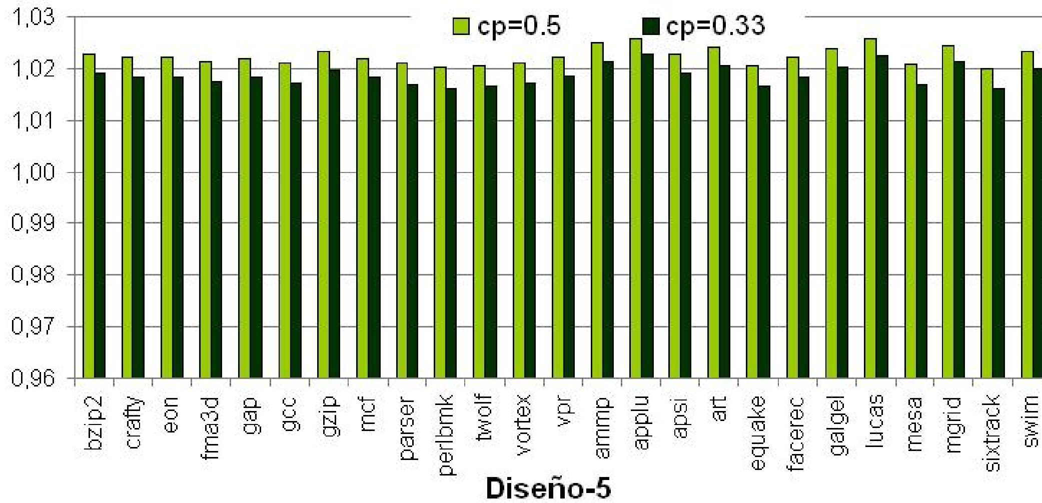


FIGURA 5.30: Producto Energía-Retardo aplicado en la unidad de ejecución para Diseño5=(S64-4S24), en todos los *Benchmarks* y cada uno de los *cp*.

5.5. Conclusiones

En este capítulo, en primer lugar se ha proporcionado un estudio donde se muestra que de todas las instrucciones que usan un sumador para ejecutarse, hay un porcentaje muy alto que no necesitan un sumador 64-bits porque se podrían ejecutar en un sumador de 32/24 bits.

Para este estudio nos hemos centrado en el conjunto de Instrucciones de la arquitectura *Alpha* [ACC] y los resultados que hemos obtenido son:

- El 73.7 % de las instrucciones pueden usar un sumador de 24-bits.
- El 4.7 % de las instrucciones pueden usar un sumador de 32-bits.
- El 21.6 % de las instrucciones pueden usar un sumador de 64-bits.

En segundo lugar hemos presentado una técnica para reducir el consumo en la unidad de ejecución, que explota estos resultados. Esta reduce el consumo en las UFs mediante el uso de sumadores de diferentes tamaños y por lo tanto diferentes consumos. Para elegir el tipo de sumador donde ejecutar cada instrucción se mira su código de operación (sólo en algunos casos hace falta comprobar algún bit más). La propuesta se presenta a nivel de microarquitectura y reduce tanto el consumo dinámico como el estático. Como se ha demostrado, con esta técnica se consigue reducir el consumo en los sumadores hasta un 55 %.

Se han probado seis diseños diferentes para la unidad de ejecución con el objetivo de encontrar el número óptimo de sumadores de cada tipo (64, 32 o 24-bits) que permita reducir el máximo de consumo con el mínimo de penalización en el rendimiento del procesador. A modo de resumen la tabla 5.9 presentan los seis diseños que se han estudiado:

TABLA 5.9: Diferentes diseños de la unidad de ejecución

Diseños	**Sum64**	**Sum32**	**Sum24**
Diseño-Base	4	0	0
Diseño-1	1	2	1
Diseño-2	1	1	2
Diseño-3	1	1	3
Diseño-4	1	0	3
Diseño-5	1	0	4

Para las simulaciones se han elegido los *Benchmarks* del conjunto SPEC CPU2000 [SPE]. Las simulaciones para este trabajo han sido realizadas con el simulador *FU-Wattch* descrito en el capítulo 4. La figura 5.31 muestra a

modo de resumen, el ahorro de consumo estático, dinámico y total en los sumadores, promediado para todos los *Benchmarks*, usando los dos valores de cp_{S32}, y para todos los diseños. Obeservando dicha figura podemos llegar a las siguientes conclusiones:

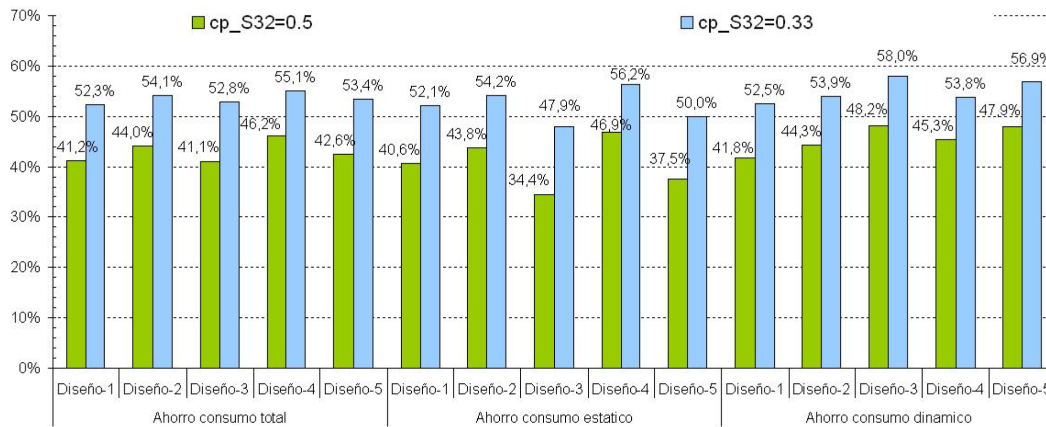


FIGURA 5.31: Ahorro de potencia estática, dinámica y total en los sumadores para cada uno de los diseños, y cada uno de los *cp*; Los valores están promediados sobre todos los *Benchmarks*. (Diseño-Base=(4S64); Diseño-1=(S64-2S32-S24); Diseño-2=(S64-S32-2S24); Diseño-3=(S64-S32-3S24); Diseño-4=(S64-3S24); Diseño-5=(S64-4S24)).

- El mejor diseño, en cuanto al consumo estático se refiere, es Diseño-4 donde el ahorro de consumo estático es del 56,2 %, en promedio. Esto es debido a que es el diseño que tiene el mejor balance entre el mínimo número de sumadores (cuatro) y el máximo número de ellos que son de menor consumo (tres de 24-bits).

- El diseño más óptimo, en cuanto al consumo dinámico se refiere, es Diseño-3 que consigue un ahorro del consumo dinámico del 58 %, en promedio, para el caso de $cp_{S32} = 0{,}33$. Esto es debido a que con este diseño es posible lanzar el mayor número instrucciones a los sumadores

de 32/24-bits.

- En lo referente al consumo total se puede concluir que Diseño-4 es el mejor llegando a ahorrar hasta el 55 %, en promedio, para el caso de $cp_{S32} = 0{,}33$. Este diseño no es el que más consumo dinámico ahorra, pero si el que más consumo estático. Esto nos viene a confirmar la importancia que está tomando el consumo estático frente al dinámico, por lo que es algo que, no se puede despreciar a la hora de evaluar cualquier optimización que se haga respecto al consumo.

- La aplicación de esta técnica consigue ahorrar potencia en todos los diseños, llegando a reducir el consumo total en los sumadores hasta un 55 %, en promedio, y en la unidad de ejecución hasta un 23.2 %, en promedio, para Diseño-4 cuando se usa un $cp_{32} = 0{,}33$.

- También se ha demostrado que esta técnica reduce el consumo sin prácticamente afectar al rendimiento. Para ello se ha obtenido el tiempo de ejecución de cada *Benchmarks* para los seis diseños estudiados. Los resultados muestran que la penalización introducida en el rendimiento es como mucho un 2 %. Esto ocurre al aplicar Diseño-4 y Diseño-5.

- En lo referente al compromiso entre rendimiento y ahorro de potencia se ha visto que todos los diseños son mejores que Diseño-Base. De todos ellos, Diseño-4 presenta los mejores resultados.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En el capítulo introductorio de este trabajo se marcaron unos objetivos fundamentales:

1. Mejorar las herramientas actuales de simulación para evaluar el consumo de potencia en UFs de procesadores superescalares.

2. Buscar posibles mejoras en las UFs para reducir su consumo.

Estos objetivos se han alcanzado con éxito como muestran los resultados experimentales de los capítulos 4 y 5. En la siguiente sección se presenta un resumen de las principales aportaciones y conclusiones obtenidas en este trabajo.

Para poder alcanzar el primero de los objetivos, en el capítulo 2 se ha realizado previamente un breve repaso de las herramientas para evaluar el

consumo de potencia y con ello se ha dado una visión general de las ideas básicas en esta área.

Como hemos visto en el capítulo 2, para evaluar nuevos diseños y realizar comparativas es necesario aplicar un modelo de estimación que permita modelar el consumo de un procesador. Es decir, debemos caracterizar cada una de las partes del sistema para la cual queramos estimar el consumo de potencia. Támbien hemos visto que según el nivel de descripción en que se esté trabajando existen distintos modelos, nivel bajo, medio y alto.

Nuestro interés se ha centrado en los modelos de alto nivel que utilizan los simuladores, porque, aunque con los modelos de niveles de abstracción inferiores se consiguen resultados más precisos, presentan dos inconvenientes importantes para ser utilizados en etapas tempranas de diseño:

- Necesitan una descripción completa del procesador, incluyendo la implementación física (layout).

- Su elevado coste computacional, ya que para calcular con precisión los valores de potencia, es necesario resolver una gran cantidad de ecuaciones, muchas de ellas bastante complejas.

Por estos dos motivos, parece inviable evaluar el consumo de un procesador con un simulador de este estilo, y mucho menos simular una carga de trabajo representativa como puede ser un juego de *Benchmarks* SPEC.

Dentro de los simuladores que utilizan modelos de alto nivel, nos hemos interesado por los basados en *SimpleScalar* [ALE02], ya que éste se ha convertido en la principal herramienta de simulación y modelado de sistemas de

alto rendimiento. Como se ha visto en la sección 2.3.2, este tipo de simuladores basados en *SimpleScalar* calculan el consumo a partir de la actividad de computo obtenida al ejecutar trazas de cargas representativas.

Dentro de este estilo de simuladores hemos elegido *Wattch* [BTM00] para evaluar la técnica que se ha propuesto en este trabajo porque es un simulador que incluye un modelo de alto nivel para el consumo del procesador y que se ha convertido en la herramienta más usada para estimar el consumo en el área de conocimiento de la arquitectura de computadores.

Analizando a fondo esta herramienta se ha llegado a las siguientes conclusiones:

1. Observando la clasificación de los simuladores que hemos presentado en la tabla 2.1 podemos decir que *Wattch*:

    - Respecto al alcance o ambito de la simulación, es un simulador de conjunto de instrucciones, es decir simula únicamente el repertorio de instrucciones de un procesador o microcontrolador y está codificado con lenguaje de alto nivel, de modo que imita la funcionalidad del procesador manteniendo los valores de sus registros en variables internas.
    - Si nos fijamos en el tipo de entrada, *Wattch* es un simulador basado en trazas, es decir, toma como valores de entrada un flujo de instrucciones previamente ejecutado en la misma arquitectura, 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 tomada como entrada contiene toda la información rela-

cionada 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.

Todas estas caracteristicas hacen de *Wattch* un simulador con una alta velocidad de ejecución y válido para la evaluación del consumo, puesto que en este tipo de pruebas no es tan importante el código simulado como los accesos a las estructuras.

2. Sin embargo, este simulador estima el consumo en las UFs de manera poco precisa. Por ejemplo, *Wattch* considera una UF para la multiplicación de enteros. Sin embargo a la hora de calcular el consumo asociado a una instrucción de multiplicación de enteros, *Wattch* supone que el consumo de esta operación es el mismo que el asociado al sumador de enteros y lo mismo ocurre con otras UFs. Es decir, en términos de consumo de potencia, *Wattch* modela toda la unidad de ejecución como un sumador, sin tener en cuenta que tipo de operación se está realizando.

Debido a esta carencia, llegamos a la conclusión de que necesitamos conseguir un modelo más preciso para estimar los consumos de potencia, tanto estática como dinámica, en los clusters de las unidades funcionales de enteros y de punto flotante.

En el capítulo 4 hemos presentado una versión del simulador *Wattch*, que hemos llamado *FU-Wattch*, a la cual hemos añadido algunas modificaciones para estimar con mayor precisión el consumo en las UFs. Con los resultados presentados en dicho capítulo se puede asegurar que *FU-Wattch* es un simulador que estima el consumo en toda la unidad de ejecución de manera más

precisa de lo que lo hace *Wattch*. Esto permite usar este simulador como herramienta en el estudio del diseño de técnicas de bajo consumo. La comparativa de *Wattch* con *FU-Wattch* muestra que hay una gran diferencia entre los valores obtenidos con cada uno de los simuladores. Con *Wattch* el porcentaje que representa el consumo de las UFs respecto al consumo total del procesador es un 8 % (de media) para los *Benchmarks* de enteros y un 10 % (de media) para los de punto flotante. Mientras que con *FU-Wattch* estos porcentajes son el 11 % y 13,5 % (de media) respectivamente, que son valores más acordes con lo referenciado en la literatura [GBJ98] [BTM00] [LFDD03].

Con esto hemos conseguido nuestro primer objetivo: obtener una herramienta más realista para examinar nuevas arquitecturas y organizaciones de la unidad de ejecución de enteros como las que hemos propuesto en este trabajo.

En lo que se refiere a la segunda de las metas planteadas, en el capítulo 3 hemos hecho un repaso de las tecnicas a nivel de microarquitectura más relevantes para reducir el consumo en las diferentes estructuras de los procesadores actuales. Dentro de estas, se ha dedicado una sección especial a las técnicas de bajo consumo aplicadas a las unidades funcionales.

Como hemos visto en este repaso, muchas de las propuestas aplicadas a unidades funcionales explotan el hecho de que en la ejecución de un programa existe un alto porcentaje de *operandos narrow* ([CJC00] [BM00] [GM05] [Kim07]). El inconveniente principal de estas técnicas es que necesitan identificar los operandos que son *narrows*. Esto implica añadir una lógica que chequee todos los operandos para identificar cuales lo son, lo que conlleva un consumo extra, y una penalización en el rendimiento mientras se hacen estas deteccio-

nes. Otra de las desventajas de las anteriores ténicas es que éstas sólo reducen el consumo dinámico.

También hemos visto técnicas que reducen ambos consumos apagando la unidad de ejecución cuando las UFs llevan un periodo de tiempo sin usarse, como las propuestas en [DKA+02] [RPOG02] [HBS+04] [YAE06] y [TSC07]. Pero estos mecanismos necesitan una lógica adicional que detecte cuando una unidad funcional no se está utilizando y que genere una señal que la permita entrar en modo-durmiente. Dicha lógica aporta un consumo extra e introduce un retardo debido a que hay que sacar a las UFs de su modo-durmiente antes de poder usarlas de nuevo.

En el capítulo 5 hemos presentado diferentes diseños para reducir el consumo en las UFs que también explotan el alto porcentaje de *operandos narrow* que existen en los programas, pero que a diferencia de estas propuestas:

- Se basa en el código de operación de la instrucción. Es decir, no necesitan chequear todos los bist de los operandos para detectar si son *operandos narrow*. Esto hace que la lógica de detección sea muy sencilla ya que puede reusar parte del *Hardware* de asignación de UFs a las instrucciones que ya existe en los procesadores, y por lo tanto que prácticamente no se añada consumo extra.

- Usan UFs de diferentes tamaños y por lo tanto de diferentes consumos por lo que no sólo se consigue reducir el consumo dinámico sino también el estático.

- Debido a la sencillez de la lógica de asignación de UFs y a que no necesitamos usar técnicas de gating para reducir el consumo estático (ya que

esto lo hacemos usando diferentes modelos de UFs) con nuestras diseños no introducimos practicamente penalización en el rendimiento.

- Además, es compatible con las técnicas de gating descritas en el capítulo 3, por lo que se podría incluso consiguir una mayor reducción del consumo.

En el capítulo 5, en primer lugar se ha proporcionado un estudio donde se muestra que de todas las instrucciones que usan un sumador para ejecutarse, hay un porcentaje muy alto que no necesitan un sumador 64-bits porque se podrían ejecutar en un sumador de 32/24 bits. Para este estudio nos hemos centrado en el conjunto de instrucciones de la arquitectura *Alpha* [ACC] y los resultados que hemos obtenido son:

- El 73.7 % de las instrucciones pueden usar un sumador de 24-bits.
- El 4.7 % de las instrucciones pueden usar un sumador de 32-bits.
- El 21.6 % de las instrucciones pueden usar un sumador de 64-bits.

En segundo lugar se ha presentado una técnica para reducir el consumo en la unidad de ejecución que explota estos resultados. Esta reduce el consumo en las UFs mediante el uso de sumadores de diferentes tamaños y por lo tanto diferentes consumos. Para elegir el tipo de sumador donde ejecutar cada instrucción se mira su código de operación (sólo en algunos casos hace falta comprobar algún bit más). La propuesta se presenta a nivel de microarquitectura y reduce tanto el consumo dinámico como el estático. Como se ha demostrado, con esta técnica se consigue reducir el consumo en los sumadores hasta un 55 %.

Se han probado seis diseños diferentes para la unidad de ejecución con el objetivo de encontrar el número óptimo de sumadores de cada tipo (64, 32 o 24-bits) que permita reducir el máximo de consumo en la unidad de ejecución, con el mínimo de penalización en el rendimiento del procesador. A modo de resumen, la tabla 6.1 presenta los seis diseños que se han estudiado:

TABLA 6.1: Diferentes diseños de la unidad de ejecución

Diseños	**Sum64**	**Sum32**	**Sum24**
Diseño-Base	4	0	0
Diseño-1	1	2	1
Diseño-2	1	1	2
Diseño-3	1	1	3
Diseño-4	1	0	3
Diseño-5	1	0	4

Como han demostrado los resultados de las simulaciones expuestos en el capítulo 5, todos los diseños consiguen ahorrar potencia respecto a Diseño-Base, siendo Diseño-4 el mejor. Este diseño no es el que más consumo dinámico ahorra, pero si el que más consumo estático. Esto nos viene a confirmar la importancia que está tomando el consumo estático frente al dinámico, por lo que es algo que no se puede despreciar a la hora de evaluar cualquier optimización que se haga respecto al consumo.

Como conclusión final podemos decir que la aplicación de esta técnica consigue ahorrar potencia en todos los diseños, llegando a reducir el consumo total en los sumadores hasta un 55 %, en promedio, y en la unidad de ejecución hasta un 23.2 %,en promedio, para Diseño-4 cuando se usa un $cp_{32} = 0{,}33$, como se puede ver en la figura 5.31.

También se ha demostrado que esta técnica reduce el consumo sin afectar demasiado al rendimiento. Para ello se ha obtenido el tiempo de ejecución de cada *Benchmarks* para los cinco diseños estudiados. Los resultados muestran que la penalización introducida en el rendimiento está alrededor del 2 %.

En lo referente al compromiso entre rendimiento y ahorro de potencia se ha visto que todos los diseños son mejores que Diseño-Base. De todos ellos, Diseño-4 presenta los mejores resultados.

6.2. Trabajo Futuro

Una vez presentadas las conclusiones de este trabajo de investigación, en esta sección se describen las diferentes líneas de investigación que quedan abiertas y que serán afrontadas en el futuro.

La línea de investigación más inmediata es la optimización de la lógica de asignación de sumadores a las instrucciones. Como se ha visto en el capítulo 5, con el protocolo que hemos implementado para la asignación de UFs se sobreestima el número de instrucciones que no se pueden ejecutar en el sumador de 24-bits. Esto es debido a que dicho protocolo se ha diseñado teniendo en cuenta dos principios fundamentales: por un lado que consuma lo menos posible, por otro lado que afecte lo mínimo posible al rendimiento del sistema.

Nuestra intención ahora se centrará en mejorar el protocolo para ser capaz de detectar el número exacto de instrucciones que pueden usar un sumador de 24-bits. Por ejemplo, en el trabajo aquí presentado, todas las instrucciones ARIT se ejecutan en un sumador de 64-bits. Sin embargo se podría mirar si alguno de los operandos de estas instrucciones tiene un valor *narrow* y

en tal caso podría usarse un sumador de menor tamaño. Con esta mejora en el protocolo de asignación de sumadores se espera conseguir aumentar el número de instrucciones que se ejecutan en sumadores de menor tamaño y como consecuencia se obtendría una mayor reducción del consumo en la unidad de ejecución.

Por otro lado, en el capítulo 3 se han expuesto algunas técnicas para reducir el consumo en las UFs. Estamos estudiando la posibilidad de combinar algunas de ellas con nuestros diseños para obtener una mayor reducción en el consumo. Por ejemplo:

- La técnica de gating aplicada a las UFs. Es decir, cuando las UFs llevan un periodo de tiempo sin usarse se activa el modo durmiente. En este modo, se minimiza el consumo de potencia estática. Nuestra técnica se puede combinar con la propuesta de [TSC07], en la que usan el compilador para identificar largos periodos en los que las UFs estarán inactivas. Esto lo consiguen mirando los códigos de operación de las instrucciones. Esta información se comunica al *Hardware* usando directivas, de forma que durante la ejecución de un programa el *Hardware* conoce los periodos de tiempo durante los cuales las UFs no van a ser usadas y podrían desconectarse, evitando el consumo estático. También se sabe en qué momento se vuelven a necesitar y por lo tanto cuándo hay que volver a activarlas. Una mayor reducción en el consumo estático de la unidad de ejecución se conseguiría al aplicar esta técnica a los sumadores disponibles en cada uno de los diseños que hemos propuesto en este trabajo

- Otra de las técnicas estudiadas consiste en usar UFs con distintas latencias, por ejemplo de 1 y de 2 ciclos, de manera que las instrucciones cuyo resultado no se vaya a usar antes de dos ciclos se ejecutan en la UF lenta y por lo tanto de menos consumo como proponen Yan et ál. en [YT06]. En nuestros diseños todos los sumadores tienen latencia de un ciclo. Una manera de reducir más el consumo en estos diseños consiste en sustituir alguno de estos sumadores por sumadores con dos ciclos de latencia y por lo tanto con menor consumo tanto dinámico como estático.

- Además podemos explotar el hecho de que existe un alto porcentaje de operandos que tienen valor cero (un 45 % de las instrucciones *Load/Store* tienen el desplazamiento con valor cero [MLOJ98] y el 22 % de las instrucciones de suma o resta tienen alguno de sus operandos con valor cero [Kim07]). Estudiaremos evitar el acceso a un sumador cuando se detecte que uno de los dos operandos de una instrucción suma, o el sustraendo de una instrucción resta, es cero. Lo mismo se hará con las instrucciones *Load/Store* que tienen el desplazamiento con valor cero. De esta manera se ahorran cómputos innecesarios en las UFs y por lo tanto se reduce el consumo dinámico.

Finalmente, sería interesante explorar la aplicación de nuestros diseños en otras arquitecturas. Por un lado, las actuales tecnologías de integración han creado también nuevos horizontes en las prestaciones y capacidades de mercado de los sistemas empotrados. Estos han pasado de ser simples dispositivos de control diseñados para realizar una función o un pequeño conjunto de funciones específicas en entornos más o menos reactivos, a ser sistemas complejos con

una funcionalidad comparable en ciertos aspectos a los sistemas de propósito general, pero con unos fuertes requisitos, sobre todo en consumo, que satisfacer.

Los microprocesadores de los nuevos sistemas empotrados de alto rendimiento están basados en arquitecturas RISC de 32 bits y 64-bits, y son más complejos que los procesadores RISC tradicionales (de 8 y 16 bits) en muchos aspectos, por ejemplo son capaces de ejecutar más de una instrucción por ciclo. Es decir, los sistemas empotrados de alto rendimiento presentan características de los procesadores superscalares. Como ejemplos tenemos:

- El microprocesador Freescale Semiconductor MPC7455 que tiene cuatro unidades funcionales de enteros y que puede ejecutar tres instrucciones por ciclo.

- El microprocesador C6000, de Texas Instrument, que tiene ocho unidades funcionales de las cuales dos son multiplicadores, otras dos son unidades dedicadas a transferencia con memoria y las cuatro restantes para el resto de operaciones (sumas, restas, operaciones lógicas, etc...)

Como acabamos de ver, en estos procesadores existen UFs replicadas por lo que una de nuestras propuestas futuras consiste en hacer un estudio del repertorio de instrucciones usando como cargas de trabajo los mibenchs [GRE+01] y aplicar nuestra tecnica en sus unidades funcionales.

Otra arquitectura donde sería interesante aplicar nuestra técnica es en los procesadores SMT (*simultaneous multithreading*). Estos procesadores surgieron para mejorar el rendimiento de los procesadores superscalares. En los procesadores superscalares existen recursos que no se pueden aprovechar comple-

tamente en cada ciclo. Como ejemplo tenemos a las UFs, estas están replicadas para permitir ejecutar varias instrucciones en el mismo ciclo.

Sin embargo, debido al bajo paralelismo a nivel de instrucción (ILP) de algunas hebras (*thread*) de ejecución, hay ciclos en los que no se usan todas las UFs que proporciona el procesador. Los SMT presentan una solución a este problema permitiendo ejecutar múltiples hebras independientes a la vez. De esta manera, si en un mismo ciclo, para una hebra dada, no se puede lanzar una instrucción a cada una de las UFs debido a su bajo ILP, se pueden ejecutar instrucciones de otras hebras. Esto permite aprovechar al máximo los recursos que proporcionan los procesadores.

Dado que el objetivo de los SMT es aprovechar mejor los recursos que proporcionan los procesadores, hay un mayor uso de las UFs en cada ciclo. Esto conlleva un aumento del consumo dinámico en la unidad de ejecución y pueden dar lugar a la aparición de puntos calientes. Nuestra propuesta cosiste en aplicar nuestros diseños en los SMT para lograr reducir el consumo de la unidad de ejecución.

6.3. Publicaciones

Los trabajos realizados durante el desarrollo de esta tesis han sido recogidos en distintas publicaciones científicas.

En el siguiente artículo presentamos una visión general de las ideas básicas en consumo de potencia y analizamos las herramientas de simulación más utilizadas. El objetivo de esto fué encontrar la herramienta más adecuada para poder evaluar las propuesta de este trabajo.

- G. Miñana, O. Garnica, J.I. Hidalgo, J. Lanchares, J.M. Colmenar. Métricas, Metodologías y Herramientas de simulación para evaluar mejoras en Arquitecturas de Bajo Consumo. Revista Enlaces Nº 3. ISSN: 1695-8543. Junio 2005

La adaptación y validación del simulador usado para evaluar el consumo en la unidad de ejecución, se presentó en el artículo:

- G. Miñana, O. Garnica, J.I. Hidalgo, J. Lanchares. Adaptación de un Simulador de Potencia para Unidades Funcionales en Procesadores de Alto Rendimiento. Actas de las XVI jornadas de paralelismo. Páginas: 293-300. ISBN: 84-9732-430-7. septiembre 2005.

Las siguientes publicaciones recogen los resultados, de aplicar diferentes diseños a la unidad de ejecución de enteros para reducir su consumo, explicados en el capítulo 5:

- G. Miñana, O. Garnica, J.I. Hidalgo, J. Lanchares, J.M. Colmenar. Power Reduction of Superscalar Processor Functional Units by Resizing Adder-

Width. Lecture Notes in Computer Science. Volumen 3728, Páginas: 40-48. ISSN: 0302-9743. PATMOS Septiembre 2005.

- 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 Processors. Lecture Notes in Computer Science. Volumen 4148, Páginas: 514-523. ISSN: 0302-9743. PATMOS Septiembre 2006.

- G. Miñana, J.I. Hidalgo, O. Garnica, J. Lanchares, J.M. Colmenar. S. López. A Power-Aware Technique for Functional Units in High Performance Processors. Proceedings of the 9th EUROMICRO Conference on Digital System Desing. Agosto 2006.

- G. Miñana, J.I. Hidalgo, O. Garnica, J. Lanchares, J.M. Colmenar. S. López. Reducing Power of Functional Units in High Performance Processors by checking instructions codes and resizing adders. IEE-proceedings Computers and Digital Techniques. Institution of Engineering and Technology (IET). Volumen 1, Issue 2. Páginas: 113-119. ISSN: 1751-8601. 2007.

De manera colateral, la experiencia en el estudio y simulación de procesadores de propósito general, ha permitido la colaboración en las siguientes publicaciones:

- G. Miñana, J.I. Hidalgo, O. Garnica, D. Gil. Asynchronous circuit implementation using FPGA. IADAT Journal of Advanced Technology on

Automation, Control and Instrumentation, IJAT-aci. Volumen 1, Páginas: 16-18. ISSN: 1885-6403. Septiembre 2005.

- J. M. Colmenar, O. Garnica, J. Lanchares, J. I. Hidalgo, G. Miñana, S. Lopez. Sim-async: an Architectural Simulator for Asynchronous Processor Modelling using Distribution Functions. Proceedings of the 12th International Euro-Par Conference (Euro-Par 2006 Parallel Processing), Lecture Notes in Computer Science, Volumen 4128, Páginas: 495-505. ISSN: 0302-9743. Septiembre 2006.

- J. M. Colmenar, O. Garnica, J. Lanchares, J. I. Hidalgo, G. Miñana, S. Lopez. Comparing the Performance of a 64-bit Fully-Asynchronous Superscalar Processor versus its Synchronous Counterpart. Proceedings of the 9th EUROMICRO Conference on Digital System Desing (DSD 2006). Páginas 423-432. ISBN:0-7695-2609-8. IEEE Computer Society. Septiembre 2006

- S. Lopez, O. Garnica, J. I. Hidalgo, J. Lanchares, J. M. Colmenar, G. Miñana. Study of the Communication Channels in a Globally Asynchrnous Locally Synchronous Simultaneous Multithreading Architecture. Proceedings of the 2th International Conference on Automation, Control and Instrumentation (IADAT 2005). ISBN: 84-933971-8-0. Julio 2005.

Bibliografía

[ACC] Compaq computer corporation. alpha architecture handbook.

[AG03] J. Abella and A. González. Power-aware adaptive issue queue and register file. In *High Performance Computing - HiPC 2003*, volume 2913/2003, pages 34–43, 2003.

[AGG03] J.L. Aragón, J. González, and A. González. Power-aware control speculation through selective throttling. *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 103–112, 8-12 Feb. 2003.

[AGS05] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[Alb00] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *IEEE Journal on Instruction Level Parallelism*, 2000.

[ALE02] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[Asl04] A Aslund. Power estimation of high speed bit-parallel adders. Technical Report LiTH-ISY-EX-3534-2004, 2004.

[AVGDA02] J. M. S. Alcántara, A. C. C. Vieira, F. Gálvez-Durand, and V. Castro Alves. A methodology for dynamic power consumption estimation using vhdl descriptions. In *SBCCI '02: Proceedings of the 15th symposium on Integrated circuits and systems design*, page 149, Washington, DC, USA, 2002. IEEE Computer Society.

[BA97] D. Buger and T. Austin. The simplescalar tool set, version 2.0. In *University of Wisconsin-Madison Computer Sciences Department Technical Report*, 1997.

[BAB96] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[BABC02] A. Buyuktosunoglu, D.H. Albonesi, P. Bose, and P.W. Cook. Tradeoffs in power-efficient issue queue design. *Low Power Electronics and Design, 2002. ISLPED '02. Proceedings of the 2002 International Symposium on*, pages 184–189, 2002.

[BAM98] R.I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 64–69, 10-12 Aug 1998.

[BAS+01] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *GLSVLSI '01: Proceedings of the 11th Great Lakes symposium on VLSI*, pages 73–78, New York, NY, USA, 2001. ACM.

[BCG99] I. Bahar, B. Calder, and D. Grunwald. A comparison of software code reordering and victim buffers. *SIGARCH Comput. Archit. News*, 27(1):51–54, 1999.

[BGS99] R. Bodik, R. Gupta, and M. L. Soffa. Load-reuse analysis: design and evaluation. *SIGPLAN Not.*, 34(5):64–76, 1999.

[BM99a] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 288–293, New York, NY, USA, 1999. ACM.

[BM99b] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 13–22, 9-13 Jan 1999.

[BM00] D. Brooks and M. Martonosi. Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance. *ACM Trans. Comput. Syst.*, 18(2):89–126, 2000.

[BM01] A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 16–21, New York, NY, USA, 2001. ACM.

[BOI96] M. Borah, R. Owens, and M. Irwin. Transistor sizing for low power cmos circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(6):665–671, 1996.

[Bor99] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, Jul-Aug 1999.

[BPN03] K. Buyuksahin, P. Patra, and F.Ñajm. Estima: An architectural level power estimator for multi ported pipelined register files. In *ISLPED*, 2003.

[BS00] J.A. Butts and G.S. Sohi. A static power model for architects. *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 191–201, 2000.

[BSB+01] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. An adaptive issue queue for reduced power at high performance. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 25–39, London, UK, 2001. Springer-Verlag.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, 2000.

[CDD+04] R. Chau, S. Datta, M. Doczy, B. Doyle, J. Kavalieros, and M. Metz. High-k / metal-gate stack and its mosfet characteristics. *IEEE Electron Device Letters*, 25(6):408–410, 2004.

[CGE96] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. *High-Performance Computer Architecture, 1996. Proceedings. Second International Symposium on*, pages 244–253, 3-7 Feb 1996.

[CGG04] P. Chaparro, J. González, and A. González. Thermal-aware clustered microarchitectures. *Computer Design: VLSI in Computers*

and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on, pages 48–53, 11-13 Oct. 2004.

[CIB01] R. Y. Chen, M. J. Irwin, and R. S. Bajwa. Architecture-level power estimation and design experiments. *ACM Trans. Des. Autom. Electron. Syst.*, 6(1):50–66, 2001.

[CJC00] J. Choi, J. Jeon, and K. Choi. Power minimization of functional units partially guarded computation. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 131–136, New York, NY, USA, 2000. ACM.

[CL99] G. Cai and C. H. Lim. Architectural level power/performance optimization and dynamic power estimation, 1999.

[CLSL02] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti. Precise and accurate processor simulation. In *In Proc. of Fifth Workshop on Computer*, pages 13–22, 2002.

[CRP+05] D. Chaver, M. A. Rojas, L. Piñuel, M. Prieto, F. Tirado, and M. C. Huang. Energy-aware fetch mechanism: trace cache and btb customization. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 42–47, New York, NY, USA, 2005. ACM.

[CSBP94] Z . Chen, J. Shott, J. Burr, and J. D. Plummer. Cmos technology scaling for low voltage low power applications. In *in Symp. Low Power Electr*, pages 56–57, Oct 1994.

[CSW02] O. T.-C. Chen, R. R. B. Sheen, and S. Wang. A low-power adder operating on effective dynamic data ranges. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(4):435–453, 2002.

[CWS05] M. Co, D. A. B. Weikle, and K. Skadron. A break-even formulation for evaluating branch predictor energy efficiency. In *Workshop on Complexity-Effective Design conjuntamente con el Annual ACM/IEEE International Symposium on Computer Architecture*, 2005.

[CWS06] M. Co, D. A. B. Weikle, and K. Skadron. Evaluating trace cache energy efficiency. *ACM Trans. Archit. Code Optim.*, 3(4):450–476, 2006.

[CYL99] S. Cho, P. Yew, and G. Lee. Decoupling local variable accesses in a wide-issue superscalar processor. *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 100–110, 1999.

[DBKA01] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a validated execution driven alpha 21264 simulator, 2001.

[DKA+02] S. Dropsho, V. Kursun, D.H. Albonesi, S. Dwarkadas, and E.G. Friedman. Managing static leakage energy in microprocessor functional units. *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 321–332, 2002.

[DRJW99] S. Dey, A. Raghunathan, N.K. Jha, and K. Wakabayashi. Controller-based power management for control-flow intensive designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(10):1496–1508, Oct 1999.

[DSN] A. David, T. Shyamkumar, and J.Ñorman. Cacti v.4. HPL-2006-86 20060606.

[EEAS02] W. El-Essawy, D.H. Albonesi, and B. Sinharoy. A microarchitectural-level step-power analysis tool. In *International Symposium on Low Power Electronics and Design, ISLPED*, pages 263–266, 2002.

[EKRZ04] D. Edenfel, A. B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *0018-9162/04/ IEEE Computer*, 2004.

[FG01] D. Folegnani and A. González. Energy-effective issue logic. *SIGARCH Comput. Archit. News*, 29(2):230–239, 2001.

[FKM+02] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.

[Fre97] J. Frenkil. Tools and methodologies for low power design. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 76–81, New York, NY, USA, 1997. ACM.

[GBJ98] M.K. Gowan, L.L. Biro, and D.B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. *Design Automation Conference, 1998. Proceedings*, pages 726–731, 15-19 Jun 1998.

[GG01] S. Ghiasi and D. Grunwald. A comparison of two architectural power models. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 137–152, London, UK, 2001. Springer-Verlag.

[GH96] R. González and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996.

[Gho00] K. Ghose. Reducing energy requirements for instruction issue and dispatch in superscalar microprocessors. *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pages 231–233, 2000.

[GM94] B. M. Gordon and T. H.-Y. Meng. A low power subband video decoder architecture. In *in Int. Conf Acoustics, Speech and Signal Processing*, pages 409–412, Abril 1994.

[GM03] K. R. Gandhi and N. R. Mahapatra. A detailed study of hardware techniques that dynamically exploit frequent operands to reduce power consumption in integer function units. In *Proceedings of the 21th International Conference on Computer Design (ICCD 03)*, 2003.

[GM05] K. R. Gandhi and N. R. Mahapatra. Dynamically exploiting frequent operand values for energy efficiency in integer functional units. In *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID 05)*, 2005.

[GRE+01] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Deciembre 2001.

[GS04] J. González and K. Skadron. Power-aware design for high-performance processors. In *10th International Symposium on High-Performance Computer Architecture (HPCA-10)*, 2004.

[HBS+04] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37, New York, NY, USA, 2004. ACM.

[HIVK02] J. Hu, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Selective trace cache: A low power and high performance fetch mechanism. In *Technical Report CSE-02-016.*, 2002.

[HK03] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48, New York, NY, USA, 2003. ACM.

[HK05] J. Hom and U. Kremer. Inter-program optimizations for conserving disk energy. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 335–338, New York, NY, USA, 2005. ACM.

[HKH01] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in mircoprocessors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 275–278, New York, NY, USA, 2001. ACM.

[HKY+95] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. Sh3: high code density, low power. *Micro, IEEE*, 15(6):11–19, Dec 1995.

[HLR05] H. Homayoun, K.F. Li, and S. Rafatirad. Functional units power gating in smt processors. *Communications, Computers and signal Processing, 2005. PACRIM. 2005 IEEE Pacific Rim Conference on*, pages 125–128, 24-26 Aug. 2005.

[Hor96] M.Ñ. Horestein. *Microelectronic Circuits and Devices.* Prentice Hall, 1996.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers, 4ª Edición, 2007.

[HRBM03] S. Haga, N. Reeves, R. Barua, and D. Marculescu. Dynamic functional unit assignment for low power. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 1052–1057, 2003.

[HRBM05] S. Haga, N. Reeves, R. Barua, and D. Marculescu. Dynamic functional unit assignment for low power. *Teh Journal of Supercomputing, Editor Springer Netherlands*, 31(1):47–62, 2005.

[HRT02] M. Huang, J. Renau, and J. Torrellas. Energy-efficient hybrid wakeup logic. *Low Power Electronics and Design, 2002. ISLPED '02. Proceedings of the 2002 International Symposium on*, pages 196–201, 2002.

[HRYT01] M. Huang, J. Renau, S.M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. *Low Power Electronics and Design, International Symposium on, 2001.*, pages 10–15, 2001.

[HSS+04] W. Huang, M.R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature-aware design. *Design Automation Conference, 2004. Proceedings. 41st*, pages 878–883, 2004.

[HVIK03] J.S. Hu, N. Vijaykrishnan, M.J. Irwin, and M. Kandemir. Using dynamic branch behavior for power-efficient instruction fetch. *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*, pages 127–132, 20-21 Feb. 2003.

[HVIK07] J. Hu, N. Vijaykrishnan, M.J. Irwin, and M. Kandemir. Optimising power efficiency in trace cache fetch unit. *Computers and Digital Techniques, IET*, 1(4):334–348, July 2007.

[HVKI02] J.S. Hu, N. Vijaykrishnan, A. Kandemir, and A.J. Irwin. Power-efficient trace caches. *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 1091–, 2002.

[IBM] Ibm powerpc: http://www.chips.ibm.com/products/powerpc.

[IIM99] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pages 273–275, 1999.

[Int] Intel xscale: http://www.intel.com/design/intelxscale/.

[INT04] Intel research silicon moore's law. http://www.intel.com/research/silicon/mooreslaw.htm, 2004.

[ITR06] International Technology Roadmap for Semiconductors ITRS 2006 Update, http://public.itrs.net/, 2006.

[JOAG05] T. M. Jones, M. F.P. O'Boyle, J. Abella, and A. González. Software directed issue queue power reduction. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 144–153, Washington, DC, USA, 2005. IEEE Computer Society.

[KAB+03] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V.Ñarayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Diciembre 2003.

[KAMG02] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. pages 317–337, 2002.

[KC97] D. Kim and K. Choi. Power-conscious high level synthesis using loop folding. *Design Automation Conference, 1997. Proceedings of the 34th*, pages 441–445, Jun 1997.

[KEPG03] G. Kucuk, O. Ergin, D. Ponomarev, and K. Ghose. Energy efficient register renaming. In *PATMOS*, volume 2799/2003, pages 219–228, 2003.

[Kes99] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[KFBM02] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 219–230, 2002.

[KFBM04] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Single-vdd and single-vt super-drowsy techniques for low-leakage high-performance instruction caches. *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 54–57, 2004.

[KGMS00] J. Kin, M. Gupta, and W.H. Mangione-Smith. Filtering memory references to increase energy efficiency. *Computers, IEEE Transactions on*, 49(1):1–15, Jan 2000.

[KGPK01] G. Kucuk, K. Ghose, D.V. Ponomarev, and P.M. Kogge. Energy-efficient instruction dispatch buffer design for superscalar processors. *Low Power Electronics and Design, International Symposium on, 2001.*, pages 237–242, 2001.

[KHM01] S. Kaxiras, Zhigang Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 240–251, 2001.

[Kim07] S. Kim. Reducing alu and register file energy by dynamic zero detection. *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pages 365–371, April 2007.

[KM06] F. Kashfi and N. Masoumi. Optimization of speed and power in a 16-bit carry skip adder in 70nm technology. *Circuits and Systems, 2006. MWSCAS '06. 49th IEEE International Midwest Symposium on*, 1:28–31, Aug. 2006.

[KMMS97] J. Kin, G. Munish, and W.H. Mangione-Smith. The filter cache: an energy efficient memory structure. *Microarchitecture, 1997. Proceedings. Thirtieth Annual IEEE/ACM International Symposium on*, pages 184–193, 1-3 Dec 1997.

[KN05] M. Kondo and H.Ñakamura. Dynamic processor throttling for power efficient computations. In *Lecture Notes in Computer Science*, pages 120–134, Tokio, Japon, 2005. Springer Berlin / Heidelberg.

[Kre05a] U. Kremer. Low power/energy compiler optimizations. In *Low-Power Electronics Design (Editor: Christian Piguet), CRC Press*, 2005.

[Kre05b] K. Krewell. Yonah does dual-core right, Marzo 2005.

[KSB02] T. Karkhanis, J.E. Smith, and P. Bose. Saving energy with just in time instruction delivery. *Low Power Electronics and Design, 2002. ISLPED '02. Proceedings of the 2002 International Symposium on*, pages 178–183, 2002.

[KVIJ03] S. Kim, N. Vijaykrishnan, M.J. Irwin, and L.K. John. On load latency in low-power caches. *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pages 258–261, 25-27 Aug. 2003.

[KVK+01] S. Kim, N. Vijaykrishnan, M. Kandemir, A. Sivasubramaniam, M.J. Irwin, and E. Geethanjali. Power-aware partitioned cache architectures. *Low Power Electronics and Design, International Symposium on, 2001.*, pages 64–67, 2001.

[LBC+03] Hai Li, S. Bhunia, Y. Chen, T.N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 113–122, 8-12 Feb. 2003.

[LFDD03] H-H. S. Lee, J. B. Fryman, A. U. Diril, and Y. S. Dhillon. The elusive metric for low-power architecture research. In *In the Workshop on Complexity-Effective Design in conjunction with ISCA*, 2003.

[LSNT01] H.H.-S. Lee, M. Smelyanskiy, C.J. Newburn, and G.S. Tyson. Stack value file: custom microarchitecture for the stack. *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 5–14, 2001.

[MAA+04] K. Mistry, M. Armstrong, C. Auth, S. Cea, T. Coan, T. Ghani, T. Hoffmann, A. Murthy, J. Sandford, R. Shaheed, K. Zawadzki, K. Zhang, S. Thompson, and M. Bohr. Delaying forever: Uniaxial strained silicon transistors in a 90nm cmos technology. *VLSI Technology, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 50–51, 15-17 June 2004.

[MAG+03] C. Molina, C. Aliagas, M. García, A. Gonzàlez, and J. Tubella. Non redundant data cache. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 274–277, New York, NY, USA, 2003. ACM.

[Mar04] D. Marculescu. Application adaptive energy efficient clustered architectures. *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 344–349, 2004.

[Mat03] S. K. Mathew. A 4 ghz 130 nm address generation unit with 32-bit sparse-tree adder core. *IEEE J. Solid-State Circuits*, 38(5):689–695, 2003.

[MC95] E. Musoll and J. Cortadella. High-level synthesis techniques for reducing the activity of functional units. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 99–104, New York, NY, USA, 1995. ACM.

[MH03] L. Piñuel M. Prieto F. Tirado M. Huang, D. Chaver. Customizing the branch predictor to reduce complexity and energy consumption. *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, 23(5):12–25, Septiembre 2003.

[MKG98] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, pages 132–141, 27 Jun-1 Jul 1998.

[MLOJ98] E. Morancho, J.M. Llaberia, A. Olive, and M. Jimenez. One-cycle zero-offset loads. In *Proceedings of the Second IASTED International Conference European Parallel and Distributed Systems*, pages 87–93, 1998.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[MRMS03] G. Memik, G. Reinman, and W.H. Mangione-Smith. Just say no: benefits of early cache miss determination. *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 307–316, 8-12 Feb. 2003.

[Nar05] S. G. Narendra. Challenges and design choices in nanoscale cmos. *J. Emerg. Technol. Comput. Syst.*, 1(1):7–49, 2005.

[NMN07] V.Ñavarro, J. A. Montiel, and S.Ñooshabadi. High performance low power cmos dynamic logic for arithmetic circuits. *Microelectron. J.*, 38(4-5):482–488, 2007.

[NVN03] D.Ñicolaescu, A. Veidenbaum, and A.Ñicolau. Reducing data cache energy consumption via cached load/store queue. In *ISLPED*

'03: Proceedings of the 2003 international symposium on Low power electronics and design, pages 252–257, New York, NY, USA, 2003. ACM.

[PAV+01] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 54–65, 1-5 Dec. 2001.

[PHC03] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 244–255, 27 Sept.-1 Oct. 2003.

[PJH07] W. Pei, W.-B. Jone, and Y. Hu. Fault modeling and detection for drowsy sram caches. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(6):1084–1100, June 2007.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25(2):206–218, 1997.

[PKG01] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 90–101, Washington, DC, USA, 2001. IEEE Computer Society.

[PKG02] D. Ponomarev, G. Kucuk, and K. Ghose. Energy-efficient design of the reorder buffer. In *PATMOS '02: Proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*, pages 289–299, London, UK, 2002. Springer-Verlag.

[PSp] PSpice. http://www.cadence.com/products/orcad/pspice-simulation/pages/default.aspx.

[PSV05] M.D. Powell, E. Schuchman, and T.N. Vijaykumar. Balancing resource utilization to mitigate power density in processor pipelines. *Microarchitecture, 2005. MICRO-38. Proceedings. 38th*

Annual IEEE/ACM International Symposium on, pages 11 pp.–, 12-16 Nov. 2005.

[PSZ+02] D. Parikh, K. Skadron, Yan Zhang, M. Barcella, and M.R. Stan. Power issues related to branch prediction. *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 233–244, 2-6 Feb. 2002.

[PYF+00] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T.Ñ. Vijaykumar. Gated Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 90–95, 2000.

[RBS96] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[RDJW96] A. Raghunathan, S. Dey, N.K. Jha, and K. Wakabayashi. Controller re-specification to minimize switching activity in controller/data path circuits. *Low Power Electronics and Design, 1996., International Symposium on*, pages 301–304, Aug 1996.

[RJ94] A. Raghunathan and N.K. Jha. Behavioral synthesis for low power. *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pages 318–322, 10-12 Oct 1994.

[Roy98] K. Roy. Leakage power reduction in low-voltage cmos designs. *Electronics, Circuits and Systems, 1998 IEEE International Conference on*, 2:167–173 vol.2, 1998.

[RPOG02] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *Computational Complexity*, pages 261–275, 2002.

[RSSS06] P. Rajamani, J.P. Shah, V. Sankaranarayanan, and R. Sangireddy. High performance and alleviated hot-spot problem in processor frontend with enhanced instruction fetch bandwidth utilization. *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 8 pp.–, 10-12 April 2006.

[RV07] R. Rao and S. Vrudhula. Performance optimal processor throttling under thermal constraints. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 257–266, New York, NY, USA, 2007. ACM.

[SAD+02] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 356–367, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[SC97] D. Shin and K. Choi. Low power high level synthesis by increasing data correlation. *Low Power Electronics and Design, 1997. Proceedings., 1997 International Symposium on*, pages 62–67, Aug 1997.

[SMB+02] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40, 2-6 Feb. 2002.

[SMH01] P.S. Stanley-Marbell and M.S. Hsiao. Fast, flexible, cycle-accurate energy estimation. *Low Power Electronics and Design, International Symposium on, 2001.*, pages 141–146, 2001.

[SPE] Standard performance evaluation corporation. spec cpu2000 benchmarks. http://www.specbench.org/osg/cpu2000.

[SS04] K. Sankaranarayanan and K. Skadron. Profile-based adaptation for cache decay. *ACM Trans. Archit. Code Optim.*, 1(3):305–322, 2004.

[SSA06] C. Senthilpari, A.K. Singh, and A. Arokiasamy. Statistical analysis of power delay estimation in adder circuit using non-clocked pass gate families. *Electrical and Computer Engineering, 2006. ICECE '06. International Conference on*, pages 509–513, Dec. 2006.

[SSS+04] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture:

Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[STT01] J.S. Seng, E.S. Tune, and D.M. Tullsen. Reducing power with dynamic critical path information. *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 114–123, 1-5 Dec. 2001.

[Syp] Sypnosys. Powermill user guide, release 5.4. Document Order Number: 32647-014 HB.

[TM01] E. Talpes and D. Marculescu. Power reduction through work reuse [superscalar processor microarchitecture]. *Low Power Electronics and Design, International Symposium on, 2001.*, pages 340–345, 2001.

[TPB98] S. Thompson, P. Packan, and M. Bohr. Mos scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, 2(3):p.Q3, 1998.

[Tra] Transmeta crusoe: http://www.transmeta.com/technology /index.html.

[TSC07] S. Talli, R. Srinivasan, and J. Cook. Compiler-directed functional unit shutdown for microarchitecture power optimization. *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pages 372–379, 11-13 April 2007.

[Vaz03] A. Maldonado Vazquez. Power-performance tradeoffs in digital arithmetic circuits. In *Summer Undergraduate Program in Engineering Research at Berkeley SUPERB, University of Puerto Rico-Mayaguez. Electrical Engineering*, 2003.

[VMR06] T. Vigneswaran, B. Mukundhan, and P.S. Reddy. A novel low power, high speed 14 transistor cmos full adder cell with 50 % improvement in threshold loss problem. *Procedings of World Academy of Science Engineering and Technology (PWASET)*, 13:81–85, Mayo 2006.

[VZA00] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 214–220, 2000.

[WCJ+98] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De. Design and optimization of low voltage high performance dual threshold cmos circuits. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 489–494, New York, NY, USA, 1998. ACM.

[WCR+99] L. Wei, Z. Chen, K. Roy, M.C. Johnson, Y. Ye, and V.K. De. Design and optimization of dual-threshold circuits for low-voltage low-power applications. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):16–24, Mar 1999.

[WE93] N. Weste and K. Eshraghian. Principles of cmos vlsi design, 1993.

[WH05] N. H. E. Weste and D. Harris. *CMOS VLSI DESING, A Cicuits and Systems Perspective.* Pearson Addison-Wesley, 3ª Edición, 2005.

[WJ96] S.J.E. Wilton and N.P. Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996.

[WM04] K. Wilcox and S. Manne. Alpha processor: A history of power issues and a look to the future, 2004.

[WV98] Q. Wang and S. B. K. Vrudhula. Static power optimization of deep submicron cmos circuits for dual vt technology. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 490–496, New York, NY, USA, 1998. ACM.

[XMM03] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 49–62, New York, NY, USA, 2003. ACM.

[YAE06] A. Youssef, M. Anis, and M. Elmasry. Dynamic standby prediction for leakage tolerant microprocessor functional units. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 371–384, Washington, DC, USA, 2006. IEEE Computer Society.

[YG01] J. Yang and R. Gupta. Energy-efficient load and store reuse. *Low Power Electronics and Design, International Symposium on, 2001.*, pages 72–75, 2001.

[YPF+01] S. Yang, M.D. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 147–157, 2001.

[YPFV02] S. H. Yang, M.D. Powell, B. Falsafi, and T.N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 151–161, 2-6 Feb. 2002.

[YT06] Pan Y and Tay Teng T. Functional unit selection in superscalar microprocessors for low power. *TENCON 2006. 2006 IEEE Region 10 Conference*, pages 1–5, Nov. 2006.

[ZF97] R. Zimmermann and W. Fichtner. Low-power logic styles: Cmos versus passtransistor logic. *IEEE Journal of solid-State Circuits*, 32(7):1079–1090, 1997.

Índice de figuras

1.1. Frecuencia del reloj de los microprocesadores de INTEL. Fuente:[WH05]. 4
1.2. Tendencia del consumo dinámico y estático basados en *the International Roadmap for Semiconductors*[KAB+03]. 5
1.3. Tendencia del porcentaje entre el consumo dinámico y estático en función de la tecnología. Fuente: Skadron et al. Universidad de Virginia. 6

3.1. Rutas de datos alternativas en un procesador de 3-vias [HRBM05]. 67
3.2. Implementación de la técnica propuesta en [HRBM05]. 68
3.3. Primera implementación de la arquitectura de *clock gating* propuesta por Brooks y Martonosi en [BM00]. 70
3.4. Segunda implementación de la arquitectura de *clock gating* propuesta por Brooks y Martonosi en [BM00]. 71
3.5. Implementación de la técnica propuesta en [CJC00]. 72
3.6. Implementación del detector dinámico de registros con valor cero para evitar cómputos en el sumador [Kim07]. 74
3.7. Control dinámico de las corrientes de fuga [YAE06]. 75

4.1. Modelo de la unidad de ejecución que se ha implementado. . . 90
4.2. Modelo de la unidad de ejecución de la arquitectura *Alpha* 21264. 96
4.3. Porcentaje del consumo de la unidad de ejecución respecto al consumo total del procesador obtenido con *Sim-Alpha* y *FU-Wattch*. 100
4.4. Porcentaje del consumo de la unidad de ejecución respecto al consumo total del procesador obtenido con *Wattch* y con *FU-Wattch*. 103

5.1. Formato de las instrucciones de operación del repertorio de la arquitectura *Alpha* 21264 109
5.2. Formato para las instrucciones de memoria del repertorio de la arquitectura *Alpha* 21264 110

5.3. Formato para las instrucciones de salto del repertorio de la arquitectura *Alpha* 21264 110
5.4. Formato para las instrucciones especiales del repertorio de la arquitectura *Alpha* 21264 111
5.5. Formato para las instrucciones de salto del repertorio de la arquitectura *Alpha* 21264, donde se representa la operación que se realiza para calcular la dirección efectiva. 112
5.6. Formato para las instrucciones de memoria del repertorio de la arquitectura *Alpha* 21264, donde se representa la operación que se realiza para calcular la dirección efectiva. 113
5.7. Porcentaje de instrucciones de cada tipo que hay en cada *Benchmarks* 117
5.8. Casos en los que las instrucciones tienen que usar el sumador de 64 bits 120
5.9. Porcentaje de instrucciones que no pueden ejecutarse ni en el sumador de 24-bits (Sum24) ni en el de 32-bits (Sum32) . . . 122
5.10. Porcentaje de instrucciones que podrían ejecutarse en cada sumador. 123
5.11. Ciclos en los que se necesita más de un sumador de 64-bits y más de una sumador de 32-bits. 125
5.12. Posible implementación del algoritmo 1 131
5.13. Instrucciones que se ejecutan en un sumador de 24-bits 141
5.14. Consumo estático en los sumadores 145
5.15. Consumo dinámico en los sumadores cuando $cp_{S32} = 0{,}5$ . . . 147
5.16. Consumo dinámico en los sumadores cuando $cp_{S32} = 0{,}33$ . . . 148
5.17. Consumo total en los sumadores cuando $cp_{S32} = 0{,}5$ 151
5.18. Consumo total en los sumadores cuando $cp_{S32} = 0{,}33$ 152
5.19. IPC/(consumo en los sumadores) cuando $cp_{S32} = 0{,}5$ 155
5.20. IPC/(consumo en los sumadores) cuando $cp_{S32} = 0{,}33$ 156
5.21. Producto Energía Retardo Completo aplicado en los sumadores para Diseño1 158
5.22. Producto Energía Retardo Completo aplicado en los sumadores para Diseño2 158
5.23. Producto Energía Retardo Completo aplicado en los sumadores para Diseño3 159
5.24. Producto Energía Retardo Completo aplicado en los sumadores para Diseño4 159
5.25. Producto Energía Retardo Completo aplicado en los sumadores para Diseño5 160
5.26. Producto Energía Retardo Completo aplicado en la unidad de ejecución para Diseño1 161

5.27. Producto Energía Retardo Completo aplicado en la unidad de ejecución para Diseño2 161
5.28. Producto Energía Retardo Completo aplicado en la unidad de ejecución para Diseño3 162
5.29. Producto Energía Retardo Completo aplicado en la unidad de ejecución para Diseño4 162
5.30. Producto Energía Retardo Completo aplicado en la unidad de ejecución para Diseño5 163
5.31. Ahorro de potencia estática, dinámica y total en los sumadores 165

Índice de tablas

1.1. Porcentaje de potencia que consume cada parte del procesador. 8

2.1. Clasificación de los simuladores. 21

4.1. Unidades funcionales y sus instrucciones asociadas. 91
4.2. *Benchmarks* del conjunto SPEC CPU2000 que se han elegido para las simulaciones. 97
4.3. Configuración del núcleo del procesador simulado. 98
4.4. Configuración del predictor del procesador simulado. 98
4.5. Jerarquía de la memoria del procesador simulado. 99
4.6. Consumo estimado de cada unidad funcional a 5V, 733MHz y 180nm [BTM00]. 99
4.7. Consumo estimado de cada unidad funcional a 5V, 733MHz y 180nm [BTM00]. 102

5.1. Instrucciones del procesador *Alpha* que necesitan sumador. . . 115
5.2. Configuración del núcleo del procesador simulado. 136
5.3. Configuración del predictor del procesador simulado. 136
5.4. Jerarquía de la memoria del procesador simulado. 137
5.5. Valores de consumo de las distintas UFs usadas, para una tecnología de 733MHz y 0.18μm 137
5.6. Porcentaje de instrucciones, promediados sobre todos las *Benchmarks*, que se ejecutan en cada uno de los tipos de sumadores para cada uno de los diseños. 143
5.7. Ahorro de potencia, promediado sobre todos los *Benchmarks*, tanto en los sumadores como en toda la unidad de ejecución, para cada uno de los diseños. 153
5.8. IPC para cada diseño, promediado sobre todos los *Benchmarks* 154
5.9. Diferentes diseños de la unidad de ejecución 164

6.1. Diferentes diseños de la unidad de ejecución 174