



**INSTITUTO POLITÉCNICO NACIONAL**

---

**CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN**

**DETECCIÓN, PREDICCIÓN Y EJECUCIÓN DE  
OPERACIONES REDUNDANTES PARA INCREMENTAR  
EL RENDIMIENTO DE UN MICROPROCESADOR**

**TESIS**

QUE PARA OBTENER EL GRADO DE  
MAESTRO EN CIENCIAS  
EN INGENIERÍA DE CÓMPUTO CON  
ESPECIALIDAD EN SISTEMAS DIGITALES

PRESENTA:

**ING. ALEJANDRO VILLAR BRIONES**

DIRECTOR: DR. LUIS ALFONSO VILLA VARGAS

CODIRECTOR: DR. OSCAR CAMACHO NIETO

MÉXICO, D.F.

OCTUBRE DEL 2003

# Dedicatoria

A mis Padres quienes lo han dado todo por que alcance mi realización como persona, profesionista e hijo; siempre sin esperar nada a cambio por tan valioso legado y apoyo. A ellos todo mi amor y cariño.

A mis hermanos Claudia, Fabiola y Armando, por el apoyo incondicional que siempre me han dado y estar conmigo en todo momento de mi vida.

A mis abuelos maternos Consuelo y Elías, de quienes herede gran parte de mi personalidad.

A mis abuelos paternos Angélica y Luis, por todo su cariño.

A mis tíos Guadalupe, Lucina, Lourdes, Elena, Patricia, Felipe y José, de quienes he aprendido el valor del cariño de la familia, el duro esfuerzo, y su amor incondicional.

A mis tíos Luis, Erasmo, Nancy, Salvador, Emmanuel, Rosario, Thomas, Ismael Alejandro y Juan Carlos, por su comprensión e impulso que me han dado.

A todos mis amigos, con quienes he compartido las alegrías y tristezas de mi vida.

A todos ellos brindo este logro mas en mi vida.

# Agradecimientos

Al M. en C. Enrique López Guzmán por los consejos y la orientación que me dio durante la carrera.

## **Al Centro de Investigación en Computación (CIC)**

Al Dr. Luis Alfonso Villa Vargas, por los conocimientos transmitidos y recomendaciones para la realización de este trabajo.

Al Dr. Oscar Camacho Nieto, por su dirección y apoyo brindado para completar esta tesis.

A todos los profesores y miembros del jurado por el tiempo y esfuerzo que dedican a nuestra superación.

A todos mis compañeros por su valiosa amistad y consejos.

## **Al Instituto Mexicano del Petróleo (IMP)**

En especial al Programa de Matemáticas Aplicadas y Computación (PMAyC) por la por el apoyo que me brindó para la realización de este trabajo, el equipo de computo utilizado y el uso de sus instalaciones.

A la competencia de Tecnologías de Información por el apoyo brindado y al Instituto por la beca concedida para concluir esta tesis.

# Índice

Índice .....	i
Lista de Figuras .....	iii
Lista de Tablas .....	vi
Resumen .....	vii
<i>Abstract</i> .....	vii
Introducción .....	ix
Antecedentes .....	xiii
Descripción y definición del problema .....	xiv
Objetivos .....	xv
Justificación .....	xvi
<b>Capítulo 1.</b> Microarquitectura de procesadores superescalares .....	1
1.1 Introducción .....	1
1.2 Microarquitectura típica de un procesador superescalar .....	1
1.2.1 Búsqueda de instrucciones y predicción de saltos .....	1
1.2.2 Decodificación, renombramiento y despacho de instrucciones .....	3
1.2.2.1 Tipos de dependencias .....	3
1.2.2.2 Despacho de instrucciones .....	4
1.2.2.3 Renombramiento de registros .....	5
1.2.3 Lanzamiento de instrucciones y ejecución en paralelo .....	9
1.2.3.1 Estructura básica de la ventana de instrucciones o estación de reserva .....	11
1.2.4 Operaciones de manejo de memoria .....	12
1.2.4.1 Memorias Caché .....	15
1.2.4.2 Localización de datos .....	17
1.2.4.3 Políticas de reemplazo .....	17
1.2.4.4 Tipos de fallos .....	18
1.2.5 Retiro de instrucciones .....	19
1.3 El papel del software .....	20
1.4 Operaciones repetitivas en un procesador superescalar .....	21
1.5 Análisis y modelado matemático del tiempo de ejecución .....	23
1.5.1 Aproximación analítica .....	25
<b>Capítulo 2.</b> Metodología de evaluación .....	31
2.1 Resumen .....	31
2.2 Herramientas de simulación .....	31
2.2.1 Resumen de simuladores de arquitecturas de microprocesadores .....	31
2.2.2 Selección de la Herramienta de simulación .....	34
2.2.3 Modificación de la herramienta de simulación .....	39
2.3 Cargas de trabajo ( <i>Benchmark</i> ) .....	39
2.4 Plataforma de cómputo .....	40

2.5 Opciones de configuración para simulación de modelos .....	40
<b>Capítulo 3. Ejecución de instrucciones fuera de orden .....</b>	<b>43</b>
3.1 Introducción .....	43
3.2 Comportamiento de la ventana de instrucciones .....	43
3.2.1 Análisis de dependencias por datos en la ventana de instrucciones .....	45
3.2.2 Políticas de comparación de entradas dentro de la ventana de instrucciones .....	47
3.2.3 Consumo de energía .....	53
3.3. Antecedentes .....	54
3.4 Ventana de instrucciones dividida en bloques .....	56
3.4.1 Submodelos de la ventana de instrucciones dividida en bloques .....	65
3.5 Ventana de instrucciones con detección de operandos por bit de identificación .....	75
3.6 Comparación y evaluación de modelos propuestos .....	81
<b>Capítulo 4. Sistema de Memoria .....</b>	<b>83</b>
4.1 Introducción .....	83
4.2 Antecedentes .....	83
4.2.1 Predicción en memorias caché .....	83
4.2.2 Mecanismos para mejorar el acceso a la memoria caché de datos .....	85
4.3 Manejo de operaciones redundantes en la memoria caché .....	87
4.4 Detección de comportamiento .....	89
4.5 Caché de acceso predictivo a arreglos ( <i>PAAC</i> ) .....	90
4.5.1 Tabla de referencia de predicción .....	92
4.5.2 Mecanismo de predicción en base a <i>stride</i> .....	95
4.5.3 Metodología de Evaluación .....	97
4.5.4 Resultados .....	98
<b>Capítulo 5. Conclusiones y trabajos futuros .....</b>	<b>100</b>
5.1 Conclusión .....	100
5.2 Comentarios .....	100
5.3 Trabajos futuros .....	101
<b>Referencias .....</b>	<b>103</b>
Introducción .....	103
Capítulo 1 .....	104
Capítulo 2 .....	106
Capítulo 3 .....	108
Capítulo 4 .....	109
Capítulo 5 .....	111
<b>Apéndice A. Símbolos .....</b>	<b>112</b>

# Lista de Figuras

Figura 1.1 Etapas de procesamiento de un procesador súper escalar .....	2
Figura 1.2. Renombramiento de registro. Archivo de registro de renombramiento .....	7
Figura 1.3. Renombramiento de registro. <i>Reorder Buffer</i> .....	8
Figura 1.4 Método de enfilamiento .....	10
Figura 1.5 Ventana de instrucciones .....	12
Figura 1.6 Jerarquía de Memoria. ....	14
Figura 1.7 Direccionamiento de la memoria caché .....	16
Figura 1.8 Campos de la dirección de memoria .....	17
Figura 1.9 Porcentaje de instrucciones promedio en una carga de trabajo típica .....	22
Figura 1.10 Tiempo de ejecución .....	23
Figura 2.1 Diagrama esquemático del simulador “ <i>sim-outorder</i> ” del Simplescalar .....	35
Figura. 2.2 Bloques de registros enteros y de punto flotante de la arquitectura simulada por “ <i>sim-outorder</i> ” .....	37
Figura 2.3 Módulos que conforman el simulador “ <i>Sim-Outorder</i> ” .....	38
Figura 2.4 Funciones que emulan las etapas del procesador en el simulador “ <i>Sim-Outorder</i> ” .....	38
Figura 3.1 Instrucciones por ciclo, SPECINT .....	44
Figura 3.2 Instrucciones por ciclo, SPECFP .....	44
Figura 3.3 Porcentaje de operandos fuente para enteros .....	45
Figura 3.4 Porcentaje de operandos fuente para flotantes .....	46
Figura 3.5 Búsqueda de instrucciones listas para ejecutarse dentro de la ventana de instrucciones .....	47
Figura 3.6 Ventana de instrucciones de 64 entradas .....	49
Figura 3.7 Ventana de instrucciones de 32 entradas .....	49
Figura 3.8 Porcentaje de comparaciones efectivas y comparaciones con un operando válido dentro de la ventana de instrucciones (SPECINT) .....	50
Figura 3.9 Porcentaje de comparaciones efectivas y comparaciones con un operando valido dentro de la IW (SPECFP) .....	50
Figura 3.10 Partición imaginaria de la ventana de instrucciones .....	51
Figura 3.11 Porcentaje de receptores (operandos) por cuadrante en la ventana (SPECINT) .....	52
Figura 3.12 Porcentaje de receptores (operandos) por cuadrante en la ventana (SPECFP) .....	52
Fig. 3.13 Potencia consumida en ventanas de instrucciones con diferente número de entradas (Watts) .....	52
Figura 3.14 Esquema <i>Direct Tag Search</i> (DTS) .....	55
Figura 3.15 Ventana de instrucciones dividida en bloques .....	57
Figura 3.16 Tabla de Mapeo de Bloques (TMB) .....	58
Figura 3.17 Diagrama de flujo de operación de la Unidad de Actualización de la TMB (UAT) .....	60
Figura 3.18 Arquitectura de una ventana de instrucciones dividida .....	61

Figura 3.19 Número de comparaciones de esquema normal vs por bloques (política de comparación activa) ventana de 64 entradas, dividida en 4 bloques .....	62
Figura 3.20 Número de comparaciones esquema normal vs por bloques (política de comparación activa) ventana de 32 entradas, dividida en 4 bloques .....	62
Figura 3.21 Número de comparaciones ideales vs por bloques (política de comparación arbitraria) ventana de 64 entradas, dividida en 4 bloques .....	63
Figura 3.22 Número de comparaciones ideales vs por bloques (política de comparación arbitraria) ventana de 32 entradas, dividida en 4 bloques .....	63
Figura 3.23 Número de comparaciones ideales vs bloques (política de comparación activa) ventana de 64 entradas, dividida en 4 bloques .....	64
Figura 3.24 Número de comparaciones ideales vs bloques (política de comparación activa) ventana de 32 entradas, dividida en 4 bloques .....	64
Figura 3.25 Número de comparaciones ideales vs. Bloques (política de comparación selectiva) ventana de 64 entradas, dividida en 4 bloques .....	65
Figura 3.26 Número de comparaciones ideales vs. Bloques (política de comparación selectiva) ventana de 32 entradas, dividida en 4 bloques .....	65
Figura 3.27 División de bloques de la ventana de instrucciones en columnas .....	66
Figura 3.28 Formato de los campos de la TMB para bloques divididos en columnas ...	67
Figura 3.29 Número de comparaciones ideales, por bloques y bloques divididos en columnas (ventana de 64 entradas) .....	68
Figura 3.30 Número de comparaciones ideales, bloques y bloques divididos en columnas (ventana de 32 entradas) .....	68
Figura 3.31 División de Bloques en Filas .....	69
Figura 3.32 Campos de la TMB para submodelo de División de Bloques en Filas .....	69
Figura 3.33 Número de comparaciones ideales, por bloques y bloques divididos en filas (ventana de 64 entradas) .....	70
Figura 3.34 Número de comparaciones ideales, por bloques y bloques divididos en filas (ventana de 32 entradas) .....	71
Figura 3.35 Número de comparaciones de bloques divididos en columnas vs bloques divididos en filas (ventana de 64 entradas) .....	71
Figura 3.36 Número de comparaciones de bloques divididos en columnas vs bloques divididos en filas (ventana de 32 entradas) .....	72
Figura 3.37 Partición imaginaria de la ventana de instrucciones .....	72
Figura 3.38 Campo de <i>ruta de habilitación</i> de 20 bits .....	73
Figura 3.39 Número de comparaciones por bloques y submodelos (ventana de 64 entradas) .....	74
Figura 3.40 Número de comparaciones por bloques y submodelos (ventana de 32 entradas) .....	74
Figura 3.41 Ventana de instrucciones dividida en bloques de tamaño dinámico .....	75
Figura 3.42 Número de comparaciones ideales vs bit ID (MSB) .....	76
Figura 3.43 Número de comparaciones ideales vs bit ID (LSB) .....	77
Figura 3.44 Ventana de instrucciones modificada para comparación por ID .....	77
Figura 3.45 Esquema de comparadores para identificación ID .....	78
Figura 3.46 Número de comparaciones ideales vs bit PMSB .....	79
Figura 3.47 Número de comparaciones con bit PLSB vs bit PMSB .....	79
Figura 3.48 Comparadores del conjunto de identificación ID .....	80

Figura 3.49 Número de comparaciones con bit MSB, LSB, PMSB, LSB y grupo de bits CID .....	81
Figura 3.50 Número de comparaciones ideales vs comparaciones con grupo de bits CID .....	81
Figura 3.51 Número de comparaciones de bloque/cuadrantes vs comparaciones con grupo de bits CID .....	82
Figura 4.1 Latencia en ciclos (Mapeo Directo vs. Asociativa nivel dos ) .....	86
Figura 4.2 Miss rate (Mapeo Directo vs. Asociativa nivel dos) .....	86
Figura 4.3 Carga de trabajo redundante .....	88
Figura 4.4 Redundancias en tiempo de ejecución .....	88
Figura 4.5 Hit rate % (Mapeo Directo, PSA-Caché, Asociativa Nivel dos) .....	89
Figura 4.6 Hit rate (%) por cada conjunto de una memoria asociativa nivel 2 .....	90
Figura 4.7 Datapath de predicción de conjunto a acceder de una memoria AS2 secuencial .....	91
Figura 4.8 Hit rate de predicción (32 ent., 128 ent., 1024 ent., SPECINT ) .....	93
Figura 4.9 Hit rate de predicción (32 ent., 128 ent., 1024 ent., SPECFP) .....	94
Figura 4.10 Miss rate de predicción (32 ent., 128 ent., 1024 ent., SPECINT) .....	94
Figura 4.11 Miss rate de predicción (32 ent., 128 ent., 1024 ent., SPECFP) .....	94
Figura 4.12 Contador del campo <i>state</i> .....	95
Figura 4.13 Latencia de acceso (Mapeo directo, Asociativa nivel dos, PAAC) .....	98



# Lista de Tablas

Tabla 1.1 Relación tamaño/asociatividad vs efecto. ....	19
Tabla 1.2 Clasificación de instrucciones .....	21
Tabla 2.1. Simuladores para principiantes. ....	32
Tabla 2.2. Simuladores intermedios .....	33
Tabla 2.3. Simuladores avanzados .....	34
Tabla 2.4. Conjunto de simuladores de Simplescalar .....	36
Tabla 2.5. Cargas de trabajo de tipo entero (SPEC200INT) .....	39
Tabla 2.6. Cargas de trabajo de tipo entero (SPEC200INT) .....	40
Tabla 2.7 Opciones de configuración para el Simplescalar .....	41
Tabla 3.1 Bus de información de bloques divididos en filas .....	68
Tabla 4.1 Tabla de porcentaje de predicción correcta .....	94
Tabla 4.2 Fórmula de rendimiento (latencia en ciclos) .....	95
Tabla 4.3 Porcentajes de miss rate (%) .....	96
Tabla 4.4 Latencia de acceso (ciclos) .....	97
Tabla 5.1 Cuadro sinóptico modelo propuesto vs impacto en la arquitectura .....	100

# Resumen

Al detectar las causas que limitan el rendimiento de un procesador superescalar, se estudian y se proponen técnicas con el afán de resolver dicha problemática. En esta tesis se proponen varios modelos con la finalidad de resolver los problemas como la reducción de la latencia del sistema de memoria, por medio de la disminución de la tasa de fallos (enfocándose en el nivel más cercano al procesador de la jerarquía de memoria, caché de datos). El diseño de una memoria que conjunte las ventajas de los modelos de memoria de mapeo directo y asociativa, al tener un tiempo de acceso y un porcentaje de error bajos. Detección de operaciones redundantes, y su eliminación de la ventana de instrucciones, tales como el número exagerado de comparaciones que se hace en busca de receptores, dentro de la ventana, para un dato recién generado por las unidades funcionales. Proponiendo diferentes modelos de comparación selectiva y dirigida a los potenciales receptores del dato, dando solución a las dependencias de datos en la ventana de instrucciones.

# Abstract

When detecting the causes that limit superscalar processor's performance, it's possible to study them and propose techniques to set out to solve the problematic. In this thesis, we present several models with the purpose of solving the problems like the reduction of the latency of the memory system, by means of the diminution of the failure rate (focusing in the level closest to the processor of the memory hierarchy, cache of data). One of the models is the design of a memory that combines the advantages of the models of associative memory and direct mapped memory, and it has a low percentage and access time of error. Detection of redundant operations, and its elimination of the window of instructions, such as the exaggerated number of comparisons that become in search of receivers, within the window, for a data just generated by the functional units. Proposing different models from comparison selective and directed to the receiving potentials of the data, giving solution to the dependencies of data in the window of instructions.

# Introducción

En los últimos años, el desarrollo de nuevas tecnologías aplicadas a los procesadores ha permitido obtener ganancias significativas en cuanto al tiempo de procesamiento, logrando de esta forma procesadores cada vez más poderosos. Esto ha sido producto no solo del desarrollo alcanzado en la manipulación de los semiconductores y su alta integrabilidad, sino también del buen diseño de la microarquitectura de los procesadores. En este último campo se han dado la mayoría de las innovaciones debido a la observación del comportamiento de los procesadores bajo cargas de trabajo comunes y se parte desde este punto para dar solución a los problemas existentes.

Siguiendo este punto de vista, en cuanto a la observación del comportamiento de la microarquitectura del procesador, es sabido, que al paso de los últimos años es cada vez más difícil proporcionar al procesador el número de instrucciones que es capaz de procesar en forma simultánea, característica que presentan los procesadores superescalares.

Este tipo de problema es denominado cuello de botella (bottleneck), cuyo nombre se debe a que impide el libre flujo de la información a través de las etapas del procesador. Este cuello de botella específico, no proporciona la cantidad de instrucciones que el procesador puede computar, teniendo como causas algunas de las siguientes:

- a) Latencia del sistema de almacenamiento (memoria).
- b) Procesamiento de operaciones redundantes.
  - Dependencia entre instrucciones o bajo nivel de paralelismo del código

Las diferentes técnicas que se aplican para resolver esta clase de problemas influyen de manera considerable en el consumo de energía del procesador, ya que se implementan otros dispositivos que pueden aumentar en forma considerable el consumo de energía. En estudios realizados, se sabe que al tratar de disminuir el consumo de energía de un procesador se sacrifica parte del rendimiento de operación.

Cada una de las causas mencionadas anteriormente afectan al rendimiento en forma diferente y obviamente la forma de atacarlas también es diferente para cada caso.

- a) Latencia del sistema de almacenamiento (memoria)

La latencia del sistema de almacenamiento de memoria se ha visto en desventaja en el avance o desarrollo de la velocidad de respuesta en comparación con las velocidades que alcanzan los procesadores actuales [1]. El resultado de este contraste es la limitación del sistema de memoria para satisfacer las necesidades del procesador en lo relacionado a accesos a la memoria. Esto ha permitido que en la última década muchos trabajos se hayan enfocado al desarrollo del sistema de memoria tratando de disminuir esta diferencia entre el procesador y la memoria.

Sin embargo algunas de estas técnicas incrementan el ancho de banda de la memoria ocasionando que esto llegue a ser progresivamente una limitante mayor para el desempeño del procesador [2].

Otras técnicas propuestas como software y hardware prefetching [3], buffers de almacenamiento del flujo de datos [4], ejecución de cargas de almacenamiento especulativas [5] y multithreading [6], reducen la latencia de la memoria caché, pero incrementan el tráfico entre el procesador y la memoria principal.

La técnica más aceptada por parte de los fabricantes de procesadores es la reducción de la latencia de la memoria, por medio de disminuir la tasa de fallos. Esto es, si la memoria más cercana al procesador tiene una tasa de fallos mínima, quiere decir que no se pierde tiempo en conseguir el dato deseado de los siguientes niveles del sistema de memoria, y el procesador tiene la información deseada en el menor tiempo posible.

b) Procesamiento de operaciones redundantes

- Dependencia de instrucciones o bajo nivel de paralelismo del código

El número de instrucciones procesadas por ciclo de máquina en los procesadores actuales puede ser alto según las características de la micro arquitectura, sin embargo, esto no se logra debido a la interdependencia de las instrucciones. Este problema es originado desde la forma de programar ya que se realiza en forma secuencial, lo que no permite el paralelismo del código. Ante esta problemática se manejan técnicas como *multithreading* o precompilación del código a fin de detectar posibles bloques de código que sean procesados al mismo tiempo.

Muchos de los programas ejecutados en el procesador presentan ciertas características, que pueden ser analizadas y explotadas para tener un mejor rendimiento. Dentro de esas características o fenómenos se encuentra el cálculo constante de las mismas instrucciones. Esto es, si una instrucción consta de los operandos A y B, produciendo el resultado C y luego durante la ejecución del programa se presenta el mismo tipo de instrucción con los operandos A y B de forma repetitiva, se sabe que el resultado será C, a esto se le llama repetición de instrucciones dinámica [7].

El procesador puede lanzar a ejecutar operaciones según el número de unidades funcionales con que cuenta y que se encuentren disponibles en ese momento. Si se tiene una operación redundante que se puede determinar su valor sin la necesidad de computar, permitirá que otras operaciones utilicen esa unidad funcional incrementando el número de instrucciones ejecutadas por ciclo (ILP-Instruction Level Parallelism).

Otro beneficio que aporta la detección de este tipo de operaciones es que se pueden eliminar dependencias de otras instrucciones con respecto a la instrucción de tipo redundante, además de aliviar la señalización en tiempo por errores de especulación (miss prediction) en el flujo que sigue el programa ejecutado en forma dinámica.

Al detectar las causas que limitan el rendimiento de un procesador superescalar se estudian y se proponen técnicas con el afán de resolver dicha problemática. En esta tesis se proponen varios modelos con la finalidad de resolver los problemas mencionados anteriormente, y estos son:

- Reducción de la latencia del sistema de memoria por medio de la disminución de la tasa de fallos (enfocándose en el nivel más cercano al procesador de la jerarquía de memoria, caché de datos). El diseño de una memoria que conjunte las ventajas de los modelos de memoria de mapeo directo y asociativa, al tener un tiempo de acceso y un porcentaje de error bajos. Para lograr este objetivo se han planteado diferentes modelos de acceso, manejo de errores, información reemplazada (basura) y modelos de predicción para localización del dato. Una sencilla técnica para lograr esta clase de comportamiento es el uso de memorias asociativas pero con acceso secuencial con ayuda de un predictor en base a *stride*, denominado PAAC.
- Detección de operaciones innecesarias, y su eliminación de la ventana de instrucciones, tales como el número exagerado de comparaciones que se hace en busca de receptores, dentro de la ventana, para un dato recién generado por las unidades funcionales. Proponiendo diferentes modelos de comparación selectiva y dirigida a los potenciales receptores del dato, dando solución a las dependencias de datos en la ventana de instrucciones.

Estos dos problemas tienen en común que entran en la definición de operaciones redundantes. La palabra redundante parte del latín *Redundantia* que significa demasiada abundancia de una cosa, y ya aplicado para nuestro campo se puede entender como “*Exceso de operaciones con un valor nulo para el desempeño del procesador*”. Por ejemplo el número de comparaciones en busca de un receptor en la ventana de instrucciones o el acceso a la memoria en busca de un dato con la posibilidad de no encontrarlo.

Los capítulos que componen esta tesis presentan la investigación, el diseño y la implementación de modelos capaces de reducir el número de operaciones redundantes, así como los resultados obtenidos. Se ha dividido en cinco capítulos los cuales se resumen a continuación:

En el primer capítulo se da una explicación del funcionamiento de un procesador superescalar, permitiendo entender mejor la problemática presentada en esta introducción. En ese capítulo se muestra las etapas en la que se secciona el *datapath* de un procesador superescalar con sus respectivas ventajas y desventajas de realizar la segmentación del procesamiento de instrucciones y las propuestas que se han hecho a lo largo de los últimos años para solucionar los problemas característicos de una arquitectura superescalar.

En el segundo capítulo se presentan las herramientas utilizadas para el desarrollo, modelado, simulación y parámetros de medición empleados en los modelos propuestos en esta tesis, dando muestra del nivel de semejanza de las simulaciones realizadas con un modelo implementado en forma física.

En el tercer capítulo se describe el modelo de ejecución de instrucciones fuera de orden empleado en las arquitecturas superescalares, conocido como ventana de instrucciones, donde radican las instrucciones que no es posible procesar en ese momento. Al analizar el comportamiento de este esquema es fácil detectar operaciones de tipo redundante y proponer esquemas que reducen el consumo de energía.

El cuarto capítulo se enfoca en el análisis de operaciones redundantes hacia la jerarquía de memoria, en especial en la memoria caché de datos de nivel uno, la cual es la más cercana al procesador y de mayor uso por parte del procesador, además de la propuesta de un modelo de predicción-acceso a una memoria caché de datos de tipo asociativa de nivel dos.

En el último y quinto capítulo se redacta la conclusión de la tesis, así como comentarios finales con respecto al trabajo de investigación realizado a lo largo de la tesis. Los trabajos futuros, que surgieron como producto de los modelos desarrollados en la tesis, se presentan en este capítulo haciendo mención del posible impacto del desarrollo de los mismos y publicación de resultados. En la parte final se encuentra la bibliografía seccionada por capítulos.

# Descripción y definición del problema

Actualmente el rendimiento del cómputo es cada vez mas rápido y eficiente para atender los trabajos de investigación, principalmente, y los de la vida diaria en todos los aspectos, hacen evidente la necesidad constante de satisfacer la demanda de procesadores de alto rendimiento con tiempos de operación cada vez menores, ponen al investigador ante el estudio de barreras limitantes que requieren franquear para conseguirlo, barreras como el consumo de energía y el calor que genera su operación.

En México como en todo el mundo se requieren de equipos que ayuden de forma mas eficiente a técnicos y científicos para el desarrollo de nuestra civilización y cultura; es imperativo contar con procesadores cada vez mas rápidos con capacidades escalares, con menor consumo de energía, menor costo de operación y mayor facilidad de uso.

Rápidamente la energía que consume y el calor que genera un dispositivo de cómputo dejan de ser simples irritantes para convertirse en grandes limitaciones de esta clase de dispositivos. La demanda de sistemas cada vez más poderosos y veloces eleva la cantidad de calor que se requiere disipar en muchos dispositivos nuevos a un grado excesivo. La fase de innovación en el campo de los dispositivos de computadoras, esta relacionada con la meta de mejorar simultáneamente el rendimiento y la reducción de consumo de energía.

En los últimos años el desarrollo de nuevas arquitecturas de procesadores ha empujado a los desarrolladores a sacar ventaja de ciertos comportamientos de funcionamiento del procesador. Mas aun si lo que se desea lograr es el incremento de rendimiento de microprocesadores con un bajo costo de consumo de energía, sobretodo enfocándose al ultimo punto.

El consumo de energía actualmente es una variable directamente proporcional al rendimiento, sin embargo, el reto de hacer lo inverso o de no incrementar el gasto de energía en mayor procesamiento, motivan a desarrollar la presente tesis. Así también, el aumento de calor en condiciones actuales de operación para obtener un mayor rendimiento, hace pensar en la manera de contribuir con esta tesis a la reducción o eliminación de operaciones innecesarias para disminuir de forma importante el calor que se genera e incrementar los procesos.

La motivación para la investigación de esta tesis es quitar las barreras en la agilización de procesos de cómputo, la capacidad de operación, posibilidad de desarrollo a mejores niveles, reducción de consumo de energía; traerá como beneficio el poder contar con procesadores eficientes que sirvan a científicos y técnicos, para tener un soporte adecuado en su labor de mejorar la calidad de vida de la humanidad.

# Objetivos

## Objetivo general

Plantear modelos de arquitecturas enfocadas a la detección de posibles operaciones redundantes, realizando una reducción en el número de operaciones realizadas en diferentes etapas (lanzamiento de instrucciones y memoria caché), de tal forma que se acelere la respuesta esperada por el procesador.

## Objetivos Particulares

- Detección y análisis de operaciones redundantes en las diferentes etapas del *datapath* de un procesador superescalar.
- Diseño e implementación de modelos propuestos para solucionar comportamientos redundantes.
- Simulación y análisis del procesador a partir de la implementación de los modelos propuestos y métrica de su impacto en el comportamiento del procesador en las etapas implicadas.



# Justificación

La demanda de sistemas cada vez más poderosos y veloces, eleva la cantidad de calor que se requiere disipar en muchos dispositivos nuevos a un grado excesivo. La fase de innovación en el campo de dispositivos de computadoras esta relacionada con la meta de mejorar simultáneamente el rendimiento y reducir el consumo de energía.

En los últimos años el desarrollo de nuevas arquitecturas de procesadores ha empujado a los diseñadores a sacar ventaja de ciertos comportamientos de funcionamiento del procesador. Más aún si lo que se desea lograr es el incremento del rendimiento del procesador elevando el número de instrucciones procesadas en el mismo tiempo, a un menor costo de consumo de energía.

# Capítulo 1

## Microarquitectura de procesadores superescalares

### 1.1 Introducción

El procesamiento superescalar significa la habilidad de procesar múltiples instrucciones en un mismo ciclo de reloj. Introducidos al inicio de la década de los 90's. Los procesadores superescalares han sido producidos por todos los fabricantes de procesadores.

Un procesador superescalar típico hace la búsqueda y la decodificación de un grupo de instrucciones al mismo tiempo. Como parte del proceso de decodificación, el resultado de un salto condicional es predicho por adelantado para asegurar una ejecución continua de instrucciones. El flujo entrante de instrucciones es analizado para saber si tiene dependencias de datos y las instrucciones son distribuidas a las unidades funcionales de acuerdo al tipo de operación. El siguiente paso, es la ejecución en paralelo de las instrucciones lanzadas simultáneamente a cada una de las unidades funcionales, basándose primero en la disponibilidad de los operandos de la instrucción, en vez del orden del programa. Esta es la principal característica de los procesadores superescalares, y es llamada *dynamic instruction scheduling*.

Al final, los resultados de las instrucciones son reordenados para que puedan ser usados al momento de actualizar el estado de los procesos en el correcto orden del programa. Debido a que cada instrucción se le considera como una entidad, puede ser ejecutada en paralelo, lo que se le llama *instruction level parallelism* (ILP).

### 1.2 Microarquitectura típica de un procesador superescalar

La figura 1.1, ilustra los principales componentes de la arquitectura de un procesador superescalar. Las etapas principales son: búsqueda de instrucciones y predicción de saltos, decodificación y análisis de dependencias de registros, emisión y ejecución de instrucciones, análisis de operaciones de memoria y ejecución, reordenamiento y retiro de instrucciones. Estas etapas son listadas más o menos en el mismo orden en que las instrucciones son ejecutadas.

#### 1.2.1 Búsqueda de instrucciones y predicción de saltos

La fase de búsqueda de instrucciones, proporciona las instrucciones al resto de las fases del procesamiento. Una memoria caché, la cual es una pequeña memoria que contiene las instrucciones recientemente utilizadas [1], es usada en todos los procesadores actuales, para reducir la latencia e incrementar el ancho de banda del número de instrucciones buscadas en esta etapa. Una memoria cache esta organizada por bloques o líneas que contienen múltiples líneas de código del programa que se está ejecutando; el contador del programa (*program counter*, PC), es usado para buscar en forma asociativa si la

instrucción siguiente se encuentra en una de las líneas de la memoria caché. Si esto es verdad, se considera que se tiene un éxito o acierto en la memoria (*cache hit*), en caso contrario es un error o falla (*cache miss*) y la línea que contiene la instrucción solicitada es traída desde la memoria principal para ser colocada en la memoria caché.

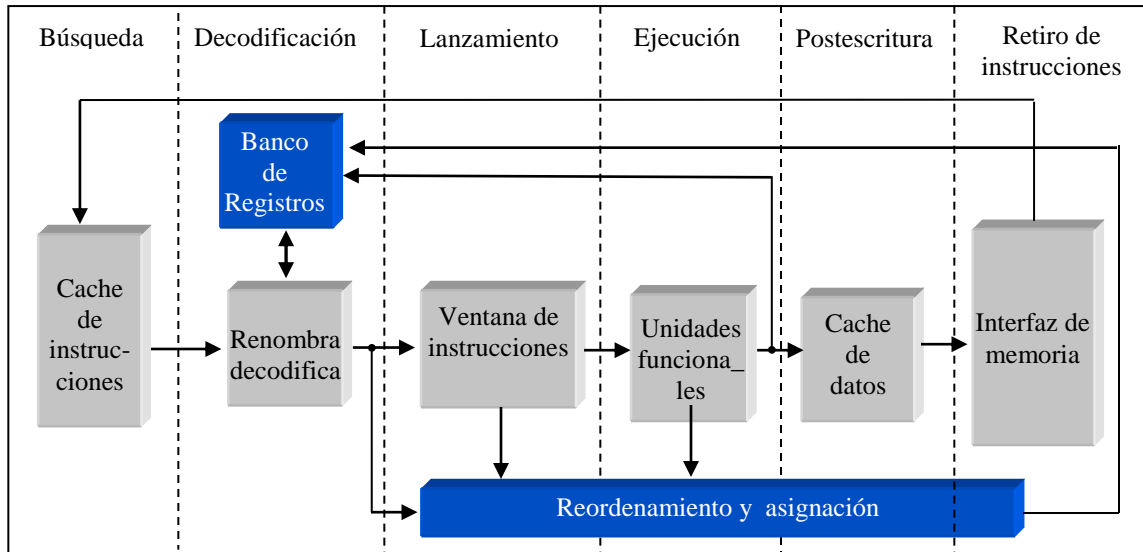


Fig. 1.1 Etapas de procesamiento de un procesador superescalar

Para soportar el ancho de banda de instrucciones que es capaz de buscar el procesador, se ha separado la memoria caché de instrucciones de la memoria caché de datos. La memoria caché de instrucciones se ha enfocado sobre todo a tener múltiples lecturas en un mismo ciclo de reloj.

El número de instrucciones por ciclo buscadas o traídas de la memoria caché de instrucciones, debería ser al menos igual al número de instrucciones que pueden ser decodificadas y ejecutadas, aunque usualmente es mayor, debido a que el promedio de instrucciones ejecutadas es menor. Este margen permite que la cache de instrucciones pueda cometer algunos errores y no afectar en forma significativa el desempeño del procesador. En la mayoría de los procesadores se explota el fenómeno de localidad de datos dentro de las memorias cache, esto es, si una instrucción es solicitada, hay una alta probabilidad de que las instrucciones que están cerca de ésta sean llamadas en futuras referencias del procesador, por lo que no solo se manda la instrucción solicitada sino también las instrucciones aledañas.

Para suavizar las irregularidades de la búsqueda de instrucciones en la caché de instrucciones debido a “misses”, se hace uso de un buffer para almacenar instrucciones entre la caché de instrucciones y la etapa de decodificación y renombramiento (Fig. 1.1), lo que permite tener un suministro de instrucciones durante periodos en los que está restringido el acceso a la memoria cache de instrucciones debido a un *miss*.

El método habitual para realizar la búsqueda de instrucciones es incrementar el valor del *program counter* por el número de instrucciones a buscar, y usar este valor para acceder

al siguiente bloque de instrucciones. En el caso de instrucciones de salto, el mecanismo de búsqueda debe de ser redirigido para buscar las instrucciones marcadas por la etiqueta de salto. Debido a que pueden ocurrir retardos en este proceso de redireccionamiento de la máquina de búsqueda, el manejo de estas instrucciones de salto es crítico para el buen funcionamiento del procesador. El procesamiento de las instrucciones de salto puede dividirse en las siguientes partes:

- a) Reconocimiento de una instrucción de salto condicional.
- b) Determinación del resultado de la condición del salto (brincar o no brincar).
- c) Calcular la etiqueta a la cual saltar.
- d) Transferir el control a la nueva dirección (redireccionamiento en caso de salto).

Para cada una de estas partes, existen varias técnicas propuestas para mejorar el mecanismo de manejo de brincos, incluyéndose la predicción de saltos, sin embargo pasaremos por alto esta parte debido a que no se requiere profundizar para nuestro objetivo y aplicación de la tesis.

### 1.2.2 Decodificación, renombramiento y despacho de instrucciones

Durante esta etapa las instrucciones son retiradas del buffer de instrucciones, examinadas, y renombrados los registros de las instrucciones que tienen dependencias para el resto de las etapas de procesamiento. Esta etapa incluye la detección de dependencias de datos (conocidos como dependencias tipo RAW = *Read After Write*) y la solución de dependencias entre registros (tipo WAW = *Write After Write* y WAR = *Write After Read*) causados por el re-uso de registros. Un riesgo o dependencia de registros, es el fenómeno que se da al tratar de elevar el número de instrucciones procesadas por ciclo en forma paralela, ya que el programador realiza en forma secuencial muchas operaciones que dependen de resultados de operaciones que se realizan en forma previa. Al momento de computar varias operaciones o instrucciones es muy posible que estén ligadas unas con otras al depender del resultado de una operación. Las dependencias o riesgos entre registros (WAW, WAR) se deben a que muchas operaciones de nivel ensamblador hacen uso común de uno o dos registros (por ejemplo el registro cero), por lo que al tratar de paralelizar un fragmento del código, es muy posible que dos operaciones traten de leer o escribir en el mismo registro; esta clase de riesgos se solucionan haciendo uso de técnicas de renombramiento, que explicaremos en la sección de decodificación de esta etapa.

#### 1.2.2.1 Tipos de dependencias

- *Dependencias de datos verdaderas*

Dos operaciones conforman una dependencia de dato verdadera si una de ellas crea un valor que es usado por la otra operación (también llamada dependencias tipo RAW). Esta dependencia forja un orden de las operaciones, sujeta a la secuencia en como se van generando los datos necesarios o como serán utilizados por operaciones subsecuentes.

- *Dependencias de almacenamiento*

Las dependencias también pueden existir por tener un límite en el espacio de almacenamiento. Tal como en las dependencias de memoria, se requiere que un cálculo sea desarrollado para ubicar la posible localización del dato. Las dependencias de almacenamiento son comúnmente catalogadas dentro de las categorías de Escritura-después-Lectura (WAR) y Escritura-después-Escritura (WAW). Debido a que diferentes datos pueden ser almacenados en una misma localidad durante la vida del programa, se requiere de sincronización para asegurar que una operación esta accediendo al valor correcto guardado en esa localidad. La violación de una dependencia de almacenamiento podría resultar en el acceso a una localidad no inicializada, o a otro dato almacenado en la misma localidad.

- *Dependencias de Control*

Una dependencia de control es introducida cuando no se sabe que instrucción será la próxima a ejecutarse hasta que se sepa el resultado de operaciones anteriores. Tales dependencias se producen debido a instrucciones de saltos condicionales que escogen entre varias rutas de ejecución basadas en el resultado de ciertas pruebas.

- *Dependencias de recursos*

Las dependencias de recursos (algunas veces llamados riesgos estructurales) ocurren cuando algunas operaciones deben retardarse antes de ser lanzadas ya que no son suficientes los recursos de hardware para procesarlas. Ejemplos de limitaciones de recursos en los procesadores incluye el número de unidades funcionales, número de entradas de la ventana de instrucciones, y número de registros físicos (cuando es aplicada la técnica de renombramiento).

### 1.2.2.2 Despacho de instrucciones

El paradigma básico de la secuencia a través de un programa, por ejemplo, el ciclo de búsqueda-ejecución usando un contador de programa, ha sido usado desde hace 50 años. Una consecuencia de este paradigma, es que los programas son escritos dando por tácito que las instrucciones serán ejecutadas en el mismo orden en que aparecen en el programa. Sin embargo para lograr un alto rendimiento, los procesadores modernos tratan de ejecutar múltiples instrucciones simultáneamente, y en algunos casos en orden diferente a la secuencia original del programa. Este reordenamiento puede hacerse en la compilación o en hardware, en forma dinámica. Los procesadores superescalares pertenecen a esta clase de arquitecturas que explotan el nivel de paralelismo de instrucciones (ILP).

Los procesadores superescalares y los compiladores para estos, normalmente convierten el orden total de las instrucciones como aparece en el programa, en un ordenamiento parcial determinado por las dependencias de datos y de control. Las dependencias de control (las cuales aparecen como saltos condicionales) presentan un mayor obstáculo a

la ejecución altamente paralela debido a que estas dependencias deben ser resueltas antes que todo el bloque subsecuente de instrucciones para saber si es válido.

Enfocándose en las dependencias de control, un programa estático puede representarse como un diagrama de control de flujo (*Control Flow Graph – CFG*) donde los bloques básicos son nodos, y los arcos o flechas representan el flujo de control de un bloque básico a otro. La ejecución dinámica de un programa puede ser vista como un camino a través del CFG del programa, generando una secuencia de bloques básicos que han de ser ejecutados por una sección de la ejecución del programa.

Para alcanzar un alto rendimiento, un procesador superescalar debe tratar de recorrer este CFG con un alto grado de paralelismo. Un predictor de saltos con ejecución especulativa es una de las técnicas comúnmente utilizadas para lograr el mayor nivel de paralelismo que permite el programa. La principal restricción en cualquier ejecución paralela, es que debe mantener la semántica secuencial asumida en el programa original.

En el caso de dependencias de datos, aunque parece sencillo, se requiere de un buen ruteo de datos por medio del hardware. Los datos que son esperados para solucionar las dependencias de datos son pasados por registros o localidades de memoria, lo que requiere un correcto ruteado por medio del hardware implementado. Además, en el área de comunicación de datos entre tareas, es donde difieren totalmente las arquitecturas superescalares de otras.

### 1.2.2.3 Renombramiento de registros

Típicamente esta etapa también se encarga de distribuir o despachar las instrucciones, a los *buffers* asociados con cada una de las unidades funcionales con las que cuenta el procesador para su posterior ejecución.

El trabajo de decodificación de instrucciones de esta etapa esta dado por tres pasos:

- i) Identificar la operación a realizar,
- ii) Identificar los elementos de almacenamiento donde residen los operandos de entrada de la operación a realizar y
- iii) La localización donde va ser colocado el resultado de la operación.

En el programa estático los elementos de almacenamiento son los registros arquitecturales o llamados registros lógicos. Para prevenir riesgos tipo WAW y WAR e incrementar el paralelismo en la ejecución dinámica del código, se hace uso de otros elementos de almacenamiento llamados registros físicos. Mediante una lógica de renombramiento, es posible ver varios datos almacenados en diferentes registros físicos, pero todos ligados a un mismo registro lógico, de tal forma que se puede almacenar diferentes datos para un mismo registro lógico, sin embargo, cada valor corresponde a un punto diferente en el tiempo de ejecución del programa visto desde una forma secuencial.

Cuando una instrucción crea un nuevo valor para un registro lógico, la localización física de este valor es dándole un nombre conocido por el hardware. Cualquier subsecuente instrucción que haga uso de este valor como una entrada, le será dado el nombre o etiqueta del registro físico en donde esta almacenado. Esta acción es realizada en la etapa de decodificación/renombramiento/despacho (Fig. 1.1), al reemplazar el nombre del registro lógico con el nombre del registro físico de donde se encuentra el dato; después de esto se dice que el registro ha sido *renombrado* [2].

Existen dos métodos muy usados para realizar el renombramiento de registros.

1. Archivo de renombramiento de registros independiente (*Stand-alone rename register file*).
2. Renombramiento con ROB (*Reorder Buffer*).

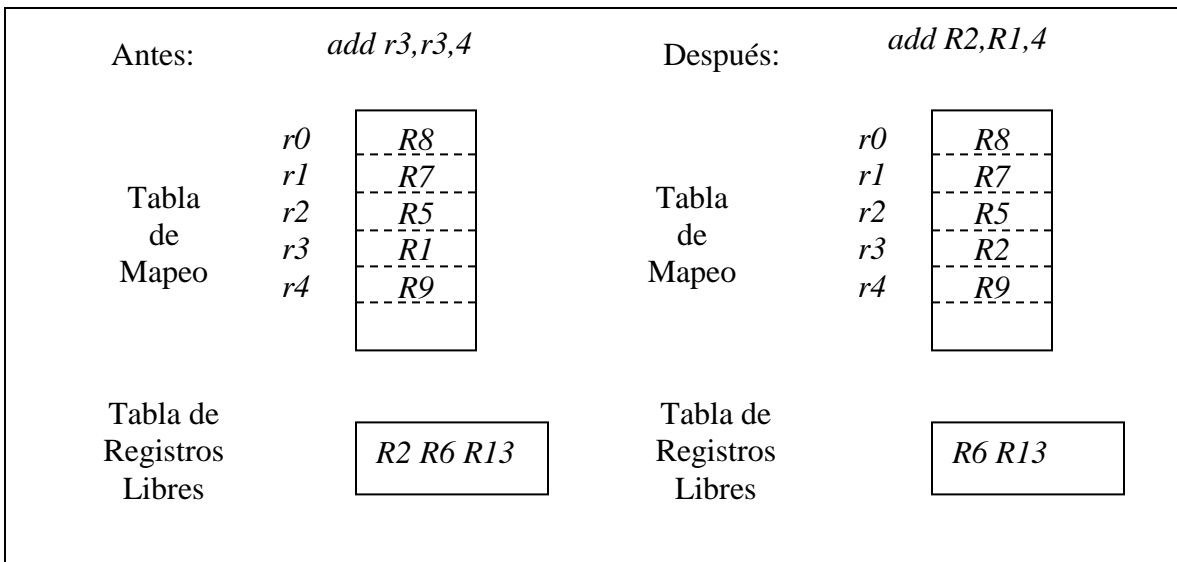
*1) Archivo de renombramiento de registros independiente (Stand-alone rename register file).*

Las instrucciones son decodificadas y los registros renombrados en forma secuencial de acuerdo al orden del programa. Cuando una instrucción es decodificada, su registro destino lógico es asignado a un registro físico de una lista de registros físicos libres en ese momento, es decir que en ese momento no tienen un valor lógico asignado, y la tabla de mapeo es ajustada para reflejar la nueva dirección (lógica o física) del registro. En el proceso, los registros físicos son removidos de la lista de registros libres., También, como parte de la operación de renombramiento, los registros fuente de la operación son renombrados con los registros físicos que contienen el valor adosado a estos registros físicos, esto es, el registro físico de donde debe ser leído el valor para realizar la operación. Se tiene un número mayor de registros físicos, que registros lógicos [3] [4] [5] [6].

La figura 1.2 es un ejemplo sencillo de cómo se lleva a cabo el renombramiento de registros. La instrucción es primero considerada como `add r3,r3,4`. Los registros lógicos son denotados con letras minúsculas. En la parte inferior de la figura se muestra como en la tabla de mapeo el registro 3 (r3) esta mapeado al registro físico 1 (R1). Durante el renombramiento el registro de entrada es reemplazado con el registro físico de donde se va a leer el dato. En la parte inferior también se muestra la tabla de registros físicos libres de los cuales el primero es el R2, registro que servirá para reemplazar el registro lógico destino de nuestra operación. Al final queda la instrucción `add R2,R1,4` y la tabla de mapeo como lo muestra la figura en la parte derecha. Las siguientes instrucciones tomaran el valor del registro lógico r3 del registro físico R2.

Lo importante en este proceso es contar con registros físicos libres. Después de que ha sido leído por última vez, y no va ser necesitado más, puede ser colocado nuevamente en la lista de registros libres para ser renombrado nuevamente. Dependiendo de la implementación específica de este mecanismo de renombramiento, la liberación de registros físicos puede requerir un mecanismo de hardware complicado. Un método es asociar un contador a cada registro físico, de tal forma que cada vez que el registro fuente

es renombrado (registro lógico) se incrementa este contador, y cada vez que una instrucción lee el valor de este registro fuente, el contador es decrementado. El registro es liberado siempre y cuando el contador sea cero.



**Fig. 1.2.** Renombramiento de registro. Archivo de registro de renombramiento

En la práctica, un método más simple que un contador es esperar a que el correspondiente registro lógico no haya sido renombrado por la última instrucción (vea la sección de retiro de instrucciones).

### 2) Renombramiento con ROB (Reorder Buffer)

Este método de renombramiento de registros usa la misma cantidad de registros lógicos como físicos, y se trata de mantener una relación uno a uno. Además, existe un buffer con una entrada por cada instrucción activa (por ejemplo, una instrucción que ha sido despachada para su ejecución pero que no ha sido asentado su resultado en los registros lógicos, *commit*). Este *buffer* es típicamente llamado “*reorder buffer*” (*ROB*) [7][8][9], ya que es usado para mantener el orden propio del programa.

La figura 1.3 muestra el *reorder buffer*. Es fácil de comprender si se piensa en ello, como una memoria FIFO (*First In First Out*) implementado en hardware como un *buffer* con apuntadores de cola y cabecera. Como una instrucción es despachada según el orden del programa, las instrucciones son asignadas al apuntador de la cola del *reorder buffer*. Cuando una instrucción finaliza su ejecución, su respectivo valor de resultado es asignado a la respectiva entrada de esta instrucción dentro del *ROB* (*reorder buffer*). Cada vez que una instrucción alcanza la cabecera del *ROB* y ha completado su ejecución es retirada del *ROB*, obviamente conservando el orden del programa, y su resultado es colocado en el archivo de registros (registros lógicos, etapa de *commit*). Cuando una instrucción alcanza la cabecera del *ROB* y aun no termina su ejecución es detenida hasta que arribe su resultado, claro un procesador superescalar es capaz de retirar varias instrucciones del *ROB* a la vez.



El valor de un registro puede residir en los registros físicos o en el *reorder buffer*. Cuando una instrucción es decodificada, el valor de su resultado primero es asignado a una de las entradas físicas del *reorder buffer* y una entrada de la tabla de mapeo es activada en conformidad. Esto es, la tabla indica que el resultado puede ser encontrado en la entrada específica del reorder buffer correspondiente a la instrucción que lo generó. A cada registro fuente de las instrucciones se le asigna una etiqueta que sirve para acceder a la tabla de mapeo. La tabla indica que el registro correspondiente tiene un valor requerido, o que puede ser encontrado en el *reorder buffer*. Finalmente, cuando el *reorder buffer* esta lleno, el despacho de instrucciones es detenido temporalmente.

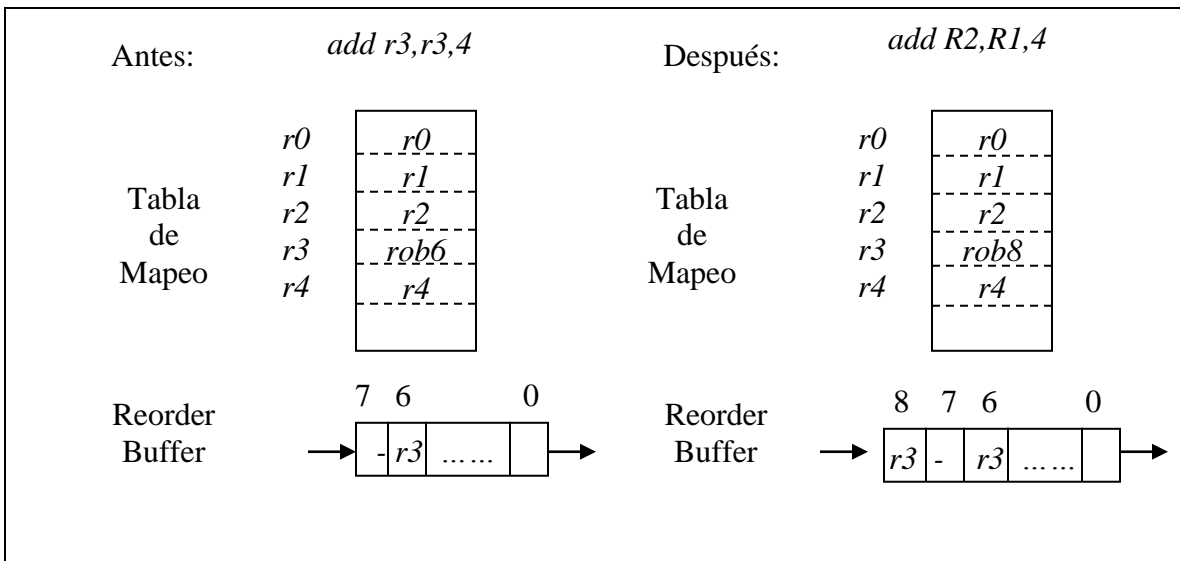


Fig.1.3 Renombramiento de registro. *Reorder Buffer*

La figura 1.3 muestra el proceso de renombramiento aplicado a la misma operación, *add r3, r3, 4*, como en la figura 1.2. En el lapso de tiempo antes de que la instrucción sea despachada, los valores de r1 y r2 residen en el archivo de registros. Sin embargo el valor de r3 reside (o residirá) en la entrada número 6 del *reorder buffer* hasta que la entrada del reorder buffer es desactivada y el valor puede ser escrito en el archivo de registros. Consecuentemente, como parte del proceso de renombramiento, el registro fuente r3 es reemplazado con la entrada 6 del ROB (rob6). La instrucción *add* es colocada en la cola del ROB, entrada número 8 (rob8). Este número de entrada del *reorder buffer* es guardado en la tabla de mapeo para que futuras instrucciones que usen el resultado de esta operación lo tomen de esta dirección del ROB.

En suma, sin importar el método que se use, el renombramiento de registros elimina los riesgos o dependencias tipo WAW y WAR solo dejando las verdaderas dependencias que son del tipo RAW.

### 1.2.3 Lanzamiento de instrucciones y ejecución en paralelo

En la sección anterior se explicó cómo en la etapa de decodificación/renombramiento/despacho se realiza el ordenamiento de las instrucciones en los diferentes *buffers* para su ejecución, ahora el siguiente paso es determinar cual de esas instrucciones almacenadas en los *buffers* de instrucciones puede ser lanzada a ejecutarse. Esta etapa (lanzamiento de instrucciones, *issue*) es el corazón de muchos procesadores superescalares ya que contiene la ventana de ejecución.

Idealmente una instrucción esta lista para ejecutarse tan pronto como sus operandos están listos. Sin embargo otras restricciones pueden impedir el lanzamiento de ejecuciones, más notablemente la disponibilidad de recursos físicos, tales como unidades de ejecución, interconexión, y puertos del archivo de registros (*reorder buffer*). Otras restricciones están relacionadas con la organización de los buffers que almacenan las instrucciones a ejecutarse.

La Fig. 1.4 es un ejemplo de como se lleva a cabo la ejecución en paralelo de una iteración de un segmento de programa. Esta orden de ejecución toma en cuenta los recursos de hardware con los que cuenta el procesador, en este caso dos unidades de ejecución de enteros, un bus de direcciones para la memoria y una unidad de brinco. La dirección vertical corresponde a los pasos en tiempo, la dirección horizontal corresponde a las operaciones ejecutadas en ese lapso de tiempo (unidad básica de tiempo, ciclo). En el código que se presenta solo el registro r3 está renombrado con sus respectivos registros físicos.

Los siguientes párrafos describen brevemente algunos tipos de organizaciones de *buffers* de instrucciones, incrementando su complejidad de uno a otro.

#### Método de enfilamiento sencillo (*Single Queue Method*)

Este caso consiste de una fila de espera o cola (*queue*) sencilla, Fig. 5(a), en la que no se aplica el lanzamiento de instrucciones fuera de orden, por lo que la técnica de renombramiento de registros no es requerida, y la disponibilidad de las operaciones puede ser manejada por medio de una simple reservación de registros con un bit. Un registro reservado es liberado cuando la instrucción se completa. Una instrucción puede ser lanzada (sujeto a la disponibilidad de recursos físicos o de hardware) si no hay reservaciones en sus operaciones.

#### Método de enfilamiento múltiple (*Multiple Queue Method*)

Con múltiples colas (*queue*), las instrucciones de cada cola son lanzadas en orden, pero las colas pueden lanzar las instrucciones fuera de orden con respecto unas de otras. Las colas están organizadas según el tipo de instrucciones. Por ejemplo, puede haber una cola para instrucciones de tipo flotante, otra para enteros, y otra mas para instrucciones de carga y almacena (*load/store*). Aquí, el renombramiento de registros puede ser usado en forma restringida. Por ejemplo, solo los registros cargados de la memoria pueden ser

renombrados. Esto permite a la cola de carga/almacena que vaya al frente de las demás colas, ocasionando que las instrucciones de búsqueda o carga de datos se realicen de forma adelantada cuando esto sea necesario. Algunas implementaciones superescalares usan este método [10] [11] [12].

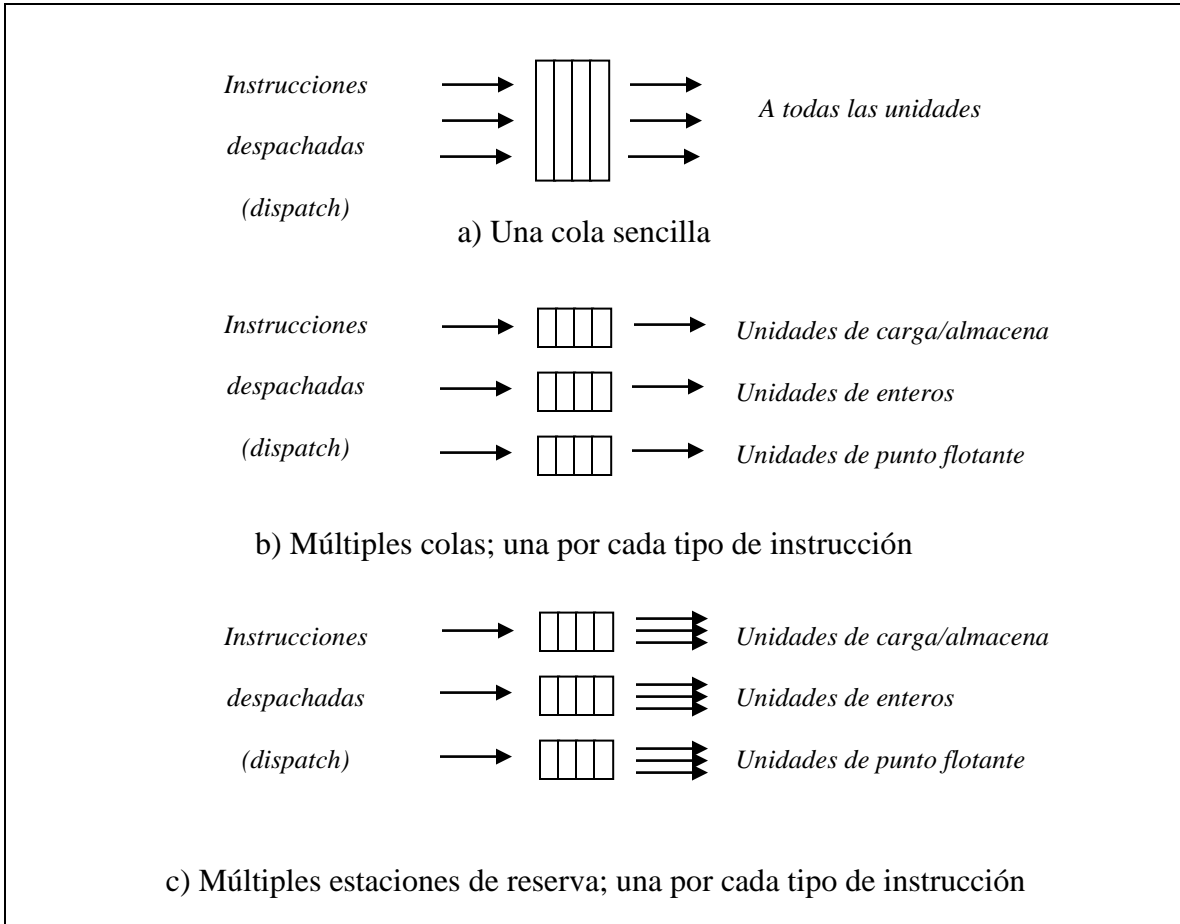


Fig. 1.4 Método de enfilamiento

### Estaciones de Reserva (*Reservation Stations*)

Con las estaciones de reserva, las cuáles fueron propuestas primero como parte del algoritmo de Tomasulo [13], las instrucciones pueden ser lanzadas fuera de orden con respecto al programa, consecuentemente, todas las estaciones de reserva simultáneamente monitorean sus operandos fuentes (registros) para la disponibilidad de datos. El camino tradicional para hacer esto es mantener el dato en la estación de reserva. Cuando una instrucción es despachada a la estación de reserva, cualquier dato de esta operación que está disponible es leído del archivo de registros y colocado en la estación de reserva. La lógica de la estación de reserva compara la etiqueta de un dato que no está disponible con la etiqueta de un dato que acaba de ser procesado y generado, cuando estas coinciden el dato recién procesado es colocado en la estación de reserva, y la instrucción que necesitaba el dato puede ser lanzada a ejecutarse. Las estaciones de reserva pueden

particionarse de acuerdo al tipo de instrucciones [14] [15] o tener una estación de reserva grande en la que se manejan todos los tipos de instrucciones. Finalmente, las implementaciones de estaciones de reserva más recientes no guardan los datos, sino que manejan apuntadores a su ubicación en el banco de registros o en el “*reorder buffer*”.

### 1.2.3.1 Estructura básica de la ventana de instrucciones o estación de reserva

En la ejecución de instrucciones en orden, el procesador para la decodificación de instrucciones al encontrarse con una instrucción decodificada que crea un conflicto de recursos, tiene una dependencia verdadera o una dependencia con respecto al dato generado por otra instrucción. Como resultado, el procesador es incapaz de computar las siguientes instrucciones que pueden ser ejecutadas. Para solucionar esta limitante, el procesador debe aislar la etapa de decodificación, de la etapa de ejecución, de tal forma que continúe decodificando instrucciones no importando si pueden ser ejecutadas de forma inmediata. Este aislamiento de etapas se realiza por medio de un *buffer* entre la etapa de decodificación y la de ejecución, denominado *ventana de instrucciones* (fig. 1.5).

Esta característica permite a la etapa de ejecución del procesador tener una mejor visión del flujo de instrucciones, de tal forma que es posible realizar un mejor ordenamiento de instrucciones.

Para el siguiente código,

```
R1<=mem[R0]    /* Instruction 1 */
R2<=R1+R2      /* Instruction 2 */
R5<=R5+1       /* Instruction 3 */
R6<=R6-R3      /* Instruction 4 */
```

La primera instrucción es una carga hacia el registro R1 que, en tiempo de ejecución, causa un error (miss) en la caché. Un CPU tradicional espera a que su unidad de interfaz de bus lea este dato de la memoria principal y regrese con el dato antes de pasar a la instrucción 2. Este procesador se detiene mientras espera el dato y esto provoca que sea un tiempo desaprovechado. Sin embargo, al aplicar lógica de lanzamiento de instrucciones, es posible verificar si las subsecuentes instrucciones están listas para ser lanzadas a las unidades funcionales. En el ejemplo de arriba la instrucción 2 no está lista para ser ejecutada, ya que depende de la primera instrucción, pero las instrucciones 3 y 4 sí están listas. De tal forma, si incrementamos el tamaño de la ventana de instrucciones, es posible incrementar la posibilidad de tener más instrucciones listas para ejecutarse, aumentando el ILP. Sin embargo, no podemos colocar los resultados de estas operaciones en los registros lógicos (registros visibles para el programador), puesto que necesita mantenerse la secuencia original del programa. De esta forma, el lanzamiento de instrucciones fuera de orden depende únicamente de la facilidad para ejecutar o no instrucciones en el orden del programa original, el cual puede incrementar el número de instrucciones a ejecutar. Esto es llamado tecnología de ejecución dinámica.

La unidad de despacho de instrucciones, selecciona las instrucciones de la ventana de instrucciones dependiendo de su estado. Si el estado indica que una instrucción tiene todos sus operandos listos, la unidad de despacho verifica la disponibilidad de recursos de hardware para ejecutar la instrucción. Si ambas condiciones son verdaderas, la unidad de despacho remueve la instrucción de la ventana de instrucciones y la envía a la unidad funcional correspondiente a esta instrucción, donde es ejecutada. El resultado de la instrucción es reenviado a la ventana de instrucciones para solucionar dependencias entre instrucciones con respecto a este valor.

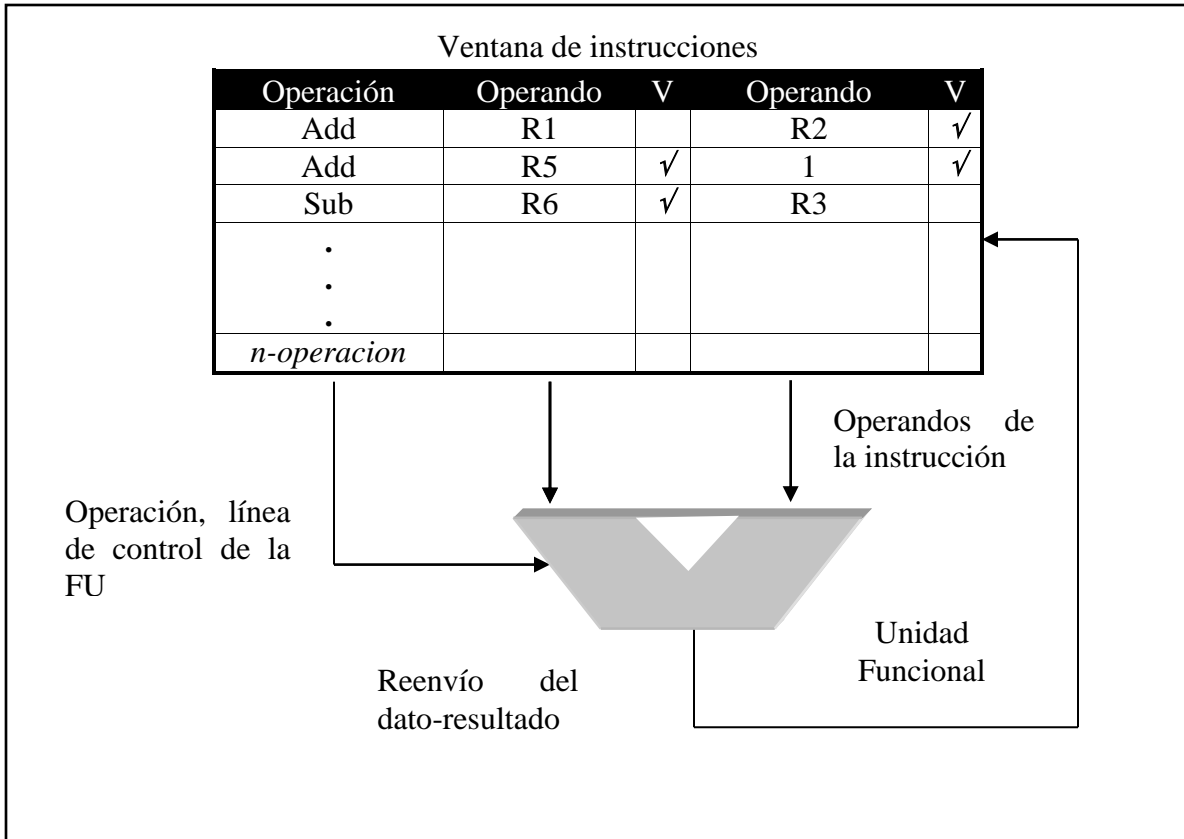


Fig. 1.5 Ventana de instrucciones

### 1.2.4 Operaciones de manejo de memoria

Las operaciones de manejo de memoria necesitan un tratamiento especial en procesadores superescalares. En los conjuntos de instrucciones RISC más modernos, solo se tiene instrucciones de carga y almacena para acceder a la memoria, y deben de usar estas instrucciones en la revisión de las operaciones de memoria.

Para reducir la latencia de las operaciones de memoria, se usa una jerarquía de memoria. La expectativa es que muchos de los datos sean proporcionados o servidos por la memoria caché, que reside en la parte mas baja de la jerarquía de memoria (concepto que

se trata en el capítulo 4, así como su optimización). Actualmente casi todos los procesadores cuentan con memoria caché de datos, y pueden ser divididas para tener múltiples niveles de almacenamiento dentro del procesador. Por ejemplo, una memoria pequeña pero rápida funciona como el primer nivel de caché, y una segunda memoria, no tan rápida, pero si mas grande, funciona como segundo nivel.

A diferencia de las instrucciones de la ALU (unidades funcionales), para las cuáles es fácil identificar los registros que van a ser accedados en la etapa de decodificación de instrucciones (*decode/dispatch*), no es posible identificar las localidades de memoria que van a ser accedidas por las instrucciones de carga y almacena hasta después de la fase de lanzamiento de instrucciones (*issue*). Para determinar las localidades de memoria que serán accedidas es necesario el cálculo de una dirección (*address*), usualmente una suma de enteros. Acorde a esto, las instrucciones de carga/almacena son lanzadas a una etapa de ejecución en donde se calcula la dirección (*address*).

Después del cálculo de la dirección, una traslación o traducción de la dirección puede ser requerida, esto consiste en acceder con la dirección calculada a una memoria denominada TLB (*Translation Lookaside Buffer*), de la cuál se obtienen las direcciones recientemente mas utilizadas. Hasta este punto se obtiene la dirección efectiva de la memoria, dirección que sirve para direccionar la memoria. Debe notarse, que el concepto sugiere el cálculo de la dirección efectiva en forma secuencial. En algunos procesadores superescalares el cálculo de la dirección y el acceso al TLB se lleva a cabo en forma paralela, de tal forma que se accede a la memoria con la dirección calculada y el valor obtenido del TLB es usado para comparar la etiqueta del dato en la memoria caché y determinar si es un dato valido (*hit*) o es erróneo (*miss*).

Como en el caso de operaciones realizadas con registros, sería deseable poder ejecutar operaciones de memoria en el menor tiempo posible. Esto significa reducir la latencia de operaciones de memoria, ejecutando múltiples operaciones de memoria en el mismo tiempo y con operaciones que no son de acceso a memoria, y posiblemente permitiendo la ejecución fuera de orden. Sin embargo, debido a que hay más localidades de memoria que registros, y la memoria es direccionada por medio de los registros, no es práctico usar las soluciones descritas en la sección anterior para lograr la ejecución fuera de orden.

Algunos procesadores superescalares sólo permiten una operación de memoria por ciclo, pero esto genera un cuello de botella. Para solucionar simultáneamente los requerimientos de memoria, se ha replicado el número de puertos de lectura y escritura de la memoria. Usualmente, esto funciona para los niveles bajos de la jerarquía de memoria. Esta técnica se logra teniendo múltiples celdas que pueden ser accedidas individualmente dentro de la memoria, al contar con múltiples bancos de memoria [16] [17], o al hacer múltiples requerimientos seriales durante el mismo ciclo [18].

Cada operación que ha sido lanzada a la jerarquía de memoria, puede obtener un acierto (*hit*) o un fracaso error (*miss*) en la memoria caché. La política de manejo de aciertos, errores, y reemplazo de datos difiere en cada una de las implementaciones de la memoria caché.

### Jerarquía de Memoria

La jerarquía de memoria consiste en tener diferentes niveles de almacenamiento de datos, cada uno de diferente tamaño, velocidad y proximidad al procesador. Este concepto parte de principios de localidad temporal y espacial [23] que se explican a continuación.

*Localidad temporal* (localidad en el tiempo): Si un elemento es referenciado, volverá a ser referenciado pronto. Esto se entiende que si un dato ha sido requerido de la memoria, hay una alta probabilidad de que vuelva a ser requerido o consultado, razón por lo que se importante tener presente donde se localiza este dato o moverlo a una posición cercana al procesador para tener un acceso mucho mas rápido a éste.

*Localidad espacial* (localidad en el espacio): Si un elemento es referenciado, los elementos cuyas direcciones están próximas tendrán que ser referenciados pronto. Esto es, la mayoría de los programas presentan un comportamiento secuencial, lo que da entender que si se accede a un dato en la memoria es muy probable que los datos contiguos también sean requeridos en la ejecución del programa.

Estas dos características de la localidad de datos (temporal y espacial), se implementan en la memoria de las computadoras como una jerarquía de memoria. Una jerarquía de memoria consta de varios niveles con diferentes velocidades y tamaños. Las memorias más rápidas son más caras por bit, que las memorias más lentas y, por tanto, habitualmente son más pequeñas. La memoria principal se implementa a partir de DRAM (memoria dinámica de acceso aleatorio), mientras que los niveles mas próximos al procesador (cachés) utilizan SRAM (memoria estática de acceso aleatorio). Las DRAM son menos costosas por bit que las SRAM, aunque son mucho más lentas.

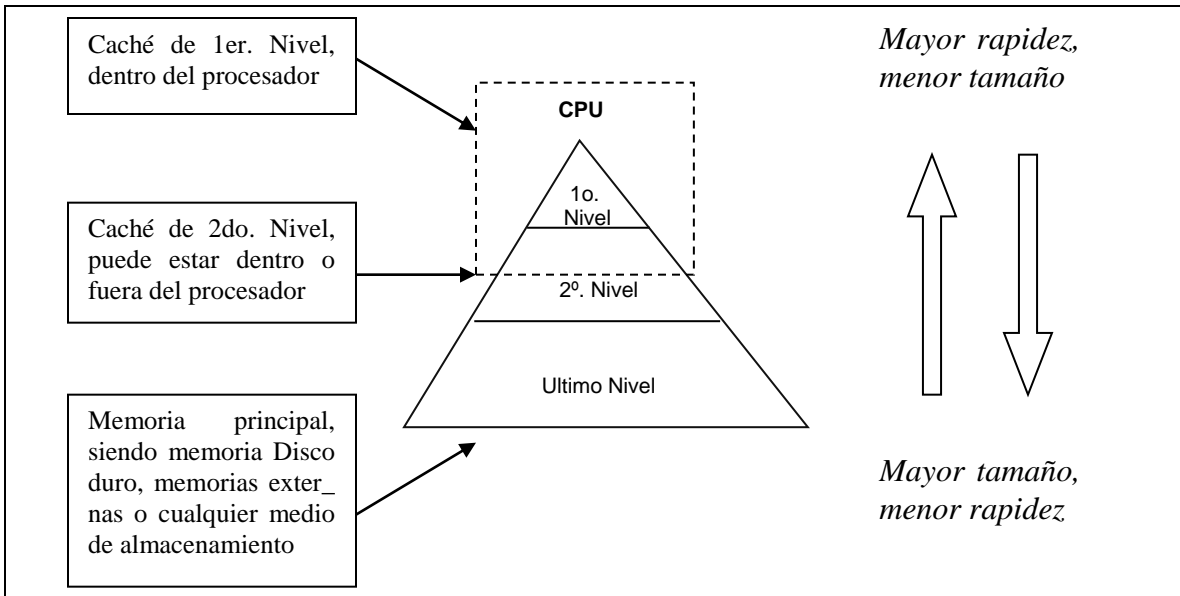


Fig. 1.6 Jerarquía de Memoria.

Debido al costo y tiempo de acceso de los diferentes tipos de memoria, es conveniente colocar la memoria más rápida cerca del procesador y la más lenta y de mayor tamaño en la última escala de la jerarquía de memoria, quedando lo más lejos del procesador. El sistema de memoria está organizado como una jerarquía: un nivel más cercano del procesador es un subconjunto de cualquier nivel más apartado, y todos los datos están almacenados en el nivel más bajo.

En una jerarquía de memoria se utilizan las memorias más pequeñas y más rápidas cerca del procesador. Por tanto, los accesos que son aciertos en el nivel superior de la jerarquía pueden procesarse rápidamente. Los accesos que son fallos van a los niveles inferiores de la jerarquía, que son mayores pero más lentos. Si la tasa de aciertos es suficientemente alta, la jerarquía de memoria tiene un tiempo de acceso cercano al de los niveles más altos (y más rápidos) y un tamaño igual al del nivel más bajo (y mayor).

### 1.2.4.1 Memorias Caché

La caché se compone de Memorias Estáticas de Acceso Aleatorio, en inglés “*Static Random Access Memory*” (SRAM), y para explicar su funcionamiento empezaremos con el ejemplo de una petición del procesador de un dato hacia la memoria Caché y a esto surgen las siguientes dos preguntas: ¿cuando sabemos si un dato está en la Caché?, y de ser así ¿cómo lo encontramos?, las respuestas a estas preguntas están relacionadas. Si cada dato o palabra puede estar en una posición exacta entonces sabemos como encontrarla si está en la caché.

La forma más sencilla de asignar a cada palabra de la memoria una posición en la caché es asignar la posición de la caché en base a la dirección que tiene la palabra dentro de la memoria. Esta estructura de caché se denomina de *correspondencia directa (directa mapeo)*, ya que cada posición de la memoria le corresponde una posición en la caché, obviamente sabemos que la memoria principal es de un tamaño mucho mayor que el de la memoria caché, por lo que varias direcciones de la memoria principal pueden radicar en una misma dirección dentro de la memoria caché.

Esto se puede ejemplificar de la siguiente manera, si uno tiene una dirección conformada por 4 bits para la memoria principal, este dato puede ser ubicado dentro de la memoria caché con tan solo los primeros 3 bits, lo que indica que dos direcciones de la memoria principal pueden radicar en la memoria caché en la misma dirección. Como en cada posición de la caché se puede encontrar el contenido de diferentes posiciones de memoria, ¿cómo sabemos si el dato solicitado corresponde con el que se encuentra en la caché?, esto se resuelve al adicionar etiquetas (*tags*) a la caché. Las etiquetas contienen la información necesaria para identificar si una palabra de la caché corresponde al dato o palabra buscada.

También se requiere de un mecanismo para identificar si un bloque de la caché no tiene información válida. Por ejemplo, cuando el procesador arranca la memoria caché se encuentra vacía, y los campos de etiqueta no tienen sentido. El procedimiento más común



es añadir un bit de validez (*Valid*) para identificar si una entrada contiene una dirección válida. Si el bit no está inicializado (a 1) no puede haber coincidencia en esta dirección.

Hasta este punto se ha descrito los componentes que conforman un bloque en una memoria caché, además de mencionar la forma en que se pueden ubicar los datos para una memoria caché de correspondencia directa. Una misma dirección en la caché puede albergar diferentes datos o asociarse a diferentes direcciones de la memoria principal en diferentes etapas de tiempo y uno a la vez. Esto se considera como una característica de asociatividad, es decir, se puede replicar el módulo completo de la memoria caché con las mismas direcciones de la caché y albergando diferentes datos en cada una de esas mismas direcciones.

Haciendo uso del ejemplo de una dirección de 4 bits de la memoria principal, donde sólo se usan los 3 bits menos significativos para direccionar la memoria caché, pueden haber dos datos diferentes para una misma dirección; si se manejan dos módulos idénticos de la caché, cada uno de los bloques almacena los diferentes datos que pueden ubicarse en la misma dirección de la caché.

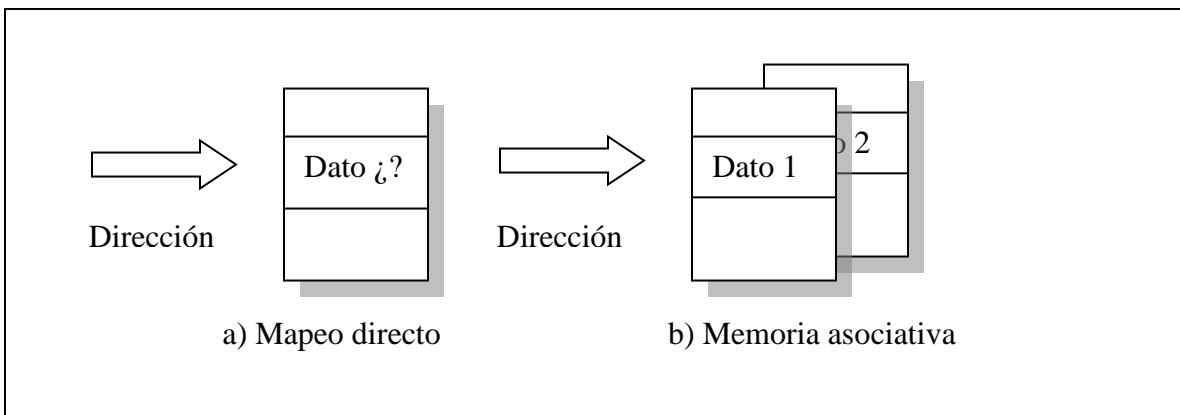


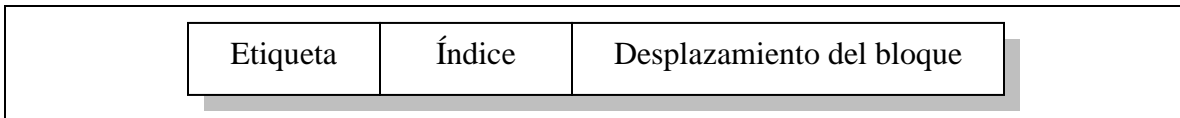
Fig. 1.7 Direccionamiento de la memoria caché

En una memoria caché asociativa por conjuntos, hay un número fijo de posiciones (como mínimo dos) donde puede ubicarse cada dato; una caché asociativa por conjuntos de  $n$  posiciones para un dato se denomina, una caché asociativa por conjuntos de  $n$  vías.

Para localizar un dato en una caché asociativa se hace una búsqueda en cada uno de los módulos o bloques que conforman a la memoria, esto es, se busca el dato en dos, tres o más posibles ubicaciones para un mismo dato. La forma en como se realiza la búsqueda depende de la política de búsqueda implementada en cada caso, sin embargo, la tendencia general es que la búsqueda se realice en forma simultánea para todos los bloques y al final comparar las etiquetas de cada uno de los datos con la solicitada, de esta forma se localiza el dato buscado. La ventaja de incrementar el grado de asociatividad es que habitualmente disminuye la tasa de fallos, ya que el dato permanece en la memoria caché al tener varias localidades donde ubicarse y no salir de la memoria por ser reemplazado con un dato mas reciente, por otro lado, esto también incrementa la latencia de accesos a la memoria siendo factor importante en el detrimento del rendimiento del procesador.

### 1.2.4.2. Localización de datos

En una memoria caché de correspondencia directa, indexamos la caché para encontrar el dato de interés. En una memoria caché asociativa, por conjuntos se analiza la etiqueta que lleva cada dato, que indica la dirección del mismo. La etiqueta de cada dato de la caché que puede contener la información deseada es comprobada para ver si corresponde a la dirección del dato que solicita el procesador. La figura 1.8 ilustra como se descompone la dirección. El valor índice se utiliza para seleccionar el conjunto que contiene la dirección de interés, y las etiquetas de todos los datos que deben ser buscados. Como la velocidad es la esencia, todas las etiquetas del conjunto seleccionado se buscan en forma paralela. Una búsqueda en serie de las etiquetas haría un tiempo de acceso muy grande.



**Fig. 1.8** Campos de la dirección de memoria

Si el tamaño total no cambia, al incrementar la asociatividad; aumenta el número de bloques por conjunto, que es el número de comparaciones simultáneas necesarias para realizar la búsqueda en paralelo, cada incremento de un factor de dos en la asociatividad multiplica el número de bloques por conjunto y divide a la mitad el número de conjuntos. Por consiguiente, cada incremento de la asociatividad en un factor de dos disminuye en un bit el tamaño del índice e incrementa en un bit el tamaño de la etiqueta. En una caché totalmente asociativa, sólo hay un conjunto, y todos los bloques de datos deben ser comprobados en paralelo. Así, no hay índice, y la dirección entera, excluyendo el desplazamiento de bloque, se compara con la etiqueta de cada bloque. En otras palabras, buscamos en toda la caché sin ninguna indexación.

Una elección entre correspondencia directa, asociativa por conjuntos, o totalmente asociativa en cualquier jerarquía de memoria dependerá del costo de un fallo frente al costo de implementar la asociatividad, en tiempo y en partes adicionales. Muchos sistemas han utilizado cachés de correspondencia directa debido a sus ventajas en tiempo de acceso y simplicidad. La ventaja en el tiempo de acceso se presenta, porque encontrar cada bloque requerido no depende de una comparación. Las cachés grandes nunca utilizan ubicación totalmente asociativa, debido al costo y penalización de tiempo en aciertos.

### 1.2.4.3. Políticas de reemplazo

Cuando se presenta un fallo en una caché asociativa, debemos decidir que bloque reemplazar. En una caché totalmente asociativa, todos los bloques son candidatos para el reemplazo. Si la caché es asociativa por conjuntos, debemos escoger entre los bloques del conjunto. Por supuesto, la situación es fácil en una caché de correspondencia directa, porque solo hay un candidato.

Hay dos estrategias principales empleadas para seleccionar el bloque a reemplazar.

- *Aleatoria*: los bloques candidatos son seleccionados aleatoria mente.
- *Menos recientemente utilizado (LRU)*: el bloque reemplazado es el que hace más tiempo que no se ha utilizado.

Una virtud de la selección aleatoria es su facilidad para construirlo en hardware. Cuando el número de bloques a gestionar aumenta, la estrategia LRU se hace enormemente cara y, en la práctica, es solo aproximada. En una caché asociativa por conjuntos de dos vías, el reemplazo aleatorio tiene una tasa de fallos de aproximadamente 1,1 veces mayor que el reemplazo LRU. El reemplazo LRU muestra una mayor ventaja con mayores grados de asociatividad, pero también es difícil implementarla.

### 1.2.4.4 Tipos de fallos

En este nivel de la jerarquía de memoria todos los fallos se pueden clasificar en las siguientes categorías:

- *Fallos compulsorios (o forzosos)*: El primer acceso a un bloque no está en la caché, así que el bloque debe de ser traído a la caché. Estos también se denominan fallos de arranque frío.
- *Fallos de capacidad*: Si la caché no puede contener todos los bloques necesarios durante la ejecución de un programa, los fallos de capacidad se presentarán debido a bloques que se descartan y recuperan mas tarde.
- *Fallos de conflicto*: Si la estrategia de ubicación de bloques (o datos) es asociativa por conjuntos o correspondencia directa, los fallos de conflicto, además de los forzosos y de capacidad, se presentarán, porque un bloque puede ser descartado y mas tarde recuperado si a un conjunto le corresponden demasiados bloques. Estos fallos también se denominan fallos de colisión.

Estas fuentes de fallos pueden ser tratadas de forma directa cada uno, cambiando algún aspecto del diseño de la caché. Puesto que los fallos de conflicto surgen directamente de la contención para el mismo bloque de caché, una ubicación totalmente asociativa solucionaría este problema, sin embargo esto trae como consecuencia que el tiempo de acceso sea mucho mayor.

Los fallos de acceso pueden reducirse fácilmente ampliando la caché; en realidad, las cachés han estado creciendo durante los últimos años. Por supuesto, cuando se hace la caché más grande se incrementa el tiempo de acceso llegando a afectar el rendimiento global del procesador.

Como los fallos forzosos (compulsorios) son generados por la primera referencia a un bloque, la principal forma para que el sistema de caché reduzca el número de fallos es incrementar el tamaño del bloque. Esto reducirá el número de referencias requeridas para tocar cada bloque del programa una vez, por que el programa constara de menos bloques

de caché. Incrementar demasiado el tamaño del bloque puede tener un efecto negativo en el rendimiento, a causa del incremento en la penalización de fallos.

En los diseños reales de la caché, interactúan muchas opciones de diseño y cambiar una característica de la caché afectará con frecuencia factores como la tasa de fallos, rendimiento o tiempo de acceso (tabla 1.1).

Cambio de diseño	Efecto en la tasa de fallos	Posible efecto de rendimientos negativos
Aumento de tamaño	Disminuyen los fallos de capacidad	Puede incrementar el tiempo de acceso
Aumento de asociatividad	Disminuye la tasa de fallos debido a fallos de conflicto	Puede incrementar tiempo de acceso
Aumento de tamaño de bloque	Disminuye la tasa de fallos para un amplio rango de tamaños de bloques	Puede incrementar tiempo de acceso

**Tabla 1.1** Relación tamaño/asociatividad vs efecto.

La dificultad en el diseño de la jerarquía o sistema de memoria con las mismas prestaciones que el procesador, se están acentuando cada vez más [24] [25]. Como la velocidad del procesador continua incrementándose en forma continua y mas rápido que los tiempos de acceso a la memoria caché, la memoria está incrementándose en un factor que limita el rendimiento. La dificultad en el diseño del sistema de memoria para acercar esta diferencia de crecimiento entre el procesador y la memoria, es que todas las posibilidades de diseño tienen un efecto positivo y negativo sobre el rendimiento. Si este es el caso ¿Cómo podemos superar la diferencia de crecimiento entre las velocidades del procesador y la memoria caché? Esta pregunta es parte de la cuestión que atañe a esta investigación y tema de tesis.

### 1.2.5 Retiro de instrucciones

La etapa final en el tiempo de vida de una instrucción, es la etapa de retiro de ésta, donde el efecto de la instrucción es permitido para modificar el estado del proceso lógico. El propósito de esta etapa es implementar la apariencia de una ejecución secuencial del programa, cuando por supuesto, se realiza de otra forma.

Las acciones necesarias para esta etapa depende del método empleado para recobrar la secuencialidad del programa. Algunos de los métodos son los siguientes:

En el primer método, el estado de la máquina a sido salvado o guardado hasta cierto punto, en un *buffer* histórico [19] [20]. Las instrucciones actualizan el estado de la máquina al momento de que son ejecutadas y cuando un estado preciso es necesario, es decir, cuando se realiza ejecución de operaciones en forma especulativa (*Branch prediction*) y se detecta un error en la predicción se recupera el estado antes de la ejecución especulativa del *buffer* histórico.

El segundo método consiste en separar el estado de la máquina en dos: el estado de la implementación físico y el estado de la parte lógica. La parte física es actualizada conforme las operaciones se completan. La parte lógica se actualiza en forma secuencial al programa. Con esta técnica la ejecución especulativa es mantenida en el “*reorder buffer*”(ROB); al retirar una instrucción, su resultado es movido del *reorder buffer* hacia los registros arquitecturales o lógicos (y a la memoria en el caso de operaciones de almacenamiento), y un espacio es liberado en el *reorder buffer*.

Como ya se explicó, el ROB contiene la parte de control de interrupciones además de la información que se requiere en la etapa de renombramiento de registros, de tal forma, que realiza el control de la tabla de mapeo de registros y de la tabla de registros libres, lo que significa que al momento de retirar una instrucción el “reorder buffer” se actualiza y a su vez actualiza la tabla de mapeo y la de registros físicos libres.

Aunque el método de un *buffer* histórico ha sido usado en varios procesadores superescalares, el uso del ROB se ha hecho más común al paso de los últimos años, en adición de que provee un estado preciso del procesador (condición en la que se encuentran cada una de las instrucciones que se están procesando en las diferentes etapas del camino de procesamiento), el ROB ayuda a implementar la función de renombramiento de registros.

### 1.3 El papel del software

Pensando en el incremento de la velocidad de ejecución de un programa binario para una arquitectura superescalar, estos programas pueden ser optimizados por medio del software. Los nuevos binarios pueden usar el mismo conjunto de instrucciones y el mismo modelo de ejecución secuencial que los viejos programas, pero la diferencia radica en el orden en que las instrucciones están acomodadas. El software puede asistir en la creación de estos nuevos programas al incrementar la eficiencia en las etapas de búsqueda (*fetching*), lanzamiento (*issue*) y ejecución (*execute*) de instrucciones. Esto se puede lograr mediante dos formas:

- (i) Incrementando la probabilidad que un grupo de instrucciones que son consideradas para ser lanzadas lo hagan de forma simultáneamente, y
- (ii) Decrementar la probabilidad de que una instrucción tenga que esperar el resultado de una instrucción previa cuando está siendo considerada para ejecución.

Ambas técnicas pueden ser completadas al agendar u ordenar las instrucciones en forma estática. Ordenar las instrucciones en grupos que son fácilmente ejecutables de forma paralela, se ayuda a obtener un aprovechamiento óptimo de los recursos con los que cuenta el procesador. Otra forma de lograr una mayor ejecución de instrucciones en forma paralela, es colocar lejos las instrucciones (hacia delante) que generan un dato que ha de ser utilizado por otras instrucciones, de tal forma se evitan las dependencias entre instrucciones, ya que se calcula este dato mucho antes de ser solicitado.

El soporte que da el software al alto rendimiento de los procesadores es un tema demasiado amplio, que sea abordado en forma más detallada en la bibliografía [21].

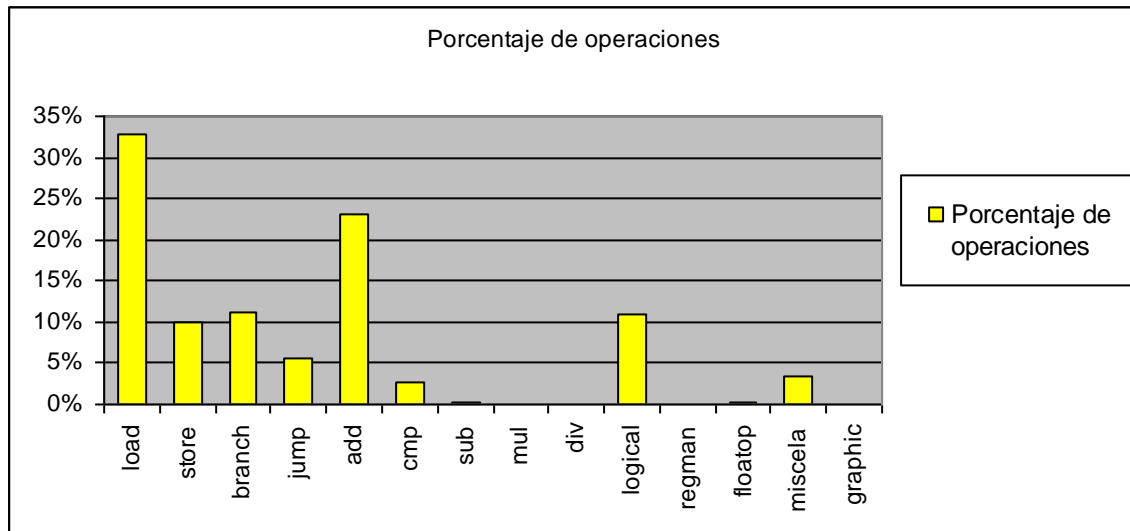
### 1.4 Operaciones repetitivas en un procesador superescalar

En procesadores superescalares de uso general, son variadas las cargas de trabajo a las que se puede ver sometido y debido a ello, varía las operaciones mas frecuentes a lo largo de la ejecución del programa o carga de trabajo. El tipo de operaciones que realiza un procesador súper escalar, sin importar la carga de trabajo que tenga, se pueden clasificar de la siguiente forma:

Tipo de instrucción	Descripción	Recursos
Load	Operación de referencia a memoria, en el que se realiza una carga de un valor traído desde una de las memorias	Unidad funcional ALU, registro fuente y destino, memoria cache
Store	Operación de referencia a memoria, en el que se realiza un almacenamiento de un valor generado	Unidad funcional ALU, registro fuente y destino, memoria cache
Branch	Operación de salto condicionado a una comparación	Unidad funcional ALU, registro fuente, dirección de salto, actualización de registro PC
Jump	Operación de salto incondicional	Unidad funcional ALU, registro fuente, dirección de salto, actualización de registro PC
Add	Operación de suma del valor almacenado en registros o con un valor inmediato	Unidad funcional ALU, registro destino y registros fuentes
Cmp	Operación de comparación entre registros o con un valor inmediato	Unidad funcional ALU, registro destino y registros fuentes
Sub	Operación de resta del valor almacenado en registros o con un valor inmediato	Unidad funcional ALU, registro destino y registros fuentes
Mul	Operación de multiplicación del valor almacenado en registros o con un valor inmediato	Unidad funcional de multiplicación/división, registros destino y fuentes
Div	Operación de división del valor almacenado en registros o con un valor inmediato	Unidad funcional de multiplicación/división, registros destino y fuentes
Logical	Operación de tipo lógico con el valor almacenado en registros o con un valor inmediato	Unidad funcional ALU, registro destino y fuentes
Regman	Manipulación de registros	Banco de registros, unidad funcional ALU
Floatop	Operaciones de punto flotante	Unidades funcionales especiales para operaciones de punto flotante, registros especiales
Miscela	Operaciones propias de la arquitectura del procesador (llamadas de subrutinas, interrupciones, etc)	Módulos propios de la arquitectura
Graphic	Operaciones con registros especiales dedicados al tratamiento de imágenes	Módulos propios de la arquitectura

**Tabla 1.2** Clasificación de instrucciones

La gráfica de la figura 1.9 muestra los porcentajes de los tipos de instrucciones de una carga de trabajo, según la tabla 1.2



**Fig. 1.9** Porcentaje de instrucciones promedio en una carga de trabajo típica

La forma de obtener esta gráfica fue simular varios y diferentes tipos de carga de trabajo, simulando su ejecución en un procesador superescalar con la ayuda de la herramienta de simulación SimpleScalar Toolset [22]. Los resultados de la figura 1.9 son el promedio de todas las cargas de trabajo. Las cargas de trabajo utilizadas para estos resultados se detallan en el siguiente capítulo.

La gráfica resalta un mayor número de operaciones de carga (*load*), de suma (*add*), lógicas (*logical*), y en menor grado de almacenamiento (*store*), brinco condicional e incondicional (*branch* y *jump*). Esto representa que una parte muy significativa de las instrucciones hacen referencia al sistema de memoria, representando una carga de trabajo grande para la memoria cache de datos. Además las instrucciones de tipo suma y lógico, requiere hacer uso del banco de registros.

Las operaciones de referencia a memoria se manejan en una cola de instrucciones diferente al resto de los demás tipos de instrucciones. Por esta razón atenderemos en un capítulo por separado el análisis de operaciones redundantes en la jerarquía de memoria, sobre todo enfocándose en la memoria caché de datos.

La ventana de instrucciones es el punto donde se lleva a cabo la asignación de recursos disponibles y la solución de dependencias de datos para agilizar la ejecución de operaciones. Por ello dedicamos otro capítulo a la ventana de instrucciones practicando el mismo análisis que a la memoria caché de datos, al buscar comportamientos de tipo redundante y planteando modelos que dan solución a este comportamiento.

A este respecto es posible modelar en forma matemática el tiempo de ejecución de los programas o cargas de trabajo, de tal forma que es posible determinar la forma de impactar el rendimiento del procesador y el por que de enfocarse en la memoria cache de

datos y la ventana de instrucciones. En la siguiente sección se hace un análisis matemático de esta problemática.

### 1.5 Análisis y modelado matemático del tiempo de ejecución

El tiempo de ejecución de un programa esta dado por el número total de ciclos de CPU necesitados para ejecutar un programa,  $N_{Total}$ , y el tiempo que dura el ciclo de CPU,  $t_{CPU}$ . Una forma de comparar la importancia de la variación aparentemente independiente de los parámetros es la relación entre ellos para minimizar el tiempo de ciclo,  $T_{LI}$ , y como consecuencia el tiempo de ejecución. En realidad, la falta de continuidad entre estas variables es debido al lazo existente entre tiempo de ciclo de la memoria caché y el tiempo de ciclo del CPU. Un cambio en la organización de la memoria cache podría o no afectar el tiempo de ciclo del procesador, dependiendo de los caminos críticos en el diseño. Claramente, si el tiempo de la caché no esta determinado por tiempo de ciclo del CPU, cualquier cambio en su organización puede mejorar el conteo total de ciclos en forma genérica, proporcionando un nuevo tiempo de ciclo de caché menor o igual que un tiempo de ciclo de CPU sin cambio. Desde que los obstáculos en la cache son claros y poco interesantes cuando el tiempo de ciclo de caché es menor que le tiempo de CPU, podemos consistentemente asumir que *el tiempo de ciclo de sistema es determinado por la caché. El tiempo de ciclo de CPU y del sistema son por consecuencia parte mínimo del tiempo de ciclo de la caché*, y los términos son usados como sinónimos.

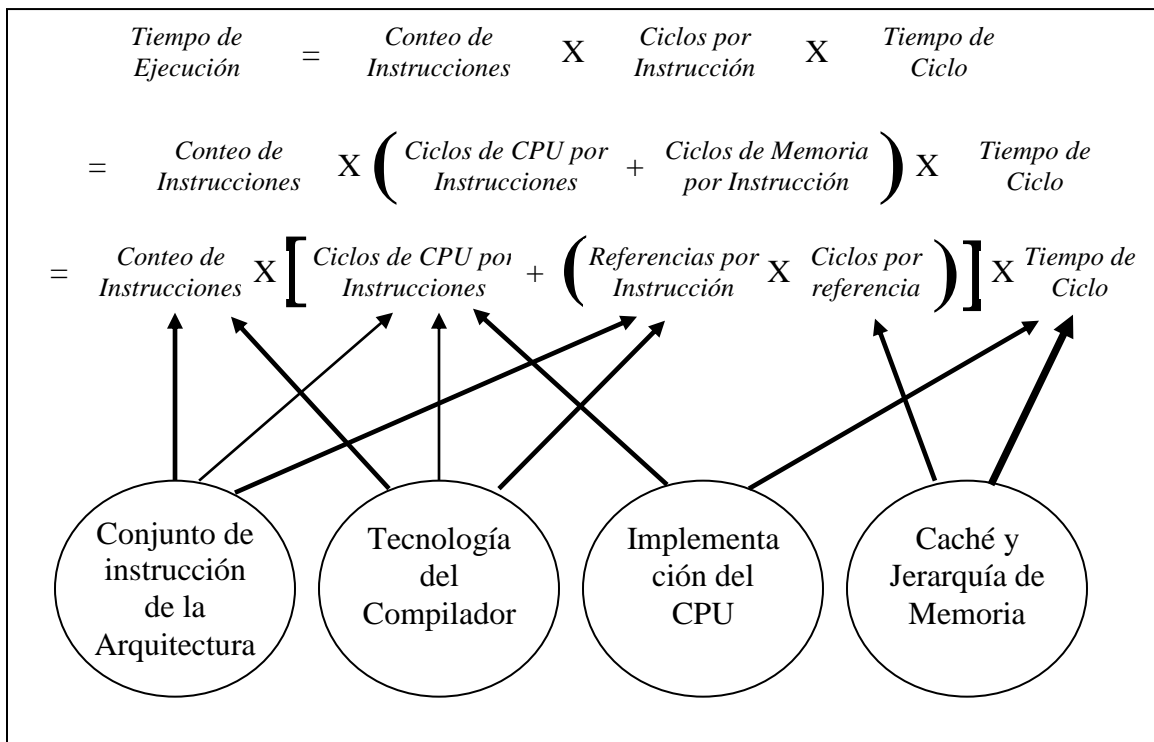


Fig. 1.10 Tiempo de ejecución



Para un sistema de memoria con un nivel de caché, la cuenta total de ciclos es, en primer orden, una función de la velocidad de memoria y el porcentaje de error de la cache. El porcentaje de error de una memoria caché  $C$  esta dado por  $m(C)$ . Las características primarias de la organización de la memoria que determinan el porcentaje de error de la caché son el tamaño de la caché,  $C$ , su nivel de asociatividad,  $A$ , el número de conjuntos,  $S$ , y el tamaño de los bloques,  $B$ . Las opciones de las estrategias de búsqueda y escritura introducen otros factores en la ecuación del porcentaje de error, pero esto puede ser ignorado temporalmente para lograr un concepto claro. En consecuencia, el porcentaje de error esta dado como una función de estos cuatro parámetros.

$$m(C) = f(S, C, A, B) \quad [1]$$

Es generalmente aceptado que cuando se incrementa alguno de estos parámetros, el porcentaje de error disminuye, pero a partir de cierto punto, ocasiona que se incremente ligeramente. El tamaño de bloque es el único entre todos los parámetros mencionados en que hay un tamaño de bloque que minimiza el porcentaje de error. Para bloques grandes, el incremento de tamaño a lo largo afecta en forma negativa el porcentaje de error.

La porción del ciclo total medido atribuible a la memoria principal,  $N_{MM}$ , depende de las características de la memoria, el tamaño del bloque, y sorprendentemente, el tiempo de ciclo de CPU. Esta última relación resulta del factor de que algunos de los atributos de la memoria son típicamente especificados en segundos. El mas significativo de estos es la latencia ( $LA$ ) entre el inicio de la búsqueda de la memoria y el inicio de la transferencia de datos solicitados. El número de ciclos desperdiciados por la espera en la memoria principal depende de la latencia expresada en ciclos, denotada por  $la$ , el cual es dado por el valor mas alto del porcentaje de latencia expresado en segundos y el tiempo de ciclo. La otra característica dominante de la memoria es el porcentaje en el cual los datos pueden ser transferidos hacia y desde la memoria principal. El porcentaje de transferencia,  $tr$  en *words* por ciclo y  $TR$  en *words* por segundo, es actualmente el mas pequeño de los tres porcentajes o tasas: la tasa en la cual la memoria puede presentar un dato, la velocidad en la cual la cache acepta el dato, y la tasa máxima de transferencia determinada por las características físicas y eléctricas del diseño (material y construcción). El tiempo utilizado en transferir un dato en una sencilla búsqueda es denominado periodo de transferencia. Esto es igual al porcentaje o tasa del tamaño de búsqueda en *words* y la tasa de transferencia.

La función que relaciona los parámetros de la memoria y la caché en un conteo total de ciclos, en primer orden, es lineal en el porcentaje de error, latencia y periodo de transferencia.

$$N_{Total} = g(m(C), B, LA, TR, t_{CPU}) \quad [2]$$

La relación entre el tiempo de ciclo y los parámetros de la caché es la mas difícil de cuantificar, primeramente por que esta es extremadamente dependiente en los niveles más bajos de la implementación. La declaración más fuerte que puede ser hecha es que el tiempo de ciclo es invariante, aunque no en un sentido totalmente estricto, en cada uno de

los cuatro parámetros básicos: tamaño, asociatividad, número de conjuntos y tamaño de conjunto:

$$t_{CPU} = h(C, S, A, B) \quad [3]$$

El tiempo de ciclo mínimo es también influenciado por la complejidad de la lógica de control utilizada en las políticas de búsqueda, escritura y reemplazo. La poca predicción de esta función,  $h$ , ocasiona que se dificulte determinar cual es la configuración óptima de la caché sin realizar un número de diseños individuales para acercarse a un grado de detalle.

### 1.5.1 Aproximación Analítica

La motivación detrás del siguiente análisis es examinar la naturaleza de las ecuaciones 1, 2, 3, y así relacionar el comportamiento del sistema de niveles a los varios parámetros de la memoria y caché.

El tiempo total de ejecución es el producto del tiempo de ciclo y el conteo total de ciclos:

$$T_{Total} = t_{CPU} \times N_{Total} = t_{LI} (C) \times N_{Total} \quad [4]$$

Recordando que el tiempo de CPU esta dado por el tiempo de ciclo de la caché, el cual es una función de sus parámetros organizacionales.

Desde que el tiempo de ejecución es una función convexa de las variables organizacionales y temporales, el tiempo de ejecución mínimo es obtenido cuando la derivada parcial con respecto a algunas variables es igual a cero. Por ejemplo, si el tamaño de caché es el centro de atención, la organización óptima es una que iguale las magnitudes del cambio relativo en el tiempo de ciclo y el conteo de ciclos:

$$\frac{1}{t_{LI}} \times \frac{\delta t_{LI}}{\delta C} = - \frac{1}{N_{Total}} \times \frac{\delta N_{Total}}{\delta C}$$

Muchos de los parámetros de la caché no son continuos: estos están restringidos a valores enteros o binarios. Para estas variables discretas, la ecuación balanceada cercana tiene una forma diferente – una que iguala el cambio relativo en el tiempo de ciclo y el conteo de ciclos a través de los cambios de una organización de la caché a una adyacente.

$$\frac{1}{t_{LI}} \times \frac{\Delta t_{LI}}{\Delta C} = - \frac{1}{N_{Total}} \times \frac{\Delta N_{Total}}{\Delta C} \quad [5]$$

Dado un cambio particular de una organización u otra, si el lado izquierdo y el lado derecho de la ecuación son iguales, luego cambio es de comportamiento neutral. Si, sin embargo, el lado izquierdo de la ecuación es mayor (cercano a  $+\infty$ ), luego entonces el cambio incrementa sobretodo el tiempo de ejecución, y si el lado derecho de la ecuación es mas positivo, hay una ganancia neta en rendimiento.

Una estimación comúnmente utilizada para el conteo total de ciclos,  $g( m( C ), B, LA, TR, t_{CPU} )$ , es la suma de ciclos utilizados en la espera en cada uno de los niveles de la jerarquía de memoria. Para un nivel sencillo o un sólo nivel de caché, el número total de ciclos esta dado por el número de ciclos utilizados en referencias a memoria no medidas; el número de ciclos utilizados en realizar instrucciones de búsqueda, carga y almacenamiento: además el tiempo utilizado de espera en la memoria principal.

$$\begin{aligned}
 N_{Total} = & N_{Execute} + N_{Ifetch} + N_{Ifetch} \times m( C ) \times \check{n}_{MMread} \\
 & + N_{Load} + N_{Load} \times m( C ) \times \check{n}_{MMread} \\
 & + N_{Store} \times \check{n}_{L1write}
 \end{aligned} \tag{6}$$

donde

$N_{Execute}$	Número de ciclos utilizados en referencias a memoria no medidas
$N_{Ifetch}$	Número de instrucciones buscadas en el programa o traza
$N_{Load}$	Número de cargas
$N_{Store}$	Número de almacenamientos
$\check{n}_{MMread}$	Promedio de números de ciclos utilizados en satisfacer un error de lectura de caché
$\check{n}_{L1write}$	Promedio de números de ciclos utilizados en el comportamiento de escritura de caché

Nosotros podemos aplicar unos cuantos conceptos aceptados sin perder la globalización del problema. Para una máquina RISC con un ciclo de ejecución sencillo, el número de ciclos en el cual ninguna referencia de flujo es activa,  $N_{Execute}$ , es cero. Las ecuaciones de arriba asumen que la referencia de lectura (instrucciones de búsqueda y carga) toman un ciclo cuando son un acierto en la caché. Esto también asume que hay un solo puerto en una sola memoria, por lo que las instrucciones de búsqueda y carga están serializadas. Mediante notar que el porcentaje de error de lectura de la caché es el promedio de las instrucciones de caché y el porcentaje de error de lectura de datos, nosotros podemos colapsar todas las instrucciones de búsqueda y carga paralelas en un termino común. Esto nos deja una forma más simple, en la cual el conteo total de ciclos es la suma de ciclos utilizados en realizar lecturas y escrituras. Luego para cada lectura, hay un ciclo sencillo de máximo valor, además de las penalizaciones por error en la caché por referencias.

$$N_{Total} = N_{Read} ( 1 + m_{Read}( C ) \times \check{n}_{MMread} ) + N_{Store} \times \check{n}_{L1write} \tag{7}$$

El principal tiempo para recuperar un bloque de la memoria principal,  $\check{n}_{MMread}$ , tiene dos componentes: el tiempo actual para buscar un bloque y, ocasionalmente, algunos retardos debido a que la memoria esta ocupada al momento de la petición. Si no hay prebúsqueda o trafico del *bus* de *I/O*, la única fuente posible para retardo es una escritura en progreso. El promedio de la cantidad de retardos por referencia de memoria es denotada por  $\check{n}_{ReadDelay}$ . El número de ciclos utilizados en la búsqueda de un bloque es la suma de la

latencia del periodo,  $la$ , y el periodo de transferencia,  $B/tr$ . Cuando tratamos con obstáculos que envuelven el tamaño de bloque, es frecuentemente necesario asumir que el promedio de retardo en la memoria dado por una escritura en progreso,  $\check{n}_{ReadDelay}$ , es pequeño con respecto a la suma de los periodos de latencia y transferencia. En la práctica, este es generalmente el caso si la estrategia de escritura es efectiva y los ciclos utilizados de la espera en la memoria principal no dominan el tiempo de ejecución.

$$\begin{aligned}
 N_{Total} &= N_{Read} [1 + m_{Read}(C) X (la + B/tr + \check{n}_{ReadDelay})] \\
 &\quad + N_{Store} X \check{n}_{LIwrite} \\
 &\approx N_{Read} [1 + m_{Read}(C) X (la + B/tr)] \\
 &\quad + N_{Store} X \check{n}_{LIwrite}
 \end{aligned}$$

Similarmente el número de ciclos para completar una escritura,  $\check{n}_{LIwrite}$ , incluye el tiempo para realizar la escritura y cualquier ciclo adicional necesario para poner el dato en el *buffer* de escritura. Este último componente también incluye cualquier tiempo de espera para escribir en el *buffer* cuando esta lleno. Para una caché de escritura en línea, el dato escrito siempre requiere ser colocado en el *buffer*, cuando para una caché de postescritura, esta potencial penalización ocurre solamente cuando un bloque no utilizado esta siendo retirado. Si una lectura causa el reemplazo de un bloque no valido, se asume que le tiempo necesario para reemplazarlo es ocultado por el tiempo de búsqueda en la memoria, contabilizado en un ciclo de lectura. Pero, para cachés de postescritura el principal número de ciclos para realizar una escritura,  $\check{n}_{LIwrite}$ , puede ser aproximado por el número de ciclos necesarios para realizar una escritura en la caché,  $n_{LIwrite}$ , cuando la frecuencia de postescrituras ocasionada por escrituras es baja.

La ecuación 6 asume una cache simple unificada y el lanzamiento serial de instrucciones y referencias de datos; ambos conceptos contribuyen simétricamente al tiempo de ejecución. Para una máquina con un tiempo de ejecución sencillo y una organización tipo Harvard, capaz de generar instrucciones y referencias de datos paralelos. Como consecuencia, las escrituras solamente se suman al tiempo de ejecución si estas toman un lapso mayor a un ciclo, o si ellas incurren en un retardo por escritura en el *buffer* o postescritura. La base del conteo total de ciclos es el número de instrucciones buscadas.

$$\begin{aligned}
 N_{Total} &= N_{fetch} + N_{fetch} X m(C_{LI}) X \check{n}_{MMread} \\
 &\quad + N_{Load} X m(C_{LID}) X \check{n}_{MMread} \\
 &\quad + N_{Store} X (\check{n}_{LIwrite}^{-1}) \tag{8}
 \end{aligned}$$

La ecuación 7 puede ser usada en lugar de la 8 para esta clase de sistemas si el número de lecturas,  $N_{Read}$ , el porcentaje de error de lectura,  $m_{Read}(C)$ , y el número principal de ciclos por escritura,  $\check{n}_{LIwrite}$ , son definidos apropiadamente. El número de lecturas es definido a ser el número de instrucciones buscadas, el porcentaje de error de lectura es el

porcentaje de error de la caché de instrucciones mas el porcentaje de error de la caché de datos soportado por la tasa de número de instrucciones buscadas y cargadas, y el principal tiempo de escritura es uno menor que el tiempo para realizar la escritura.

$$N_{Read} = N_{ffetch}$$

$$m_{Read} = m( C_{LI1} ) + \frac{N_{Load}}{N_{ffetch}} X m( C_{LI1D} )$$

Similarmente, si el número de ciclos libres de cualquier actividad de memoria,  $N_{Execute}$ , no es cero, como en una máquina CISC, el modelo representado por la ecuación 7 puede aun ser usado sin modificarlo si el número de lecturas y la tasa de de error de lecturas son localmente definibles.

Para encontrar el tiempo de ejecución, el conteo total de ciclos debe de ser multiplicado por el tiempo de duración del ciclo. El tiempo de duración del ciclo tiene una función de un tiempo de ciclo,  $t_{LI}$ , el porcentaje de error,  $m( C )$ , el principal tiempo de acceso de lectura de la memoria principal,  $t'_{MMread}$ , el promedio de número de ciclos para retirar una escritura,  $\check{n}_{LIwrite}$ , y el número de escrituras y lecturas:

$$T_{Total} = N_{Read} X t_{LI} + N_{Read} X m( C ) X t'_{MMread} + N_{Store} X \check{n}_{LIwrite} X t_{LI} \quad [9]$$

Cuando tratamos con una jerarquía de memoria caché multinivel, promedio de tiempo para satisfacer una petición de lectura en cada uno de los niveles es una función del tiempo de acceso de lectura de la caché, tasa de error y penalización por error. Cada penalización de error es exactamente el tiempo más significativo utilizado para satisfacer una lectura en el siguiente nivel en la jerarquía de memoria. Esta recursión se extiende hasta todos los niveles de la jerarquía, donde la penalización por error en la caché está dada por el tiempo de acceso a lectura de la memoria principal. Todo este tiempo de acceso de lectura,  $n_{Li}$ , esta expresado en ciclos de CPU.

$$N_{Total} = N_{Read} (n_{L1} + m_{L1}( C ) X (n_{L2} + m_{L2}( C ) X (n_{L3} + \dots + m_{Ln}( C ) X \check{n}_{MMread} ) \dots )) + N_{Store} X \check{n}_{LIwrite}$$

El porcentaje de error global,  $M_{Li}$ , esta definido como el número de lecturas erróneas en el nivel  $i$  dividido por el número de referencias de lectura generados por el CPU. Expresando esto en términos, el conteo total de ciclos llega a depender de la suma de los productos del porcentaje de error global de caché y el próximo tiempo de ciclo de la caché.

$$\begin{aligned}
 N_{Total} = N_{Read} ( &n_{L1} + M_{L1}(C_{L1}) \times \check{n}_{L2} + M_{L2}(C_{L2}) \times \check{n}_{L3} \\
 &+ \dots + M_{Ln}(C_{Ln}) \times \check{n}_{MMread} ) \\
 &+ N_{Store} \times \check{n}_{L1write}
 \end{aligned} \tag{10}$$

Estas ecuaciones relacionan las tasas de error de la cache al conteo total de ciclos y, consecuentemente, al tiempo de ejecución. Sin embargo, para propiamente investigar el lazo entre los parámetros organizacionales y el tiempo de ejecución, nosotros necesitamos investigar la tasa de error como una función dada por los parámetros de la cache, expresada como la función  $f(S, C, A, B)$ .

El modelo de Agarwal esta basado en la descomposición de los errores en diferentes clases. Un error no estacionario es inevitablemente causado por las referencias de memoria de un programa en su inicio. Intrínsecamente los errores producidos por hacer referencia a una misma localidad de memoria. Si el número de distintas direcciones mapeadas en un dado conjunto es mayor que el tamaño del conjunto, entonces estas competirán entre si por el espacio disponible, chocando entre si ocasionando que queden fuera de la memoria caché. Extrínsecamente los errores son causados por los efectos de la multiprogramación: entre espacios de tiempo, un programa tendrá bloqueados algunos conjuntos mientras que otro requiere hacer uso de los datos en esos bloques. En esta forma completa, el porcentaje de error o tasa cubre el  $t$  considerablemente de las  $\tau$  referencias dadas como la suma de tres términos, uno para cada tipo de error:

$$\begin{aligned}
 m(C(S, A, B), t) = &\frac{u(B)}{\tau t} \left[ 1 + \frac{(U - u(B)) \times (t - 1)}{Tu(B)} \right] && \text{Errores no estacionarios} \\
 &+ \frac{c(A)}{\tau} \left[ u(B) - \sum_{a=0}^A S a P(a) \right] && \text{Errores intrínsecos} \\
 &+ \frac{f}{\tau t} \left[ \frac{t - 1}{t_s} \right] \sum_{a=0}^A S a P(a) && \text{Errores extrínsecos}
 \end{aligned}$$

La tasa de error esta definida en términos del número de bloques solamente tocados en el fragmento de referencias,  $u(B)$ , el largo total de la traza,  $T$ , el número total de bloques tocados en la traza,  $U$ , la tasa de colisión de la traza,  $c(A)$ , y la probabilidad de que a bloques de un  $u(B)$  mapee en el mismo conjunto,  $P(a)$ . La tasa de colisión esta dentro del intervalo de 0 y arriba de  $\tau/u$ .

Assumiendo un reemplazo aleatorio, la probabilidad,  $P(a)$ , esta dada por una distribución binomial, la cual puede ser aproximada por una distribución Poisson para caché grandes (una  $S$  grande) y un número grande de referencias por evento (una  $u$  grande). Finalmente el porcentaje de error también depende de la fracción de los bloques,  $f(t)$ , perteneciente a

un proceso dado, que son purgados por otros procesos en espacios de tiempo consistentes. Para una caché de mapeo directo, la fracción puede ser aproximada a una simple forma exponencial del número de bloques,  $u(B)$ , y el número de bloques utilizados por otros procesos en sistemas multiprogramados,  $u'(B)$ :

$$f(S, B, t) = 1 - e^{-u(B)/S} (1 + u'(B)/S)$$

Para una caché de mapeo directo ( $A = 1$ ), todas las sumatorias pueden ser eliminadas, dejando cada uno de los tres términos fácilmente diferenciable:

$$m(C(S, A = 1, B), t) = \frac{u(B)}{\tau t} \left[ 1 + \frac{(U - u(B)) X(t-1)}{Tu(B)} \right]$$

$$+ \frac{cu(B)}{\tau} \left[ 1 - e^{-u(B)/S} \right]$$

$$+ \frac{1}{\tau t} \left[ \frac{t-1}{t_s} \right] u(B) e^{-u(B)/S} \left[ 1 - e^{-u(B)/S} (1 + u'(B)/S) \right]$$

De lo anterior se puede entender que el tiempo de ejecución de una carga de trabajo esta directamente relacionado con le tiempo que se requiere para acceder al sistema de memoria, siendo la memoria principal, memoria cache de datos, banco de registros también almacenado como parte de la memoria cache y la ventana de instrucciones al ser tomado como una pequeña memoria de instrucciones. Por esta razón si de disminuye el tiempo de acceso a estos elementos se influye en forma proporcional en el rendimiento del procesador. En el siguiente capítulo se trata la forma, metodología y herramientas utilizadas en la evaluación de una arquitectura superescalar y sus posibles modificaciones en pro de un mejor rendimiento.

# Capítulo 2

## Metodología de evaluación

### 2.1 Resumen

En este capítulo se detallan las herramientas utilizadas para implementar, validar y comparar cada uno de los modelos propuestos en los dos capítulos siguientes, como lo es la selección de las cargas de trabajo. La implementación de los modelos propuestos requiere la comprensión y manipulación de las herramientas existentes, así como su modificación desde su nivel de programación. La parte correspondiente a la validación de resultados se logra mediante la comparación de las métricas obtenidas del comportamiento normal de un procesador bajo cargas de trabajo. La comparación se realiza al someter a los modelos propuestos a las mismas cargas de trabajo con las que se obtienen las métricas de comportamiento del procesador.

### 2.2 Herramientas de simulación

La metodología empleada para evaluar nuestra propuesta se basa principalmente en la simulación. Los arquitectos de computadoras necesitan herramientas que les permitan variar los parámetros de los diseños propuestos hasta obtener el rendimiento esperado en el producto terminal, evitando de esta forma la necesidad de construir prototipos que finalmente no funcionen como se esperaba. Los microprocesadores modernos son dispositivos de ingeniería muy compleja y cada vez más difíciles de evaluar, por lo tanto es necesario contar con herramientas de simulación que permitan emularlos para mejorar su diseño y rendimiento conforme los requerimientos de los avances tecnológicos.

Con la simulación se evita la pérdida de tiempo y dinero que se requiere para desarrollar un prototipo. Si el prototipo no funciona como se esperaba, es necesario reconstruirlo, dedicando mucho más tiempo del que se esperaba con todas las complicaciones que esto ocasiona. En cambio, si se quiere variar algún parámetro en el simulador, basta con solo modificar algunas líneas de código. Existen varios tipos de simuladores y de igual forma se cuenta con diferentes técnicas de simulación, por ejemplo, la simulación dirigida por traza, la dirigida por emulación, la dirigida por ejecución, por mencionar solo algunas. Estos difieren en complejidad de acuerdo a las estadísticas que se requieren de la simulación.

#### 2.2.1 Resumen de simuladores de arquitecturas de microprocesadores

Existe una amplia variedad de simuladores con arquitecturas completas y parciales. Cada uno de estos simuladores difiere en lo flexible de su código, el soporte técnico, y visualización, así como en su nivel de aplicación. Para nuestro caso de estudio, lo dividiremos en tres categorías, haciendo referencia al nivel de conocimiento del usuario al que están enfocados.



- *Simuladores para principiantes.*
- *Simuladores intermedios.*
- *Simuladores avanzados.*

A continuación se describe cada una de las categorías y en cada una de ellas se anexa un cuadro sinóptico con los simuladores existentes.

### *Simuladores para principiantes*

Esta clase de simuladores están enfocados a estudiantes que no cuentan con conocimientos sobre arquitecturas de microprocesadores y solo requieren una simple introducción. Estos simuladores pretenden representar las ideas básicas de la organización de un procesador o algunos pequeños detalles de complejidad. Algunas características de estos simuladores son:

- Visualización de los componentes y su interacción.
- Un lenguaje de conjunto de instrucciones simplificado.
- Accesibilidad y uso sencillo.

El principal objetivo de estos simuladores es dirigir a los estudiantes al uso de simuladores de tipo intermedio y avanzado.

Nombre	Descripción
Babbage's Analytical Engine <a href="http://www.fourmilab.ch/babbage/applet.html">http://www.fourmilab.ch/babbage/applet.html</a> (John Walker/Fourmilab Switzerland)	Un applet de Java que emula el mecanismo de una computadora llamada <i>Analytical Engine de Charles Babbage</i> .
CASLE (Compiler/Architecture Simulation for Learning and Experience) <a href="http://shay.ecn.purdue.edu/~casle/">http://shay.ecn.purdue.edu/~casle/</a> (Purdue University)	Un script CGI y un simulador a nivel de lenguaje de máquina. La memoria y los registros lógicos son visibles conforme se ejecuta la carga de trabajo.
CPU/Disk Subsystem <a href="http://www.cis.ufl.edu/~fishwick/CPUDisk/">http://www.cis.ufl.edu/~fishwick/CPUDisk/</a> (Paul Fishwick/U Florida)	Simulación basada en diseño Web, que representa con un servidor a un procesador y con otros servidores representa múltiples discos.
CPU SIM <a href="http://www.cs.colby.edu/~djskrien/">http://www.cs.colby.edu/~djskrien/</a> (Dale Skrien/Colby College)	Simulador de CPU basado en el procesador MC680X0 o Power PC.
CPU Simulator applet <a href="http://www.cs.gordon.edu/courses/cs111/module6/cpusim/cpusim.html">http://www.cs.gordon.edu/courses/cs111/module6/cpusim/cpusim.html</a> (Irvin Levy/Gordon College)	Simulador visual, que representa los principales componentes de la arquitectura de un procesador, junto con sus señales.
EasyCPU <a href="http://www.cteh.ac.il/departments/education/cpu.html">http://www.cteh.ac.il/departments/education/cpu.html</a> (Cecile Yehezkel/Weizman Institute of Science)	Conjunto de ventanas basado en el conjunto de instrucciones del procesador Intel 80X86 con modo básico y avanzado.
Little Man Computer <a href="http://www.acs.ilstu.edu/faculty/javila/lmc/">www.acs.ilstu.edu/faculty/javila/lmc/</a> (Bill Yurcik and Larry Brumbaugh/Illinois State U.)	Simula el modelo Little Man Computer de Madnick (1965) del MIT. Ensamble código máquina y lo simula paso a paso. Estado de registros visible.
Simple Computer <a href="http://beachstudios.com/sc/">http://beachstudios.com/sc/</a> (Beach Studios) (utilizado en Saddleback College y Cal State Fullerton)	Presenta una ventana en la que se visualiza el estado de los registros, acumulador, IR (Instruction Register), PC (Program Counter) y la memoria.
CPU Simulator for Windows <a href="http://www.spasoft.co.uk/cpusim.html">http://www.spasoft.co.uk/cpusim.html</a> (SPA Corp.)	Simula al Simple Computer pero de forma animada.

**Tabla 2.1** Simuladores para principiantes.

*Simuladores intermedios*

Estos simuladores están enfocados a estudiantes con cierto grado de conocimiento en la arquitectura de microprocesadores y requieren un simulador que cubra las principales características de un procesador con mayor detalle. Estos simuladores sirven principalmente para ilustrar y enseñar dos conceptos generales: el conjunto de instrucciones de una arquitectura y la microarquitectura de un procesador.

El simulador del conjunto de instrucciones emula las instrucciones que han sido programadas, mostrando el contenido de la memoria, registros lógicos, el registro de instrucciones, y el contador del programa conforme se ejecuta cada instrucción. No se muestran los elementos que se requieren para realizar las instrucciones.

Los simuladores de microarquitecturas se enfocan en desplegar el mecanismo donde la ejecución de cada instrucción se lleva a cabo dentro de la microarquitectura. La mayoría de estos simuladores son gratuitos.

Nombre	Descripción
<b>Simuladores a nivel de instrucciones</b>	
LC2 Simulator <a href="http://www.mhhe.com/patt/">http://www.mhhe.com/patt/</a>	Este simulador representa el procesador presentado por Patt y Patel en su libro <i>Introduction to Computer Systems</i> .
SPIM <a href="http://www.cs.wisc.edu/~larus/spim.html">http://www.cs.wisc.edu/~larus/spim.html</a> (James Larus/U Wisconsin-Microsoft Research)	Un simulador de lenguaje ensamblador para el procesador MIPS (R2000/R3000) con interfase en modo texto y gráfico [11].
SPIMSAL <a href="http://www.cs.wisc.edu/~larus/spim.html">http://www.cs.wisc.edu/~larus/spim.html</a> (James Larus/U Wisconsin-Microsoft Research)	SPIMSAL es una vieja versión de SPIM que corre en Windows 3.12 y Macintosh [10].
THRSim11 <a href="http://www.hc11.demon.nl/thrsim11/thrsim11.htm">http://www.hc11.demon.nl/thrsim11/thrsim11.htm</a> (Harry Broeders/Rijswijk Inst. of Technology)	Un simulador a nivel de instrucciones del microcontrolador de Motorola 68HC11A8, simulando sus registros, puertos, terminales y memoria.
Microprocessor Simulator <a href="http://www.softwareforeducation.com/simulator.htm">http://www.softwareforeducation.com/simulator.htm</a> (Software for Education)	Este es un programa tipo shareware que simula el conjunto de instrucciones del procesador Intel 8086, muy popular en el ámbito académico.
<b>Simuladores de Microarquitecturas</b>	
Mic-1 Simulator <a href="http://www.ontko.com/mic1/">http://www.ontko.com/mic1/</a> (escrito por Ray Ontko y Dan Stone con mejoras de Andrew S. Tanenbaum)	Es un simulador del Mic-1 descrito en el capítulo 4 del libro de texto de Tanenbaum en 1998. Existen versiones para Windows y Unix.
Micro Architecture <a href="http://www.kagi.com/fab/msim.html">http://www.kagi.com/fab/msim.html</a> (Fabrizio Oddone) (used at Earlham College)	Este simulador representa un procesador microprogramado similar al descrito en el libro de Tanenbaum de 1999[16], <i>Structured Computer Organization</i> .
MIPSim <a href="http://mouse.vlsivie.tuwien.ac.at/lehre/rechnerarchitekturen/download/Simulatoren/">http://mouse.vlsivie.tuwien.ac.at/lehre/rechnerarchitekturen/download/Simulatoren/</a> (Institut fur Technische Informatik)	Un simulador de un procesador con pipeline, basado en el libro de Hennesy y Patterson en 1996 [11], <i>Computer Organization and Design</i> .

**Tabla 2.2** Simuladores intermedios

*Simuladores avanzados*

Estos simuladores están enfocados a estudiantes con un conocimiento significativo de arquitectura de microprocesadores. Este conjunto de simuladores refleja el estado del arte del diseño de procesadores, implementando los modelos de procesadores actuales y algunos de ellos son empleados como herramientas de investigación para el diseño de nuevas arquitecturas.

Existen dos tipos de simuladores que pueden ser considerados como simuladores avanzados. Los simuladores de microarquitecturas son más sofisticados que los enfocados a estudiantes de nivel intermedio. Estos simuladores incorporan la complejidad de procesadores de alto rendimiento, por ejemplo detalles de ejecución fuera de orden y nivel de paralelismo de instrucciones. El otro tipo de simuladores avanzados se enfoca en el comportamiento lógico de una arquitectura, subdividiéndose en transistores, compuertas lógicas y circuitos digitales.

Nombre	Descripción
<b>Simuladores de micro arquitecturas avanzados</b>	
DLXsim <a href="http://www.mkp.com/books_catalog/ca/hp2e_res.htm">www.mkp.com/books_catalog/ca/hp2e_res.htm</a> (DLX Processor of [Hennesey and Patterson 1996] Computer Architecture book)	Un simulador de un procesador DLX, basado en el libro de Hennesey y Patterson en 1996, <i>Computer Organization and Design</i> .
DLXview <a href="http://yara.ecn.purdue.edu/~teamaaa/dlxview/">http://yara.ecn.purdue.edu/~teamaaa/dlxview/</a> (Purdue)	Una modificación del DLXsim que presenta una interfaz gráfica e interactiva.
RSIM <a href="http://www-ece.rice.edu/~rsim/">http://www-ece.rice.edu/~rsim/</a> (Rice)	Este simulador presenta la ventaja de simular dos vertientes de diseño de procesadores: procesadores que explotan el ILP (nivel de paralelismo de instrucciones) y sistemas multiprocesadores con memoria compartida.
SimOS <a href="http://simos.stanford.edu/">http://simos.stanford.edu/</a> (Stanford)	Es un ambiente de simulación, que recrea la interacción entre CPU, caches, buses, discos duros, ethernet, consolas y otros dispositivos.
SimpleScalar <a href="http://www.simplescalar.org/">http://www.simplescalar.org/</a> (Todd Austin, Wisconsin)	Es un conjunto de herramientas: compilador, enlazador, simulador y herramientas de visualización. El simulador más complejo es el denominado <i>sim-outorder</i> .
<b>Simuladores Lógicos</b>	
LogicWorks <a href="http://www.capilano.com/LogicWorks">http://www.capilano.com/LogicWorks</a>	Herramienta para crear, editar y probar circuitos de diseño lógico.
Multimedia Logic Kits <a href="http://www.softronix.com/logic.html">www.softronix.com/logic.html</a>	Sistema visual para diseño lógico de pequeños circuitos.
WinBreadboard <a href="http://www.yoeric.com/">http://www.yoeric.com/</a> (YOERIC software)	Simulador a nivel TTL.

**Tabla 2.3** Simuladores avanzados

**2.2.2 Selección de la Herramienta de simulación**

El simulador necesario para implementar nuestro modelo debe representar arquitecturas actuales en uso, lo que reduce nuestra búsqueda a la categoría de simuladores de nivel avanzado, y del tipo de representación de micro arquitecturas; de la lista presentada en la

tabla 2.3 se seleccionó el conjunto de herramientas que proporciona el SimpleScalar Toolset, y en especial su herramienta denominada “sim-outorder” por las siguientes características:

- Ejecución fuera de orden.
- Predicción de saltos.
- Recuperación de estado después de un error de especulación (error de salto y ejecución posterior en forma especulativa).
- Representación de unidades funcionales con latencias de operación.
- Representación de jerarquía de memoria (caché de datos e instrucciones) con latencias de acceso dependiendo de acierto o fallo en la búsqueda.
- Uso de tablas de traducción para las direcciones de las memorias cache.
- Implementación de un Reorder Buffer y una ventana centralizada de instrucciones en un elemento denominado RUU.
- Entrega de estadísticas de ejecución.
- Fácil manipulación de código (3900 líneas código en lenguaje C).
- Representación de dos arquitecturas: PISA (arquitectura ficticia) y ALPHA (arquitectura basada en los procesadores Alpha).

El simulador “sim-outorder” recrea la simulación de la arquitectura por bloques que realiza una tarea específica. En la figura 2.1 se muestra el seguimiento de cada una de las etapas de acuerdo a las etapas del procesador.

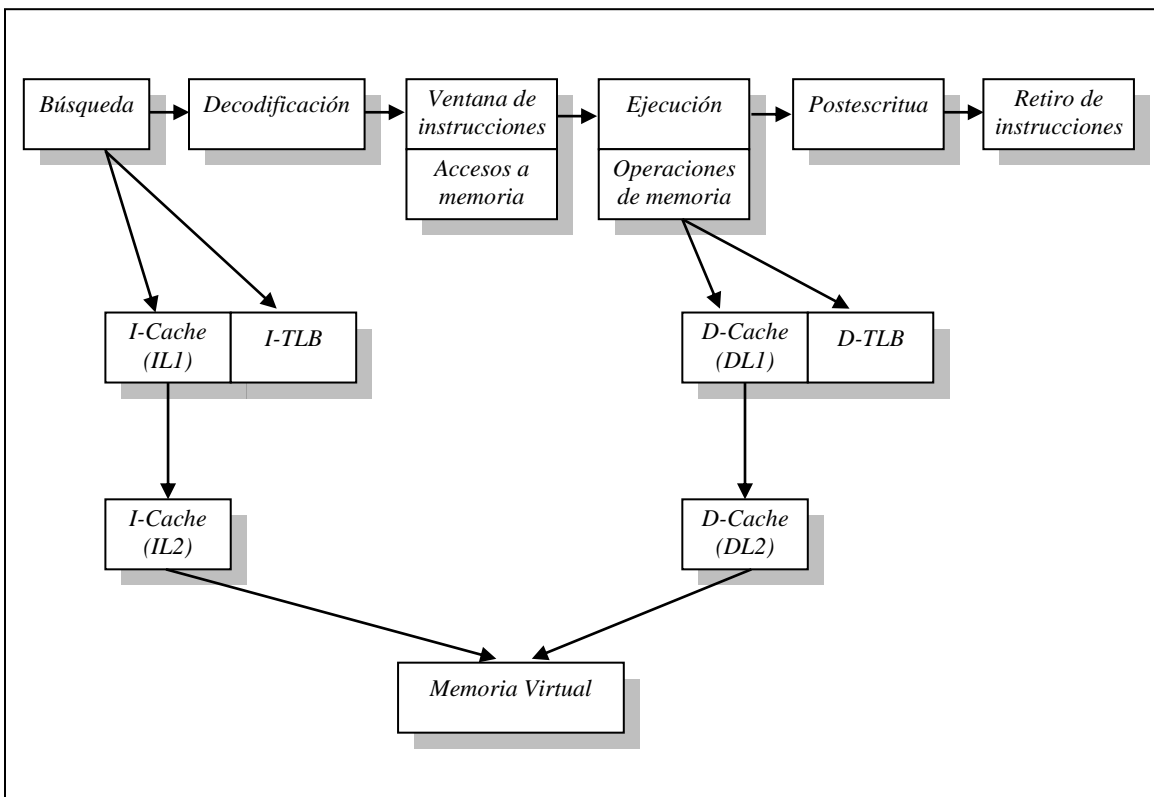


Fig. 2.1 Diagrama esquemático del simulador “sim-outorder” del SimpleScalar

Además de esta herramienta el *set* del SimpleScalar cuenta con las siguientes:

Simulador	Descripción
<i>Sim-Safe</i>	Implementación de un simulador funcional. Es el simulador más sencillo y amigable de todo el conjunto de herramientas.
<i>Sim-Fast</i>	Simulador funcional, el más veloz del conjunto de instrucciones, no verifica posibles errores en la simulación, por lo que no es 100% confiable.
<i>Sim-Profile</i>	Simulador funcional, que presenta mensajes sobre el tipo de instrucción que se esta ejecutando en el momento ( <i>profiling</i> ).
<i>Sim-Cache</i>	Simulador funcional de una memoria caché. Las estadísticas son acordes al tipo de memoria y configuración que da el usuario al programa.
<i>Sim-Cheetah</i>	Simulador funcional de una memoria tipo Cheetah, desarrollada por Rabin Sugumar y Santoch Abraham, la cual puede simular eficientemente de forma simultánea caché de mapeo directo, asociativas y completamente asociativas.

**Tabla 2.4** Conjunto de simuladores de SimpleScalar

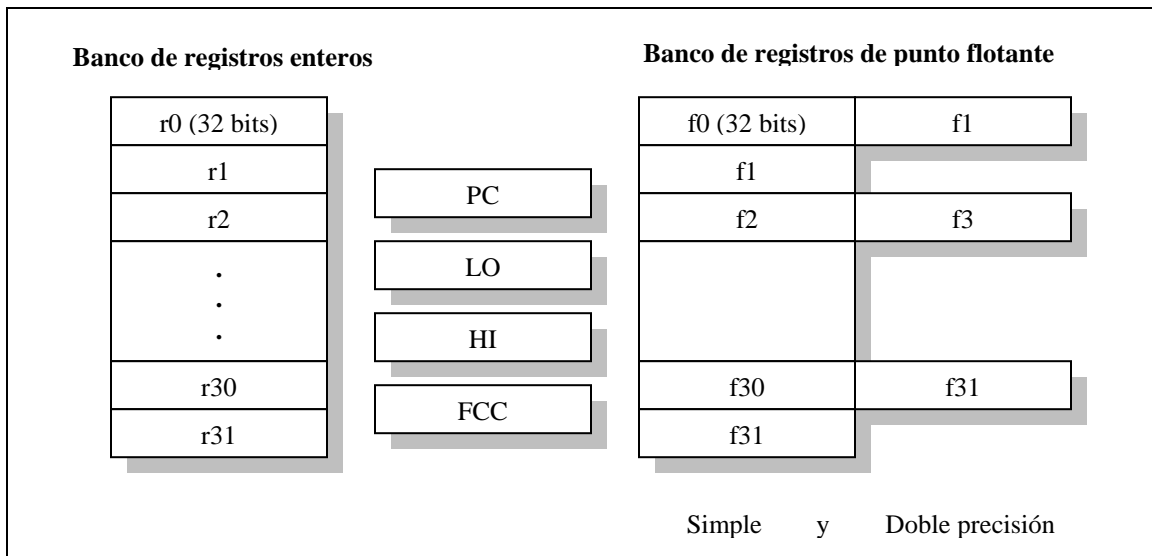
Todos los simuladores presentados en la tabla 2.4, o por lo menos sus partes esenciales, se hayan en el simulador *sim-outorder*, permitiendo una simulación confiable y completa de la operación y comportamiento de procesadores superescalares.

El “*sim-outorder*” simula un procesador con las siguientes opciones de la arquitectura:

<i>-fetch:ifqsize &lt;size&gt;</i>	Tamaño del <i>buffer</i> de búsqueda de instrucciones.
<i>-fetch:mplat &lt;cycles&gt;</i>	Latencia (ciclos) de un error de predicción de salto.
<i>-bpred &lt;type&gt;</i>	Especifica el tipo de predictor de saltos.
<i>-decode:width &lt;insts&gt;</i>	Ancho de banda del decodificador de instrucciones.
<i>-issue:width &lt;insts&gt;</i>	Ancho de banda de instrucciones lanzadas a ejecutar.
<i>-issue:inorder</i>	Ejecución de instrucciones en orden.
<i>-issue:wrongpath</i>	Número de instrucciones permitidas que se lanzan después de un error de predicción o especulativo.
<i>-ruu:size &lt;insts&gt;</i>	Tamaño de la ventana de instrucciones (ROB y RS).
<i>-lsq:size &lt;insts&gt;</i>	Capacidad de la cola de instrucciones de memoria.
<i>-cache:d11 &lt;config&gt;</i>	Configuración de caché de datos de nivel 1.
<i>-cache:d11lat &lt;cycles&gt;</i>	Latencia de acierto de caché de datos de nivel 1.
<i>-cache:d12 &lt;config&gt;</i>	Configuración de caché de datos de nivel 2.
<i>-cache:d12lat &lt;cycles&gt;</i>	Latencia de acierto de caché de datos de nivel 2.
<i>-cache:i11 &lt;config&gt;</i>	Configuración de caché de instrucciones de nivel 1
<i>-cache:i11lat &lt;cycles&gt;</i>	Latencia de acierto de caché de instrucciones de nivel 1.
<i>-cache:i12 &lt;config&gt;</i>	Configuración de caché de instrucciones de nivel 2.

-cache:il2lat <cycles>	Latencia de acierto de caché de instrucciones de nivel 2.
-cache:flush	Actualiza todas las caché s con llamadas al sistema.
-cache:icompress	Mapea dirección de 64 bits con su equivalente de 32.
-mem:lat <1st> <next>	Especifica latencia de acceso a memoria.
-mem:width	Ancho de bus de memoria (bits).
-tlb:itlb <config>	Configuración del TLB de instrucciones.
-tlb:dtlb <config>	Configuración de TLB de datos.
-tlb:lat <cycles>	Latencia de servicio de un error en el TLB.
-res:ialu	Número de ALU's para enteros.
-res:imult	Número de unidades para multiplicar/dividir enteros.
-res:memports	Número de puertos de la caché de primer nivel.
-res:fpalu	Número de ALU's para punto flotante.
-res:fpmult	Número de unidades para multiplicar/dividir punto flotante.
-pcstat <stat>	Graba estadísticas de una dirección de programa (texto).
-ptrace <file> <range>	Genera traza de ejecución.

La arquitectura simulada cuenta con un tamaño de instrucción de 64 bits, 32 registros para enteros (32 bits c/u), 32 registros para números de punto flotante de simple precisión (32 bits c/u) o 16 registros de punto flotante de doble precisión (64 bits c/u), y cuatro registros especiales (*Program Counter o PC, LO, HI y FCC*)



**Fig. 2.2** Bloques de registros enteros y de punto flotante de la arquitectura simulada por “sim-outorder”

La implementación del simulador esta conformado por varios módulos, los cuales se muestran en la figura 2.3. Cada uno de los componentes esta formado por un archivo de cabecera (\*.h) y un archivo de implementación (\*.c) en lenguaje C. El módulo “*Bpred*” corresponde a la implementación del predictor de saltos, teniendo varios tipos de predictor disponibles. El módulo “*Resource*” cuenta con las rutinas para manejo de recursos como son unidades funcionales y la ventana de instrucciones. El módulo “*EventQ*” tiene la implementación de rutinas para el manejo de las colas de la

arquitectura (colas de instrucciones carga y almacena, instrucciones en espera de un operando, instrucciones retiradas del ROB). El módulo “*Loader*” es el primero en tener contacto con el programa de carga que se le asigna al simulador, ya que se encarga de actualizar los valores en la memoria virtual simulada. El módulo “*Regs*” cuenta con la implementación de los registros de la arquitectura. El módulo “*Memory*” tiene las rutinas de manejo del espacio de memoria virtual simulado, además de las operaciones propias del segmento de memoria. El módulo “*Cache*” tiene la implementación de la memoria caché de datos, instrucciones y TLB, además de las rutinas de acceso a dicho elementos y rutinas de actualización. El módulo “*Stats*” tiene el manejo de las estadísticas del simulador (contadores, variables, distribuciones y expresiones).

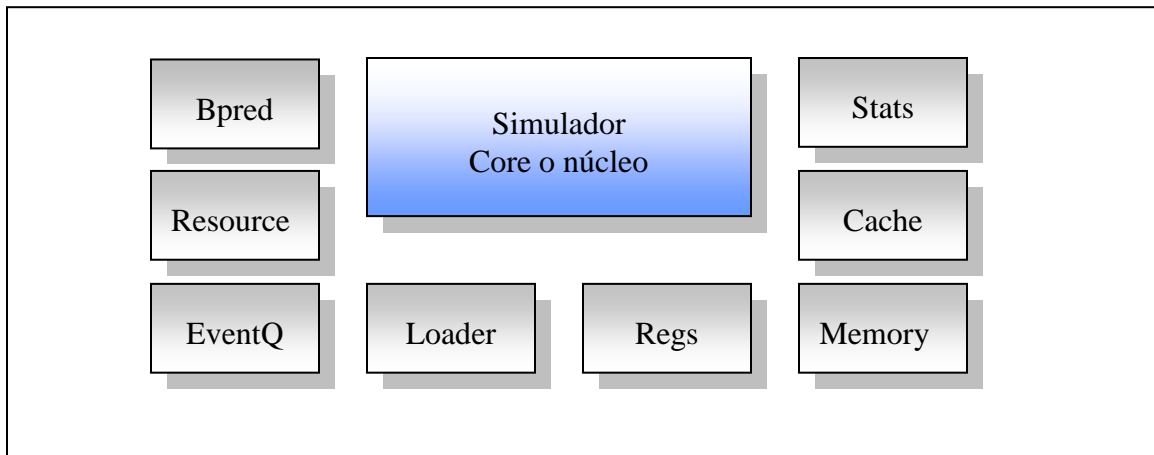


Fig. 2.3 Módulos que conforman el simulador “*Sim-Outorder*”

Cuando se requiere evaluar un diseño, se adecua la herramienta de simulación para que simule una arquitectura propuesta por el diseñador. En el caso de que la modificación tenga impacto en la jerarquía de memoria, los módulos afectados serían los módulos “*Cache*” y “*Memory*”.

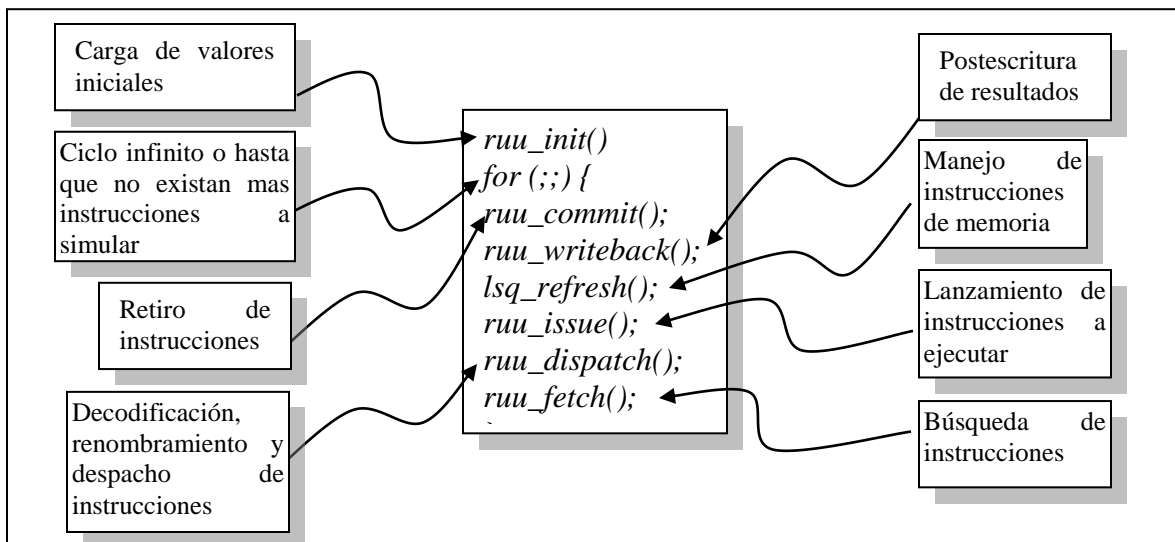


Fig. 2.4 Funciones que emulan las etapas del procesador en el simulador Core o Núcleo.

Sin embargo en el caso de modificar la forma en que se lleva a cabo cada uno de los pasos del *datapath* se requiere modificar el núcleo del simulador representado en la figura 2.3 por el módulo denominado *simulador core*. La figura 2.4 muestra las funciones asignadas a las etapas del procesador.

### 2.2.3 Modificación de la herramienta de simulación

En la sección anterior se explica como está conformada la herramienta “*Sim-Outorder*”, y los módulos que tienen que ser modificados para implementar cambios en la jerarquía de memoria o en alguna de las etapas del procesador. La aplicación de los modelos de esta tesis impactan directamente en la memoria caché de datos de primer nivel y la ventana de instrucciones.

Para realizar las modificaciones al modelo de memoria caché de datos se modificaron los módulos *cache.h* y *cache.c* para simular un comportamiento diferente al convencional. En cuanto a modificaciones sobre la ventana de instrucciones los cambios se efectuaron en el core o núcleo del simulador, archivo *sim-outorder.c*, y archivos anexos como *machine.c*, *machine.h*, *machine.def*, *main.c* y *stats.h*.

### 2.3 Cargas de trabajo (Benchmark)

Un benchmark es un estándar de medición o evaluación para comparar el nivel de rendimiento de componentes y/o sistemas en arquitectura de computadoras. SPEC CPU2000 es un software tipo benchmark producido por *Standard Performance Evaluation Corp.* (SPEC) que incluye a fabricantes de computadoras, integradores de sistemas, universidades, organizaciones de investigación y otros de todo el mundo. Este software está dividido en dos grupos, uno diseñado para medir cargas de trabajo con enteros y otro con número de punto flotante. Los *benchmarks* de tipo entero utilizados se muestran en la tabla 2.5. En esta tabla se pueden observar 11 *benchmarks* de los cuales se tiene la referencia del número de instrucciones y la categoría a la que pertenece cada clase de carga de trabajo. Los *benchmarks* de tipo punto flotante utilizados son mostrados en la tabla 2.6.

Benchmark	Número de instrucciones (REF)	Categoría
<b>Enteros</b>		
Bzip200	143565170182	Compresión de archivos
crafty00	191882992138	Juego de Ajedrez
eon00	80614082986	Visualización por computadora
gap00	269035813916	Interpretador, Grupo de Teoría
gcc00	46917715262	Compilador de Lenguaje C
gzip00	84367396419	Compresión
mcf00	61867464461	Optimización Combinatoria
parser00	546749947290	Procesador de Palabras
twolf00	346485090453	Simulador de Ubicación y Ruta
vortex00	118972498011	Base de Datos orientada a Objetos
vpr00	84068782517	Ruteo y colocación de circuitos FPGA

**Tabla 2.5** Cargas de trabajo de tipo entero (SPEC200INT)



Benchmark	Número de instrucciones (REF)	Categoría
<b>Punto Flotante</b>		
ammp00	326548908872	Química Computacional
applu00	223883654822	Ecuaciones Parciales Diferenciales
apsi00	347922865726	Distribución Polución Metereologica
art00	41798847063	Reconocimiento de Imágenes
equake00	131518587328	Simulación de Propagación Sísmica
facerec00	211026623983	Procesamiento de Imágenes
Fma3d00	268371963517	Simulación de elemento finito
galgel00	409366711473	Dinámica de Fluidos Computacional
lucas00	142398815620	Teoría de Números
mesa00	281694702246	Librería de Gráficos 3D
mgrid00	419156006947	Solucionador Multigrid
sixtrack00	470948915113	Diseño de Acelerador Físico Nuclear
swim00	225830961569	Modelado de ondas de agua
wupwise00	349623858517	Física Quántica Dinámica

**Tabla 2.6** Cargas de trabajo de tipo punto flotante (SPEC200FP)

Para realizar las pruebas a un nuevo modelo de arquitectura se compara su comportamiento con una arquitectura convencional y se le aplican cargas de trabajo representativas (*benchmarks*) para medir su rendimiento; posteriormente se adecua la herramienta de simulación para simular el nuevo diseño y se le aplican los mismos benchmarks. Los resultados de ambas simulaciones, tanto de la arquitectura convencional como la del nuevo diseño, se comparan para medir el rendimiento obtenido con el nuevo diseño, si los resultados no son alentadores, se plantea otro diseño o se realizan modificaciones al diseño propuesto.

Los *benchmarks* se compilaron en un computador basado en el procesador Alpha 21164, con sistema operativo OSF1 V4.0. Para los *benchmarks* de enteros usamos la opción -04 del compilador “cc” propietario. Los *benchmarks* de coma flotante se compilaron con la opción -05 del compilador “fortran” propietario y los resultados corresponden con una ejecución completa de estos.

## 2.4 Plataforma de cómputo

La plataforma de cómputo utilizada para realizar las simulaciones fue un Cluster de 128 nodos del Instituto Mexicano del Petróleo; cada nodo del Cluster cuenta con dos procesadores Intel Pentium 4, que pueden trabajar en forma individual o paralela.

## 2.5 Opciones de configuración para simulación de modelos

La configuración de la arquitectura utilizada para simular los modelos planteados en esta tesis corresponden a la tabla 2.7, de la cual solo se varia los parámetros *ruu:size* y *lsq:size* para modificar el tamaño de la ventana de instrucciones, así como los parámetros *dl1* y *dl2* para modificar la configuración de la memoria cache de datos.

*sim-outorder*: This simulator implements a very detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support. This simulator is a performance simulator, tracking the latency of all pipeline operations.

```
# -config                # load configuration from a file
# -dumpconfig           # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -i                    false # start in Dlite debugger
-seed                   1     # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt                <null> # restore EIO trace execution from <fname>
-nice                   0     # simulator scheduling priority
-max:inst               300000000 # maximum number of inst's to execute
-fastfwd               100000000 # number of insts skipped before timing starts
# -ptrace               <null> # generate pipetrace, i.e., <fname|stdout|stderr> <range>
-fetch:ifqsize         8     # instruction fetch queue size (in insts)
-fetch:mplat           3     # extra branch mis-prediction latency
-fetch:speed           1     # speed of front-end of machine relative to execution core
-bpred                 2lev #                branch                predictor                type
{nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod           2048 # bimodal predictor config (<table size>)
-bpred:2lev            1 16384 14 1 # 2-level predictor config (<l1size> <l2size> <hist_size>
<xor>)
-bpred:comb            1024 # combining predictor config (<meta_table_size>)
-bpred:ras             32    # return address stack size (0 for no return stack)
-bpred:btb             4096 4 # BTB config (<num_sets> <associativity>)
-bpred:spec_update    <null> # speculative predictors update in {ID|WB} (default non-
spec)
-decode:width          4     # instruction decode B/W (insts/cycle)
-issue:width           4     # instruction issue B/W (insts/cycle)
-issue:inorder         false # run pipeline with in-order issue
-issue:wrongpath       true  # issue instructions down wrong execution paths
-commit:width          8     # instruction commit B/W (insts/cycle)
-ruu:size              32    # register update unit (RUU) size
-lsq:size              16    # load/store queue (LSQ) size
-cache:d11             d11:256:32:2:1 # l1 data cache config, i.e., {<config>|none}
-cache:d11lat          2     # l1 data cache hit latency (in cycles)
-cache:d12             ul2:1024:128:4:1 # l2 data cache config, i.e., {<config>|none}
-cache:d12lat          10    # l2 data cache hit latency (in cycles)
-cache:il1             il1:256:32:4:1 # l1 inst cache config, i.e., {<config>|d11|d12|none}
-cache:il1lat          1     # l1 instruction cache hit latency (in cycles)
-cache:il2             dl2   # l2 instruction cache config, i.e., {<config>|d12|none}
-cache:il2lat          10    # l2 instruction cache hit latency (in cycles)
```

**Tabla 2.7** Opciones de configuración para el SimpleScalar

-cache:flush	false	# flush caches on system calls
-cache:icompress	false	# convert 64-bit inst addresses to 32-bit inst equivalents
-mem:lat	100 4	# memory access latency (<first_chunk> <inter_chunk>)
-mem:width	32	# memory access bus width (in bytes)
-tlb:itlb	itlb:64:8192:4:1	# instruction TLB config, i.e., {<config> none}
-tlb:dtlb	dtlb:64:8192:4:1	# data TLB config, i.e., {<config> none}
-tlb:lat	30	# inst/data TLB miss latency (in cycles)
-res:ialu	4	# total number of integer ALU's available
-res:imult	2	# total number of integer multiplier/dividers available
-res:mempport	2	# total number of memory system ports available (to CPU)
-res:fpalu	4	# total number of floating point ALU's available
-res:fpmult	2	# total number of floating point multiplier/dividers available
-pcstat	<null>	# profile stat(s) against text addr's (mult uses ok)
-bugcompat	false	# operate in backward-compatible bugs mode (for testing only)

**Tabla 2.7** Opciones de configuración para el SimpleScalar

En base a estos parámetros es posible obtener las métricas deseadas para validar los modelos propuestos en esta tesis. Los valores que se modifican del archivo de configuración son:

<i>ruu:size</i>	Tamaño de la ventana de instrucciones
<i>lsq:size</i>	Tamaño de la cola de instrucciones de memoria
<i>cache:dll</i>	Tamaño de la memoria cache de datos, nivel de asociatividad y política de reemplazo
<i>cache:dlllat</i>	Latencia de acceso a la memoria caché de datos de primer nivel

En base a estas opciones es posible simular ventanas de instrucciones de diferentes tamaños, así como memorias cache de diferente tamaño, nivel de asociatividad y política de reemplazo de datos de la memoria caché, necesarios para validar los modelos propuestos en esta tesis.

# Capítulo 3

## Ejecución de instrucciones fuera de orden

### 3.1 Introducción

Los procesadores superescalares y los compiladores para estos, normalmente convierten el orden total de las instrucciones como aparece en el programa, en un ordenamiento parcial determinado por las dependencias de datos y de control. Las dependencias de control (las cuales aparecen como saltos condicionales), o la dependencia de datos entre instrucciones, de almacenamiento o de recursos (ítem 1.2.2.1) presentan un mayor obstáculo a la ejecución altamente paralela, debido a que estas dependencias deben ser resueltas antes que todo el bloque subsecuente de instrucciones para saber si es válido. Esto trae como consecuencia que se hagan comparaciones con todos los registros fuentes de las operaciones almacenadas en la ventana, tratando de resolver esta clase de dependencias, las cuales en su mayoría no tienen un resultado satisfactorio, como lo demuestra la siguiente sección.

### 3.2 Comportamiento de la ventana de instrucciones

La ventana de instrucciones es un modelo presentado desde hace varios años, que ha sufrido modificaciones en su funcionamiento de acuerdo a su comportamiento. Es de notar que el tamaño de este *buffer*, se ha incrementado dependiendo del número de instrucciones que es posible decodificar y en su defecto renombrar, con la finalidad de permitir un mayor nivel de paralelismo, al momento de ejecutar las instrucciones. Sin embargo, a pesar de incrementar el tamaño de la ventana, no se ha podido elevar el número de *instrucciones por ciclo lanzadas a ejecutar* (IPC), esto debido a las dependencias que se presentan dentro de la ventana, explicadas en la sección anterior.

La figura 3.1 muestra el comportamiento de la ventana de instrucciones bajo diferentes cargas de trabajo (*benchmark*) del tipo SPEC2000 (enteros). En el que se puede apreciar que a pesar de incrementar el tamaño de la ventana no se tiene una ganancia significativa en el número de IPC.

El IPC, es el número que se obtiene al dividir, el número de instrucciones ejecutadas entre el número de ciclos que se consumieron para ejecutar esas instrucciones, de tal forma que da una proporción del número de instrucciones que fueron lanzadas de la ventana de instrucciones. Al apreciar las gráficas de las figuras 3.1 y 3.2, es de notar que el número máximo de instrucciones que se lanzan por ciclo son tres (15% del total de veces que se lanzan operaciones), cuando se tienen ventanas de instrucciones de tamaño de 32 o 64 entradas con un ancho de banda para lanzar instrucciones de 4 instrucciones por ciclo, lo que representa que se detiene por mas de un ciclo a la mayoría de las instrucciones dentro de la ventana de instrucciones.

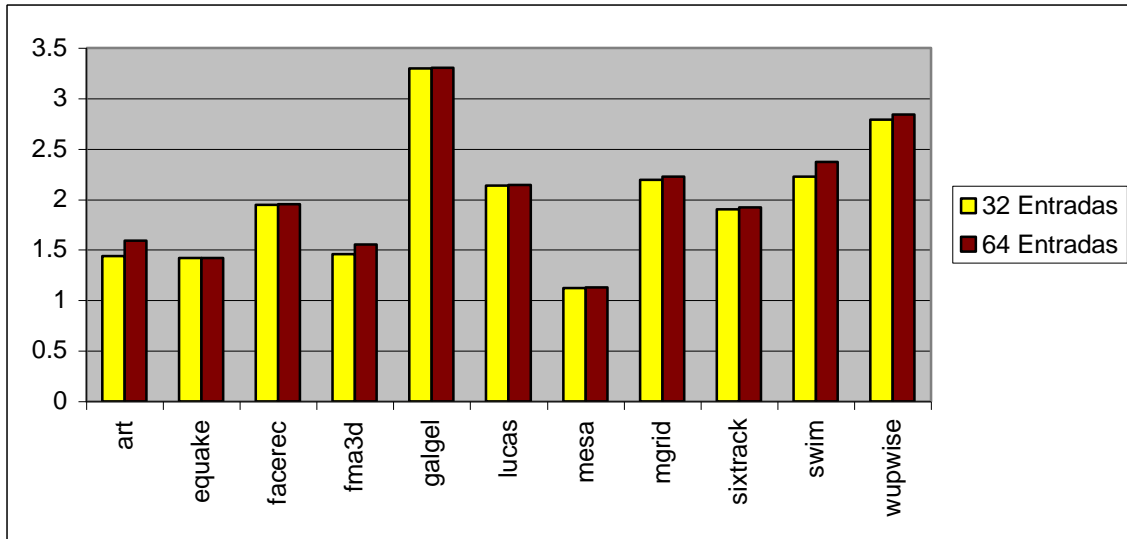


Fig. 3.1 Instrucciones por ciclo, SPECINT (ventana de 32 entradas ■ , 64 entradas ■ )

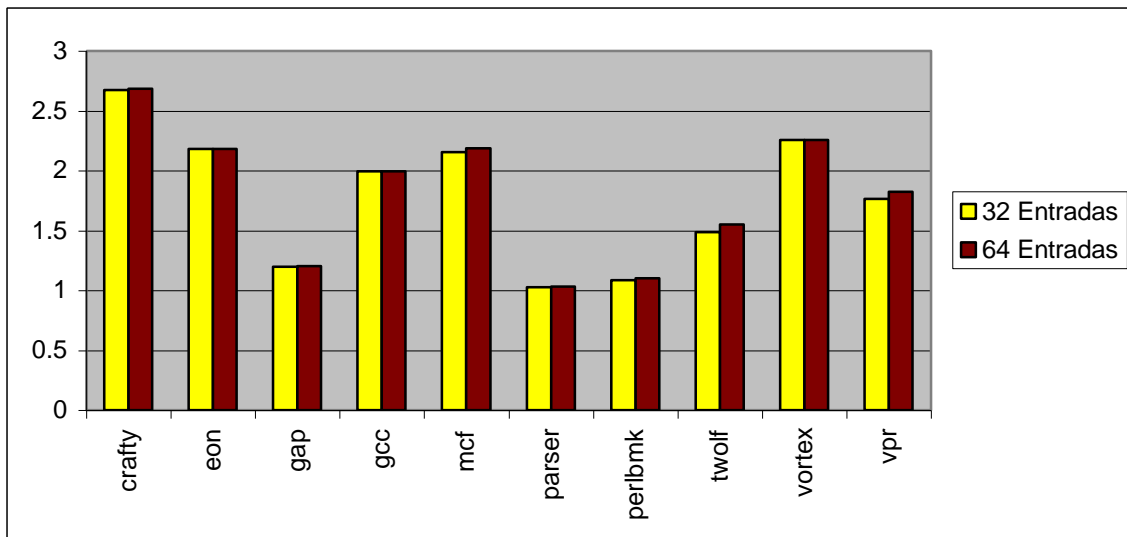


Fig. 3.2 Instrucciones por ciclo, SPECFP (ventana de 32 entradas ■ , 64 entradas ■ )

Las ventajas de tener una ventana de instrucciones mayor puede verse limitada por los siguientes factores:

- (1) Operandos fuentes no disponibles. Esta es la principal causa de que no sea mayor el número de instrucciones lanzadas a ejecutar por ciclo, a pesar del incremento de la ventana. Este factor se analiza en la sección 3.2.1.
- (2) Falta de recursos, es decir, falta de unidades funcionales disponibles.
- (3) Ejecución fuera de orden deshabilitada.
- (4) Poco nivel de paralelismo en el programa. Por ejemplo, una instrucción tiene que realizar una lectura o escritura a un registro que no está renombrado, o instrucciones que cambian o sincronizan el estado de la máquina.

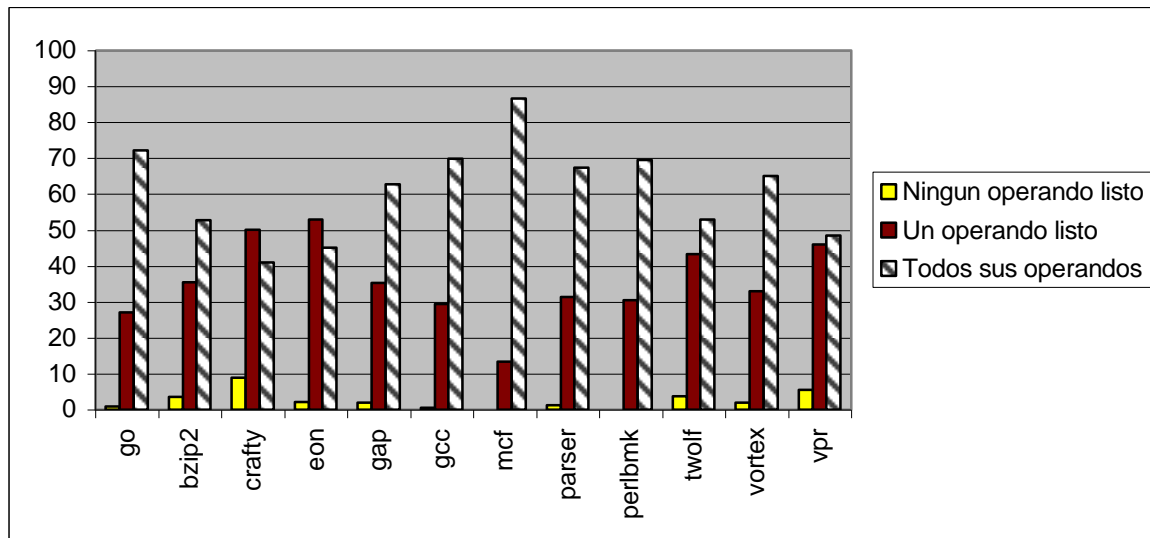
En la ventana de instrucciones se sigue una política o *lógica de selección*, de las instrucciones que serán removidas de la misma para ser lanzadas a la etapa de ejecución, o unidades funcionales. Dentro de esta *lógica de selección*, se manejan las siguientes condiciones:




- Operandos fuentes listos. Esto implica que su valor está listo, sea tomado desde los registros físicos, lógicos o como valor recientemente producido (etapa de *forward* o escritura).
- Disponibilidad de unidades funcionales.
- Antigüedad de las instrucciones dentro de la ventana.

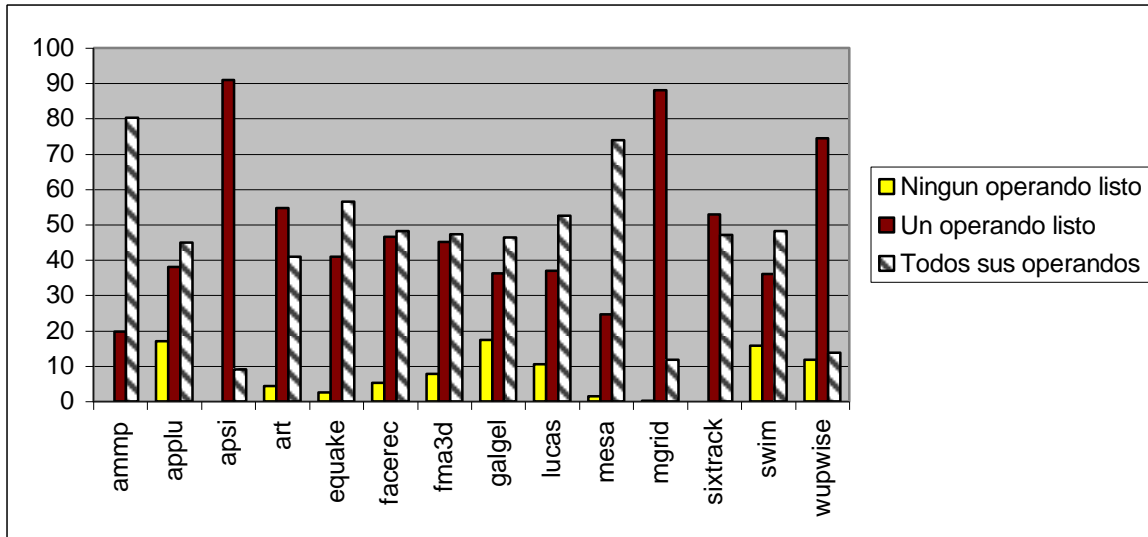
De acuerdo a las condiciones anteriores, y que una de las principales causas del bajo nivel de retiro de instrucciones de la ventana es la primera condición, la siguiente sección hace un análisis cuantitativo del impacto de está condición sobre una ventana de instrucciones de tipo centralizado.

#### 3.2.1 Análisis de dependencias por datos en la ventana de instrucciones

El uso de una ventana de instrucciones de tipo centralizado es común en la mayoría de los procesadores, sin embargo, las tendencias en los procesadores modernos han sido dividir esta ventana central en varias ventanas atendiendo un solo tipo o género de instrucciones. El análisis que se muestra en la figura 3.3(enteros) y 3.4 (flotantes) es para una ventana de tipo centralizada, de un tamaño de 32 entradas.



**Fig. 3.3** Porcentaje de operandos fuente para enteros (Instrucción con ningún operando listo , Instrucción con un operando listos , Instrucción con todos los operandos listos )



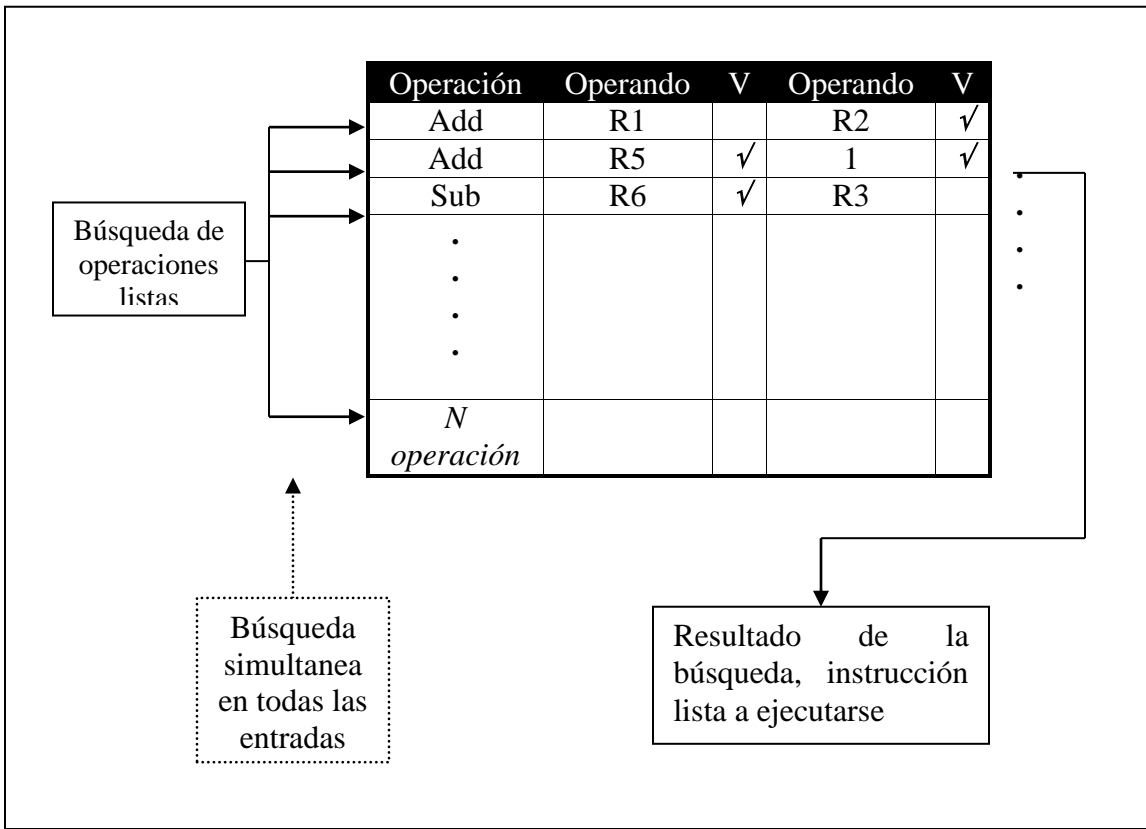
**Fig. 3.4** Porcentaje de operandos fuente para flotantes (Instrucción con ningún operando listo ■, Instrucción con un operando listo ■, Instrucción con todos los operandos listos ▨)

Es de apreciar que tanto en los *benchmark* de tipo entero como en los de tipo flotante, un gran porcentaje de las instrucciones contenidas en la ventana están a la espera de un dato. Por ejemplo, en el caso de *apsi*, el porcentaje de instrucciones que solo están a la espera de un dato son más del 90%, lo que implica un bajo nivel de paralelismo del programa.

Esta espera, nos conduce a la etapa posterior a la ejecución de las instrucciones; tal y como se menciona en el capítulo 1 se hace uso del “*reorder buffer*” para mantener la secuencia del programa, así como solucionar las dependencias de datos de otras instrucciones con los resultados que se van generando de las unidades funcionales (retiro de instrucciones). A esta parte en especial del retiro de instrucciones, se le llama postescritura *writeback*, la cual consiste en actualizar el valor de los registros lógicos, usados en la etapa de renombramiento y que permiten solucionar dependencias entre las instrucciones.

Sin embargo, esta operación de postescritura de datos en la ventana de instrucciones, es una *operación redundante o repetitiva*, la cual en su mayoría no logra su objetivo. Esto se menciona en base a la implementación de la ventana de instrucciones y la política que se sigue para encontrar el operando fuente que hace uso del dato que se acaba de generar.

La ventana de instrucciones es semejante en muchos aspectos a una memoria caché completamente asociativa. En la que se realizan búsquedas de instrucciones listas a ejecutarse, u operandos que solicitan un dato determinado en forma paralela a todas las entradas, de ahí que se mencione que se asemeja a una memoria caché completamente asociativa. La figura 3.5 muestra como se realiza la búsqueda dentro de la ventana de instrucciones, haciendo un barrido de todas las entradas en forma paralela, y dependiendo de las banderas de estado (*status*) de cada operando, se determina si se puede lanzar a ejecutar esa operación, o en caso contrario esperar al siguiente ciclo.



**Fig. 3.5** Búsqueda de instrucciones listas para ejecutarse dentro de la ventana de instrucciones

En forma similar se realiza la búsqueda del operando que solicita el dato acabado de generar en las unidades funcionales, para eliminar dependencias de datos. Esta búsqueda se realiza comparando cada uno de los operandos de las instrucciones almacenadas en la ventana de instrucciones, lo que representa dos comparaciones por instrucción. Sin embargo, el número de comparaciones efectivas está por debajo del número de comparaciones que se realizan. A este tipo de comportamiento se le denomina *operaciones redundantes* dentro de la ventana de instrucciones.

### 3.2.2 Políticas de comparación de entradas dentro de la ventana de instrucciones

El tipo de comparaciones a realizar dentro de la ventana de instrucciones se pueden dividir en tres categorías principalmente, y estas dependen del tipo de arquitectura empleada en el procesador.

- *Comparación arbitraria:* en esta categoría se hace un barrido completo de todas las entradas de la ventana de instrucciones. Por ejemplo, se tiene una ventana de 16 entradas, de la cual cada vez que se tiene un dato para solucionar dependencias, se compara las 16 entradas de la ventana (contando que cada instrucción tiene dos o tres potenciales receptores; para efectos de sencillez



tomaremos un número fijo de 2 receptores) sin importar si están activas todas las entradas de la ventana, lo que trae como consecuencia 32 comparaciones cada ocasión que se realiza el proceso de postescritura. Este método ha dejado de ser utilizado por la mayoría de los fabricantes debido a su alto índice de consumo de energía.

- *Comparación activa:* la comparación sólo se realiza con las entradas que están activas de la ventana de instrucciones. Por ejemplo, utilizando la misma ventana de 16 entradas, si solo se tiene 10 entradas activas, solo se realizarían 20 comparaciones en busca de un posible receptor del dato que han generado las unidades funcionales. Las entradas inactivas de la ventana de instrucciones no son sujeto de comparación, lo que evita la realización de comparaciones innecesarias, sin embargo, no todas las entradas activas están a la espera del mismo dato, y tomando en cuenta que solo una o dos instrucciones lo requieren, se tiene un 90% de comparaciones infructuosas.
- *Comparación selectiva:* al realizar la comparación de etiquetas sólo se realiza con los operandos (no entradas activas) que requieren el dato, sin embargo al manejarse la ejecución de las operación fuera de orden, existe la posibilidad de que dos instrucciones hacen uso del mismo registro pero en tiempos diferentes, por lo que puede darse el caso de que se hagan comparaciones con operandos con el mismo registro, pero que ya cuentan con su valor correspondiente (operandos listos o *ready*), por lo que también se considera comparaciones innecesarias o redundantes. Esta categoría es la de menor índice de comparaciones redundantes, siendo posible su implementación en hardware.
- *Comparación ideal:* esta última categoría engloba el número exacto de comparaciones necesarias para satisfacer la demanda de datos por parte de la ventana de instrucciones, descartando entradas inactivas, operandos con registros fuente diferentes al que se esta comparando, y evitando operandos con el mismo registro pero que ya cuentan con su valor (operandos *ready*). Este tipo de comparaciones implica un desarrollo en hardware muy complicado, excediendo el tiempo asignado a cada etapa del procesador, sacrificando rendimiento y en algunos casos excediendo el consumo de energía original.

La figura 3.6 muestra una gráfica comparativa para el número de comparaciones realizadas en una ventana de 64 y la figura 3.7 la gráfica de comparaciones para una ventana de 32 entradas respectivamente. Es de notar que el numero de comparaciones ejecutadas normalmente (comparaciones arbitrarias) en contra de las realizadas contra en forma ideal es muy grande. En la gráfica se representa el porcentaje de comparaciones realizadas en forma activa, selectiva e ideal en contra de las comparaciones arbitrarias, las cuales tomaremos como *baseline* (100%)

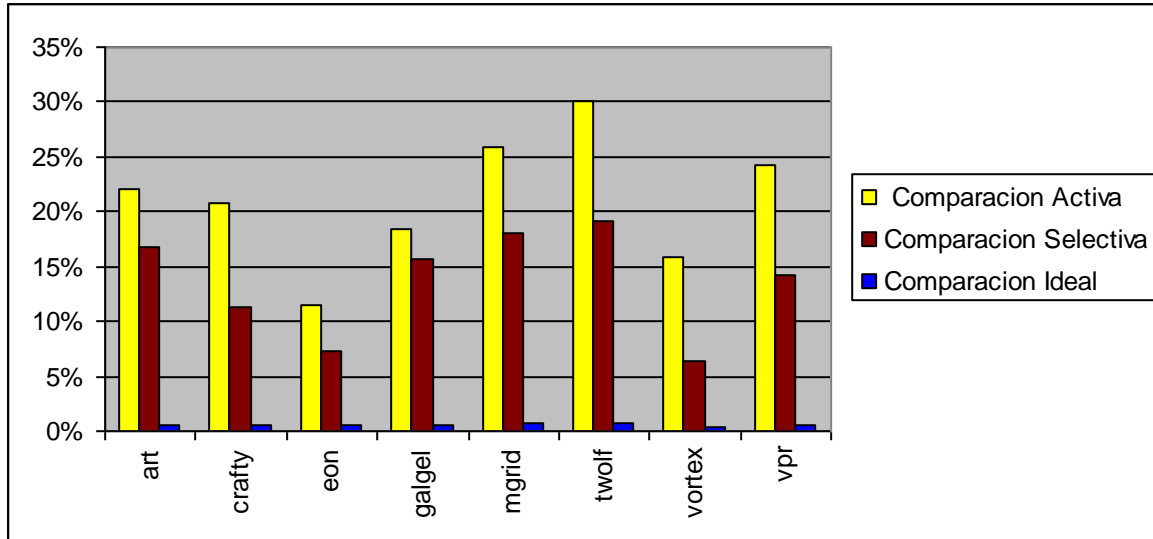


Fig. 3.6 Ventana de instrucciones de 64 entradas

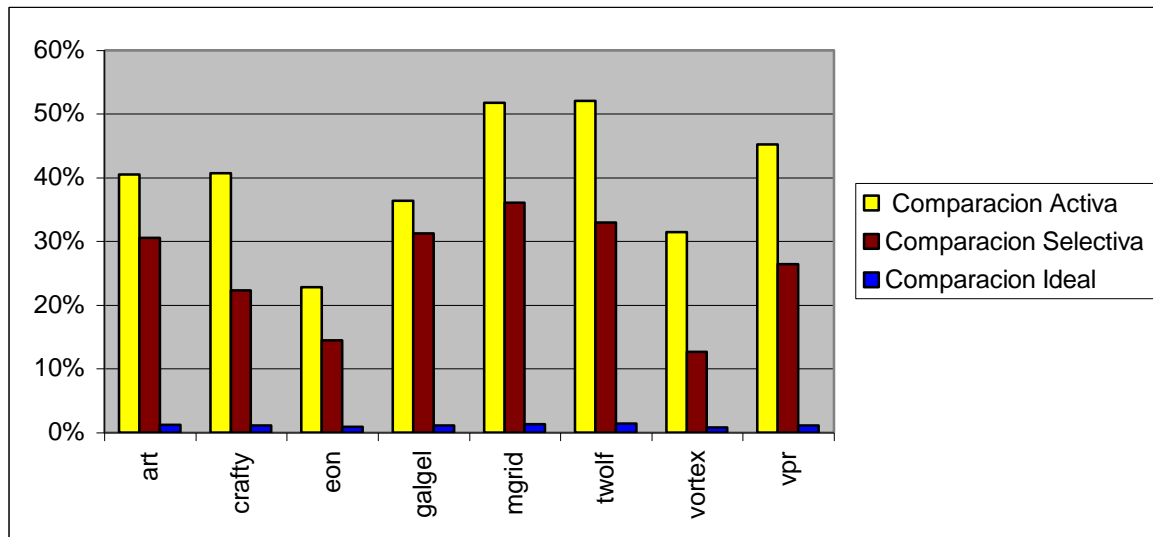
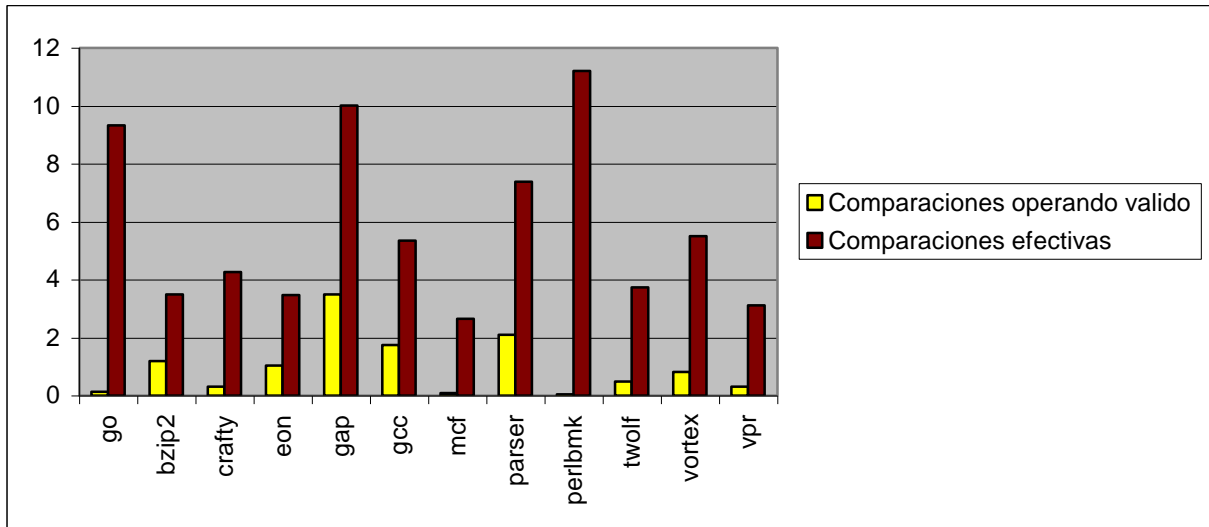


Fig. 3.7 Ventana de instrucciones de 32 entradas

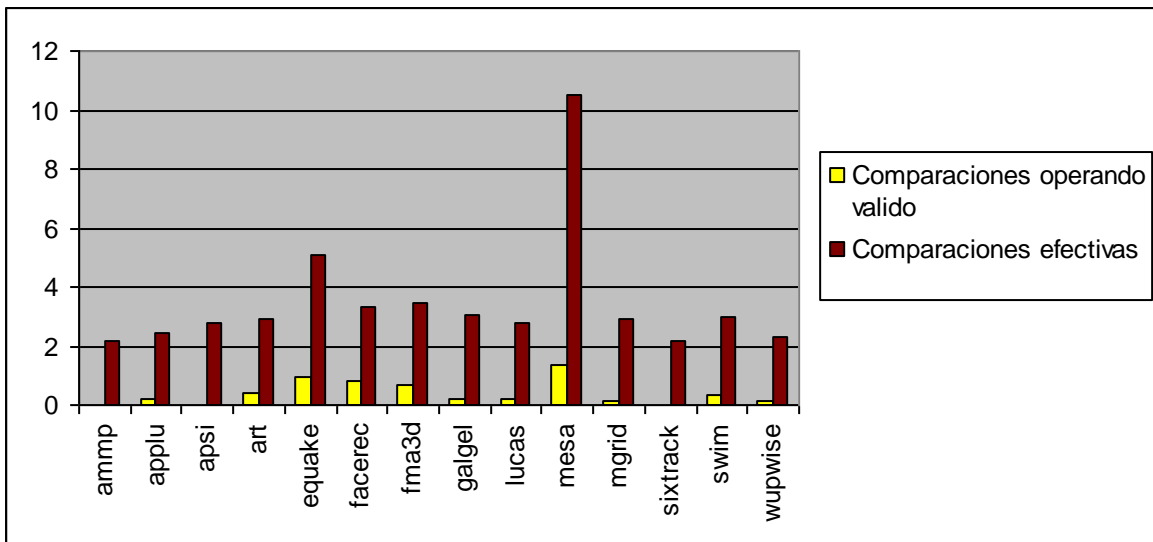
La figura 3.8 y 3.9 se muestra el número de *comparaciones* efectivas (comparaciones realmente necesarias para solucionar dependencias) que se realizan normalmente en una ventana centralizada y las comparaciones con un operando que ya tenia su dato validado, es decir, cuenta con el mismo registro fuente pero con un dato diferente ya validado. Las *comparaciones efectivas* son aquellas que tienen éxito, y que el operando fuente que se comparó en realidad solicitaba el dato en cuestión.

Pero aún dentro de estas comparaciones existen algunas que no son válidas. Esto es producto de la ejecución fuera de orden de las instrucciones; en la ventana se albergan instrucciones de diferentes secciones del programa, razón por la cuál se puede suscitar que dos operaciones cuentan con el mismo registro físico, pero renombrado en diferentes lapsos de tiempo. Por ejemplo, supongamos que la instrucción *add R1,R2,R3* ya cuenta

con el valor del registro R2 pero no el de R3; ahora una instrucción *mov* R5,R2 está a la espera del valor de R2; cuando se busca dentro de la ventana los registros que solicitan el dato de R2, se compara el operando de la primera y segunda instrucción, pero solo la segunda instrucción aprovecha el dato traído de las unidades funcionales, ya que la otra ya cuenta con ese valor. A este tipo de comparaciones las llamaremos *comparaciones efectivas con un operando válido*.



**Fig. 3.8** Porcentaje de comparaciones efectivas y comparaciones con un operando válido dentro de la ventana de instrucciones (SPECINT)



**Fig. 3.9** Porcentaje de comparaciones efectivas y comparaciones con un operando valido dentro de la IW (SPECFCP)

El número de comparaciones efectivas que se realizan, en la mayoría de los casos no supera el 12% de las comparaciones totales que se realizan en la ventana de instrucciones. Este tipo de comportamiento da como resultado el desaprovechamiento de

más del 90% de estas comparaciones durante cada ciclo. Si se parte de la idea de que solo un pequeño porcentaje de las comparaciones son efectivas, y la gran mayoría solo se realizan en uno de los operandos de las instrucciones almacenadas en la ventana; esto en base a las gráficas de la figuras 3.11 y 3.12 en donde se muestra que un gran porcentaje de las instrucciones que no están listas para ejecutarse solo les falta un operando, surge la pregunta ¿los operandos del lado izquierdo o derecho son los mas propensos a solicitar el dato que se genera de las unidades funcionales?

Esta pregunta pudiera tener una respuesta a nivel de software, sin embargo, como nuestro interés es puramente hardware, se detectó el comportamiento de la ventana en forma dinámica y a nivel de registros. La figura 3.10 muestra la grafica en la que denota que cuadrante de la ventana es el que tiene más operandos a la espera de un dato. La ventana de instrucciones puede ser fraccionada en cuatro cuadrantes.

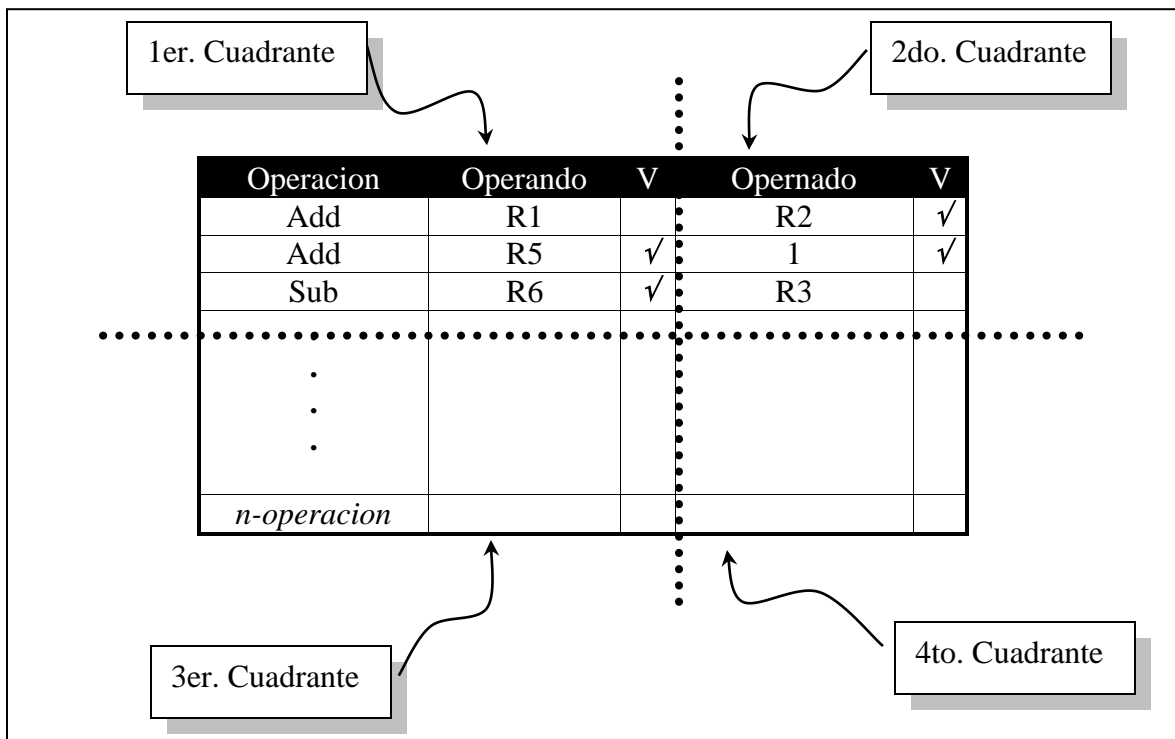


Fig. 3.10 Partición imaginaria de la ventana de instrucciones

Los operandos correspondientes a cada cuadrante se dividen de la siguiente forma:

- 1er. Cuadrante: parte superior izquierda
- 2do. Cuadrante: parte superior derecha
- 3er. Cuadrante: parte inferior izquierda
- 4to. Cuadrante: parte inferior derecha

Es fácil pensar que si la ventana se actualiza de la parte superior hacia abajo, los dos primeros cuadrantes serán las ubicaciones con más operandos a la espera del dato.

Las figura 3.11 y 3.12 muestran este comportamiento, acentuándose la cantidad de operandos en la parte superior izquierda (1er. Cuadrante).

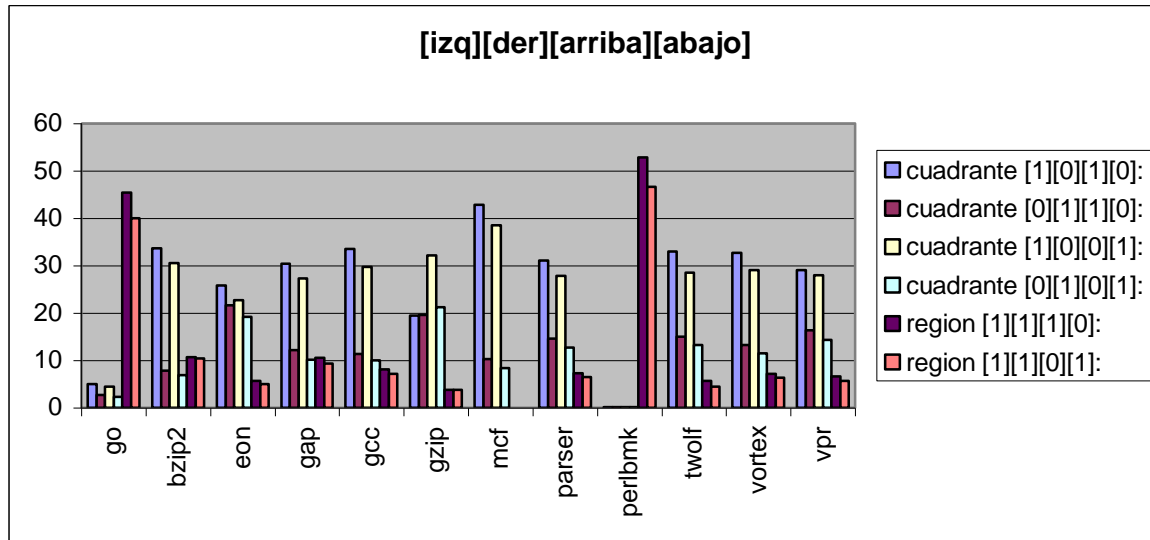


Fig. 3.11 Porcentaje de receptores (operandos) por cuadrante en la ventana (SPECINT)

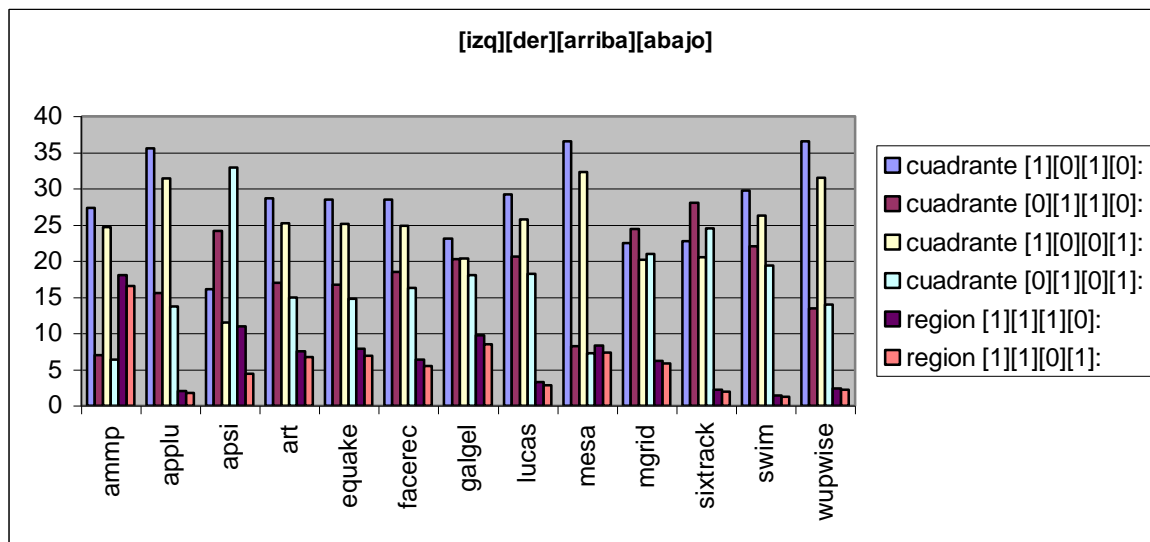


Fig. 3.12 Porcentaje de receptores (operandos) por cuadrante en la ventana (SPECFP)

El porcentaje total, se reparte entre todos los cuadrantes en los que se dividió la ventana de instrucciones, además de dos regiones que se forman, la parte superior de la ventana y la parte inferior. En estas gráficas se aprecia como la mayoría de los receptores se ubican en uno o dos cuadrantes. Esta clase de comportamientos fácilmente pueden ser aprovechados para proponer nuevos modelos de búsqueda en la ventana de instrucciones.

Sin embargo, ¿cuál es el beneficio de *no* realizar un alto número de comparaciones, si se sabe que las comparaciones se realizan en forma paralela y no afecta al rendimiento del procesador? En la siguiente sección se comenta el concepto de ahorro de consumo de energía y posibles técnicas.

### 3.2.3 Consumo de energía

El consumo de energía es un concepto recientemente adoptado por los grupos de investigación y diseño de procesadores. Con el incremento de los niveles de integración, el consumo de energía se ha vuelto uno de los parámetros críticos en el diseño de procesadores. Consecuentemente, una gran cantidad de trabajos han sido enfocados en la búsqueda de técnicas que permitan disminuir el nivel de consumo de energía del procesador [1].

En general, existen tres niveles en los que se aplican las técnicas de reducción de consumo de energía:

- *Sistema*: métodos de compresión de datos, manejadores de energía, control de actividades según un orden (agenda), protocolos de acceso medio y particiones del sistema, y control de errores de comunicación.
- *Arquitectura*: hardware paralelo (procesamiento en forma paralela), jerarquía del sistema de memoria, compiladores.
- *Tecnológico*: diseño asíncrono, control de la frecuencia de reloj, reducción de voltaje, reducción de ruteado en el chip.

Para este caso, nos enfocamos en el nivel de *arquitectura*, en que se manejan diferentes modelos, y se puede entender mejor el impacto de cada uno atendiendo a la siguiente fórmula:

$$P_d = C_{eff} V^2 f$$

Esta fórmula corresponde al cálculo de consumo de energía dinámico en tecnología CMOS (85 a 90%) en una primera aproximación, donde:

- $P_d$ : es el consumo de energía en Watts
- $C_{eff}$ : es la capacitancia efectiva en Faradios
- $V$ : es la alimentación de voltaje en Volts, y
- $f$ : es la frecuencia de operación en Hertz.

De estos factores, el único que es posible modificar a nivel de arquitectura es la capacitancia ( $C_{eff}$ ), ya que ésta a su vez está dada por la siguiente fórmula:

$$C_{eff} = \alpha C$$

Donde:  $C$  es la capacitancia de carga/descarga y  
 $\alpha$  es la actividad correspondiente al número de transiciones por compuerta.

Si se reduce el número de operaciones realizadas ( $\alpha$ ), disminuye por consecuencia la actividad y así en forma directa el consumo de energía. Siguiendo este razonamiento

queda sustentada la razón para disminuir el número de comparaciones inútiles en la ventana de instrucciones en busca de un receptor.

La figura 3.13, muestra el aumento del consumo de energía conforme se incrementa el número de entradas de la ventana de instrucciones. Esto es debido al número de comparaciones (actividad a nivel bit) que tiene que realizar cuando se introduce un dato que se acaba de generar en las unidades funcionales. Los valores de las variables de la fórmula para el cálculo de consumo se toman como valores fijos ( $V = 5$  volts,  $f = 400$  MHertz y  $C$  es tomado como el valor regular de capacitancia en tecnología CMOS)

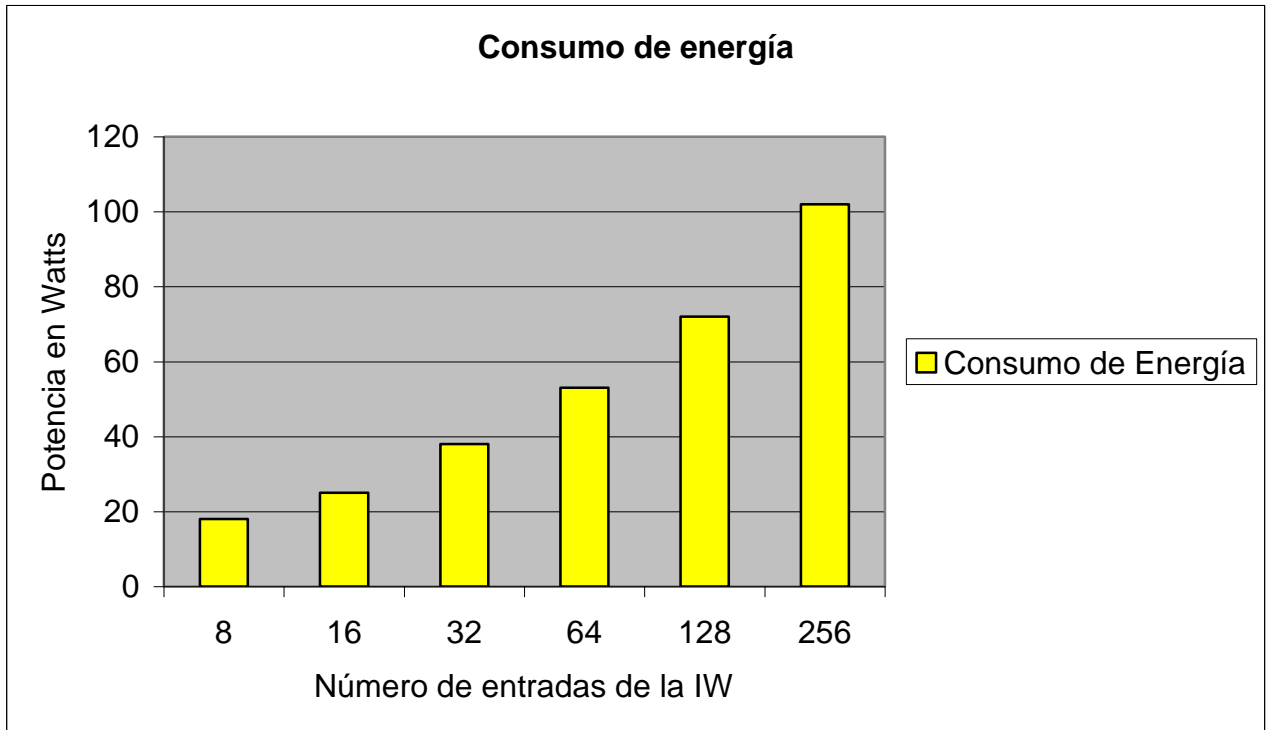


Fig. 3.13 Potencia consumida en ventanas de instrucciones con diferente número de entradas (Watts)

### 3.3. Antecedentes

Para reducir el consumo de energía al momento de realizar las comparaciones con los operandos fuente (consumidor): proceso denominado *wake-up*, Folegnani y González proponen varias optimizaciones [2]. Específicamente proponen que las entradas de la ventana que se encuentran vacías o contienen una instrucción con todos sus operandos fuente listos, no sean comparadas contra la etiqueta del dato que se acaba de producir en las unidades funcionales. Además, el tamaño de la ventana es ajustado en forma dinámica para reducir el área vacía. Acorde a los resultados que presentan Folegnani y González, al deshabilitar las comparaciones con entradas vacías o listas en una ventana de 128 entradas, se puede disminuir el número de comparaciones realizadas a un promedio de 14.2 comparaciones.

El uso de técnicas como indexar la ventana de instrucciones en lugar de realizar una búsqueda asociativa, han sido abordados por varios trabajos. En su trabajo S. Weiss y J. Smith, proponen un algoritmo denominado *Direct Tag Search (DTS)* [3]. En este esquema se adiciona un campo a la Tabla de Alias de Registros (*Register Aliases Table*), un campo y un bit a la ventana de instrucciones, como se ve en la figura 3.12.

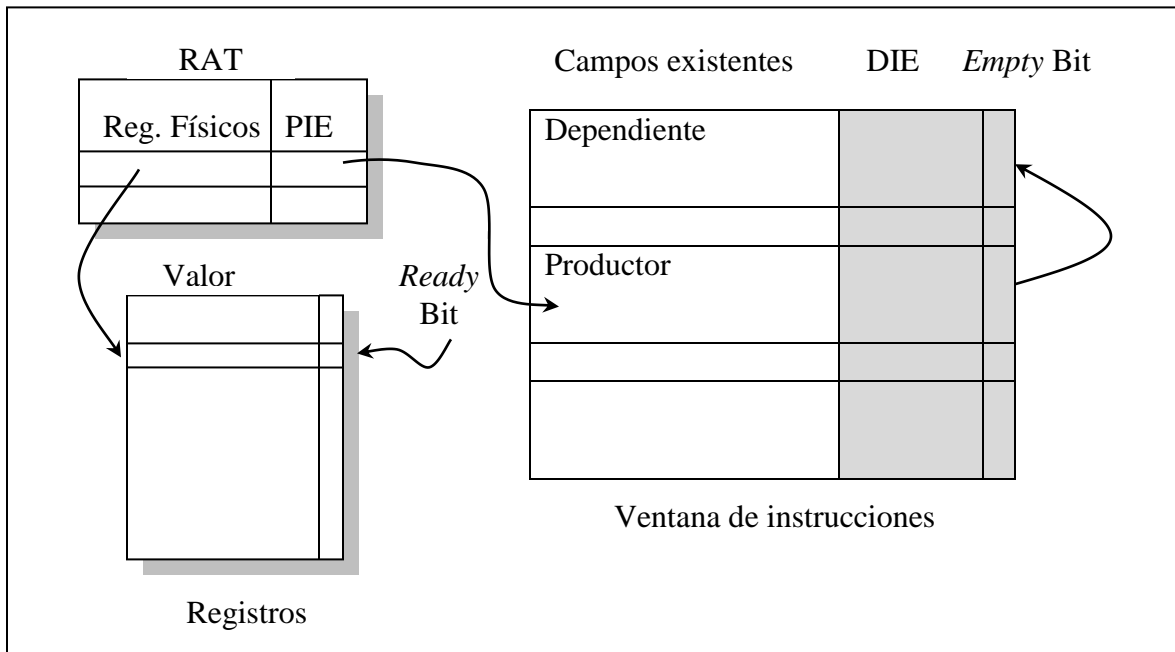


Fig. 3.14 Esquema Direct Tag Search (DTS)

Típicamente, cuando una instrucción sea introducida en la ventana de instrucciones, se realiza un chequeo para determinar si sus operandos fuente están disponibles o no. En caso de no estar listos, se identifica la entrada en la ventana de instrucciones encargada de producir el dato faltante. Luego, en esa entrada, se almacena un apuntador a la instrucción consumidora. El apuntador es simplemente el índice de un consumidor en la ventana. El apuntador es almacenado en un campo denominado (DIE) *Dependent Instruction-window Entry*. Además de adicionar un bit denominado *Empty* para operaciones de reset.

En la tabla RAT se tiene un campo denominado PIE (*Producer Instruction-window Entry*) que guarda un apuntador hacia una entrada de la ventana de instrucciones, que alberga la instrucción que genera el resultado esperado. El PIE es activado cuando una instrucción es insertada en la ventana de instrucciones.

Cuando una instrucción es decodificada e insertada en la ventana de instrucciones, y si cualquiera de sus operandos no esta listo, se llena el campo PIE con el apuntador que indica la instrucción productora de los valores esperados. Enseguida, el apuntador DIE de cada instrucción, es activado con la entrada correspondiente a la instrucción consumidora. Mas tarde, cuando una instrucción productora es concluida, esta pasa su apuntador DIE a través de un decodificador para únicamente habilitar el comparador de



la instrucción consumidora de ese resultado. A esta técnica se le denomina *Direct Tag Search*.

M. Huang, J. Renau y J. Torrelas [4] presentan un esquema híbrido llamado de la misma forma, “*Hybrid*”, que conjunta la técnica anterior (indexado en forma directa) y la búsqueda del consumidor en forma asociativa, ya que el primer modelo tiene el inconveniente de solo manejar una liga por cada instrucción, lo que impide localizar otros posibles consumidores del mismo resultado. Al aplicar un modelo combinado o híbrido, se manejan las dos ventajas de ambos modelos, por un lado la búsqueda selectiva de un consumidor, y por otro la búsqueda asociativa de varios consumidores para un mismo dato recientemente producido.

M. Goshima y K. Nishino, proponen el uso de un arreglo de dos niveles de memoria RAM [5], con las cuales soportan el indexado de las instrucciones para localizar a los consumidores, sin importar el número de dependencias para cada instrucción. Sin embargo, su modelo resulta muy complejo y poco eficiente desde el punto de vista de consumo de energía.

Otros trabajos utilizan una tabla de direccionamiento directo para mantener la cadena de dependencias, y como resultado, disminuir el tamaño requerido de la ventana de instrucciones. Entre estos, el trabajo propuesto por S. Onder y R. Gupta, *Dynamic Data Forwarding* (DDF) usa una memoria de espera (*Wait Memory*) para completar la unidad de búsqueda [6], es un equivalente a la ventana de instrucciones.

En el trabajo presentado por R. Canal y A. González [7], se estudia el caso del manejo de una tabla que mantiene o guarda al primer consumidor. Cuando esta instrucción está lista para ser ejecutada, se mueve a una cola de instrucciones listas, evitando parar por la ventana de instrucciones (*bypassing*). Sin embargo, esta técnica depende del factor de que gran parte de las operaciones solo cuentan con un operando dependiente.

En la mayoría de los modelos mencionados, se tiene un alto nivel de complejidad para realizar su implementación en hardware, por lo que no muchas son funcionales. El uso de varios bloques, módulos o tablas, que manejan apuntadores a entradas de la ventana de instrucciones complican su implementación en hardware, sin embargo, es posible tomar como referencia el modelo propuesto por S. Weiss y J. Smith, en el manejo de tablas que estipulan la ubicación de receptores para el dato que se genera en las unidades funcionales [8].

#### **3.4 Ventana de instrucciones dividida en bloques**

Al tener referencia del comportamiento de la ventana de instrucciones, en lo que respecta al porcentaje de instrucciones que están a la espera de un dato (dependencias de datos), y su posible ubicación dentro de la ventana (cuadrantes). Es posible plantear la idea de hacer una división de la ventana de instrucciones en cuatro bloques (ver Fig. 3.15). Al dividir la ventana se tiene un acceso más sencillo y un costo menos alto, al acceso en forma asociativa a una ventana de instrucciones grande (128 o 256 entradas). En este

caso el acceso puede ser semi asociativo, es decir, tener un comportamiento similar al de una memoria asociativa por conjuntos. Sin embargo se presenta el inconveniente de tener que agregar una etapa más o un ciclo más para realizar las comparaciones del receptor correcto, al igual que una memoria asociativa.

Para evitar la limitante de un comparador de resultados se hace uso de una *tabla de mapeo de bloques (TMB)* que permite determinar de antemano el bloque o bloques (note que no existe limite de dependencias por instrucción), donde se localiza el receptor o la instrucción dependiente del dato. De esta forma se puede evitar habilitar bloques en forma innecesaria.

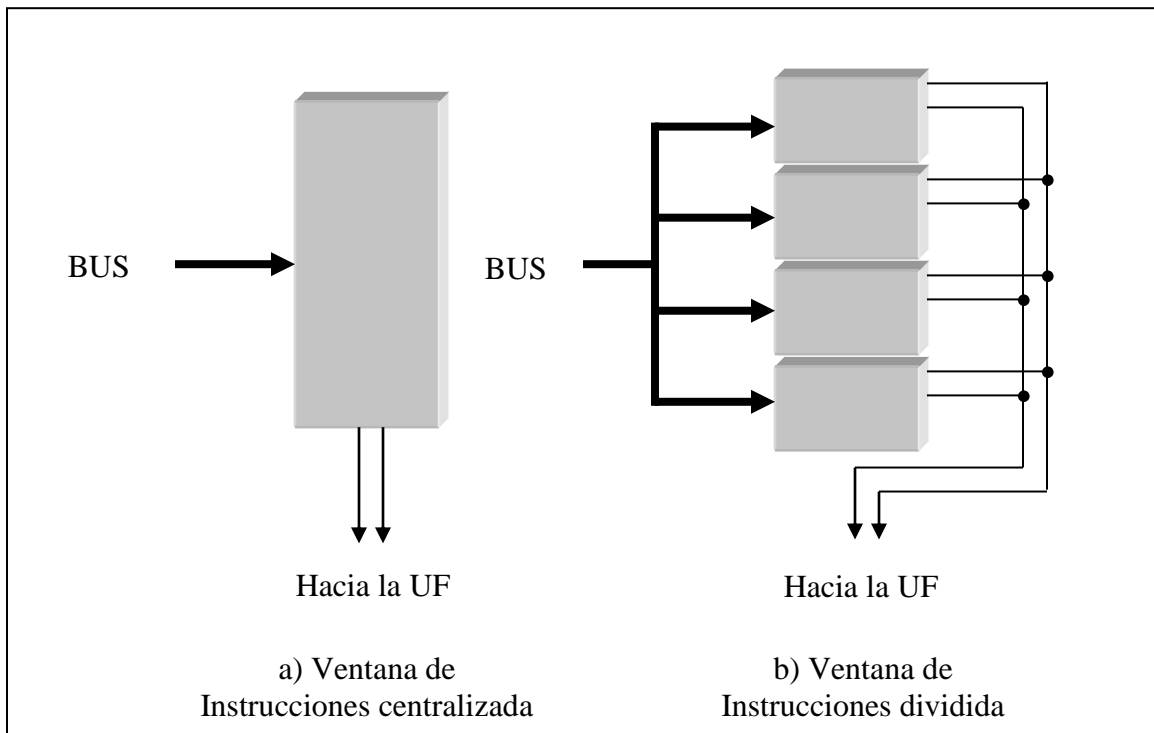


Fig. 3.15 Ventana de instrucciones dividida en bloques

La *TMB* esta conformada por los siguientes campos:

- *Ruta de habilitación:* en este campo se almacena una palabra de 4 bits, con la que se identifica el bloque, o bloques, que deben habilitarse. De esta forma, se habilitan solo los bloques necesarios y se disminuye el número de comparaciones inútiles que se realizan en una ventana de bloque único. A continuación se muestra la interpretación de la palabra de cuatro bits para realizar la identificación del bloque.

<b>0001</b>	Primer Bloque
<b>0010</b>	Segundo Bloque
<b>0100</b>	Tercer Bloque
<b>1000</b>	Cuarto Bloque

<b>1001</b>	Primer y cuarto Bloques
<b>0110</b>	Segundo y tercer Bloques
<b>1111</b>	Todos los bloques

- *Latencia:* número de ciclos que tarda la operación en realizarse, menos un ciclo (n-1). Este campo sirve para activar con un ciclo de antelación los bloques correspondientes a las instrucciones que están a la espera del resultado producido por las unidades funcionales.
- *Bit de reset:* este bit sirve para indicar cuando se ha renombrado el registro físico, y se limpian los otros dos campos de esta entrada.

Esta tabla se actualiza conforme se insertan instrucciones en los cuatro bloques en que se ha dividido la ventana de instrucciones. El tiempo que se le asigna a cada entrada depende del tipo de instrucción que lo produce. La forma de indexar la *TMB* es por medio del registro destino. El registro destino en esta etapa, ha sido renombrado por un registro físico, por lo que el número de entradas de nuestra tabla de mapeo va de acuerdo al número de registros físicos con los que cuenta la arquitectura (figura 3.16).

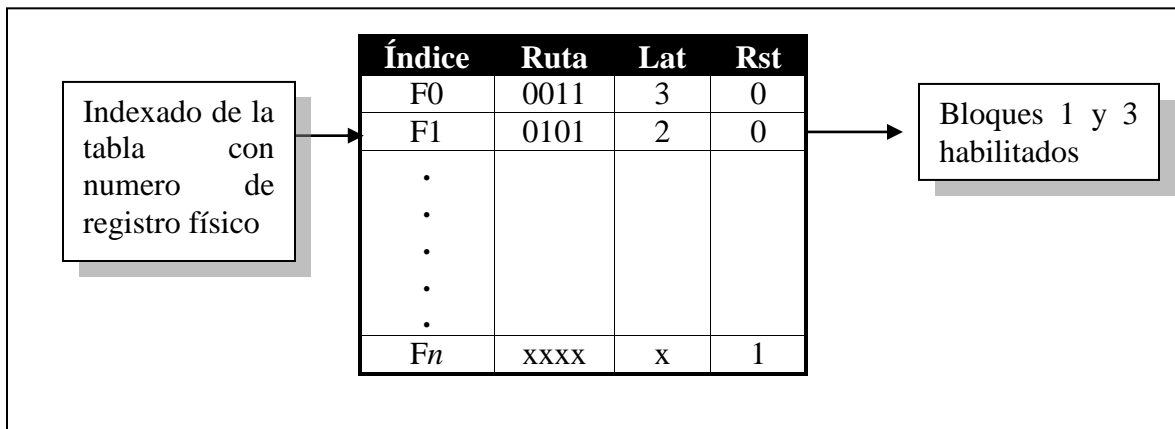


Fig. 3.16 Tabla de Mapeo de Bloques (TMB)

Pensando en un caso inicial, la *TMB* de la ventana se encuentra vacía; al momento de renombrarse un registro destino por primera ocasión, se accede a la tabla de mapeo y se activa el bit de reset de esta entrada. En el transcurso del procesamiento de otras instrucciones, habrá instrucciones que usen como operando fuente el primer registro renombrado, el cual ya tiene una entrada activa en *TMB*. La entrada activa es nuevamente direccionada, pero ahora para activar el campo de ruta de habilitación, siendo posible traslaparse cualquiera de las opciones anteriores, entendiéndose que este registro renombrado es usado por varias instrucciones como operando fuente, y que se encuentra localizado en dos o más bloques de la ventana de instrucciones dividida.

La información requerida para llenar la *TMB*, se recopila en la etapa de decodificación/renombramiento. En esta etapa se adiciona un módulo de control lógico, encargado de obtener la siguiente información:

- ✓ Próxima ubicación libre en los bloques de la ventana de instrucciones. La política de llenado de los bloques puede variarse, dependiendo de un fin específico, es decir, se puede implementar políticas de llenado de bloques de acuerdo a instrucciones con mayor frecuencia, importancia o relevancia, latencia de ejecución en las unidades funcionales, etc. Para nuestro caso, se maneja un llenado de las ventanas buscando la ubicación libre más próxima, accediendo a los bloques en forma secuencial, desde luego, empezando en el bloque uno.
- ✓ Registro físico renombrado como destino, el cual sirve para indexar la *TMB* y activar el bit de *reset* para limpiar los campos de esa entrada.
- ✓ Registros renombrados que se usan como fuentes en la instrucción próxima a ubicarse en la ventana de instrucciones. Esta información sirve para indexar la *TMB* y actualizar el campo de ruta de habilitación. Esto implica que si la próxima ubicación libre se encuentra en el bloque tres, se actualiza el campo de ruta con esta información, adicionándose a la que ya se tenga registrada.
- ✓ Tipo de operación, de la cual se determina la latencia de ejecución, y a la cual se le resta un ciclo. Esto permite a la *TMB* tener listos los bloques deseados al momento de que se obtiene el dato esperado por las operaciones dependientes.
- ✓ En el caso de operaciones de carga (referencia a la memoria), existe una alta probabilidad de que el dato si se encuentre, sin embargo, como se analizará en el capítulo 4, existe un pequeño margen de error (*miss*), lo que produce la posibilidad de que el dato no se obtenga en la latencia asignada a un acierto (*hit*) de memoria. En el caso de un *miss*, se recibe la señal de *miss* desde la memoria caché, lo que sirve para actualizar el campo de latencia de la *TMB*, de acuerdo a la latencia de la jerarquía de memoria.

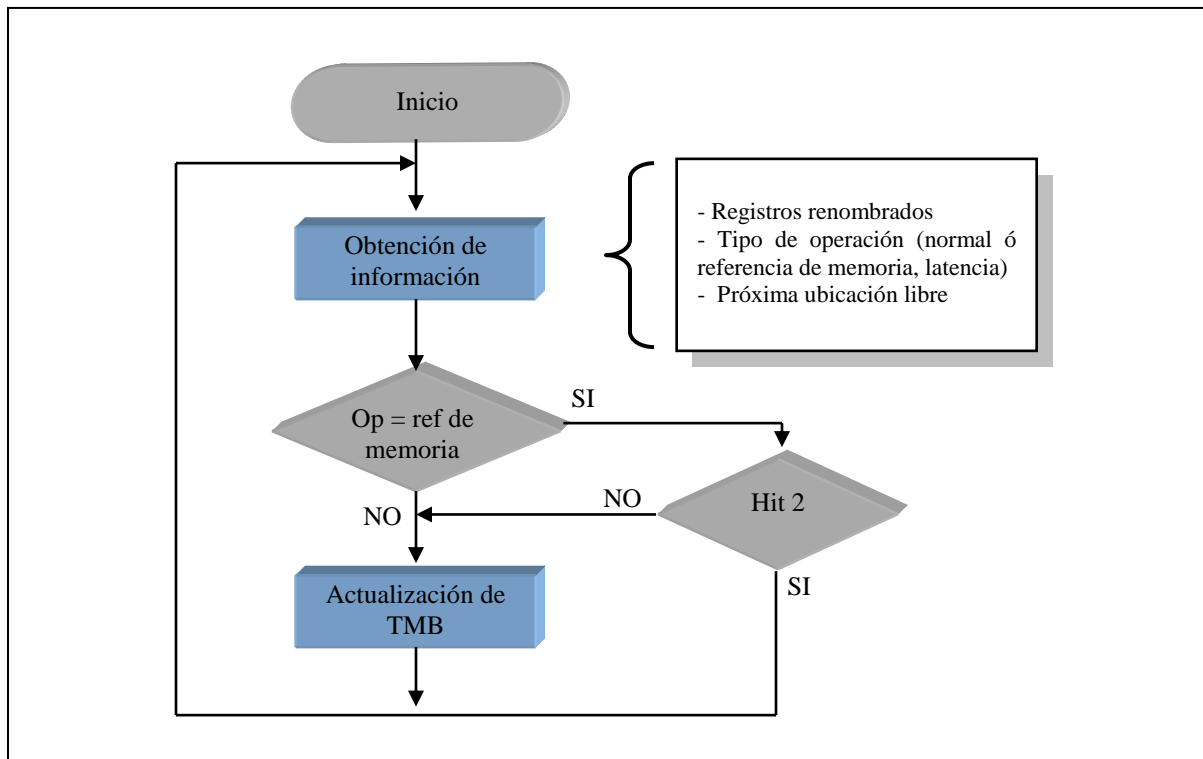
Este módulo de control lógico fue denominado *Unidad de Actualización de la TMB (UAT)*. La *UAT* obtiene información de diferentes partes de la arquitectura, puntos mencionados en la parte superior. La *UAT* obtiene del modulo de renombramiento de registros lógicos, información sobre el registro físico usado como registro destino en la instrucción que se acaba de decodificar. Esta información es usada para restablecer a cero los campos de la entrada correspondiente en la *TMB* a este registro. Los registros físicos usados como fuente indexan la *TMB* para actualizar el campo de ruta de habilitación.

Por otra parte la *UAT* obtiene información de cada uno de los bloques en que está dividida la ventana de instrucciones con un bus de 2 bits respectivamente, uno de los bits es denominado señal de llenado (*Full Bit*), y el otro, *señal de activación de bloque para escritura (Write Bit)*.

Además, la UAT recibe la señal de acierto o error (*hit/miss*) por parte de la memoria caché de datos con la finalidad de actualizar el valor de latencia de operación. Por último, recibe información sobre el tipo de operación y su latencia correspondiente en las unidades funcionales para ser completada. Toda esta información se conjunta en un bus de datos denominado Bus de Información *TMB (BIT)*, con el siguiente formato:

26 .....	21	20 .....	15	14	13	12 .....	7	6	5	4 .....	0
Reg. Destino		Reg. Fuente		Bloque		Reg. Fuente		Bloque		Latencia -(1)	

La UAT en este punto, es capaz de determinar el bloque de la ventana de instrucciones donde se ubicará el dato entrante a la ventana, la latencia de su ejecución en caso de contar con dependencias de datos, y actualizar los valores almacenados en la TMB. El siguiente diagrama de flujo da una idea del funcionamiento por pasos de este esquema. (Figura 3.17)



**Fig. 3.17** Diagrama de flujo de operación de la Unidad de Actualización de la TMB (UAT)

La figura 3.18 muestra el esquema del modelo propuesto, seccionado en las etapas del datapath del procesador.

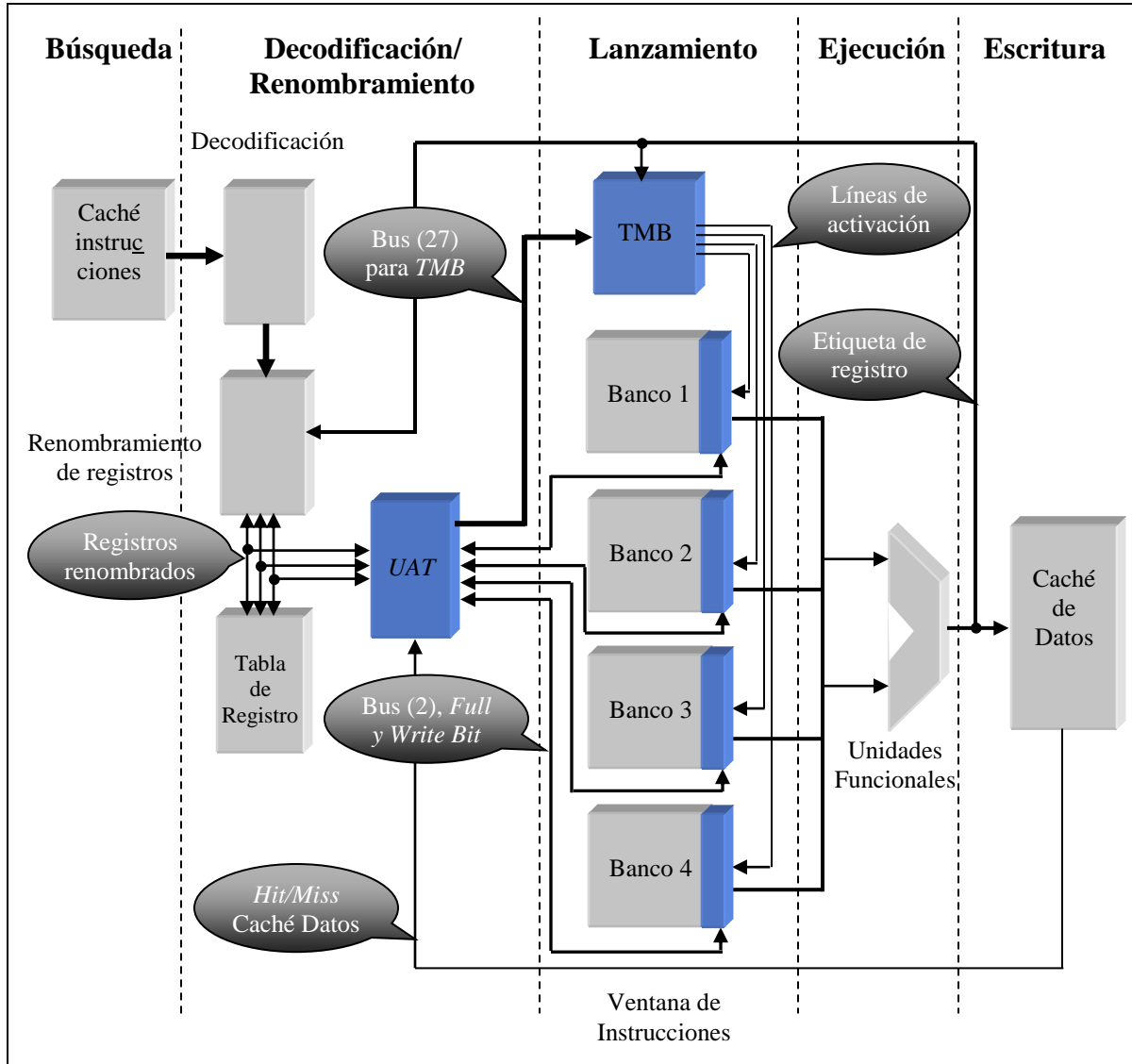
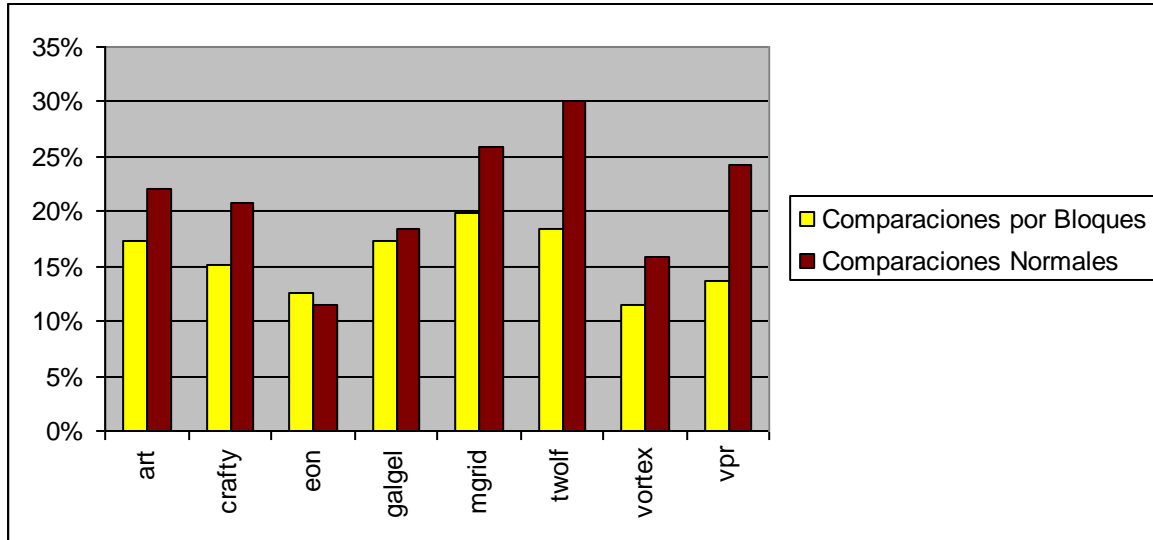


Fig. 3.18 Arquitectura de una ventana de instrucciones dividida

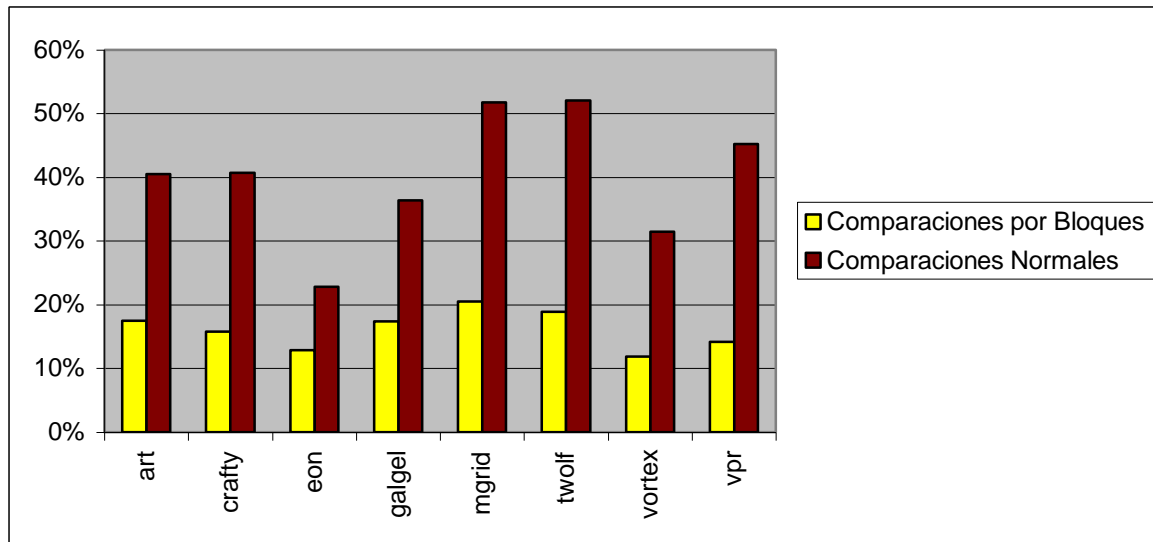
En la figura 3.18 no se incluye el bus de instrucciones que entra a la ventana de instrucciones, ya que se da por tácito el pensar que es un bus que tiene acceso a todos los bloques en los que esta dividida la ventana en forma paralela y simultánea. Las líneas de bus de 2 bits son direccionales, ya que se usa el *Write Bit* para activar el bloque seleccionado por la *UAT* para alojar la instrucción entrante a la ventana, de esta forma no solo se logra ganancia en el proceso de postescritura de datos en la ventana de instrucciones, sino también al momento de escribir instrucciones.

Al concluirse la ejecución de un dato que es esperado por otras instrucciones, se usa la etiqueta del registro físico que lo almacena para indexar la *TMB* y activa el bloque(s) que contiene operaciones en busca de este resultado.

El modelo de una ventana dividida en bloques disminuye en un 20% aproximadamente el número de comparaciones realizadas dentro de la ventana para un tamaño de 64 entradas (figura 3.19) y en un 60 % para una ventana de 32 entradas (figura 3.20), debido a que se limita el número de comparaciones como máximo al número de entradas activas de un bloque.



**Fig. 3.19** Número de comparaciones de esquema normal vs por bloques (política de comparación activa) ventana de 64 entradas, dividida en 4 bloques



**Fig. 3.20** Número de comparaciones esquema normal vs por bloques (política de comparación activa) ventana de 32 entradas, dividida en 4 bloques

Sin embargo este porcentaje sigue siendo alto en relación al número de comparaciones ideales a realizarse. En la figura 3.21 y 3.22 se denota la diferencia entre el número de comparaciones ideales y el modelo planteado, siguiendo una política de comparación arbitraria. Las figuras 3.23, 3.24 muestran el número de comparaciones ideales contra el

modelo de una ventana dividida, pero siguiendo una política de comparación de tipo activa.

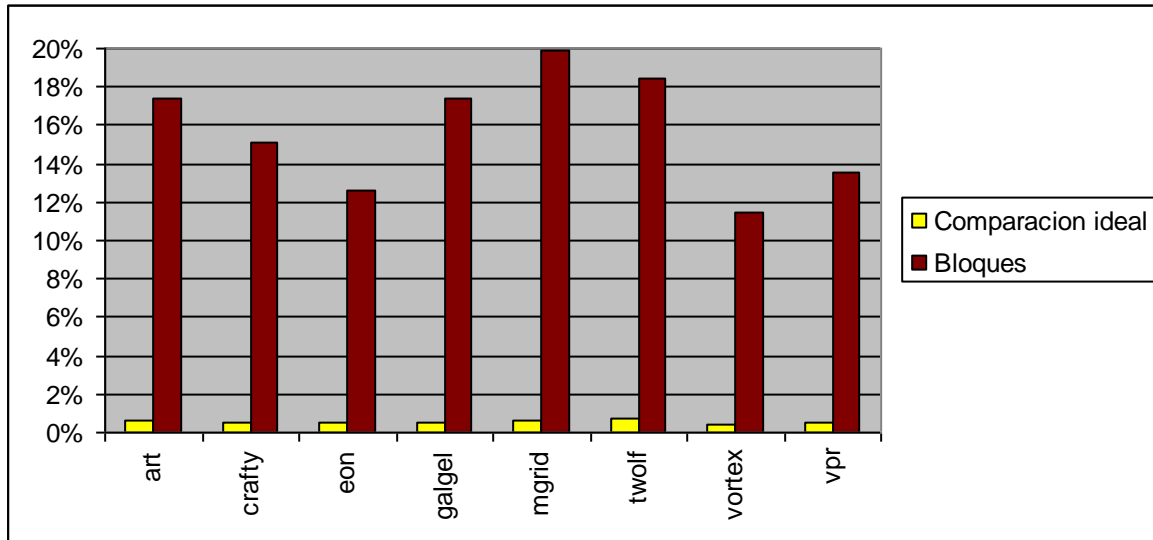


Fig. 3.21 Número de comparaciones ideales vs por bloques (política de comparación arbitraria) ventana de 64 entradas, dividida en 4 bloques

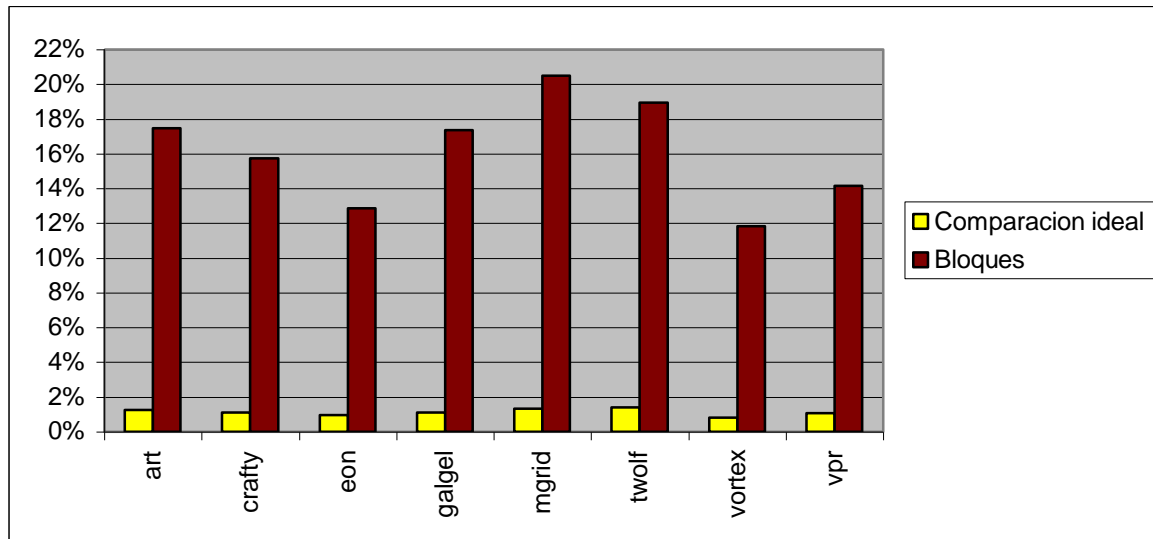
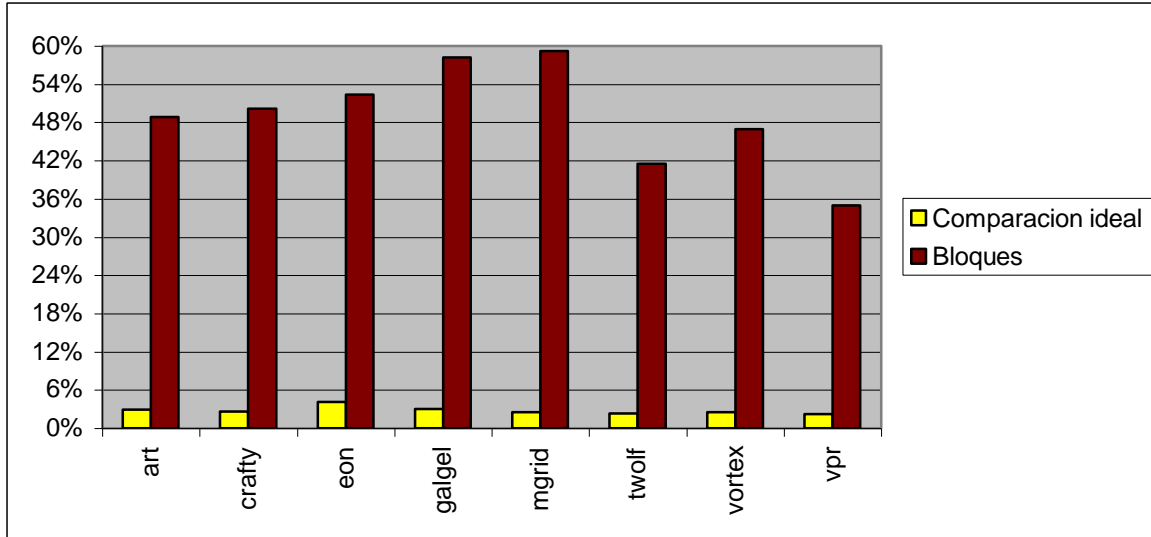


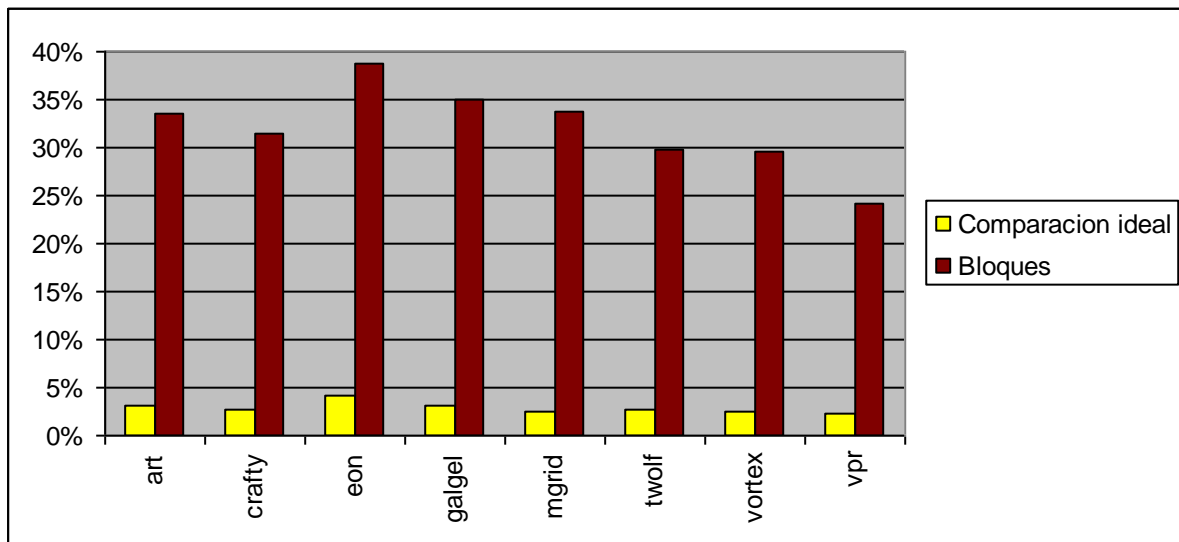
Fig. 3.22 Número de comparaciones ideales vs por bloques (política de comparación arbitraria) ventana de 32 entradas, dividida en 4 bloques

Las figuras 3.25 y 3.26 muestran la misma comparación, entre el número ideal y el manejado por el modelo de ventana dividida en bloques, pero usando la política de comparación selectiva respectivamente.



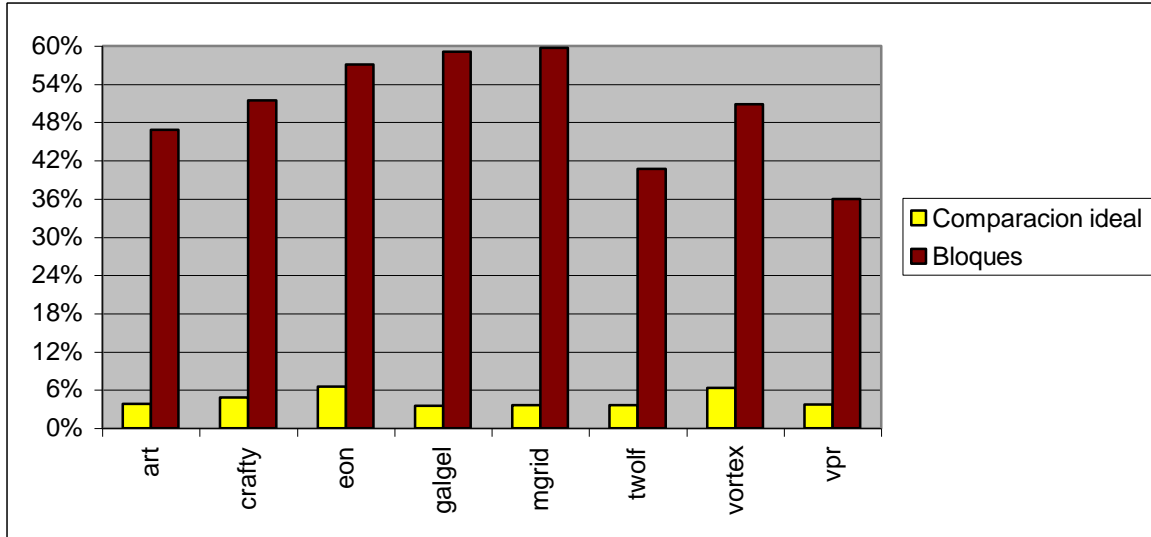


**Fig. 3.23** Número de comparaciones ideales vs bloques (política de comparación activa) ventana de 64 entradas, dividida en 4 bloques

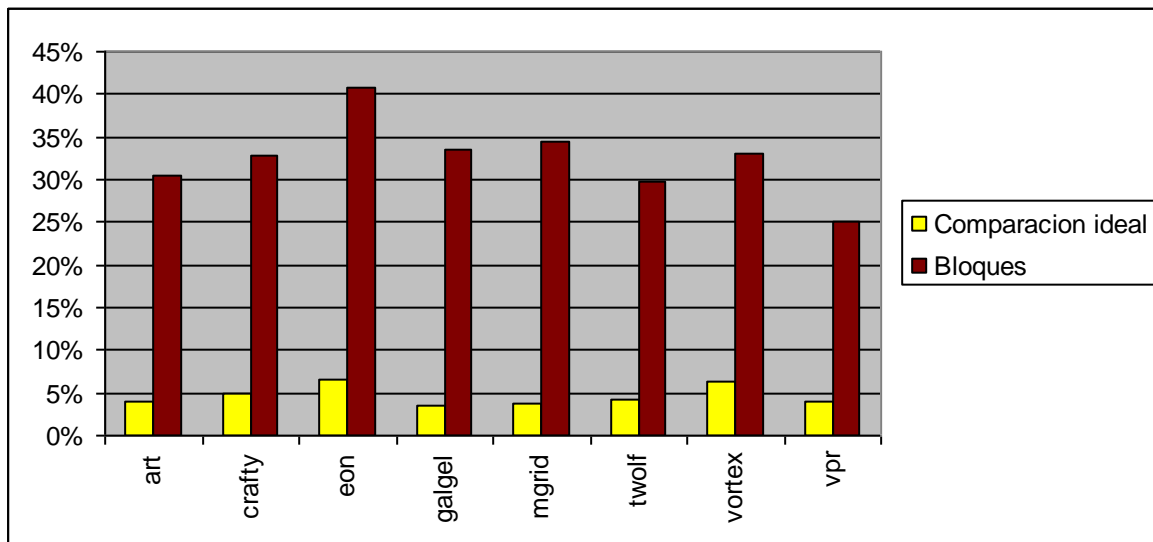


**Fig. 3.24** Número de comparaciones ideales vs bloques (política de comparación activa) ventana de 32 entradas, dividida en 4 bloques

En la mayoría de los procesadores actuales se maneja la política de comparación activa, ya que se usa un solo bit en cada entrada para activarla, lo que representa un bajo costo de implementación en hardware y un ahorro considerable de energía en la ventana. Por esta razón se tomará esta política de comparación activa, como parámetro de medición de impacto de nuestro modelo propuesto. Las gráficas de las figuras 3.25 y 3.26, que corresponden a la política de comparación selectiva sería el modelo ideal, sin embargo representa mayor complejidad en su implementación.



**Fig. 3.25** Número de comparaciones ideales vs. Bloques (política de comparación selectiva) ventana de 64 entradas, dividida en 4 bloques



**Fig. 3.26** Número de comparaciones ideales vs. Bloques (política de comparación selectiva) ventana de 32 entradas, dividida en 4 bloques

Debido a la alta diferencia entre el número de comparaciones ideales y las del modelo propuesto, sin importar la política de comparación (arbitraria, activa o selectiva), se proponen otros modelos derivados del modelo de la ventana dividida en bloques.

### 3.4.1 Submodelos de la ventana de instrucciones dividida en bloques

La división de la ventana de instrucciones en bloques favorece la reducción de comparaciones redundantes, sin embargo, aún es posible reducir este número, subdividiendo cada bloque en dos o cuatro partes. A continuación se explica las

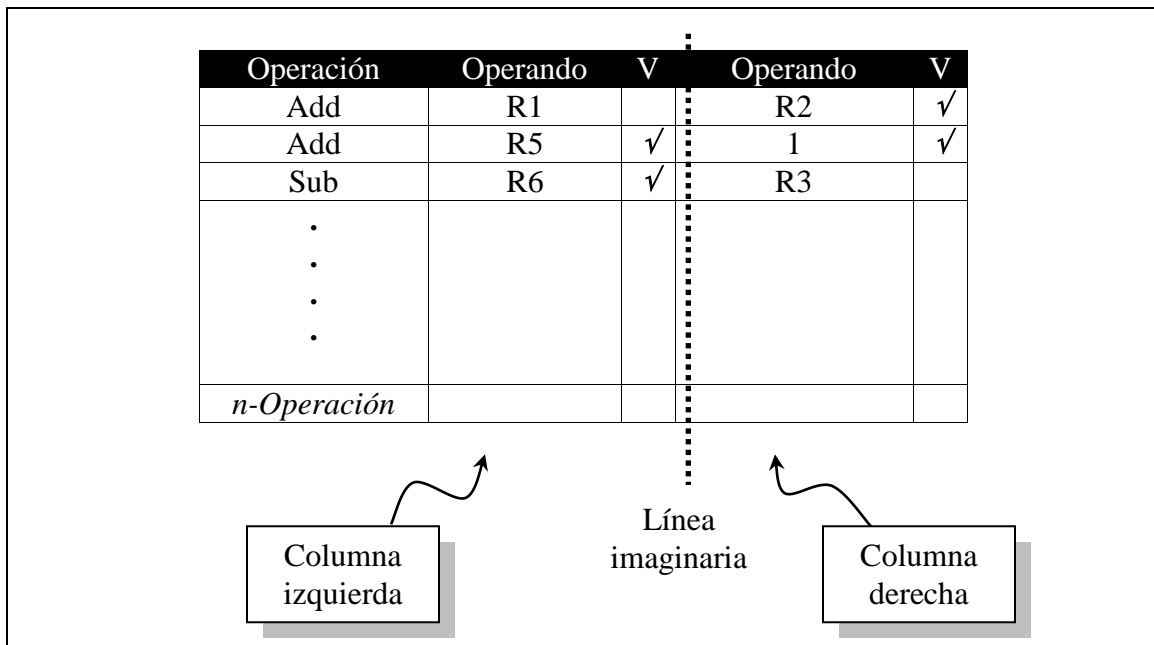
modificaciones al modelo original para generar tres modelos de ubicación y activación de los bloques de la ventana.

*División de bloques en columnas*

La figura 3.27 muestra los campos que conforman la ventana de instrucciones, ya sea en forma dividida en bloques o de tipo bloque único, en la cuál se aprecian claramente dos columnas en las que se puede dividir cada bloque, ya que prácticamente se cuenta con dos columnas de comparadores. Cada columna esta conformada por los registros fuente de cada instrucción, y como es común tener dos operandos fuente, se asigna una columna a los registros fuente del primer operando y otra a los registros correspondientes a la columna del segundo operando.

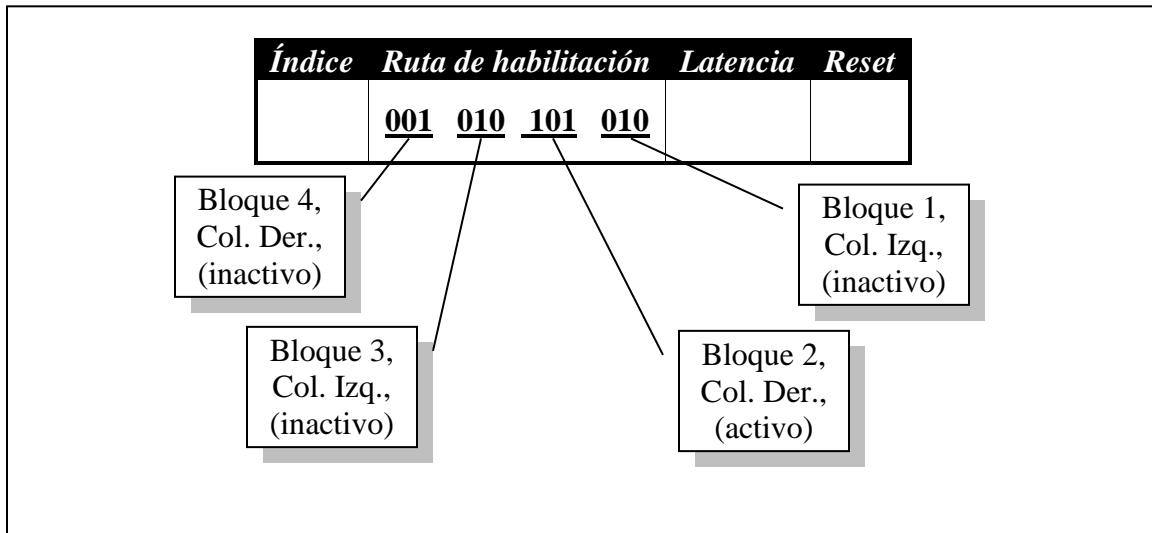
**Operación    Operando 1    Operando 2**

Según el estudio realizado en la primera sección de este capítulo, en la ventana de instrucciones se encuentran más consumidores en la columna del lado izquierdo en comparación con la columna de la derecha. Esta clase de comportamiento disminuye del número de comparadores activos, ya que a diferencia del modelo original, no se activan los dos comparadores de cada entrada activa del bloque, sino únicamente un comparador para cada entrada.



**Fig. 3.27** División de bloques de la ventana de instrucciones en columnas

La *TMB* sufre modificaciones en su campo de ruta de habilitación, al adicionarse ocho bits más a este campo, el cual se interpreta de la siguiente forma (figura 3.28):



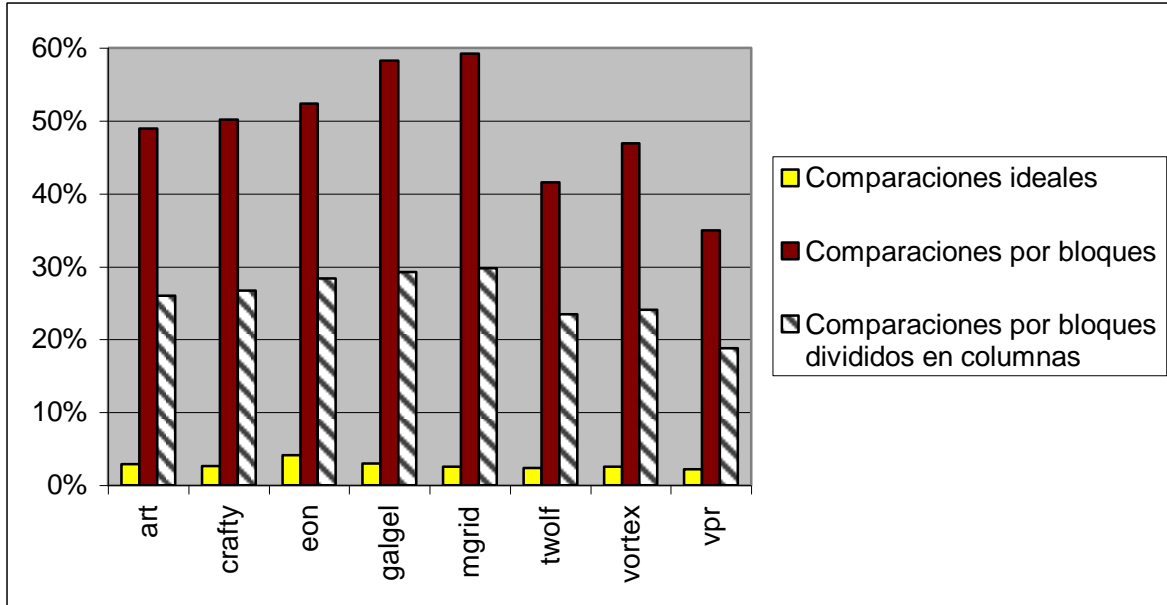
**Fig. 3.28** Formato de los campos de la TMB para bloques divididos en columnas

Ahora la palabra en el campo de ruta de habilitación cuenta con 12 bits, los cuales podemos dividirlos en cuatro conjuntos de tres bits para identificar donde está el posible receptor del dato, dentro de la ventana. El primer conjunto corresponde al bloque 4 (empezando de izquierda a derecha), el segundo conjunto al bloque 3, y así sucesivamente. Esto es, el primer bit (MSB) de cada conjunto significa si ese bloque debe de activarse, el segundo bit se interpreta que el receptor está en la columna izquierda y el tercer bit, indica si se el receptor está en la columna derecha.

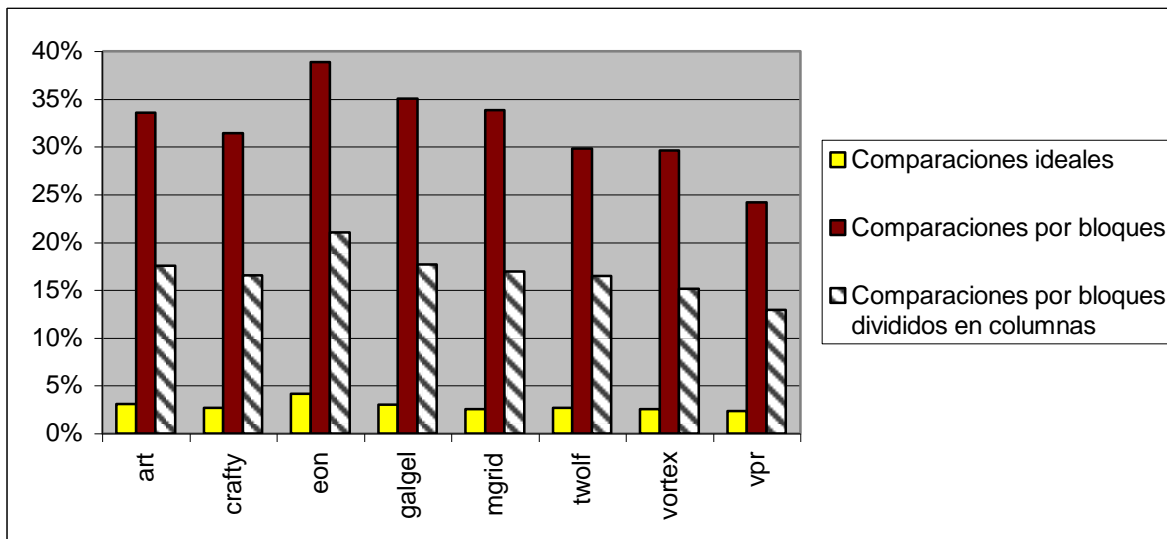
El bus *BIT*, también se modifica de igual forma, en cada uno de sus dos campos denominados *bloque*, donde se da la posición del receptor dentro de los bloques de la ventana.

La forma de obtener la ubicación dentro de los bloques (columna izquierda o derecha), es de acuerdo a la sintaxis de las instrucciones, siendo proporcionada por la unidad de renombramiento de registros. A nivel de hardware no requiere de otras modificaciones en las líneas que van hacia la *UAT*. Para los siguientes modelos, si se requiere realizar algunas modificaciones en las líneas de llegada a la *UAT*.

Los resultados de esta implementación se muestran en las figuras 3.29 (ventana de 64 entradas, 4 bloques, 2 columnas c/bloque) y 3.30 (ventana de 32 entradas, 4 bloques, 2 columnas c/bloque), donde se aprecia una reducción, de al menos 45% en todos los casos, de comparaciones entre la propuesta original (bloques) y este submodelo (división de bloques en columnas).



**Fig. 3.29** Número de comparaciones ideales, por bloques y bloques divididos en columnas (ventana de 64 entradas)



**Fig. 3.30** Número de comparaciones ideales, bloques y bloques divididos en columnas (ventana de 32 entradas)

*División de Bloques en filas*

Los bloques de la ventana de instrucciones no están llenos en su totalidad la mayoría del tiempo, ya que sugiere tener un alto nivel de paralelismo de instrucciones. Al no tener todas las entradas activas, la mayoría de las instrucciones se concentran en la parte alta de la ventana de instrucciones o de cada bloque. En este modelo la división de los bloques se hace en forma vertical, quedando dos regiones, las cuales denominaremos filas, fila superior e inferior. Al realizar esta división se procura que la parte inferior se active en

menor número que la fila superior. La figura 3.31 muestra la forma de dividir cada uno de los bloques, además se adiciona un bit en la entrada que está a la mitad del bloque, para indicar cuando una instrucción es colocada en la parte superior o inferior.

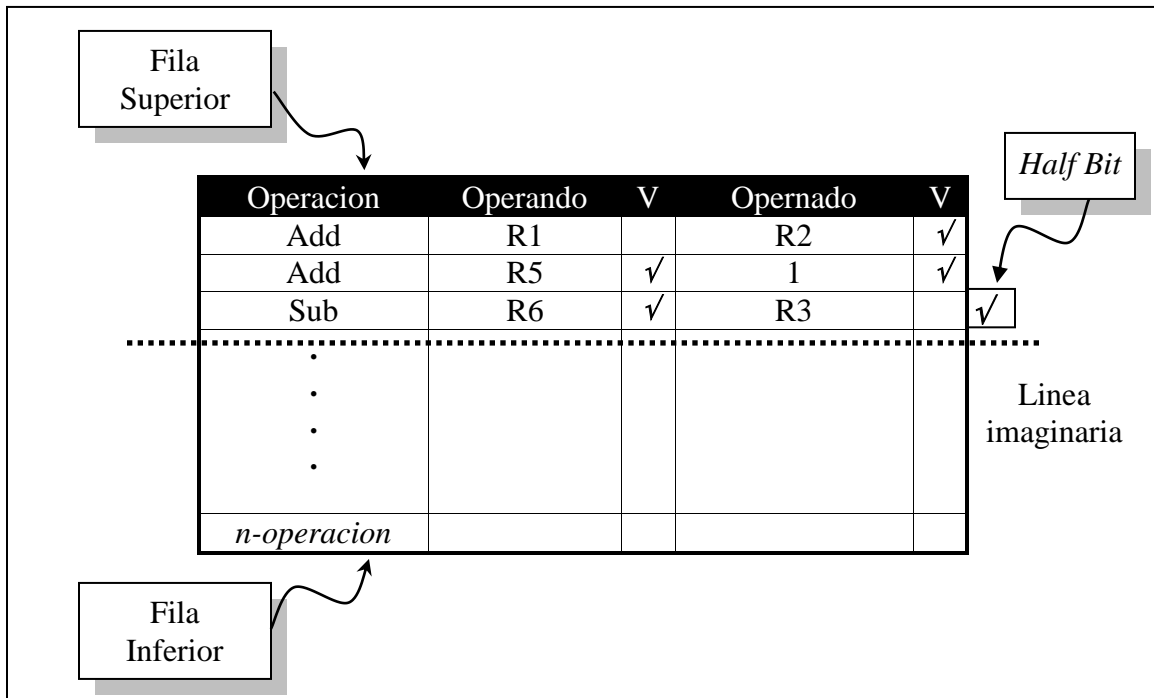


Fig. 3.31 División de Bloques en Filas

De la misma forma en que se modificó el campo de ruta de habilitación de la TMB para el submodelo anterior, se modifica para este submodelo. La diferencia radica en que el segundo y tercer bit de cada conjunto de bits indica si el receptor se encuentra en la fila superior o inferior (fila superior e inferior respectivamente), como vemos en la figura 3.32

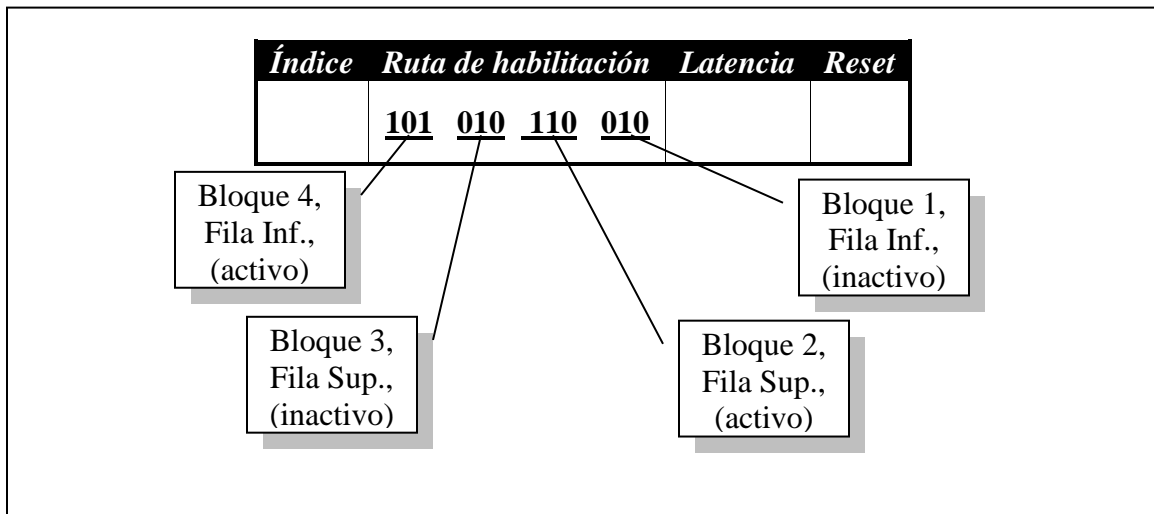


Fig. 3.32 Campos de la TMB para submodelo de División de Bloques en Filas

En este modelo se requiere modificar el bus que va de cada bloque hacia la *UAT*, ya que el modelo original maneja un bus de 2 bits (*Write* y *Full Bit*). El bit de escritura (*Write*), es utilizado para activar este bloque al momento de escribir una instrucción de entrada a la ventana de instrucciones. El bit de lleno o completo (*Full*), es usado para mantener la lógica de ubicación de instrucciones en la ventana (secuencial). A este bus se le adiciona otro bit, el cual es llamado *Half Bit*, o bit de mitad. Este bit es activado cuando la entrada intermedia del bloque es ocupada. Esta bandera o bit, indica a la *UAT*, que la siguiente instrucción será ubicada en la parte superior del bloque (bit apagado) o en la parte inferior (bit encendido). La política de ubicación se maneja de acuerdo a los lineamientos de la tabla 3.1.

<i>Full Bit</i>	<i>Half Bit</i>	Condición
0	0	Instrucción en la parte alta del bloque
0	1	Instrucción en la parte baja del bloque
1	1	Ubicar instrucción en el siguiente bloque
1	0	Estado no válido

Tabla 3.1 Bus de información de bloques divididos en filas

Los resultados en las gráficas de las figuras 3.33 y 3.34, muestran un alto índice de comparaciones en la parte superior de cada bloque. Sin embargo, este submodelo tiene la desventaja que ataca el submodelo anterior (división de bloques en columnas), ya que se activan los dos comparadores de cada entrada activa, independientemente de su ubicación (parte superior o inferior). En este punto surge proponer un tercer submodelo.

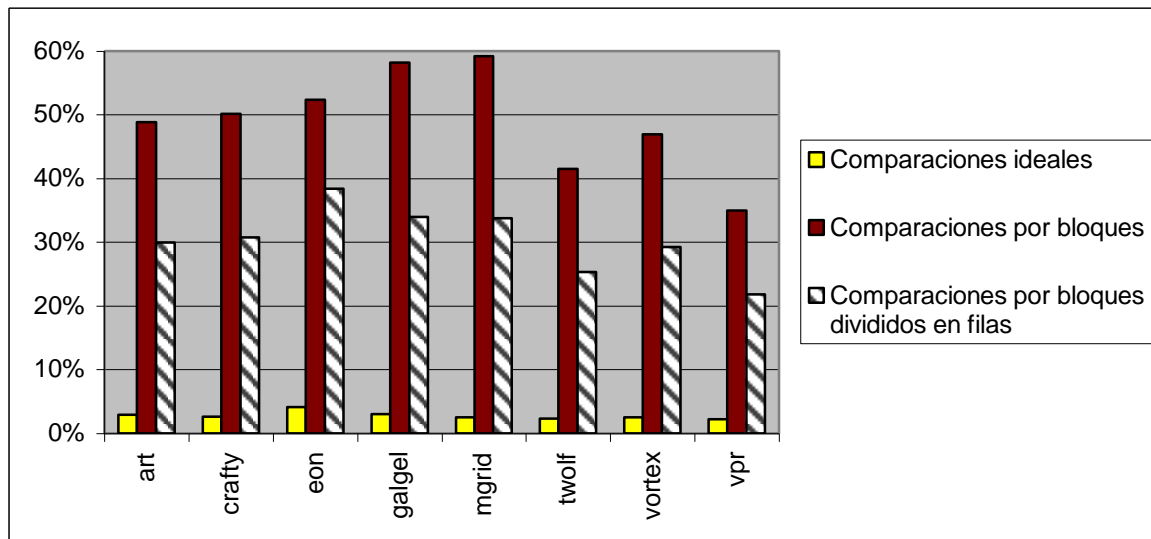
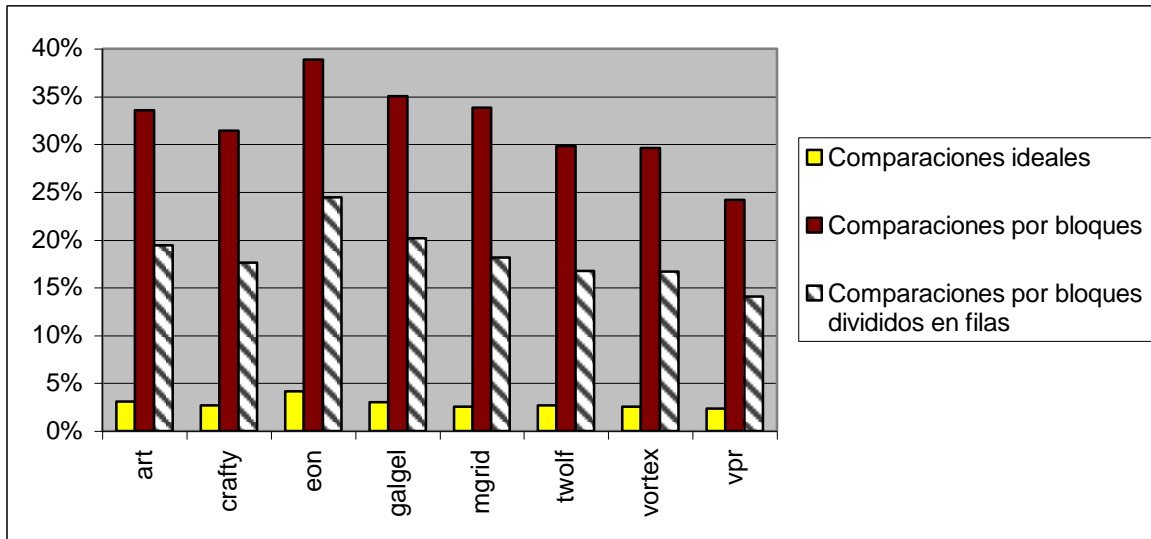
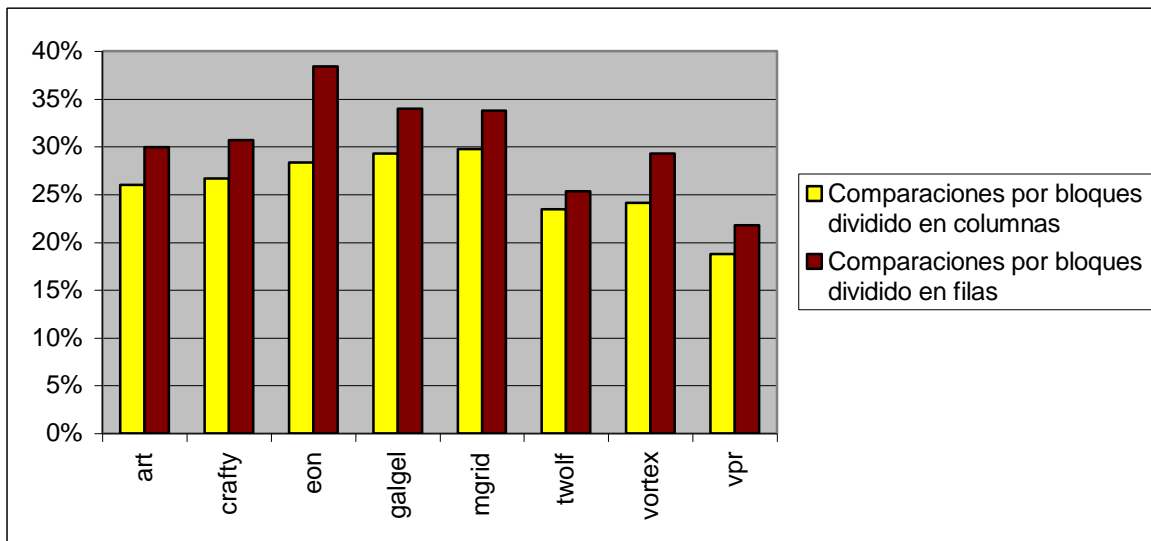


Fig. 3.33 Número de comparaciones ideales, por bloques y bloques divididos en filas (ventana de 64 entradas)



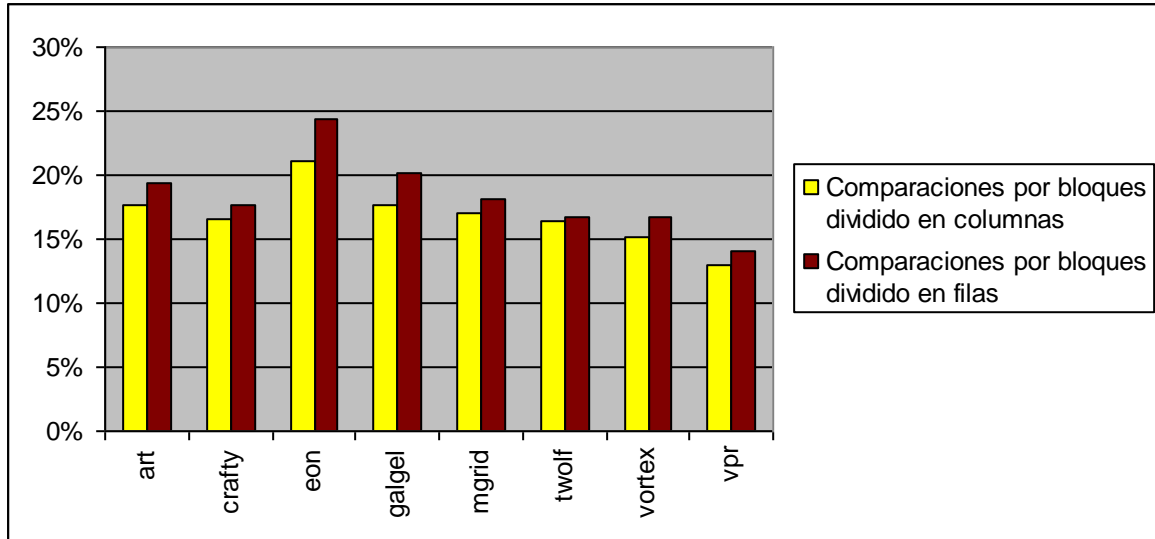
**Fig. 3.34** Número de comparaciones ideales, por bloques y bloques divididos en filas (ventana de 32 entradas)

Este submodelo presenta un mayor número de comparaciones que el submodelo anterior, la figura 3.35 y 3.36 muestran la comparación entre ambos submodelos en ventanas de 64 y 32 entradas respectivamente.



**Fig. 3.35** Número de comparaciones de bloques divididos en columnas vs bloques divididos en filas (ventana de 64 entradas)

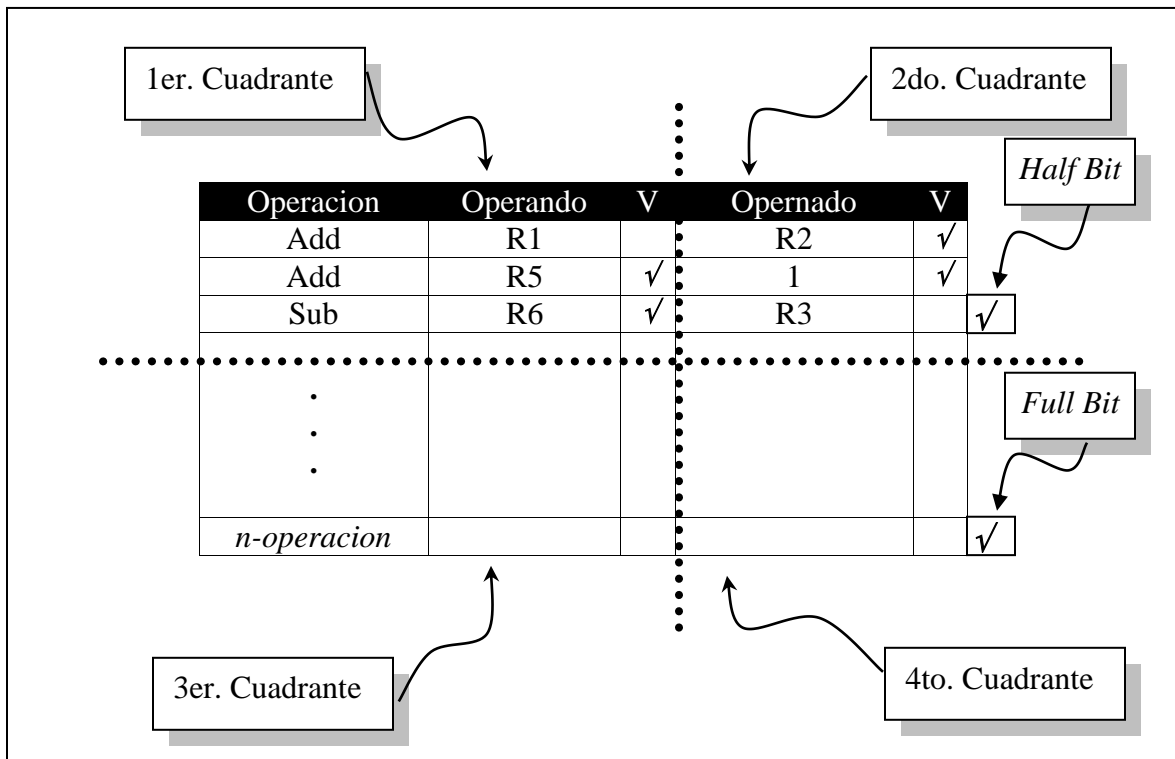




**Fig. 3.36** Número de comparaciones de bloques divididos en columnas vs bloques divididos en filas (ventana de 32 entradas)

*División de bloques en cuadrantes*

La figura 3.37, muestra la representación de cada uno de los bloques de la ventana de instrucciones.



**Fig. 3.37** Partición imaginaria de la ventana de instrucciones

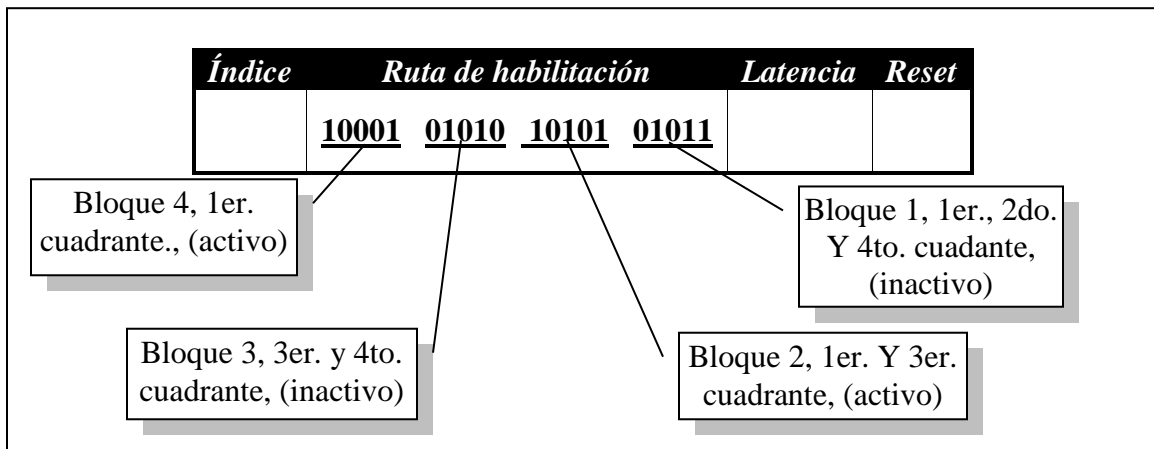
Para este submodelo se emplea todo el hardware implementado en los dos submodelos anteriores, de tal forma, que se hace uso de un bus de 3 bits (*write, full, half bit*), para determinar la ubicación de la instrucción (parte superior e inferior), además de la información proporcionada por el módulo de renombramiento de registros (sintaxis de instrucción = columna derecha/izquierda).

Toda información permite ubicar en cuadrantes los consumidores dentro de cada uno de los bloques de la ventana.

<b>0001</b>	Primer cuadrante
<b>0010</b>	Segundo cuadrante
<b>0100</b>	Tercer cuadrante
<b>1000</b>	Cuarto cuadrante

Debido a que puede existir el caso en que un dato es esperado por varias instrucciones, cada una de las opciones anteriores puede traslaparse, esto es, que cada posición de la palabra de cuatro bits representa uno de los cuadrantes. Esta palabra de cuatro bits se adiciona a los cuatro bits con que ya contaba el campo de ruta de habilitación de la *TMB*.

La *TMB* por supuesto vuelve a sufrir modificaciones en cuanto al tamaño de su campo ruta de habilitación, al igual que el bus *BIT* que parte de la *UAT* hacia la *TMB*, como lo muestra la figura 3.38.



**Fig. 3.38** Campo de ruta de habilitación de 20 bits

El campo de ruta de habilitación cuenta con 20 bits, los cuales se dividen en cuatro conjuntos de cinco bits para identificar donde está el posible receptor del dato, dentro de la ventana. El primer conjunto corresponde al bloque 4 (empezando de izquierda a derecha), el segundo conjunto al bloque 3, y así sucesivamente. Esto es, el primer bit (MSB) de cada conjunto significa si ese bloque debe de activarse, el segundo bit se interpreta que el receptor esta en el 4to. cuadrante (en caso de estar activo), y así en forma sucesiva.

Las figuras 3.39 3.40 muestran el comportamiento de este submodelo en comparación de los otros dos submodelos y el modelo original propuesto, manejándose una reducción de mas del 70% de operaciones innecesarias en la ventana de instrucciones, acercándose al número de comparaciones ideales.

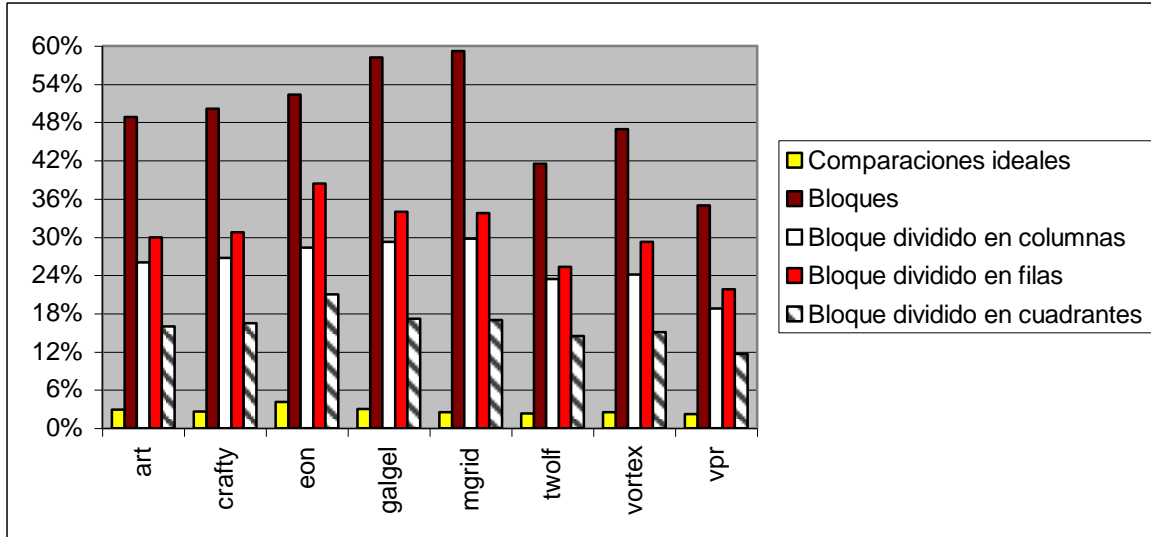


Fig. 3.39 Número de comparaciones por bloques y submodelos (ventana de 64 entradas)

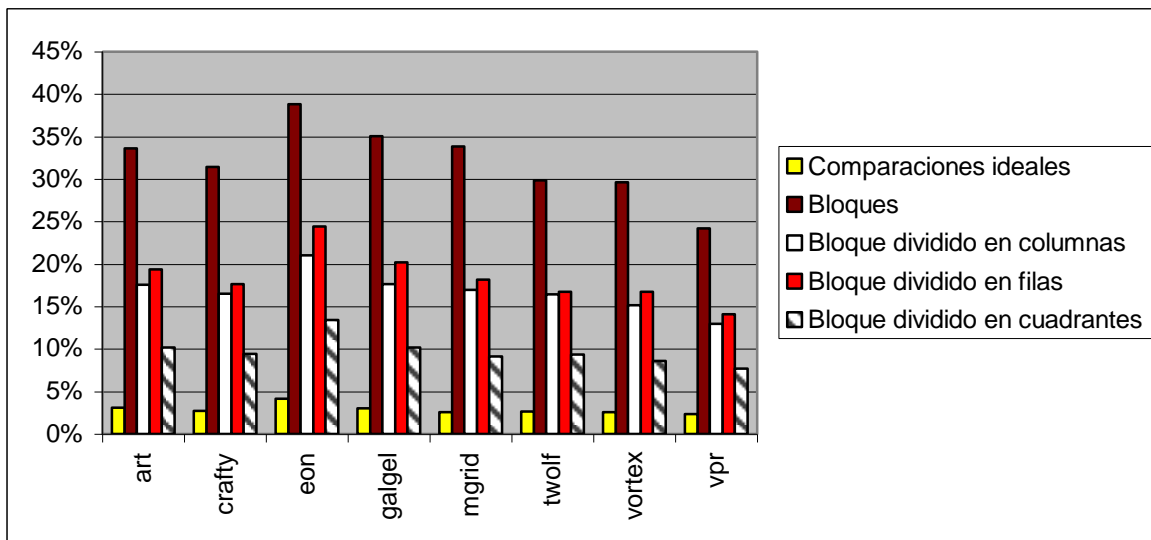


Fig. 3.40 Número de comparaciones por bloques y submodelos (ventana de 32 entradas)

La utilización de un módulo de control lógico (UAT), una tabla de direccionamiento (TMB) y varios buses de información, incrementan la complejidad del diseño en hardware, sin embargo se consigue una reducción del consumo de energía, en el mejor de los casos, de hasta 80% en comparación con los modelos de los procesadores actuales.

### 3.5 Ventana de instrucciones con detección de operandos por bit de identificación

En la búsqueda de un modelo que permita la identificación de operandos receptores del dato generado, se manejan diferentes esquemas. Otro modelo desarrollado, parte de la idea de subdividir la ventana de instrucciones en dos bloques de tamaño dinámico. Esta técnica, es similar a la empleada por la memoria caché tipo Hash-Rehash (*HR-Cache*) propuesta por Agarwal. En este modelo como se explica en el capítulo 4, se hace uso de un bit para dividir una caché de mapeo directo en dos bloques, de la misma forma se puede usar uno de los bits de las etiquetas de los registros fuentes, localizados en la ventana, para hacer una división en dos bloques principales. La figura 61 muestra los elementos que conformarían cada uno de los bloques dinámicos.

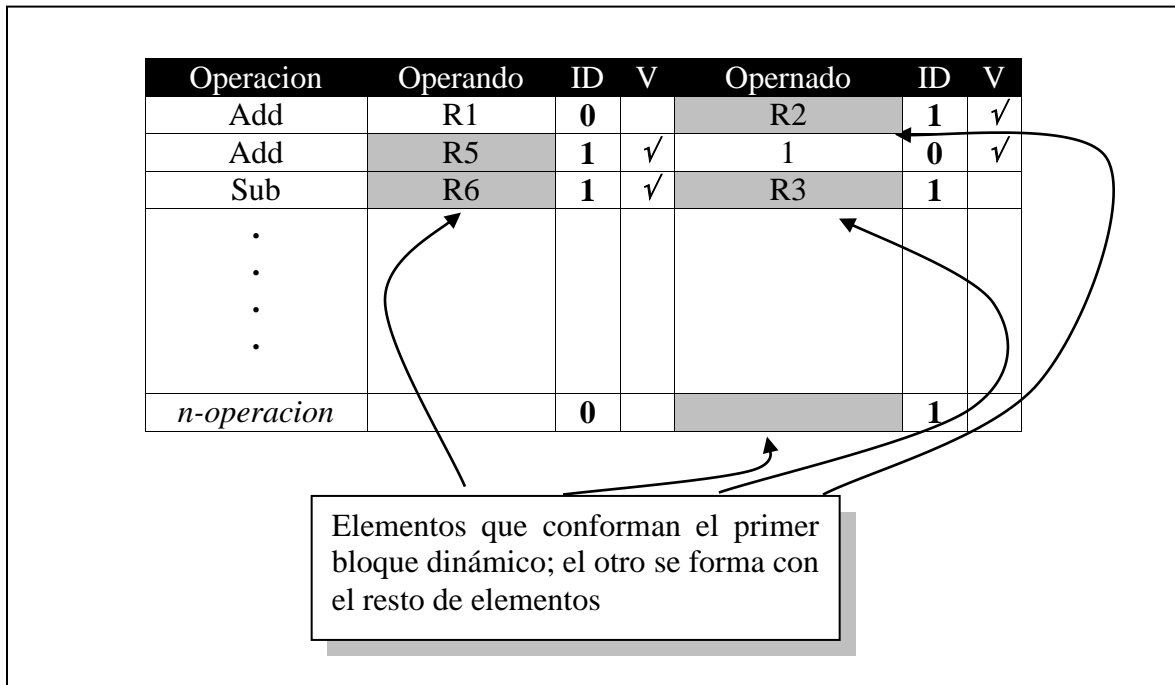


Fig. 3.41 Ventana de instrucciones dividida en bloques de tamaño dinámico

El campo ID de un bit se utiliza para identificar y formar dos bloques de tamaño dinámico, la principal diferencia con el modelo anterior, en el que el tamaño de los bloques es de tipo rígido y definido por el diseñador de la arquitectura. El bit ID es tomado de la etiqueta de cada uno de los registros fuente de la ventana de instrucciones. La posición dentro de la etiqueta de este bit, da pie a presentar diferentes modelos. Este bit es comparado contra el bit de la misma posición de la etiqueta del registro que contiene el dato que se acaba de generar en las unidades funcionales y se trae hacia la ventana para solucionar dependencias. Por ejemplo si el bit de identificación fuera “1”, solo se activarían los elementos que conforman el bloque que está sombreado en la figura 3.41. De esta forma se trata de reducir el número de comparaciones dentro de la ventana en una forma dinámica.

Si en la ventana sólo existiera un elemento con un bit de identificación “1”, representaría que solo se compara con un operando válido y se alcanzará el nivel de comparación ideal, sin embargo, en la practica eso es algo remoto. A continuación, se presentan cada uno de los submodelos que utilizan esta técnica desarrollada.

*Bit de identificación MSB (Bit más significativo)*

El bit mas significativo de la etiqueta del registro fuente es utilizado para conformar los dos bloques dinámicos en que se divide la ventana de instrucciones, de igual forma se maneja el bit MSB de la etiqueta del registro que contiene el dato generado en las unidades funcionales. La figura 3.42 muestra la gráfica de resultados de la implementación de este modelo.

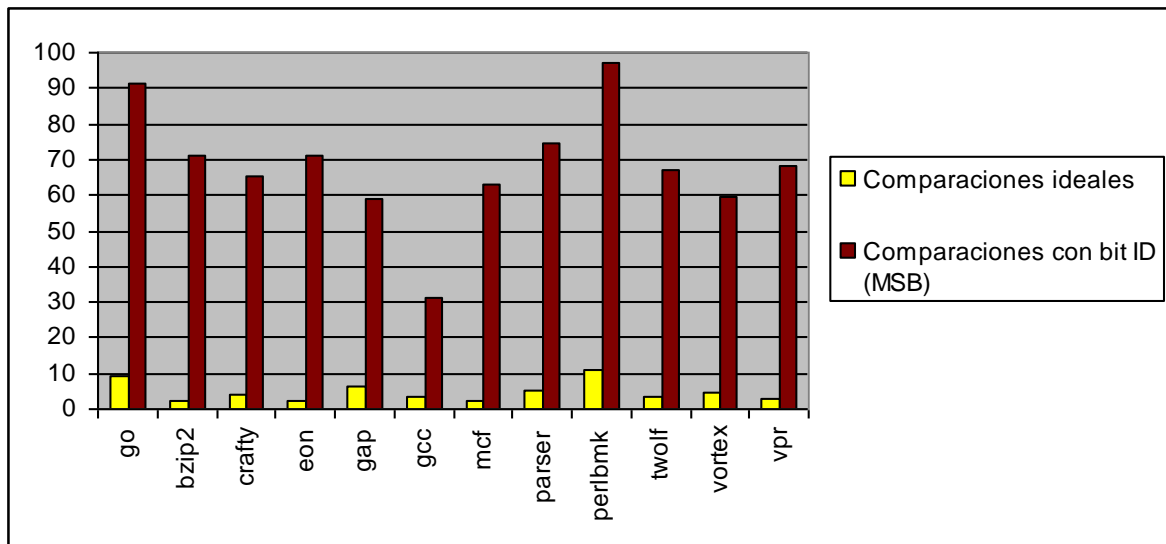


Fig. 3.42 Porcentaje del número de comparaciones ideales vs bit ID (MSB)

Para este submodelo la ganancia es de 30% debido a que muchas de las etiquetas comparten el mismo valor para el último bit, a saber valor “1”, por lo que no en todos los casos se tiene el desempeño deseado.

*Bit de identificación LSB (Bit menos significativo)*

Es el mismo esquema que el anterior, pero con la diferencia que se utiliza el primer bit de cada etiqueta para conformar los bloques dinámicos. La figura 3.43 muestra los resultados de esta implementación bajo las mismas cargas de trabajo que el submodelo anterior. En este submodelo, se presenta una ganancia de 48% promedio, lo cual representa una mayor reducción de comparaciones innecesarias en la ventana, en comparación del submodelo anterior.

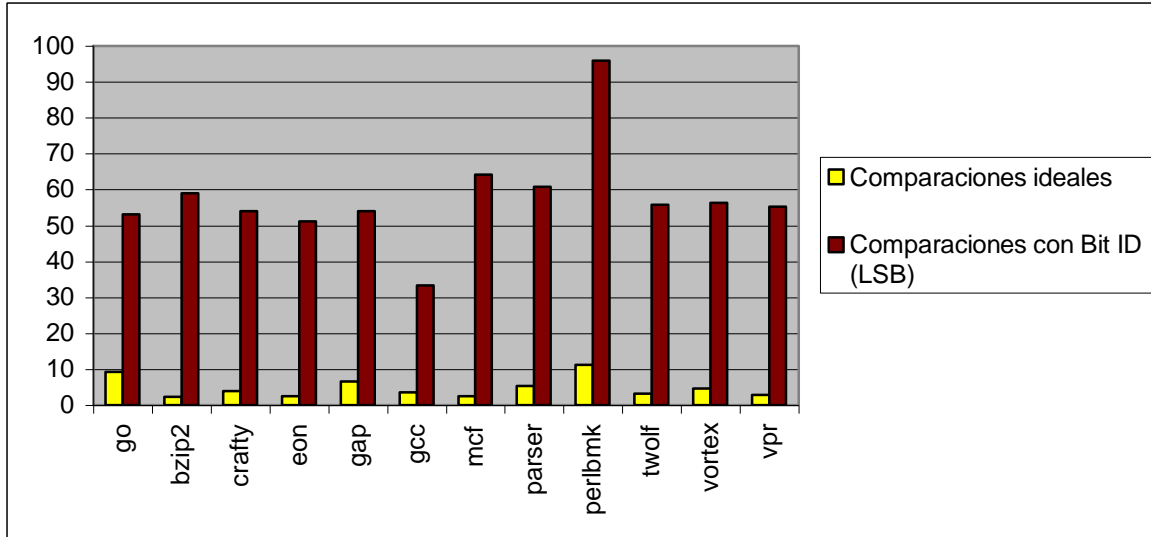


Fig. 3.43 Número de comparaciones ideales vs bit ID (LSB)

*Bit de identificación PMSB* (primer bit con valor uno mas significativo)

Este submodelo es denominado *Primer bit con valor uno mas significativo (PMSB)*, que consiste en identificar el primer bit MSB que contenga valor uno. Esta búsqueda del bit MSB con valor uno se restringe a los primeros cuatro bits (de izquierda a derecha) de la etiqueta del registro que contiene el dato recientemente generado. Teniendo la ubicación de este bit, dentro del grupo de 4 bits que se marca como límite, se compara contra el bit en la misma ubicación de las etiquetas de los registros fuentes dentro de la ventana de instrucciones, la figura 3.44 muestra como queda el campo ID de la ventana de instrucciones.

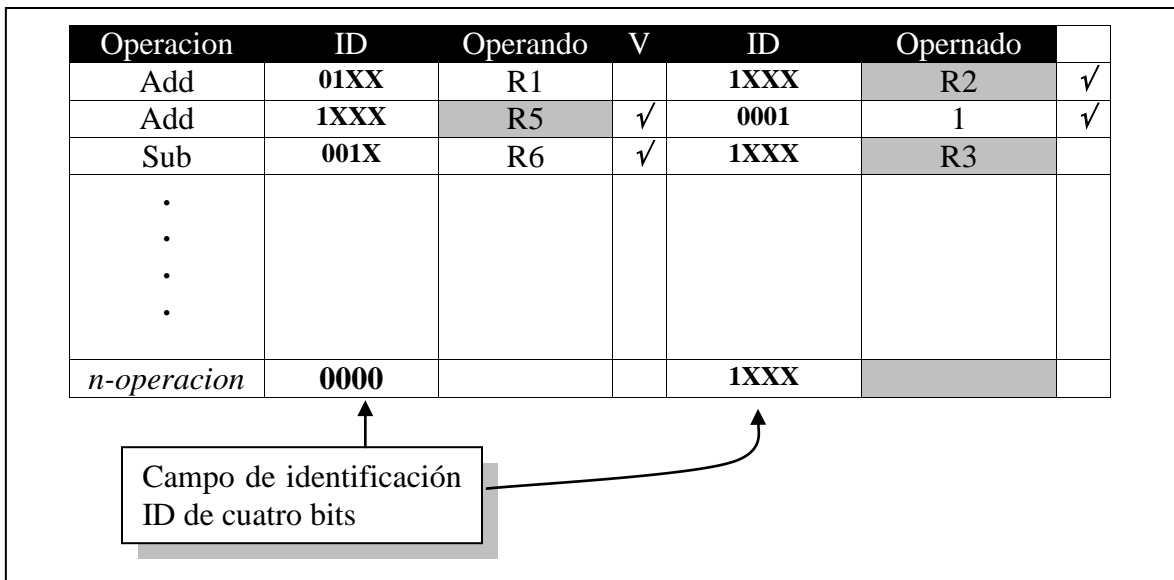


Fig. 3.44 Ventana de instrucciones modificada para comparación por ID

De esta forma, el campo de bit ID crece, de un solo bit de activación para los comparadores a un campo de cuatro bits, de tal forma que se pueden tener cuatro bloques de tamaño dinámico en el que se divide la ventana de instrucciones.

La forma en que se lleva a cabo la activación y la comparación de los elementos que conforman cada uno de los cuatro bloques dinámicos, es la siguiente:

- Se detecta el primer bit MSB con valor uno, de la etiqueta del registro que contiene el valor que se acaba de generar.

0101XXXXXXXXXXXXX

- Se activan solo los comparadores de las entradas de la ventana, para que las etiquetas de sus registros fuente tenga el mismo bit PMSB con valor uno.

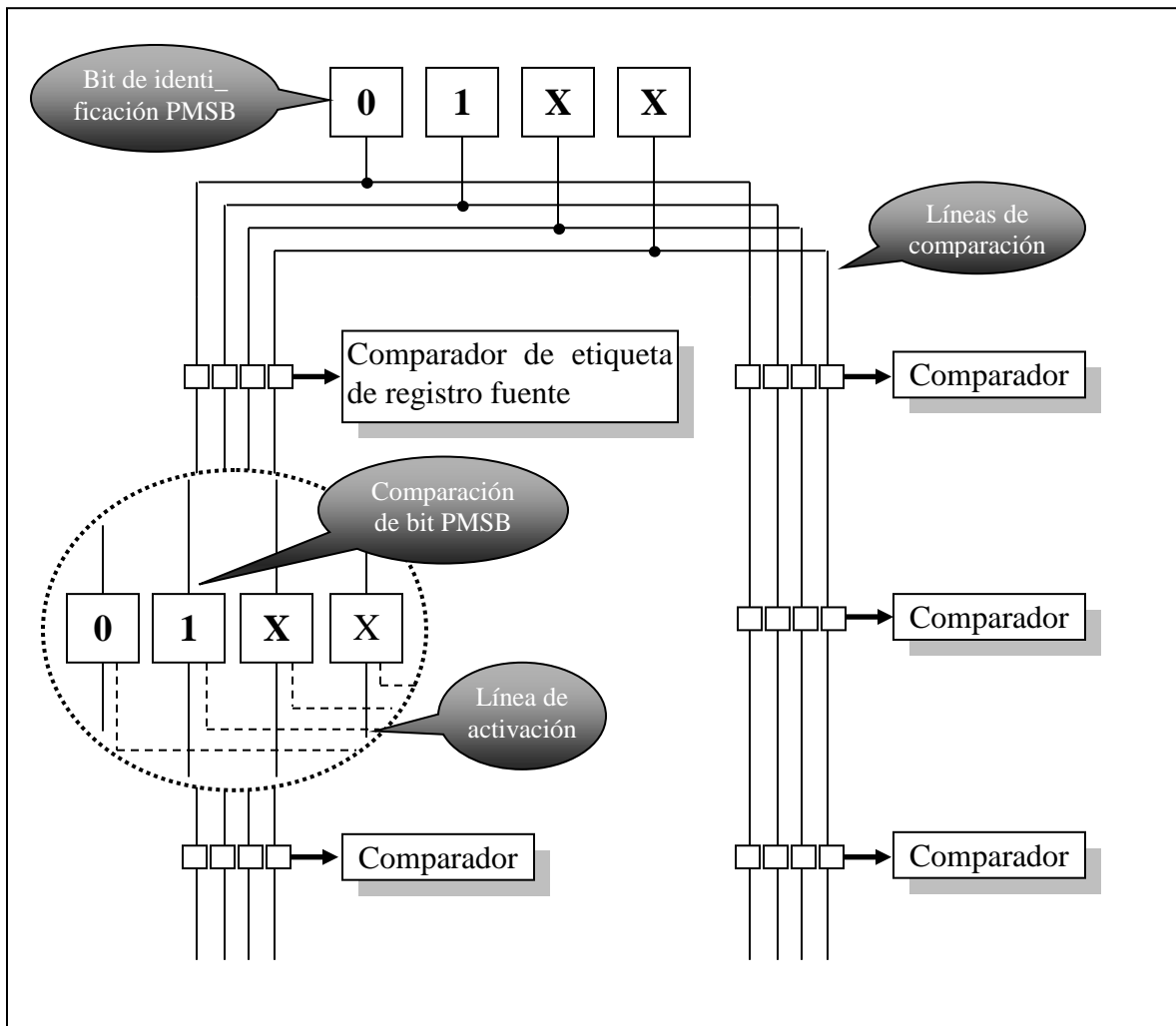


Fig. 3.45 Esquema de comparadores para identificación ID

Note que cualquiera de las cajas de comparación de la figura 3.45, puede activar al comparador de esa etiqueta, lo que representa que solo un bit del conjunto de cuatro bits que se pueden comparar, es el que realmente activa al comparador. Los resultados de esta propuesta se presentan en la figura 3.46.

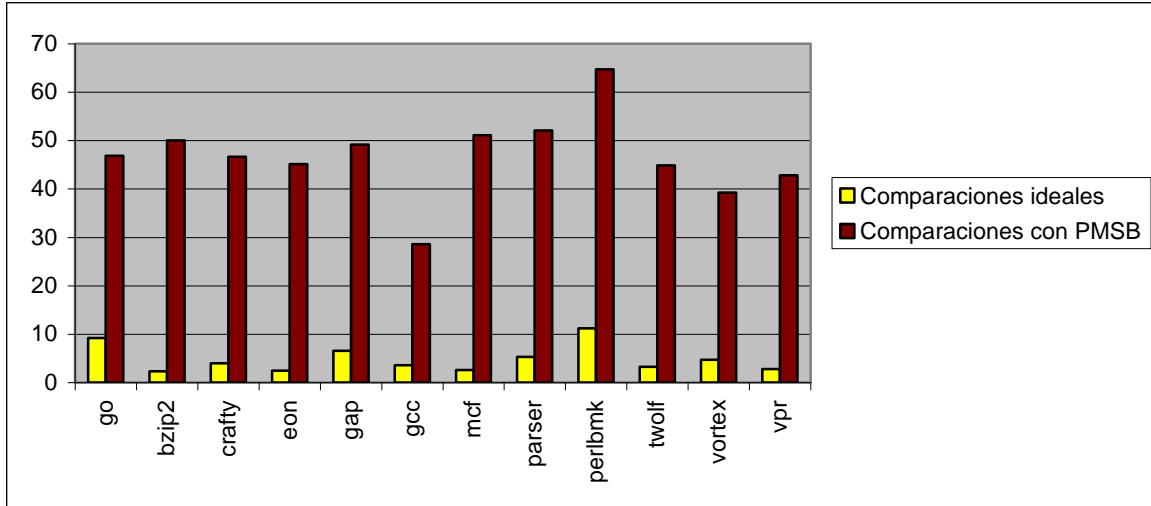


Fig. 3.46 Porcentaje del número de comparaciones ideales vs bit PMSB

El mismo submodelo PMSB se puede emplear en forma inversa, es decir, se busca el primer bit que tenga valor uno, pero del conjunto de cuatro bits de la parte menos significativa (LSB) de la etiqueta del registro, al cuál llamaremos PLSB.

XXXXXXXXXXXXX1010

Los resultados de la comparación de PMSB y PLSB aparecen en la gráfica de la figura 3.47, en el que se muestra como obviamente se tiene mayor diferencia entre los bits menos significativos de las etiquetas de los registros, lo que resulta en un menor número de comparaciones innecesarias para el submodelo PLSB.

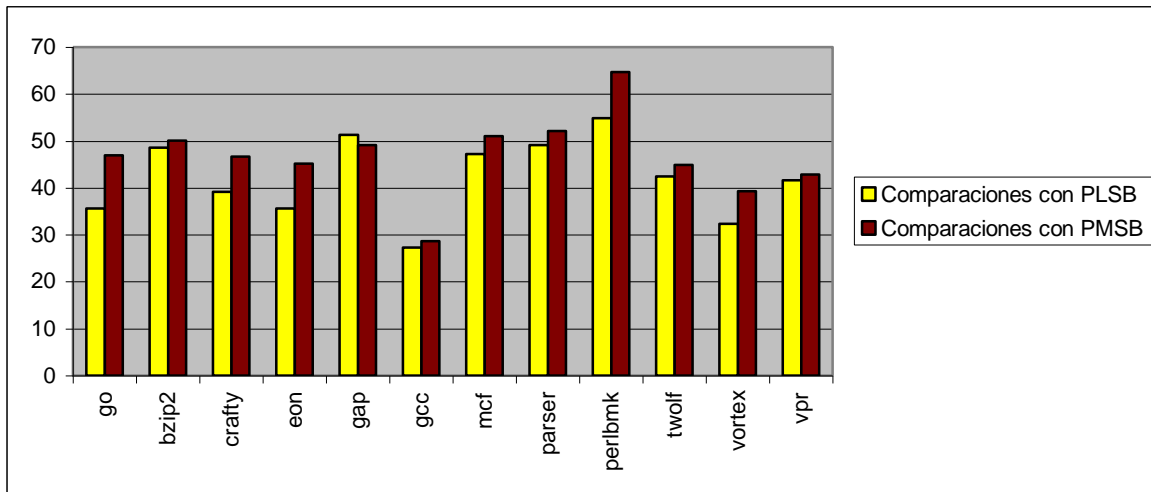


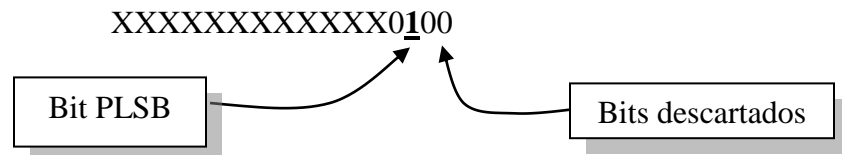
Fig. 3.47 Porcentaje del número de comparaciones con bit PLSB vs bit PMSB



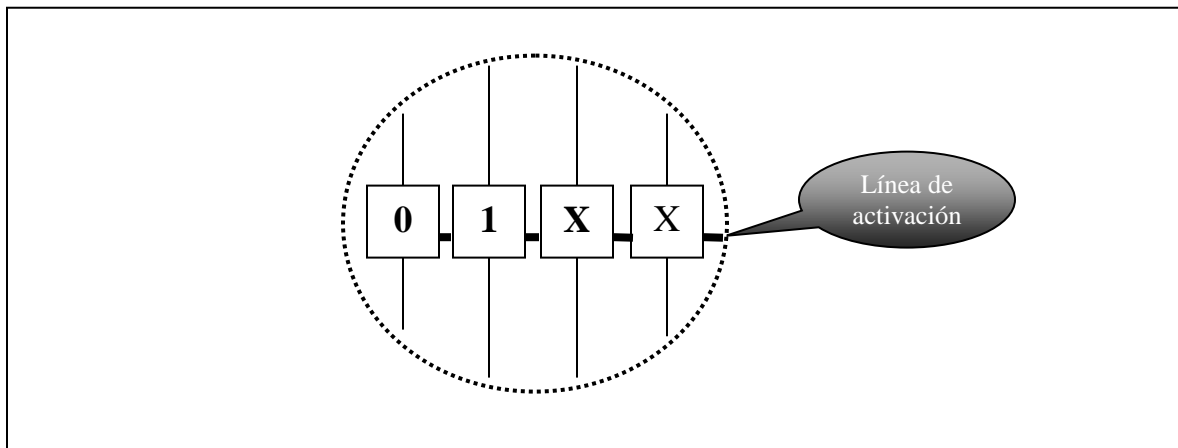
Sin embargo, es posible utilizar no solo uno bit, sino todos los bits que se descartan al ir en busca del bit PMSB o PLSB según sea el caso. El siguiente submodelo consiste en aplicar esa técnica.

*Conjunto de identificación dinámico (CID)*

Al tener implementada la detección del bit PLSB, submodelo con mejor resultado, se puede emplear también los primeros bits que se descartan al ir buscando el bit PLSB.



Esto representa que el conjunto de bits que activan los comparadores de las etiquetas en la ventana, puede ser variables, permitiendo una completa flexibilidad sobre el número de comparadores que se activan en busca de dependencias.



**Fig. 3.48** Comparadores del conjunto de identificación ID

En este caso, cualquier comparador con una etiqueta que tenga el mismo PLSB y los mismo bits antes de ese bit (conjunto de bits) significa que se activa ese comparador, lo que da pie a una búsqueda de operandos con dependencias, siguiendo una política de comparación casi ideal, ya que se reduce el número de comparadores que tienen los primeros bits iguales (fig. 3.48). Por ejemplo el bit PLSB fue hallado en la tercera posición partiendo de derecha a izquierda (100), lo que representa que se activan los comparadores con una etiqueta que tenga los mismos tres bits menos significativos (XXXXXXXX100).

Los resultados de este submodelo en comparación de los otros tres mencionados para esta técnica de bloques de tamaño dinámico se presentan en la figura 3.49

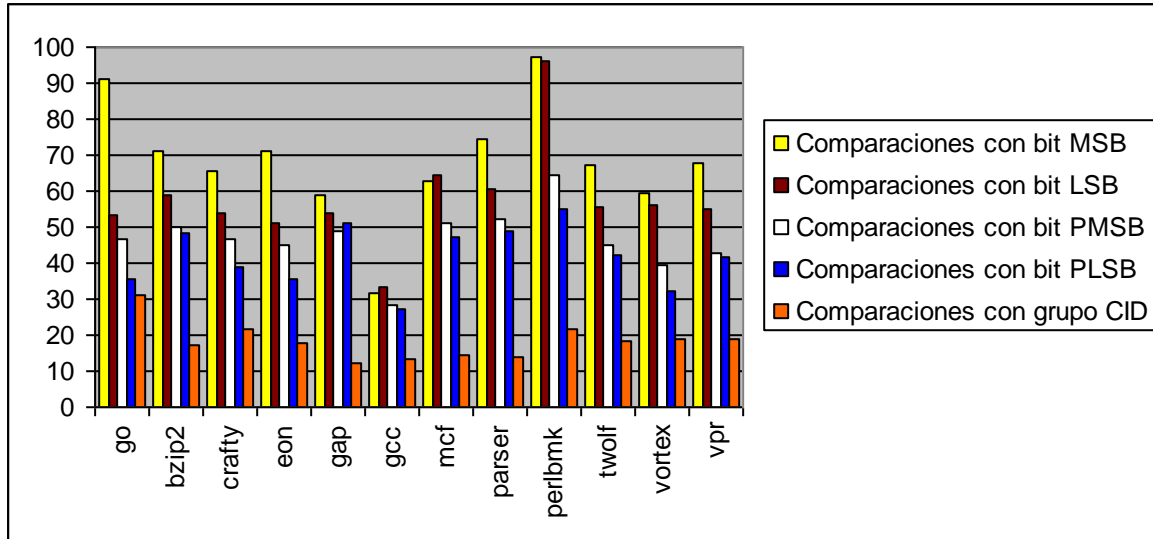


Fig. 3.49 Número de comparaciones con bit MSB, LSB, PMSB, LSB y grupo de bits CID

El submodelo con comparación de conjunto CID obtiene una reducción del 82% del número de comparaciones innecesarias, acercándose al número de comparaciones ideales, como lo muestra la gráfica de la figura 3.50.

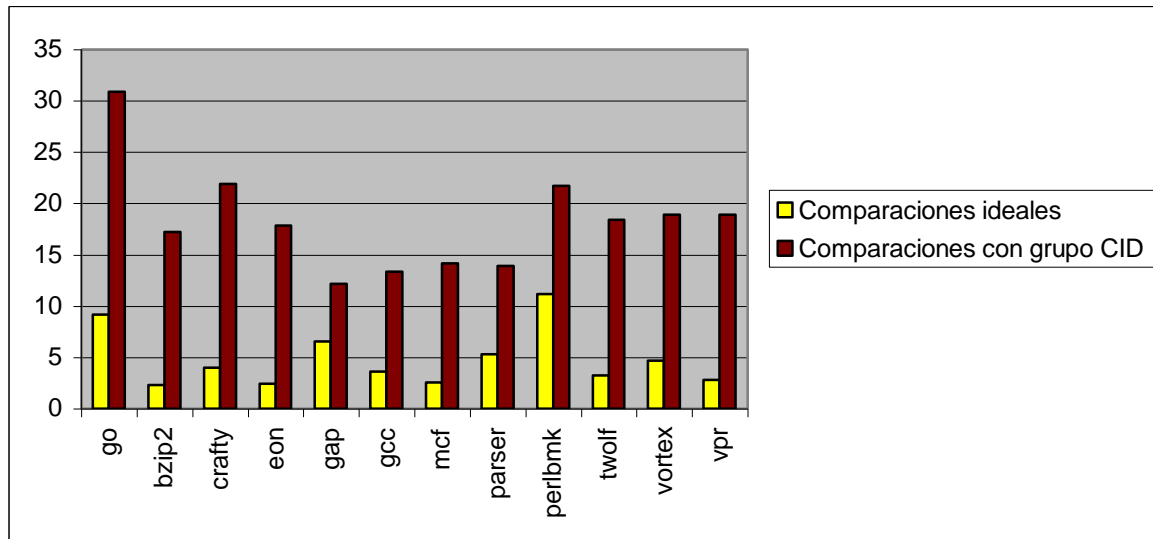


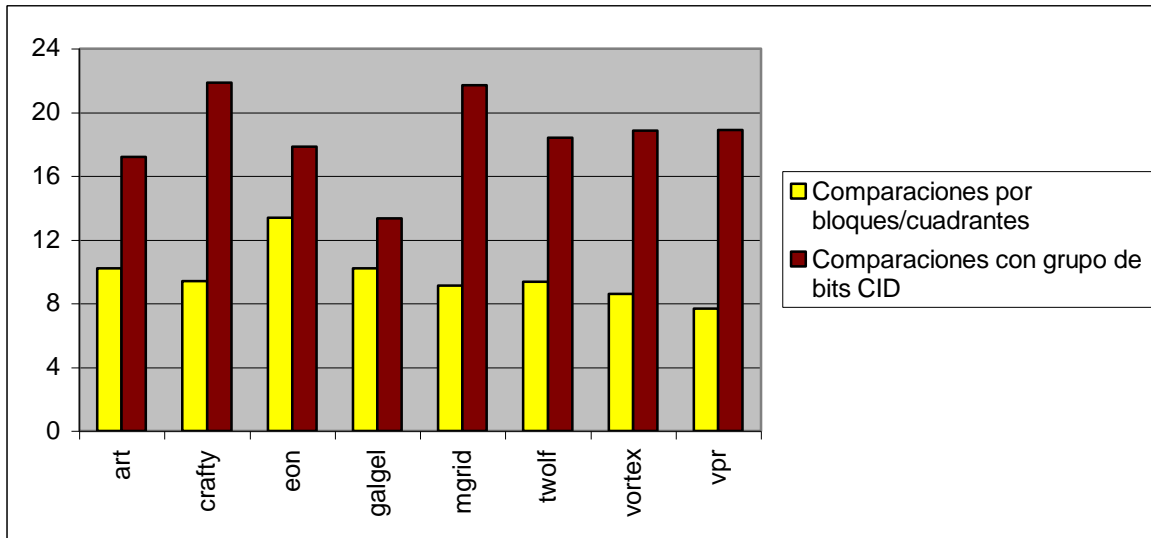
Fig. 3.50 Porcentaje del número de comparaciones ideales vs comparaciones con grupo de bits CID

### 3.6 Comparación y evaluación de modelos propuestos

Cada uno de los modelos propuestos y sus variantes, que ofrecen mejores resultados tiene sus ventajas. El modelo de ventana dividida en bloques y a su vez cada bloque dividido en cuadrantes ofrece un número de comparaciones muy cercano al número de comparaciones ideales, sin embargo, para poder implementar este modelo se requiere de mas elementos externos a la propia ventana de instrucciones (*UAT*, *TMB*, buses, etc.), a

diferencia del modelo de activación de comparadores por medio de bit de identificación ID, ya que este solo requiere apartar cuatro bits para activar los comparadores correspondientes al dato que se genera de las unidades funcionales.

La figura 3.51 muestra una comparación entre ambos modelos, con sus mejores variantes cada uno. Para el caso de la ventana dividida en bloques se usa el submodelo de división de bloques en cuadrantes, y en el caso de identificación de comparación por bit, se maneja el submodelo de activación por grupo de bits CID (conjunto de identificación dinámico).



**Fig. 3.51** Porcentaje del número de comparaciones de bloque/cuadrantes vs comparaciones con grupo de bits CID

Esta clase de resultados da pie a pensar en una combinación de ambos modelo, es decir, aplicar la división de la ventana de instrucciones en bloques, que a su vez están divididos en cuadrantes y que se aplique la activación de comparadores de cada cuadrante por medio del grupo de bits CID. Esta clase de implementaciones se deja para trabajos futuros, además, de que se toma en cuenta la complejidad de un modelo de tal proporción y su difícil manejo en términos de sencillez y rapidez, que a final de cuentas son parte de los factores que se toman en cuenta al momento de diseñar arquitecturas de microprocesadores.

# Capítulo 4

## Sistema de Memoria

### 4.1 Introducción

Desde los inicios de la computación, los diseñadores de procesadores han buscado la forma de tener memorias más rápidas, que sean capaces de satisfacer las demandas de datos por parte del procesador. Para ello se han implementado diversas técnicas para lograr este fin. El tema que compete a este capítulo se enfoca a la jerarquía de memoria; modelo que se ha generalizado en todos los procesadores, en donde se busca tener el máximo de rendimiento (menor número de errores) en el menor tiempo posible.

### 4.2 Antecedentes

Las innovaciones en microarquitecturas han sido concebidas, por lo general, con la observación del comportamiento de los programas. Los diseñadores frecuentemente introducen nuevas características a la microarquitectura para explotar los patrones de comportamiento del programa con el fin de mejorar el rendimiento del procesador.

Sin embargo, el avance en la mejora del rendimiento de las memorias no es tan acelerado como en el caso del procesador. Por lo que es necesario explotar características de comportamiento de las memorias con la finalidad de reducir el tiempo de acceso a ellas y el número de errores en cuanto a datos no encontrados.

Una de las principales etapas del proceso de ejecución de un programa es la recopilación de valores de la memoria y su escritura (instrucciones de carga y almacenamiento). En este proceso, se puede aprovechar la característica de la localidad espacial y la localidad temporal (*ítem 1.2.4*). Teniendo en cuenta estos conceptos es como se desarrolla la jerarquía de memoria, la cual consta de diferentes niveles de memoria con diferentes tamaños y velocidades.

#### 4.2.1 Predicción en memorias caché

Las memorias caché explotan la localidad de las referencias a memoria, una propiedad exhibida por muchos programas, reduciendo el tiempo de acceso a memoria, además en estudios recientes se ha propuesto técnicas de especulación basadas en la predicción del flujo de datos o de su localización (dirección) en la memoria.

Las memorias caché dentro del procesador han sido punto de estudio y de innovación en la aplicación de esquemas de predicción de valores o de direcciones de datos, sobre todo en la memoria de instrucciones; algunos modelos de predicción son de tipo de saltos o procesamiento especulativo de instrucciones posibles a realizarse en un tiempo futuro. En cuanto a la memoria de datos se ha trabajado en menor grado pero existen varias técnicas

que se han tratado de aplicar a ésta para favorecer su rendimiento en el número de accesos exitosos y la velocidad o frecuencia de la misma. Las siguientes técnicas están enfocadas a la memoria de datos dentro del procesador.

Las técnicas de predicción de valores se pueden categorizar de forma general en dos: la *estática* y la *dinámica*.

La *estática* es aquella que no cambia su información de predicción conforme se obtiene información de como está realizándose la ejecución del programa. Esta puede ser implementada en hardware dentro del microprocesador o también como parte del compilador de la arquitectura, basado en la descripción del valor usado en el momento de la compilación.

Un tipo de predictor *estático*, predice el mismo valor que tendrá una instrucción de escritura en un registro durante toda la ejecución del programa. En contraste el predictor *dinámico* adapta la información recopilada dinámicamente durante la ejecución del programa. Los predictores dinámicos a su vez pueden ser clasificados en *history-based* (basado en valores obtenidos con anterioridad) y *computation-based* (basado en computar el próximo valor o dirección del valor en base a los valores que se generaron con anterioridad).

Los predictores de tipo *history-based* toman el valor de un conjunto de valores previamente calculados y que se parecen al valor buscado, sin embargo, no le es posible determinar un valor que nunca ha sido calculado. La profundidad y tamaño de esta clase de predictor puede variar, pero el más utilizado es el que tiene tamaño de un solo valor.

Esta clase de predictor es comúnmente utilizado en memorias caché de datos, ya que se aprovecha la característica de localidad espacial del dato. Esto es, se guarda el dato o la dirección que se acaba de acceder, para que en el siguiente acceso se compare y se envíe el mismo dato en caso de ser ese el requerido. Su principal limitante es que el valor que se predice tiene que haber sido generado previamente durante la ejecución del programa.

Los predictores de tipo *computation-based* pueden generar nuevos valores durante la ejecución del programa. Al tener conocimiento de la relación que hay entre los datos (temporalidad y espacialidad), y considerando que puede realizar ciertas operaciones aritméticas sencillas, se puede calcular la próxima dirección de dato.

En el caso de memorias caché para datos, los predictores más aplicados de este tipo son: *Last Value* y *Stride*. El predictor de último valor (*Last Value*), el cuál en su forma más sencilla guarda el último valor consultado en la memoria, utiliza un contador de eventos, que lleva cuenta del número de veces que ha sido solicitado este dato y en base a este predice si la siguiente solicitud será este mismo; a esto se le llama mecanismo de histéresis.

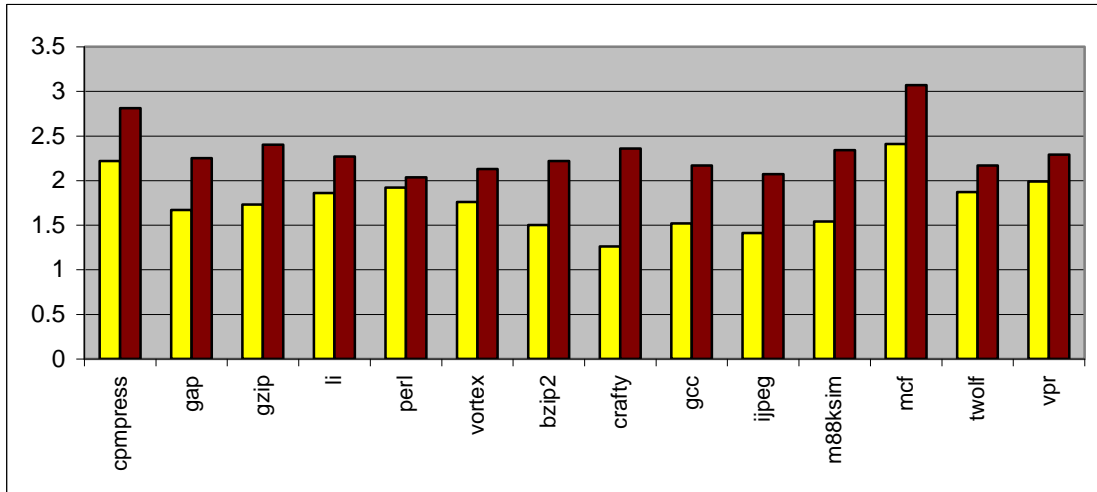
### 4.2.2 Mecanismos para mejorar el acceso a la memoria caché de datos

Se han propuesto varios esquemas con los que se puede disminuir el tiempo medio de acceso y mejorar la tasa de aciertos. Para explicar algunos de ellos se hace una profunda caracterización, lo cual es de vital importancia para tener una visión global del estado del arte. De los artículos publicados en la última década, sobre el tema que estamos tratando, podemos caracterizar tres tipos de sistemas de memoria caché propuestos:

- *Cachés Desacopladas:* El acceso al arreglo de datos y la selección de la línea, se realiza independientemente del acceso al arreglo de etiquetas y su comparación, esto con la finalidad de aprovechar la diferencia en tiempo de acceso que se tiene entre las trayectorias hacia los arreglos de etiquetas y datos.
- *Cachés Agregadas:* A una caché de mapeo directo se le agrega una caché más pequeña pero totalmente asociativa, con la finalidad de mejorar la tasa global de aciertos sin aumentar el tiempo de acceso.
- *Cachés de Múltiples Accesos:* Una caché de mapeo directo se accede de forma secuencial mas de una vez, con la finalidad de alcanzar el tiempo de acceso que proporcionan este tipo de cachés cuando se acceden en forma paralela, pero al mismo tiempo se trata de obtener la tasa de aciertos de una caché asociativa por conjuntos.

Actualmente existen varios modelos de caché que tratan de disminuir el porcentaje de error al tener una mejor política de reemplazo, manejo de datos reemplazados (basura o desecho de la caché), y búsqueda de datos. Si a esto le adicionamos que se puede disminuir el porcentaje de error al dividir la caché en varios bloques (nivel de asociatividad), se disminuye el tiempo de espera del procesador por un dato que tiene que ser extraído desde los siguientes niveles de memoria, disminuyendo el “miss rate”. El problema que presenta este tipo de esquemas, al dividir la memoria caché en varios bloques, es que se incrementa el tiempo de búsqueda del dato, ya que se duplica en comparación con una memoria de un solo bloque o de mapeo directo.

Cuando es referenciada una memoria asociativa, los bloques que la conforman son examinados en forma paralela. Cuando han sido leídas las etiquetas de los bloques, un comparador de etiqueta selecciona el bloque que contiene el dato deseado, y pasa el dato del bloque al procesador. El selector de etiquetas introduce un retardo extra entre la habilitación del bloque que contiene el dato deseado y el momento en que puede ser usado el dato por el procesador. Esto entre otros factores incrementa el tiempo de búsqueda de un dato en este tipo de memorias. La figura 2.1 muestra una gráfica en la que se compara como es que la memoria de acceso directo tiene una latencia de acceso menor en casi un 45% con respecto a la memoria de acceso asociativo.



Fi

g. 4.1 Latencia en ciclos (Mapeo Directo vs. Asociativa nivel dos )

Por otro lado, la memoria asociativa siempre ha demostrado tener un “miss rate” (porcentaje de error) más bajo que el de una memoria de acceso directo, en la figura 2.2 se muestra una diferencia de 20% a casi 90% en cuanto a “miss rate” entre estos dos modelos.

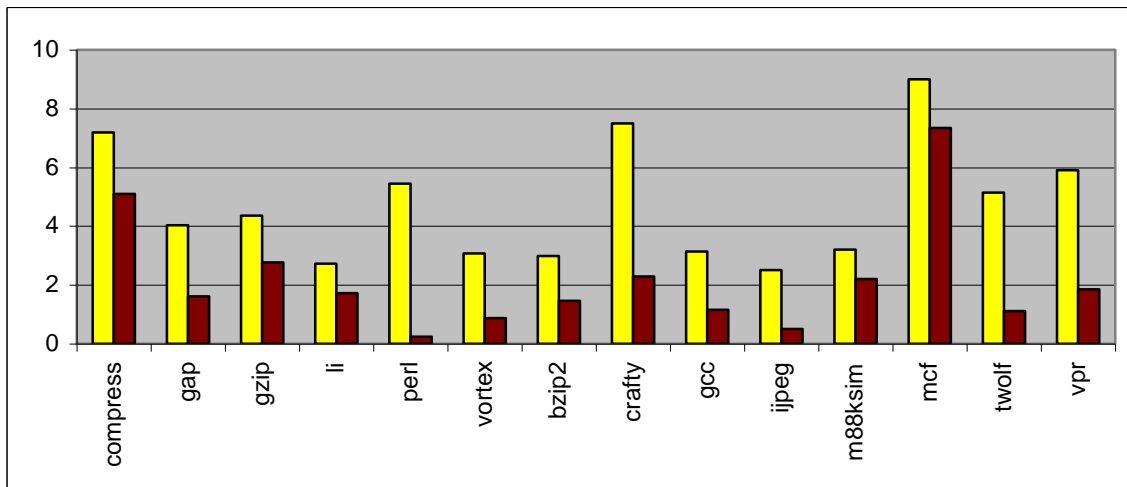


Fig. 4.2 Porcentaje de miss rate (Mapeo Directo vs. Asociativa nivel dos )

El diseño de una memoria que conjunte las ventajas de ambos modelos, mapeo directo y asociativa, ha sido el objetivo de muchos investigadores. Esta memoria debe tener un tiempo de acceso y un porcentaje de error bajos, es decir, no por tratar de obtener eficiencia sacrificar la velocidad de operación de la memoria. Para lograr este objetivo se han planteado diferentes modelos de acceso, manejo de errores, información reemplazada (basura) y modelos de predicción para localización del dato.

Una sencilla técnica para lograr esta clase de comportamiento es el uso de memorias asociativas pero con acceso secuencial. Las implementaciones de memorias caché de tipo secuencial o serial pueden ser divididas en dos categorías. Primero, pueden ser divididas

en la forma como son examinadas al momento de realizar una búsqueda de dato, ya que esto puede ser en un orden estático y rígido o de una forma dinámica, realizando la primera búsqueda no necesariamente en el primer bloque. Segundo, este tipo de memorias se pueden dividir por el tipo de política que usan para colocar sus datos en cada uno de los bloques, o de retirarlos de los mismos.

La memoria caché tipo *Hash-Rehash* (HR-Caché) propuesta por Agarwal [4] reduce el *miss rate* de una memoria de mapeo directo, y aunque este modelo puede implementar el nivel de asociatividad que se proponga el diseñador, tiene un desempeño por debajo de una memoria asociativa de nivel dos con política de reemplazo LRU, en cuanto al *miss rate*. Este modelo consiste en el uso de una memoria de acceso directo que es accedida por medio de dos funciones [5].

El modelo de memoria caché tipo *Column-Associative* (CA-Caché) de Agarwal y Pudar [6], mejora el *miss rate* y el tiempo de acceso de la HR-Caché; basándose en un funcionamiento similar a la HR-Caché la CA-Caché hace uso de una memoria de mapeo directo que se subdivide en dos bloques, si al realizar una búsqueda en el primer bloque el dato no es localizado, se dirige al segundo bloque, y posteriormente cambia la entrada de este segundo bloque hacia el primero, para que búsquedas posteriores localicen el dato en el primer bloque disminuyendo el tiempo de acceso. Sin embargo el uso de esta técnica implica un consumo extra de energía para conmutar los datos de entradas de los dos o más bloques que se implementen. En cuanto al *miss rate* sigue siendo más alto que el de una memoria asociativa de nivel dos.

Un modelo que conjunta las ventajas de las técnicas antes mencionadas es el propuesto por Calder y Elmer [7], la *Predictive Sequential Associative Cache* (PSA-Caché), que en vez de manejar una política de reubicación de datos se apoya de un bit que identifica la posible ubicación del dato, lo que disminuye el consumo de energía de la memoria CA-Caché, y hace uso de una política de reemplazo MRU y una tabla de un bit denominada tabla de *steering bit* o de dirección, con el que determina el bloque al que debe acceder al realizar la primera búsqueda. Según resultados presentados por estos investigadores, consiguen un *miss rate* similar al de una memoria asociativa de nivel dos con un tiempo de acceso similar o igual al de una memoria de mapeo directo.

### 4.3 Manejo de operaciones redundantes en la memoria caché

Siguiendo la tendencia de conformar el funcionamiento del hardware del procesador con el comportamiento de los programas, es necesario detectar el comportamiento de las cargas de trabajo sobre los modelos de memoria caché existentes en el mercado.

En la figura 4.2, la gráfica nos muestra que una memoria asociativa de nivel dos, tiene un mejor comportamiento en comparación con la tradicional memoria de mapeo directo. Sin embargo, existen varias técnicas con la finalidad de tener un rendimiento óptimo de la memoria.



```

1  struct parm { 2 int x;
3  int y;
4  };
5  struct parm pa = {3, 7};
6  struct parm pb = {2001, 2002};
7
8  void Compute(struct parm *p, int *result)
9  { 10 result[0] = p->x + p->y;
11  result[1] = p->x - p->y;
12  }
13  void Client()
14  { 15 int ra[2], rb[2];
16  Compute(&pa, ra);
17  Compute(&pb, rb);
18 ...
19  }

```

Fig. 4.3 Carga de trabajo redundante

Las operaciones redundantes de memoria pueden ser clasificadas en tres categorías:

- Redundancias en tiempo de ejecución (dinámicas)
- Redundancias parcialmente estáticas
- Redundancias completamente estáticas.

Las *redundancias en tiempo de ejecución* son la forma más común; las cargas y almacenamientos son redundantes si en la ejecución del programa estás operaciones accesan a la misma dirección de la memoria y usan el mismo valor, como una operación previa de memoria.

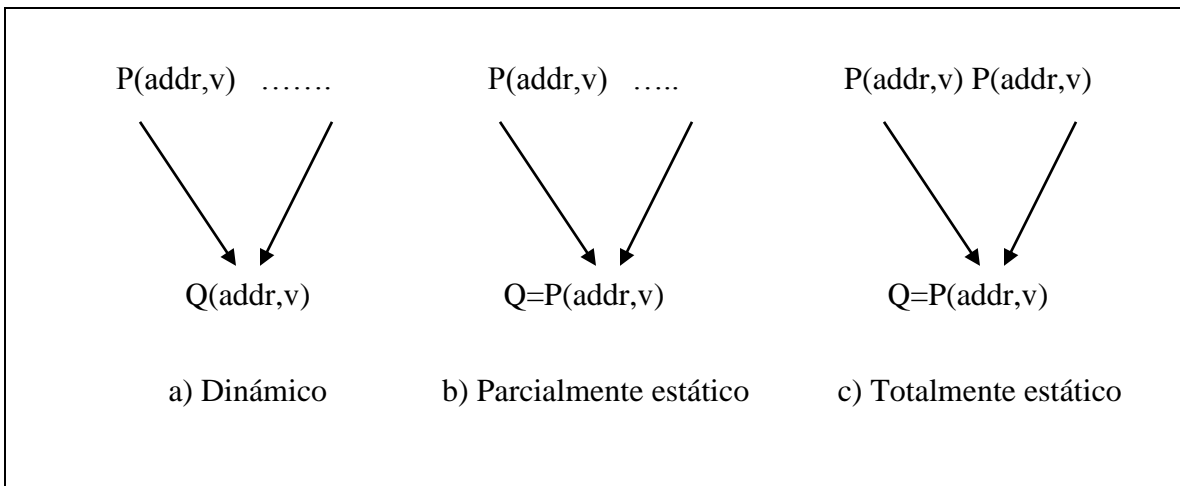


Fig. 4.4 Redundancias en tiempo de ejecución

Esto se muestra en la figura 4.4(c), donde las operaciones P y Q accesan a la misma dirección de memoria y operaran con en el mismo valor. A veces, el análisis constante

puede probar que en alguna dirección de flujo de control P y Q operan exactamente en la misma dirección y valor. A este caso, se le llama redundancia parcialmente estática, figura 4.4 (b). Si el análisis constante puede probar que P ocurre en todo flujo de operaciones posibles y que P y Q accesan a la misma dirección con mismo valor, entonces Q es una redundancia completamente estática, figura 4.4 (a).

Actualmente se han propuesto técnicas tanto en hardware [8][9][10][11] como en software [12][13][14][15], para detectar y remover operaciones de memoria redundantes. En el trabajo sobre rehúso de instrucciones dinámico [16], Sodani propuso el uso de tablas a las cuáles se accede para almacenar las entradas y resultados de las operaciones. Una operación redundante es identificada al comprobar con la tabla de historia que esa operación fue realizada anteriormente y por lógica se tiene de antemano su resultado. Si la operación se encuentra en la tabla, la operación actual es cancelada y el resultado es reutilizado. Para operaciones de carga (*load*), la dirección de la memoria puede ser usada para indexar una tabla del mismo tipo que la propuesta por Sodani. Para operaciones de almacenamiento (*stores*), la dirección y el valor almacenado son utilizados para buscar dentro de la tabla de historia la existencia de una operación anterior, detectando una operación redundante. Yang describe un esquema en hardware mas elaborado para la reducción de operaciones redundantes [17].

#### 4.4 Detección de comportamiento de una memoria caché de datos

Siguiendo la tendencia de conformar el funcionamiento del hardware del procesador con el comportamiento de los programas, es necesario detectar el comportamiento de las cargas de trabajo sobre los modelos de memoria caché que hay en el mercado. En la figura 4.5, observamos en la gráfica como una memoria asociativa de nivel dos tiene un mejor comportamiento en comparación de la tradicional memoria de mapeo directo y uno de los modelos más recientes de memorias asociativas de acceso secuencial (PSA-Caché).

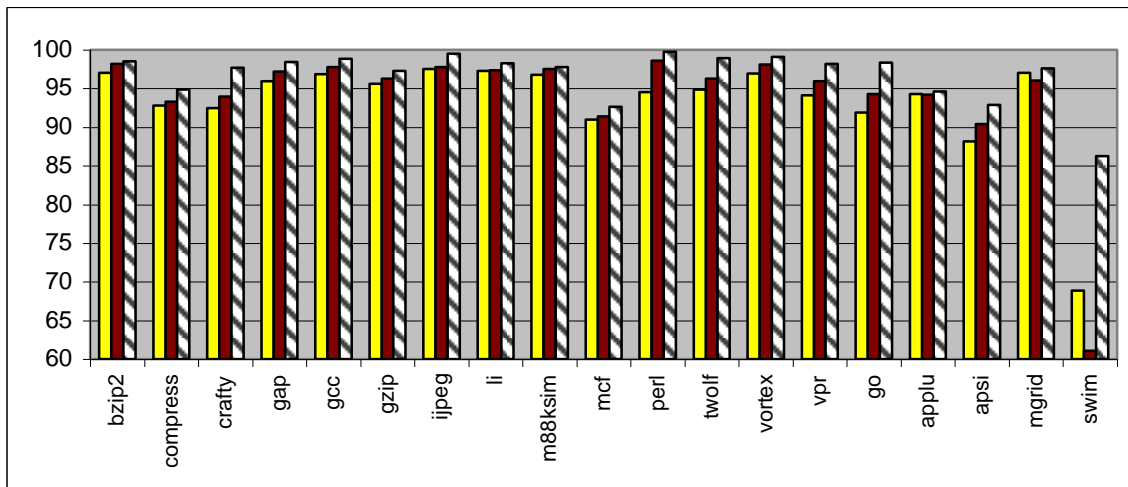
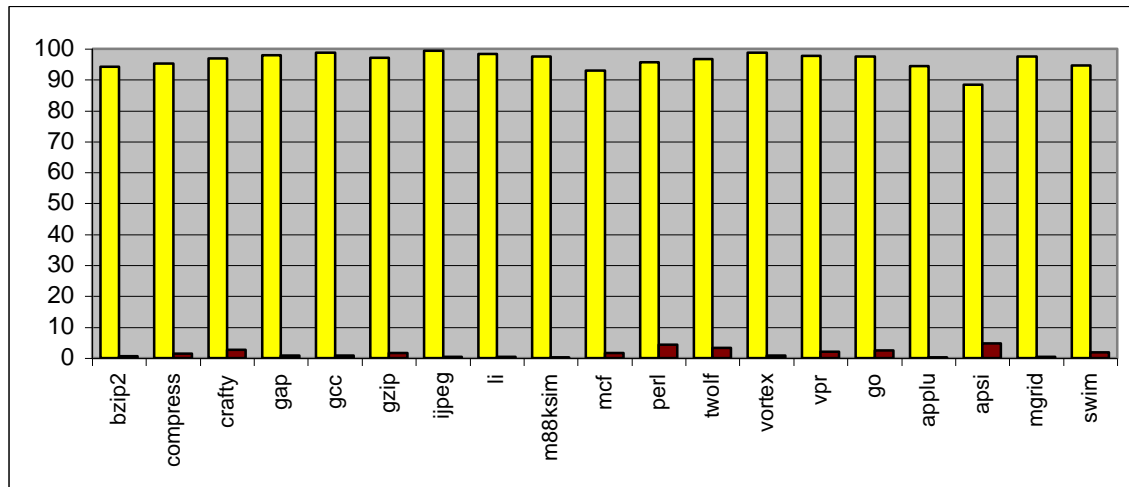


Fig.4.5 Hit rate % (Mapeo Directo ■, PSA-Caché ■, Asociativa Nivel dos ▨ )

A pesar de que los datos pueden ser almacenados en forma aleatoria dentro de la memoria caché asociativa por conjuntos, la mayoría de los datos se alojan en uno o dos conjuntos de la memoria, sin ocupar al 100% de los conjuntos. La figura 4.6 muestra como más del 90% de los aciertos se localizan en el bloque uno de la memoria asociativa (32 k) de dos vías.



**Fig. 4.6** Hit rate (%) por cada conjunto de una memoria asociativa nivel dos (AS-N2, conjunto uno ■, conjunto dos ■ )

Teniendo en cuenta que el tiempo de acceso de una memoria asociativa de nivel dos es de dos ciclos, se sabe por la gráfica que la búsqueda del dato que se está realizando en el segundo conjunto es poco exitosa en la mayoría de los casos, por lo que se manejan modelos de acceso de tipo secuencial o hasta selectivo (ítem Pág.77), con la finalidad de reducir la latencia de acceso de este tipo de memorias. Es en este punto, donde se puede determinar un comportamiento fácilmente predecible, junto con la presencia del fenómeno de localidad de los datos dentro de la memoria se plantea el modelo de *caché de acceso predictivo a arreglos* (PAAC, Predictive Access Array Cache).

### 4.5 Caché de acceso predictivo a arreglos (PAAC)

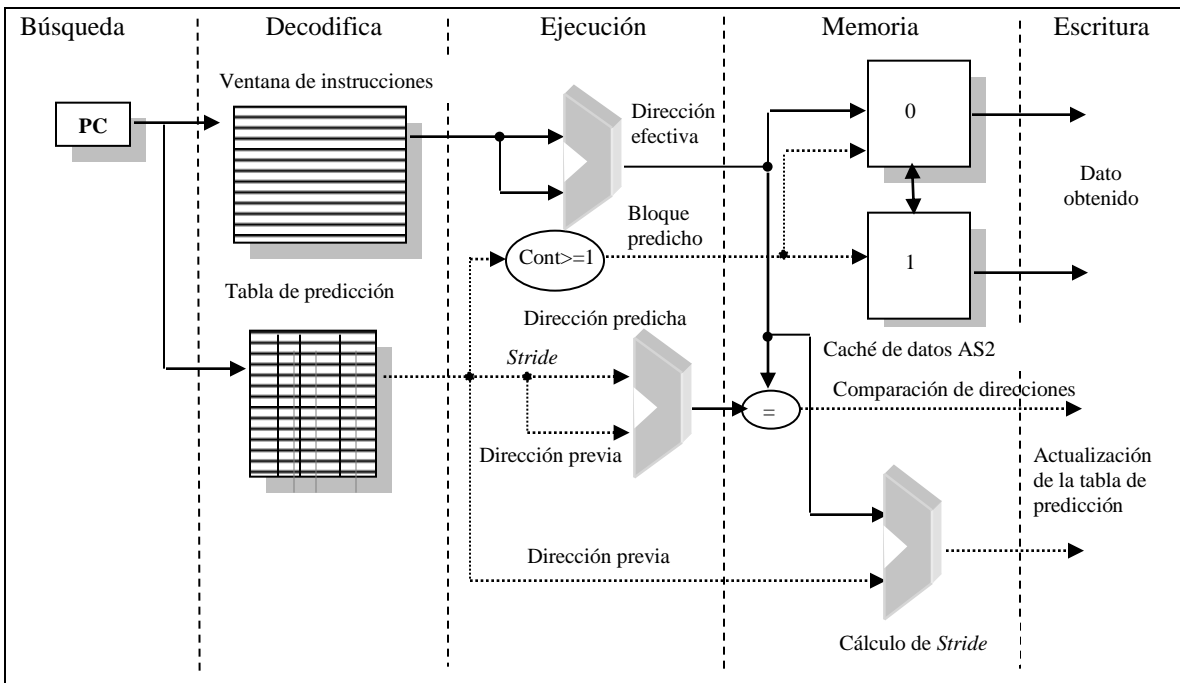
El modelo presentado en esta sección tiene como finalidad reducir la latencia de acceso al primer nivel de la jerarquía de memoria (caché) en la parte de datos (caché de datos, *d11*). Teniendo en cuenta que el tiempo de acceso de una memoria asociativa de nivel dos es de dos ciclos, se plantea que la búsqueda del dato en la memoria no se haga simultáneamente en ambos bloques, lo que consume primero un ciclo de reloj, posteriormente comparar los valores que arrojan cada uno de los bloques y selecciona el válido por medio de un multiplexor en un segundo ciclo de reloj.

Sin embargo, si todos los accesos se dirigieran primero al conjunto uno más del noventa por ciento de los accesos se lograría en un ciclo de reloj en vez de dos, en caso de no encontrar el dato en este conjunto se haría el acceso al segundo bloque, claro está con una penalización de un ciclo adicional, sin embargo se consigue un porcentaje de error aún por debajo del de una memoria de mapeo directo o de un sólo conjunto. Como este

comportamiento es fácilmente predecible se pensó en el uso de un predictor en base a stride [18] [19] [20].

El predictor en base a *stride*, en su forma más sencilla predice el próximo valor al determinar el comportamiento del contador del programa. La idea básica es predecir futuras referencias a la memoria en base a la historia pasada para las mismas instrucciones de acceso a memoria. El esquema de predicción más intuitivo, es el que tiene varias iteraciones de la misma búsqueda con sucesos anteriores. Esto es, cuando el Contador del Programa (PC) decodifica una instrucción de carga/almacena, se revisan las entradas de una tabla de referencia para ver si existe un historial de esa instrucción. Si no existe, se usa una entrada de esta tabla para almacenar el historial de dicha instrucción. En caso de existir la entrada y la referencia para la siguiente iteración (predecible), se realiza una prebúsqueda. Este esquema básico envuelve el uso del PC y de una tabla para almacenar el historial de instrucciones de carga/almacena que han accedido a la memoria caché de datos. Sin embargo para nuestro caso esta tabla servirá para determinar en que bloque se realizara la búsqueda del dato solicitado.

La PAAC en conjunto esta conformada por una tabla de predicción, una forma de acceso secuencial y una política de reemplazo LRU (Fig. 4.7). El acceso a la PAAC se realiza de la siguiente manera: si la predicción no se puede realizar, se accesa siempre al primer bloque de la memoria ya que se demostró (Fig. 4.6) que la mayoría de los datos se alojan en este; si el dato es encontrado en este bloque, se tiene una latencia de un ciclo de acceso, lo que es similar a una memoria de mapeo directo, pero de no encontrar el dato se accede al segundo bloque, lo que trae como consecuencia una latencia similar al de una memoria asociativa.



**Fig. 4.7** Datapath de predicción de conjunto a acceder de una memoria AS2secuencial (flujo normal  $\longrightarrow$ , flujo de predicción  $\cdots\longrightarrow$ ).

El camino que se sigue para predecir el bloque en el que se ubica el dato buscado inicia con el uso del *Program Counter* de la instrucción de acceso a la memoria, mientras por un lado se calcula la dirección efectiva de la memoria mediante una suma (consume por lo menos un ciclo), por otro, se indexa la tabla de predicción de 128 entradas, en la cual se verifica el valor del contador para esa entrada. Si el estado del contador permite la predicción, se accede al bloque indicado por el campo *bloque* en esa misma entrada de la tabla.

En caso de acierto se logra un acceso en un tiempo de un ciclo, similar al de una memoria de mapeo directo, sin importar si el dato estaba en el bloque primario o secundario. En caso de un error de predicción se accede al bloque contrario al de la predicción, lo que trae como consecuencia la penalización de un ciclo adicional al que ya se usó para revisar el bloque predicho.

El llenado de la tabla de predicción se realiza al comparar el *stride* actual con el *stride* almacenado en la tabla de predicción. El *stride* es obtenido de la diferencia de la dirección anterior con la dirección actual, operación que puede ser realizada solo hasta que termina el cálculo de la dirección efectiva. Además, se realiza una comparación entre la dirección predicha (suma de dirección previa con *stride* almacenado) y la dirección efectiva. Si ambas comparaciones son acertadas se incrementa el contador de la entrada de la tabla, además de verificar en que bloque se encontró el dato dentro de la memoria caché, para actualizar el campo *bloque* de la tabla. En el próximo acceso a la memoria se verifica la tabla de predicción y en base al contador de eventos de *stride* y del campo *bloque* es posible predecir en que parte de la memoria se encuentra almacenado nuestro dato.

Si la dirección del dato es predecible, se accede al bloque indicado por la tabla de predicción; si el dato es localizado en este bloque, se conserva la latencia de mapeo directo, pero de no encontrar el dato se accede al otro bloque en busca de este. Si la predicción tuvo éxito y el dato se localizó en el bloque dos, se elimina el ciclo adicional de tener que buscar el dato en el primer bloque de forma secuencial y luego acceder al bloque dos. Al tener un porcentaje significativo de predicción se puede reducir el tiempo de acceso a la memoria.

### 4.5.1 Tabla de referencia de predicción

La tabla de referencia de predicción, organizado como una memoria Caché de instrucciones, mantiene el historial de previas referencias o accesos a la memoria de datos y les asocia un *stride* (distancia entre instrucciones conforme al PC) para cada instrucción de carga o almacenamiento. Las entradas de la tabla están conformadas por los siguientes elementos:

☞ *Etiqueta*: Dirección de la instrucción de carga/almacenamiento.

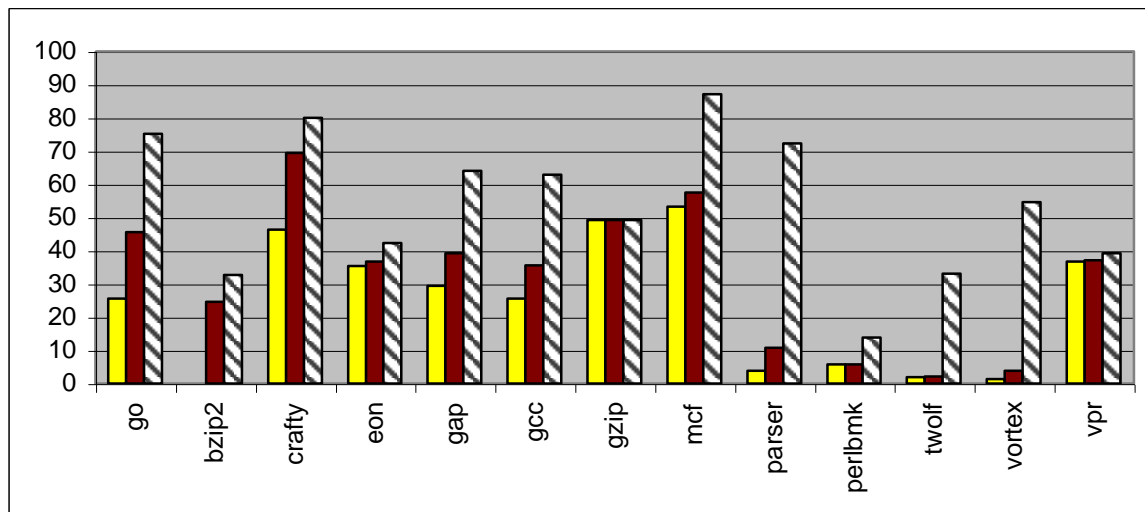
↻ *Dirección\_previa*: Última dirección (operación) que fue referenciada cuando el PC buscaba la instrucción.

↻ *Stride*: Diferencia entre las dos últimas direcciones que fueron generadas.

↻ *State*: dos bits que sirven como contador para establecer un historial del estado de los accesos de la instrucción. Al ser el primer acceso, el contador en uno; indica una condición inicial, la segunda vez el contador se incrementa y se considera una situación en la que es posible realizar predicción de la ubicación (bloque) del siguiente valor solicitado, y en el tercer estado (contador = 3) se da una condición de una predicción con alto porcentaje de certeza.

↻ *Bloque*: Se almacena la ubicación del dato, siendo el primer o segundo bloque de la memoria. Este campo es actualizado después de obtener el dato de la memoria y corroborar su ubicación.

El tamaño de la tabla de predicción originalmente se planteó de 32 entradas, sin embargo, al aumentar el tamaño de esta tabla se tiene un porcentaje de predicción más elevado. Obviamente, el porcentaje de error de predicción se incrementa, pero no en la misma proporción que las predicciones acertadas. En la figura 4.8 se aprecia el comportamiento del predictor para diferentes tamaños de la tabla de predicción con los SPECINT2000.



**Fig. 4.8** Porcentaje de *hit rate* de predicción (32 ent. ■ , 128 ent. ■ , 1024 ent.   , SPECINT )

Al igual que la figura 4.8, en la figura 4.9 se tiene el *hit rate* de predicción para diferentes tamaños de la tabla de predicción con los SPECFP2000 de coma flotante, en el que se muestra como a mayor tamaño de la tabla de predicción se tiene un mayor nivel de *hit rate*, lo que ayuda a un mejor rendimiento de nuestro modelo al tomar las consideraciones de medición de rendimiento expuestas en el capítulo 1. Las figuras 4.10 y 4.11 muestran el “miss rate” para los specs enteros y de coma flotante respectivamente. El grado de error para la tabla de 1024 entradas se incrementa notablemente.

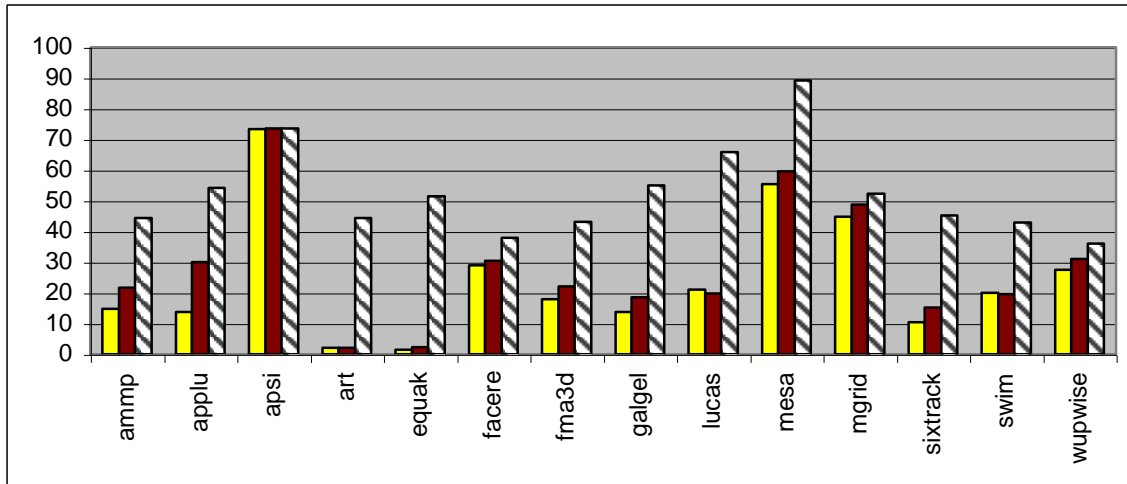


Fig. 4.9 Porcentaje de *hit rate* de predicción (32 ent. , 128 ent. , 1024 ent. , SPECFP)

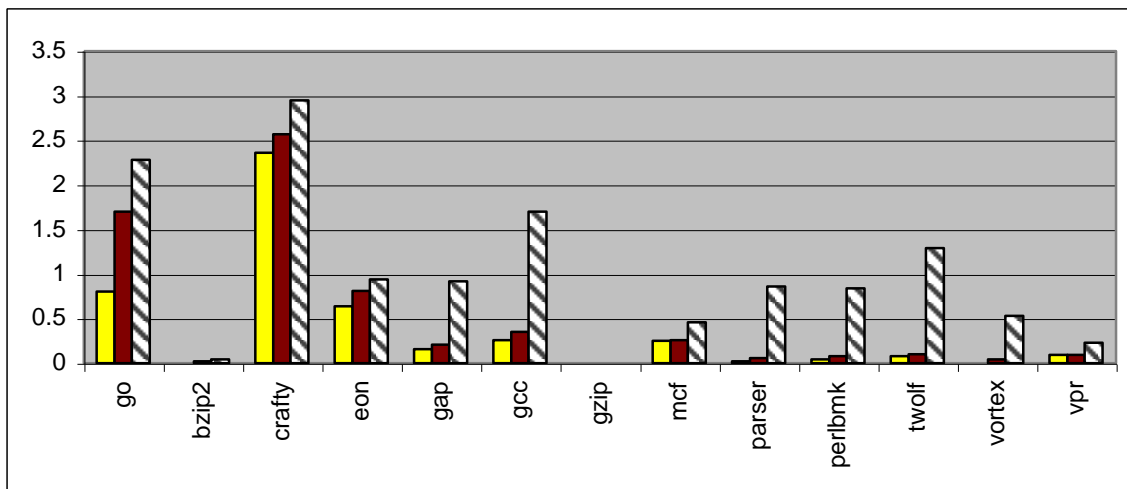


Fig. 4.10 Porcentaje de *miss rate* de predicción (32 ent. , 128 ent. , 1024 ent. , SPECINT)

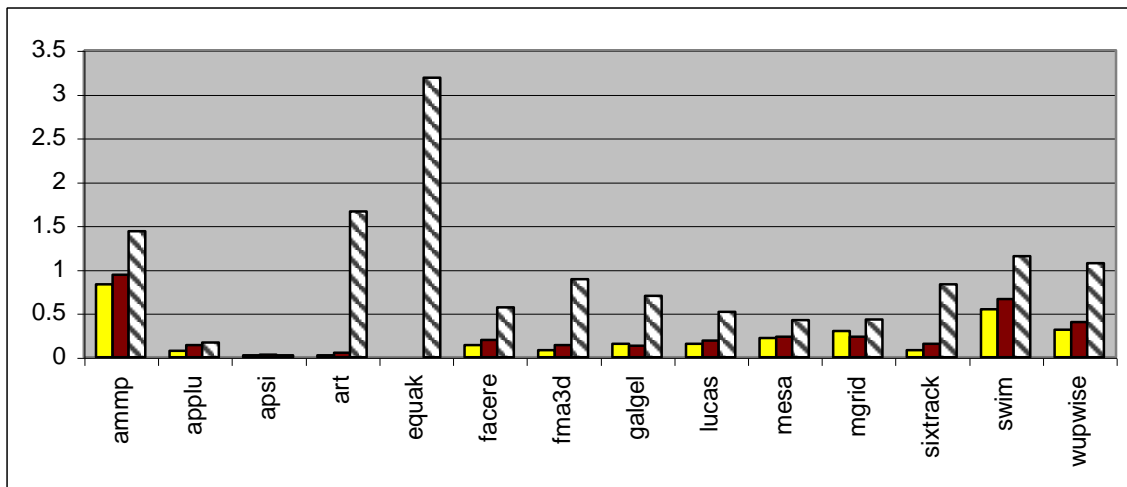


Fig. 4.11 Porcentaje de *miss rate* (32 ent. , 128 ent. , 1024 ent. , SPECFP)

Al manejar un autómata de predicción basado en un comportamiento de stride, y tener una tabla tan grande, como lo es de 1024, ya no se toma solo los accesos con un comportamiento muy repetitivo, sino que se consideran también otros accesos con comportamientos ya no tan repetitivos, lo que da pie a tener errores de predicción, por esta razón se decidió manejar una tabla de un tamaño de 128 entradas, con la que según los resultados obtenidos se puede cubrir la mayoría de los accesos con comportamiento repetitivo.

#### 4.5.2 Mecanismo de predicción en base a stride

El mecanismo básico en la predicción en base a *stride*, es guardar la dirección efectiva de los operandos de memoria. Calcula el *stride* para el acceso y activa el campo *state* para verificar si se puede predecir la siguiente referencia al comparar el *stride* previamente guardado con el que se acaba de calcular. El *stride* es obtenido al realizar la diferencia de la dirección efectiva de los dos accesos mas recientes, hechos por la instrucción.

La predicción se dice que es correcta cuando la dirección que se viene por el flujo normal, es igual a la suma del *stride* calculado más la *dirección\_previamente* almacenada, de no ser cierta esta condición no se predice y se modifica la condición del campo *state*. Al presentarse una condición verdadera se incrementa el contador del campo *state*, pero si es falsa se decrementa este contador. En base al valor del contador se utiliza el valor predicho para acceder al bloque correcto de la memoria caché en donde se encuentra el dato almacenado.

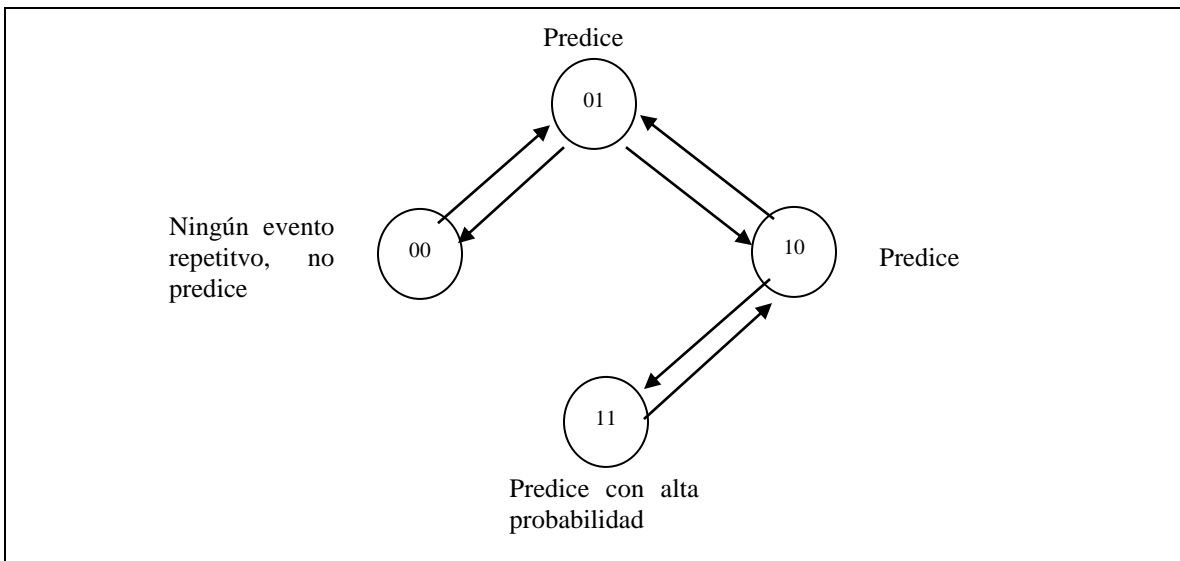


Fig. 4.12 Contador del campo *state*

En el contador que se maneja para el campo *State*, el primer estado corresponde a una condición de inicialización de la tabla de predicción activando una entrada de la tabla. El contador cambia al siguiente estado cada vez que el resultado del *stride* es similar al guardado en el primer estado; en este estado solo se actualizan los valores y no se toma en cuenta el valor de la tabla de predicción. El tercer estado se da al momento de repetirse



el *stride* y ahora si se toma en cuenta el valor del campo *bloque* para acceder al bloque predicho. El cuarto estado se considera un estado de muy alta probabilidad de acertar al bloque que contiene el dato, ya que el patrón de comportamiento se ha repetido por lo menos tres veces con anterioridad (tres estados anteriores).

Sin embargo, al realizar pruebas y comparaciones entre simulaciones, se encontró que al disminuir el valor del contador de este autómata se aumenta el porcentaje de predicción en cada una de las simulaciones. En lugar de que la predicción se realice en el estado tres del contador ahora se realiza en el segundo estado. La figura 4.12 muestra el funcionamiento del contador.

La razón de utilizar el valor del campo *bloque* en el segundo estado del contador, se debe a que muchos de los accesos realizados a una misma dirección de memoria se repiten de forma constante, si el autómata permitía predecir solo a partir de la tercera vez que se presenta el mismo patrón de comportamiento se dejaba ir un evento que muy posiblemente se puede predecir con un alto nivel de certeza. El número de veces que se presenta el mismo comportamiento, es decir, la misma dirección para una instrucción determinada, se presenta por lo general dos veces en su mayoría, y en un menor grado tres o más veces. Si la predicción se permite a partir del segundo estado del contador, se aprovecha esta característica del comportamiento del hardware y se logra un aprovechamiento del 40% en posibles predicciones correctas por encima del modelo anterior.

La tabla 1 presenta los datos en porcentaje del *hit rate* de predicción, a partir del segundo y tercer estado del contador.

SPEC	Pred. 2° edo. (%)	Pred 3° edo. (%)
go	45.55	18.22
bzip2	24.6	14.76
crafty	69.44	34.72
Eon	36.76	29.408
Gap	39.32	23.592
Gcc	35.51	10.653
gzip	49.24	44.316
mcf	57.46	34.476
parser	10.69	5.8795
perlbmk	5.69	1.707
twolf	2.05	0.82
vortex	3.89	1.167
vpr	37.12	14.848

**Capítulo 4** Sistema de Memoria

SPEC	Pred. 2° edo. (%)	Pred 3° edo. (%)
ampp	21.57	6.471
applu	30	15
apsi	73.59	51.513
Art	2.1	1.26
equak	2.29	1.832
facere	30.38	18.228
fma3d	22.11	13.266
galgel	18.54	10.197
lucas	19.88	11.928
mesa	59.67	38.7855
mgrid	48.65	14.595
sixtrack	15.31	9.186
swim	19.65	11.79
wupwise	30.95	18.57

**Tabla 4.1** Tabla de porcentaje de predicción correcta

A esta clase de predictor se le puede añadir otros elementos para ir elevando su nivel de complejidad y disminuir la tasa de error, pero esto trae como consecuencia que no sea práctico, ya que el tiempo que se consume para calcular el próximo valor es más alto del que se lleva el pipeline en ejecutar la instrucción en cuestión.

**4.5.3 Metodología de Evaluación**

Se comparó el comportamiento de diferentes modelos de memorias caché usando simulaciones de cargas de trabajo, para ello se empleó el Simplescalar [21], cuyas modificaciones e implementación de los modelos se detalla en el capítulo 4.

Se recopiló información de 20 programas, obtenidos de SPEC95 y SPEC2000. Los modelos de memoria simuladas fueron: memoria de mapeo directo (MP), memoria tipo *Column-Associative* provista de un predictor en base a stride (CA-Pred), una memoria asociativa nivel dos (AS-2) y nuestro modelo (PAAC); todos los modelos con tamaños de 8K, 16K, 32K y 64K.

*Medición del rendimiento*

Para determinar el rendimiento de la memoria se usó la fórmula presentada en la tabla 4.2, en la que se desglosan los componentes de la fórmula. Para determinar el rendimiento de nuestro modelo tenemos que descomponer el factor *K* y *L* en diferentes elementos, ya que se toma en cuenta diferentes tiempos de acceso según el tipo de error y acierto que se realiza.

$$Rendimiento = \frac{K + L}{M}$$

$K = N + O + P$   
 $L = Q + R$                       donde:

*N* = No. de aciertos en bloque uno por la latencia de acceso a bloque uno  
*O* = No. de aciertos en bloque uno por la latencia de acceso a bloque dos  
*P* = No. de aciertos predichos por la latencia de acceso a bloque uno  
*Q* = No. de errores por la latencia de error  
*R* = No. de errores de predicción por la latencia de acceso a bloque dos

**Tabla 4.2** Fórmula de rendimiento (latencia en ciclos).

La latencia de acierto para una memoria de mapeo directo es de 1 ciclo, para una memoria asociativa nivel dos es de 2 ciclos, para nuestro modelo es de un ciclo para el bloque uno y de dos ciclos para el bloque dos (acceso secuencial). La latencia de error será de 18 ciclos para los tres modelos. La figura 4.13, muestra la comparación de los tiempos de acceso para una memoria de acceso directo, una memoria asociativa de nivel dos y el modelo propuesto de una memoria asociativa secuencial con predictor.

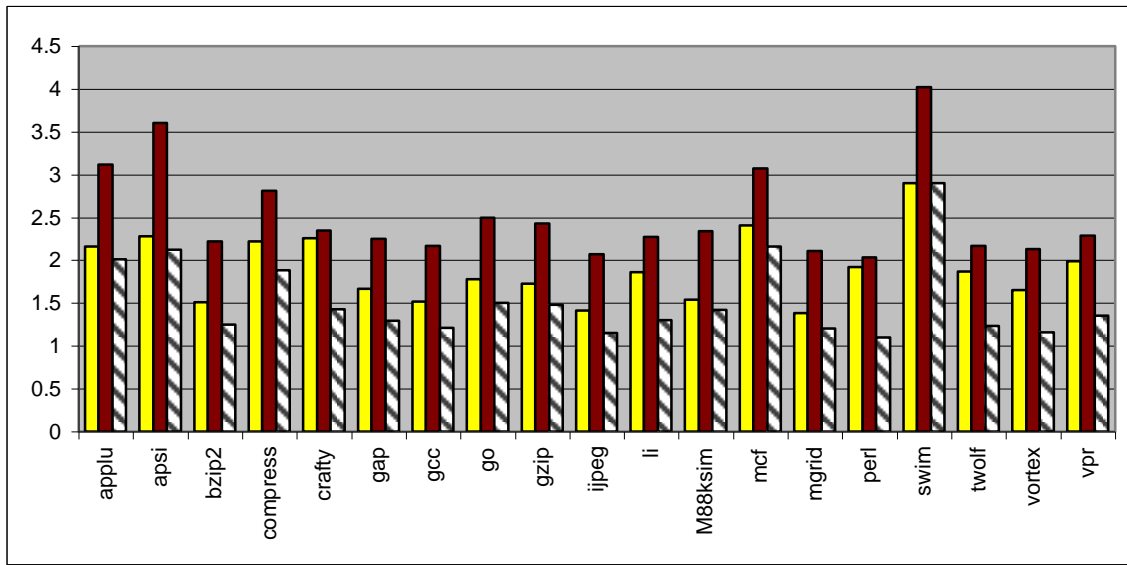


Fig. 4.13 Latencia de acceso (Mapeo directo ■ , Asociativa nivel dos ■ ,PAAC ▨ )

#### 4.5.4 Resultados

La tabla 4.3 muestra los números obtenidos para cada uno de los modelos. El índice de errores para nuestro modelo (PAAC) es similar al de una memoria de tipo asociativa, ya que parte del mismo principio de colocación de datos, pero la ventaja que presenta es el tiempo de acceso, la tabla 4 muestra los tiempos de acceso obtenidos para cada uno de los modelos.

Spec	MP	CA_Pred	AS2	PAAC
applu	5.74	5.77	5.39	5.39
apsi	11.88	9.6	7.11	7.11
bzip2	2.99	1.79	1.46	1.46
compress	7.197	6.7	5.1	5.1
crafty	7.497	6.05	2.29	2.29
gap	4.02	2.80	1.60	1.60
gcc	3.13	2.26	1.14	1.14
go	8.11	5.74	1.67	1.67
gzip	4.348	3.76	2.76	2.76
ijpeg	2.49	2.20	0.49	0.49
li	2.72	2.66	1.72	1.72
M88ksim	3.20	2.49	2.20	2.20

mcf	9.00	8.64	7.34	7.34
mgrid	2.96	3.97	2.36	2.36
perl	5.45	1.38	0.23	0.23
swim	31.13	38.91	13.73	13.73
twolf	5.15	3.76	1.11	1.11
vortex	3.07	1.93	0.87	0.87
vpr	5.9	4.09	1.84	1.84

**Tabla 4.3** Porcentajes de *miss rate* (%)

En la tabla 4.4 se resalta el menor índice de latencia para la PAAC, incluso más bajo que el de una memoria de mapeo directo del mismo tamaño, teniendo una reducción de más del 25 %. Al realizar comparaciones entre el número de ciclos de latencia entre la PAAC y una de mapeo directo, la PAAC de 16K tiene una latencia menor que una memoria de mapeo directo de 32K, dato que es sobresaliente teniendo en cuenta que conforme se incrementa el tamaño de nuestro dispositivo de almacenamiento, se reduce el índice de *miss rate*.

SPEC	MP	CA_Pred	AS2	PAAC
applu	2.16	2.34	3.12	2.01
apsi	2.28	2.62	3.6	2.12
bzip2	1.51	1.59	2.22	1.25
compress	2.22	2.42	2.81	1.88
crafty	2.26	2.25	2.35	1.43
gap	1.67	1.77	2.25	1.29
gcc	1.52	1.68	2.17	1.21
go	1.78	2.01	2.5	1.5
gzip	1.73	1.83	2.43	1.48
jpeg	1.41	1.66	2.07	1.15
li	1.86	1.69	2.27	1.3
M88ksim	1.54	1.72	2.34	1.42
mcf	2.41	2.87	3.07	2.16
mgrid	1.38	1.72	2.11	1.2
perl	1.92	1.7	2.03	1.1
swim	2.9	3.28	4.02	2.9
twolf	1.87	1.78	2.17	1.23
vortex	1.65	1.76	2.13	1.16
vpr	1.99	2.02	2.29	1.35

**Tabla 4.4** Latencia de acceso (ciclos)

En base a los resultados pronosticados y a los obtenidos se pueden realizar comparaciones del grado de efectividad del modelo propuesto. En el caso del área de predicciones de localización de datos se puede seguir implementando mejoras en cuanto al predictor, ya que a pesar de tener un porcentaje de 94% de efectividad se puede disminuir el *miss rate* de predicción con la finalidad de incrementar el rendimiento de nuestro modelo. Sin embargo al aumentar la complejidad del hardware implementado,

también se reduce en forma proporcional el rendimiento del procesador, ya que se hace uso de ciclos de reloj adicionales para dar pie a ese tipo de implementaciones.

# Capítulo 5

## Conclusión y Trabajos futuros

### 5.1 Conclusión

Al conjuntar los mejores modelos presentados en esta tesis, tanto para elevar el rendimiento de la memoria caché de instrucciones, como para la disminución de operaciones redundantes en la ventana de instrucciones, se logra el objetivo de esta tesis al presentar modelos capaces de detectar, predecir y ejecutar operaciones redundantes dentro de un procesador superescalar, atendiendo a los dos factores principales en el diseño de arquitecturas, a saber incremento de rendimiento y disminución de consumo de energía.

El impacto de los modelos de detección, predicción y ejecución de operaciones redundantes presentados en esta tesis tiene impacto en los siguientes campos:

Modelo propuesto	Área que ataca	Impacto
PAAC	Memoria Caché	Incremento de rendimiento
Ventana dividida en bloques	Ventana de instrucciones	Reducción de consumo de energía
Ventana con comparadores activados por grupo de bits (CID)	Ventana de instrucciones	Reducción de consumo de energía

**Tabla 5.1** Cuadro sinóptico modelo propuesto vs impacto en la arquitectura

### 5.2 Comentarios

En el modelo PAAC nos enfocamos a mejorar el rendimiento del sistema de memoria sin considerar el consumo de potencia que presenta esta mejora. Puesto que a finales de la última década la reducción del consumo de potencia en procesadores de propósito general viene a ser un compromiso de diseño [1] [2] y dado que los accesos a cache consumen entre el 30 y 60% del total de energía consumida por el procesador [3] [4] [5] [6]. Como parte de un trabajo futuro y usando como base el trabajo realizado en esta tesis, propondremos esquemas de gestión de memoria que mantengan el óptimo equilibrio *rendimiento-bajo consumo*.

En la otra sección de la tesis, se analizo el comportamiento de la ventana de instrucciones referente al número de operaciones redundantes que se llevan a cabo en esta sección de la arquitectura. Realizando la propuesta de 2 modelos (ventana dividida en bloques y comparación por identificación de bit) con cuatro vertientes respectivamente, eliminando en un alto porcentaje del número de operaciones redundantes en esta sección (comparaciones inútiles) y de esta forma disminuyendo gran parte del consumo de energía de este modulo, siguiendo los lineamientos de cálculo y técnicas de reducción de

energía expuestos en este trabajo. Un trabajo futuro con respecto a la ventana de instrucciones es la reducción de consumo de energía al proponer una ventana de instrucciones capaz de seguir la traza de un dato, impacto de la solución de una dependencia de dato dentro de la ventana, en otras operaciones también dentro de la ventana. Esto siendo posible al aplicar técnicas, tanto dinámicas como estáticas, de reutilización de instrucciones y sus valores generados [7].

Esta clase de resultados da pie a pensar en una combinación de ambos modelos, es decir, aplicar. Esta clase de implementaciones se deja para trabajos futuros, además, de que se toma en cuenta la complejidad de un modelo de tal proporción y su difícil manejo en términos de sencillez y rapidez, que a final de cuentas son parte de los factores que se toman en cuenta al momento de diseñar arquitecturas de microprocesadores.

Al conjuntar los mejores modelos presentados en esta tesis, tanto para elevar el rendimiento de la memoria cache de instrucciones, como para la disminución de operaciones redundantes en la ventana de instrucciones, se logra el objetivo de esta tesis al presentar modelos capaces de detectar, predecir y ejecutar operaciones redundantes dentro de un procesador superescalar atendiendo a los dos factores principales en el diseño de arquitecturas, a saber incremento de rendimiento y disminución de consumo de energía.

### 5.3 Trabajos futuros

A continuación se enlista una serie de trabajos futuros que se han vislumbrado gracias a la investigación de esta tesis.

- Unión de modelos de detección de receptores en la ventana de instrucciones

Al conjuntar los dos mejores modelos para la ventana de instrucciones, presentados en esta tesis, la división de la ventana de instrucciones en bloques, que a su vez es dividido en cuadrantes y que se aplique la activación de comparadores de cada cuadrante por medio del grupo de bits CID. De esta forma se puede manejar un número más acercado al número de comparaciones ideales realizadas en la ventana de instrucciones, con un bajo consumo de energía debido a la flexibilidad de los comparadores.

- Modificación de modelos de la ventana de instrucciones para incrementar el rendimiento

Una de las limitantes de los modelos que se enfocan a la reducción de consumo de energía es la disminución de rendimiento en el procesador al adicionar hardware que implica el consumo de ciclos de reloj del procesador. Es en este rubro donde se propone enfocar uno de los futuros trabajos de esta investigación, en cuanto al desarrollo de los modelos de reducción de consumo de energía en la ventana de instrucciones, en forma paralela al incremento de rendimiento de este módulo.

- Detección de trazas en la ventana de instrucciones en busca de receptores

Una de las líneas de investigación en cuanto a elevar el rendimiento de la ventana de instrucciones (mayor número de instrucciones lanzada a ejecutar) es por medio de la detección de trazas de dependencias de datos. Esto implica determinar los posibles receptores de un dato que es próximo a producirse en las unidades funcionales y el impacto de estos receptores en otras instrucciones ligadas a su resultado, llevándose este seguimiento o traza en forma ascendente en la ventana de instrucciones.

- Detección de ceros en la ventana de instrucciones (*bypassing*)

Al detectar instrucciones con operandos cuyo valor es cero, se puede evitar su cálculo o saberlo antes de ser calculado, siendo retiradas estas operaciones de la ventana de instrucciones, permitiendo elevar el IPC, además de determinar su impacto en instrucciones dependientes del resultado de estas operaciones.

- Modificación de modelo PAAC para reducción de consumo de energía

El modelo PAAC presentado en esta tesis ataca en forma exitosa la tasa de errores de la memoria caché de datos del procesador, sin embargo el consumo de energía se incrementa al contar con módulos adicionales a esta memoria. El propósito de continuar desarrollando este proyecto es la disminución del consumo de energía por parte de este módulo.



# Referencias

## Introducción

[1] J. R. Goodman, “Using Cache Memory to Reduce Processor Memory Traffic” *In Proc. of the 10th Int. Symp. On Computer Architecture*, 1983.

[2] D. Burger, J. R. Goodman, and A. Kägi, “Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors”. *In Proc. of the 23rd Int. Symp. On Computer Architecture*, 1996.

[3] T.-F. Chen and J.-L. Baer. “A performance Study of Software and Hardware Data Prefetching Schemes”. *In Proc. of the 21st Int. Symp. On Computer Architecture*, 1994.

[4] N.P. Jouppi. “Improving Direct-Mapped Cache Performance by the Addition of Small Fully Associative Cache and Prefetch Buffers”. *In Proc. of the 17th Int. Symp. On Microarchitecture*, 1990.

[5] J. González and A. González. “Speculative Execution via Address Prediction and Data Prefetching”. *In Proc. of the 11th ACM Int. Conf. on Supercomputing*, 1997.

[6] D.M. Tullsen, S.J. Eggers and H.M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”. *In Proc. of the 22nd Int. Symp. On Computer Architecture*, 1995.

[7] Avinash Sodani. “Dynamic Instruction Reuse” A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Sciences) at the University of Wisconsin—Madison 2000

## Capítulo 1

- [1] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp.473-530, Sept. 1982.
- [2] R. M. Keller, "Look-Ahead Processors," *ACM Computing Surveys*, vol. 7, pp. 66-72, December 1975.
- [3] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, pp. 37-58, January 1990.
- [4] W. W. Hwu and Y. N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Proc. 13th Annual International Symposium on Computer Architecture*, pp. 297-307, June 1986.
- [5] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Annual Workshop on Microprogramming*, pp. 103-108, December 1985.
- [6] Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture," *Proc. 18th Annual Workshop on Microprogramming*, pp. 109-116, December 1985.
- [7] M. Johnson, *Superscalar Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.
- [8] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.
- [9] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. on Computers*, vol. 39, pp. 349-359, March 1990.
- [10] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, pp. 37-58, January 1990.
- [11] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM Journal of Research and Development*, vol. 34, pp. 23-36, January 1990.
- [12] J. E. Smith, et al, "The ZS-1 Central Processor," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 199-204, October 1987.
- [13] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, pp. 25-33, January 1967.

- [14] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Annual Workshop on Microprogramming*, pp. 103-108, December 1985.
- [15] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, pp. 25-33, January 1967.
- [16] B. Case, "Intel Reveals Pentium Implementation Details," *Microprocessor Report*, pp. 9-13, Mar. 29, 1993. , P. Y. T. Hsu, "Design of the R8000 Microprocessor," *IEEE Micro*, pp. 23-33, April 1994.
- [17] G. S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 53-62, April, 1991.
- [18] S. Weiss and J. E. Smith, *Power and PowerPC: Principles, Architecture, Implementation*. San Francisco, CA: Morgan Kaufmann, 1994.
- [19] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," *IEEE Transactions on Computers*, vol. C-36, pp. 1496-1514, December 1987.
- [20] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.
- [21] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986., Wen-Mei Hwu. The Microarchitecture of Superscalar Processors.
- [22] D. Burger and T. Austin; "The Simplescalar Tool Set, Version 3.0"; University of Wisconsin Computer Sciences Department
- [23] J-k. Peir, Y. Lee, and W. Hsu, "Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology," *Proc. 8th ASPLOS*, 1998.
- [24] K. Roland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE micro*, 14(4): 59-67, 1994.
- [25] BAKER, M. G., HARTMAN, J.H., KUPFER, M.D., SHIRRAF, K.W., AND OUSTERHOUT, J.K. Measurements of a distributed file system. *In Proceedings of Thirteenth Symposium on Operating Systems Principles* (August 1991), Association for Computing Machinery, pp. 198-212.

## Capítulo 2

- [1] Bruschi, S. M. et. al. 1999. "Simulation as a Tool for Computer Architecture Teaching," *SCS Summer Simulation Conference*, pp. 81-86.
- [2] Cassel, L., Kumar D. et. al. (to appear) "Distributed Expertise for Teaching Computer Organization & Architecture," ITiCSE 2000 Working Group Report, *ACM SIGCSE Bulletin*.
- [3] Clements, A. May/June 2000. "The Undergraduate Curriculum in Computer Architecture," *IEEE Micro Special Issue on Computer Architecture Education*, Vol. 20 No. 3, pp. 13-22.
- [4] Coe, P.S., L.M. Williams, and R.N. Ibbett. June 1996. "An Interactive Environment for the Teaching of Computer Architecture," *ACM ITiCSE*, pp. 33-35.
- [5] Coey, W. A. June 1993. "An Interactive Tutorial System for MC68000 Assembly Language Using Hypercard," *ACM SIGCSE Bulletin*, Vol. 25, No. 2. pp. 19-23.
- [6] Djordjevic, J., Milenkovic A., and N. Grbanovic May/June 2000. "An Integrated Environment for Teaching Computer Architecture," *IEEE Micro Special Issue on Computer Architecture Education*, Vol. 20 No. 3, pp. 66-74.
- [7] Foley, D. Sept. 1992. "Microcode Simulation in the Computer Architecture Course," *ACM SIGCSE Bulletin*, Vol. 24, No. 3. pp. 57-59.
- [8] Frank, P. and J. Leathrum Jr. 2000. "A Java Simulation for a Course in Computer Architecture," *Intl. Conf. On Simulation and Engineering Education (ICSEE)*, pp. 65- 70.
- [9] Fuente, S.R. et. al. June 1999. "Teaching Computer Architecture with a New Superscalar Processor Emulator," *ACM ITiCSE*. pp. 99-102.
- [10] Goodman J. and K. Miller. 1993, *A Programmer's View of Computer Architecture*, Oxford University Press.
- [11] Hennessy J. and D. Patterson. 1996. *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann.
- [12] Knox, D.L. June 1997. "Integrating Design and Simulation into a Computer Architecture Course," *ACM ITiCSE*, pp. 42-44.
- [13] Magagnosc, D. March 1994. "Simulation in Computer Organization: A Goals Based Study," *ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 178-182.

[14] Patt, Y. and S. Patel. 2001. *Introduction to Computing Systems*, McGraw-Hill.  
Patterson D. and J. Hennessy. 1998. *Computer Organization and Design, Second Edition*,

[15] Morgan Kaufmann. Skrien, D. and J. Hosack. March 1991. "A Multilevel Simulator At The Register Transfer Level For Use in An Introductory Machine Organization Class," *ACM SIGCSE Technical Symposium on Computer Science Education*, pp.347-351.

[16] Stallings, W. 2000. *Computer Organization and Architecture, Fifth Edition*, Prentice Hall. Tanenbaum, A. 1999. *Structured Computer Organization, Fourth Edition*, Prentice Hall.

### Capítulo 3

- [1] Paul J.M. Havinga, Gerard J.M. Smit, Design techniques for low power systems. *Journal of Systems Architecture*. Vol. 46: 1, 2000.
- [2] D. Folegnani and A. Gonzalez. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, May 2001.
- [3] S. Weiss and J. Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE Transactions on Computers*, 33(11):1013–1022, Nov. 1984.
- [4] Michael Huang, Jose Renau, Joseph Torrellas. *Energy-Efficient Hybrid Wakeup Logic*. International Symposium on Low-Power Electronics and Design, pages 196–201, August, 2002, Monterey, California.
- [5] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. *International Symposium on Microarchitecture*, pages 225–236, Dec. 2001.
- [6] S. Onder and R. Gupta. Superscalar Execution With Dynamic Data Forwarding. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, Oct. 1998.
- [7] R. Canal and A. Gonzalez. A Low-Complexity Issue Logic. In *International Conference on Supercomputing*, pages 327–335, June 2000.
- [8] S. Weiss and J. Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE Transactions on Computers*, 33(11):1013–1022, Nov. 1984

## Capítulo 4

[1] J-k. Peir, Y. Lee, and W. Hsu, “Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology,” Proc. 8th ASPLOS, 1998.

[2] K. Roland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE micro*, 14(4): 59-67, 1994.

[3] BAKER, M. G., HARTMAN, J.H., KUPFER, M.D., SHIRRIF, K.W., AND OUSTERHOUT, J.K. Measurements of a distributed file system. In *Proceedings of Thirteenth Symposium on Operating Systems Principles* (August 1991), Association for Computing Machinery, pp. 198-212.

[4] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating systems and multiprogramming, *ACM Transactions on Computer Systems*, 6:393-431, November 1988.

[5] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming. *ACM Transactions on Computer Systems*, 6(4): 393-431, November 1988.

[6] [Anant Agarwal and Steven D. Pudar. Column-Associative caches: A technique for reducing the miss\_rate of direct mapped caches. In *20th Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pag. 179-190, IEEE, 1993.

[7] B. Calder, and D. Grunwald, J. Elmer, “Predictive Sequential Associative Cache”, *Proc. 2<sup>nd</sup> HPCA*, 1996.

[8] S.P. Harbison. An architecture alternative to optimizing compilers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 57–65, 1982.

[9] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[10] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *International Symposium on Computer Architecture*, pages 194–205, 1997.

[11] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *International Conference on Parallel Processing*, pages 61–8, 2000.

[12] John Lu and Keith Cooper. Register promotion in c programs. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 308–319, 1997.

- [13] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.
- [14] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *Conference Record of ACM Symposium on Principles of Programming Languages (POPL)*, pages 237–251, 1998.
- [15] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 64–76, 1999.
- [16] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *International Symposium on Computer Architecture*, pages 194–205, 1997.
- [17] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *International Conference on Parallel Processing*, pages 61–68, 2000.
- [18] J. W. Fu, J. H. Patel and B. L. Janssens, “Stride Directed Prefetching in Scalar Processors,” Proc. of the 25th Int. Symp. On Microarchitecture (MICRO-25), pp. 102-110, 1992 .
- [19] Y. Jeguo and O. Temam, “Speculative Prefetching,” Proc. Of the Conf. on Supercomputing, pp. 57-60, 1993.
- [20] J. Gonzalez and A. Gonzalez, “Memory Address Prediction for Data Speculation,” Technical Report # UPC-DAC-1996-50, Universidad Politecnica de Catalunya, 1996.
- [21] M. Hill, A. Smith, “Evaluating Associativity in CPU Caches”, IEEE Trans Computers, Vol. 22(12), Dec. 1989.



## Capítulo 5

- [1] R. González and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid State Circuits*, 31(9):1277-1284, September 1996
- [2] Daniele Folegnani and Antonio González. "Energy-Effective Issue Logic," *Proceedings of the 28th Int. Symposium on Computer Architecture*, Goteborg (Sweden), June 30 - July 4, 2001
- [3] B. Amrutur and M. Horowitz. Techniques to reduce power in fast wide memories. In *Symposium on Low Power Electronics*, vol. 1, pp 92-93, October 1994
- [4] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H.S Kim, and W. Ye. Energy-driven integrated hardware-software optimization using SimplePower. In *ISCA-27*. Vancouver, Canada, June 2000
- [5] G. Albera, I. Bahar, S. Manne "Power and Performance trade-offs using various caching strategies," *Proceedings of the Int. Symposium on low-Power Electronics and Design*, 1998
- [6] Luis Villa, Michael Zhang, and Krste Asanovic, "Dynamic Zero Compression for Cache Energy Reduction," *33rd International Symposium on Microarchitecture*, Monterey, CA, December 2000
- [7] A Sodani, G. Sohi, "Dynamic Reuse of Instruction", *Proceedings of the 24th Int. Symposium on Computer Architecture*, Hawai (USA), July 30 - Ago 3, 1997

# Apéndice A

## Símbolos

$C(C, S, A, B)$	Una organización específica de caché
$C$	Tamaño de caché en <i>words</i> <sup>1</sup>
$S$	Número de conjuntos
$A$	Grado de asociatividad
$B$	Tamaño de bloques, en palabras <i>words</i>
$F$	Tamaño de búsqueda, en <i>words</i> .
$m$	(Local) porcentaje de error.
$M$	Porcentaje de error global
$L_1, L_2, L_3, \dots, L_i, \dots, L_n$	El primero, segundo, hasta el nivel $n$ de una jerarquía de memoria. La memoria principal es el nivel $L_{n+1}$ .
$T_{Total}$	El tiempo total en segundos de un programa o traza.
$m(C) = f(S, C, A, B)$	La relación funcional entre el porcentaje de error y los parámetros organizacionales de la memoria.
$N_{Total} = g(m(C), B, LA, TR, t_{CPU})$	La relación funcional entre al cuenta de ciclos de CPU, el porcentaje de error de la memoria, las características de la memoria y el tiempo de ciclo.
$t_{CPU} = h(C, S, A, B)$	La relación funcional entre el tiempo de ciclo de CPU y los parámetros de la memoria caché.

<sup>1</sup> Un *word* esta definida por 4 *bytes*