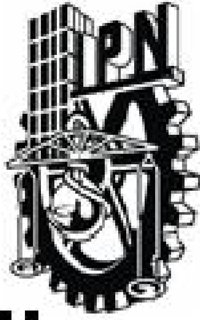


Instituto Politécnico Nacional

Centro de Investigación en Computación

Laboratorio de Microtecnología y Sistemas Embebidos

Diseño de un ROB-Distribuido para Procesadores Superescalares

TESIS

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

PRESENTA

Lic. José Raúl García Ordaz

DIRECTORES DE TESIS: Dr. Marco Antonio Ramírez Salinas
Dr. Herón Molina Lozano

MÉXICO, D.F. Enero 2011

ÍNDICE DE CONTENIDO

Resumen…………………………………………………………………………………………… I

Abstract…………………………………………………………………………………………… II

Agradecimientos…………………………………………………………………………………… III

Dedicatoria………………………………………………………………………………………… IV

Índice de contenido……………………………………………………………………………… V

Índice de Figuras………………………………………………………………………………… VII

Índice de Tablas…………………………………………………………………………………… IX

Capítulo 1 Introducción…………………………………………………………………………… 1

1.1. Antecedentes…………………………………………………………………………… 2
1.2. Planteamiento del Problema…………………………………………………………… 4
1.3. Justificación……………………………………………………………………………… 4
1.4. Objetivos………………………………………………………………………………… 5
1.5. Organización del Trabajo……………………………………………………………… 6

Capítulo 2 Estado del Arte………………………………………………………………………… 7

2.1. El procesador MIPS R10000…………………………………………………………… 7
 2.1.1. La Lista de Instrucciones Activas del MIPS R10000…………………… 9
2.2. El procesador Alpha 21264…………………………………………………………… 10
 2.2.1. La Ventana de Instrucciones del Alpha 21264…………………………… 12
2.3. Los procesadores Pentium 3 y 4……………………………………………………… 13
 2.3.1. El Búfer de Reordenamiento de Instrucciones en NetBurst……… 16
2.4. Propuestas recientes…………………………………………………………………… 17
2.5. Resumen del Capítulo…………………………………………………………………… 24

Capítulo 3 Marco Teórico………………………………………………………………………… 25

3.1. Los Procesadores Superescalares.…………………………………………………… 25
 3.1.1. Paralelismo a nivel de instrucción y ejecución fuera de orden…... 26
 3.1.2. La Ventana de Instrucciones y el ROB…………………………………… 28
3.2. Herramientas de Software para la simulación de Arquitecturas…………… 31
 3.2.1. El conjunto de herramientas de simulación SimpleScalar…………. 32

3.2.2. El simulador sim-outorder…… 34
3.3. Resumen de Capítulo…… 36

Capítulo 4 Diseño del ROB…… 37
4.1. Metodología de diseño…… 37
4.2. Criterios de diseño…… 39
4.2.1. Análisis de las instrucciones de control de flujo…… 39
4.2.2. Esquemas de recuperación del estado del procesador…… 41
4.3. Arquitectura del diseño propuesto…… 43
4.3.1. Ejecución especulativa de instrucciones…… 45
4.3.2. Reciclado de registros físicos…… 47
4.3.3. Retiro grupal de instrucciones…… 47
4.4. Modelado y Simulación del diseño…… 48
4.4.1. Modificación del código de SimpleScalar…… 49
4.4.2. Modelo de la arquitectura propuesta…… 51
4.4.3. Optimización del proceso de recuperación…… 56
4.5. Proceso de simulación de la arquitectura propuesta…… 56
4.6. Resumen del Capítulo…… 59

Capítulo 5 Pruebas y Resultados…… 60
5.1. Descripción de las estadísticas estudiadas…… 60
5.2. Análisis y evaluación del IPC…… 61
5.3. Reducción de la ejecución de instrucciones en traza equivocada…… 67
5.4. Incremento en la velocidad de ejecución de instrucciones…… 69
5.5. Ocupación del ROB…… 72
5.6. Resumen del Capítulo…… 74

Capítulo 6 Conclusiones y Trabajo Futuro…… 75
6.1. Conclusiones…… 75
6.2. Trabajo Futuro…… 76

Referencias Bibliográficas…… 78

Glosario…… 81

Anexo A…… 83
Anexo B…… 86
Anexo C…… 89
Anexo D…… 92

ÍNDICE DE FIGURAS

Figura 2-1 Pipeline del MIPS R10000........ 8
Figura 2-2 Diagrama de bloques del MIPS R10000........ 9
Figura 2-3 Diagrama de bloques del Alpha21264........ 11
Figura 2-4 Pipeline del Alpha 21264........ 12
Figura 2-5 Pipeline del P6........ 14
Figura 2-6 Diagrama de bloques del NetBurst........ 15
Figura 2-7 ROB en las microarquitecturas NetBurst y P6........ 17
Figura 2-8 El ROB como una gran estructura FIFO multipuerto........ 18
Figura 2-9 Arquitectura superescalar que emplea al ROB........ 18
Figura 2-10 Propuesta para un ROB separado en componentes........ 19
Figura 2-11 Propuesta para un ROB comprimido........ 20
Figura 2-12 Comparación del ROB convencional con el Cherry ROB........ 22
Figura 2-13 Microarquitectura basada en WIB........ 23
Figura 2-14 Microarquitectura que usa el VB como sustituto del ROB........ 24
Figura 3-1 Pipeline de una arquitectura escalar........ 25
Figura 3-2 Pipeline de una arquitectura superescalar........ 26
Figura 3-3 Diagrama de bloques de un procesador superescalar........ 26
Figura 3-4 Diagrama que ilustra la dependencia entre instrucciones........ 27
Figura 3-5 Diagrama que ilustra la ejecución fuera de orden........ 28
Figura 3-6 Diagrama conceptual de un ROB........ 29
Figura 3-7 Campos de datos de una entrada del ROB........ 30
Figura 3-8 Arquitectura del simulador SimpleScalar........ 33
Figura 3-9 Diagrama de bloques del pipeline de sim-outorder........ 34
Figura 3-10 Ciclo principal del simulador sim-outorder........ 35
Figura 3-11 Diagrama de bloques del RUU implementado por sim-outorder........ 36
Figura 4-1 Diagrama de flujo para el diseño del ROB........ 38
Figura 4-2 Frecuencia de aparición de instrucciones de salto condicional........ 41
Figura 4-3 Esquema típico del ROB para recuperación........ 42
Figura 4-4 Esquema optimizado del ROB para recuperación........ 43
Figura 4-5 Diagrama de bloques de la arquitectura del diseño propuesto........ 44
Figura 4-6 Mecanismo para habilitar la ejecución especulativa........ 46
Figura 4-7 Reciclado de registros y actualización de valores en el RF........ 47
Figura 4-8 Retiro grupal de instrucciones........ 48
Figura 4-9 Sim-outorder con un ROB monolítico........ 49
Figura 4-10 Miembros de la estructura RUU_station........ 50
Figura 4-11 Diagrama que ilustra la construcción de la estructura RUU........ 51
Figura 4-12 Sim-outorder con un ROB distribuido........ 52

Figura 4-13	Código para añadir la opción –bruu:size en sim-outorder.c…………	52
Figura 4-14	Contenido de una entrada del RUU distribuido……………………………	53
Figura 4-15	Construcción de FIFOs RUU y B-RUU……………………………………………	54
Figura 4-16	Condiciones para el funcionamiento de las FIFOs RUU y B-RUU……	55
Figura 4-17	Configuraciones a simular para la arquitectura ROB Monolítico…..	58
Figura 4-18	Configuraciones a simular para la arquitectura ROB Distribuido…..	58
Figura 5-1	IPC, configuración M-ROB de 32 entradas contra D-ROB…………….….	62
Figura 5-2	IPC, configuración M-ROB de 64 entradas contra D-ROB…………….….	63
Figura 5-3	IPC, configuración M-ROB de 128 entradas contra D-ROB……………	63
Figura 5-4	IPC, configuración D-ROB de 32 entradas contra OD-ROB…………...	65
Figura 5-5	IPC, configuración D-ROB de 64 entradas contra OD-ROB…………...	65
Figura 5-6	IPC, configuración D-ROB de 128 entradas contra OD-ROB………….	66
Figura 5-7	FI, configuración D-ROB de 32 entradas contra OD-ROB.……………	67
Figura 5-8	FI, configuración D-ROB de 64 entradas contra OD-ROB.……………	68
Figura 5-9	FI, configuración D-ROB de 128 entradas contra OD-ROB.…………	69
Figura 5-10	Speedup, configuración D-ROB de 32 entradas contra OD-ROB……	70
Figura 5-11	Speedup, configuración D-ROB de 64 entradas contra OD-ROB……	71
Figura 5-12	Speedup, configuración D-ROB de 128 entradas contra OD-ROB….	71
Figura 5-13	ROB lleno, configuración D-ROB de 32 entradas contra OD-ROB….	73
Figura 5-14	ROB lleno, configuración D-ROB de 64 entradas contra OD-ROB….	73
Figura 5-15	ROB lleno, configuración D-ROB de 128 entradas contra OD-ROB..	74
Figura A-1	Listado de opciones para configurar una arquitectura a simular…..	85
Figura B-1	Dando de alta estadísticas para la subestructura BRUU………………	86
Figura B-2	Declarando RUU y BRUU………………………………………………………..	87
Figura B-3	Inicializando RUU y BRUU……………………………………………………..	88
Figura C-1	Listado de estadísticas obtenidas de la simulación………………………	91
Figura D-1	Script para extraer datos de interés del archivo…………………………	92
Figura D-2	Script para ejecutar todos los scripts……………………………………………	93

ÍNDICE DE TABLAS

Tabla 3-1 Características de los simuladores que conforman a SimpleScalar…. 33
Tabla 4-1 Tipos de instrucciones más frecuentemente ejecutadas………………… 40
Tabla 4-2 Configuración de la arquitectura base simulada……………………………. 57
Tabla 4-3 Benchmarks utilizados y sus características……………………………………. 57
Tabla 5-1 Valor promedio del IPC, configuración M-ROB contra D-ROB………… 64
Tabla 5-2 Valor promedio del IPC, configuración D-ROB contra OD-ROB………. 66
Tabla 5-3 Valor promedio de FI, configuración D-ROB contra OD-ROB…………. 69
Tabla 5-4 Valor promedio del speedup, configuración D-ROB contra OD-ROB. 72
Tabla 5-5 Valor promedio del ROB lleno, configuración D-ROB contra OD-ROB…. 74

RESUMEN

Para lograr un alto desempeño, los procesadores superescalares modernos utilizan diversas técnicas implementadas en hardware, tales como el renombramiento de registros, la predicción de saltos condicionales, y la ejecución especulativa de instrucciones. Estas técnicas tienen el objetivo de exponer la mayor cantidad posible de paralelismo a nivel de instrucción encontrado en un programa, mantener ocupadas al máximo las unidades funcionales del procesador, e incluso permitir que la ejecución de las instrucciones se realice fuera del orden original del programa, pero conservando la consistencia de los resultados, mediante el retiro de cada instrucción en estricto orden de programa. Estas técnicas funcionan en combinación con una importante estructura funcional, conocida comúnmente como Búfer de Reordenamiento de Instrucciones.

Al conjunto de instrucciones que se encuentran al vuelo en el procesador, se le da el nombre de Ventana de Instrucciones. El estado en el que se encuentra cada una de estas instrucciones, a medida que atraviesan las diferentes etapas de ejecución, es mantenido por el Búfer de Reordenamiento de Instrucciones, (ROB, por sus siglas en inglés). Con la información que almacena, esta estructura funcional es capaz de soportar la ejecución especulativa de instrucciones, realizar el reciclado de registros físicos, recuperar el estado exacto del procesador (en caso de eventos como errores de predicción de salto ó excepciones), y llevar a cabo el retiro de instrucciones.

El Búfer de Reordenamiento de Instrucciones se implementa típicamente como una estructura FIFO circular monolítica, con apuntadores de cabeza y cola, el primero apuntando a la próxima instrucción a ser retirada y el segundo apuntando a la próxima entrada libre. El número de entradas que tenga el ROB determina el tamaño de la Ventana de Instrucciones. El diseño monolítico típico del ROB presenta varios inconvenientes como la cantidad de recursos que se necesitan para su implementación, el tamaño del área que ocupa, la concentración de calor debido a la disipación de energía que genera y las fallas en el mecanismo de restauración del estado del procesador.

En el presente trabajo de tesis se presenta un nuevo diseño de la arquitectura de un Búfer de Reordenamiento de Instrucciones. Este diseño concibe al ROB como un conjunto de subestructuras distribuidas e introduce una pequeña subestructura llamada B-ROB, la cual almacena la información para soportar la ejecución especulativa, la recuperación de saltos y el retiro de instrucciones. De esta manera, se obtiene una arquitectura optimizada y distribuida que permite incrementar el desempeño del procesador en promedio, en un 8%, usando una configuración distribuida donde el tamaño de la subestructura B-ROB puede reducirse hasta en 1/4 del tamaño de la subestructura equivalente en un diseño monolítico tradicional.

ABSTRACT

In order to achieve high performance, modern superscalar processors use various techniques implemented in hardware, such as register renaming, branch prediction, and speculative execution. The target of these techniques is to expose as much instruction-level parallelism as possible, keeping functional units busy to the maximum, and even allowing out-of-order execution of instructions. These techniques work in combination with an important functional structure commonly known as ReOrder Buffer.

The set of in-flight instructions in the processor is named Instruction Window. The state of each one of these instructions, as they pass through the different execution stages of the pipeline, is maintained by the ReOrder Buffer (ROB). With the information it stores, this functional structure can support speculative execution, physical register recycling, recovery of the exact state of the processor (in the event of a branch misprediction or exception), and instruction commit.

The ReOrder Buffer is typically implemented as a monolithic circular FIFO structure, with head and tail pointers, the first pointing to the next instruction to be retired, and the second pointing to the next free entry. The number of entries in the ROB determines the size of the Instruction Window. The typical monolithic design of the ROB has several drawbacks such as the amount of resources that are needed to implement it, the size of area that it occupies, the hotspot it generates, and the flaws in the processor state recovery mechanism.

This thesis presents a new design of the ReOrder Buffer architecture. In this design the ROB is conceived as a set of distributed substructures and introduces a small structure called B-ROB, which stores information to support speculative execution, misprediction recovery and instruction commit. This way, an optimized distributed architecture is obtained. This new design increases the processor performance by 8% in average, using a distributed configuration where the size of the B-ROB substructure can be reduced to 1/4 of the equivalent substructure in a traditional monolithic design.

CAPÍTULO 1

INTRODUCCIÓN

La incesante búsqueda de métodos para mejorar el desempeño de los procesadores, elementos básicos para el funcionamiento de toda clase de dispositivos modernos, que van desde computadoras personales hasta teléfonos móviles, ha llevado al desarrollo de una clase de microarquitectura conocida como superescalar, la cual tiene la capacidad de ejecutar más de una instrucción en un mismo ciclo de reloj.

Una mejora realizada a la arquitectura superescalar, fue permitirle ejecutar las instrucciones de un programa en un orden diferente al original, con el objetivo de aprovechar el paralelismo a nivel de instrucción, pero retirando cada instrucción en estricto orden de programa para conservar la coherencia de los resultados. A esta técnica se le conoce como ejecución fuera de orden y su implementación en las arquitecturas superescalares actuales está ampliamente difundida dado el alto desempeño que le permite alcanzar al procesador.

Uno de los parámetros más importantes que se toman en cuenta al diseñar una nueva microarquitectura superescalar, es el tamaño de la Ventana de Instrucciones, la cual está constituida por el conjunto de instrucciones que se encuentran activas en el procesador. Ventanas de Instrucciones de mayor tamaño, exponen más instrucciones lo que hace posible extraer mayor paralelismo a nivel de instrucciones, incrementando así el desempeño del procesador. El tamaño de la Ventana de Instrucciones está determinado por una estructura funcional que mantiene el estado de cada una de las instrucciones activas, conocida típicamente como Búfer de Reordenamiento de Instrucciones o Lista de Instrucciones Activas, dependiendo de la microarquitectura que lo implementa.

Las tareas de las que está encargada esta estructura funcional son permitir la ejecución especulativa de instrucciones, el retiro de instrucciones completadas y del reciclado de registros físicos. El diseño de la arquitectura de un Búfer de Reordenamiento de Instrucciones no es un proceso trivial, pues para impactar positivamente en el desempeño del procesador, no basta con simplemente incrementar su tamaño, sino que implica realizar un estudio de las características de las arquitecturas actuales para detectar fallos y encontrar posibles mejoras que puedan implementarse para obtener un diseño eficiente.

El presente trabajo de tesis presenta el diseño de la arquitectura de un Búfer de Reordenamiento de Instrucciones para un procesador superescalar. Para ello se

seguirá proceso un que incluirá un análisis de las características de las arquitecturas actuales y de las propuestas más recientes para su implementación encontradas en la literatura, con el objetivo de encontrar posibles puntos de mejora. Enseguida se presenta una propuesta propia, que es evaluada mediante la generación de un modelo en software de dicha arquitectura, la cual será simulada, usando programas de prueba especializados para tal fin. Una vez terminado el proceso de simulación, se procederá a realizar un análisis de los diferentes datos obtenidos, para realizar una evaluación del desempeño de este diseño.

1.1. Antecedentes

La segmentación o *pipelining* es una técnica dentro del área de arquitectura de computadoras en la cual, para incrementar el desempeño de un procesador, una instrucción se realiza a través de cierto número de etapas en una secuencia. Cuando la arquitectura de un procesador implementa esta técnica, se le llama procesador segmentado o escalar. El procesador IBM 7030, desarrollado a principios de la década de 1960, se considera el primer procesador segmentado, el cual contaba con cuatro etapas en su *pipeline* para la ejecución de las instrucciones. Posteriormente surgió en 1964 el procesador CDC 6600, el cual contaba con diez unidades funcionales. Este procesador era capaz de decodificar y emitir una instrucción por ciclo, pero hasta tres instrucciones podían iniciar su ejecución a la vez.

También en ese año surgió el procesador IBM S/360 Modelo 91, el cual decodificaba una instrucción por ciclo y además permitía la ejecución de instrucciones fuera de orden de programa en su unidad funcional de punto flotante. Este procesador usaba la técnica conocida como algoritmo de Tomasulo, llamado así en honor de su inventor (Robert Tomasulo) [9].

La idea de un procesador que pudiera ejecutar más de una instrucción por ciclo de reloj, surgió en IBM en la década de 1960, aunque el uso del término superescalar (superscalar) se empleó, al parecer por primera vez, en un reporte técnico interno de esta compañía escrito por John Cocke y Tilak Agerwala, llamado *High Performance Reduced Instruction Set Processors*, en 1987, a partir de entonces dicho término se popularizó. La puesta en práctica de la idea de un procesador superescalar se llevó a cabo hasta 1965, con el desarrollo del procesador IBM ACS-1. En este diseño, hasta 16 instrucciones serían decodificadas cada ciclo y hasta siete instrucciones serían emitidas hacia las unidades funcionales, sin embargo este proyecto fue cancelado en 1969.

A partir de entonces, la investigación y desarrollo de este tipo de arquitecturas continuó de un modo lento hasta principios de la década de 1980 cuando surgieron los primeros procesadores RISC, los cuales fueron diseñados desde un principio con

una estructura segmentada. Tal es el caso de los proyectos *Cheetah* y *America* desarrollados por IBM para estudiar la ejecución superescalar. El procesador *Cheetah,* cuyo desarrollo inició en 1982, era capaz de ejecutar cuatro instrucciones por ciclo y fue la base para el desarrollo del procesador *America,* de 1985, el cual a su vez generó el desarrollo del procesador RS/6000 en 1990, el cual fue renombrado posteriormente como POWER1. Además de IBM, otras compañías tenían proyectos similares para generar procesadores RISC superescalares, como es el caso de DEC con su proyecto *Multititan* de 1985 que a su vez dio origen al desarrollo, iniciado en 1988, de la línea de procesadores Alpha (α).

En cuanto a los proyectos desarrollados en el área académica, se puede destacar la arquitectura prototipo de la universidad norteamericana de Stanford, creada por el grupo de trabajo dirigido por John Hennessy llamada MIPS, en 1981, quien junto a David Patterson fundó en 1984 la compañía MIPS Computer Systems. Dicha compañía sacó al mercado en 1985 el procesador R2000 que fue el primero de una línea de procesadores RISC superescalares que está vigente hasta la actualidad.

Los procesadores RISC superescalares surgieron mediante dos diferentes aproximaciones. Por una parte están los que fueron modificados de una arquitectura escalar existente hacia una arquitectura superescalar. Como es el caso de las líneas 960 de Intel, HP-PA de Hewlett-Packard, Sun Sparc de Sun Microsystems, MIPS R de MIPS Inc. y Am29000 de AMD. Otra aproximación fue la creación de una nueva arquitectura implementada desde un principio como superescalar, ese es el caso del Power1 de IBM, de la línea α de DEC y de la PowerPC desarrollada por Apple, IBM y Motorola.

En cuanto a los procesadores CISC superescalares se puede mencionar que aparecieron en el mercado de manera muy posterior a los RISC, debido a que son más difíciles de implementar que sus contrapartes RISC. El procesador Pentium de Intel y el MC 68060 de Motorola son ejemplos de los primeros procesadores CISC superescalares, los cuales salieron al mercado en 1993. Ambos procesadores fueron el resultado de convertir una línea de procesadores CISC escalares en arquitecturas superescalares. En contraste, el procesador K5 de AMD, que salió al mercado en 1996, fue diseñado desde el principio como un procesador CISC superescalar.

Cabe mencionar que este tipo de procesadores, como el Pentium o el K5, fueron implementados usando un núcleo RISC, es decir, las instrucciones complejas primero se convierten en micro-operaciones las cuales son ejecutadas posteriormente por este núcleo RISC. Desde su introducción, los procesadores superescalares han ganado popularidad y actualmente son la tecnología dominante en el mercado, existiendo diferentes clases que varían en su microarquitectura y desempeño, dependiendo de su campo de aplicación. Estos campos de aplicación van desde consolas de videojuegos, computadoras personales de escritorio y *Workstations* de alto desempeño, dominados hasta ahora por compañías como Intel y AMD, hasta

aplicaciones embebidas y de dispositivos móviles, dominadas por compañías como ARM Ltd. con sus procesadores ARM, cuyo primer prototipo, el ARM1 estuvo listo en 1985 y su primera versión comercial, el ARM2 llegó al mercado al siguiente año. Actualmente cerca del 90% de los procesadores embebidos de 32 bits utilizados en dispositivos electrónicos como teléfonos celulares, PDAs, reproductores de música digital, calculadoras y consolas de videojuegos portátiles, usan al menos un procesador de esta compañía.

1.2. Planteamiento del Problema

El diseño de las estructuras de hardware que constituyen los bloques funcionales de la arquitectura de un procesador no es una tarea trivial. Esta tarea se vuelve aún más complicada cuando la arquitectura del procesador que se desea diseñar es de tipo superescalar y con ejecución de instrucciones fuera de orden, debido a que, aunque se logre alcanzar un mayor desempeño, se incrementa la complejidad de las estructuras que la componen.

Para el caso de la estructura funcional conocida como Búfer de Reordenamiento de Instrucciones, el reto de diseño consiste en llegar a la propuesta de una arquitectura que no solo sea capaz de extraer grandes volúmenes de paralelismo a nivel de instrucción, mediante el aumento del tamaño la Ventana de Instrucciones, sino que también mejore las características que presentan las arquitecturas actuales, como los mecanismos de recuperación de errores que implementan, la cantidad de recursos que utilizan, o la gran cantidad de energía que consumen, todo esto sin incrementar demasiado la complejidad de dicha arquitectura, de modo que al final se obtenga un diseño novedoso que se caracterice por su baja complejidad y alto desempeño.

1.3. Justificación

En la medida en que un país sea capaz de desarrollar tecnología propia, será capaz también de eliminar su dependencia económica de otros países y generar riqueza que se vea reflejada en el bienestar de su población. Un claro ejemplo de esto es China, quién desde hace un par de décadas estableció una política de estado orientada a dar un fuerte impulso al desarrollo de la ciencia y la tecnología en ese país.

Un esfuerzo particular que se realizó como parte de esta política, fue la investigación y el desarrollo de un microprocesador de fabricación nacional que dio como resultado la introducción en 2002 de un CPU al que se le dio el nombre clave de *Godson-1*, el cual, aunque basado en la arquitectura MIPS, fue implementado completamente con tecnología china y es capaz de ejecutar el sistema operativo

Linux, más aún, este esfuerzo ha llevado incluso al desarrollo del microprocesador *Godson-3*, el cual será compatible con la arquitectura x86 de Intel y se espera que sea capaz de ejecutar el sistema operativo Windows [7].

De este modo, China está en condiciones de utilizar estos procesadores como motor de su creciente industria electrónica, al usarlos en toda clase de dispositivos, que van desde automóviles, electrodomésticos, dispositivos móviles y computadores personales, lo que le permite a ese país asiático obtener todos los beneficios que implica eliminar la dependencia de tecnología extranjera, entre los que se destaca el hecho de evitar pagar por patentes a otros países.

México podría seguir este ejemplo y obtener los mismos beneficios realizando un esfuerzo similar. Sin embargo la investigación y desarrollo en el área de diseño de arquitecturas de computadoras no se ha explotado lo suficiente, es por ello que este trabajo de tesis presenta una propuesta para el diseño de un Búfer de Reordenamiento de Instrucciones, el cual constituye una estructura funcional de gran importancia dentro de la microarquitectura de un procesador superescalar con ejecución fuera de orden, con lo cual se espera contribuir al desarrollo de esta área y motivar una mayor investigación que eventualmente puedan permitirle al país alcanzar un cierto grado de independencia tecnológica en el área en cuestión.

1.4. Objetivos

Para alcanzar resultados positivos en el esfuerzo de dar solución al problema descrito en la sección 1.2, es necesario establecer objetivos, tanto en forma general como particular, los cuales son presentados a continuación.

1.4.1. Objetivo General

- Realizar el diseño de un Búfer de Reordenamiento de Instrucciones para un procesador superescalar.

1.4.2. Objetivos Particulares

- Proponer una nueva arquitectura para el Búfer de Reordenamiento de Instrucciones para un procesador con ejecución fuera de orden, que se caracterice por su baja complejidad y alto desempeño, explicando además sus características y funcionamiento.
- Modelar mediante herramientas de software propias para la simulación de procesadores, la arquitectura propuesta para el Búfer de Reordenamiento de Instrucciones.

- Analizar y evaluar los resultados obtenidos tras realizar las simulaciones correspondientes para emitir un juicio respecto a las características observadas en dicha arquitectura.

1.5. Organización del Trabajo

El resto del trabajo está organizado de la siguiente manera: En el segundo capítulo se presenta el estado del arte correspondiente al Búfer de Reordenamiento de Instrucciones, esto incluye un estudio de la forma en que dicha estructura funcional es implementada por las microarquitecturas de los procesadores superescalares comerciales más importantes, así como un análisis de las propuestas encontradas en la literatura reciente para la implementación de esta estructura funcional.

En el capítulo 3 se presenta el marco teórico que describe el funcionamiento de un procesador superescalar con ejecución fuera de orden y en particular del Búfer de Reordenamiento de Instrucciones y se estudian detalladamente sus características.

El capítulo 4 presenta la propuesta para el diseño de un Búfer de Reordenamiento de Instrucciones para un procesador superescalar con ejecución fuera de orden, se describen las modificaciones realizadas al simulador utilizado para modelar dicha arquitectura y se explica el proceso para la realización de las simulaciones.

En el capítulo 5 se describen las pruebas que se realizaron al simular el funcionamiento del modelo en software del diseño y se presenta la evaluación de los datos obtenidos.

Finalmente en el capítulo 6 se presentan las conclusiones a las que se llega una vez concluido el proceso anterior y se propone el trabajo futuro que puede realizarse a partir de éste.

CAPÍTULO 2

ESTADO DEL ARTE

En este capítulo se presenta un análisis de los diferentes diseños de las estructuras funcionales encargadas de mantener el estado del procesador, implementadas por las microarquitecturas de tres diferentes procesadores superescalares con ejecución fuera de orden. Las micro-arquitecturas analizadas son las correspondientes a los procesadores MIPS R10000, Alpha 21264 y Pentium 3 y 4, este estudio se encuentra en las secciones 2.1, 2.2 y 2.3 respectivamente.

Adicionalmente, en la sección 2.4, se analizan varias propuestas encontradas en la literatura reciente donde se discuten varios esquemas novedosos para implementar el Búfer de Reordenamiento de Instrucciones.

2.1. El microprocesador MIPS R10000

El procesador MIPS R10000 (también llamado MIPS R10K) es un procesador RISC superescalar, es decir que es capaz de procesar más de una instrucción en cada ciclo de máquina. Es parte de una familia de procesadores que incluye, en forma cronológica, al R2000, R3000, R6000, R4400 Y R8000. El procesador R10000 se implementa usando tecnología CMOS VLSI de 0.35 µm en una superficie de 17x18 mm y contiene alrededor de 6.7 millones de transistores. Los procesadores MIPS previos al R10000 tenían arquitecturas segmentadas no superescalares, es decir, no podían ejecutar más de una instrucción en forma simultánea, sólo una instrucción era ejecutada en cada ciclo, sin embargo, la arquitectura del procesador MIPS R10000 permite ejecutar 4 instrucciones en forma paralela [4].

En idioma inglés el número de instrucciones que un procesador superescalar puede ejecutar en paralelo es expresado como *"W-way"*, siendo entonces el procesador MIPS R10000 un procesador *4-way*, es decir, en cada ciclo, cuatro instrucciones son manejadas en paralelo. Entre las características más importantes de este procesador se puede resaltar que implementa una arquitectura con un conjunto de instrucciones ***ISA*** de 32 bits. Cada pipeline consta de la etapa de extracción de instrucciones ***IF***, la etapa de decodificación de instrucciones ***ID***, la cual además incluye el renombramiento de instrucciones para evitar dependencias falsas.

En la etapa de emisión, ***IS,*** es donde las instrucciones se envían a una de sus tres colas de instrucciones (*instruction queues*), las cuales son: cola de enteros (*integer queue*), de punto flotante (*floating-point queue*) y de dirección (*address queue*). En la etapa de ejecución de instrucciones ***EX,*** se encuentran cinco unidades funcionales (*Functional Unit*), dos de las cuales son ALUs que realizan operaciones de números enteros, una FU de *load/store* para el cálculo de direcciones de memoria y dos FU para sumar y multiplicar números de punto flotante respectivamente y finalmente la etapa de escritura a registros ***WB***, este pipeline se muestra en la figura 2-1.

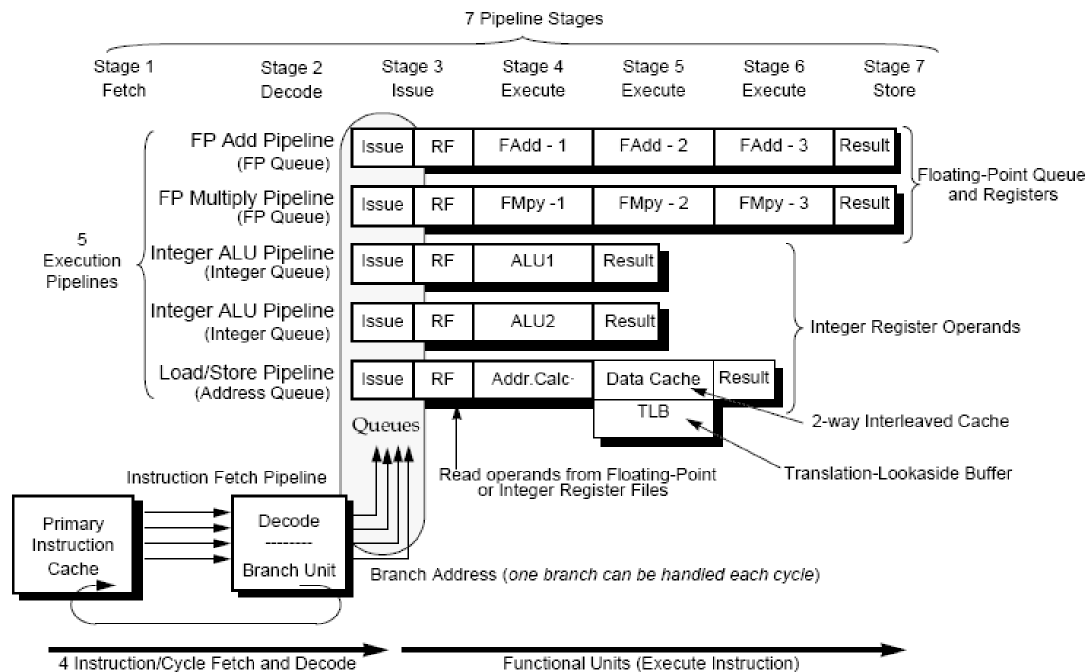


Figura 2-1. Pipeline del MIPS R10000.

El MIPS R10000 usa planificación dinámica de instrucciones y las ejecuta fuera de orden de programa pero dándole al programador la impresión de que las instrucciones se ejecutan en forma secuencial, desde la extracción hasta su retiro mediante el uso de una estructura funcional llamada Lista de Instrucciones Activas (*Active List).* Implementa también la emisión especulativa de instrucciones de control (*speculative branching*) y tiene un mecanismo para la recuperación precisa de excepciones en donde la Lista de Instrucciones Activas desempeña también una función muy importante, el diagrama de bloques de esta arquitectura se muestra en la figura 2-2.

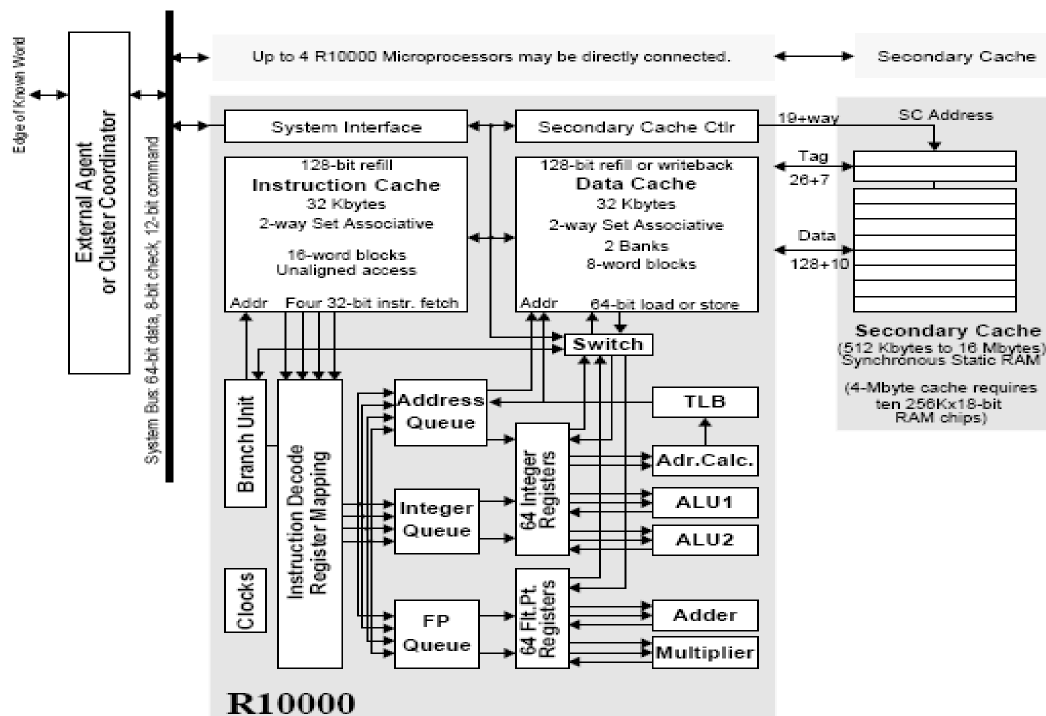


Figura 2-2. Diagrama de bloques del MIPS R10000.

2.1.1. La Lista de Instrucciones Activas del MIPS R10000

La estructura funcional conocida como Lista de Instrucciones Activas (*Active List)*, es equivalente a estructuras implementadas en otras arquitecturas superescalares con ejecución fuera de orden como el Búfer de Reordenamiento de Instrucciones (*ROB)* del Pentium 4. Para el MIPSR10K se trata de una lista de 32 entradas que contiene todas las instrucciones que se encuentran activas dentro del procesador [4]. A medida que se decodifica cada instrucción se va anexando al final de la Lista de Instrucciones Activas, esto significa que hasta 32 instrucciones pueden estar activas al mismo tiempo.

Cuando se introduce una instrucción en la lista de instrucciones activas, se introduce la dirección de un registro físico que mapea al registro destino lógico de la instrucción, que es la localidad en donde se almacenará el resultado de la instrucción, conocido como registro físico destino, se introduce también un segundo registro físico que es el último mapeo que se realizó antes al registro lógico destino de la instrucción (registro destino viejo), todo este proceso se lleva a cabo en la etapa de renombrado de registros.

La lista de instrucciones activas en ésta arquitectura consiste de cuatro FIFOs circulares de ocho entradas. La función de esta Lista de Instrucciones Activas es asegurar que aunque las instrucciones se hayan ejecutado fuera de orden, el procesador entregue el resultado de dichas instrucciones manteniendo el orden de

programa. Cada instrucción es identificada con una etiqueta de cinco bits que es la longitud de una dirección de una entrada en la Lista de Instrucciones Activas. Cuando una unidad funcional completa una instrucción, se envía una señal hacia su entrada en la Lista de Instrucciones Activas, la cual activa un bit de bandera especial (*done bit*).

Una instrucción es retirada de la parte superior de la Lista de Instrucciones Activas cuando esa instrucción y todas las precedentes se han completado. Si esta instrucción se retira (acción a la que en inglés se le conoce como: *instruction commit*) el valor calculado se escribe en los registros arquitecturales direccionados por el registro físico destino y el nombre del registro destino viejo es regresado a una lista de registros libres para volver a ser utilizado.

La Lista de Instrucciones Activas, además de mantener la información de las instrucciones que han sido ejecutadas, es capaz de liberar entradas (*flushing*) cuando se causa una excepción o cuando el resultado de una operación especulativa es incorrecto (*mispredicted branch*). Si una instrucción causa una excepción, las instrucciones subsecuentes no se retiran y entonces la Lista de Instrucciones Activas es usada para deshacer los resultados de cualquier instrucción sucesiva que haya sido ejecutada fuera de orden, estos resultados son eliminados simplemente borrando el contenido del registro físico que recibió el resultado.

Como las instrucciones ejecutadas fuera de orden no han sido retiradas aún, el archivo de registros (*register file*) contiene todavía los datos viejos en otros registros físicos, gracias a esto, si se encuentra un error de predicción de salto o una excepción, el procesador puede restaurar el estado del procesador y continuar la ejecución del programa con el estado correcto ya que el contador de programa PC se incrementa el número de instrucciones que se retiran en el ciclo previo. En el MIPS R10000, las instrucciones pueden ser deshechas a razón de cuatro por ciclo.

2.2. El microprocesador Alpha 21264

La segunda arquitectura que se estudia es la del procesador Alpha 21264 desarrollada por DEC (Compañía adquirida por COMPAQ en 1998, la cual, a su vez fue adquirida por HP en 2002). El Alpha 21264 es un procesador RISC de tercera generación, después del Alpha 21064 y del 21164. El Alpha fue diseñado como una arquitectura de 64 bits. Todos los registros tienen 64 bits de longitud y todas las operaciones son realizadas entre registros de 64 bits.

Es un procesador superescalar con ejecución fuera de orden implementado en tecnología CMOS de 0.35 µm en un área aproximada de $3\,cm^2$. Al igual que el MIPS R10000, emplea ejecución fuera de orden, es decir, las instrucciones pueden ser

ejecutadas en un orden diferente al del orden en que fueron extraídas y decodificadas [5].

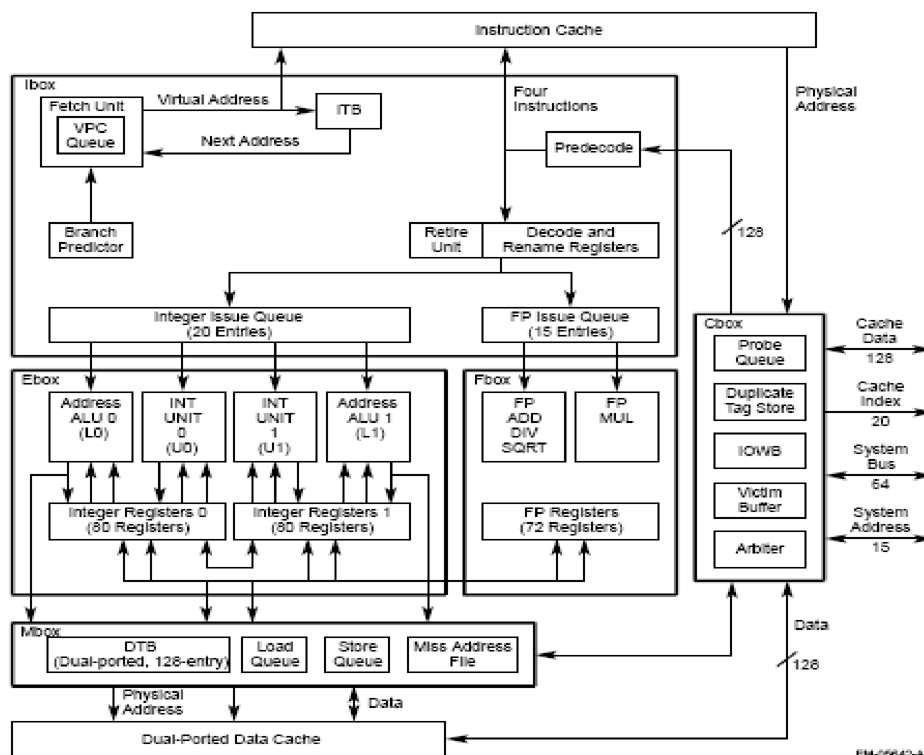


Figura 2-3. Diagrama de bloques del Alpha 21264.

El Alpha 21264 consiste de las siguientes secciones, mostradas en la figura 2-3.

- La sección Ibox que incluye las unidades de Fetch, Issue y Retire.
- La sección Ebox que incluye la unidad de ejecución de enteros.
- La sección Fbox que incluye la unidad de ejecución de punto flotante.
- La secciones Icache y Dcache que contiene las unidades de memoria de datos e instrucciones.
- La unidad de Cbox que contiene el sistema de interfaces.
- La secuencia de operación en pipeline.

El pipeline del procesador Alpha 21264, se muestra en la figura 2-4. En la primera etapa de este pipeline, cuatro instrucciones de 32 bits de longitud son extraídas de la caché de instrucciones de 64KB en cada ciclo. La unidad de Fetch usa varias técnicas para predicción de saltos para generar un camino de datos (*datapath*) de ejecución en el pipeline. En caso de que ocurra una excepción o un error de predicción de salto, la etapa de extracción redirige el camino de ejecución y reinicia el pipeline.

En la siguiente etapa llamada *rename*, las instrucciones son renombradas para evitar dependencias falsas. En el renombramiento de registros se eliminan las

dependencias de registros *write-after-write* y *write-after-read*, pero se preservan todas las dependencias *read-after-write* que son necesarias para preservar la consistencia en los resultados del programa. El renombramiento registros extrae el máximo de paralelismo de una aplicación pues sólo las dependencias verdaderas son mantenidas y además permite que las instrucciones puedan ser ejecutadas en forma especulativa dentro del flujo de instrucciones.

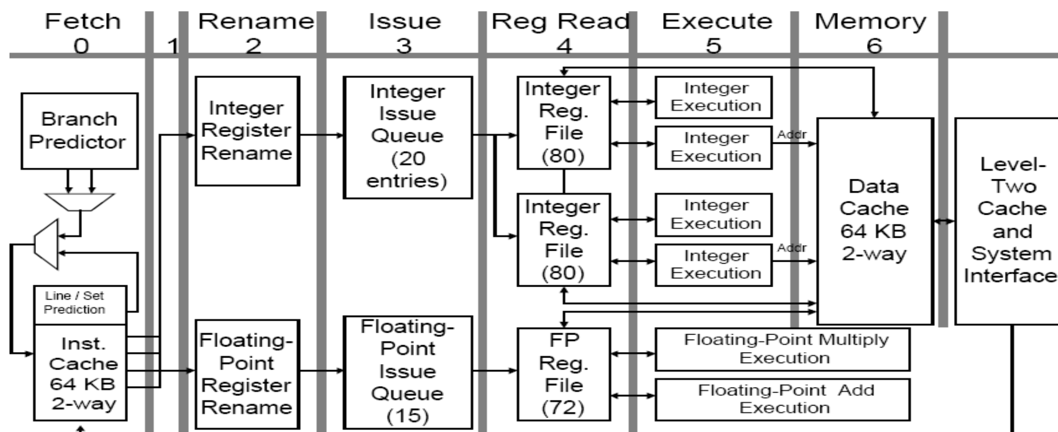


Figura 2-4. Pipeline del Alpha 21264.

Cada ciclo, cuatro instrucciones son cargadas en las dos colas de emisión, estas colas son la de números enteros y la de punto flotante. La cola de punto flotante (Fqueue) tiene 15 entradas y puede emitir dos instrucciones por ciclo. La cola de enteros (Iqueue) elige cuatro instrucciones de sus 20 entradas y las emite hacia las unidades funcionales cuando éstas están disponibles. Las colas pueden enviar las instrucciones en forma especulativa. A las instrucciones más viejas se le da prioridad sobre las más nuevas. En cada ciclo, cada cola selecciona la instrucción más vieja que tenga listos sus operandos y la unidad funcional que esté disponible.

2.2.1. La Ventana de Instrucciones del Alpha 21264

La Ventana de Instrucciones del procesador Alpha 21264 puede almacenar 80 instrucciones, esto implica que hasta 80 instrucciones pueden estar en diferentes estados parciales de ejecución antes de ser completadas, permitiendo concurrencia en su ejecución y reduciendo su tiempo de latencia. El tamaño de la Ventana de Instrucciones está estrechamente relacionada con la cantidad de registros físicos, por ejemplo en el Alpha 21264 además de los 64 registros estructurales o lógicos (32 de enteros y 32 de punto flotante), se cuenta con registros físicos para enteros y para punto flotante para almacenar los resultados de las operaciones antes de que

las instrucciones se retiren (commit) y sean retiradas de la ventana, de tal forma que puede mantener 80 instrucciones al vuelo (*80 instruction in-flight window*) [5].

El Alpha 21264 mantiene un registro de las instrucciones pendientes que no han sido retiradas así como la información de sus respectivos registros para que el procesador pueda preservar su estado estructural en caso de que ocurra un error en la especulación (*misspeculation*). El bloque Ibox extrae las instrucciones en orden de programa, las ejecuta fuera de orden y entonces las retira en orden. La lógica de retiro de instrucciones de Ibox mantiene el estado arquitectural del procesador retirando una instrucción sólo si todas las instrucciones previas se han ejecutado sin generar excepciones o errores de salto.

2.3. Los procesadores Pentium 3 Y 4

La microarquitectura P6 constituye la base del Pentium Pro, Pentium II y Pentium III. Además de un conjunto de instrucciones extendidas, estos tres procesadores difieren entre sí en la frecuencia de reloj, la arquitectura de la caché y de la interfaz de memoria. La microarquitectura P6 convierte cada instrucción IA-32 en una serie de micro-operaciones (también llamadas: "*µops*") que son ejecutadas por el pipeline. Las micro-operaciones son similares a las instrucciones RISC típicas. Tres instrucciones IA-32 son extraídas, decodificadas y convertidas en µops cada ciclo de reloj. Si una instrucción se decodifica en más de cuatro µops, ésta se implementa mediante una secuencia de micro-código que genera las µops necesarias en múltiples ciclos de reloj.

Las µops son ejecutadas por un pipeline especulativo fuera de orden usando renombramiento de registros y un ROB. Hasta tres µops pueden ser renombradas por ciclo de reloj y enviadas a las estaciones de reserva (*reservation station*). También se retiran hasta tres instrucciones por ciclo. El pipeline del P6, que se muestra en la figura 2-5, está estructurado en 14 etapas, compuestas por lo siguiente:

- Ocho etapas son usadas para las operaciones de fetch, decode y dispatch realizadas en orden de programa. La siguiente operación es seleccionada usando un predictor de saltos de 512 entradas.
- Tres etapas son usadas para ejecución fuera de orden en una de las cinco unidades funcionales, las cuales son la unidad de enteros, unidad de punto flotante, la unidad de salto, unidad de acceso a memoria y de dirección de memoria.
- Tres etapas son usadas para el retiro de instrucciones (instruction commit).

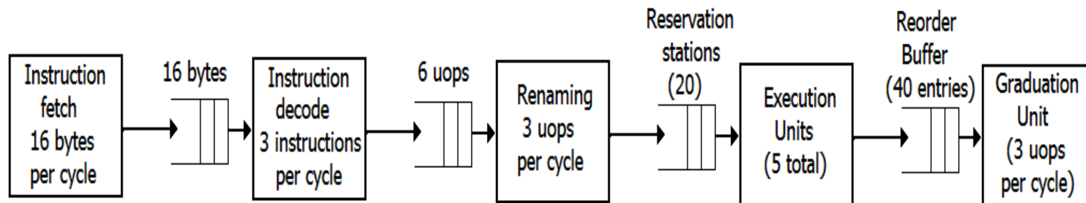


Figura 2-5. Pipeline del P6.

La microarquitectura del Pentium 4, llamada NetBurst es similar a la del Pentium III (llamada arquitectura P6), ambas extraen 3 instrucciones IA-32 por ciclo y las decodifican en micro-operaciones y envían estas μops a un motor de ejecución fuera de orden que puede retirar hasta 3 operaciones por ciclo [3]. Sin embargo existen diferencias que le permiten al NetBurst operar a una frecuencia de reloj significativamente más alta que el P6 y con la misma capacidad (*throughput*).

El procesador Pentium 4, basado en la arquitectura del NetBurst está implementado en tecnología CMOS de 0.18 micras en un área de 217 mm^2. El diagrama de bloques de la microarquitectura NetBurst del procesador Pentium 4 se muestra en la figura 2-6. En este diagrama se muestran las cuatro secciones principales. La etapa inicial llamada *front-end* que trata las instrucciones en orden, un motor de ejecución (*execution engine*) que ejecuta las instrucciones fuera de orden, las unidades de ejecución de enteros y de punto flotante y el subsistema de memoria. A continuación se hace una descripción de cada una de estas secciones.

La sección *In-Order Front End*, es la parte del procesador que extrae las instrucciones que van a ser ejecutadas y las prepara para ser usadas posteriormente en el pipeline [6]. Se encarga de proveer un flujo de instrucciones decodificadas a la sección de ejecución fuera de orden en donde se realiza el retiro de estas instrucciones. El *front end* tiene una lógica de predicción de saltos muy exacta que usa la historia pasada de la ejecución del programa para especular el modo en que el programa se ejecutará a continuación. La dirección de la instrucción predicha por la lógica de predicción de saltos del *front end* es usada para extraer bytes del nivel L2 de la caché. Los bytes de estas instrucciones IA-32 son entonces decodificadas en operaciones básicas llamadas μops que pueden ser ejecutadas en la siguiente etapa.

El decodificador de instrucciones recibe los bytes de una instrucción IA-32 de la caché L2 de 64 bits, los decodifica en las primitivas llamadas micro-operaciones (μops) que posteriormente se ejecutan. Este decodificador de instrucciones puede decodificar una instrucción por ciclo de reloj, muchas instrucciones IA-32 son convertidas en una sola μop y otras necesitan varias μops para completar una instrucción IA-32, por ejemplo la instrucción *string move,* que puede estar compuesta de cientos de μops.

La microarquitectura del NetBurst tiene una forma avanzada de caché de instrucciones de nivel 1 (L1) llamada *Execution Trace Cache*. Esta caché se encuentra

entre la lógica de decodificación de instrucciones y la de ejecución de instrucciones. En esa posición, la *Trace Cache* es capaz de almacenar las instrucciones IA-32 que ya han sido decodificadas (μops).

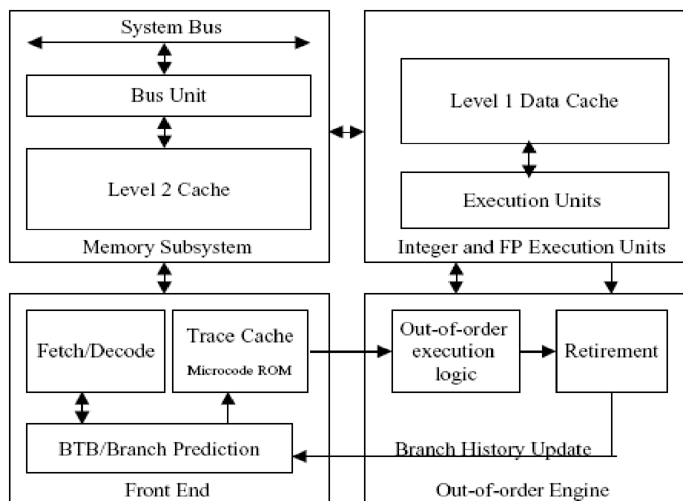


Figura 2-6. Diagrama de bloques del NetBurst.

De esta manera, las instrucciones son decodificadas una sola vez y colocadas en la Trace Cache, entonces son usadas repetidamente desde ahí como se hace en la caché de instrucciones de los procesadores descritos previamente. El decodificador de instrucciones IA-32 sólo es usado cuando se produce un fallo en caché (*cache miss)* y entonces es necesario ir al nivel L2 para obtener y decodificar los bytes de la nueva instrucción IA-32. La lógica de ejecución fuera de orden prepara a las instrucciones para su ejecución.

Esta sección cuenta con diferentes búferes usados para reordenar el flujo de instrucciones para optimizar su desempeño a medida que pasan por el pipeline y son dispuestas para ser ejecutadas. El motor de ejecución de instrucciones fuera de orden tiene las funciones de asignación, renombrado y planificación (*scheduling).* La lógica del asignador (*allocator*) se encarga de asignar los recursos que necesita cada μop para ejecutarse.

Si un recurso necesario, como una entrada del register file, no está disponible para una de las tres μops que van hacia el asignador en un ciclo de reloj, el asignador detiene al procesador. Cuando los recursos están disponibles, el asignador asigna las μops y permite que continúen el flujo por el pipeline para ser ejecutadas. El asignador también asigna una entrada del búfer de reordenamiento **ROB**, la cual sigue el estatus de cada una de las 126 μops que pueden estar ejecutándose al vuelo simultáneamente en el procesador.

Las instrucciones son reordenadas para permitir que sean ejecutadas en cuanto sus operandos estén listos. La ejecución fuera de orden permite que las

instrucciones se ejecuten siempre y cuando sus operandos no dependan de resultados de instrucciones previas. También permite que los recursos de las unidades de ejecución como ALUs y memoria caché estén ocupados el mayor tiempo posible ejecutando las instrucciones independientes que estén listas para ser ejecutadas.

La lógica de renombrado de registros renombra los registros lógicos de las instrucciones IA-32 como EAX con las 128 entradas del registro de archivos físicos del procesador. Esto permite que los 8 registros lógicos definidos arquitecturalmente se expandan a los 128 registros físicos en el procesador Pentium 4. La lógica de renombramiento recuerda la dirección más reciente de cada registro como el EAX, en una estructura conocida como Register Alias Table (RAT) y entonces cada nueva instrucción procedente del pipeline puede saber donde obtener la instancia actual de cada uno de sus registros operandos de entrada. La lógica de retiro se encarga del reordenamiento de las instrucciones, las cuales, como se mencionó previamente, fueron ejecutadas fuera de orden y son regresadas al orden original de programa.

La lógica de retiro recibe el estado de la ejecución de las instrucciones desde las unidades de ejecución y procesa los resultados de manera que las instrucciones sean retiradas de acuerdo al orden de programa. El procesador Pentium 4 puede retirar hasta tres μops por ciclo de reloj. La lógica de retiro de instrucciones se asegura de que las excepciones ocurran sólo si la operación que la causa es la operación más vieja que aún no ha sido retirada. Esta lógica también reporta la historia de los saltos al predictor de saltos en el *front end* del procesador.

2.3.1. El Búfer de Reordenamiento de Instrucciones en NetBurst

La microarquitectura NetBurst asigna y renombra los registros de forma diferente a la microarquitectura P6. Como se muestra en la figura 2-7, en la microarquitectura del P6, los registros de resultados y las entradas del *reorder buffer* son tomados como una sola entidad con campos para datos (*data field*) y de estado (*status field*). El campo de datos del ROB es usado para almacenar los valores resultantes de las micro-operaciones y, el campo de estado, para seguir el estado de las μops, a medida que van siendo ejecutadas por el procesador.

Estas entradas del ROB son asignadas y desasignadas secuencialmente y son apuntadas mediante una secuencia numérica que indica la edad de estas entradas. Al momento de retirar una instrucción, su resultado es copiado desde el campo de datos del ROB hacia una entidad separada llamada *Retirement Register File* (RRF) [6]. El RAT apunta a la versión reciente de cada registro arquitectural.

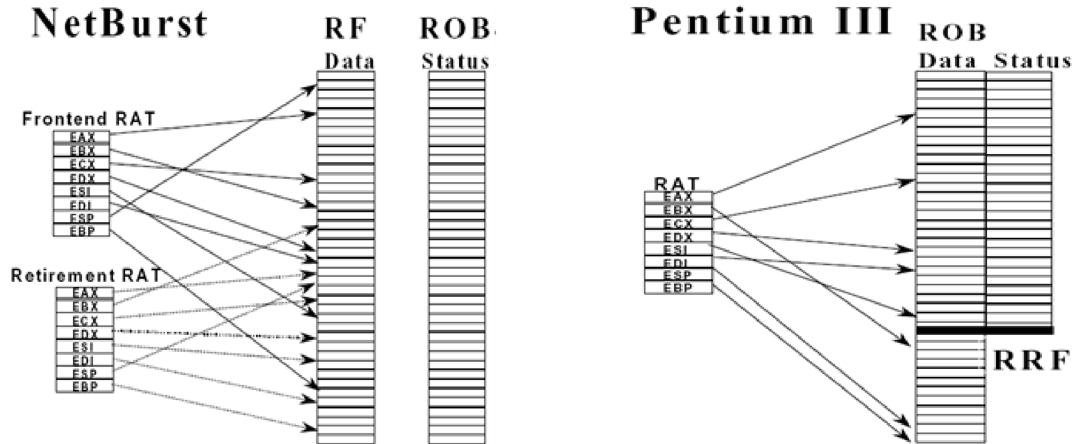


Figura 2-7. ROB en las microarquitecturas NetBurst y P6.

Como se puede observar en la figura 2-7, las entradas del ROB y las del Register File de datos están separadas. Las entradas del ROB que siguen el estado de las micro-operaciones consisten de sólo el campo de estado y son asignadas y desasignadas secuencialmente. Un número de la secuencia asignado a cada micro-operación indica su edad. Un número de la secuencia apunta a la entrada de la µop en el arreglo del ROB, el cual es similar al de la microarquitectura del P6.

El registro físico asignado, se toma de una lista de registros que se encuentran libres en el RF de 128 entradas, a diferencia de las entradas del ROB. De esta manera, al momento de retirar una instrucción, ningún resultado necesita ser movido de una estructura física a otra.

2.4. Propuestas recientes

Además de los procesadores comerciales que implementan en sus microarquitecturas una estructura funcional dedicada a mantener el estado del procesador, como es el caso de la Lista de Instrucciones Activas (*Active List*) del MIPS R10000, o del Búfer de Reordenamiento de Instrucciones (*ReOrder Buffer*) del Pentium 4, descritas en las secciones previas, existen en la literatura diversas propuestas recientes para realizar la implementación de dicha estructura funcional mediante diferentes esquemas, los cuales se discuten a continuación.

En el esquema propuesto en la serie de trabajos reportados del 2003-2004 [21][22][23][24][25] se considera el problema de que el diseño típico del ROB es una gran estructura multi-puerto, como la que se muestra en la figura 2-8.

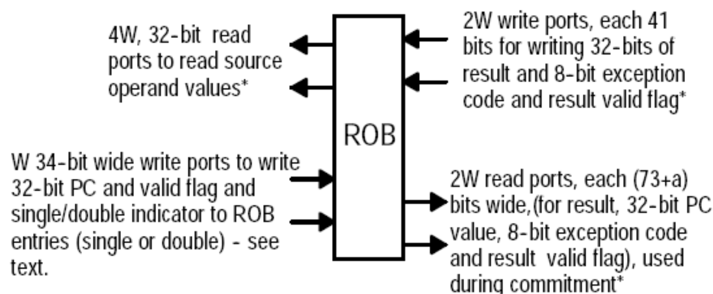


Figura 2-8. El ROB como una gran estructura FIFO multipuerto.

Se explica que esta estructura ocupa una porción significativa del área del chip y disipa una fracción considerable del total de la potencia del mismo, por lo que para una arquitectura como la que se muestra en el diagrama de bloques de la figura 2-9, resulta compleja e ineficiente, por lo que se propone reducir la complejidad de esta estructura mediante una implementación descentralizada que divide al ROB en estructuras separadas relacionadas con las unidades funcionales de la etapa de ejecución con cada componente distribuido, diseñado de acuerdo a la carga de trabajo de cada unidad funcional.

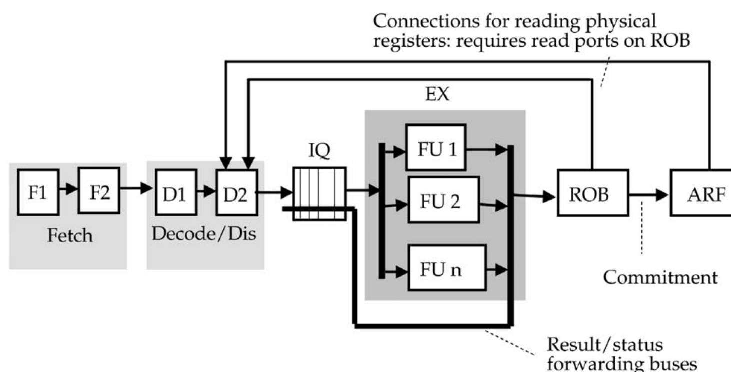


Figura 2-9. Arquitectura superescalar que emplea al ROB.

En este esquema, el único componente centralizado del ROB es una estructura principal FIFO que mantiene los apuntadores a las entradas de los componentes de este ROB descentralizado. Cada componente del ROB (ROBC) es implementado también como una lista FIFO. El apuntador a la entrada del ROB centralizado se compone de dos números: un identificador de la unidad funcional y un valor de offset dentro del ROBC asociado. La estructura del ROB es utilizada también para implementar la técnica de renombrado de registros. Esto se hace usando las entradas que componen el ROB como registros físicos.

Además los resultados de las instrucciones son almacenados en las entradas del ROB, y son escritos al *register file* en la etapa de retiro de instrucciones. Se explica además que la latencia de la recuperación del procesador debido a errores de predicción de salto no se incrementa con la descentralización de la estructura ROB. Para reconstruir el estado correcto del procesador, el apuntador de cola de cada

ROBC es actualizado cada vez que ocurre un error de predicción de salto junto con el apuntador de cola del componente principal del ROB.

De acuerdo a lo reportado en dicho trabajo, un esquema descentralizado del ROB, cuyo diagrama de bloques se muestra en la figura 2-10, aporta beneficios tanto en la disminución de la complejidad de su arquitectura como en la disminución de la cantidad de energía disipada por la misma.

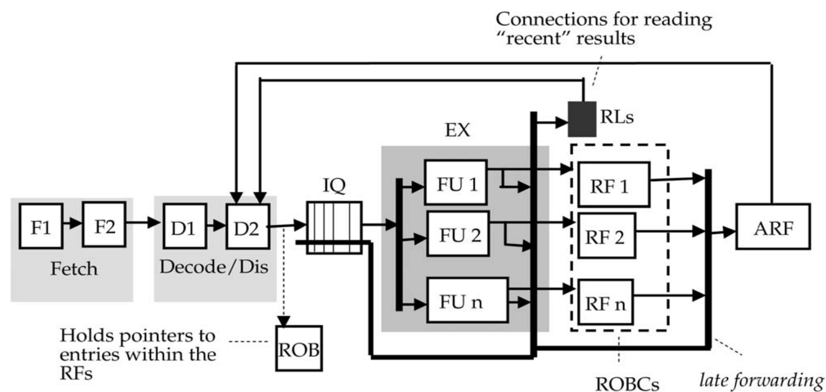


Figura 2-10. Propuesta para un ROB separado en componentes.

Se puede destacar el hecho de que es posible reducir la complejidad de la estructura del ReOrder Buffer, en este caso limitando el número de puertos de lectura/escritura utilizados haciendo una relación entre la carga de trabajo de cada unidad de ejecución con el tamaño del componente del ROB asociado, todo esto sin afectar el desempeño total del procesador y por otra parte se observa que esta disminución en la complejidad de la arquitectura del ROB puede incluir una separación de la estructura centralizada en componentes separados de menor tamaño. Esta separación tiene el objetivo de evitar la concentración de la disipación de la potencia en un área específica del procesador, lo cual se logra según lo reportado en [24] tras una serie de simulaciones de las subestructuras del ROB, usando la herramienta SPICE.

Otros trabajos, como los presentados en [30][31], destacan el hecho de que incrementar el tamaño de la Ventana de Instrucciones, es decir, el número de entradas disponibles en el ROB para alojar el estado de las instrucciones que se encuentran activas en el procesador, aumenta directamente el desempeño del mismo. Esto se debe a que se expone mayor paralelismo a nivel de instrucciones y a que se mantienen ocupadas en lo más posible las unidades funcionales del procesador.

Sin embargo, de acuerdo a este trabajo, incrementar el tamaño de la Ventana de Instrucciones sin tomar en cuenta otras consideraciones, no es una estrategia de diseño adecuada pues hay que recordar que se tienen restricciones tanto de área, lo que limita el espacio físico disponible para alojar esta estructura, como de disipación

de potencia, pues a mayor tamaño del ROB, mayor será el área en la que se concentrará la disipación de energía, lo cual puede exceder la capacidad de los mecanismos de disipación de potencia del sistema.

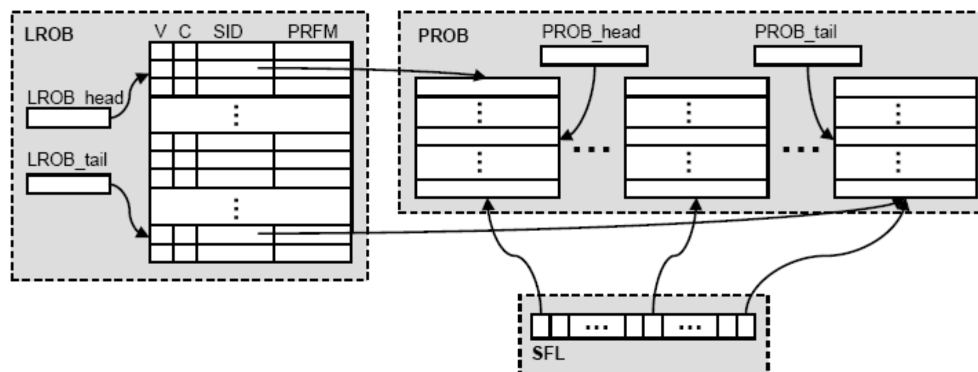


Figura 2-11. Propuesta para un ROB comprimido.

Ante el compromiso de diseño que debe existir entre incrementar el tamaño de la Ventana de Instrucciones para mejorar el desempeño del procesador y el no exceder restricciones dadas de área y de disipación de potencia, la propuesta estudiada sugiere una arquitectura nombrada de ROB comprimido. Esta arquitectura está basada en un ROB típico, es decir, una estructura FIFO circular, con apuntadores de cabeza y cola. Se propone una arquitectura llamada CROB (Compressed ROB) la cual es capaz de comprimir sus entradas y dar la ilusión de tener un ROB virtual de mayor tamaño que el número real de entradas, como se muestra en la figura 2-11.

Esto permite tener un mayor número de instrucciones al vuelo que entradas en el ROB. Esto es logrado introduciendo un nivel de indirección similar a la manera en que las páginas de memoria virtual son mapeadas a las páginas de memoria física en los sistemas operativos modernos.

Se propone que el ROB físico (PROB) del procesador sea dividido en segmentos físicos de igual tamaño que son mapeados dinámicamente hacia segmentos lógicos por medio de una tabla de mapeo llamada *Logical ReOrder Buffer* (LROB). Un segmento físico es liberado cuando se valida que todas las instrucciones que se mantienen en ese segmento serán retiradas o eliminadas (debido a un error de predicción de salto). Cada segmento del PROB contiene W entradas, un apuntador de cabeza y uno de cola hacia la instrucción más vieja y más nueva respectivamente.

El LROB debe tener un número mayor de entradas que el número de segmentos del PROB. El objetivo de esta estructura es mantener el orden en el que los segmentos del PROB fueron asignados. De esta manera, una nueva entrada del LROB es asignada cada vez que un nuevo segmento PROB es asignado o reasignado, de manera que el número de entradas en esta estructura define el número máximo

de instrucciones que el procesador puede tener al vuelo. Este número máximo de instrucciones es igual al número de entradas del LROB multiplicado por el tamaño del segmento del PROB. El LROB es una estructura FIFO circular, en la cual, aparte de tener un apuntador de cabecera y un apuntador de cola para dicha estructura (*LROB_head* y *LROB_tail*), el LROB maneja también los apuntadores de cabeza y de cola del PROB (*PROB_head* y *PROB_tail*).

En [15] se presenta una propuesta de microarquitectura llamada Checkpoint Processing and Recovering (CPR), en la cual, en lugar de usar un ROB, se propone crear *checkpoints* en saltos considerados poco confiables. Esto es, se implementa un mecanismo de recuperación en el cual se limitan el número de checkpoints a puntos cuidadosamente seleccionados. Idealmente, se intentan crear checkpoints exactamente en instrucciones que generen errores de predicción de salto. Entonces, a diferencia del mecanismo típico de checkpointing, donde los checkpoints son creados en cada instrucción de salto o bien cada cierto número de instrucciones, se crea una serie de checkpoints en cada instrucción de salto con una alta probabilidad de que cause un error de salto, la cual es seleccionada usando un esquema de estimación de confianza.

Este estimador usa una tabla de 4 bits, un contador, una compuerta XOR, junto con la dirección de salto y una tabla de la historia de los saltos (*branch history table*). Una predicción correcta incrementa el contador y un error de predicción de salto reinicia el contador a cero. Un valor de 15 del contador señaliza alta confianza mientras que los demás valores indican baja confianza. De esta manera, los checkpoints son entonces creados en saltos de poca confianza y adicionalmente cada 256 instrucciones.

Cabe destacar que en este esquema, el predictor de saltos es actualizado de manera especulativa, es decir, a diferencia del ROB, el mecanismo de checkpointing no cuenta con un campo dedicado para almacenar un valor de bandera que indique que el salto ha sido tomado correctamente, es decir que no ha habido error de predicción de salto. Entonces el predictor de saltos se actualiza cuando la instrucción se salto se termina de ejecutar, no cuando ésta se retira.

De Manera similar, en [16] se presenta un mecanismo híbrido basado en ROB y checkpointing, al que se le da el nombre de *CHeckpointed Early Resource RecYcling* (Cherry). La idea de la microarquitectura Cherry se basa en separar el reciclado de recursos utilizados por una instrucción, de la etapa de retiro de instrucciones. La implementación Cherry recicla los recursos tan pronto como resulten innecesarios en el curso normal de la operación. De esta manera, los resultados son usados de forma más eficiente.

Sin embargo, el reciclado temprano de recursos puede dificultarle al procesador lograr un estado arquitectural consistente. En consecuencia, se crea un checkpoint en hardware de sus registros arquitecturales. Este checkpoint puede ser usado para

regresar a un estado consistente previo en caso de ser necesario. Para realizar el reciclado tempano de registros, se identifica una entrada en el ROB como el Punto de No Retorno (PNR). El PNR corresponde a la instrucción más vieja que aún puede sufrir un error de predicción de salto, como se observa en la figura 2-12, el reciclado temprano de registros está permitido sólo para instrucciones más viejas que el punto de no retorno.

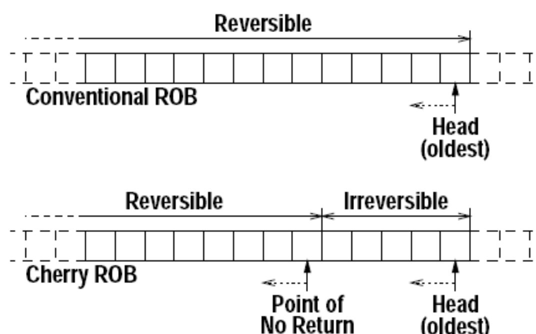


Figura 2-12. Comparación del ROB convencional con el Cherry ROB.

Las instrucciones que no son más viejas que el PNR son llamadas reversibles, estas instrucciones, cuando ocurre un error de predicción de salto, son tratadas como en un esquema convencional para ejecución fuera de orden. Las instrucciones que son más viejas que el PNR son llamadas irreversibles. Estas instrucciones pueden haber o no terminado su ejecución, sin embargo, algunas de ellas pudieron haber liberado ya sus recursos. ´

Por otra parte en [28] se propone una implementación basada también en los mecanismos de ROB y checkpointing a la que se le da el nombre de *Turbo-ROB* (TROB). El cual puede complementar o reemplazar el mecanismo de ROB. El TROB es similar al ROB pero requiere menos entradas porque sólo almacena información de pocas instrucciones de salto seleccionadas, llamadas puntos de restauración, usadas para recuperar el estado del procesador. Como los programas tienden a reusar registros frecuentemente, muchas instrucciones son ignoradas por el TROB.

En contraste, el ROB almacena información de todas las instrucciones porque asume que debe ser capaz de recuperarse de cualquier instrucción. Mientras que el ROB típico trata todas las instrucciones de igual manera, el TROB es optimizado para almacenar sólo para las instrucciones de salto.

El TROB almacena un subconjunto de la información almacenada por el ROB. En una implementación propuesta, cada entrada del TROB contiene un apuntador del ROB, por lo que es necesario un campo para almacenar este dato. La detección de la primera actualización de cada registro después de un punto de reparación es realizado con la ayuda de una estructura llamada *Architectural Register Bitvector* (ARB), la cual tiene un tamaño igual al del register file.

En [32] se usa una aproximación diferente, se parte de la observación de que las instrucciones dependientes de una operación de alta latencia (p.ej. cache miss) no se pueden ejecutar hasta que esa operación fuente sea completada. Estas instrucciones son movidas fuera de la cola de emisión convencionalmente de tamaño pequeño, hacia una estructura de mayor tamaño llamada *Waiting Instruction Buffer* (*WIB*), como se muestra en la figura 2-13.

Cuando la operación de alta latencia se completa, las instrucciones son reinsertadas en la cola de emisión de instrucciones. En este diseño las instrucciones permanecen en la cola de emisión de instrucciones, durante my poco tiempo pues inician su ejecución rápidamente o son movidas hacia el WIB debido a que dependen de operaciones de alta latencia.

Removiendo estas instrucciones del camino crítico, las entradas que ocupaban previamente en la cola de emisión de instrucciones pueden ser utilizadas por el procesador para buscar más ILP. De esta manera se consigue que las instrucciones en las entradas del ROB sean instrucciones que impliquen operaciones de baja latencia lo cual es positivo ya que históricamente, los ROBs pequeños han predominado en los diseños de las microarquitecturas modernas, pues permiten utilizar mayores frecuencias de reloj, sin embargo un ROB pequeño puede llenarse rápidamente cuando aparece una operación de alta latencia, lo cual deteriora el desempeño del procesador.

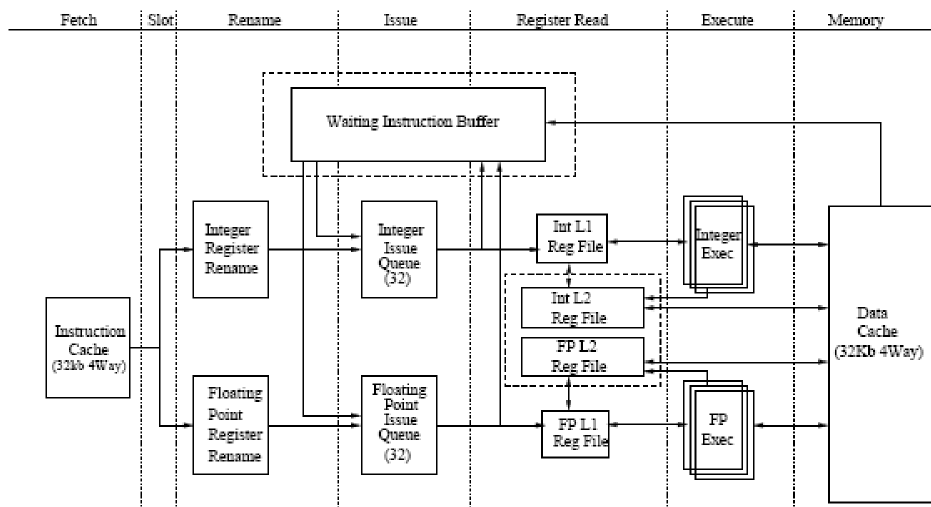


Figura 2-13. Microarquitectura basada en WIB.

Por lo que la manera de evitar esto es traer nuevas instrucciones al ROB lo que implica que se debe incrementar su tamaño. Por ello, mover estas instrucciones de latencia alta, hacia un búfer de espera de instrucciones permite utilizar estructuras ROB de tamaño pequeño sin afectar el desempeño del procesador pues adicionalmente se sugiere que las instrucciones almacenadas en el WIB podrían ejecutarse por medio de un motor de ejecución de instrucciones paralelo.

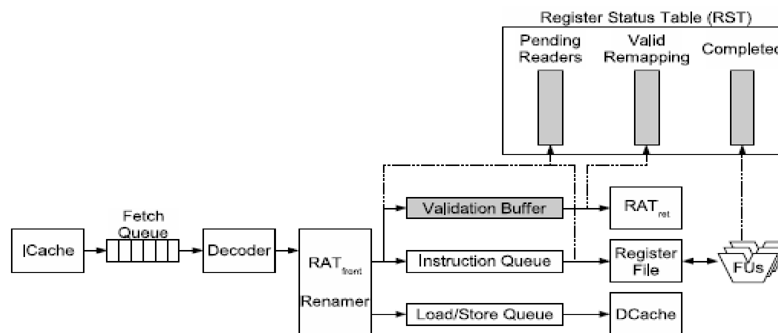


Figura 2-14. Microarquitectura que usa el VB como sustituto del ROB.

Finalmente, cabe destacar algunas de las propuestas que sugieren realizar un retiro de instrucciones de manera fuera de orden original del programa, las propuestas de [17][18] presentan un modelo de microarquitectura en donde se sustituye el uso del ROB por un una estructura llamada *Validation Buffer* (VB), en la cual, las instrucciones que aloja son retiradas tan pronto como su estado especulativo es resuelto, como se muestra en el diagrama de la figura 2-14. También se propone un agresivo mecanismo de reclamo de registros.

En este esquema, se permite que una instrucción sea retirada del *Validation Buffer* una vez que todos los saltos condicionales y excepciones sean resueltos, esto es, cuando su estado especulativo sea resuelto. Sin importar si dicha instrucción ha completado su ejecución, sólo se ha emitido o simplemente decodificado y no emitido. Se explica que este mecanismo mejora el desempeño del procesador y requiere de menos recursos.

2.5. Resumen del Capítulo

En este segundo capítulo se explicó que los procesadores superescalares comerciales implementan, con diferentes arquitecturas, la estructura funcional conocida como Búfer de Reordenamiento de Instrucciones, así mismo se describieron las características de tres de estas arquitecturas. También se presentaron varias propuestas, encontradas en la literatura reciente, donde se muestran varios esquemas alternativos que pretenden mejorar el desempeño de esta estructura funcional.

En el siguiente capítulo se explicarán con detalle las características de un procesador superescalar así como las diversas técnicas que se emplean para mejorar su desempeño y se describen la arquitectura y funciones de un ROB típico.

Adicionalmente se presentan las herramientas de software usadas para el modelado y simulación de nuevas arquitecturas.

CAPÍTULO 3

MARCO TEÓRICO

Este capítulo presenta en la sección 3.1 las bases teóricas del funcionamiento de los procesadores superescalares y en particular del Búfer de Reordenamiento de Instrucciones, sobre el cual se centra el estudio de esta Tesis.

Posteriormente, en la sección 3.2 se describen las características de las herramientas de software para la simulación y evaluación del desempeño de diferentes microarquitecturas de procesadores.

3.1. Los Procesadores Superescalares

Los primeros procesadores superescalares fueron introducidos en la década de 1990 y, desde entonces, se han vuelto la arquitectura más comúnmente implementada en los procesadores modernos [8]. El objetivo principal del procesamiento superescalar es incrementar el número de instrucciones que son ejecutadas en cada ciclo de reloj, a este indicador que se conoce como IPC.

El valor del IPC no puede ser aumentado a más de uno si la microarquitectura del procesador es escalar, es decir, que implementa sólo un pipeline, porque entonces sólo una instrucción es capaz de comenzar su ejecución en cada ciclo, como se muestra en la figura 3-1, por ello la microarquitectura de un procesador superescalar implementa dos o más pipelines de modo que más de una instrucción pueda ser procesada en el mismo ciclo, como se muestra en la figura 3-2, incrementando de esta manera el valor del IPC y por lo tanto el desempeño del procesador.

Figura 3-1. Pipeline de una arquitectura escalar.

Al número máximo de instrucciones que un procesador es capaz de ejecutar en paralelo, se le conoce con el término de *W-way o* ancho del procesador. Las arquitecturas superescalares más recientes son capaces de ejecutar hasta 4 instrucciones por ciclo, por lo que se les conoce como procesadores 4-way.

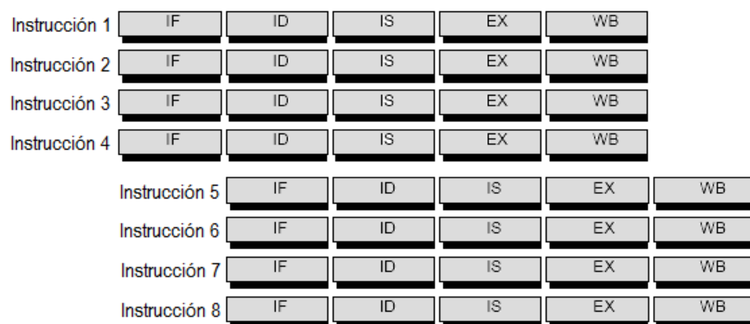


Figura 3-2. Pipeline de una arquitectura superescalar.

3.1.1. Paralelismo a nivel de instrucción y ejecución fuera de orden

El alto desempeño alcanzado por un procesador superescalar moderno, como el mostrado en la figura 3-3, es posible gracias a las características de su micro-arquitectura. Esta microarquitectura implementa técnicas de hardware para detectar y ejecutar múltiples instrucciones en paralelo en cada ciclo. A esta técnica que consiste en explotar el paralelismo de las instrucciones para mejorar el desempeño de los procesadores, se le ha dado el nombre de Paralelismo a Nivel de Instrucción (ILP, por sus siglas en inglés). Para poder aprovechar el ILP, es necesario determinar cuáles de las instrucciones que forman un programa en particular, pueden ser ejecutadas en paralelo.

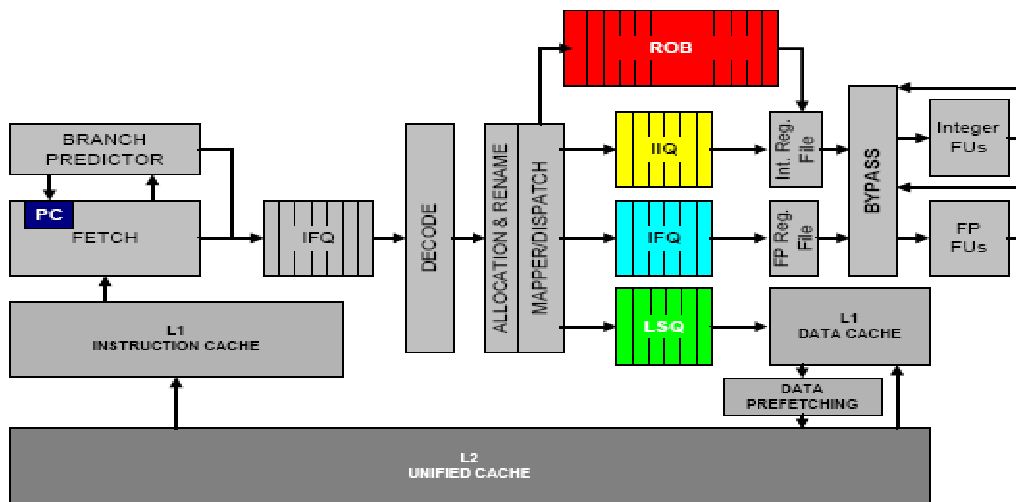


Figura 3-3. Diagrama de bloques de un procesador superescalar.

Asumiendo que existan suficientes recursos (registros físicos, unidades funcionales), dos instrucciones paralelas podrían ejecutarse a lo largo del pipeline simultáneamente sin causar que el procesador se detenga, pero si estas dos instrucciones son dependientes, no son paralelas, y entonces necesariamente se deben ejecutar en forma secuencial, lo que limita el rendimiento del procesador.

Cuando existen dependencias entre instrucciones, pudiera suceder que durante su ejecución en paralelo, estas instrucciones accedan (realicen una operación de lectura o escritura) a un mismo registro o localidad de memoria, en un orden diferente al del programa original, causando que el resultado obtenido sea incorrecto. A un evento de este tipo se le da el nombre de riesgo de datos (*data hazard).* Determinar cuáles de estos *hazards* puede ser eliminados, mediante técnicas de hardware, como el renombrado de instrucciones, y cuáles no, es de suma importancia para explotar adecuadamente el ILP.

Si se consideran dos instrucciones *i* y *j*, donde *i* ocurre antes que *j* en la secuencia del programa, un tipo de data *hazard* ocurre cuando *j* trata de leer un registro antes de que *i* escriba en él, de manera que *j* lee el valor viejo del registro, lo cual es incorrecto, a este tipo de *hazard* se le llama RAW (*Read After Write*) y se trata de una dependencia verdadera, por lo que el orden de programa debe respetarse para asegurar que *j* reciba correctamente el valor de *i*. Otro tipo de *hazard* ocurre cuando *j* trata de escribir un operando antes de que sea escrito por *i*.

Esto causa que el valor actual del registro sea el valor de *i* lo cual es incorrecto. Este hazard corresponde a una dependencia de salida y se le conoce como WAW (*Write After Write*). Finalmente cuando *j* trata de escribir en un registro antes de ser leído por *i*, se genera el hazard WAR (*Write After Read*), pues *i* lee un valor incorrecto escrito por *j.* En este caso se trata de una anti-dependencia, que puede resolverse, como en el caso del *hazard* tipo WAW, mediante técnicas de renombrado de registros. En la figura 3-4 se ilustran los tres tipos de *data hazards*.

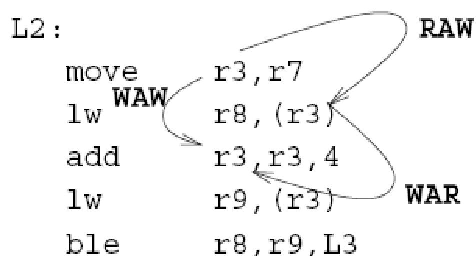


Figura 3-4. Diagrama que ilustra la dependencia entre instrucciones.

Como se ha mencionado, detectar las dependencias entre instrucciones es importante ya que se requiere que las instrucciones comiencen su ejecución lo antes posible, lo cual está limitado por la disponibilidad de los datos de sus operandos (y también por la cantidad de recursos disponibles). Eliminando la dependencia de

datos de instrucciones, el procesador superescalar aprovecha el ILP al ejecutar un conjunto de instrucciones secuenciales, es decir un programa, no necesariamente en dicho orden secuencial, sino en función de que sus operandos estén disponibles, las instrucciones por lo tanto son ejecutadas en un orden diferente, a dicho concepto se le conoce como ejecución de fuera de orden (OoO, por sus siglas en inglés), un diagrama que ilustra este concepto se muestra en la figura 3-5.

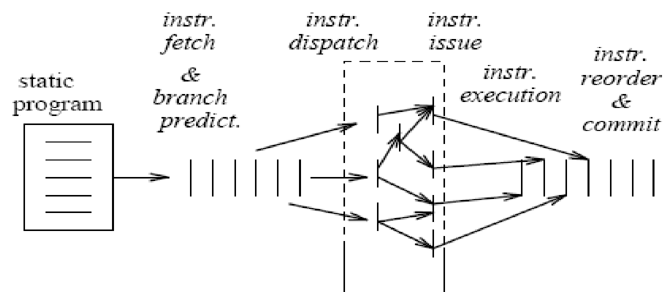


Figura 3-5. Diagrama que ilustra la ejecución de instrucciones fuera de de orden.

Mediante la ejecución de instrucciones fuera de orden, se logra incrementar el desempeño de un procesador superescalar, pues éste no detiene su ejecución a causa de falsas dependencias, sin embargo, es necesario reordenar las instrucciones antes de actualizar los registros arquitecturales del procesador, porque de lo contrario, los resultados obtenidos serían erróneos. Por ello se requiere una estructura funcional dentro de la microarquitectura del procesador que mantenga el estado secuencial de las instrucciones para obtener resultados coherentes y que además lleve a cabo otras tareas, como habilitar la recuperación del estado del procesador después de que ocurra una excepción o un error de predicción de salto, lo cual permite que se lleve a cabo la ejecución de instrucciones en forma especulativa [1].

Como se vio en el capítulo 2, en las microarquitecturas superescalares actuales, existe tal estructura funcional, la cual, aunque varia en su denominación (ROB, Active List) y en la forma de ser implementada en los diferentes procesadores, realiza básicamente las mismas funciones. El estudio de las características de esta importante estructura funcional, parte clave en el funcionamiento de la microarquitectura de un procesador superescalar, se presenta en la siguiente sección.

3.1.2. La Ventana de Instrucciones y el ROB

La Ventana de Instrucciones de un procesador superescalar se define como el conjunto de instrucciones que han sido decodificadas, renombradas, despachadas, emitidas, y en algunos casos ejecutadas, pero que aún no han sido retiradas del pipeline [32], es decir, la Ventana de Instrucciones está conformada por todas las

instrucciones que se encuentran aún activas o al vuelo (en inglés se ocupa el término: “*in-flight instruction”*) en el procesador.

El tamaño de la Ventana de Instrucciones es un importante parámetro de diseño para muchos procesadores modernos [12][13][14]. Una Ventana de Instrucciones grande, ofrece el potencial de exponer grandes cantidades de paralelismo a nivel de instrucción, lo cual, en combinación con otras técnicas como la ejecución especulativa de instrucciones, incrementa significativamente el IPC del procesador.

Como se sabe, la ejecución especulativa de instrucciones es posible gracias a la implementación de una unidad de predicción de saltos condicionales. Este predictor de saltos, elige una traza o camino de datos mediante diversos esquemas, que pueden ser tan simples como tomar siempre (*always taken*) o nunca (*never taken*) las instrucciones de salto condicional. Todas las instrucciones que se ejecutan posteriormente de acuerdo al camino elegido por el predictor, lo hacen de manera especulativa, puesto que no se tiene la certeza de que el camino tomado sea el correcto hasta varios ciclos después, cuando mediante comparaciones, es posible validar el resultado de la predicción.

Utilizar esta técnica permite que, para que una instrucción pueda ser ejecutada, se dependa sólo de la disponibilidad de sus operandos, pudiéndose ejecutar las instrucciones de cierto programa en un orden diferente al que originalmente tenía. Sin embargo, esta manera especulativa de ejecutar las instrucciones puede causar que se pierda la consistencia de los resultados del programa [8].

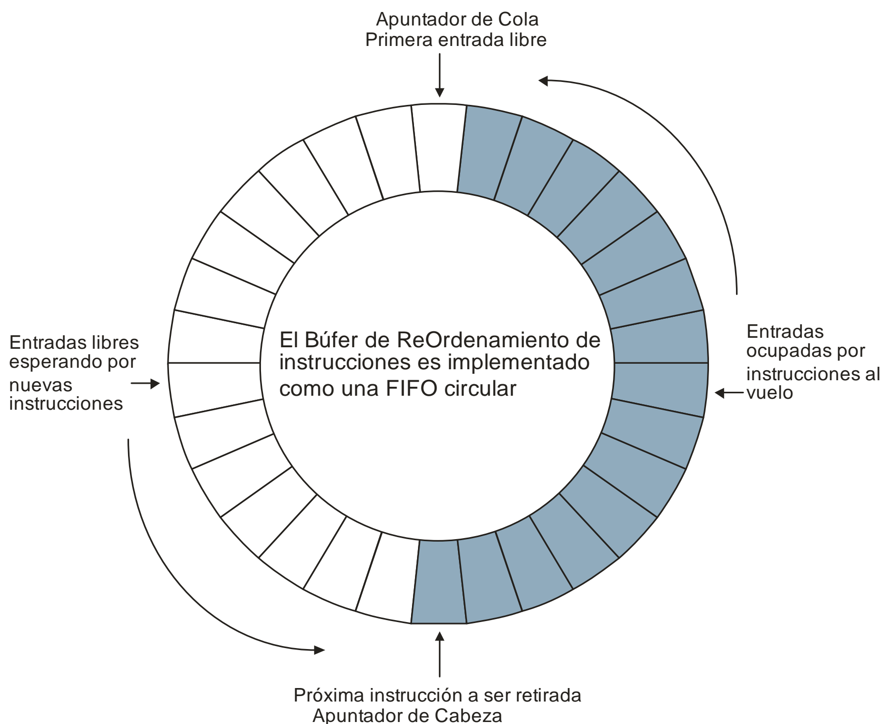


Figura 3-6. Diagrama conceptual de un ROB.

Para solucionar este inconveniente se ideó una estructura funcional descrita por primera vez en 1988 por Smith y Pleszkun [10] llamada Búfer de Reordenamiento de instrucciones (*ROB*, por sus siglas en inglés), el cual tenía el objetivo principal de asegurar un orden en el retiro de las instrucciones manteniendo el estado de cada instrucción que se encontrara al vuelo dentro del pipeline, permitiendo así una consistencia en la obtención de resultados.

El ROB puede verse conceptualmente como una estructura FIFO circular con W entradas, donde el número de entradas determina la cantidad de instrucciones que podrán encontrarse al vuelo, es decir, el valor W del ROB define el ancho de la Ventana de Instrucciones del procesador [2].

El mecanismo del ROB funciona mediante dos apuntadores, uno de cabeza (head pointer) y otro de cola de cola (tail pointer), de manera similar al diagrama mostrado en la figura 3-6. A medida que una instrucción es despachada hacia alguna de las colas de instrucciones, se añade una entrada correspondiente a dicha instrucción en la cola del ROB incrementándose en ese caso el valor del apuntador de cola.

De una manera similar, cuando una instrucción llega a la cabeza de la FIFO, significa que es la instrucción más vieja dentro del pipeline y si cumple ciertos requisitos, puede ser entonces retirada (acción que en inglés es conocida como "*instruction commit*") [2]. Es en la etapa de *commit* donde se evalúa si dicha instrucción cumple con las condiciones necesarias para esto.

Estas condiciones exigen que la instrucción haya sido despachada adecuadamente a una de las colas de instrucciones, que tenga los operandos necesarios para iniciar su ejecución, que la unidad funcional a la que fue asignada haya terminado de ejecutarla y finalmente que no se encuentre en un estado especulativo. De esta manera se asegura que las instrucciones sean retiradas en estricto orden de programa.

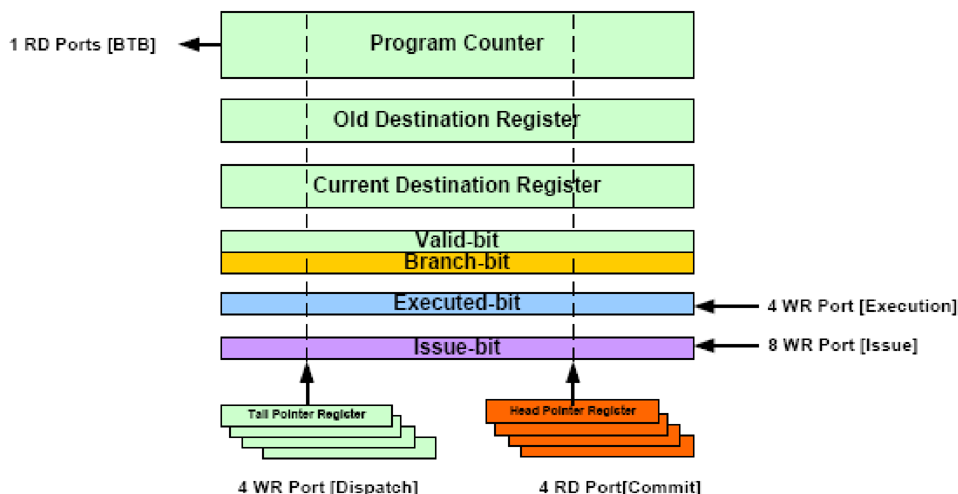


Figura 3-7. Campos de datos de una entrada del ROB.

Como se ha mencionado, cada entrada del ROB almacena información correspondiente a cada instrucción al vuelo dentro del pipeline. Como se muestra en el diagrama de la figura 3-7, cada una de las entradas contiene campos para almacenar las banderas de estado que se activan cuando una instrucción termina una etapa del pipeline, estas son, una bandera para *Dispatched*, *Issued*, *Executed* y *Non-Speculative*. Adicionalmente se almacena el contador del programa, dato que es proporcionado al predictor de saltos para realizar comparaciones y determinar si los caminos elegidos por éste fueron correctos.

Otras tareas de las que se encarga el ROB incluyen otorgarle al procesador la capacidad de restaurar su estado en caso de que exista un error de predicción de salto condicional, pues al detectarse esta situación, todas las instrucciones ejecutadas después de dicho error de salto son eliminadas (acción que en inglés se describe con los términos: *flushing o squashing*). También está encargada de emitir una señal para indicarle al mecanismo de emisión de instrucciones que se detenga, en caso de que el apuntador de cabeza se encuentre una entrada antes que el apuntador de cola, lo cual implica que la FIFO se encuentra llena. Se encarga también de la tarea del reciclado de registros físicos liberados y del retiro de instrucciones completadas.

3.2. Herramientas de software para la simulación de Arquitecturas

La simulación mediante herramientas de software es usada ampliamente en muchos campos de la ciencia y de la ingeniería. Las simulaciones son particularmente útiles en situaciones donde realizar experimentos con un sistema real no resulta factible o cuando se requiere evaluar el desempeño de un nuevo sistema aún por construir. Dentro del área de la ingeniería de cómputo, la arquitectura de computadoras es una de las ramas que depende en mayor medida de la simulación, debido a que la complejidad de las arquitecturas de los microprocesadores actuales hace que la evaluación de nuevos diseños, orientados a mejorar su desempeño, se realice en base a los resultados obtenidos en dichas simulaciones, sin las cuales esto sería poco factible o imposible [33][38].

Se utilizan modelos de software del hardware que se desea construir, los cuales se implementan mediante lenguajes de programación tradicionales o mediante lenguajes de descripción de hardware, a estos modelos se les asigna una carga de trabajo (workload) para validar el desempeño del diseño de hardware propuesto. La implementación de un modelo de software tiene tres requerimientos críticos, el desempeño, la flexibilidad y el detalle [33].

El desempeño determina la carga de trabajo que el modelo puede realizar dados los recursos disponibles para la simulación. La flexibilidad indica qué tan bien el modelo está estructurado para simplificar su modificación, permitir variantes en el

diseño o incluso crear diseños completamente diferentes con facilidad. El detalle define el nivel de abstracción usado para implementar los componentes del modelo. La simulación permite acelerar el proceso de diseño porque el modelo de software de un diseño de hardware en particular puede ser construido y probado en mucho menos tiempo que el hardware mismo, adicionalmente utilizar un simulador de hardware proporciona ventajas como la facilidad de evaluar diferentes diseños sin tener la necesidad de construir costosos sistemas de hardware, permite probar componentes o sistemas no existentes y permite obtener métricas detalladas del desempeño del sistema, es decir, un modelo altamente detallado que simula todos los aspectos de la operación de la máquina incluso si cierto aspecto en particular no es importante para alguna métrica a medir.

En la práctica, optimizar las tres características del modelo es difícil, por ello, la mayoría de los modelos implementados optimiza solo una o dos de ellas. Lo cual explica porqué existen tantos modelos de software aun para el diseño de un solo producto. Los modelos orientados a la investigación tienden a optimizar el desempeño y la flexibilidad sobre el detalle.

Los simuladores de procesadores pueden ser construidos a varios niveles de abstracción, por ejemplo, se puede modelar al procesador a nivel de micro-arquitectura, a nivel de compuerta o a nivel de circuito. Comúnmente el primer paso en la evaluación de un diseño se realiza a nivel de micro-arquitectura. Sólo cuando se encuentra que el diseño es viable entonces se realiza la simulación a otros niveles. Algunas propiedades deseables en un simulador es que debe ser preciso, rápido y flexible.

Estas cualidades no se pueden obtener a la vez. El nivel de detalle determina su exactitud, a mayor detalle, el modelo será más exacto, sin embargo, mayor detalle también significa que mayor tiempo tardará en terminar una simulación. Un simulador a nivel de microarquitectura, esencialmente hace uso de modelos de alto nivel de los componentes individuales que constituyen a un procesador.

3.2.1. El conjunto de herramientas de simulación SimpleScalar

SimpleScalar es una herramienta que consiste de un grupo de simuladores que emulan la microarquitectura de un procesador a diferentes niveles de detalle. Esta herramienta fue desarrollada por Todd Austin mientras era estudiante de doctorado en la Universidad de Wisconsin y en la actualidad es mantenida por la corporación SimpleScalar LLC y es distribuida gratuitamente y como código abierto en la página web: *www.simplescalar.com*.

Desde que se hizo público, SimpleScalar se ha convertido en una de las herramientas más usadas entre los investigadores del área de arquitectura de

computadoras, quienes ocupan esta herramienta para evaluar sus nuevos diseños, pues les proporciona una infraestructura para el modelado de sistemas de computadoras y simplifica la implementación de diseños de hardware [34]. El conjunto de simuladores de SimpleScalar va desde *sim-safe,* un simulador básico que emula sólo el conjunto de instrucciones, hasta *sim-outorder*, un modelo detallado de la microarquitectura de un procesador con ejecución especulativa y un sistema de memoria multinivel.

Tabla 3-1. Características de los simuladores que conforman a SimpleScalar

Simulador	Descripción	Líneas de código	Velocidad de simulación
sim-safe	Simulador funcional simple	320	6 MIPS
sim-fast	Simulador funcional orientado a rápida ejecución	780	7 MIPS
sim-profile	Simulador funcional optimizado	1300	4 MIPS
sim-bpred	Simulador de predictor de saltos	1200	5 MIPS
sim-cache	Simulador de memorias caché multinivel	1400	4 MIPS
sim-cheetah	Simulador de memorias caché multinivel optimizado	2300	5 MIPS
sim-outorder	Simulador detallado de un modelo micro-arquitectural	3900	0.3 MIPS

En la tabla 3-1 se muestra una descripción de los simuladores que componen a SimpleScalar. La arquitectura de software del modelo de hardware de SimpleScalar se muestra en la figura 3-8. Las aplicaciones se ejecutan en el modelo usando una técnica de simulación llamada *execution-driven*, la cual reproduce la operación interna de un dispositivo.

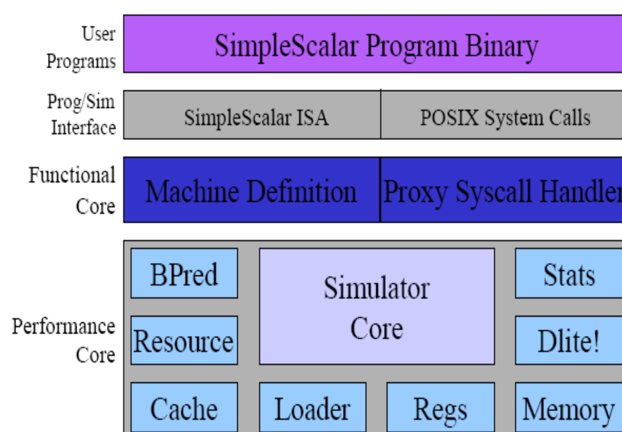


Figura 3-8. Arquitectura del simulador SimpleScalar.

Para modelos de sistemas de computadoras, este proceso requiere que se reproduzca la ejecución de las instrucciones en la máquina simulada y tiene como

principal característica el proporcionar acceso a todos los datos producidos durante la ejecución del programa, estos valores son cruciales para el estudio de la microarquitectura y para la posterior implementación de optimizaciones a la misma.

La arquitectura del modelo de SimpleScalar está basada en la arquitectura del conjunto de instrucciones (ISA) PISA (MIPS IV) o Alpha, pero al añadir y modificar ciertas características puede considerarse un superconjunto del mismo. Todas las instrucciones son de una longitud de 64 bits.

3.2.2. El simulador *sim-outorder*

Como se ha mencionado, el simulador más detallado dentro del conjunto de simuladores de SimpleScalar, es *sim-outorder.* Este simulador implementa un pipeline de seis etapas como el mostrado en la figura 3-9, en donde cada etapa es simulada mediante una función dentro del código de sim-outorder. En la función principal llamada *sim_main(),* se encuentra un ciclo como el mostrado en la figura 3-10, el cual, al ejecutarse, representa un ciclo de máquina y las funciones que simulan cada una de las etapas del pipeline son ejecutadas en sentido contrario para evitar errores de sincronización. Durante la ejecución del programa, el simulador genera estadísticas las cuales son presentadas al final de la misma [35].

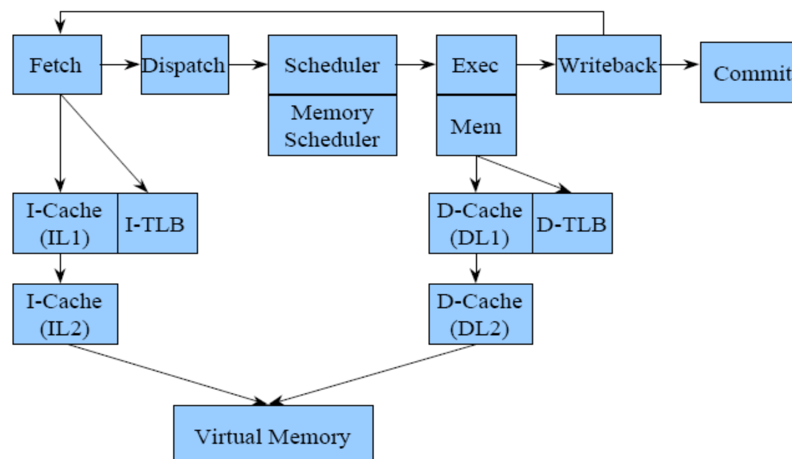


Figura 3-9. Diagrama de bloques del pipeline implementado por sim-outorder.

La etapa de fetch del pipeline está implementada en la función *ruu_fetch( ).* La unidad de fetch define el ancho de instrucciones que son extraídas de la caché de la máquina y toma las siguientes entradas: El contador de programa (PC), el estado del predictor y la detección de errores de salto. Cada ciclo trae instrucciones de la caché de instrucciones, después de lo cual las coloca en la cola de dispatch.

El código de la etapa de dispatch reside en la función *ruu_dispatch( ).* En esta función es donde la decodificación y el renombramiento de registros se llevan a

cabo. La función usa las instrucciones en su cola de entrada. En cada ciclo se toma la mayor cantidad de instrucciones posibles (determinadas por el ancho de *fetch*) y son escritas en la RUU tomando en cuenta un apuntador hacia la cola de la misma.

La etapa de issue del pipeline está contenida en *ruu_issue( )* y *lsq_refresh( )*. Estas funciones modelan el *wakeup* e *issue* de las instrucciones hacia las unidades funcionales llevando un registro de las dependencias de registros que se presentan. Cada ciclo, las instrucciones cuyos registros de entrada están listos son enviadas a la cola de instrucciones.

La etapa de ejecución también es manejada en *ruu_issue( ).* En cada ciclo, la función toma de la cola tantas instrucciones listas como sea posible (determinado por el valor de *issue width*). La disponibilidad de las unidades funcionales también se revisa y si tienen puertos de acceso disponibles, las instrucciones son emitidas. Las latencias de las unidades funcionales están predefinidas, lo cual sirve para realizar las estadísticas finales.

La etapa de *writeback* reside en *ruu_writeback( ).* Cada ciclo busca en la cola de eventos instrucciones completadas, cuando encuentra una instrucción completada busca las instrucciones que dependan de ella y si esta instrucción dependiente sólo espera el resultado de dicha instrucción, es marcada como lista para ser emitida.

La etapa de *commit,* es modelada por la función *ruu_commit( )*, en donde las instrucciones que han sido completadas son retiradas en orden y se actualizan los registros arquitecturales. La función retira las instrucciones a la cabeza de la RUU sólo si se han completado y entonces las entradas de la RUU son liberadas.

```
ruu_init();
for (;;) {
  ruu_commit();
  ruu_writeback();
  lsq_refresh();
  ruu_issue();
  ruu_dispatch();
  ruu_fetch();
}
```

Figura 3-10. Ciclo principal del simulador sim-outorder.

Como se ha comentado, este simulador soporta la emisión y ejecución de instrucciones fuera de orden basándose en una estructura llamada *Register Update Unit* (RUU) [34], la cual es una combinación de un búfer de reordenamiento de isntricciones, ROB, y de estaciones de reserva, el diagrama de bloques de esta estructura se muestra en la figura 3-11.

Este esquema usa un ROB para mantener el estado de las instrucciones, renombrar automáticamente los registros y para mantener los resultados de las instrucciones pendientes. Cada ciclo, las instrucciones completadas son retiradas del ROB en orden de programa y su resultado es escrito en los registros arquitecturales.

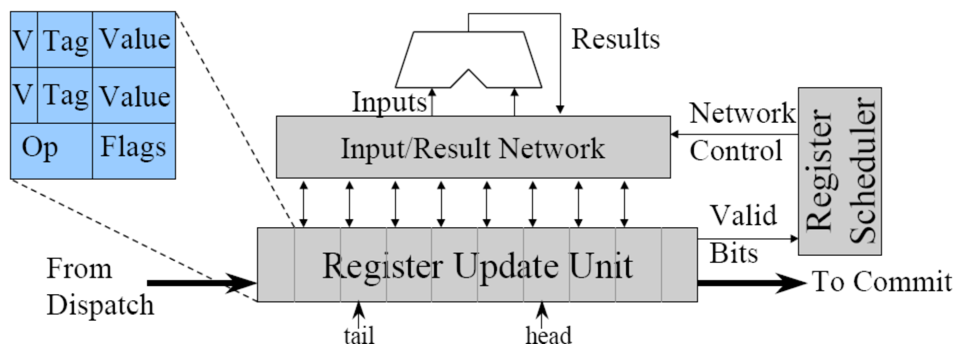


Figura 3-11. Diagrama de bloques del RUU implementado por sim-outorder.

3.3. Resumen del Capítulo

En este tercer capítulo se explicaron las características de los procesadores superescalares, así como las técnicas que se implementan para incrementar su desempeño, como el renombramiento de registros, la predicción de saltos condicionales y la ejecución especulativa de instrucciones.

Enseguida se discutió la arquitectura y función que tiene la estructura conocida como Búfer de Reordenamiento de instrucciones y como en conjunto con las técnicas mencionadas habilita la ejecución de instrucciones fuera de orden de programa, técnica que al ser implementada por un procesador superescalar logra incrementar en gran medida su desempeño.

De esta manera, se conocen las características de un ROB, lo cual, con lo encontrado en el segundo capítulo, esto es, las diferentes arquitecturas encontradas en procesadores comerciales y en propuestas recientes, forman las bases para discutir las características de un ROB típico que puedan estar sujetas a una optimización, para entonces proponer el diseño de una nueva arquitectura mejorada para el ROB, la cual al implementarse pueda incrementar el desempeño de un procesador, dicho proceso se realiza en el siguiente capítulo.

CAPÍTULO 4

DISEÑO DEL ROB

En este capítulo se presenta el proceso que se sigue para realizar el nuevo diseño de la arquitectura del Búfer de Reordenamiento de Instrucciones. En la sección 4.1 se explica la metodología que se aplica para tal fin. En la sección 4.2 se describen los criterios que se toman en cuenta para realizar el diseño del ROB. En la sección 4.3 se describen las características de la arquitectura propuesta para el Búfer de Reordenamiento de Instrucciones de un procesador superescalar con ejecución de instrucciones fuera de orden.

Después, en la sección 4.4 se presenta la generación del modelo de la arquitectura mediante el uso del simulador sim-outorder de SimpleScalar. Finalmente, en la sección 4.5 se comenta el proceso de simulación llevado a cabo para evaluar esta propuesta de diseño.

4.1. Metodología de diseño

La metodología que se sigue para realizar el diseño del Búfer de Reordenamiento de Instrucciones permite que al final de este proceso se obtenga el modelo de la arquitectura de ROB que tendrá las características necesarias para que realice adecuadamente las funciones de las que está encargado y que se han descrito en la sección 3.1, y permite además que sea posible introducir mejoras en el modelo de su arquitectura con el objetivo de incrementar el desempeño general del procesador.

La metodología inicia realizando un estudio de las características de las arquitecturas implementadas por los procesadores modernos así como de las propuestas recientes encontradas en la literatura. En base a esta información se buscan posibles puntos de mejora para entonces realizar la propuesta de diseño se tomando en cuenta las características que se desean así como los criterios y restricciones que se tienen.

Enseguida es necesario realizar una simulación mediante un modelo de software de dicha arquitectura, esto se hace mediante la modificación del código del simulador sim-outorder, de SimpleScalar, el cual es utilizado para analizar y evaluar la nueva arquitectura propuesta.

Un conjunto de estadísticas obtenidas tras la ejecución de las simulaciones permitirán obtener una aproximación de su desempeño, en base a la cual se podrán

realizar modificaciones posteriores con el objetivo de mejorar la esta arquitectura. El flujo de diseño que describe la metodología que se utiliza para el diseño de la arquitectura del ROB se muestra en el diagrama de la figura 4-1. Cabe mencionar que dicha metodología puede aplicarse también, a otros bloques funcionales que conforman al procesador.

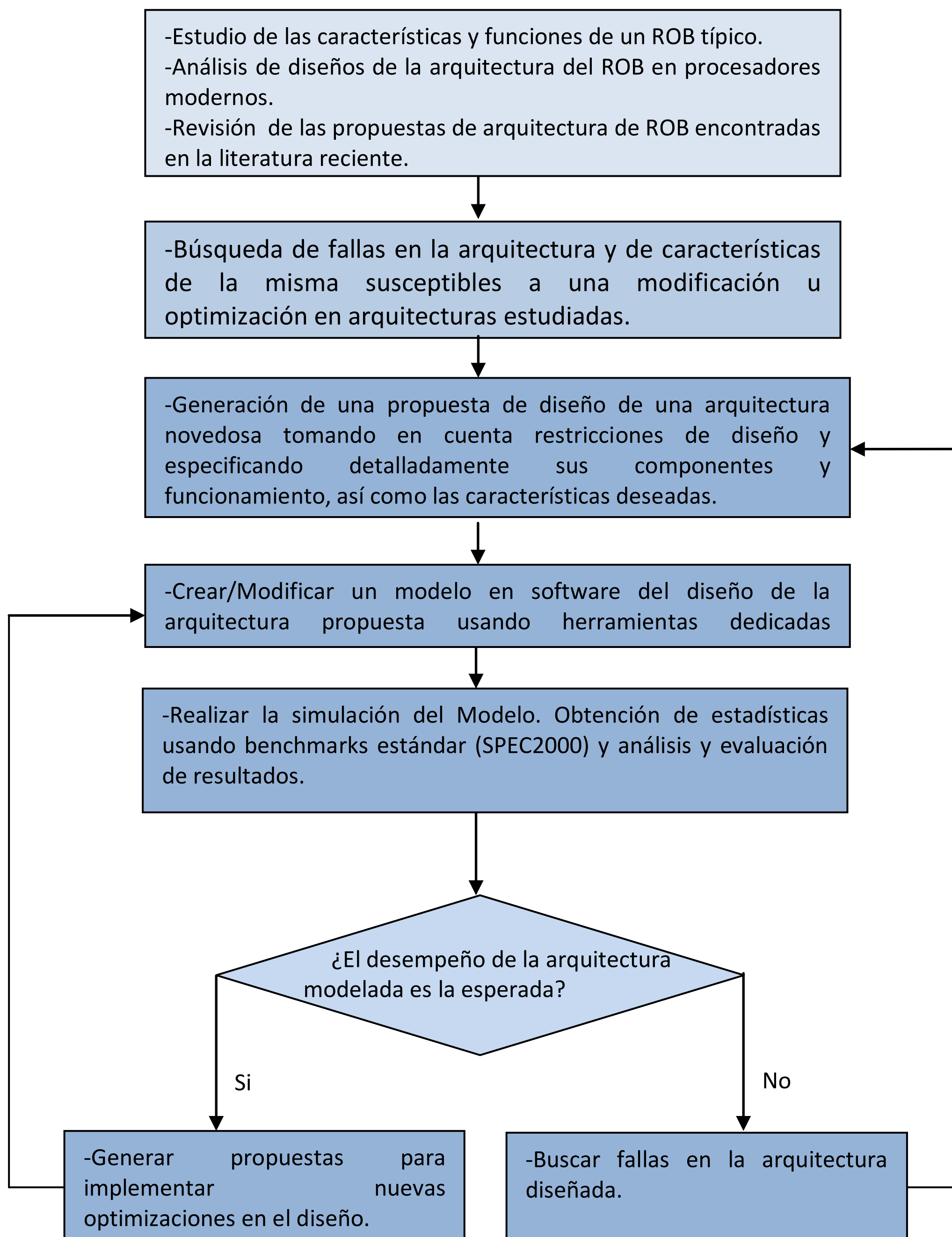


Figura 4-1. Diagrama de flujo para el diseño del ROB.

4.2. Criterios de diseño

Como se ha mostrado en la sección 2.4, las propuestas más recientes para el diseño del ROB de un procesador, se realizan mediante un análisis previo de las características de las arquitecturas actuales que no están siendo empleadas correctamente, de los recursos que no sean completamente aprovechados o de los mecanismos que puedan ser optimizados para mejorar el desempeño de la microarquitectura en su conjunto.

Para el caso particular de este trabajo de Tesis, el análisis y la búsqueda de mejoras en la arquitectura del ROB se enfoca en dos características principales, estas son, el aprovechamiento de los recursos del ROB y la optimización de los mecanismos del ROB para la recuperación del estado del procesador, la justificación de este par de criterios se presenta a continuación.

4.2.1. Análisis de las Instrucciones de Control de Flujo

La primera aproximación que se realiza para iniciar la propuesta de diseño, es considerar la implementacion típica del ROB. Recordando lo explicado en la sección 3.1, esta estructura FIFO cuenta con W entradas, cada una de ellas está constituida por varios campos, los cuales almacenan información tal como bits de bandera, contador de programa, etcétera. Estos datos son necesarios para conocer el estado de cada una de las instrucciones que conforman la Ventana de Instrucciones del procesador.

Se puede observar que en los diseños existentes del ROB, la información que se almacena en cada entrada es la misma para todos los tipos de instrucción, sin embargo, es posible notar que para algunos tipos de instrucciones, no todos los datos que se almacenan son necesariamente utilizados. Esto es particularmente cierto para el caso del PC cuyo valor solo es utilizado cuando corresponde a una instrucción de tipo salto condicional y sin embargo es almacenado para todos los tipos de instrucciones lo cual resulta en un uso inadecuado de recursos.

La instrucción de tipo salto condicional, existente en todas los *ISA*, es una de las que aparecen con mayor frecuencia, como lo muestra la información de la tabla 4-1, donde se presenta un listado con las instrucciones más frecuentemente ejecutadas por la arquitectura 80x86, para el conjunto de benchmarks SPEC CPU2000 de enteros. Como se puede notar, una de las instrucciones más frecuentemente ejecutadas es la de salto condicional (*conditional branch*), la cual aparece aproximadamente en el 20% de las instrucciones que conforman este conjunto de programas de prueba, pudiendo llegar a representar hasta una de cada cuatro instrucciones en un programa [1].

Cabe señalar que, esta estadística incluye solo el tipo de instrucción de salto condicional, excluyendo otros tipos de instrucciones de la misma categoría como las instrucciones de salto incondicional o las llamadas a procedimientos, los cuales, como lo muestra la gráfica de la figura 4-2 [1], ocurren con menor frecuencia, mientras que las instrucciones de salto condicional representan una mayor proporción.

Por claridad se realiza una discriminación de instrucciones en dos tipos, las de salto condicional, que pueden denominarse CBI (Conditional Branch Instruction) del resto de las instrucciones que pueden considerarse NCBI (Not Conditional Branch Instruction).

Como lo muestra la tabla 4-1 [1], almacenar el valor del PC que no es utilizado en lo absoluto, de las instrucciones NCBI que representan al menos un 80% del total de instrucciones del conjunto de programas de prueba ya mencionado, resulta en un obvio desaprovechamiento de recursos empleados asignar área física dedicada al almacenamiento de estos datos.

Tabla 4-1. Tipos de instrucciones más frecuentemente ejecutadas.

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

Por otro lado, dada la gran cantidad de actividad que se efectúa alrededor del ROB en un esquema monolítico, llega a disipar una gran cantidad de energía del total generado por el procesador pudiendo llegar a representar hasta un 26% en la arquitectura Pentium III, lo cual causa un fenómeno indeseable conocido como "*hotspot*", en donde una región o área del procesador genera una gran cantidad de calor, imposible de compensar por los métodos de enfriamiento del sistema y puede causar un fallo total del mismo, por ello se ha optado por esquemas distribuidos que han mostrado reducir significativamente este inconveniente [23][24].

De esta manera, se presenta una clara oportunidad para reducir los recursos empleados típicamente para la implementación del ROB reduciendo el tamaño de la estructura dedicada al almacenamiento del valor del PC, sin embargo, esta deducción aún debe ser evaluada para determinar su viabilidad en una eventual implementación en una propuesta de diseño.

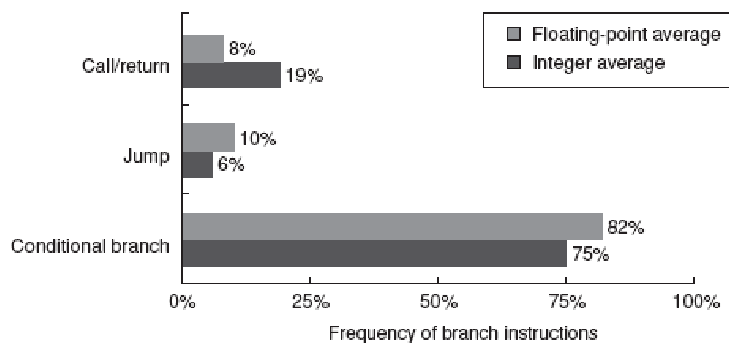


Figura 4-2. Frecuencia de aparición de instrucciones de salto condicional.

4.2.2. Esquemas de recuperación del estado del procesador

Como una segunda aproximación hacia una propuesta de diseño, se estudian los mecanismos que implementa el ROB para permitir la recuperación del estado preciso del procesador. En particular, de la forma en que los mecanismos de recuperación son iniciados por el ROB en caso de que ocurran eventos tales como errores en la predicción de la dirección en una instrucción de salto condicional.

Existen dos esquemas diferentes en el ROB bajo los cuales el proceso de recuperación del estado del procesador puede iniciar los mecanismos de restauración del estado del procesador [11]. En el primer esquema, cuando se presenta un error de predicción de salto, es decir, cuando al realizar la comparación del valor del PC predicho por el predictor de saltos, con el valor del PC calculado y se encuentra que estos valores son diferentes, se deberá esperar a que la instrucción que causó este error alcance la cabeza del ROB para que dicho error sea manejado, es decir, para que los mecanismos de restauración del estado del procesador sean lanzados.

Este primer esquema es el más simple y es implementado típicamente en la arquitectura del ROB, sin embargo, es poco eficiente ya que para manejar el error de predicción de salto detectado en un ciclo determinado, se debe esperar a que la instrucción que lo genera llegue a la cabeza de la FIFO, como se ilustra en la figura 4-3. Esto implica que las instrucciones que fueron emitidas posteriormente a esta instrucción de salto condicional, a las cuales se les ha asignado una entrada del ROB, continuarán ejecutándose de manera especulativa hasta que el error sea manejado. Más aún, aunque se haya detectado el error, las instrucciones continuarán

despachándose, añadiéndose a la cola de la FIFO y ejecutándose en una traza errónea hasta que el error sea manejado, lo que causa un ineficiente uso de los recursos, pues todas estas instrucciones que son ejecutadas en una traza errónea deberán ser eliminadas lo que significa que el motor de emisión a ejecución de instrucciones está funcionando inútilmente.

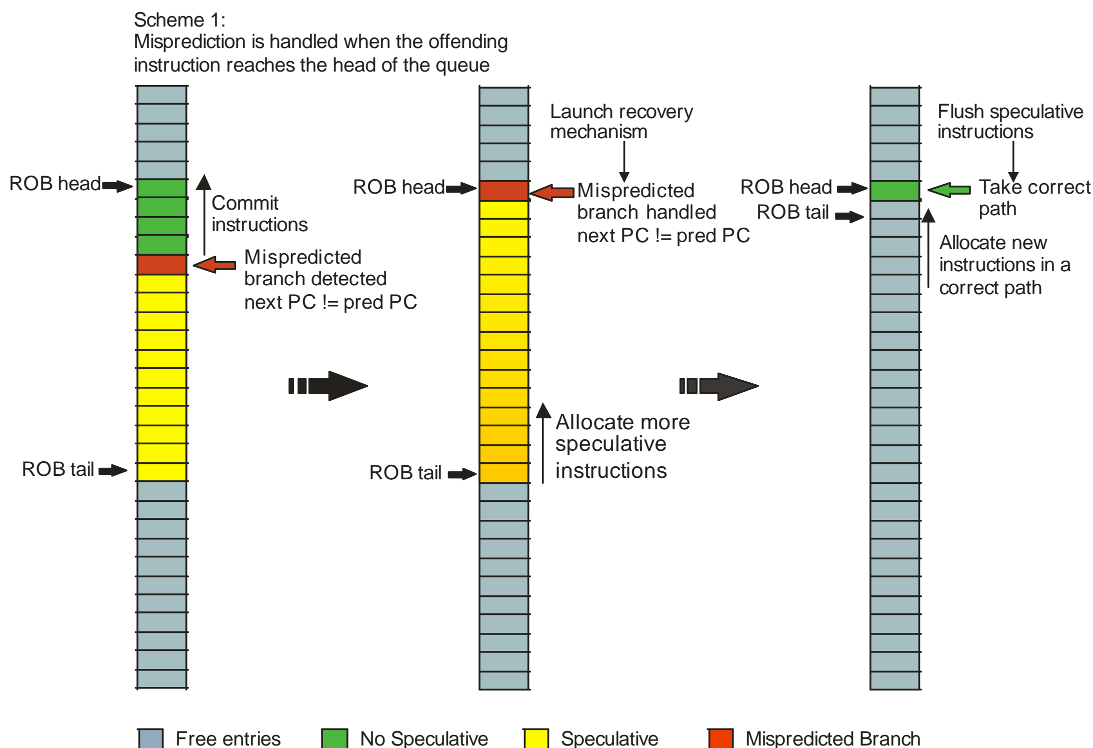


Figura 4-3. Esquema típico del ROB para recuperación de errores de salto.

Un segundo esquema, más eficiente que el anterior, implementa una optimización en la cual se permite que el error de predicción de salto sea tratado de manera inmediata a su detección. Como se muestra en el diagrama de la figura 4-4, en lugar de esperar a que la instrucción que genera el error alcance la cabeza de la FIFO, los mecanismos de recuperación son lanzados una vez que se resuelve que este error ha ocurrido. Esta recuperación inmediata evita que se sigan emitiendo y ejecutando instrucciones en una traza errónea.

Este método optimizado asegura que una vez que un error de predicción de salto sea detectado, no se emitirán ni se ejecutarán más instrucciones que seguirán una traza equivocada, permitiendo al motor de ejecución de instrucciones ejecutar instrucciones útiles. Aunque las instrucciones que se emitieron después de la instrucción de salto causante del error y antes de la detección de este deberán ser desechadas inevitablemente.

En teoría, la forma de operar de estos dos esquemas sugiere que para obtener una arquitectura del ROB más eficiente, se deberá optar por el segundo de éstos. Sin embargo, ambos deben ser modelados, evaluados y comparados para obtener un análisis completo que sustente la propuesta final.

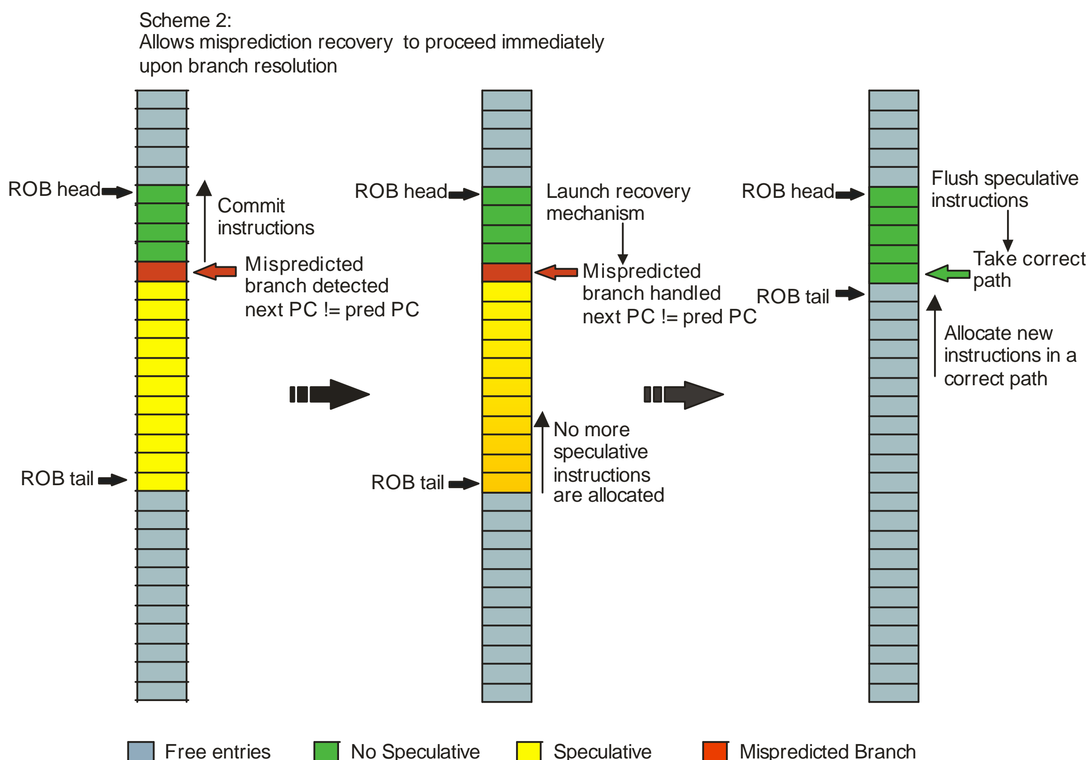


Figura 4-4. Esquema optimizado del ROB para recuperación de errores de salto.

4.3. Arquitectura del diseño propuesto

Una vez que se ha realizado un estudio de diferentes arquitecturas implementadas por varios procesadores actuales así como de las propuestas más recientes encontradas en la literatura para la implementación del ROB, y tomando en consideración las posibles optimizaciones que se ha observado, pueden implementarse en su arquitectura para impactar positivamente en su desempeño, se procede a elaborar una propuesta nueva para esta estructura funcional.

En primer lugar, se plantea el diseño del ROB no como una estructura centralizada o monolítica como típicamente se implementa, sino como un conjunto de subestructuras que operen de manera distribuida como una unidad para realizar las tareas de las que está encargada. De esta forma, se tendrá una subestructura, llamada *dispatch flag,* encargada de indicar mediante la activación de un bit de bandera, cuando una instrucción ha sido despachada hacia alguna de las colas de

instrucciones. De manera semejante, existe otra subestructura, *issue flag,* que activará un bit de bandera en su entrada correspondiente, cuando la instrucción tenga disponibles sus operandos y sea emitida para ser ejecutada por alguna de las unidades funcionales.

Cuando la unidad funcional termine de ejecutar la instrucción, se activará un bit de bandera más en la entrada correspondiente que pertenece a una tercera subestructura, *execute flag*. Como se puede ver en el diagrama de bloques de la figura 4-5, estas subestructuras tipo FIFO circular de un bit tienen un número idéntico de entradas y comparten los mismos apuntadores de cabeza y cola. Donde el apuntador de cabeza se incrementa cuando una instrucción es retirada de la FIFO y de manera parecida, el apuntador de cola se incrementa cada vez que es insertada una nueva instrucción.

Al concebir estas subestructuras como componentes individuales que operan de forma concurrente, se hará posible colocarlas en una configuración distribuida en diferentes secciones del área del layout del procesador, de acuerdo a la relación que cada una de ellas tenga con alguna etapa en particular, por ejemplo, situando la subestructura de *execute flag* cerca de las unidades de ejecución.

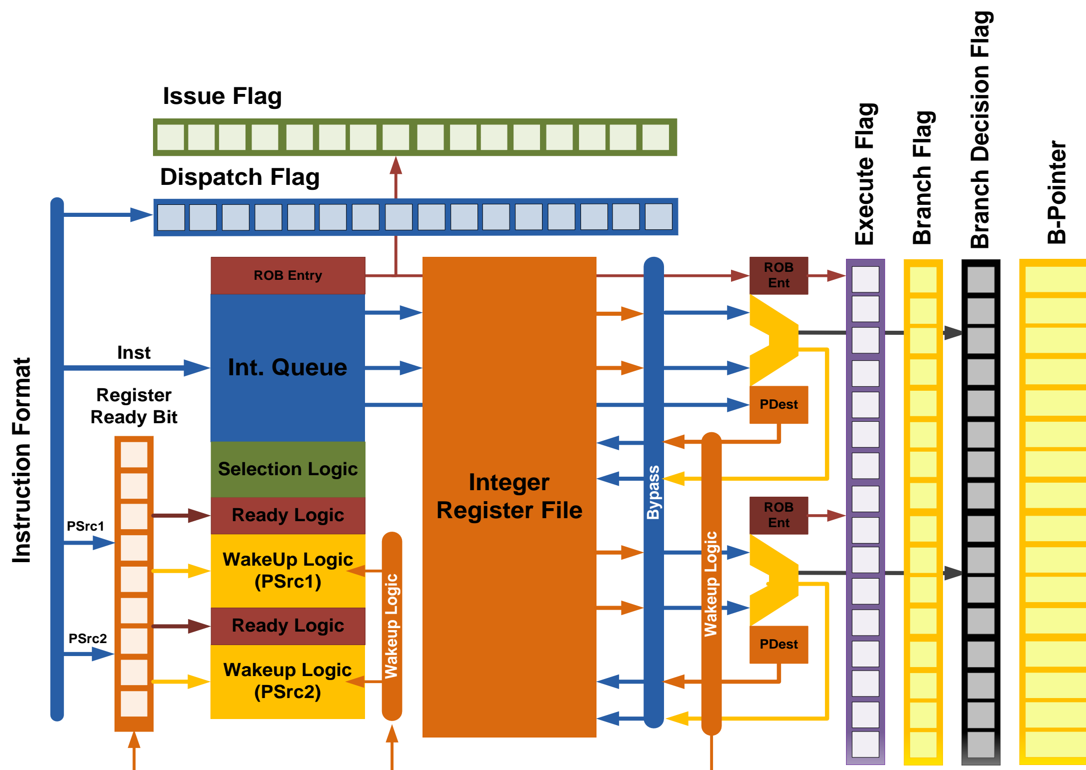


Figura 4-5. Diagrama de bloques de la arquitectura del diseño propuesto.

Esta configuración representa además la ventaja de evitar la concentración de la disipación de energía en una área determinada del layout (fenómeno conocido

como "*hot spot"*) ocasionada por la gran cantidad de accesos de lectura/escritura que involucra su funcionamiento, lo cual sí ocurre en una configuración centralizada o monolítica, generando un *"hot spot"* que puede llegar a representar hasta un 26% de la energía disipada por el procesador [26]. De esta manera se logra mantener el estado de las instrucciones que pasan a través de las diferentes etapas del pipeline mediante un esquema que, a la vez, tiene el potencial de disminuir la generación de "*hot spots"*.

Hasta ahora la parte descrita de la arquitectura propuesta solo resuelve las tareas de actualizar los estados de dispatched, issued y executed de las instrucciones que conforman la Ventana de Instrucciones, sin embargo, el resto de las funciones que el ROB debe desempeñar, es decir, permitir la ejecución especulativa de instrucciones, el reclamo de registros físicos y el retiro de instrucciones aún debe atacarse. Esto se explica en las siguientes secciones.

4.3.1. Ejecución especulativa de instrucciones

Una de las tareas más importantes de las que se encarga el ROB, consiste en trabajar en conjunto con la unidad de predicción de saltos para darle al procesador la capacidad de ejecutar instrucciones en forma especulativa. En esta propuesta de diseño, la arquitectura introduce una subestructura llamada Branch-ROB, como se muestra en el diagrama de bloques de la figura 4-6. Esta subestructura es utilizada para almacenar el valor del PC de las instrucciones de salto condicional junto con la dirección de destino, la cual es calculada por la unidad de fetch mediante una ACU usando el valor del PC actual y el offset de la instrucción de salto.

Cuando se presenta una instrucción de salto condicional, se activa un bit de bandera en la entrada correspondiente en una estructura nombrada *branch flag*, la cual es idéntica a las descritas para actualizar los estados de dispatch, issue y execute. Entonces la unidad de predicción de saltos elige una traza (taken/not taken) en función del esquema de predicción con que se implementa, la cual se indica mediante la activación de otro bit de bandera en una entrada de una estructura más, llamada *branch decisión flag*, y la ejecución de instrucciones continua en modo especulativo, pues en este punto no se tiene certeza del resultado de la predicción realizada por el predictor de saltos.

Para volver a un modo no especulativo y validar las instrucciones ejecutadas especulativamente, es necesario conocer el resultado de la predicción, lo cual ocurrirá varios ciclos después mediante una comparación entre el valor del próximo PC (next PC) calculado y el valor de PC dado por el predictor de saltos (pred PC). Cuando el valor del próximo PC es igual al del PC predicho (next PC == pred PC), se valida que la predicción fue correcta, entonces se activa un bit de bandera en la entrada correspondiente de otra subestructura llamada *non speculative*, que

indicará que la instrucción de salto, así como las subsecuentes pueden ir a un estado no especulativo. Sin embargo, si al realizar la comparación se encuentra que el valor del próximo PC no fue igual al valor del PC predicho (next PC != pred PC) se habrá presentado un error en la predicción del salto (mispredicted branch), siendo entonces necesario lanzar los mecanismos de recuperación correspondientes, para volver a la dirección de la instrucción que generó el error y a partir de este punto reiniciar la ejecucion sobre el camino correcto.

Cabe señalar que la estructura Branch-ROB, es direccionada mediante un apuntador llamado B-ROB pointer, el cual es usado para leer el valor del PC y de la dirección de destino para actualizar la bandera de *non speculative*, cuando se cumple la condición de que ambos bits de bandera, *execute flag* y *branch flag*, estén activos.

Por otra parte, se toma en cuenta lo observado en la sección 4.2 donde se describe que en una secuencia de instrucciones, la razón de las instrucciones de salto condicional con respecto a todos los demás tipos es de aproximadamente el 25%, es decir, que dentro de cuatro instrucciones consecutivas, al menos una de ellas es una instrucción de salto condicional. Lo anterior implica que el tamaño de la subestructura BROB no necesita ser del mismo tamaño que el resto de las subestructuras ya descritas, pudiendo ser su tamaño hasta de un 25% del tamaño del ROB.

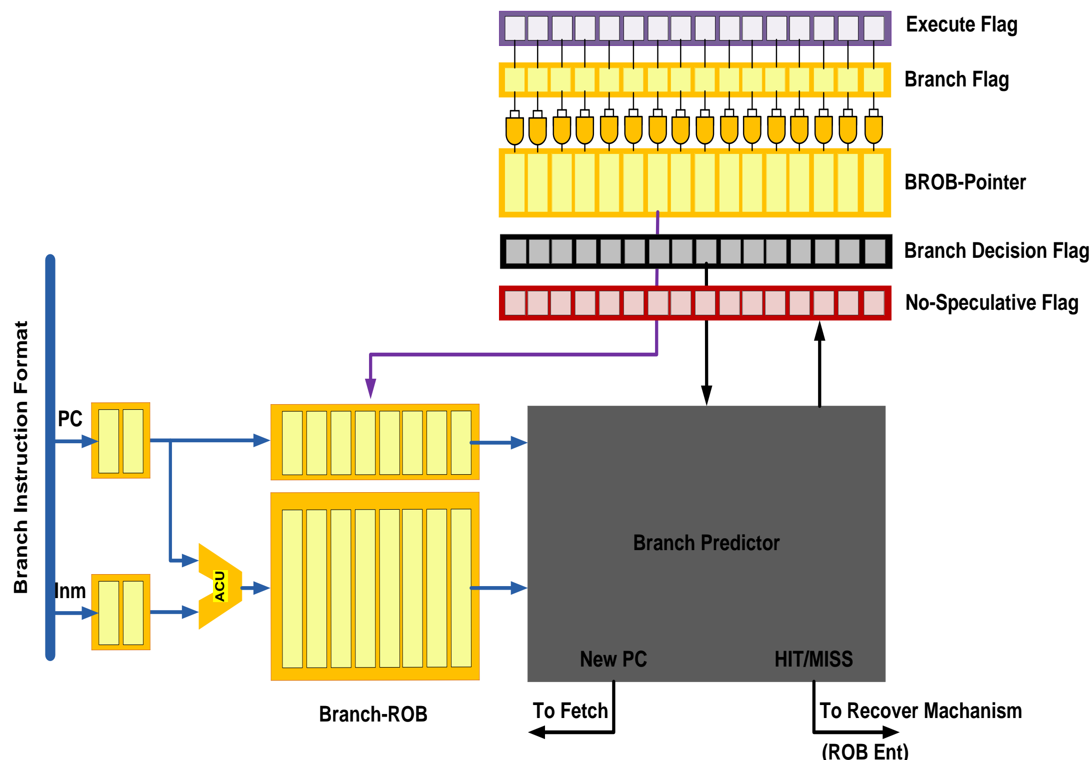


Figura 4-6. Mecanismo para habilitar la ejecución especulativa.

4.3.2. Reciclado de registros físicos

El reciclado de registros físicos es otra de las tareas realizadas por el ROB. Este proceso se realiza cuando el valor almacenado en un registro lógico ya no es útil para la ejecucion de las instrucciones subsecuentes y entonces el registro físico que tiene asociado, al ya no ser de utilidad debe ser reciclado. Cada instrucción renombrada tiene asociado un registro físico de destino actual y un registro de destino viejo y al ser insertada dicha instrucción en una entrada del ROB, ambos registros siguen ligados a ella mediante un campo dedicado en el ROB.

Como se muestra en el diagrama de bloques de la figura 4-7, la subestructura que almacena la etiqueta del registro de destino viejo trabaja junto con la lista de registros libres para que en la etapa de commit estas etiquetas sean liberadas y se devuelvan a la lista de registros físicos libres.

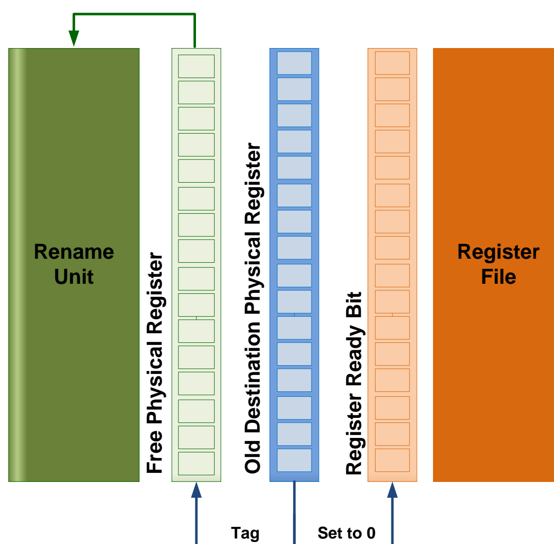


Figura 4-7. Reciclado de registros y actualización de valores en el RF.

4.3.3. Retiro grupal de instrucciones

Anteriormente se ha mencionado que para realizar el retiro de las instrucciones alojadas en las diferentes entradas del ROB, es necesario que cumplan con ciertas condiciones. Dichas condiciones exigen que las instrucciones más viejas, es decir, las que se encuentran en la cabeza de la FIFO, deben ser las primeras en ser retiradas, lo que implica forzosamente que todas las instrucciones previas ya hayan sido retiradas también.

Esto asegura la coherencia en la semántica del programa que se está ejecutando, al realizarse un retiro en estricto orden de programa de las instrucciones, aunque su ejecución se haya realizado fuera de orden.

Una vez que el apuntador de cabeza de la FIFO apunta a las instrucciones más viejas, es necesario además que cumplan con otras condiciones. Éstas exigen que los bits de bandera *dispatched flag*, *issued flag*, *executed flag* y *non speculative* de la instrucción correspondiente estén todos activados.

Para el diseño de la arquitectura propuesta en este trabajo de tesis, se implementa un retiro en grupo de instrucciones. Como se muestra en el diagrama de la figura 4-8, cuando cuatro instrucciones consecutivas se encuentran a la cabeza de la FIFO y cada una de ellas ha sido despachada, emitida, ejecutada y no se encuentra en un estado especulativo, estas instrucciones son retiradas, se actualizan los registros arquitecturales y el apuntador de cabeza se incrementa paralelamente para todas estas subestructuras de estado.

Éste apuntador de cabeza se constituye como un registro de siete bits, dos de los cuales son usados como offset para direccionar la localidad en caso de que la instrucción que causo el error de predicción de salto no sea la del inicio del grupo y los cinco bits restantes para direccionar a cada grupo.

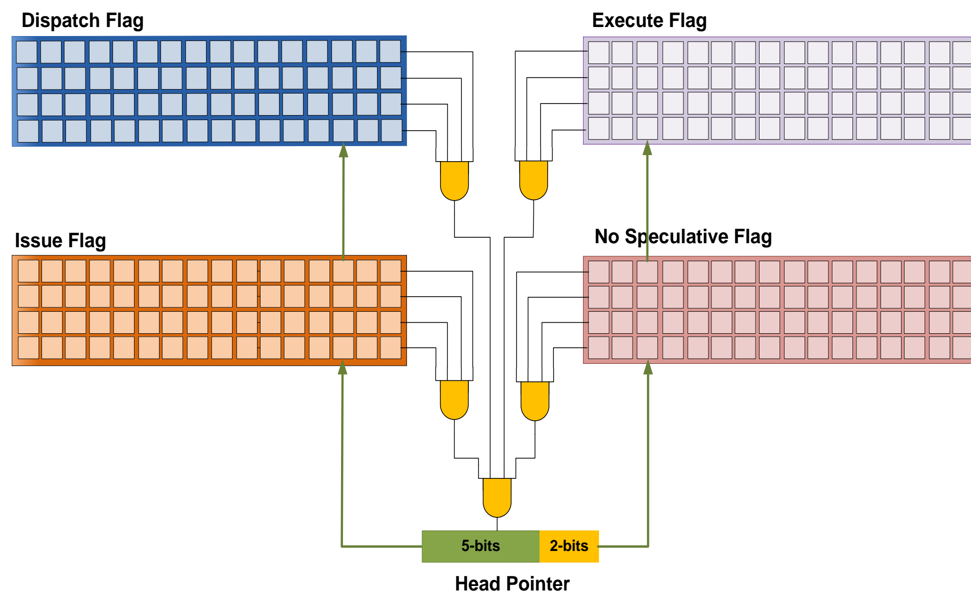


Figura 4-8. Retiro grupal de instrucciones.

4.4. Modelado y Simulación del diseño

En la sección 3.2 de esta tesis se explicó que SimpleScalar es un conjunto de simuladores con diferentes características para evaluar, a diferentes niveles de detalle, la microarquitectura de un procesador. En particular, se destacó el simulador llamado *sim-outorder*, el cual modela en forma detallada la

microarquitectura de un procesador superescalar con ejecución fuera de orden. La configuración de los parámetros de esta microarquitectura pueden variarse y el código fuente puede modificarse para modelar nuevos diseños.

Como también se ha mencionado, sim-outorder implementa una estructura llamada Register Update Unit, la cual cumple las funciones del ROB dentro de una microarquitectura superescalar. Dicha estructura es el objeto de estudio principal de este trabajo. Esta arquitectura implementada por SimpleScalar es modificada para modelar la arquitectura propuesta en este trabajo, la cual ha sido explicada en la sección 4.3.

4.4.1. Modificación del código de SimpleScalar

La figura 4-9 muestra un diagrama de bloques que ilustra la forma en que el simulador sim-outorder está estructurado. Como se puede ver, cuenta con una serie de bloques que emulan el pipeline de un procesador mediante diversas funciones, de esta manera, una vez que una instrucción es despachada, se agrega una entrada a una estructura, la cual, dentro de la nomenclatura de la arquitectura del simulador *SimpleScalar*, es llamada *Register Update Unit* y para efectos de análisis es equivalente al *ReOrder Buffer*.

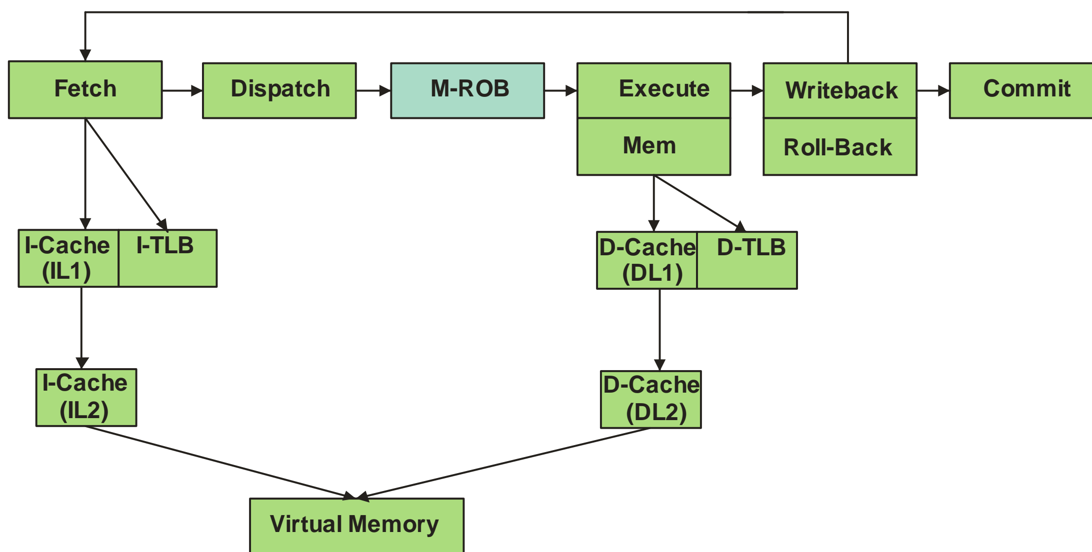


Figura 4-9. Sim-outorder con un ROB monolítico.

La forma en que esta estructura funcional está modelada, es siendo declarada como una variable tipo estructura, la cual tiene una serie de campos que conforman cada entrada del RUU y que incluyen variables encargadas de mantener el estado del procesador.

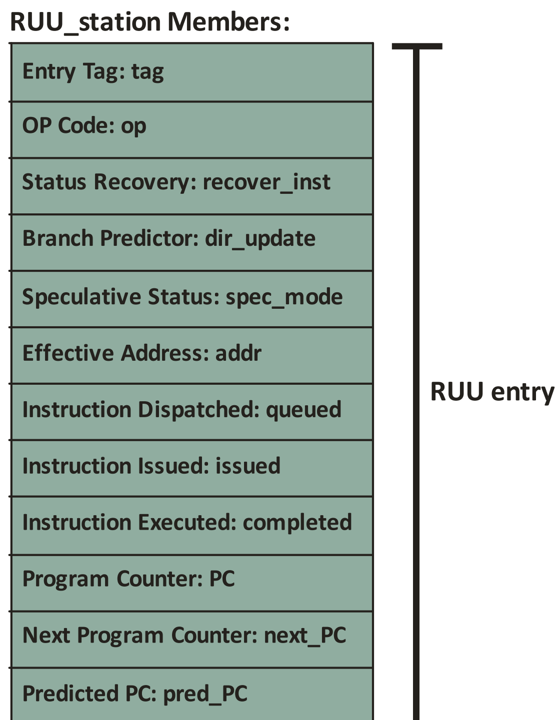


Figura 4-10. Miembros de la estructura *RUU_station*.

Los diferentes campos de la estructura almacenan datos correspondientes a cada instrucción de la Ventana de Instrucciones. Estos campos se muestran en el diagrama de la figura 4-10. Como se puede ver, existe por ejemplo, un campo llamado *tag* que almacena un valor que funciona como una etiqueta para identificar cada instrucción agregada a la cola de la FIFO. Cada entrada contiene también un campo para mantener su código de operación. Sin embargo, los campos a destacar son los que mantienen el estado de las instrucciones, los cuales son *queued, issued*, *completed y spec_mode*.

Otro grupo de campos importantes son los encargados de almacenar el valor de contador de programa, *PC*, el valor del próximo contador de programa, *next_PC*, y del valor del contador de programa predicho, *pred_PC*. También se encuentra declarada otra variable de importancia llamada *recover_inst.*

Se ha mencionado que cada entrada es asignada a la cola de la FIFO cada vez que se despacha una instrucción. De una manera similar, las instrucciones que se encuentran a la cabeza de la FIFO son retiradas si sus banderas de estado asociadas en su respectiva entrada del RUU cumplen con las condiciones necesarias, como se muestra en el diagrama de la figura 4-11. Estas condiciones, como se ha explicado previamente, exigen que sus estados correspondientes tengan valores de DISPATCHED=1, ISSUED=1, EXECUTED=1 Y NO SPECULATIVE=0, dichas condiciones son evaluadas en la etapa de *commit*. Debido a las diversas tareas de las que se

encarga la RUU, aparece constantemente dentro de las diferentes funciones que modelan los bloques que componen el pipeline del procesador, ya que constantemente se realizan actualizaciones que se escriben y se leen en el RUU. Por esta razón realizar cambios a esta estructura implica realizar también una modificación extensa del código de las funciones del simulador en donde esta estructura se ve involucrada.

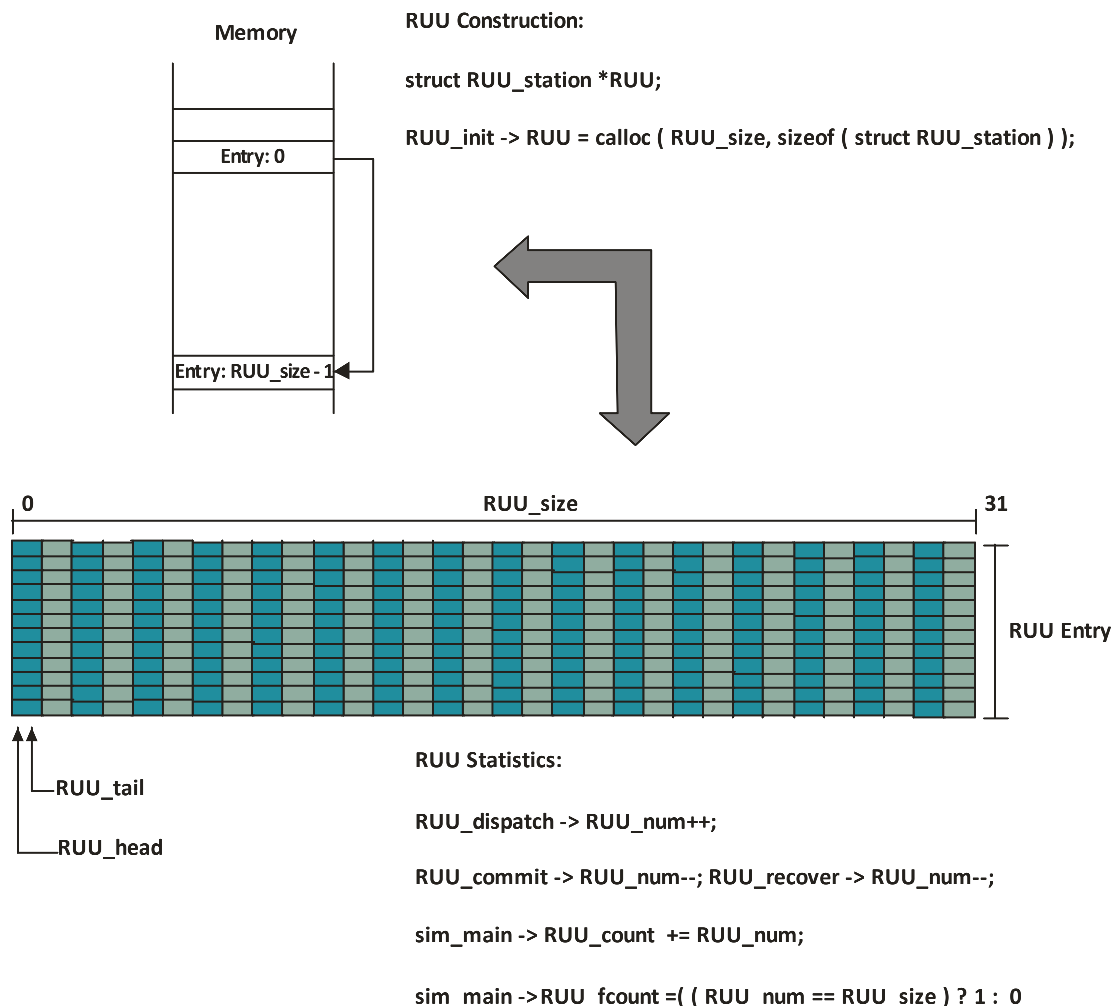


Figura 4-11. Diagrama que ilustra la construcción de la estructura RUU.

4.4.2. Modelo de la arquitectura propuesta

Previamente se ha explicado que, al menos una de cada cuatro instrucciones consecutivas de un programa es una instrucción de salto condicional. Por esta razón la arquitectura propuesta en esta tesis implementa una estructura, llamada Branch-

ROB que almacena el valor del PC de las instrucciones de salto condicional únicamente, la cual es direccionada por un apuntador, B-ROB pointer.

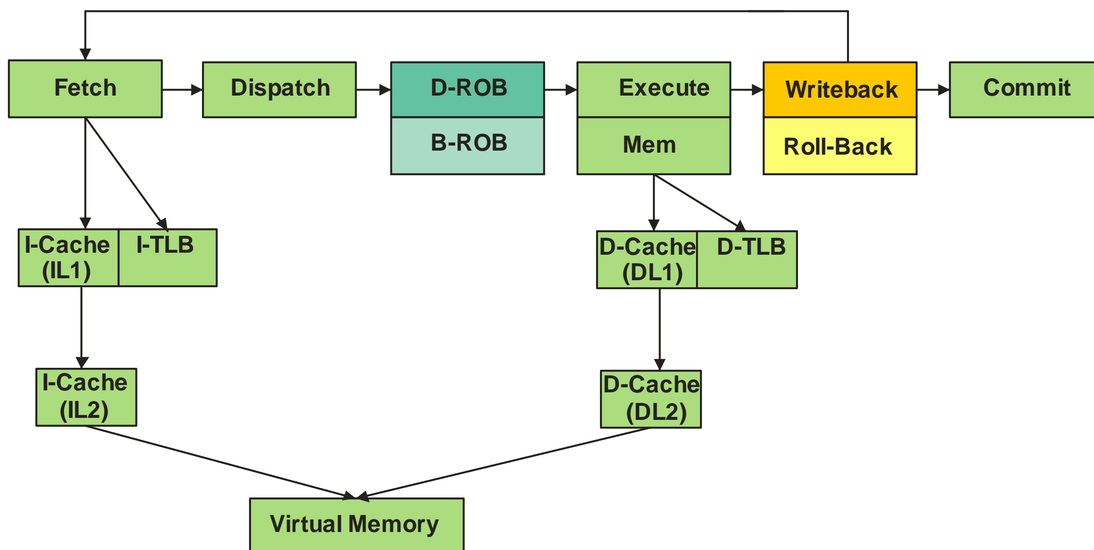


Figura 4-12. Sim-outorder con un ROB distribuido.

El tamaño de esta estructura puede reducirse a hasta un cuarto del tamaño del resto de las estructuras dedicadas a mantener los valores de las banderas de estado de las instrucciones. Como se muestra en el diagrama de bloques de la figura 4-12, se deberá modelar una estructura más pequeña que almacenará el valor del PC de las instrucciones de salto condicional únicamente, y en donde también las estructuras de salto se verán de manera independiente, pero funcionarán con los mismos apuntadores de cabeza y cola. Esta arquitectura no está considerada en la estructura del simulador, por esta razón, el código es modificado para incluir estas características. Como primer paso es necesario declarar una variable adicional a la que se le da en nombre de B-RUU_size y que determina el número de entradas para el B-RUU.

```
/* register update unit options */
opt_reg_int(odb, "-ruu:size",
            "register update unit
            &RUU_size, /* default */32,
            /* print */TRUE, /* format */NULL);

/* branch register updarte unit options */
  opt_reg_int(odb, "-bruu:size",
            "branch register update unit (BRUU) size",
            &BRUU_size, /* default */8,
            /* print */TRUE, /* format */NULL);
```

Figura 4-13. Código para añadir la opción *-bruu:size* en sim-outorder.c.

Como se muestra en el segmento de código mostrado en la figura 4-13, esta opción debe darse de alta para poder realizar posteriormente una variación del tamaño del RUU que contiene el estado de las instrucciones RUU y de la estructura de menor tamaño B-RUU que contiene el valor del PC de las instrucciones de salto. Esto es necesario para poder realizar pruebas mediante la evaluación de diferentes combinaciones de estos dos parámetros.

La figura 4-14 muestra un diagrama de la manera en que el RUU original se divide en dos subestructuras. Como se ve, cada entrada de la estructura RUU conserva todos los campos menos los correspondientes a los del PC y de la dirección efectiva, más un campo con un apuntador hacia una segunda estructura B-RUU que contiene campos para almacenar los valores correspondientes al PC.

Durante la ejecución del simulador, los parámetros *RUU_size* y *B-RUU_size* se utilizan para construir las FIFOS RUU y B-RUU mediante una asignación de espacio de memoria correspondiente al tamaño de cada uno de esos parámetros, los cuales indican el número de entradas que constituyen a cada una de estas subestructuras funcionales.

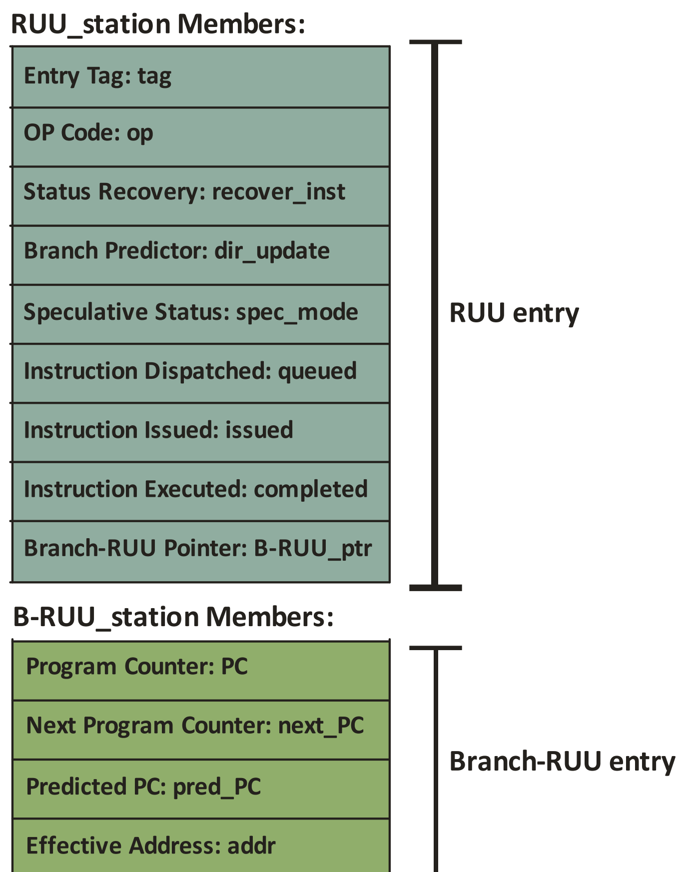


Figura 4-14. Contenido de una entrada del RUU distribuido.

Como se ve en el diagrama de la figura 4-15, durante la ejecución de la rutina *ruu_dispatch* todas las instrucciones son insertadas al final de la cola de la RUU y el valor del apuntador de cola, *RUU_tail*, es incrementado. Pero para el caso de la B-RUU, sólo el valor del PC de las instrucciones de salto es insertado al final de la cola y su apuntador de cola, *B-RUU_tail*, es incrementado. De una manera similar, las entradas correspondientes a las instrucciones más viejas los apuntadores de cabeza tanto de la RUU como de la B-RUU, son incrementados cuando llegan a la etapa de commit.

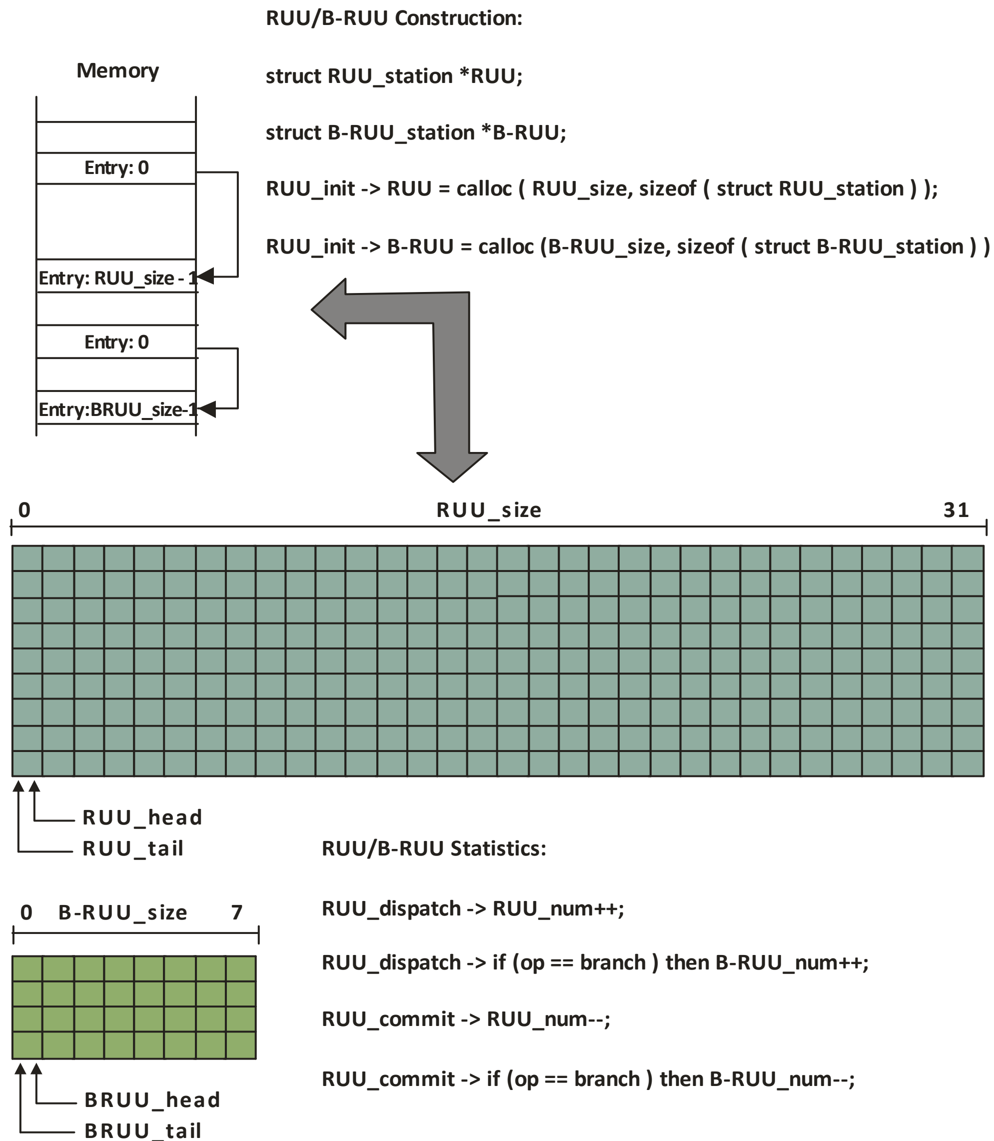


Figura 4-15. Construcción de FIFOs RUU y B-RUU.

También es importante indicar que para realizar estadísticas de ocupación de estas estructuras es necesario declarar variables adicionales como es el caso de *B-*

RUU_num, la cual es incrementada cada vez que se inserta una instrucción y se decrementa cuando una instrucción es retirada en la etapa de commit. Existen también mecanismos para preservar el correcto funcionamiento de las colas.

Dentro del ciclo principal, *sim_main,* existe código de verificación del correcto modelado de las estructuras del procesador, mediante condiciones que aseguran por ejemplo que el apuntador de cabeza de la RUU no debe traslaparse con su apuntador de cola y lo mismo ocurre para el B-RUU.

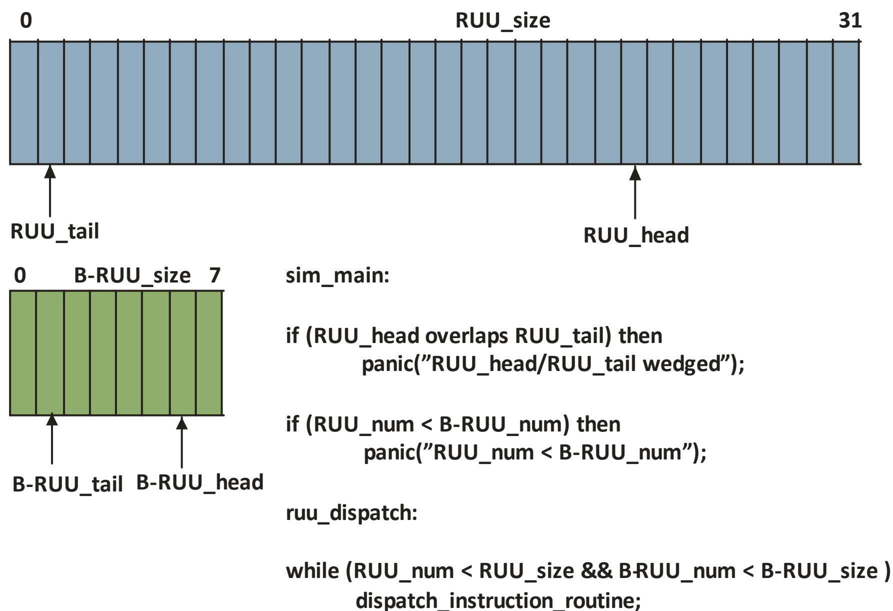


Figura 4-16. Condiciones para el funcionamiento de las FIFOS RUU y B-RUU.

También se evita la condición en donde el número de instrucciones activas dentro del RUU sea menor que el número de instrucciones activas dentro del B-RUU. Finalmente, existe una condición importante dentro del código correspondiente a la etapa de *dispatch* del pipeline. Como se ve en el diagrama de la figura 4-16, esta condición exige que para emitir una instrucción hacia las colas de instrucciones, el número de entradas ocupadas dentro del RUU sea menor al número total de entradas de que dispone y que lo mismo suceda para la estructura B-RUU. Es decir, se verifica en cada ciclo que, ni la FIFO RUU ni la B-RUU se han desbordado.

Por otra parte cada vez que estas estructuras se llenan, el mecanismo de emisión de instrucciones se detiene, lo que causa una condición llamada *stall* del procesador. Dicha condición impacta directamente en desempeño del procesador ya que lo que se busca es asegurar un flujo continuo de instrucciones a lo largo del pipeline.

Siendo el B-RUU más pequeño y aunque solo almacene saltos condicionales es posible que se llene con mayor frecuencia en aquellos programas que estresan al predictor de saltos por utilizar muchos saltos condicionales, por lo que diferentes

simulaciones de diferentes combinaciones deben realizarse para evaluar el impacto al desempeño del procesador con esta arquitectura.

4.4.3. Optimización del proceso de recuperación

Una modificación adicional que se incorpora a la arquitectura del simulador sim-outorder, es la implementación de una recuperación inmediata del estado del procesador. A diferencia de la arquitectura previa en donde el mecanismo de recuperación del estado del procesador o *roll-back*, es lanzado hasta que las instrucciones alcanzan la cabeza del ROB, se realiza una modificación para que dicho mecanismo de restauración sea lanzado inmediatamente después de que el error de salto es detectado.

Para el caso de este modelo de simulación, la restauración del estado del procesador se realiza mediante un mecanismo constituido por una rutina que lanza tres funciones. Estas son *RUU_recover*, *tracer_recover* y *bpred_recover.* En el caso de esta propuesta, se realizarán pruebas con ambos esquemas, el normal donde se espera a que la instrucción causante del error llegue a la cabeza para ser manejada y el esquema optimizado en el cual estas rutinas son lanzadas inmediatamente después de detectarse el error de salto.

Es importante comparar en forma práctica estos dos esquemas pues en teoría, el esquema optimizado tendrá la característica de proporcionar beneficios tales como evitar que las colas ROB y BROB se llenen con instrucciones resultantes de traza de instrucciones que siguen a un error de salto, disminuyendo además la cantidad de energía gastada desperdiciada en ejecutar instrucciones que serán desechadas. Un beneficio adicional es que es posible obtener un incremento en la cantidad de instrucciones que se ejecutan por segundo, medida de desempeño conocida como *speedup*, esto debido a que se incrementa el número de instrucciones ejecutadas en trazas correctas. Sin embargo para validar estos resultados esperados, es necesario realizar una serie de simulaciones, como se explica en la siguiente sección.

4.5. Proceso de simulación de la arquitectura propuesta

El proceso para realizar la serie de simulaciones que tiene como objetivo evaluar el desempeño de las arquitecturas propuestas, consiste en primer lugar, en configurar los parámetros del simulador, los cuales en este caso se establecen para coincidir con las características del procesador MIPS R10000 las cuales fueron estudiadas en la sección 2.1. A esta configuración se le da el nombre de configuración base (*baseline)*, pues a partir de ella se realizan las modificaciones al código y será contra esta configuración base, que se compararán los resultados.

En la tabla 4-2 se presenta la configuración base que se le da al simulador. Como se ha visto, la microarquitectura del procesador MIPS R10000 implementa un ROB de 32 entradas, sin embargo, adicionalmente se ha considerado que evaluar el desempeño del procesador con tamaños de Ventana de Instrucciones mayores, en este caso de 64 y 128 entradas, permitirá obtener una visión más amplia del impacto que las modificaciones que se realizan al código del simulador tendrán sobre su desempeño. Para realizar las simulaciones, al simulador se le proporcionan una serie de programas de prueba conocidos como *benchmarks*.

Tabla 4-2. Configuración de la arquitectura base simulada.

Parámetro	Configuración
Machine Width	4-width fetch, 4-width issue, 4-width commit
Window Size	32-entry ROB/8-entry LSQ, 64-entry ROB/16-entry LSQ and 128-entry ROB/32-entry LSQ
Branch Predictor	Bimodal with 2048 table size
FUs	Two 64-bit ALU
TLB	8-entry instruction Translation Look-aside Buffer

En particular se utiliza el conjunto de *benchmarks* SPEC CPU2000 desarrollado por SPEC, una organización sin fines de lucro dedicada a crear programas de prueba que sirven como estándar en la industria para evaluar diversas arquitecturas [36][37]. En la tabla 4-3, se muestra el subconjunto de benchmarks que son usados en este trabajo de tesis. Como se puede ver, se utilizan una serie de programas con diversas aplicaciones que se dividen principalmente por su contenido de instrucciones en operaciones de enteros y de punto flotante.

Tabla 4-3. Benchmarks utilizados y sus características.

Tipo	Benchmark	Descripción
CINT2000 test Operaciones de enteros	164.gzip	Data compression utility
	176.gcc	C compiler
	181.mcf	Minimum cost network flow solver
	197.parser	Natural language processing
	255.vortex	Object-oriented database
	256.bzip2	Data compression utility
	300.twolf	Computer integrated circuit design
CFP2000 test Operaciones de punto flotante	171.swim	Meteorilogy
	172.mgrid	Multigrid solver
	173.applu	Computational fluid dynamics
	179.art	Neural networks
	183.equake	Finite element simulation
	188.ammp	Computational chemistry
	301.apsi	Weather prediction

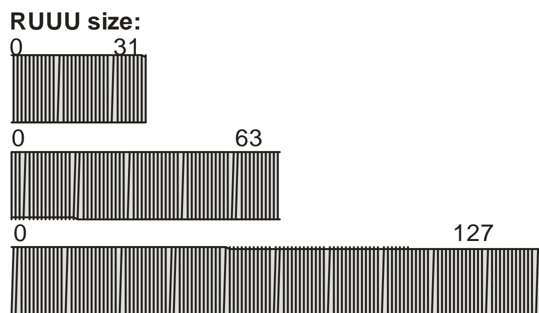


Figura 4-17. Configuraciones a simular para la arquitectura con ROB monolítico.

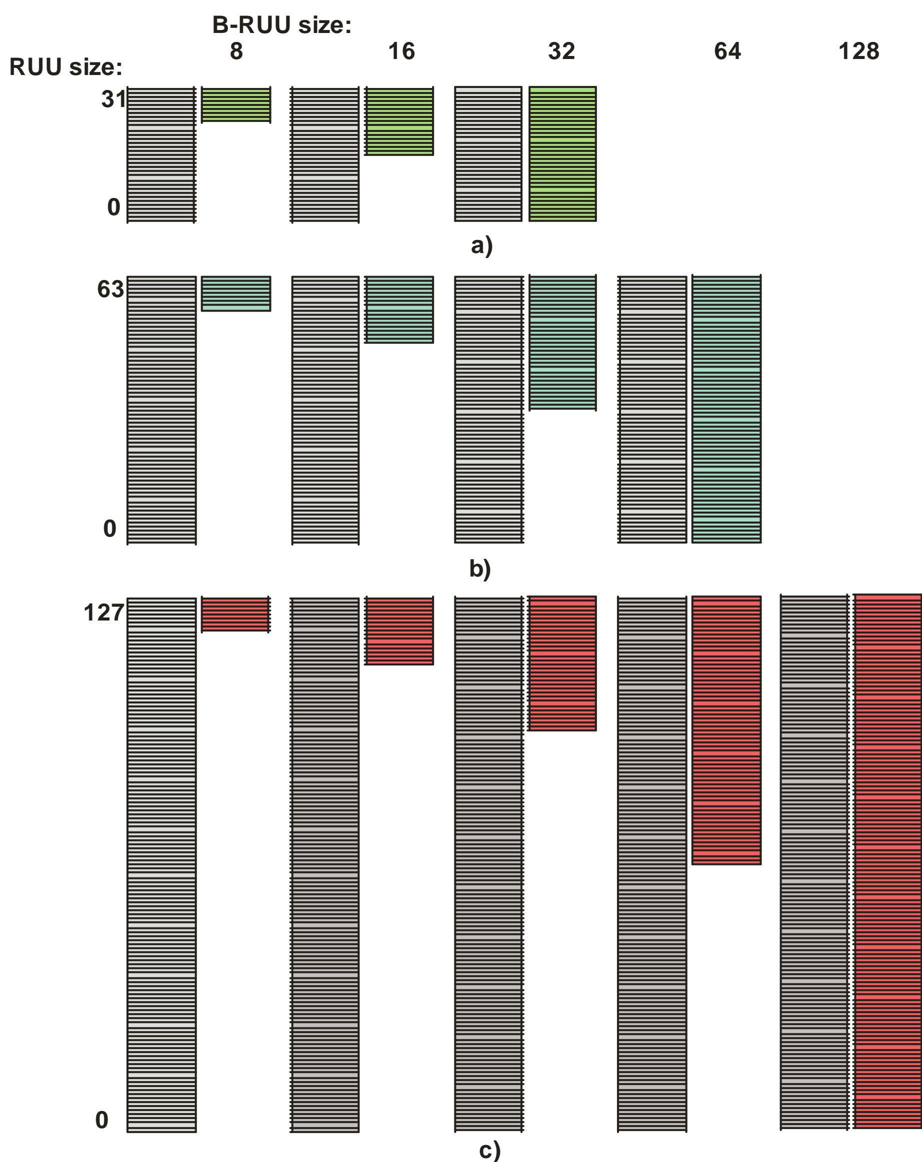


Figura 4-18. Configuraciones a simular para la arquitectura con ROB distribuido.

También es importante explicar las características de las diferentes combinaciones de RUU/B-RUU que se van a simular para cada arquitectura. En primer lugar se simula la arquitectura monolítica con tamaños para el RUU monolítico de 32, 64 y 128 entradas, como se muestra en la figura 4-17. Enseguida se realizaran simulaciones para una arquitectura distribuida de ROB de 32 entradas en combinación con entradas de 8, 16, 32 y 64 entradas de B-ROB respectivamente, como lo muestra el diagrama de la figura 4-18a. De manera similar se realizan combinaciones para los tamaños de 64 y 128 entradas para ROB y con combinaciones de B-ROB de hasta 128 entradas, tal como se ilustra en los diagramas de las figuras 14-18b y 14-18c respectivamente. Un proceso idéntico al anterior se realiza para simular la arquitectura distribuida-optimizada.

Finalmente, cabe mencionar que se simularán entonces 3 configuraciones monolíticas, 12 distribuidas y 12 optimizadas-distribuidas para cada uno de los 14 benchmarks a utilizar, obteniéndose un total de 378 diferentes arquitecturas simuladas, simulando 200 millones de instrucciones cada vez. Un ejemplo del uso del simulador para evaluar una arquitectura se presenta en el anexo A de esta Tesis.

4.6. Resumen del Capítulo

En el presente capítulo se describió la metodología utilizada para realizar el diseño de la arquitectura del ROB, la cual inicia con un análisis de las arquitecturas existentes para encontrar posibles mejoras que puedan implementarse para mejorar su desempeño. Enseguida se presentaron las características de la arquitectura propuesta para el ROB y se detallaron las subestructuras que lo componen así como las funciones de cada una de ellas, destacándose la introducción de una subestructura llamada Branch ROB.

También se destaca el hecho de que las subestructuras del diseño trabajan en conjunto para dotar al procesador de la capacidad de ejecutar instrucciones fuera de orden de programa, esto utilizando un esquema distribuido llamado D-ROB en donde, a diferencia de un esquema monolítico tradicional, M-ROB, las subestructuras se podrían disponer en diferentes regiones del layout en base a la relación que mantengan con otros bloques funcionales del procesador, evitando así el surgimiento de fenómenos indeseable como *hotspots*.

Se introdujo también una optimización al diseño propuesto donde se implementa un esquema de recuperación inmediata que deberá incrementar el desempeño del procesador. Se explica además la manera en que el diseño es modelado en software y simulado mediante el simulador de arquitecturas llamado SimpleScalar.

Finalmente se describieron las características de las simulaciones que se realizan para evaluar esta nueva arquitectura. Los resultados de este proceso de simulación se presentan en el siguiente capítulo.

CAPÍTULO 5

PRUEBAS Y RESULTADOS

El presente capítulo presenta el análisis de las estadísticas obtenidas tras realizar las simulaciones de la arquitectura diseñada mediante la evaluación de diferentes esquemas y configuraciones, las cuales fueron descritas en la sección 4.5. Una sección relevante del código modificado del simulador que modela el diseño propuesto se muestra en el anexo B.

El resultado de cada simulación es un archivo de datos que contiene alrededor de 100 estadísticas diferentes con información de diversos valores utilizados para conocer el desempeño de diferentes aspectos del modelo, como por ejemplo el IPC obtenido, la cantidad de instrucciones de salto condicional ejecutadas, la cantidad de veces que el ROB se llenó etcétera. Un ejemplo de este archivo, que contiene el conjunto de las estadísticas ya mencionadas, se presenta en el anexo C de este trabajo.

Una vez que se tienen todos los archivos de datos correspondientes a cada una de las configuraciones simuladas, se debe realizar una selección de las estadísticas de mayor interés que reflejen mejor el impacto que tienen las modificaciones implementadas en el modelo propuesto respecto a una arquitectura convencional.

Esta selección se discute en la sección 5.1. Los datos obtenidos en cada simulación para cada estadística elegida deben ser entonces condensados y presentados en forma gráfica para poder realizar un mejor análisis y evaluación de los mismos, esto se realiza mediante la generación de *scripts* que permiten extraer y ordenar los resultados, un ejemplo de dichos *scripts* se presenta en el anexo D de esta Tesis. Los resultados ya procesados se muestran en forma de gráficas de barras y de tablas, de la sección 5.2 a la 5.5.

5.1. Descripción de las estadísticas estudiadas

Para evaluar el desempeño de la arquitectura propuesta que se ha modelado, es necesario seleccionar, del conjunto de estadísticas arrojadas tras la simulación, las que mejor sirvan para este propósito. Para el presente trabajo, se analiza en primer lugar la métrica correspondiente al valor del IPC. Esta estadística es considerada la más importante para evaluar el desempeño del procesador.

También se analiza la estadística correspondiente al valor del número total de instrucciones ejecutadas y se presenta un análisis del porcentaje de instrucciones

cuya ejecución se realiza en caminos equivocados derivados de un error en la predicción de saltos condicionales, estas ejecuciones innecesarias pueden ser evitadas mediante el esquema optimizado de restauración inmediata.

Una siguiente estadística a analizar es la que indica el número de instrucciones ejecutadas por segundo. De manera similar a la métrica anterior este valor permite evaluar un posible incremento en la velocidad de instrucciones ejecutadas o *"speedup"*.

Una última estadística a evaluar es la ocupación del ROB/B-ROB que permite estimar la cantidad de veces que el procesador genera una señal de *stall* o de paro en el motor de emisión de instrucciones, lo cual, como se sabe, afecta en forma directa el desempeño del procesador.

Cabe mencionar que los datos son presentados, por una parte, en gráficas de barras, en las cuales se muestran los resultados correspondientes al conjunto de benchmarks para operaciones de enteros, llamados *SPECint* (Integer component of SPEC CPU2000), enseguida los resultados para los programas de prueba de operaciones de punto flotante, *SPECfp* (Floating Point component of SPEC CPU2000), y también se presentan unas tablas que condensan los promedios de estos dos grupos de benckmarks así como el promedio, para cada una de las estadísticas estudiadas.

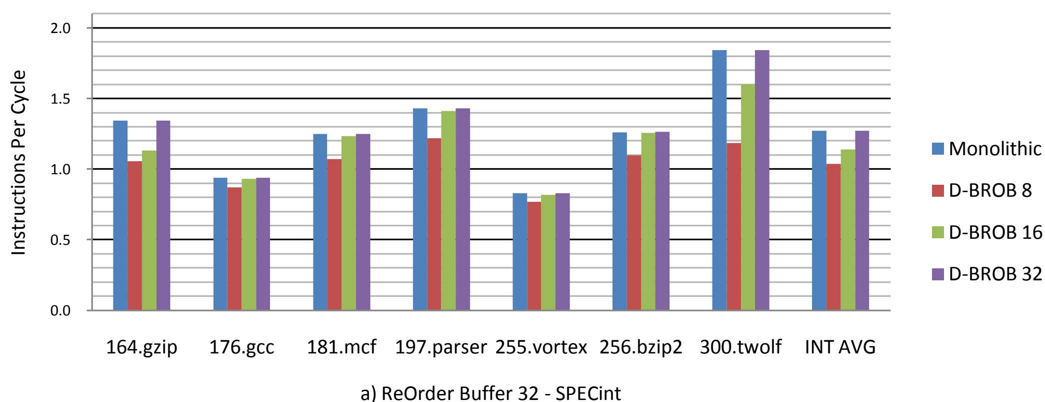
5.2. Análisis y evaluación del IPC

El IPC o Instrucciones Por Ciclo es la métrica más importante para medir el rendimiento del procesador, su inverso es el CPI ó Ciclos Por Instrucción. Para obtener el IPC, el simulador calcula la relación que existe entre el número de instrucciones graduadas, es decir, que han sido ejecutadas y retiradas correctamente, contra el número de ciclos de reloj empleados para este fin.

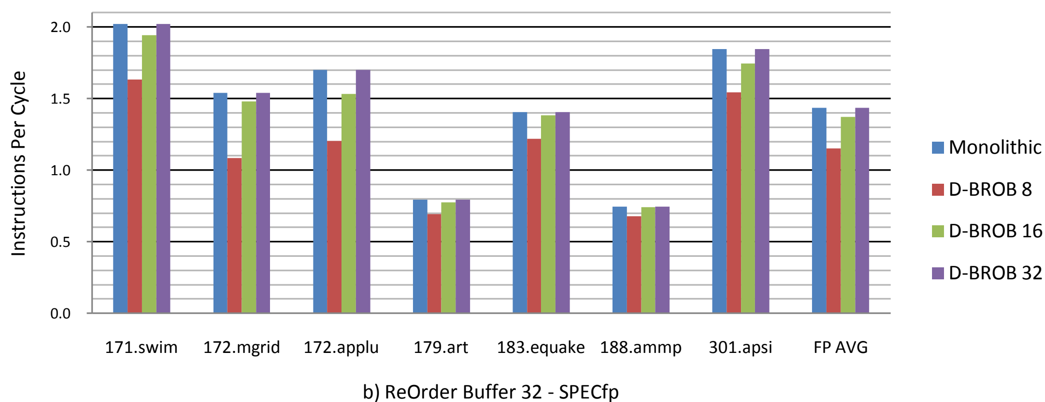
Como se sabe, mientras mayor sea el valor del IPC, significará que un mayor número de instrucciones son graduadas en un menor número de ciclos de reloj, lo que implica un mejor desempeño de la arquitectura evaluada, tal como lo muestra la ecuación 5-1.

$$\textbf{Instrucciones Por Ciclo} = \frac{\textbf{Número de Instrucciones graduadas}}{\textbf{Ciclos de reloj empleados}} \qquad 5\text{-}1$$

Las figuras 5-1a y 5-1b muestran los resultados obtenidos de la comparación de una arquitectura que implementa una estructura ROB monolítica convencional, llamada M-ROB, de 32 entradas, contra una arquitectura que implementa una estructura BROB distribuida, D-ROB, para los programas de prueba de operaciones de enteros, SPECint y de punto flotante, SPECfp, respectivamente.



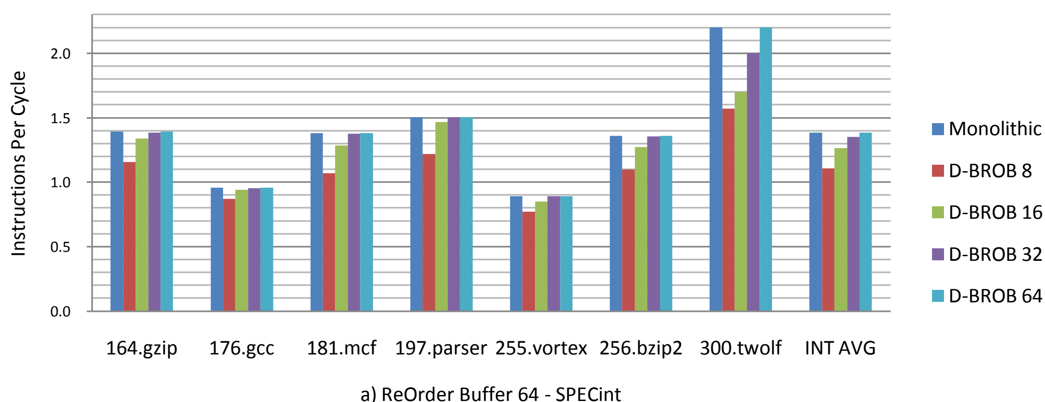
a) ReOrder Buffer 32 - SPECint



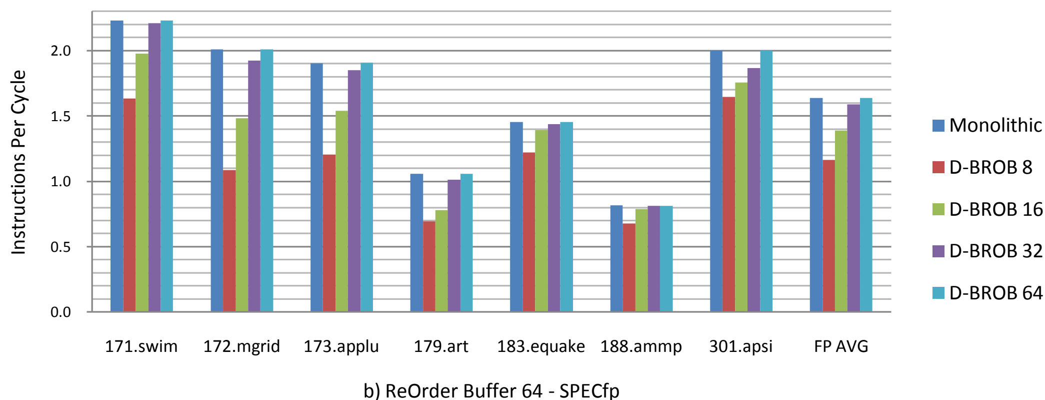
b) ReOrder Buffer 32 - SPECfp

Figura 5-1. IPC, configuración M-ROB de 32 entradas contra D-ROB.

Las figuras 5-2a y 5-2b, presentan una comparación similar a las anteriores con la diferencia de que la configuración simulada para el M-ROB es de 64 entradas. De la misma manera, las figuras 5-3a y 5-3b muestran los resultados del IPC con una configuración de 128 entradas para el ROB monolítico.

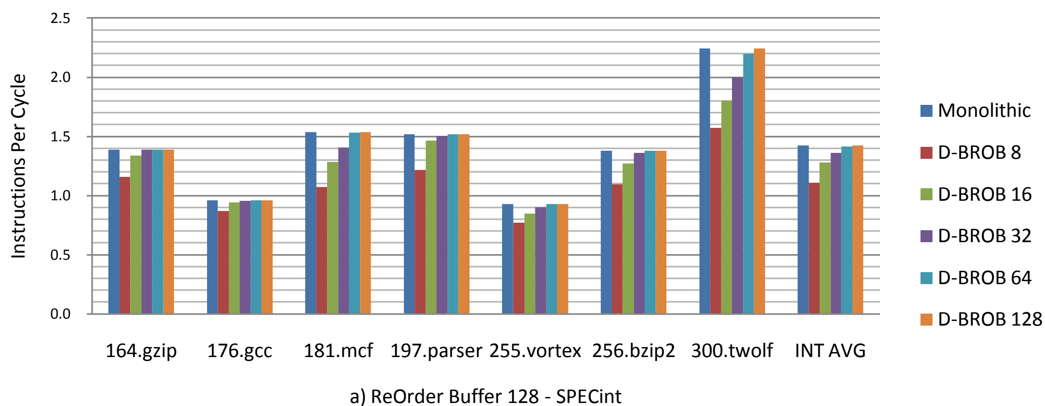


a) ReOrder Buffer 64 - SPECint

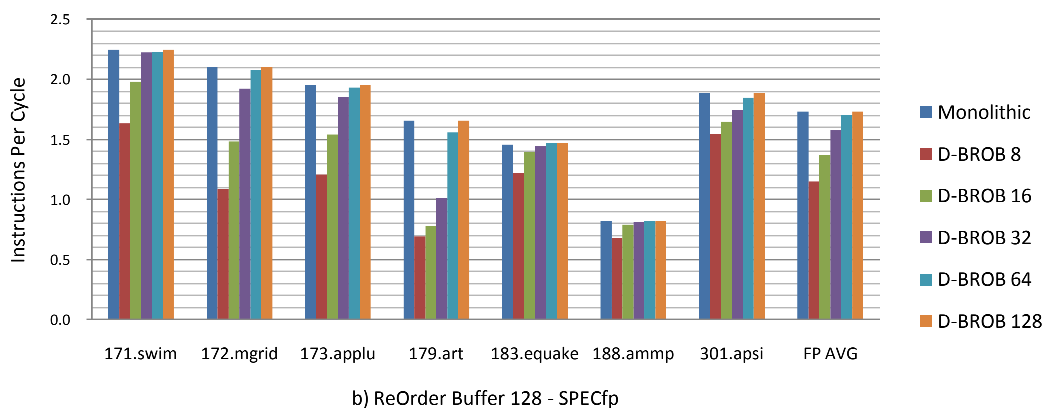


b) ReOrder Buffer 64 - SPECfp

Figura 5-2. IPC, configuración M-ROB de 64 entradas Distribuida-ROB.



a) ReOrder Buffer 128 - SPECint



b) ReOrder Buffer 128 - SPECfp

Figura 5-3. IPC, configuración M-ROB de 128 entradas contra D-ROB.

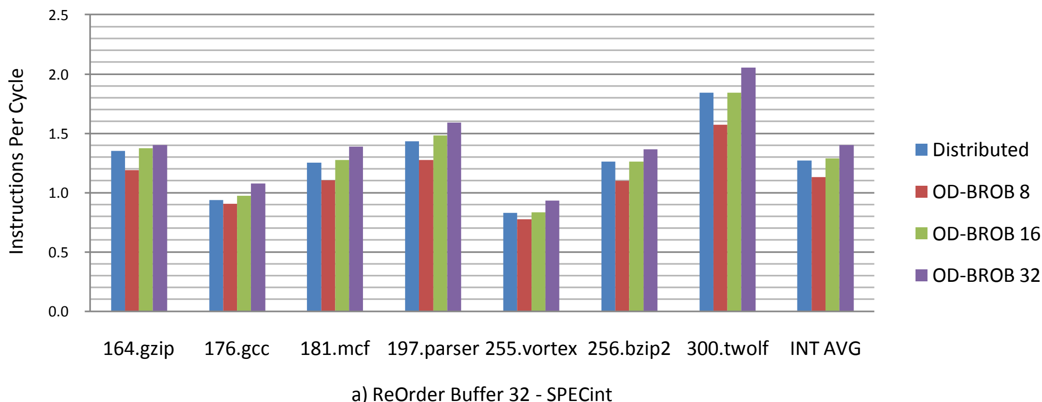
Como se puede ver, el tamaño de la subestructura B-ROB tiene un claro impacto en el valor del IPC obtenido. Los resultados muestran que el modelo distribuido sufre una ligera disminución del valor del IPC, el cual se vuelve más notorio cuando

el número de entradas del BROB es bastante reducido con respecto al número de entradas del ROB. Esto se puede notar con mayor claridad observando la tabla 5-1.

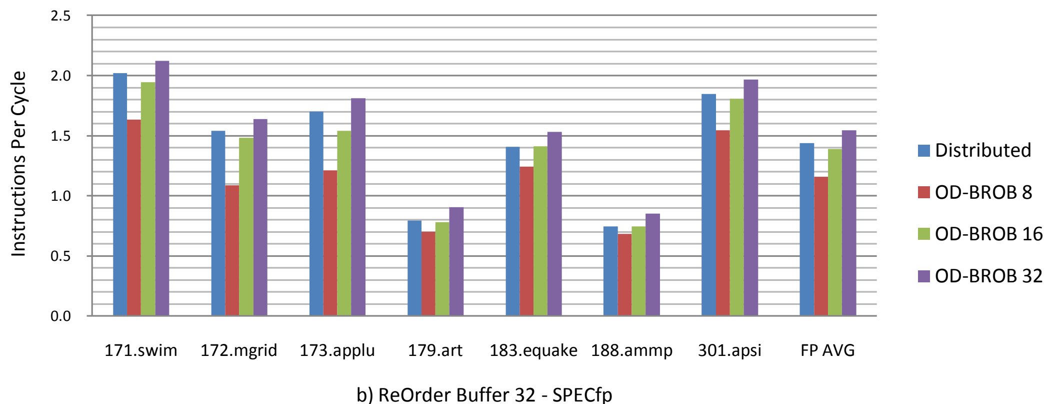
Tabla 5-1. Valor promedio del IPC, configuración M-ROB contra D-ROB.

	M-ROB		D-ROB									
	-		BROB: 8		BROB: 16		BROB: 32		BROB: 64		BROB: 128	
Conf.	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp
32	1.2708	1.4362	1.0377	1.1513	1.1386	1.3723	1.2707	1.4362	-	-	-	-
%	100	100	-18.34	-19.83	-10.40	-4.44	0	0	-	-	-	-
64	1.3834	1.6382	1.1073	1.1656	1.2638	1.3882	1.3519	1.5870	1.3832	1.6387	-	-
%	100	100	-19.95	-28.84	-8.64	-15.26	-2.27	-3.12	-0.01	+0.03	-	-
128	1.4216	1.7313	1.1073	1.1513	1.2781	1.3725	1.3590	1.5737	1.4145	1.7044	1.4216	1.7334
%	100	100	-22.10	-33.50	-10.09	-20.72	-4.40	-9.10	-0.49	-1.55	0	+0.12

Esta disminución es más notoria cuando el tamaño del BROB tiene 1/16 o menos del número de entradas del ROB. Al evaluar el desempeño del modelo distribuido contra el que implementa la optimización descrita en la sección 4.4, se observa un aumento en el valor promedio del IPC hasta de un 9.5% para el caso de SPECint y de hasta un 6.9% para SPECfp, dando en promedio un aumento de alrededor un 8%, debido a la característica del modelo optimizado que permite una recuperación inmediatamente después de la detección de un error de salto condicional, esto se demuestra mediante las gráficas de las figuras 5-4 a 5-6 y con la tabla 5-2.

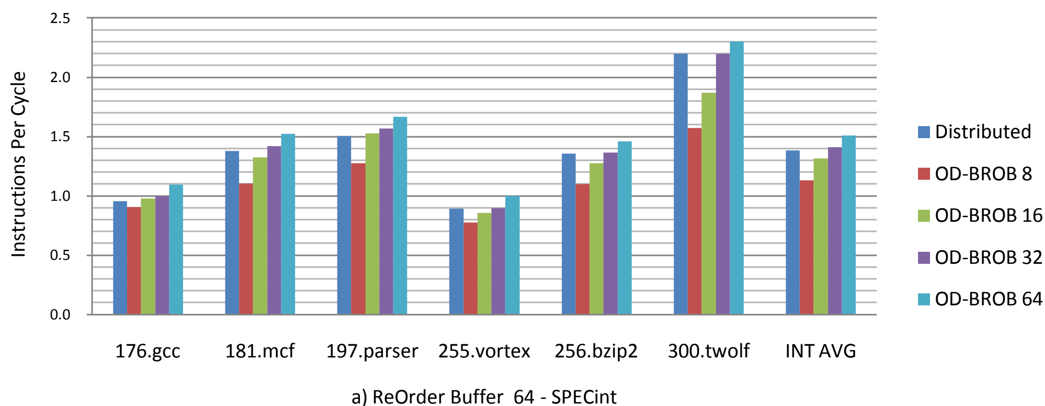


a) ReOrder Buffer 32 - SPECint

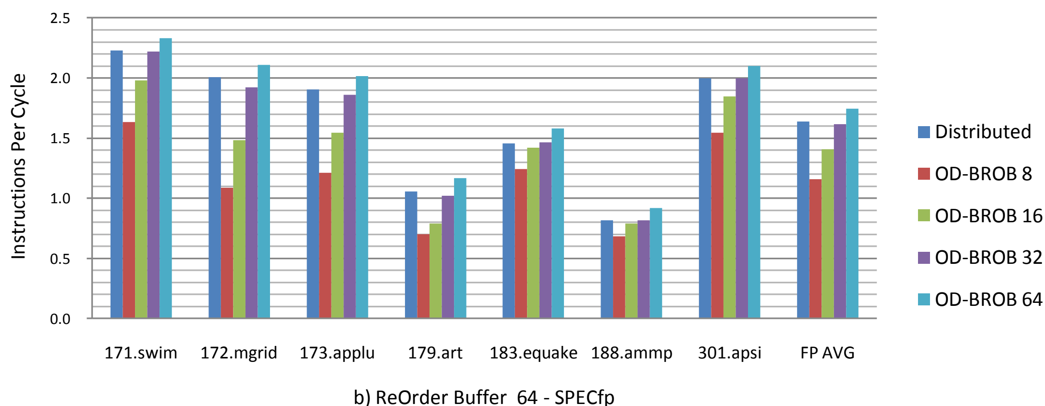


b) ReOrder Buffer 32 - SPECfp

Figura 5-4. IPC, configuración D-ROB de 32 entradas contra OD-ROB.

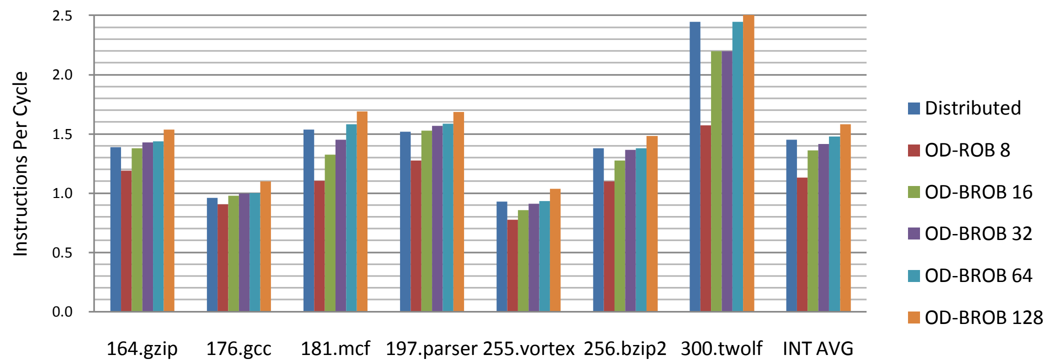


a) ReOrder Buffer 64 - SPECint

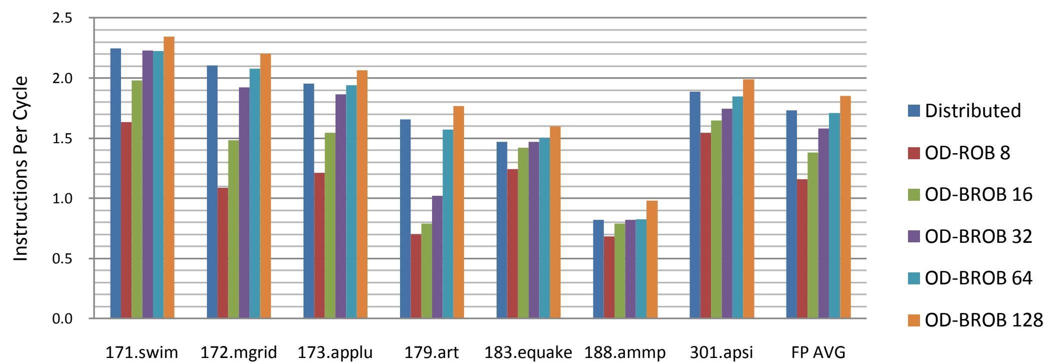


b) ReOrder Buffer 64 - SPECfp

Figura 5-5. IPC, configuración D-ROB de 64 entradas contra OD-ROB.



a) ReOrder Buffer size 128 - SPECint



b) ReOrder Buffer size 128 - SPECfp

Figura 5-6. IPC, configuración D-ROB de 128 entradas contra OD-ROB.

Tabla 5-2. Valor promedio del IPC, configuración D-ROB contra OD-ROB.

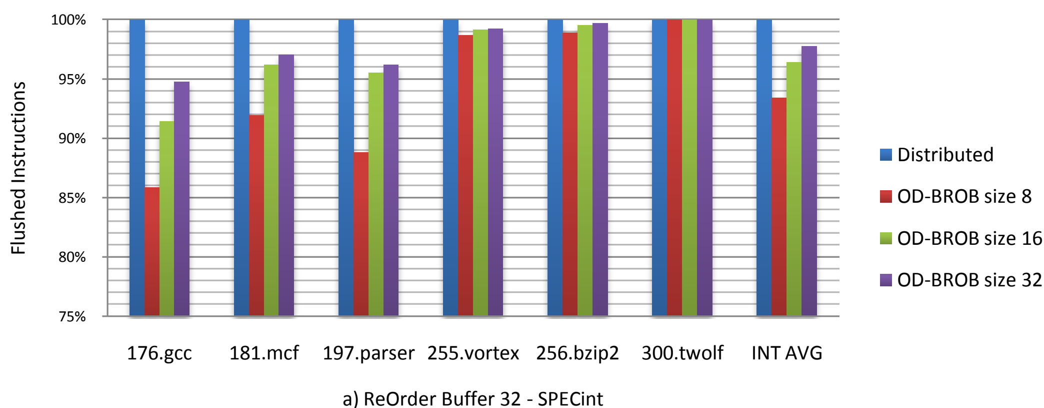
	D-ROB		OD-ROB									
	-		BROB: 8		BROB: 16		BROB: 32		BROB: 64		BROB: 128	
Conf.	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp
32	1.2720	1.4364	1.1316	1.1568	1.2904	1.3872	1.4017	1.5458	-	-	-	-
%	100	100	-11.03	-19.46	-1.44	-3.42	+10.19	+7.61	-	-	-	
64	1.3824	1.6387	1.1316	1.1568	1.3155	1.4074	1.4101	1.6347	1.5113	1.7458	-	-
%	100	100	-18.14	-29.40	-4.83	-14.11	+2.00	-0.244	+9.32	+6.53	-	-
128	1.4502	1.7334	1.1316	1.1568	1.3624	1.3788	1.4164	1.5815	1.4798	1.7214	1.5807	1.8497
%	100	100	-21.96	-33.26	-6.05	-20.45	-2.33	-8.76	+2.04	-0.69	+8.99	+6.70

5.3. Reducción de la ejecución de instrucciones en traza equivocada

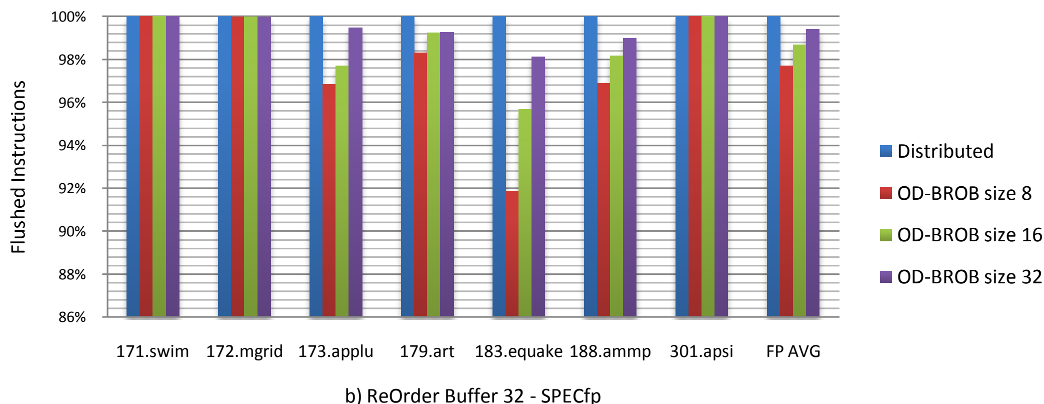
Para entender el efecto que tiene la optimización de la recuperación inmediata en el desempeño de la arquitectura modelada, se presentan las gráficas de la figuras 5-7 a 5-9. Estas gráficas utilizan las estadísticas correspondientes al número total de instrucciones ejecutadas, menos el número total de instrucciones que se retiraron correctamente, esto es, instrucciones graduadas, lo cual corresponde al número de instrucciones que fueron ejecutadas en una traza equivocada, como se muestra en la ecuación 5-2.

Inst. En traza equivocada = Inst. Ejecutadas – Inst. Graduadas (5-2)

Se puede notar que en la versión optimizada se reduce hasta un 10% el número de instrucciones que fueron ejecutadas en una traza equivocada. Esto permite que el motor de ejecución del procesador ejecute un máximo de instrucciones en la traza correcta, incrementando así su desempeño.

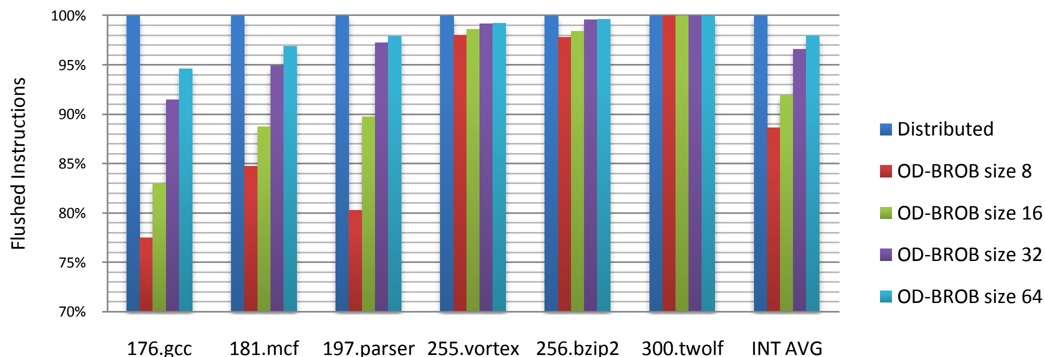


a) ReOrder Buffer 32 - SPECint

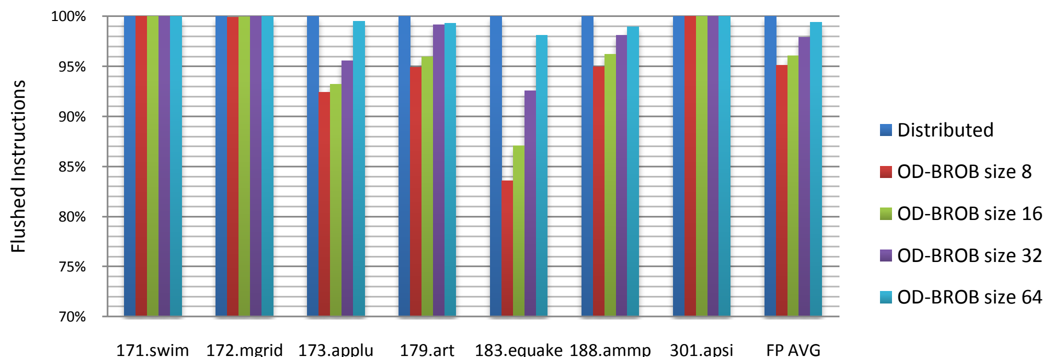


b) ReOrder Buffer 32 - SPECfp

Figura 5-7. FI, configuración D-ROB de 32 entradas contra OD-ROB.

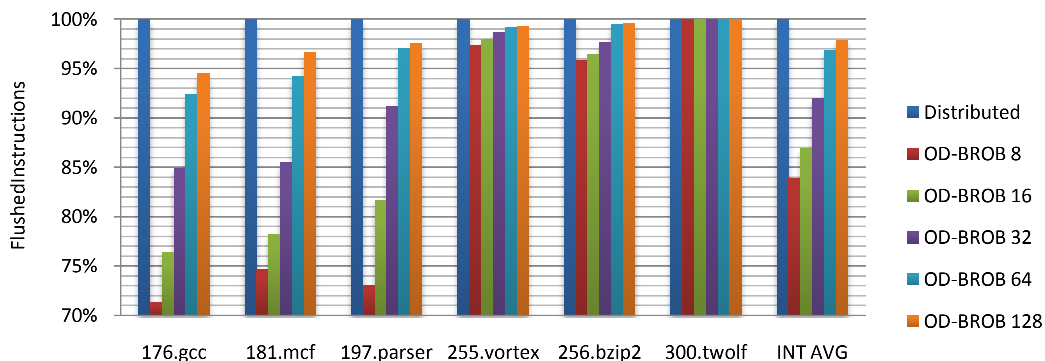


a) ReOrder Buffer 64 - SPECint



b) ReOrder Buffer 64 - SPECfp

Figura 5-8. FI, configuración D-ROB de 64 entradas contra OD-ROB.



a) ReOrder Buffer 128 - SPECint

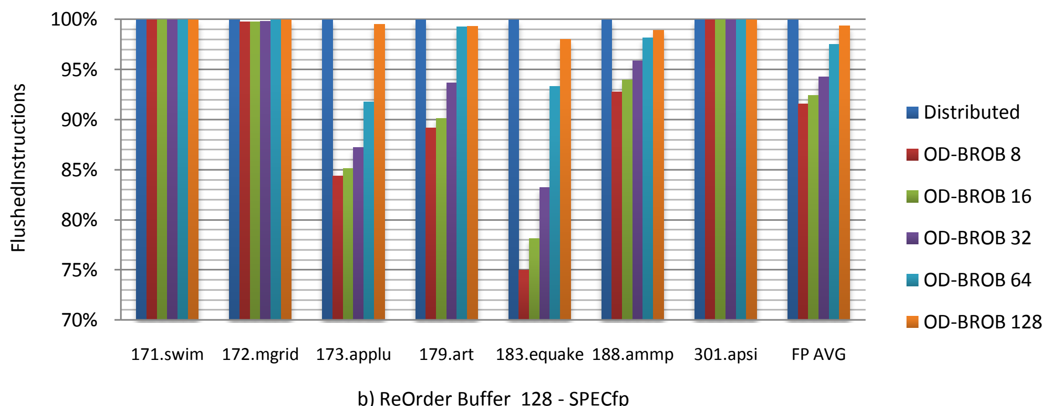


Figura 5-9. FI, configuración D-ROB de 128 entradas contra OD-ROB.

Tabla 5-3. Valor promedio de FI, configuración D-ROB contra OD-ROB.

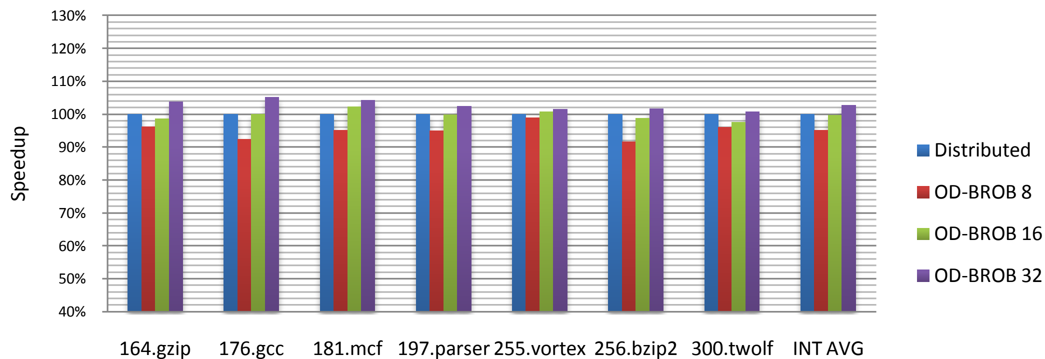
	D-ROB		OD-ROB									
	-		BROB: 8		BROB: 16		BROB: 32		BROB: 64		BROB: 128	
Conf.	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp
32	1	1	0.9340	0.9769	0.9640	0.9868	0.9775	0.9940	-	-	-	-
64	1	1	0.8861	0.9511	0.9195	0.9605	0.9661	0.9790	0.9793	0.9940	-	-
128	1	1	0.8386	0.9159	0.8690	0.9246	0.9196	0.9427	0.9680	0.9751	0.9782	0.9940

5.4. Incremento en la velocidad de ejecución de instrucciones

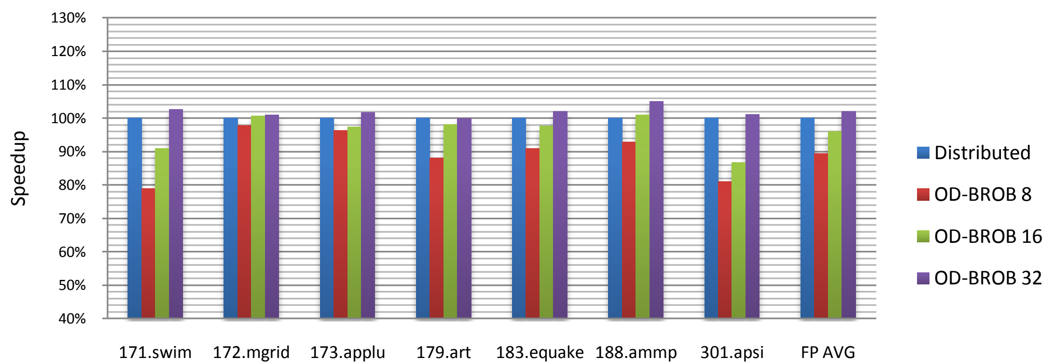
Otra estadística significativa que se analiza es la cantidad de instrucciones que son ejecutadas por segundo. Con estos datos es posible observar si existe un incremento en el número de instrucciones retiradas en el mismo tiempo de ejecución, como se muestra en la ecuación 5-3.

$$\boldsymbol{Speedup} = \frac{\boldsymbol{Inst.\ Graduadas}}{\boldsymbol{Tiempo\ de\ ejecución}} \qquad \text{5-3}$$

Como lo muestran las gráficas de la figura 5-4, la versión optimizada permite obtener un speedup de hasta un 10% con respecto a la versión que no implementa dicha optimización. Esto se debe a que se maximiza el número de instrucciones ejecutadas en una traza correcta, las cuales son retiradas posteriormente, lo cual incide positivamente en esta métrica.

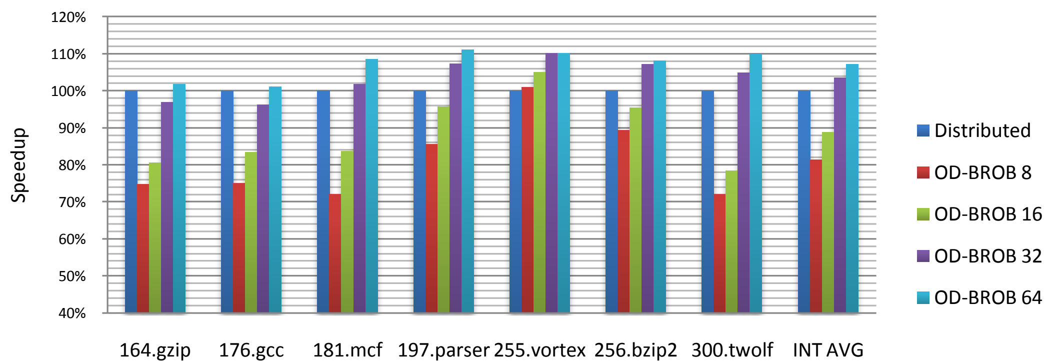


a) ReOrder Buffer 32 - SPECint

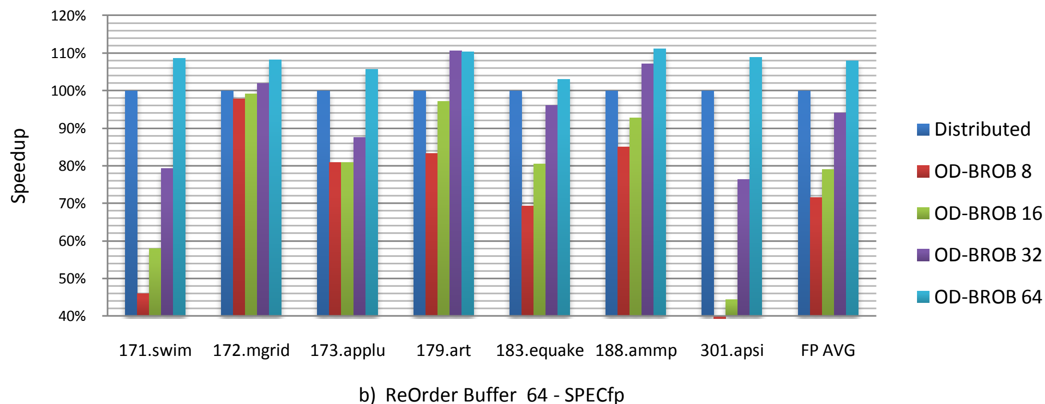


b) ReOrder Buffer 32 - SPECfp

Figura 5-10. Speedup, configuración D-ROB de 32 entradas contra OD-ROB.

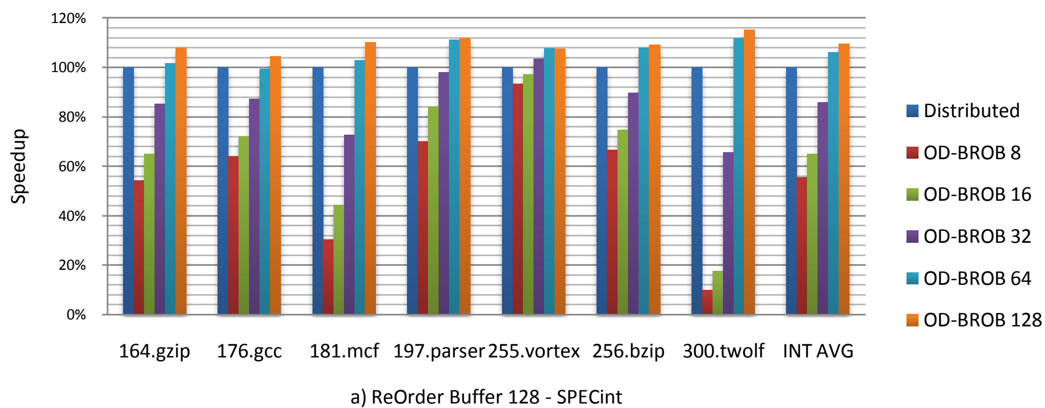


a) ReOrder Buffer 64 - SPECint

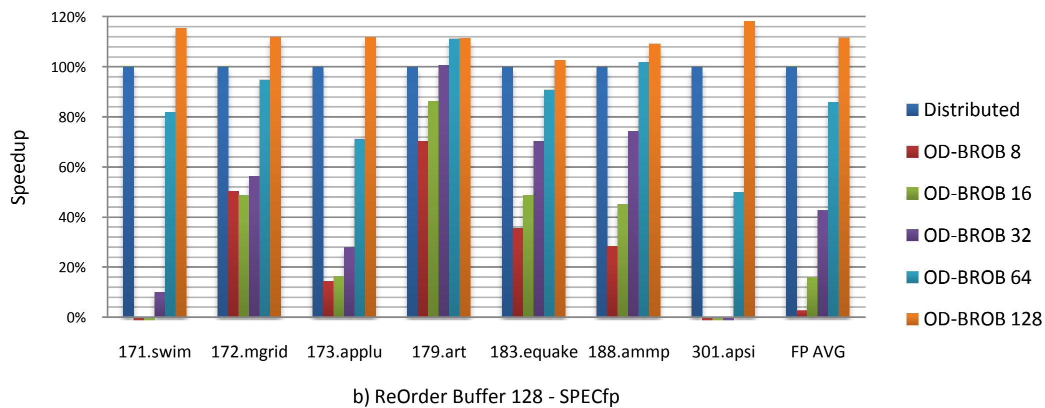


b) ReOrder Buffer 64 - SPECfp

Figura 5-11. Speedup, configuración D-ROB de 64 entradas contra OD-ROB.



a) ReOrder Buffer 128 - SPECint



b) ReOrder Buffer 128 - SPECfp

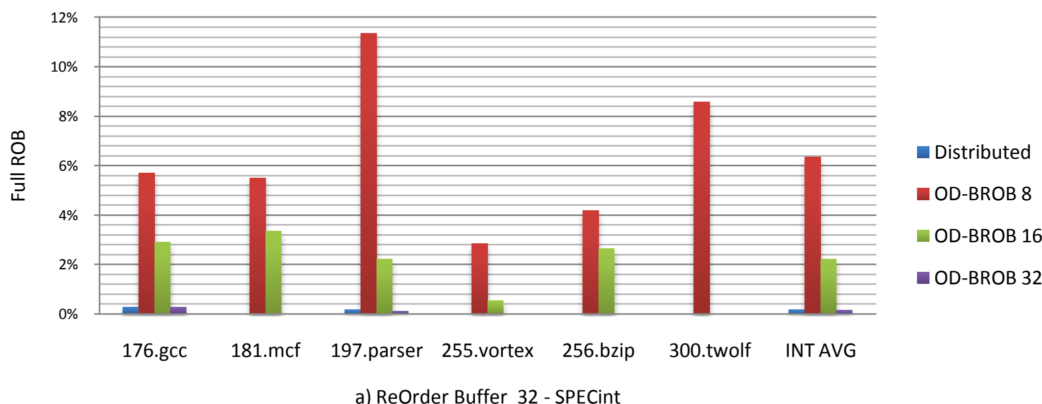
Figura 5-12. Speedup, configuración D-ROB de 128 entradas contra OD-ROB.

Tabla 5-4. Valor promedio del speedup, configuración D-ROB contra OD-ROB.

	M-ROB		D-ROB									
	-		BROB: 8		BROB: 16		BROB: 32		BROB: 64		BROB: 128	
Conf.	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp
32	1	1	0.9508	0.8934	0.9970	0.9602	1.0277	1.0189	-	-	-	-
64	1	1	0.8137	0.7161	0.8885	0.7900	1.0346	0.9414	1.0797	1.0717	-	-
128	1	1	0.5540	0.0262	0.6488	0.1589	0.8586	0.4266	1.0604	0.8589	1.1153	1.0942

5.5. Ocupación del ROB

Finalmente, se estudia la estadística que calcula la ocupación del ROB. Las gráficas de la figura 5-5 muestran el porcentaje del tiempo de simulación en que el ROB, estuvo lleno, lo cual causa un evento de *stall* donde el procesador detiene la emisión de instrucciones afectando el IPC. Al analizar estas gráficas se observa una relación consistente entre el tamaño del BROB y el porcentaje de tiempo que el ROB permanece lleno. Comparando los resultados para la arquitectura distribuida contra la distribuida optimizada, se encuentra que la optimizada con un ROB de 1/4 del tamaño de la distribuida se llena apenas un 6% más que ésta.



a) ReOrder Buffer 32 - SPECint

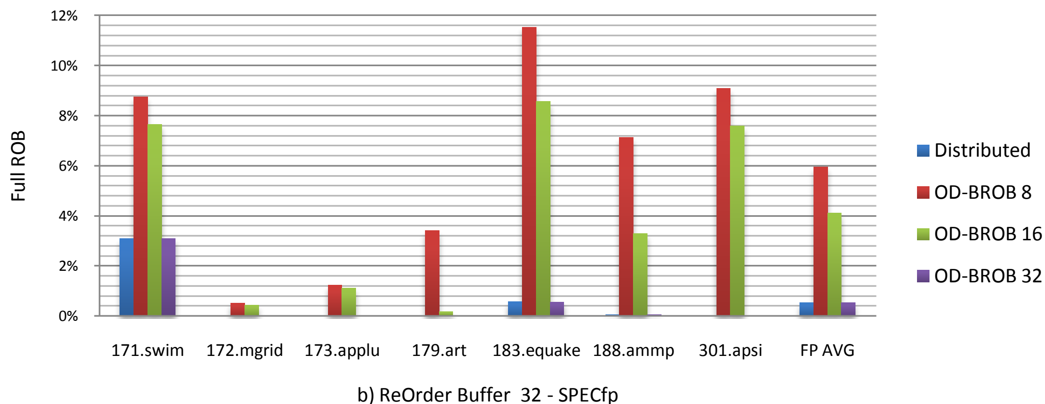
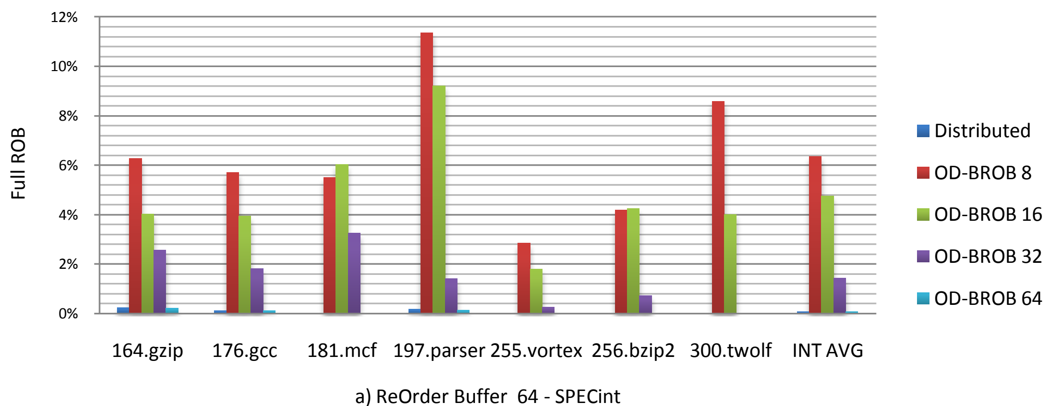


Figura 5-13. ROB lleno, configuración D-ROB de 32 entradas contra OD-ROB.



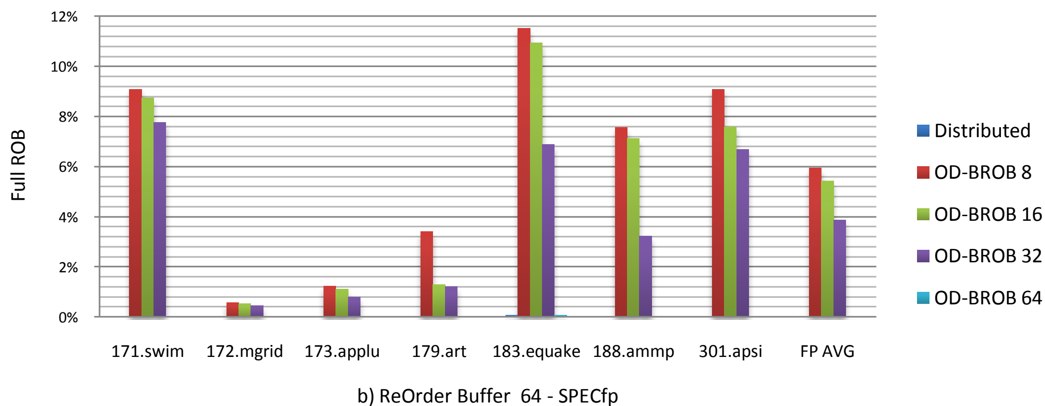
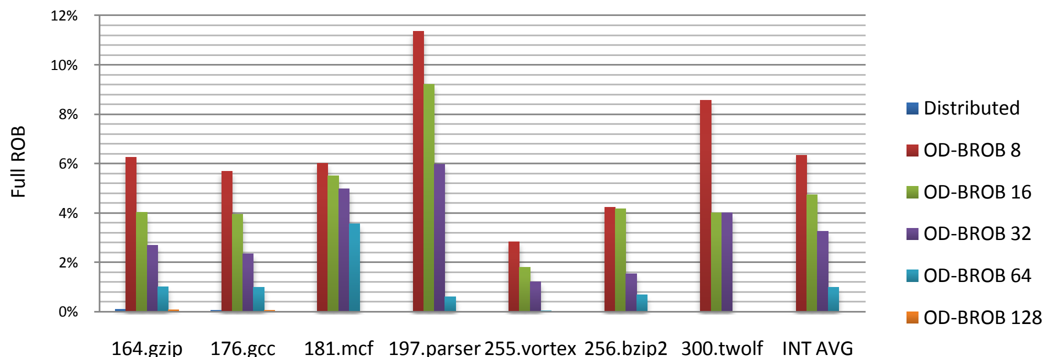
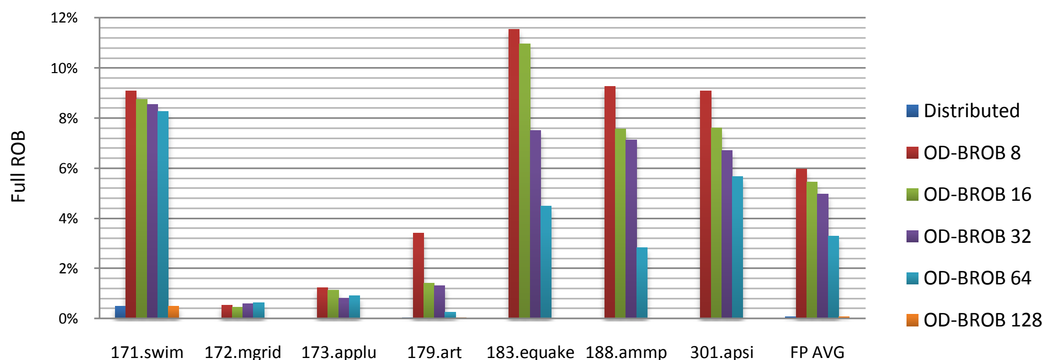


Figura 5-14. ROB lleno, configuración D-ROB de 64 entradas contra OD-ROB.



a) ReOrder Buffer 128 - SPECint



b) ReOrder Buffer 128 - SPECfp

Figura 5-15. ROB lleno, configuración D-ROB de 128 entradas contra OD-ROB.

Tabla 5-5. Valor promedio de ROB lleno, configuración D-ROB contra OD-ROB.

	M-ROB		D-ROB									
	-		BROB: 8		BROB: 16		BROB: 32		BROB: 64		BROB: 128	
Conf.	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp	SPECint	SPECfp
32	0.0016	0.0053	0.0595	0.0634	0.0222	0.0412	0.0015	0.0053	-	-	-	-
64	0.0001	0.0007	0.0595	0.0634	0.0475	0.0544	0.0143	0.0388	0.0001	0.0006	-	-
128	0.0002	0.0007	0.0595	0.0634	0.0475	0.0544	0.0325	0.497	0.0098	0.0329	0.0002	0.0007

5.6. Resumen del Capítulo

En este capítulo se presentaron los resultados obtenidos después de realizar una serie de simulaciones correspondientes a las arquitecturas que se deseaban evaluar. La conclusión final que se obtiene una vez que se han analizados dichos resultados se presenta en el siguiente capítulo.

CAPÍTULO 6

CONCLUSIONES Y TRABAJO FUTURO

6.1. Conclusiones

En este trabajo de tesis se ha realizado el diseño la arquitectura de un úfer de reordenamiento de instrucciones para un procesador superescalar con ejecución fuera de orden. En este diseño, el ROB tiene la característica de tratarse como un conjunto de subestructuras que pueden distribuirse en diferentes áreas del layout del procesador y que trabajan como una unidad para resolver tareas, como habilitar la ejecución especulativa de instrucciones, permitir la recuperación de errores de predicción de salto y reciclar recursos físicos. Este diseño aprovecha dos posibles oportunidades de mejora con respecto a las arquitecturas tradicionales que han sido analizadas.

En primer lugar, se observa que almacenar el valor del contador de programa para las instrucciones de todo tipo es innecesario ya que en realidad, este valor solo llega a ser utilizado cuando se trata de instrucciones de salto condicional, con el objeto de posibilitar la recuperación en caso de que surja un evento de error de predicción de salto. Por lo tanto se introduce una subestructura dedicada a almacenar el PC de las instrucciones de salto únicamente, llamada BROB. Dado que la proporción de instrucciones de tipo salto condicional con respecto al resto de instrucciones es de aproximadamente de 1 por cada 4, se observa que el tamaño de la subestructura BROB no necesita ser del mismo tamaño que el resto de las subestructuras, pudiendo reducirse hasta en un 75% del tamaño del ROB.

Para determinar la viabilidad de esta aproximación, se realiza un modelo de este diseño partiendo de la arquitectura base del simulador SimpleScalar, un simulador usado extensamente para la investigación de nuevos diseños dentro del área de arquitectura de computadoras, encontrándose que al utilizar un esquema donde el tamaño del BROB es cuatro veces más pequeño que el ROB, solo se observa una disminución del 6% en el valor del IPC simulado con respecto a una arquitectura monolítica tradicional con igual número de entradas para ROB y BROB.

La segunda aproximación que se implementa para mejorar este diseño, está basado en la observación de que el modelo distribuido, llamado DROB, modelado con una proporción menor de entradas del BROB puede optimizarse si se implementa un mecanismo más efectivo para la recuperación de errores de salto. Se encuentra que la manera tradicional de tratar los errores de salto es esperar a que la

instrucción que lo causa llegue a la cabeza del ROB para lanzar los mecanismos de recuperación. Esta aproximación es ineficiente porque al ocurrir un error de salto, e incluso después de que éste ha sido detectado, las instrucciones subsecuentes seguirán siendo ejecutadas en una traza equivocada y por lo tanto serán inevitablemente desechadas cuando la instrucción causante llegue a la cabeza de la FIFO, provocando la ejecución innecesaria de instrucciones del camino equivocado.

Por ello se propone una versión optimizada, llamada ODROB, en la cual los mecanismos de recuperación son lanzados inmediatamente después de que el error de salto haya sido detectado. Esta optimización permite liberar con mayor efectividad las entradas del ROB ocupadas por instrucciones ejecutadas en una traza errónea y aprovechar de mejor manera el motor de ejecución de instrucciones minimizando el número de instrucciones que se ejecutan en una traza equivocada y permitiendo que un mayor número de instrucciones ejecutadas en una traza correcta sean retiradas correctamente al llegar a la cabeza de la FIFO.

Al realizar esta modificación al modelo propuesto y una vez que se realiza la serie de simulaciones correspondientes, se observa un impacto positivo en el valor del IPC, el cual se incrementa hasta un 9% con respecto a la arquitectura que no la implementa.

Con la implementación de estas dos características, la primera capaz de reducir los recursos necesarios para implementar el ROB y en consecuencia minimizando su tamaño físico, y la segunda haciendo más efectivo su funcionamiento, mejorando el desempeño del procesador, se obtiene un diseño eficiente en recursos y de alto desempeño. De acuerdo la información obtenida tras la serie de simulaciones ejecutadas, se observa que las configuraciones donde la relación ROB a BROB es 4 a 1 son particularmente eficientes, y de ellas la configuración optimizada de 128 entradas para el ROB y 32 para el BROB tiene las mejores condiciones, pues se obtiene una Ventana de Instrucciones de gran tamaño que permite extraer grandes cantidades de paralelismo a nivel de instrucción y presenta además un desempeño similar a la de una arquitectura monolítica típica de 128 entradas.

Finalmente cabe mencionar que aunque el incremento del valor del 9% en el valor del IPC puede parecer modesto, implica una mejora significativa si se considera además que se trata de la modificación de solo una estructura funcional, de modo que una mejora en el desempeño total del procesador puede llegar a incrementarse si este diseño se implementa junto con otros componentes optimizados como el bloque de renombramiento de registros o la unidad de predicción de saltos.

6.2. Trabajo Futuro

Como trabajo futuro se propone continuar con la búsqueda de posibles mejoras para la estructura del ROB, como puede ser la reducción de recursos utilizados o los

mecanismos de recuperación del estado del procesador, además se sugiere combinar el diseño propuesto con otros bloques optimizados del procesador, como la etapa de renombramiento de registros o la unidad de predicción de saltos, realizando un modelo que incluya de esta manera varios bloques optimizados para verificar mediante simulaciones que en conjunto se logre un mayor incremento total en el desempeño de esta arquitectura.

Por otra parte se propone también tomar los resultados obtenidos descritos anteriormente como base para realizar un modelo a nivel VLSI del diseño propuesto del ROB. Este proceso debe incluir en primer lugar la generación de un modelo usando un lenguaje de descripción de hardware y posteriormente la elaboración de su correspondiente layout, el cual que podrá unirse a diseños posteriores de otros bloques del procesador siguiendo esta misma metodología de diseño, hasta lograr obtener eventualmente un diseño completo a nivel layout para un procesador superescalar.

REFERENCIAS BIBLIOGRÁFICAS

[1] J. Hennessy, D. Patterson, "Computer Architecture, A Quantitative Approach", Morgan Kauffman, 2003.

[2] D. Sima, T. Fountain, P. Kacsuk, "Advanced Computer Architectures, A Design Space Approach", Addison-Wesley, 1998.

[3] T. Shanley, "The Unabridged Pentium 4", Addison-Wesley, 2005.

[4] K. Yeager, "The MIPS R10000 Superscalar Microprocessor", *IEEE Micro*, vol. 16, no. 2, Apr. 1996.

[5] R. E. Kessler, E. J. McLellan, and D. A. Webb "The Alpha 21264 Microprocessor Architecture", *IEEE Micro,* vol. 19, no. 2, Mar. 1999.

[6] D. Sager et al. "The micro architecture of the Pentium 4 processor", *Intel Technology Journal*, vol. 5, no. 1, 2001.

[7] W. Hu, F. Zhang, Z. Li, "Microarchitecture of the Godson-2 processor", *ACM Journal of Computer Science and Technology,* vol. 20, no.2, Mar. 2005.

[8] S. Palacharla, N. P. Jouppi, J. E. Smith, "Complexity-Effective Superscalar Processors", *Proceedings of The annual International Symposium on Computer Architecture,* Jun. 1998.

[9] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal, Vol. 11, Jan. 1967.

[10] J. Smith, A. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", *IEEE Transactions on Computers*, vol. 16, no. 5, May. 1998.

[11] M. Butler and Y. Patt, "A Comparative Performance Evaluation of Various State Maintenance Mechanisms", *Proceedings of the Annual International Symposium on Microarchitecture*, 1993.

[12] C. Kozyrakis, D. Patterson, "A New Direction for Computer Architecture Research", IEEE Computer, 1998.

[13] K. Skadron, P. Ahuja, M. Martonosi, D. Clark, "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques", *IEEE Transactions on Computers*, vol. 48, no. 11, Nov. 1999.

[14] T. Taha, and D. Willis, "An Instruction Throughput Model of Superscalar Processors", *IEEE Transactions on Computers*, vol. 57, no. 3, Mar. 2008.

[15] H. Akkary, R. Rajwar, S.T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors". *Proceedings of International Symposium on Microarchitecture*, Dec. 2003.

[16] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, J. Torrellas, "Cherry: Checkpointed Early Recycling in Out-of-Order Microprocessors". *Proceedings of International Symposium on Microarchitecture,* Nov. 2002.

[17] S. Petit, R. Ubal, J. Sahuquillo, P. Lopez, and J. Duato, "An Efficient Low-Complexity Alternative to the ROB for Out-of-Order Retirement of Instructions", *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, Aug. 2009.

[18] S. Petit, J. Sahuquillo, P. Lopez, R. Ubal, and J. Duato, "A Complexity-Effective Out-of-Order Retirement Microarchitecture", *IEEE Transactions on Computers,* vol. 58, no. 12, Dec. 2009.

[19] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors", *Proc. Int'l. Symp. Microarchitecture,* Dec. 2003.

[20] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors", *Proc. 36th Int'l Symp. On High performance Architecture*, Feb. 2004.

[21] G. Kucuk, D. Ponomarev, O. Ergin, and, K. Ghose, "Complexity-Effective Reorder Buffer Designs for Superscalar Processors", *IEEE Transactions on Computers*, vol. 53, no. 6, Jun. 2004.

[22] G. Kucuk, D. Ponomarev, O. Ergin, K. Ghose, "Reducing Reorder Buffer Complexity through Selective Operand Caching", *ISLPED'03*, Aug. 2003.

[23] G. Kucuk, O. Ergin, D. Ponomarev, K. Ghose, "Distributed Reorder Buffer Schemes for Low Power", *Proceedings of the 21st International Conference on Computer Design (ICCD'03)*, Oct. 2003.

[24] D. Ponomarev, G. Kucuk, K. Ghose, "Energy-Efficient Design of the Reorder Buffer", *Proceedings of the International Workshop on Power and Timing Modeling, Optimizations and Simulation*, Sep. 2002.

[25] G. Kucuk D. Ponomarev K. Ghose, "Low-Complexity Reorder Buffer Architecture", *Proceedings of the International Conference on Supercomputing*, Jun. 2002.

[26] Folegnani, D., Gonzalez, A., "Energy-Efficient Issue Logic", *Proceedings of ISCA*, Jul. 2001.

[27] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Checkpointed Early Load Retirement", *Proc. Int'l Symp. High Performance Architecture,* Feb. 2005.

[28] A. Moshovos, P. Akl, "Turbo-ROB: A Low Cost Checkpoint/Restore Accelerator", *HiPEAC*, 2008.

[29] I. Gonzalez, M. Galluzzi, A. Veidenbaum, M. Ramirez, A. Cristal, M. Valero, "A Distributed State Management Architecture for Large-Window Processors", Int. Symposium on Microarchitecture (Micro-41), 2008.

[30] F. Latorre, G. Magklis, J. Gonzalez, P. Chaparro, and A. Gonzalez, "CROB: Implementing a Large Instruction Window through Compression", *Transactions on High-Performance Embedded Architectures and Compilers,* vol. 3, no. 2, Springer 2008.

[31] F. Latorre, G. Magklis, J. Gonzalez, and P. Chaparro, “Building a Large Instruction Window through ROB Compression”, *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA’07), Sep.* 2007.

[32] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, E. Rotenberg “A Large Fast Instruction Window for Tolerating Cache Misses”, *Proceedings of the 29th annual international symposium on computer architecture*, May 2002.

[33] T. Austin, E. Larson, D. Ernst, “SimpleScalar: an infrastructure for computer system modeling”, *IEEE Computer, Computer System Modeling,* vol. 35, no.2, 2002.

[34] D.C. Burger and T.M. Austin, “The SimpleScalar Tool Set: Version 2.0”, Tech Report 1342. University of Wisconsin-Madison Computer Science, June 1997.

[35] D.C. Burger, T.M. Austin, and S. Bennett, “Evaluating Future Microprocessors, the SimpleScalar Tool Set”. Technical Report 1308, University of Wisconsin-Madison Computer Sciences Department, Jul. 1996.

[36] A. Kleinosowski, J. Flynn, N. Meares, D. Lilja, “Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer architecture Research”, *Workshop on Workload Characterization in International Conference on Computer Design*, 2000.

[37] J. L. Henning. “SPEC CPU2000: Measuring CPU Performance in the New Millennium”. IEEE Computer, Vol. 33, no 7, Jul. 2000.

[38] J. Yi, D. Lilja, and D. Hawkins, “Improving Computer Architecture Simulation Methodology by Adding Statistical Rigor”, *IEEE Transactions on Computers,* vol. 54, no. 11, Nov. 2005.

GLOSARIO

ACU (**A**ddress **C**alculation **U**nit): Una ACU es una unidad funcional en los procesadores modernos dedicada a ejecutar operaciones aritméticas o lógicas únicamente con datos de direcciones.

ARM (**A**dvanced **R**ISC **M**achine): Se denomina ARM a una familia de procesadores RISC, cuya arquitectura de 32 bits está orientada a aplicaciones de sistemas embebidos.

BP (**B**ranch **P**redictor): Un predictor de saltos es un mecanismo utilizado en los procesadores para reducir los ciclos de paro ocasionados por las condiciones de salto, resolviendo especulativamente dicha condición.

BROB (**B**ranch ROB): Subestructura que forma parte de la arquitectura propuesta para el diseño del ROB encargada de almacenar el valor de IPC de las instrucciones de salto condicional.

BRUU (**B**ranch **RUU**): Nombre que se le da a la subestructura que cumple las funciones del BROB en la nomenclatura del simulador SimpleScalar y que se modela modificando el código de dicho simulador.

BTB (**B**ranch **T**arget **B**uffer): El búfer de destino de saltos, es una memoria caché que almacena la dirección de la siguiente instrucción que sigue a un salto.

CAM (**C**ontent-**A**ddressable **M**emory): Una CAM, también llamada memoria asociativa, es un tipo de dispositivo de almacenamiento de datos, usados en cachés y MMUs, el cual incluye lógica de comparación con cada bit almacenado.

CPI (**C**lock **C**ycles **P**er **I**nstruction): En arquitectura de computadoras es un término usado para describir un aspecto del desempeño del procesador, el número de ciclos de reloj que transcurren para ejecutar una instrucción, es el inverso del IPC.

DROB (**D**istributed **ROB**): Esquema propuesto para el búfer de reordenamiento de instrucciones en el que la disposición de los elementos físicos que componen al ROB se distribuye en diferentes regiones del *layout* del procesador, de acuerdo con el bloque funcional con el que interactúa.

GCC (**G**NU **C**ompiler **C**ollection): Es un conjunto de compiladores desarrollados por el proyecto GNU. Estos compiladores incluyen soporte para los lenguajes C, C++, Fortran, Pascal y Java, y son considerados estándar en ambientes Linux y Unix.

ILP (**I**nstruction-**L**evel **P**arallelism): En arquitectura de computadoras, el paralelismo a nivel de instrucción es una medida de cuantas operaciones pueden ser ejecutadas simultáneamente en un programa de computadora.

IPC (**I**nstructions **P**er **C**ycle): En arquitectura de computadoras, el IPC es un término para describir un aspecto del desempeño del procesador, el número promedio de instrucciones ejecutadas por cada ciclo de reloj.

ISA (**I**nstruction **S**et **A**rchitecture): Es el conjunto de instrucciones que la arquitectura de un procesador en particular puede entender y ejecutar.

MIPS (**M**icroprocessor without **I**nterlocked **P**ipelined **S**tages): Es un conjunto de instrucciones para una arquitectura tipo RISC desarrollado en principio por la Universidad de Stanford cuyo objetivo era simplificar el diseño de un procesador.

MROB (**M**onolithic **ROB**): Esquema convencional de la disposición física de un ROB, en donde sus estructuras componentes se encuentran en un solo bloque monolítico en el layout del procesador.

PC (**P**rogram **C**ounter): Es un registro del procesador que contiene la dirección del apuntador que corresponde al valor de la instrucción de la secuencia de programa que está siendo ejecutada.

RAT (**R**egister **A**lias **T**able): Es una unidad funcional del procesador que hace un mapeo de los registros arquitecturales físicos a registros lógicos

ROB (**R**e**O**rder **B**uffer): Estructura funcional dentro de la arquitectura de un procesador superescalar, que tiene la función principal de mantener el estado de cada una de las instrucciones que se encuentra al vuelo dentro del pipeline, permitiendo así una consistencia en la obtención de resultados.

RUU (**R**egister **U**pdate **U**nit): Nombre que en la nomenclatura del simulador SimpleScalar se le da al ROB.

SPEC (**S**tandard **P**erformance **E**valuation **C**orporation): Es una organización sin fines de lucro que tiene dos objetivos, crear un benchmark estándar para medir el rendimiento de las computadoras y publicar los resultados de estas pruebas.

ANEXO A

EJEMPLO DEL USO DEL SIMULADOR SIMPLESCALAR

Para usar el simulador *sim-outorder* del conjunto de simuladores SimpleScalar, la manera más sencilla es invocarlo desde línea de comandos de Linux, de la siguiente manera:

./sim-outorder {-options} executable {arguments}

Como se puede ver, el programa sim-outorder, es ejecutado con opciones, un ejecutable y sus argumentos. La lista de opciones con las que se puede configurar para emular las características de diferentes arquitecturas se muestra en el listado de la figura A-1, e incluyen el número de unidades funcionales simuladas, el número de entradas que contendrá el ROB, etcétera.

Existen opciones como *–config* o *–dumpconfig*, que permiten elegir o modificar un archivo de configuración que servirá para simular una arquitectura en particular, por ejemplo, en este trabajo de Tesis se utilizó un archivo llamado MIPSR10K_cfg, basado en las características de la arquitectura del procesador MIPS R10000.

El ejecutable es el programa de prueba ó benchmark que se utiliza, en este trabajo de tesis se usó un subconjunto de SPEC2000, con los que el simulador trabajará para evaluar el desempeño de dicha arquitectura. Finalmente los argumentos son archivos que el programa de prueba utiliza para probar la arquitectura, por ejemplo el *bechmark* 1644.gzip tiene como argumentos una imagen y archivos de texto que comprimirá, mientras el simulador obtiene estadísticas de la configuración a evaluar.

```
# load configuration from a file
# -config
# dump configuration to a file
# -dumpconfig
# print help message
# -h                              false
# verbose operation
# -v                              false
# enable debug message
# -d                              false
# start in Dlite debugger
# -i                              false
# random number generator seed (0 for timer seed)
-seed                                 1
```

```
# initialize and terminate immediately
# -q                          false
# restore EIO trace execution from <fname>
# -chkpt                     <null>
# redirect simulator output to file (non-interactive only)
# -redir:sim                 <null>
# redirect simulated program output to file
# -redir:prog                <null>
# simulator scheduling priority
-nice                             0
# maximum number of inst's to execute
-max:inst                         200000000
# number of insts skipped before timing starts
-fastfwd                          100000000
# generate pipetrace, i.e., <fname|stdout|stderr> <range>
# -ptrace                    <null>
# instruction fetch queue size (in insts)
-fetch:ifqsize                    4
# extra branch mis-prediction latency
-fetch:mplat                      3
# speed of front-end of machine relative to execution core
-fetch:speed                      1
# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                        bimod
# bimodal predictor config (<table size>)
-bpred:bimod           2048
# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev            1 1024 8 0
# combining predictor config (<meta_table_size>)
-bpred:comb            1024
# return address stack size (0 for no return stack)
-bpred:ras                        8
# BTB config (<num_sets> <associativity>)
-bpred:btb             512 4
# speculative predictors update in {ID|WB} (default non-spec)
# -bpred:spec_update         <null>
# instruction decode B/W (insts/cycle)
-decode:width                     4
# instruction issue B/W (insts/cycle)
-issue:width                      4
# run pipeline with in-order issue
-issue:inorder                false
# issue instructions down wrong execution paths
-issue:wrongpath               true
# instruction commit B/W (insts/cycle)
-commit:width                     4
# register update unit (RUU) size
-ruu:size                        32
# load/store queue (LSQ) size
-lsq:size                         8
# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:l
# l1 data cache hit latency (in cycles)
-cache:dl1lat                     1
```

```
# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:1024:64:4:l
# l2 data cache hit latency (in cycles)
-cache:dl2lat                     6
# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:512:32:1:l
# l1 instruction cache hit latency (in cycles)
-cache:il1lat                     1
# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                      dl2
# l2 instruction cache hit latency (in cycles)
-cache:il2lat                     6
# flush caches on system calls
-cache:flush                  false
# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress              false
# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               18 2
# memory access bus width (in bytes)
-mem:width                        8
# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:l
# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:l
# inst/data TLB miss latency (in cycles)
-tlb:lat                         30
# total number of integer ALU's available
-res:ialu                         2
# total number of integer multiplier/dividers available
-res:imult                        1
# total number of memory system ports available (to CPU)
-res:memport                      2
# total number of floating point ALU's available
-res:fpalu                        2
# total number of floating point multiplier/dividers available
-res:fpmult                       1
# profile stat(s) against text addr's (mult uses ok)
# -pcstat                    <null>
# operate in backward-compatible bugs mode (for testing only)
-bugcompat                    false
```

Figura A-1. Listado de opciones para configurar una arquitectura a simular.

ANEXO B

CÓDIGO QUE MODELA LA ARQUITECTURA PROPUESTA

Como se explicó en el capítulo cuatro de esta tesis, el simulador sim-outorder de SimpleScalar implementa una estructura funcional que realiza las tareas del ROB, llamada RUU. En este trabajo de tesis se propone una subestructura llamada BROB que almacena el valor del IPC de las instrucciones de salto condicional. Dado que el BROB es una subestructura nueva que no se encuentra implementada convencionalmente por el simulador sim-outorder, es necesario realizar modificaciones al código original, escrito en lenguaje C, para modelar una estructura llamada BRUU, como se explico en el capítulo 4.

A continuación se presentan algunos segmentos importantes del código que se escribió para modelar la arquitectura propuesta modificando el código del simulador. En primer lugar cabe mencionar que fue necesario dar de alta las variables que contendrán los valores de las estadísticas calculadas durante la simulación para la subestructura BRUU, como se muestra en la figura B-1.

```
stat_reg_counter(sdb, "RUU_count", "cumulative RUU occupancy",
                 &RUU_count, /* initial value */0, /* format */NULL);
stat_reg_counter(sdb, "RUU_fcount", "cumulative RUU full count",
                 &RUU_fcount, /* initial value */0, /* format */NULL);
stat_reg_formula(sdb, "ruu_occupancy", "avg RUU occupancy (insn's)",
                 "RUU_count / sim_cycle", /* format */NULL);
stat_reg_formula(sdb, "ruu_rate", "avg RUU dispatch rate (insn/cycle)",
                 "sim_total_insn / sim_cycle", /* format */NULL);
stat_reg_formula(sdb, "ruu_latency", "avg RUU occupant latency (cycle's)",
                 "ruu_occupancy / ruu_rate", /* format */NULL);
stat_reg_formula(sdb, "ruu_full", "fraction of time (cycle's) RUU was full",
                 "RUU_fcount / sim_cycle", /* format */NULL);

stat_reg_counter(sdb, "BRUU_count", "cumulative BRUU occupancy",
                 &BRUU_count, /* initial value */0, /* format */NULL);
stat_reg_counter(sdb, "BRUU_fcount", "cumulative BRUU full count",
                 &BRUU_fcount, /* initial value */0, /* format */NULL);
stat_reg_formula(sdb, "bruu_occupancy", "avg BRUU occupancy (insn's)",
                 "BRUU_count / sim_cycle", /* format */NULL);
stat_reg_formula(sdb, "bruu_rate", "avg BRUU dispatch rate (insn/cycle)",
                 "sim_total_insn / sim_cycle", /* format */NULL);
stat_reg_formula(sdb, "bruu_latency", "avg BRUU occupant latency (cycle's)",
                 "bruu_occupancy / bruu_rate", /* format */NULL);
stat_reg_formula(sdb, "bruu_full", "fraction of time (cycle's) BRUU was full"
                 "BRUU_fcount / sim_cycle", /* format */NULL);
```

Figura B-1. Dando de alta estadísticas para la subestructura BRUU.

También se presenta una sección de código correspondiente a la declaración de las variables que modelan a las estructuras RUU y BRUU. Como se puede ver el código mostrado en la figura B-2, las estructuras de hardware correspondientes al ROB y BROB son modeladas en software siendo declaradas como variables tipo estructura, las cuales contienen una serie de campos. El contenido y función de estos campos es explicado con detalle en el capítulo 4 de esta Tesis.

```
struct BRUU_station{
  md_addr_t PC;
  md_addr_t next_PC;
  md_addr_t pred_PC;
};

/* distributed register update unit (RUU) station*/
struct RUU_station {
  /* inst info */
  md_inst_t IR;                       /* instruction bits */
  enum md_opcode op;                  /* decoded instruction opcode */
  int in_LSQ;                         /* non-zero if op is in LSQ */
  int ea_comp;                        /* non-zero if op is an addr comp *
  int recover_inst;                   /* start of mis-speculation? */
  int stack_recover_idx;              /* non-speculative TOS for RSB pred
  struct bpred_update_t dir_update;   /* bpred direction update info */
  md_addr_t addr;                     /* effective address for ld/st's */
  INST_TAG_TYPE tag;                  /* RUU slot tag, increment to
                                         squash operation */
  INST_SEQ_TYPE seq;                  /* instruction sequence, used to
                                         sort the ready list and tag inst
  unsigned int ptrace_seq;            /* pipetrace sequence number */
  int slip;

  /* instruction status flag structures */
  struct dispatched_flag DISPATCHED_FLAG;
  struct issued_flag ISSUED_FLAG;
  struct executed_flag EXECUTED_FLAG;
  struct spec_flag SPEC_FLAG;

  /* branch instruction status flag structure  */
  struct BRUU_station *BRUU_ptr;

  /* output operand dependency list, these lists are used to
     limit the number of associative searches into the RUU when
     instructions complete and need to wake up dependent insts */
  int onames[MAX_ODEPS];              /* output logical names (NA=unused)
  struct RS_link *odep_list[MAX_ODEPS]; /* chains to consuming operations *
  int idep_ready[MAX_IDEPS];          /* input operand ready? */
};
```

Figura B-2. Declarando RUU y BRUU

Finalmente se presenta una sección del código en donde se realiza la inicialización de la estructuras RUU y BRUU. Como se puede ver en el código mostrado en la figura B-3, estas estructuras son inicializadas y se les asigna memoria de acuerdo al tamaño determinado por las variables *RUU_size* y *BRUU_size*, las cuales son opciones que pueden configurarse como se ha explicado en el anexo A.

Variando el valor de *RUU_size* y *BRUU_size* se pueden simular arquitecturas para diferentes esquemas RUU y BRUU, es decir, diferente número de entradas para el ROB y para el BROB.

```c
/* register update unit, combination of reservation stations and reorder
   buffer device, organized as a circular queue */
static struct RUU_station *RUU;         /* register update unit */
static int RUU_head, RUU_tail;          /* RUU head and tail pointers */
static int RUU_num;                     /* num entries currently in RUU */

static struct BRUU_station *BRUU;       /* branch register update unit */
static int BRUU_head, BRUU_tail;        /* BRUU head and tail pointers */
static int BRUU_num;                    /* num entries currently in BRUU */

static void
ruu_init(void)
{
/* allocate and initialize register update unit (RUU) */
  RUU = calloc(RUU_size, sizeof(struct RUU_station));
  if (!RUU)
    fatal("out of virtual memory");

  RUU_num = 0;
  RUU_head = RUU_tail = 0;
  RUU_count = 0;
  RUU_fcount = 0;
/* allocate and initialize branch register update unit (BRUU) */
  BRUU = calloc(BRUU_size, sizeof(struct BRUU_station));
  if (!BRUU)
    fatal("out of virtual memory");

  BRUU_num = 0;
  BRUU_head = BRUU_tail = 0;
  BRUU_count = 0;
  BRUU_fcount = 0;
}
```

Figura B-3. Inicializando RUU y BRUU.

ANEXO C

EJEMPLO DE ESTADISTICAS OBTENIDAS DE LA SIMULACION

El siguiente listado muestra algunas de las estadísticas arrojadas al término de una simulación. Estas estadísticas contienen datos que miden diferentes aspectos del desempeño del procesador, incluyendo el IPC, el CPI, etcétera.

Estos resultados mostrados, pertenecen en particular a la salida de la simulación de una arquitectura con una configuración distribuida que implementa un ROB (llamado RUU en la nomenclatura de SimpleScalar) de 128 entradas, y un BROB (llamado en esta Tesis BRUU para conservar la nomenclatura de SimpleScalar) de 32 entradas. Como se puede ver en el listado de la figura C-1, se presentan las estadísticas para conocer el desempeño del BROB, las cuales fueron añadidas mediante la modificación del código de SimpleScalar, dado que originalmente, esta métrica no está implementada. Las estadísticas más importantes se destacan en el listado presentado en la figura C-1.

```
sim: ** fast forwarding 100000000 insts **
sim: ** starting performance simulation **

sim: ** simulation statistics **
sim_num_insn    200000003 # total number of instructions committed
sim_num_refs   89387562 # total number of loads and stores committed
sim_num_loads     66687400 # total number of loads committed
sim_num_stores   22700162.0000 # total number of stores committed
sim_num_branches  20080580 # total number of branches committed
sim_elapsed_time     344 # total simulation time in seconds
sim_inst_rate      581395.3576 # simulation speed (in insts/sec)
sim_total_insn     204748347 # total number of instructions executed
sim_total_refs 91741699 # total number of loads and stores executed
sim_total_loads            68584673 # total number of loads executed
sim_total_stores 23157026.0000 # total number of stores executed
sim_total_branches  20539104 # total number of branches executed
sim_cycle         146957685 # total simulation time in cycles
sim_IPC                      1.3609 # instructions per cycle
sim_CPI                      0.7348 # cycles per instruction
sim_exec_BW                   1.3932 # total instructions per cycle
sim_IPB                      9.9599 # instruction per branch
Read_DF                  146957685 # Read Access to Dispatched Flag
Read_IF                  146957685 # Read Access to Issued Flag
Read_EF                  146957685 # Read Access to Executed Flag
Read_NSF           146957685 # Read Access to Non-Speculative Flag
Write_DF           204748347 # Write Access to Dispatched Flag
Write_IF                  231072142 # Write Access to Issued Flag
```

Write_EF 204748347 # Write Access to Executed Flag
Write_NSF 40161160.0000 # Write Access to Non-Speculative Flag
W_access_BROB 20539104 # Write Access to Branch ROB structure
IFQ_count 346074133 # cumulative IFQ occupancy
IFQ_fcount 78323908 # cumulative IFQ full count
ifq_occupancy 2.3549 # avg IFQ occupancy (insn's)
ifq_rate 1.3932 # avg IFQ dispatch rate (insn/cycle)
ifq_latency 1.6902 # avg IFQ occupant latency (cycle's)
ifq_full 0.5330 # fraction of time (cycle's) IFQ was full
RUU_count 2774178289 # cumulative RUU occupancy
RUU_fcount 0 # cumulative RUU full count
ruu_occupancy 18.8774 # avg RUU occupancy (insn's)
ruu_rate 1.3932 # avg RUU dispatch rate (insn/cycle)
ruu_latency 13.5492 # avg RUU occupant latency (cycle's)
ruu_full 0.0000 # fraction of time (cycle's) RUU was full
BRUU_count 183613878 # cumulative BRUU occupancy
BRUU_fcount 2257128 # cumulative BRUU full count
bruu_occupancy 1.2494 # avg BRUU occupancy (insn's)
bruu_rate 1.3932 # avg BRUU dispatch rate (insn/cycle)
bruu_latency 0.8968 # avg BRUU occupant latency (cycle's)
bruu_full 0.0154 # fraction of time (cycle's) BRUU was full
LSQ_count 1299311528 # cumulative LSQ occupancy
LSQ_fcount 0 # cumulative LSQ full count
lsq_occupancy 8.8414 # avg LSQ occupancy (insn's)
lsq_rate 1.3932 # avg LSQ dispatch rate (insn/cycle)
lsq_latency 6.3459 # avg LSQ occupant latency (cycle's)
lsq_full 0.0000 # fraction of time (cycle's) LSQ was full
sim_slip 4298431769 # total number of slip cycles
bpred_bimod.lookups 20611854 # total number of bpred lookups
bpred_bimod.updates 20080578 # total number of updates
bpred_bimod.misses 1208779 # total number of misses
bpred_bimod.jr_non_ras_seen.PP 0 # total number of non-RAS JR's seen
bpred_bimod.bpred_jr_non_ras_rate.PP <error: divide by zero> # non-RAS JR addr-pred rate (ie, non-RAS JR hits/JRs seen)
bpred_bimod.ras_hits.PP 1465377 # total number of RAS hits
il1.accesses 214030413 # total number of accesses
il1.hits 205370566 # total number of hits
il1.misses 8659847 # total number of misses
il1.replacements 8659344 # total number of replacements
il1.writebacks 0 # total number of writebacks
il1.invalidations 0 # total number of invalidations
il1.miss_rate 0.0405 # miss rate (i.e., misses/ref)
dl1.accesses 76076616 # total number of accesses
dl1.hits 74143741 # total number of hits
dl1.misses 1932875 # total number of misses
dl1.replacements 1932363 # total number of replacements
dl1.writebacks 1761459 # total number of writebacks
dl1.invalidations 0 # total number of invalidations
dl1.miss_rate 0.0254 # miss rate (i.e., misses/ref)
dl1.repl_rate 0.0254 # replacement rate (i.e., repls/ref)
dl1.wb_rate 0.0232 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate 0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses 12354181 # total number of accesses
ul2.hits 11602966 # total number of hits

```
ul2.misses                    751215 # total number of misses
ul2.replacements              747119 # total number of replacements
ul2.writebacks                674997 # total number of writebacks
ul2.invalidations                  0 # total number of invalidations
ul2.miss_rate                 0.0608 # miss rate (i.e., misses/ref)
ul2.repl_rate          0.0605 # replacement rate (i.e., repls/ref)
ul2.wb_rate           0.0546 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate            0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses              214030413 # total number of accesses
itlb.hits                  214030404 # total number of hits
itlb.misses                        9 # total number of misses
itlb.replacements                  0 # total number of replacements
itlb.writebacks                    0 # total number of writebacks
itlb.invalidations                 0 # total number of invalidations
itlb.miss_rate                0.0000 # miss rate (i.e., misses/ref)
itlb.repl_rate         0.0000 # replacement rate (i.e., repls/ref)
itlb.wb_rate          0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.hits                   89340783 # total number of hits
dtlb.misses                   522767 # total number of misses
dtlb.replacements             522639 # total number of replacements
dtlb.writebacks                    0 # total number of writebacks
dtlb.invalidations                 0 # total number of invalidations
dtlb.miss_rate                0.0058 # miss rate (i.e., misses/ref)
dtlb.repl_rate          0.0058 # replacement rate (i.e., repls/ref)
dtlb.wb_rate            0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.inv_rate         0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base       0x00400000 # program text (code) segment base
ld_text_size          154800 # program text (code) size in bytes
ld_data_base     0x10000000 # program initialized data segment base
ld_stack_size                  16384 # program initial stack size
ld_prog_entry          0x00400140 # program entry point (initial PC)
mem.page_count          2099 # total number of pages allocated
mem.page_mem         8396k # total size of memory pages allocated
mem.ptab_misses        45598 # total first level page table misses
mem.ptab_accesses         2167499438 # total page table accesses
mem.ptab_miss_rate       0.0000 # first level page table miss rate
```

Figura C-1. Listado de estadísticas obtenidas de la simulación.

ANEXO D

SCRIPTS PARA EXTRAER DATOS DE ARCHIVOS DE SIMULACIÓN

Cuando se requieren extraer los datos de un solo archivo para analizar las características de una arquitectura de interés, puede realizarse en forma manual, sin embargo, cuando los datos que se desean recolectar se encuentran en una gran cantidad de archivos, como fue en el caso de este trabajo de Tesis, donde se simularon 378 configuraciones de arquitecturas diferentes, resultó necesario escribir scripts en el *bash* de Linux para poder extraer y organizar los datos encontrados en los 378 archivos de salida.

En la figura D-1, se muestra una sección del script que usa comandos como *grep* y *awk* para buscar, cortar y ordenar los datos correspondientes a las estadísticas de interés obtenidos tras realizar el proceso de simulación.

```
1 #!/bin/sh
2 #Script name: fetch_data
3 #Description: Script for fetching data of interest for analisys from
  simulation results files
4 echo " "
5 echo "Simulation results for gcc benchmark for all configurations of ROB-BROB"
6 echo "Extract data from statistics files for analisys..."
7 echo " "
8 #                Extract IPC and Executed Instructions
9 #Instructions Per Cycle (IPC)
10 grep "instructions per cycle" /home/RaulGO/Programs/3_gcc_results/* | grep
  'sim_IPC\>' | awk 'BEGIN {print "Instructions Per Cycle (IPC):"}''{printf
  "%.4f\t", $2}''END {print"\n\n"}' >> extracted_data/3_gcc_data.dat
11 #Number of Executed Instructions (EI)
12 grep "total number of instructions executed" /home/RaulGO/
  Programs/3_gcc_results/* | grep 'sim_total_insn\>'| awk 'BEGIN {print "Number
  of Executed Instructions (EI):"}''{printf "%d\t", $2}''END {print"\n\n"}' >>
  extracted_data/3_gcc_data.dat
```

Figura D-1. Script para extraer datos de interés del archivo.

En la figura D-2, se muestra el script principal que se encarga de extraer los datos para cada uno de los bencharks, en este caso particular, para los datos de interés correspondientes al subconjunto de SPECint. El uso de estos scripts facilita en gran

medida la capacidad de ordenar y evaluar los datos generados y agiliza la tarea de desplegar estos datos en forma de gráficos o tablas para mejorar la comprensión de dichos resultados.

```
1 #!/bin/sh
2 #Script name: fetch_data
3 #Description: Script for fetching data of interest
4 #for analisys from simulation results files
5 ./fetch_bzip2_data
6 ./fetch_equake_data
7 ./fetch_gcc_data
8 ./fetch_gzip_data
9 ./fetch_mcf_data
10 ./fetch_parser_data
```

Figura D-2. Script principal para ejecutar todos los scripts.