Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería Mecánica Eléctrica

CONSIDERACIONES DE DISEÑO DEL PARALELISMO INTERNO EN *HARDWARE* A NIVEL DE INSTRUCCIÓN EN MICROPROCESADORES

LUIS FERNANDO RUIZ JUAREZ

ASESORADO POR: ING. GUSTAVO A. VILLEDA VÁSQUEZ

GUATEMALA, OCTUBRE DE 2004

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

CONSIDERACIONES DE DISEÑO DEL PARALELISMO INTERNO EN HARDWARE A NIVEL DE INSTRUCCIÓN EN MICROPROCESADORES

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

LUIS FERNANDO RUIZ JUÁREZ
ASESORADO POR: ING. GUSTAVO A. VILLEDA VÁSQUEZ

AL CONFERÍRSELE EL TÍTULO DE
INGENIERO ELECTRÓNICO

GUATEMALA, OCTUBRE DE 2004

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Sydney Alexander Samuels Milson
VOCAL I	Ing. Murphy Olympo Paiz Recinos
VOCAL II	Lic. Amahán Sánchez Álvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

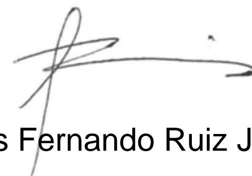
DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Ing. Luis Eduardo Durán Córdova
EXAMINADOR	Ing. Gustavo Adolfo Villeda Vásquez
EXAMINADOR	Ing. Francisco Javier González López
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

CONSIDERACIONES DE DISEÑO DEL PARALELISMO INTERNO EN HARDWARE A NIVEL DE INSTRUCCIÓN EN MICROPROCESADORES

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería Mecánica Eléctrica, con fecha agosto de 2003.



Luis Fernando Ruiz Juárez

AGRADECIMIENTOS:

A DIOS	El alfa y la omega, Creador del Universo, mi aliento de vida, merecedor de toda gloria y majestad.
A mis padres	Por su apoyo diario y comprensión en cada etapa de mi vida; le doy gracias a Dios por tenerlos y este paso es algo que lo hemos logrado juntos.
A mis hermanos	Gracias Manolo por tus consejos y por todos esos momentos que los hemos pasado juntos. Gracias Patty por tu compañía en todo momento.
A mis Tíos	Por estar siempre al tanto, por preocuparse por mí y por hacer posible la culminación de mi carrera.
A mis Primos	Por compartir tantos momentos especiales en mi vida.
A mis compañeros de universidad	Gracias a todos aquellos compañeros que en algún paso de mi vida me los encontré en mi carrera y que compartieron conmigo varios momentos inolvidables. Gracias Helmunt, Ferdi, Kenneth, Poncho, Edgar, David, Ulises, Robin y a todos lo demás.
A mis catedráticos	Les agradezco por haber contribuido en mi formación profesional y en especial al Ing. Gustavo Villeda por sus consejos y su amistad brindada, y el apoyo para poder desarrollar este trabajo de graduación.
A la universidad	Gracias Universidad de San Carlos, gracias por darme la dicha de pertenecer a un grupo selecto de profesionales, por lo que me esforzaré para colocar tu nombre en alto; sea el lugar donde me encuentre.

DEDICATORIAS:

A Jesucristo

Por esa intereseción divina que realizas día con día a la diestra del Padre y por el consuelo de tu Espíritu Santo, por ese toque que le dio vida a mi espíritu y que me hizo verte como una realidad, porque sin ti estaría perdido y nada de esto ni cualquier cosa del mundo valdría la pena.

A mi padre, Ing. José Luis Ruiz

Por todo ese conocimiento transmitido desde mis primeros días de estudio y por fomentarme a tener esa capacidad de análisis característica de todo buen ingeniero y sobre todo por tus consejos, apoyo y comprensión que me has proporcionado durante toda mi carrera.

A mi madre, Estelita

Tú has contribuido de gran manera en el desarrollo de mi carrera y has hecho la tarea de una madre ejemplar, estando presente en todo momento apoyándome incondicionalemente y siempre trantando de ayudarme, es por ello que formas una gran parte de este logro.

A mis hermanos, José Manuel y Silvia Patricia

Es un honor dedicarles este trabajo porque además de ser buenos hermanos, han sido buenos amigos y han sido parte de mi soporte para completar esta carrera, Dios los bendiga siempre.

A mis tíos, Mario y Flory

No solo les agradezco, sino que también les dedico este trabajo, por ser tan especiales y tan buenos conmigo en todo momento y por ser parte muy importante de este logro en mi vida.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES .... **v**

LISTA DE SÍMBOLOS .... **ix**

GLOSARIO .... **xi**

RESUMEN .... **xvii**

OBJETIVOS .... **xix**

INTRODUCCIÓN .... **xxi**

1. CONCEPTOS GENERALES DEL PARALELISMO A NIVEL DE INSTRUCCIÓN (*ILP*) **1**

1.1. Condiciones de paralelismo 1
1.2. Dependencias entre instrucciones 4
1.2.1. Dependencias de datos 5
1.2.1.1 Dependencias *RAW* (*read after write*) 6
1.2.1.2 Dependencias *WAR* (*write after read)* 8
1.2.1.3 Dependencias *WAW* (*write after write*) 9
1.2.2. Dependencias de control 10
1.2.3. Dependencias de recursos 11
1.3. Introducción al *pipeline* de instrucciones. 12
1.4. Paralelismo en *software* y paralelismo en *hardware* 16
1.5. Conceptos de planificación de instrucciones 20
1.5.1. Planificación dinámica. 22
1.5.2. Planificación estática 25

2. CONSIDERACIONES DE DISEÑO E IMPLEMENTACIÓN DEL *PIPELINE* 29

2.1. Etapas y registros del *pipeline*. 29
2.2. Ecuación de rendimiento del pipeline. 34
2.3. Paradas debidas a dependencias (*pipeline hazards*) 38
2.3.1. Rendimiento del *pipeline* con paradas *(stalls)* 39
2.3.2. Dependencias estructurales. 40
2.3.3. Dependencias de datos. 42
2.3.3.1 Minimización de paradas por medio de anticipación *(forwarding)* 44
2.3.4. Dependencias de control (*branch hazards)* 47
2.3.4.1 Reducción de las dependencias de ramificación. 49
2.4. Manejo de excepciones (interrupciones y *traps*). 54
2.5. Extensión del *pipeline* para manipulación de operaciones multiciclo 59
2.5.1. Dependencias y anticipación en *pipelines* de gran latencia. 61
2.5.2. Tratamiento preciso de excepciones. 63

3. CONSIDERACIONES DE DISEÑO EN LA PLANIFICACIÓN DINÁMICA DE INSTRUCCIONES 67

3.1. Planificación dinámica del *pipeline* (*scoreboarding*) 68
3.2. Planificación dinámica usando el algoritmo de Tomasulo 72
3.3. Reduciendo los costos de ramificación con predicción dinámica 77
3.3.1. Búfer de predicción. 78
3.3.1.1 Predictores correlacionados ó de varios niveles. 82
3.4. Entrega de instrucciones. 86
3.4.1. *Branch-target Buffers* 87

3.4.2. Unidades IF integradas 90
3.4.3. *Return Address Predictors* 91
3.5. Ejecución especulativa basada en *hardware*. 92
3.5.1. Consideraciones de diseño 98
3.5.1.1 Renombre de registros. 98
3.5.1.2 Especulación a través de múltiples ramificaciones. 102

4. CONSIDERACIONES DE DISEÑO GENERALES EN BASE A LIMITACIONES DEL ILP EN *HARDWARE* 105

4.1. Consideraciones respecto al modelo de *hardware*. 105
4.2. Limitaciones en el tamaño de ventana. 107
4.3. Efectos en la predicción de saltos y ramificaciones reales. 109
4.4. Los efectos de registros finitos. 112
4.5. Limitaciones del ILP para procesadores realizables. 113

5. HERRAMIENTAS DE SIMULACIÓN COMO BASE PARA EL DISEÑO 119

5.1. Conceptos generales de simuladores utilizados. 119
5.1.1. Tipos de simuladores. 120
5.1.1.1 Simuladores de modelo estático (*static modeling*). 122
5.1.1.2 Simuladores dirigidos mediante trazas (*trace-driven simulation*) 123
5.1.1.3 *Execution-driven simulators* 125
5.1.1.4 Análisis comparativo de las técnicas de simulación 125
5.2. Descripción de algunos simuladores 126
5.2.1. Windlx (*Windows Deluxe Simulator*) 127
5.2.2. Winmips64 132

5.2.3. Dlxview 135
5.2.3.1 Simulación del *pipeline* básico. 137
5.2.3.2 Simulación de planificación dinámica de instrucciones mediante el algoritmo de Tomasulo. 141
5.2.3.3 Simulación de *scoreboarding* 143
5.2.3.4 Superscalar (Superdlx) 145

CONCLUSIONES 151

RECOMENDACIONES 154

BIBLIOGRAFÍA 156

ÍNDICE DE ILUSTRACIONES

FIGURAS

1. Elementos presentes en el procesamiento de una instrucción 1
2. Ejecución secuencial y ejecución paralela de instrucciones 3
3. Clasificación de las dependencias de datos 6
4. Organización de los estados del *pipeline* en función del tiempo. 13
5. Segmentación de instrucciones y su ejecución... 15
6. Paralelismo en *hardware* y paralelismo en software................................. 18
7. Organización típica de un procesador superescalar 23
8. El proceso de compilación 26
9. Unidades funcionales en el pipeline.. 33
10. Conflicto en el acceso a memoria, debido a dependencias estructurales. 41
11. Obstáculos en el *pipeline* debido a dependencias de datos. 43
12. Utilización de la técnica de anticipación (*forwarding*)................................. 44
13. Dependencia de datos de una instrucción de carga. 46
14. Secuenciamiento implícito. 47
15. Planificación del *delay slot*. 53
16. Estructura del *pipeline* con 3 unidades funcionales de punto flotante........ 60
17. Etapa de ejecución multiciclo parcialmente segmentada........................... 61
18. Estructura básica de un procesador con *scoreboard*. 71
19. Estructura de unidad de punto flotante utilizando el algoritmo de Tomasulo.. 73
20. Detalle del algoritmo de Tomasulo.. 76

21. Acceso mediante una porción del PC al *buffer* de predicción....................78
22. Estados en un esquema con 2 bits de predicción......................................80
23. Precisión de predicción para un búfer de 4096 entradas y dos bits por entrada 81
24. Comparación de un búfer de predicción de 2 bits con 4096 entradas y uno de entradas ilimitadas...82
25. Predictor (1,2) que utiliza el comportamiento del último salto.....................84
26. Predictor correlacionado utilizando 2 bits de historia global para escoger 4 predictores para cada dirección de ramificación. 85
27. Comparación entre predictores de 2 bits. 86
28. Estructura de un *branch target buffer* (BTB). 88
29. Pasos involucrados en la manipulación de una instrucción con un BTB. ...89
30. Precisión de predicción para un *return address predictor* operando como pila...92
31. Estructura básica de una unidad de punto flotante usando el algoritmo de Tomasulo especulativo. 96
32. Diagrama de bloques general del proceso de emisión y ejecución de instrucciones. 107
33. Efecto de diferentes esquemas de predicción. 109
34. Utilización de la simulación durante el proceso de diseño........................121
35. Ventana de *pipeline* en el simulador Windlx 127
36. Ventana del diagrama de ciclos de reloj del simulador Windlx. 128
37. Ventana de código del simulador Windlx. 129
38. Ventana de *breakpoints* en el simulador Windlx. 129
39. Ventana de registros del simulador Windlx...130
40. Ventana de estadísticas del Windlx. 131
41. Interfaz gráfica del simulador Winmips64. 133
42. Ventana de configuración de arquitectura del simulador Winmips64. 134

43. Ventana de configuración de memoria y modo de operación en el simulador DlxView 0.9. 137
44. Ventana de configuración del pipeline básico en el simulador................. 137
45. Ventana de control del simulador Dlxview. 138
46. Ventana del *pipeline* básico en el simulador Dlxview............................... 139
47. Ventana del flujos de datos en la unidad entera del *pipeline* del 140
48. Ventana de configuración del algoritmo de Tomasulo 142
49. Ventana de flujo del algoritmo de Tomasulo en el simulador................... 142
50. Ventanta de configuración del *scoreboarding* de Dlxview........................ 143
51. Ventana de ejecución del algoritmo del *scoreboarding* de Dlxview. 144
52. Ambiente en el que se desarrolla Superdlx... 145
53. Modelo de procesador superescalar utilizado por Superdlx. 146
54. Funcionamiento global del manejo de instrucciones de Superdlx............ 147

TABLAS

I. Tipos de planificación. 20
II. Características principales de las diferentes arquitecturas de microprocesadores y algunos ejemplos. 21
III. Estrategias de emisión de instrucciones utilizadas en la planificación dinámica. 25
IV. *Pipeline* básico. 32
V. Parada debida a dependencia estructural. 42
VI. Paradas causadas por la dependencia de datos de una instrucción de carga 47
VII. Parada de un ciclo debido a una ramificación. 48
VIII. Esquema de toma de ramificación. 50
IX. Comportamiento del *pipeline* con esquema *delayed branch*. 52
X. Acciones necesitadas para diferentes tipos de excepciones. 57
XI. Marcador de estado de las instrucciones. 69
XII. Marcador de estado de las unidades funcionales. 70
XIII. Marcador de estado de escritura a registros. 70
XIV. Características de hardware de procesadores de los últimos años. 116
XV. Atributos de varias técnicas de simulación. 126

LISTA DE SÍMBOLOS

//	Paralelismo
←	Transferencia de datos de una fuente de almacenamiento a otra
→	Transferencia de resultado de una operación
∩	Intersección entre el uso elementos de diferentes procesos
{ …. }	Cuerpo del programa
CDB	Bus de datos común entre unidades funcionales
CPI	Ciclos de reloj por instrucción
GHz	Gigahertz
ILP	Paralelismo a nivel de instrucción
Ns	Nanosegundos
PEP	Promedio de ejecución paralela
UF	Unidad funcional

GLOSARIO

ALU Unidad encargada de realizar operaciones del tipo aritmético-lógicas.

Amdahl Ley utilizada para calcular la ganancia del rendimiento debido al perfeccionamiento de alguna porción de la arquitectura.

Antidependencia Dependencia causada por la reutilización del nombre de un registro, también llamada dependencia falsa, correspondiente a los riesgos del tipo WAR y WAW.

Average Promedio de instrucciones realizadas por unidad de tiempo.

Benchmark Software utilizado para medir el rendimiento de un sistema. Algunos de los benchmarks están especialmente diseñados para probar áreas específicas de un sistema, mientras que otros son usados para ejecutar múltiples pruebas y dar un índice de calidad general del rendimiento medido.

Branch delay slot Espacio destinado para una instrucción, donde la instrucción es ejecutada independientemente si la ramificación es tomada o no, utilizado en la técnica de *branch delay*.

Cache	Memoria de rápido acceso, basada en el principio de localidad.
Commit	Proceso clave en la implementación de la especulación de instrucciones, permitiendo que las instrucciones sean ejecutadas fuera de orden pero forzando a que sean entregadas en orden y prevenir cualquier acción irrevocable.
Compilador	*Software* encargado de llevar el código de un lenguaje de alto nivel a lenguaje ensamblador, mediante un análisis sintáctico y semántico.
Estación de reserva	Estaciones encargadas de almacenar las instrucciones que están pendientes de entrar a una unidad funcional.
Excepción	Correspondiente a cualquier tipo de interrupción ocurrida en el flujo normal de instrucciones.
Fetch	Etapa en el procesamiento de la instrucción donde ésta es extraída de la memoria. Corresponde al estado IF (*Instruction Fetch*) del *pipeline*.
Forwarding	Técnica de paralelismo basada en la anticipación con el objeto de minimizar paradas en el *pipeline*, también conocida como *bypassing ó shorcircuiting.*
Hardware	Correspondiente a toda circuitería física interna del microprocesador.

Hyperpipeline Tecnología basada en el principio de *pipeline* la cual esta constituída de un gran número de estados o etapas en el procesamiento de instrucciones.

Idle Estado inactivo en el flujo de instrucciones.

PC (*Program Counter)* Registro encargado de almacenar la dirección de memoria correspondiente a la instrucción a ejecutar.

Pipeline Técnica de paralelismo, basada en la segmentación del proceso de ejecución de instrucciones, permitiendo que diferentes estados de varias instrucciones sean ejecutados al mismo tiempo.

Pipeline interlock *Hardware* encargado de la detección de las dependencias en el *pipeline*, capaz de provocar una parada hasta que la dependencia sea eliminada.

Planificación dinámica Planificación realizada mediante *hardware* en tiempo de ejecución a través de múltiples instrucciones.

Planificación estática Planificación realizada mediante el proceso de compilación con el objeto de optimizar el código para su ejecución, reduciendo la cantidad de *hardware*, respaldándose; sin embargo, en técnicas correspondientes a la planificación dinámica.

Principio de Localidad Principio basado en el hecho de que varios programas tienden a reutilizar datos e instrucciones que fueron utilizados recientemente.

Rate	Inverso del tiempo de ciclo de reloj, usualmente medido en hertz.
RAW (Read After Write)	Dependencia de datos que produce el riesgo la lectura a una fuente antes de que la instrucción predecesora escriba a ésta.
ROB	Búfer de reordenamiento que mantiene el resultado de las instrucciones que han finalizado su ejecución pero no han llevado a cabo el proceso de *commit.*
Scoreboarding	Técnica de paralelismo basada en el uso de un marcador de control, por medio del cual es posible realizar una ejecución especulativa.
Simulación	Herramienta básica en muchos campos de la ciencia y la ingeniería, para complementar (y a veces, sustituir) a otras técnicas como el estudio analítico y el desarrollo de prototipos.
Speedup	Aumento del rendimiento al utilizar una característica adicional en nuestro sistema.
Stall	Parada en el flujo del *pipeline* debida a dependencias.
Superescalar	Procesador capaz de ejecutar múltiples instrucciones por ciclo de reloj con emisión de instrucciones controlada dinámicamente por *hardware*.

Tomasulo Algoritmo basado en la técnica de *scoreboarding* teniendo como gran ventaja el uso de renombramiento de registros. Lleva el nombre en honor a Robert Tomasulo quien lo implemento por primera vez en el procesador IBM 360.

Traza Describe la forma, orden y secuencia en que diferentes elementos (instrucciones, datos y direcciones) son tratados en determinado proceso, para hacer posible la simulación.

Unidad functional (UF) Unidades destinadas a una función especifica en el proceso de ejecución de instrucciones.

VLIW Procesador que ejecuta múltiples instrucciones por ciclo de reloj con emisión de instrucciónes controlada estáticamente mediante *software.*

WAR (Write After Read) Dependencia de datos que produce el riesgo de la escritura a un destino determinado antes de que sea leído por la instrucción predecesora.

WAW (Write After Write) Dependencia de datos que produce el riesgo de la escritura a un operando antes de que la instrucción predecesora escriba éste.

Wire delay Retardo de una señal en una línea conductiva

RESUMEN

Este trabajo de graduación presenta una serie de consideraciones de los aspectos más relevantes empleados en el diseño de paralelismo a nivel de instrucción (ILP) basado en *hardware* dentro de las arquitecturas de microprocesadores empleados en los últimos años, mostrando las opciones utilizadas para emplear al máximo las ventajas que ofrece el ILP y las limitantes a las que está sometido.

El primer capítulo describe un panorama general de las diferentes ventajas, obstáculos y herramientas existentes en la implementación del ILP y la necesidad de implementación; también presenta una introducción a los diferentes tipos de planificación empleados.

El segundo capítulo describe la técnica de *pipeline* utilizada actualmente en cualquier procesador, describiendo sus etapas básicas en cuanto a segmentación, enfatizando en la afección positiva al rendimiento que esta técnica posee y la influencia que existe de obstáculos presentados por medio de dependencias, tanto estructurales como dependencias entre instrucciones. Además, se desarrolla la interdependencia del *pipeline* respecto al manejo de excepciones y operaciones de punto flotante, presentando a lo largo del capítulo, limitantes y soluciones alternativas de acuerdo con los problemas presentados en el diseño de esta técnica.

El tercer capítulo desarrolla el enfoque requerido al planificar dinámicamente el ILP para emitir instrucciones múltiples a pesar de las dependencias que se presenten, empezando el desarrollo desde la técnica básica de *scoreboarding* para luego tratar la técnica ya perfeccionada: el algoritmo de Tomasulo. En éste se introduce la técnica de renombramiento de registros, creando entonces, el concepto de rastrear y procesar las dependencias en instrucciones para permitir la ejecución tan pronto como los operandos estén disponibles y un renombramiento de registros para eliminar las dependencias, siendo lo anterior la clave del procesamiento de diferentes microprocesadores actuales. Posteriormente, se trata sobre herramientas de predicción, permitiendo de esta manera completar el proceso de especulación de instrucciones presentando a la vez varias consideraciones de diseño.

El cuarto capítulo abarca desde un punto de vista general las consideraciones en el diseño de cualquier arquitectura que tenga como base el ILP en *hardware*, enfatizando en este limitantes tecnológicas; además incluye un estudio sobre los logros e impedimentos en el desarrollo de microprocesadores realizables.

El último capítulo describe la simulación como una herramienta útil en el transcurso de las diferentes etapas del diseño, y toma en consideración la relación flexibilidad-precisión. Luego de ello, a lo largo del capítulo, se describen diferentes simuladores que hacen posible desarrollar conceptos básicos de diseño y estudio del comportamiento respecto a la capacidad de las diferentes técnicas utilizadas en el ILP.

OBJETIVOS

- **General**

Desarrollar las consideraciones de diseño necesarias del paralelismo interno en *hardware* a nivel de instrucción en microprocesadores.

- **Específicos**

1. Mostrar un panorama general del desarrollo del paralelismo a nivel de instrucción, para enmarcar correctamente, el diseño del paralelismo en *hardware*.

2. Desarrollar las consideraciones de diseño necesarias en la implementación de *pipeline*, respecto a los diferentes procesos que se ven involucrados, su estructura, así como conocer sus limitaciones.

3. Desarrollar consideraciones de diseño en el momento de planificar dinámicamente las instrucciones respecto a los diferentes procesos implicados, algoritmos, elementos de *hardware*, así como describir algunas de las limitaciones de dicha planificación.

4. Desarrollar consideraciones de diseño generales respecto a limitaciones, implícitas en el paralelismo en *hardware* a nivel de instrucción.

5. Presentar diferentes herramientas de simulación, útiles para el diseño y estudio del comportamiento ante diferentes situaciones que se presentan en algunas de las técnicas utilizadas en el paralelismo en *hardware* a nivel de instrucción.

INTRODUCCIÓN

Una de las tendencias actuales en el diseño de microprocesadores consiste en la ejecución de instrucciones en paralelo de una manera segmentada (*pipeline)*, mediante la implementación del procesamiento fuera de orden utilizando técnicas de predicción, especulación y renombramiento de registros, formando la base de una planificación dinámica.

La planificación dinámica es implementada en la microarquitectura por medio de *hardware* mediante el uso de varias unidades funcionales, estaciones de reserva, bancos de registros y un sinnúmero de búfers y registros intermedios, permitiendo así el mejor aprovechamiento del tiempo, realizando una ejecución en paralelo, sólo tiene en cuenta la disponibilidad de datos y recursos sin tomar el orden del programa.

Uno de los grandes obstáculos para el desarrollo del paralelismo a nivel de instrucción (ILP) por medio de la planificación dinámica, surge de las dependencias entre instrucciones en lo correspondiente a operandos y el control de ramificaciones, además de límites tecnológicos como consecuencia de la miniaturización y consumo de potencia, agregando a esto dependencias estructurales debidas a la cantidad de *hardware* disponible. Son varias las limitantes existentes, así como las soluciones que se han ido desarrollando a través de los años permitiendo la manifestación de diferentes modelos de *hardware*, con ello se debe tomar en cuenta aspectos relacionados con el desarrollo y evaluación para un diseño en particular.

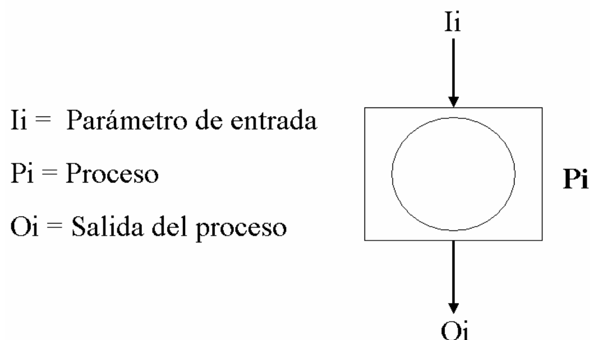
En este documento se exponen varias consideraciones de diseño con base en cualidades de las diferentes técnicas de ILP existentes respecto a *hardware* y en algunas oportunidades se elabora un análisis cuantitativo de acuerdo con resultados obtenidos por medio *software* de evaluación (*benchmarks*); al final la simulación se muestra como una herramienta auxiliar en el diseño.

1. CONCEPTOS GENERALES DEL PARALELISMO A NIVEL DE INSTRUCCIÓN (*ILP*)

1.1. Condiciones de paralelismo

Una de las formas más efectivas para mejorar el desempeño de los microprocesadores es la implementación de técnicas de *hardware* y *software* que permitan la ejecución de más de una instrucción en paralelo durante un sólo ciclo de máquina. Las condiciones necesarias para satisfacer lo expuesto, son los principios de las técnicas de paralelismo implementadas en los microprocesadores. Al analizar el paralelismo a nivel de instrucción, es necesario determinar los diferentes elementos, a los que está sujeto el procesamiento de una instrucción, tal como se muestra en la figura 1.

Figura 1. Elementos presentes en el procesamiento de una instrucción



Todo proceso *Pi* necesita parámetros de entrada, cuyo conjunto denotaremos por *Ii*, éstos parámetros de entrada representan los valores de las variables de entrada relacionadas al proceso. Sin estas variables no es posible que el microprocesador realice su tarea fundamental que es procesar datos. Adicionalmente, la ejecución del proceso genera valores de salida, los cuales son escritos en los registros de propósito general del *CPU*, posiciones de memoria o puertos. Estos valores corresponden a las variables de salida del proceso *Pi* cuyo conjunto denotaremos por *Oi*. Esto implica que los valores de entrada de un proceso denominado *P1* no dependan de los valores de salida de un proceso *P2*, y que los valores de entrada del proceso *P2* no dependan de los valores de salida del proceso *P1*. Además de ello que los parámetros de salida de los procesos *P1* y *P2* no dependan entre sí, es decir:

$$I1 \cap O2 = 0$$

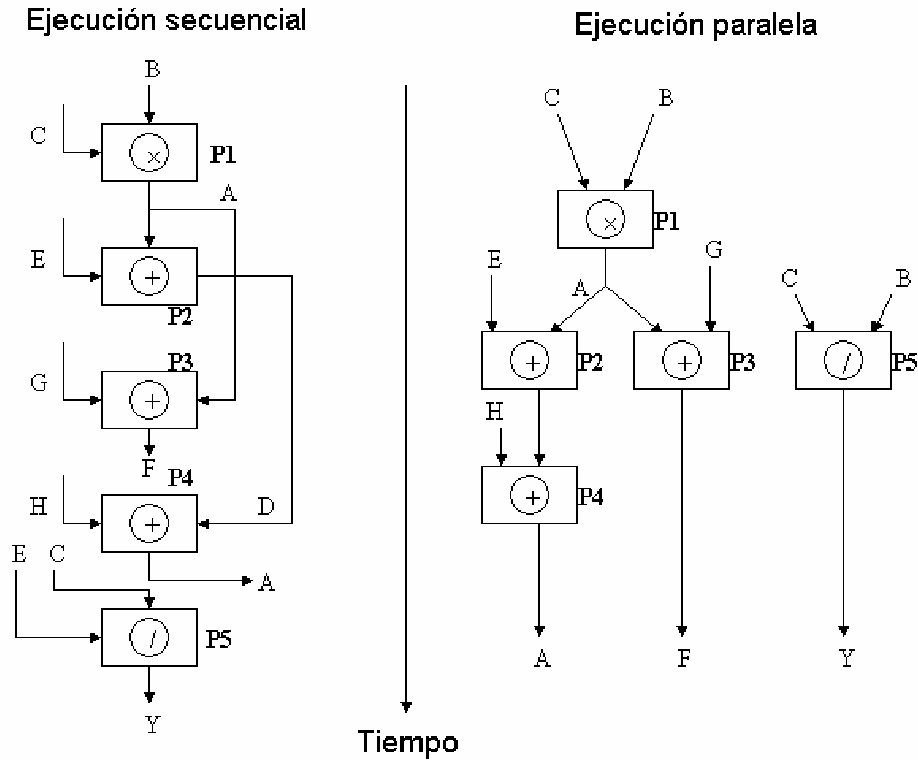
$$I2 \cap O1 = 0$$

$$O1 \cap O2 = 0$$

Para que dos o más procesos puedan ser ejecutados en paralelo deben cumplirse las tres condiciones expuestas anteriormente. Cuando esto ocurre se dice que los procesos *P1* y *P2* pueden ser ejecutados en paralelo, lo cual se denota por *P1||P2*.

Para demostrar la función de los principios antes expuestos en un sistema microprocesado, analicemos el diagrama de interdependencia de procesos que se presenta en la figura 2.

Figura 2. Ejecución secuencial y ejecución paralela de instrucciones



Este diagrama de interdependencia de procesos consta de cinco procesos distribuidos de la siguiente manera:

P1 A = C * B

P2 D = A + E

Se presume que cada proceso tiene un tiempo de ejecución t, el tiempo de ejecución total del programa de manera secuencial es igual a 5t; y el tiempo de ejecución aplicando los principios de paralelismo es de 3t.

Los procesos P1 || P5 y P5 || P2 pueden ser ejecutados en paralelo. En cambio P1 y P2 no pueden ser ejecutados en paralelo ya que "A" es la variable de salida del proceso P1 y la variable de entrada del proceso P2.

Con este ejemplo hemos demostrado que el uso de las condiciones de paralelismo antes expuestas nos sirven para incrementar el *average* de ejecución de instrucciones disminuyendo con esto el tiempo de ejecución de un programa.

1.2. Dependencias entre instrucciones

Al analizar los principios de paralelismo antes expuestos se observa que la ejecución en paralelo de dos o más procesos depende:

- **Del grado de interdependencia de datos.** Se dice que existe interdependencia de datos cuando la ejecución de un proceso depende del resultado de la ejecución de otro proceso.

- **Del grado de interdependencia de control.** Se dice que existe interdependencia de control cuando el orden de ejecución de un programa no puede ser determinado antes de que el mismo sea ejecutado. Un ejemplo de ello es la instrucción condicional de salto en un lenguaje de alto nivel (IF).

- **Del grado de interdependencia de recursos.** Se dice que existe interdependencia de recursos cuando dos o más procesos requieren del mismo recurso de *hardware* al escribir los resultados de salida de cada proceso.

1.2.1. Dependencias de datos

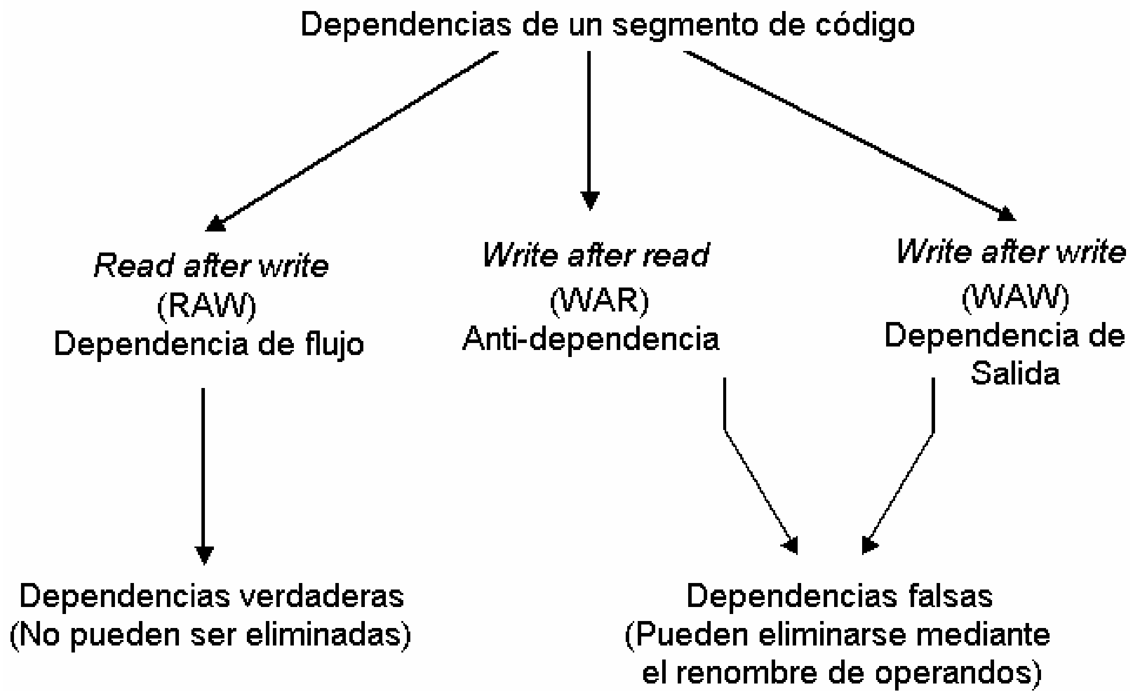
Las dependencias de datos pueden ser clasificadas de acuerdo con los riesgos que se presenten al tratar de trabajar las instrucciones de manera paralela con respecto al orden en el acceso de lectura o escritura generado por las instrucciones; también pueden ser clasificadas de acuerdo a la forma en que sea tratada la dependencia.

Los riesgos que existen al tratar de trabajar paralelamente las instrucciones, se pueden presentar de tres maneras diferentes:

- Leer después de escribir (RAW)
- Escribir después de leer (WAR)
- Escribir después de escribir (WAW)

De los que se pueden eliminar por medio del renombramiento de operandos, los últimos dos (WAR y WAW) generando así, las dependencias falsas, mientras que las dependencias debido a RAW, no pueden ser eliminadas, toman el nombre de dependencias verdaderas, tal como se muestra en la figura 3.

Figura 3. Clasificación de las dependencias de datos



1.2.1.1 Dependencias RAW (*Read After Write*)

Este tipo de dependencias es el más notable, ya que corresponde al grupo de dependencias verdaderas, las cuales no pueden ser eliminadas, en donde una instrucción *j* es dependiente de una instrucción *i* si cumple cualquiera de las siguientes condiciones:

- La instrucción *i* produce un resultado que puede ser usado por la instrucción j.

- La instrucción j tiene una dependencia verdadera sobre la instrucción *k*, y la instrucción *k* tiene una dependencia verdadera sobre la instrucción *i*.

La segunda condición, se basa en que si una instrucción es dependiente de otra sí existe una cadena de dependencias del tipo de la primera condición, entre las dos instrucciones; ésta cadena de dependencias puede ser tan larga, como el programa entero.

Para ejemplificar éste tipo de dependencia, consideremos los siguientes segmentos de código:

Dependencias de carga

load r1, a
add r1, r1 → r2

Dependencias de registros

mul r4, r5 → r1
add r1, r1 → r2

Se puede observar que, en ambos ejemplos, la segunda instrucción es dependiente de la primera instrucción en sus operandos; en el primer ejemplo, el dato “a” debe ser cargado a el registro “r1” para luego ser sumado, mientras que en el segundo ejemplo la multiplicación del contenido de los registros “r4” y “r5” debe ser cargada primero a “r1”, antes de ser sumada, es por ello que en la instancia de estas dependencias, dos instrucciones no se pueden ejecutar al mismo tiempo, ya que se corre el riesgo inminente de leer después de escribir.

1.2.1.2 Dependencias *WAR* (*Write After Read)*

Este tipo de dependencias se presenta cuando dos instrucciones usan el mismo registro o localización de memoria, sin haber flujo de datos o relación, entre las instrucciones asociadas a ese registro o localización de memoria.

Una antidependencia o dependencia WAR entre una instrucción *i* y una instrucción *j* ocurre, en términos generales, cuando una instrucción *j* escribe a un registro o posición de memoria que la instrucción *i* lee, donde el orden original de las instrucciones debe ser preservado para asegurarse que *i* lea el valor correcto; este tipo de antidependencia forma parte del grupo de las dependencias por nombre o dependencias falsas, que al contrario de las dependencias verdaderas, ningún valor es transmitido entre las instrucciones.

Por ejemplo, consideremos el siguiente segmento de código:

mul r2, r3 → r1
add r4, r5 → r2

Según el ejemplo, vemos que en esta dependencia existe el riesgo de escribir a r1 en la primera instrucción, después de leer el resultado de r2 de la segunda instrucción. Esta dependencia, se puede eliminar mediante renombre de registros modificando la segunda instrucción de nuestro segmento de código:

add r4, r5 → r7

Este renombramiento puede ser realizado estáticamente por un compilador o dinámicamente, mediante técnicas de *hardware*.

1.2.1.3 Dependencias WAW (Write After Write)

Al igual que las dependencias del inciso anterior, este tipo de dependencias ocurre cuando dos instrucciones usan el mismo registro o localización de memoria, sin haber flujo de datos o relación, entre las instrucciones asociadas a ese registro o localización de memoria, con la diferencia de que las dependencias WAW o de salida, ocurren entre una instrucción *i* y una instrucción *j* cuando escriben a el mismo registro o posición de memoria, el orden entre las instrucciones debe ser preservado para asegurar que el valor finalmente escrito corresponda a la instrucción *j*.

Si se consideran las siguientes instrucciones:

mul r1, r3 → r2
add r4, r5 → r2

Estas instrucciones pueden ser ejecutadas simultáneamente o ser reordenadas si el registro o posición de memoria de salida es cambiado, en este caso cambiando r2 de la segunda instrucción por r7:

add r4, r5 → r7

Debido a que la dependencia puede ser eliminada mediante el renombramiento de un registro o posición de memoria, sin que exista relación entre las dos instrucciones; este tipo de dependencia, pasa a ser una dependencia falsa.

1.2.2. Dependencias de control

Una dependencia de control, determina el ordenamiento de una instrucción *i*, respecto a una instrucción de ramificación; tanto que la instrucción *i* sea ejecutada en el orden correcto del programa y solamente cuando se deba hacer. Uno de los ejemplos más simples de dependencia de control es la dependencia de la declaración *then* como parte de la declaración *if* en una ramificación. Por ejemplo, en el segmento de código:

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

S1 tiene dependencia de control de p1, y S2 tiene dependencia de control de p2 pero no de p1. En general, hay dos restricciones impuestas por dependencias de control:

1. Una instrucción que depende del control sobre una ramificación no se puede mover antes de la rama, para que su ejecución no se controle a lo largo de esta. Por ejemplo, no podemos tomar una instrucción desde la porción del programa correspondiente al *then* de una declaración *if* y mover esta antes de dicha declaración.

2. Una instrucción que no depende del control sobre una ramificación no se puede mover después de la rama, para que su ejecución se controle junto con esta. Por ejemplo, no podemos tomar una instrucción antes del declaración *if* y moverla dentro de la porción del *then* del programa.

Preservando las dependencias de control es una manera de mantener el programa en orden y que se efectúe correctamente, sin embargo, los microprocesadores actuales no trabajan de una manera conservativa, respecto a las dos restricciones anteriores, sino que se basan mediante técnicas de especulación y predicción, para anticipar, para poder trabajar lo más paralelamente posible los diferentes procesos involucrados; e este motivo de estudio en los capítulos posteriores.

1.2.3. Dependencias de recursos

Este tipo de dependencias ocurre cuando dos instrucciones utilizan el mismo recurso, debido a limitaciones en *hardware* con respecto al número de unidades disponibles para realizar un proceso determinado.

Consideremos el siguiente fragmento de código:

div r1, r2 → r3
div r4, r2 → r5

La dependencia radica en la cantidad divisores insuficientes para emitir ambas instrucciones en paralelo, asumiendo que sólo existiera una unidad divisora. Entre otros posibles recursos en disputa se pueden mencionar a los buses, unidades de ejecución, búfers (almacenamiento temporal), en fin, todo aquel recurso donde por alguna circunstancia de nuestro código, resulte insuficiente para realizar instrucciones en paralelo.

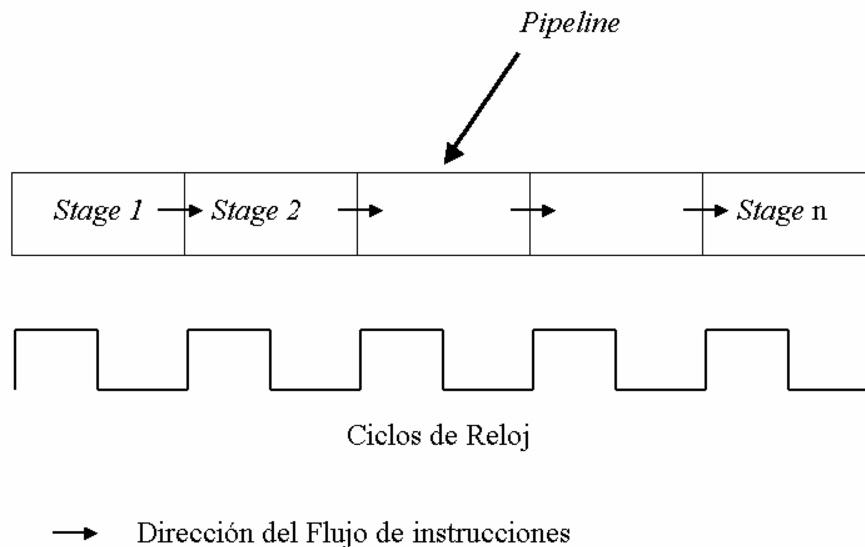
1.3. Introducción al *Pipeline* de instrucciones.

Pipelining es una técnica donde múltiples instrucciones son realizadas al mismo tiempo, tiene como ventaja, el paralelismo existente entre las acciones necesarias para ejecutar una instrucción, logrando así un menor tiempo de ejecución.

Un *Pipeline* es como una línea de ensamblaje. En una línea de ensamblaje de automóviles, existen muchos pasos, donde cada uno contribuye para la construcción total del automóvil, cada paso opera en paralelo con otros pasos de un automóvil diferente. Ahora bien, en el *pipeline* de una computadora, cada paso del *pipeline* completa una parte de una instrucción y así como en la línea de ensamblaje, diferentes pasos completan diferentes partes de diferentes instrucciones en paralelo, cada uno de estos pasos es llamado *pipe stage* ó *pipe segment*.

Cada estado o segmento, recibe sus entradas de la etapa anterior y proporciona las entradas de la etapa siguiente, tal como es mostrado en la figura 4.

Figura 4. Organización de los estados del Pipeline en función del tiempo



En una línea de ensamblaje de autos, el rendimiento específico es definido por el número de autos por hora y es determinado por la rapidez con que sale un auto completo de la línea de ensamblaje; asimismo, el rendimiento de una instrucción *pipeline* se determina por la rapidez con que una instrucción completa sale del *pipeline*.

En el *pipeline* todos los estados actúan al mismo tiempo, donde el largo de un ciclo del procesador es determinado por el tiempo requerido por el estado más lento, este es usualmente un ciclo de reloj; aunque algunas veces son utilizados dos ciclos de reloj y muy rara vez una cantidad mayor de ciclos.

Uno de los objetivos principales del diseñador de microprocesadores es balancear el tiempo de ejecución de cada estado del *pipeline*; al lograr este, el tiempo por instrucción en un procesador *pipeline* asumiendo condiciones ideales es igual a:

$$\frac{\text{Tiempo por instrucción sobre un procesador sin } \textit{pipeline}}{\text{Número de estados del } \textit{pipeline}}$$

Sobre estas condiciones, el aumento de productividad es proporcional al número de estados (*pipe stages)*, tal como una línea de ensamblaje de automóviles con *n* estados, que puede producir automóviles tan rápido como *n* veces. Usualmente, los estados no están perfectamente balanceados, además de ello, pipelining posee otras desventajas.

La técnica de *pipeline* reduce el tiempo promedio de ejecución por instrucción, dependiendo la base; la reducción puede ser vista como una disminución del número de ciclos de reloj por instrucción (*CPI*), como decremento en el tiempo del ciclo de reloj o como una combinación. Si tomamos el primer punto de vista de que un procesador toma múltiples ciclos de reloj por instrucción, el *pipeline* es visto como una reducción del CPI; este éste el punto primordial que debe tomarse.

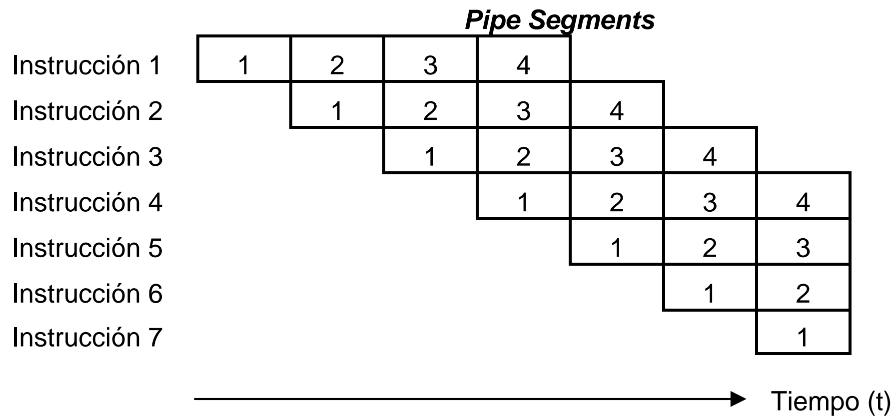
En el segundo punto de vista un procesador toma un largo ciclo de reloj por instrucción, entonces el *pipeline* disminuye el tiempo del ciclo de reloj. Aquí el *pipeline* es una técnica implementada que explota el paralelismo en un flujo secuencial de instrucciones.

Las características del *pipeline* son:

- Subdivisión de la ejecución en fases (segmentos ó estados).
- Cada fase se asigna a una etapa hardware diferente.
- Fases de igual duración (balanceadas).
- Una etapa recibe sus entradas de la etapa anterior y proporciona las entradas. de la etapa siguiente.
- Etapas sincronizadas con la señal de reloj.
- Duración de una fase: un ciclo de reloj, algunas veces dos y rara vez más.

En la subdivisión de la ejecución por fases (*pipe segments*) al trabajar con varias instrucciones, se tiene la ventaja del paralelismo existente entre las acciones necesarias para ejecutar una instrucción, que cada fase ejecuta una acción diferente, así se pueden trabajar diferentes fases de distintas instrucciones paralelamente, tal como se muestra en la figura 5.

Figura 5. Segmentación de instrucciones y su ejecución.



Según el ejemplo de la figura 5, vemos que la ejecución de instrucciones ha sido segmentada en cuatro diferentes fases numeradas de 1 a 4, donde se puede observar respecto a la línea de tiempo, que al ejecutarse el primer segmento de la instrucción 4, justamente en el mismo tiempo se ejecutan los segmentos 4, 3 y 2 de las tres instrucciones anteriores, por orden de aparición, respectivamente, así se logra la ejecución de cuatro instrucciones en paralelo, justamente desde ese momento en adelante, esto se logra únicamente en un caso ideal.

Los problemas para lograr una ejecución ideal y un aprovechamiento máximo del tiempo utilizando la técnica de *pipeline* radican en la existencia de dependencias y sus riesgos entre instrucciones, así como en el uso efectivo de cada segmento de los diferentes tipos de instrucciones.

1.4. Paralelismo en *software* y paralelismo en *hardware*

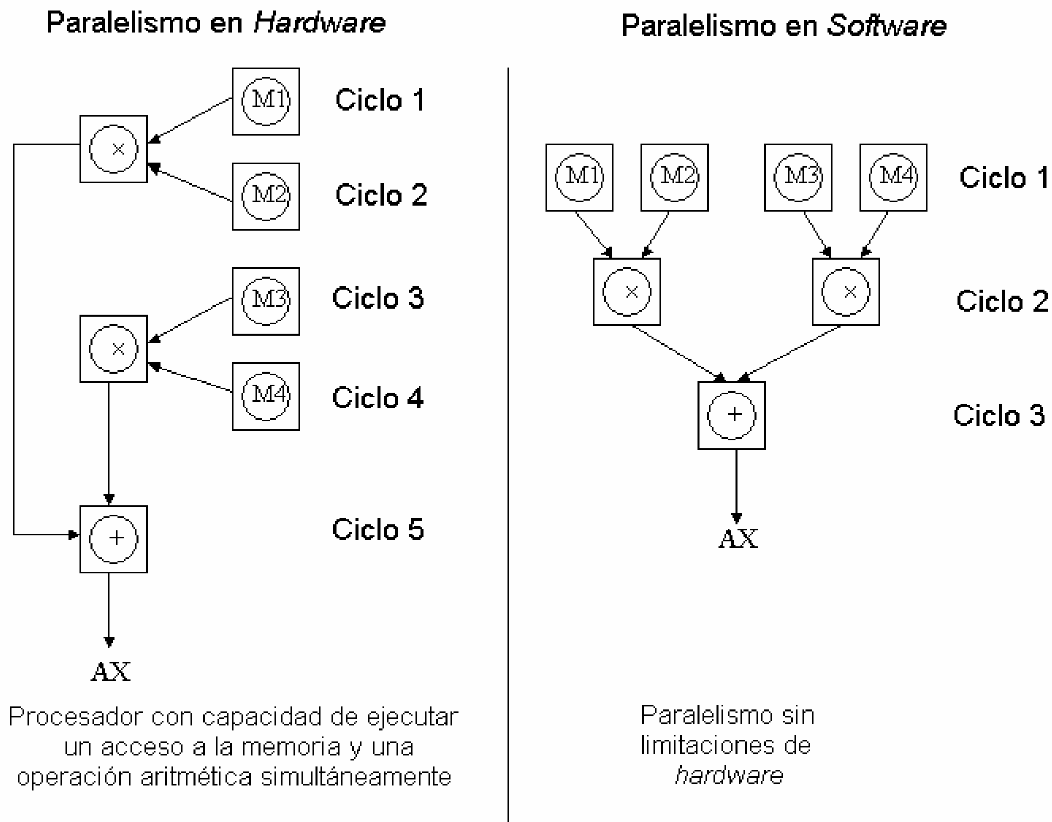
Paralelismo en software es la capacidad de ejecutar un programa paralelamente sin tomar en cuenta el hardware con que va ser ejecutado. El paralelismo en *software* es considerado como el caso ideal de la ejecución de los procesos, ya que no toma en cuenta las limitantes del *hardware* con que el mismo va ser ejecutado; mientras que paralelismo en *hardware* es la capacidad de ejecutar un programa paralelamente tomando en consideración el *hardware* con que va a ser ejecutado.

Un ejemplo claro se observa, en el modo de ejecución del siguiente comentario de programa:

Mov Ax, [A]	Ax ← Mem[A]
Mov Bx, [B]	Bx ← Mem[B]
Mul Ax, Bx	Ax ← Ax x Bx
Mov Cx, [C]	Cx ← Mem[C]
Mov Dx, [D]	Dx ← Mem[D]
Mul Cx, Dx	Cx ← Cx x Dx
Add Ax, Cx	Ax←Ax + Cx

El programa presentado está compuesto por siete instrucciones (cuatro instrucciones de movimiento de una posición de memoria a un registro, dos instrucciones de multiplicación de registros y una instrucción de suma de registros). El gráfico de interdependencia de este grupo de instrucciones se puede observar en la figura 6.

Figura 6. Paralelismo en hardware y paralelismo en software



El diagrama de paralelismo en software representa el caso ideal con que dicho programa puede ser ejecutado; la ejecución de las siete instrucciones se realiza solamente en tres ciclos de máquina. Por otro lado, las limitantes que genera la ejecución de este mismo programa con un *hardware* en particular (procesador con capacidad de ejecutar un acceso a la memoria y una operación aritmética simultáneamente) obteniendo cinco ciclos de máquina para ejecutar el programa.

Tomando como base el ejemplo anterior, la ejecución paralela de un programa se mide mediante el parámetro conocido como Promedio de Ejecución Paralela (PEP). Este parámetro se define como la relación entre el número de procesos del programa y el número de ciclos de máquina realizados en su ejecución. Su expresión matemática es:

$$PEP = \frac{\text{No. de Procesos}}{\text{No. de Ciclos de Máquina}}$$

Por consiguiente, el promedio de ejecución paralela en *software* para este ejemplo es: 7/3 = 2.33 y el promedio de ejecución paralela en *hardware* es: 7/5 = 1.4, donde es notable observar que el promedio de ejecución paralela en *software* es aproximadamente un 67% mayor que el PEP en *hardware.*

En el paralelismo en *hardware* y en *software* es importante hacer notar que ambos están sujetos a las dependencias y riesgos en la ejecución paralela de instrucciones, mencionadas con anterioridad, donde el concepto de paralelismo en *software,* no toma en cuenta las dependencias de recursos, limitantes físicas y arquitectura de nuestro procesador, por lo que posee como única limitante las dependencias de datos y de control entre instrucciones, no pudiendo ser eliminadas las dependencias *RAW*.

1.5. Conceptos de planificación de instrucciones

El concepto de planificación nace de acuerdo con la forma como se ejecute la detección y resolución de dependencias entre instrucciones, como mejora en el rendimiento de los microprocesadores.

El enfoque de la planificación en un microprocesador, se puede desarrollar, ya sea, por medio de herramientas de *hardware* o *software* (compiladores) o una combinación de ambas; toma los nombres de planificación dinámica ó planificación estática, respectivamente; lo anterior se muestra en la siguiente tabla.

Tabla I. Tipos de planificación

Tipo de Planificación	Planificación Realizada por	¿Cómo recibe le código el procesador?
Estática (Reforzada por la optimización del código).	Compilador	Libre de dependencias conflictivas y optimizado para su ejecución paralela.
Dinámica (pura)	Procesador	No optimizado y con dependencias
Dinámica (reforzada por el compilador)	Procesador y Compilador	Optimizado para ejecución paralela pero con dependencias

El enfoque de planificación dinámica (*hardware-intensive)* es dominante en el mercado de servidores y computadoras personales y es utilizado en una amplio rango de procesadores, incluyendo el Pentium III y 4; el Athlon; el MIPS R10000/12000; el Sun UltraSPARC III; el Power PC 603, G3 y G4; y el Alpha 21264, entre otros. Mientras que el enfoque estático (*compiler-intensive*), ha visto mayor adopción en el mercado de microprocesadores de aplicación especifica (*embedded processors*), una de las excepciones es la arquitectura IA-64 con el procesador Itanium desarrollado por Intel, tal como se muestra en la siguiente tabla.

Tabla II. Características principales de las diferentes arquitecturas de microprocesadores y algunos ejemplos

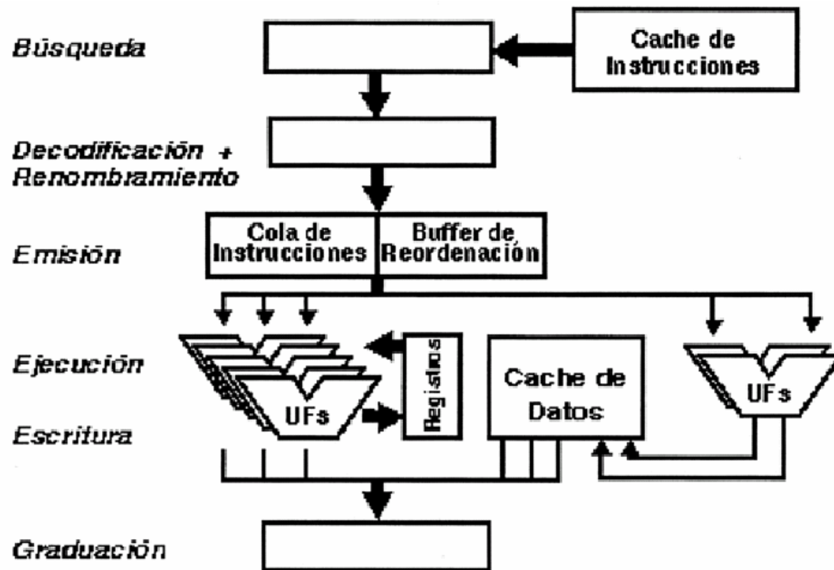
Arquitectura	Enfoque de estructura	Detección de dependencias	Planificación	Ejecución de instrucciones	Ejemplos
Superescalar (estático)	dinámico	*hardware*	Estática	en orden	Sun UltraSparc II/III
Superescalar (dinámico)	dinámico	*hardware*	Dinámica	Algunas fuera de orden	IBM Power 2
Superescalar (especulativo)	dinámico	*hardware*	dinámica con especulación	Fuera de orden con especulación	Pentium III/IV MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW/LIW	estático	*software*	Estática	Sin riesgos entre paquetes emitidos	Trimedia, i860
EPIC	Muy estático	Muy orientada a *software*	Muy estática	Dependencias marcadas por el compilador	Itanium I y II

1.5.1. Planificación dinámica

La mayoría de los procesadores actuales son procesadores *segmentados* con una organización *superescalar*. Un procesador segmentado es aquel que divide la ejecución de una instrucción en diversas etapas, de forma que tan pronto como una instrucción finaliza una etapa *n*, a la vez que està realizando la etapa *n+1*, la siguiente instrucción puede llevar a cabo la etapa *n,* así se usa la técnica de *pipeline*, consiguiendo ejecutar varias instrucciones a la vez.

Un procesador superescalar es capaz de procesar más de una instrucción simultáneamente en cada una de las etapas (el símil sería tener varias cadenas de montaje ó varios *pipelines*), de esta manera se aumenta el paralelismo a nivel de instrucción. La mayoría de los procesadores superescalares actuales disponen de un mecanismo de planificación dinámica de instrucciones. Es decir, las instrucciones no se ejecutan en el orden en que aparecen en el programa sino que el *hardware* decide en que orden se ejecutan para un mayor rendimiento. Por eso, a estas máquinas también se les denomina procesadores con *ejecución fuera de orden*. La siguiente figura, muestra una organización típica de un procesador superescalar.

Figura 7. Organización típica de un procesador superescalar



Fuente: González, Antonio. Tendencias en la microarquitectura de los procesadores. Pág. 3.

El procesador dispone de una memoria cache de instrucciones, donde en cada ciclo, la unidad de búsqueda va a traer varias instrucciones. A continuación las instrucciones son decodificadas y a la vez los operandos registro son renombrados. El renombramiento de registros tiene por objetivo que los operandos destino de todas las instrucciones cuya ejecución ha empezado, pero no finalizado tengan un identificador diferente. De esta forma se eliminan las llamadas *dependencias de nombre (WAR y WAW).* Al reducir el número de dependencias, aumenta la cantidad de paralelismo que se puede explotar.

Tras la etapa de decodificación las instrucciones son *despachadas* (*dispatched*) a la *cola de instrucciones* y al *buffer de reordenación*. Las instrucciones permanecen en la cola de instrucciones hasta que pueden iniciar su ejecución, mientras que en el buffer de reordenación están hasta que el procesador está seguro de que su ejecución es correcta y no debe deshacerse. Los procesadores actuales realizan muchas actividades de forma *especulativa*, es decir, sin estar completamente seguros de si son correctas. Esto se hace para adelantar dichas actividades pero, en caso de fallo de especulación, éstas deben deshacerse.

La cola de instrucciones es inspeccionada en cada ciclo por un *hardware* que se encarga de determinar que instrucciones pueden iniciar la ejecución, o *emitirse*. Para ello, una instrucción debe de tener todos sus operandos, y los recursos que necesita deben de estar libres.

Tras emitirse las instrucciones son ejecutadas y al finalizar escriben el resultado. Tras ello la instrucción es retenida en el procesador hasta que se está seguro de que su ejecución no debe deshacerse. Para ello, las instrucciones abandonan el procesador (se *gradúan)* en orden secuencial.

De lo anterior, se puede observar, que una planificación dinámica explota el nivel de paralelismo, mediante el enfrentamiento al problema de las dependencias, tal como se muestra en la siguiente tabla:

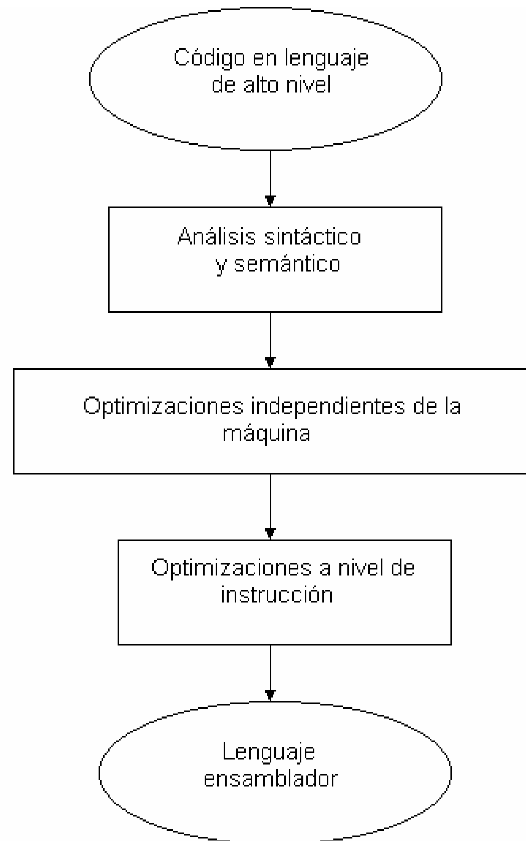
Tabla III. Estrategias de emisión de instrucciones utilizadas en la planificación dinámica

Qué hacer con:	**Solución alternativa, planificando dinámicamente**
Dependencias falsas (*WAR,WAW*) en registros.	Evitar Detenciones: Renombre de registros.
Dependencias de control	Especulación en saltos (no esperar por la resolución del salto).
Bloqueos en la emisión (debido a dependencias de datos).	Emisión de estaciones de reserva. Si se produce un bloqueo las instrucciones independientes pueden emitirse aunque halla alguna instrucción anterior bloqueada (ejecución fuera de orden).

1.5.2. Planificación estática

La principal característica de estas arquitecturas es que el lenguaje máquina incorpora mecanismos para que el compilador pase información al *hardware* sobre determinadas características del programa. Por ejemplo, el compilador puede identificar que instrucciones son independientes. De esta manera el hardware puede ser mucho más sencillo, lo que mejora sustancialmente los problemas de tiempo de verificación y consumo de energía.

Figura 8. El proceso de compilación



El proceso de compilación general, de cualquier programa realizado en un lenguaje de alto nivel, para generar su código ensamblador respectivo, debe estar constituido de por lo menos tres pasos (tal como se muestra en la figura 8):

1. **Análisis sintáctico y semántico del código en alto nivel:** en donde se lleva a cabo la descomposición en instrucciones y expresiones, revisión de errores, generación de tabla con los nombres de los objetos y símbolos.

2. **Optimizaciones independientes de la máquina:** donde se realiza la simplificación de expresiones aritméticas.

3. **Optimizaciones a nivel-instrucción:** donde se genera el código ensamblador final.

En el último paso de compilación, involucra la planificación estática de instrucciones, que en contraste con la planificación dinámica, el compilador tiene la completa responsabilidad de crear un paquete de instrucciones que puede ser simultáneamente emitido, donde el *hardware* no realiza ninguna decisión acerca de la emisión múltiple de las instrucciones del paquete.

En la planificación estática son utilizadas diferentes técnicas de compilación, donde se reorganiza el código generado para que las instrucciones independientes se ejecuten cuanto antes y se de una optimización de la utilización del *hardware*, proporcionando así un código paralelo optimizado, en base a dependencias falsas, de control y de recursos.

A pesar de que el compilador es el encargado de realizar varias tareas, es necesario implementar algunas de las técnicas de hardware como lo es el uso indiscutible del *pipeline*, el cual recibe un código optimizado, así se aprovecha al máximo. Además, en técnicas propias de la planificación estática como lo es, la predicción estática de ramificaciones con el objetivo de eliminar las dependencias de control, puede ser también apoyada por predictores dinámicos a base de hardware, así como también lo es el uso de técnicas de especulación, todo ello con el objeto de optimizar y explotar en mayor manera el ILP.

2. CONSIDERACIONES DE DISEÑO E IMPLEMENTACIÓN DEL *PIPELINE*

Es importante tomar en consideración que todo diseño posee sus ventajas y distintas cualidades, que permiten adoptar un criterio adecuado en el momento de implementarlo o plantear una solución al problema que se presente.

La técnica de *pipeline*, ha sido desarrollada en diferentes versiones, sin embargo es utilizada en cualquier procesador, donde el número de unidades funcionales segmentadas, definen el número de etapas, dependiendo del tipo de instrucción.

2.1. Etapas y registros del *pipeline*.

En el capítulo uno se mostró la similitud existente entre la técnica de pipeline y una línea de montaje; estas tienen en común diferentes etapas, donde el traslape y la interdependencia entre varias de ellas, permite la ejecución paralela de etapas de diferentes instrucciones. Cada etapa realiza un tratamiento diferente, que es necesario, para la ejecución de nuestra instrucción.

Los procesadores previos al 8086 de Intel estaban limitados en su desempeño por la necesidad de realizar los dos pasos principales de ejecución del procesador: *Fetch/Execute*, en forma secuencial; es decir, no se puede ejecutar una instrucción hasta que se traiga de memoria (*Fetch*); y no podían traerse instrucciones de memoria mientras ejecutaba una instrucción, pues el procesador estaba ocupado. Por lo que un alto porcentaje del tiempo, el procesador estaba haciendo *Fetch*, cuando su función debiera ser ejecutar las instrucciones. La capacidad de ejecutar instrucciones sólo se ocupaba en un bajo porcentaje.

Para solucionar esto, fue implementada la arquitectura *pipeline Fetch/Execute*, en la que simplemente se divide la tarea en dos secciones: una encargada del *Fetch* (BIU), y otra del *Execute* (EU). De esta manera, existen circuitos separados para cada función, los cuales trabajan en paralelo. Si bien el proceso aún es secuencial, solamente al principio donde se requiere desperdiciar tiempo en el *Fetch*. A partir de ahí, *Fetch* va adelante del *Execute*, y trae instrucciones al procesador mientras este ejecuta las anteriores, constituyendo así un pipeline de 2 etapas.

Con el paso del tiempo, se ha ido implementando la técnica de *pipeline* de diferentes maneras, como lo vemos en el microprocesador Pentium IV con su tecnología de *hyperpipeline*, la cual está constituida de 22 a 24 etapas, mientras que en un Pentium con arquitectura P6 (Pentium Pro, Pentium II, Pentium III y Celeron), el número de etapas es acerca de 10 a 14. Por lo que dicha técnica ha obtenido un significado y aplicación global en el marco de los microprocesadores en donde los conceptos y criterios desarrollados a un microprocesador, son significativamente similares y aplicables a otros procesadores.

Un *pipeline* básico, está conformado de cinco etapas, donde cada instrucción puede ser implementada en al menos cinco ciclos de reloj, por lo que dependiendo del tipo de instrucción, puede ser tratada en etapas diferentes:

1. ***Instruction fetch cycle* (IF):** en este primer ciclo, la tarea de ésta etapa es traer la instrucción de memoria de acuerdo al contador de programa (PC).

2. ***Instruction decode/register fetch cycle* (ID):** en esta etapa, se decodifica la instrucción, se preparan los registros según el código fuente y se verifican registros, si hubiese una posible ramificación.

3. ***Execution/effective address cycle* (EX):** en este ciclo, la ALU opera sobre los operandos preparados en el ciclo anterior, ejecutando una de tres funciones, dependediendo del tipo de instrucción:

    - Referencia de Memoria: La ALU suma el registro base y el *offset* para formar una dirección efectiva.
    - Instrucción de ALU de Registro a Registro: La ALU ejecuta la operación especificada por el código sobre los valores leídos del registro fuente.
    - Instrucción de ALU de registro inmediato.

4. ***Memory access* (MEM):** si la instrucción es una carga, la memoria hace una lectura usando la dirección efectiva calculada en el ciclo anterior. Si la instrucción es un almacenamiento en memoria, se escriben los datos desde el registro a memoria, usando la dirección efectiva.

5. ***Write-back cylcle* (WB):** esta etapa es utilizada ya sea en una instrucción de ALU de registro a registro o en una instrucción de carga. Escribe el resultado dentro del registro si éste viene de la memoria del sistema (para una instrucción de carga) ó desde la ALU (para una instrucción de ALU).

Cada uno de los ciclos corresponde a una etapa del pipeline (*pipe stage*), como resultado de esto, el patrón de ejecución se puede trazar como en la siguiente tabla:

Tabla IV. *Pipeline* básico

	Número de ciclos de reloj								
Número de instrucción	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrucción I	IF	ID	EX	MEM	WB				
Instrucción i+1		IF	ID	EX	MEM	WB			
Instrucción i+2			IF	ID	EX	MEM	WB		
Instrucción i+3				IF	ID	EX	MEM	WB	
Instrucción i+4					IF	ID	EX	MEM	WB

IF = instruction Fetch ID= instruction decode EX= execution
MEM= Memory Access WB= write back

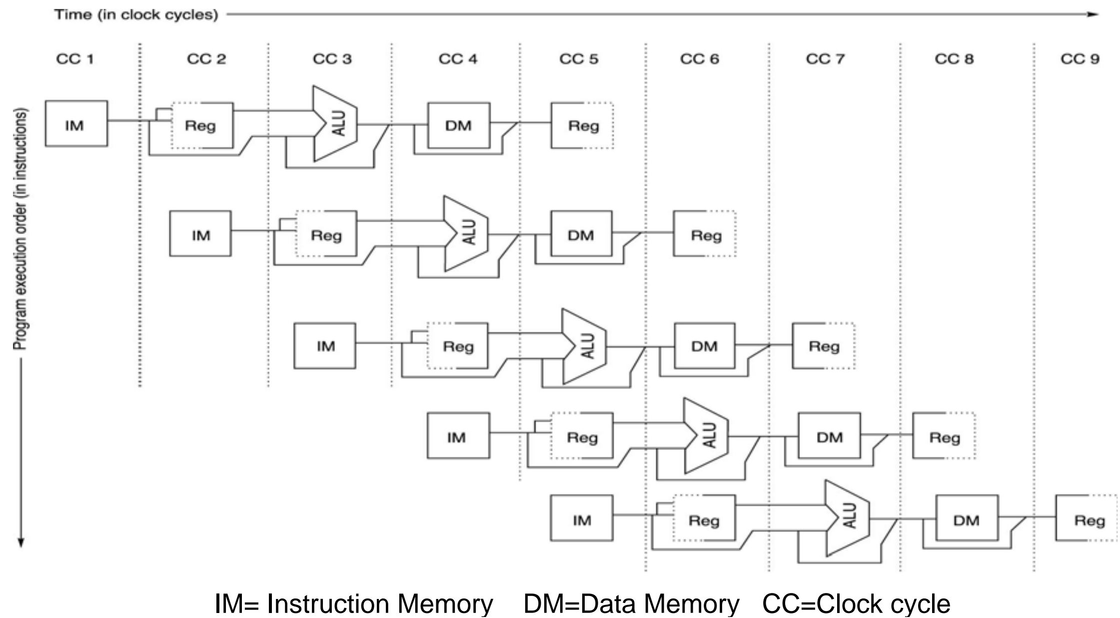
Cada instrucción toma cinco ciclos de reloj para ser completada, durante cada ciclo de reloj el *hardware* inicializará una nueva instrucción y será ejecutada alguna etapa de cinco instrucciones diferentes.

Para conocer las consideraciones que se deben de tomar en la implementación, se debe determinar que sucede en cada uno de los ciclos del procesador.

Por ejemplo, debemos estar seguros que dos diferentes operaciones no utilicen los mismos recursos en el mismo ciclo de reloj, tal es el caso de que si poseemos en nuestro procesador una sola ALU no podremos calcular al mismo tiempo una operación de suma con el cálculo de una dirección efectiva. De esta manera debemos garantizar que el traslape de instrucciones en el *pipeline* no cause cierto conflicto.

Diferentes unidades funcionales interactúan, en cada estado del pipeline, tal como se puede observar en la figura 9. En el quinto ciclo se ejecutan diferentes segmentos de cinco instrucciones, este es el número máximo de instrucciones ejecutadas.

Figura 9. Unidades funcionales en el *pipeline*



IM= Instruction Memory DM=Data Memory CC=Clock cycle

Fuente: Elsevier Science (USA) 2003

El tipo de instrucción y sus dependencias respecto a otras, determinará el número de unidades funcionales utilizadas, donde los registros intermedios juegan el rol de acarrear datos para una instrucción desde un estado a otro.

La arquitectura de nuestro sistema de procesador influirá en la capacidad de ejecución paralela de los segmentos de las instrucciones, como lo es el caso de la utilización de memorias separadas de datos e instrucciones, donde se elimina el conflicto de acceso paralelo entre el *fetch* y el acceso de datos a memoria, actualmente realizado por caches separadas.

La administración adecuada en el acceso de lectura y escritura a los registros internos del procesador dará una mejor implementación del pipeline, tal es el caso que sucede cuando se ejecutan al mismo tiempo los estados ID y WB para un mismo registro, este es un estado de lectura y el otro de escritura respectivamente, se opta por realizar una división de tiempo a medio ciclo para realizar la lectura y escritura secuencialmente en un solo ciclo.

2.2. Ecuación de rendimiento del pipeline.

El pipelining incrementa el rendimiento de una CPU a través del número de instrucciones completadas por unidad de tiempo, pero esto no reduce el tiempo de ejecución de una instrucción individual.

Esencialmente, todas las computadoras son construidas usando un reloj a un *rate* constate. Esos eventos de tiempo discreto son llamados *ticks, clock ticks, clock periods, clocks, cycles o clock cycles.* Los diseñadores de computadoras refieren el tiempo de un período de reloj (*clock period)* por su duracion (ej. 1ns) o por su *rate* (ej. 1 GHz). El tiempo de CPU para un programa puede entonces ser expresado de dos maneras:

Tiempo de CPU = Ciclos de reloj de CPU para un programa X Tiempo del ciclo de reloj

O como:

$$\text{Tiempo de CPU} = \frac{\text{Ciclos de reloj de CPU para un programa}}{\textit{Rate} \text{ del reloj}}$$

Además del número de ciclos de reloj necesitados para ejecutar un programa, también se puede contar el número de instrucciones ejecutadas (*instruction count)*. Si se conoce el número de ciclos de reloj y la cantidad de instrucciones ejecutadas, se puede calcular el número promedio de ciclos de reloj por instrucción (CPI). El CPI puede ser calculado como:

$$\text{CPI} = \frac{\text{Ciclos de reloj de CPU para un programa}}{\text{Cantidad de Instrucciones}}$$

Tomando en cuenta CPI para el calculo del tiempo de CPU, tendríamos:

Tiempo de CPU= Cantidad de instrucciones X tiempo del ciclo de reloj X Ciclos por instrucción

Expandiendo la fórmula anterior se tiene:

$$\text{Tiempo de CPU} = \frac{\text{Instrucciones}}{\text{Programa}} \times \frac{\text{Ciclos de reloj}}{\text{Instrucción}} \times \frac{\text{Segundos}}{\text{Ciclo de reloj}} = \frac{\text{Segundos}}{\text{Programa}}$$

Esta fórmula demuestra que el rendimiento de la CPU, depende de tres características y cada una de ellas poseerá distintas interdependencias, tal como sigue:

- **Tiempo del ciclo de reloj:** depende de la tecnología de *hardware* y organización.
- **CPI :** depende de la organización y la arquitectura del *set* de instrucciones.
- **Cantidad de instrucciones:** depende de la arquitectura del set de instrucciones y de la tecnología de compilación.

Algunas veces, es útil para el diseñador, calcular el número total de ciclos de reloj de la CPU como:

$$\text{Ciclos de reloj de CPU} = \sum_{i=1}^{n} IC_i \times CPI_i$$

Donde IC_i representa el número de veces que la instrucción i es ejecutada en un programa y CPI_i representa el número promedio de ciclos de reloj para la instrucción i. De esta forma, el tiempo de CPU puede ser expresado como:

$$\text{Tiempo de CPU} = \left(\sum_{i=1}^{n} IC_i \times CPI_i \right) \times \text{Tiempo del ciclo de reloj}$$

y en conjunto, CPI se puede representar como:

$$CPI = \frac{\left(\sum_{i=1}^{n} IC_i \times CPI_i\right)}{\text{Cantidad de instr.}} = \sum_{i=1}^{n} \frac{IC_i}{\text{Cantidad de instr.}} \times CPI_i$$

El aumento del rendimiento puede ser obtenido por el perfeccionamiento de alguna porción en la arquitectura, este puede ser calculado usando la ley de Amdahl. La ley de Amdahl define el *speedup* como el aumento del rendimiento por usar una característica particular. Supongamos que se puede mejorar el microprocesador, entonces el *speedup* es:

$$\textit{Speedup} = \frac{\text{Rendimiento para una tarea usando la característica mejorada}}{\text{Rendimiento para una tarea, sin usar la mejora}}$$

El *pipeline* es una técnica de paralelismo que aumenta y perfecciona el desempeño en nuestro microprocesador, siendo el *speedup* una medida de rendimiento en nuestro sistema, estando definido este en el caso del *pipelining* por:

$$\textit{Speedup} \text{ del } \textit{Pipelining} = \frac{\text{Tiempo promedio de instrucción sin pipeline}}{\text{Tiempo promedio de instrucción con pipeline}}$$

Donde el tiempo promedio de ejecución de una instrucción en un microprocesador sin pipeline está dado, según lo visto anteriormente, por:

Ciclo de reloj X CPI promedio

Para determinar el tiempo de ejecución promedio, utilizando la técnica de *pipeline*, es necesario tomar en consideración diferentes aspectos, tales como paradas o atrasos debido a dependencias entre instrucciones y retardos debido a dependencias estructurales y de control, por lo que será desarrollado en los siguientes incisos.

2.3. Paradas debidas a dependencias (*Pipeline Hazards*)

Las dependencias, reducen el rendimiento para un *speedup* ideal en el proceso de *pipeline*, ocasionando paradas y retardos, haciendo que éste no trabaje de manera tan efectiva. Las dependencias afectan el rendimiento del pipeline de la siguiente manera:

1. **Dependencias estructurales:** causan conflicto cuando el *hardware* no puede soportar todas las posibles combinaciones de instrucciones simultáneamente, en una ejecución traslapada.
2. **Dependencias de datos:** cuando una instrucción depende del resultado de instrucción previa, imposibilita el traslape entre ciertas etapas entre instrucciones en el *pipeline.*
3. **Dependencias de control:** impiden el procesamiento paralelo en el *pipeline* debido a las dependencias de las ramificaciones con sus argumentos.

Estas dependencias en cierta manera atascan el pipeline, causando paradas conocidas como "*stalls*". Para evitar cualquier riesgo de una mala ejecución debido a una dependencia, requiere que algunas instrucciones sean ejecutadas, mientras otras sean retardadas.

2.3.1. Rendimiento del *pipeline* con paradas *(Stalls)*

Una parada (*stall*) causa que el rendimiento del *pipeline* disminuya respecto a un rendimiento ideal , donde el *pipeline* sin paradas, está definido como:

$$\textit{Speedup del pipelining} = \frac{\text{Tiempo promedio de instrucción sin pipeline}}{\text{Tiempo promedio de instrucción con pipeline}}$$

Donde, se puede expresar como:

$$\textit{Speedup del pipeline} = \frac{\text{CPI sin } \textit{pipeline} \text{ X Ciclo de reloj sin } \textit{pipeline}}{\text{CPI con } \textit{pipeline} \text{ X Ciclo de reloj con } \textit{pipeline}}$$

El CPI es directamente proporcional al *speedup del pipeline*. El CPI ideal en un procesador con un *pipeline* sin paradas, es casi siempre 1. De aquí, podemos calcular el CPI, considerando su afección a las paradas:

CPI con pipeline = Ideal CPI + Ciclos de reloj por instrucción debido a paradas
= 1 + Ciclos de reloj por instrucción debido a paradas

Si asumimos que cada estado o etapa del *pipeline* está perfectamente balanceado, considerando el tiempo consumido por las paradas, podemos definir el *speed up* del *pipeline* como:

$$\textit{Speedup del pipeline} = \frac{1}{\text{1 + Ciclos por instrucción debido a paradas}} \times \frac{\text{Ciclo de reloj sin pipeline}}{\text{Ciclo de reloj con pipeline}}$$

Por lo que vemos que en el caso ideal, si no existiesen paradas, el *speedup* sería igual al número de estados del *pipeline.*

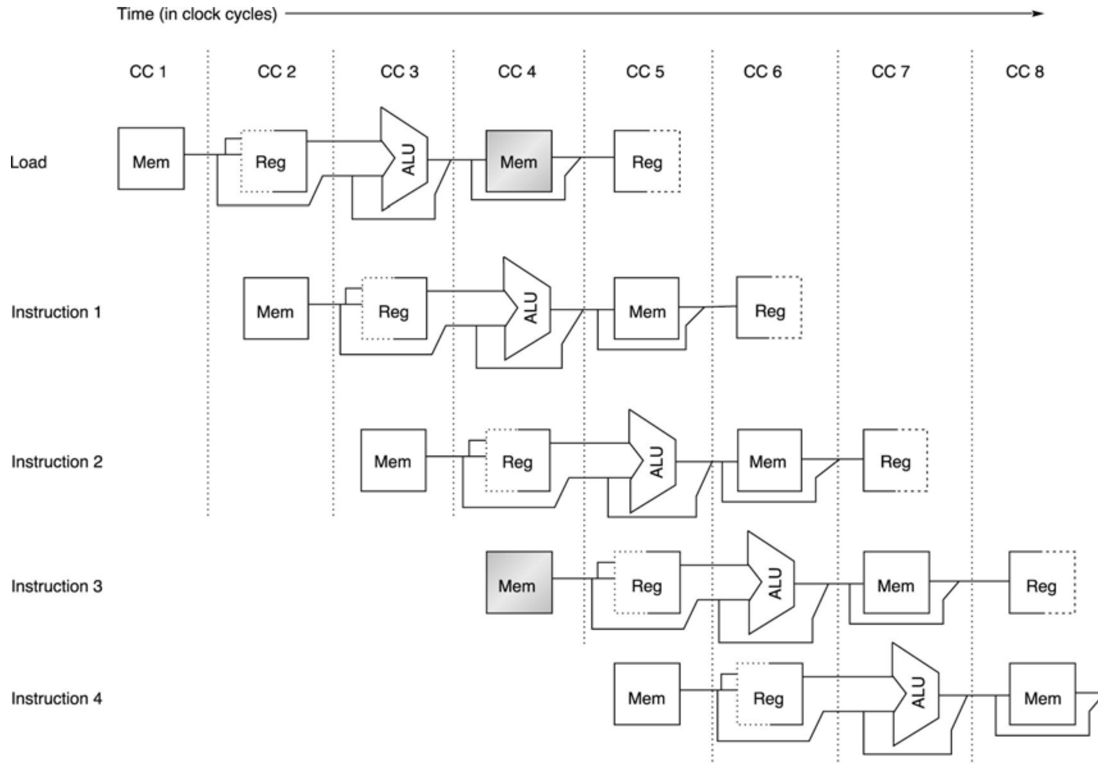
2.3.2. Dependencias estructurales

Cuando en un procesador esta implementada la técnica de *pipeline*, el traslape entre la ejecución de instrucciones, requiere la ejecución paralela de varias unidades funcionales y la duplicación de recursos para poder procesar todas las posibles combinaciones de instrucciones en el *pipeline*, si algunas de las combinaciones de instrucciones, no puede ser acomodada por conflictos entre recursos, se dice que el procesador está susceptible a tener obstáculos debido a dependencias estructurales.

Si una secuencia de instrucciones, encuentra esta dependencia, el *pipeline* deberá entrar en estado de espera, causando así una parada, hasta que la unidad funcional requerida este disponible.

Algunos procesadores tienen compartida una sola memoria para datos e instrucciones, esto causa, que cuando una instrucción contiene una referencia de datos de memoria, tendrá conflicto con una instrucción posterior, tal como se muestra en la figura 10.

Figura 10. Conflicto en el acceso a memoria, provocadas por dependencias estructurales



Fuente: Elsevier Science (USA) 2003

En el ejemplo de la figura anterior, la instrucción de carga (*load*), utiliza la memoria para el acceso de datos al mismo tiempo que la instrucción 3 va a ser extraída (*fetch*) de memoria. Por lo que en este instante, se hace necesario, hacer una parada (*stall*) tal como se muestra en la tabla V.

Tabla V. Parada debida a dependencia estructural

Número de instrucción	**Número de ciclos de reloj**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción de carga	IF	ID	EX	MEM	WB					
Instrucción 1		IF	ID	EX	MEM	WB				
Instrucción 2			IF	ID	EX	MEM	WB			
Instrucción 3				*stall*	IF	ID	EX	MEM	WB	
Instrucción 4						IF	ID	EX	MEM	WB

La parada (*stall)* permite que el dato de la instrucción de carga sea extraído de su posición de memoria sin ningún riesgo, pudiendo hacer el acceso a memoria para la instrucción 3 en el siguiente ciclo, incrementado de esta manera el CPI.

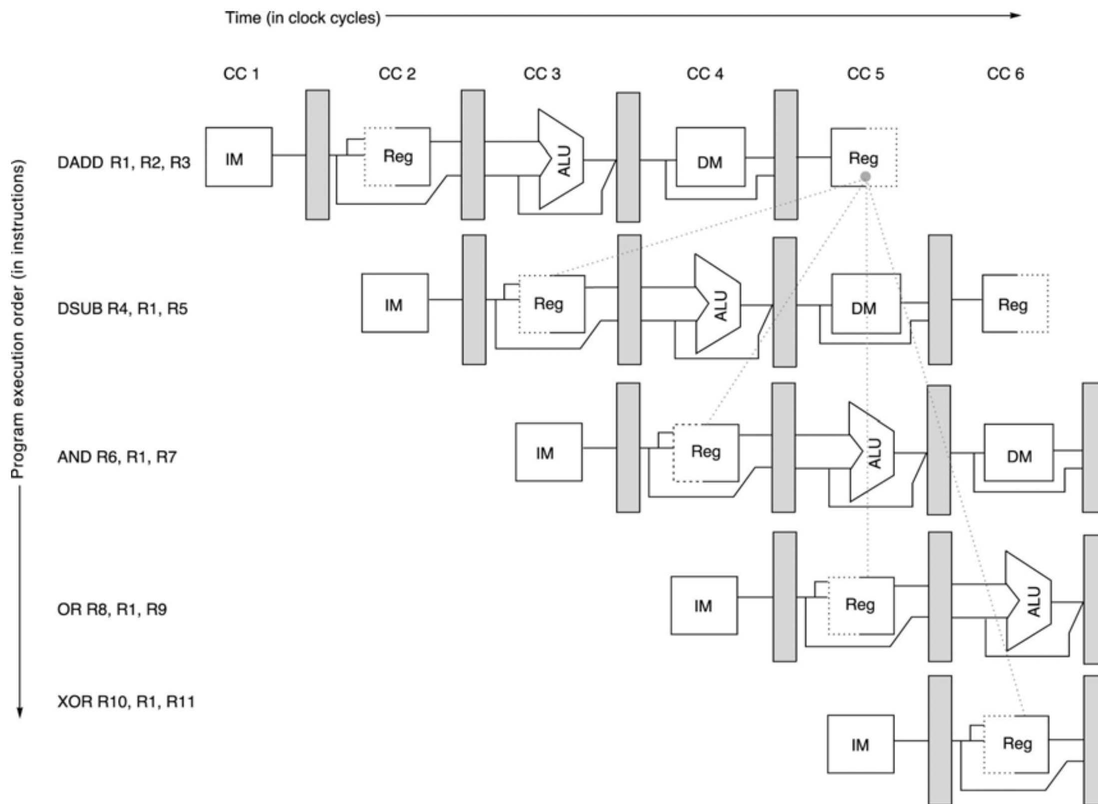
2.3.3. Dependencias de datos

Consideremos el siguiente segmento de código y su ejecución en el *pipeline*:

```
DADD     R1,R2,R3
DSUB     R4,R1,R5
AND      R6,R1,R7
OR       R8,R1,R9
XOR      R10,R1,R11
```

Todas las instrucciones después de la instrucción de suma DADD usan el resultado de esta, como se muestra en la figura 11.

Figura 11. Obstáculos en el *pipeline* debido a dependencias de datos



Fuente: Elsevier Science (USA) 2003

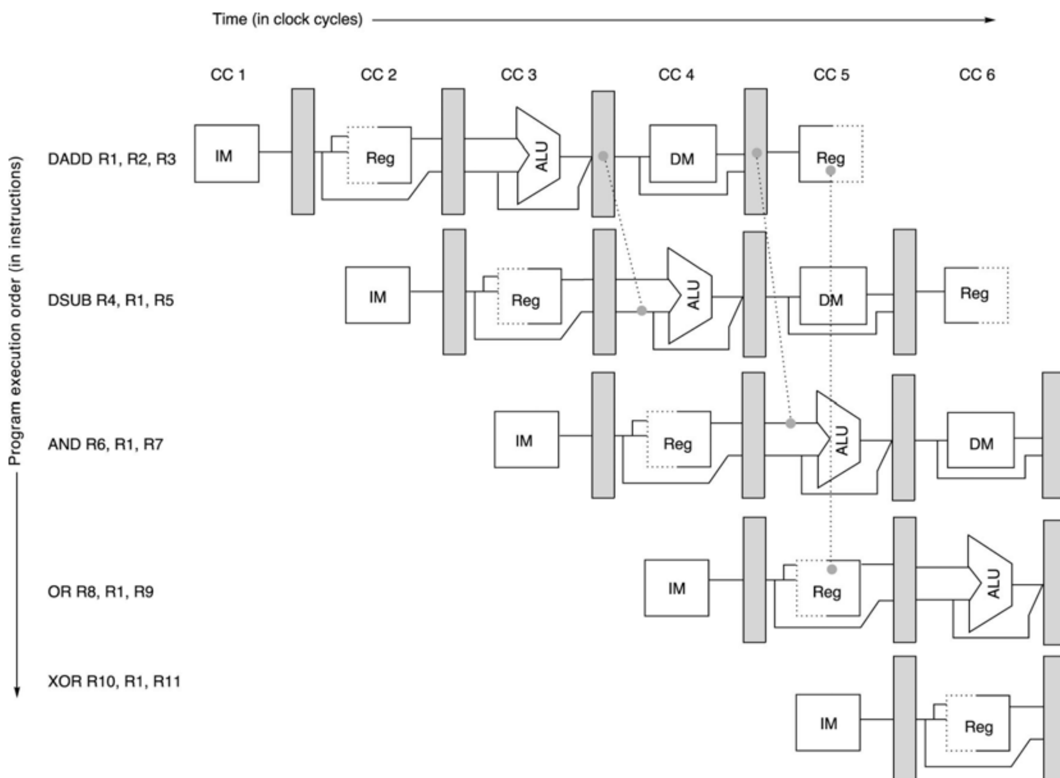
La instrucción DADD escribe el valor de R1 en el estado de WB (*Write Back*) del *pipeline*, pero la instrucción DSUB (resta), lee el valor durante el estado ID (*Instruction Decode*). En este caso, se debe prevenir que la instrucción DSUB lea un valor incorrecto e intente usarlo, al igual que las otras instrucciones dependientes de R1.

2.3.3.1 Minimización de paradas por medio de anticipación *(forwarding)*

El problema presentado en la figura 11 puede ser resuelto con una técnica de *hardware* llamada *forwarding* (también llamada *bypassing* y algunas veces *short-circuiting*), que no es más que una anticipación.

El objetivo de la anticipación, es que el resultado no se necesite realmente hasta que DADD lo produzca en el registro, sino que el resultado puede ser movido desde el registro del *pipeline* cuando DADD lo carga, de esta manera las paradas pueden ser removidas, tal como se muestra en la siguiente figura:

Figura 12. Utilización de la técnica de anticipación (*forwarding*)



Fuente: Elsevier Science (USA) 2003

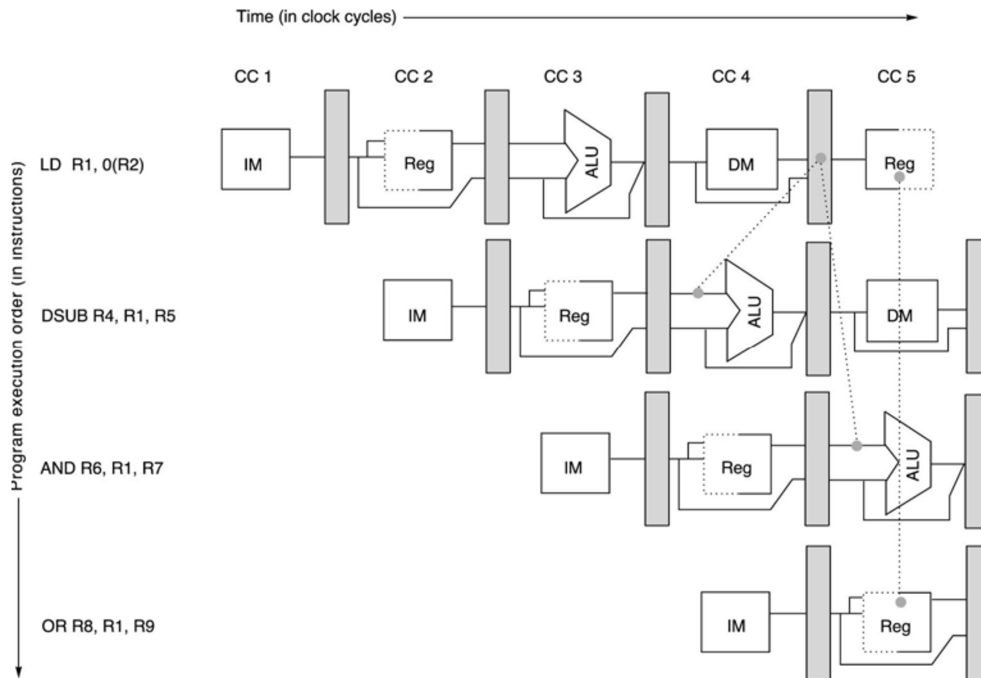
Las entradas para las instrucciones DSUB y AND son tomadas desde los registros intermedios del *pipeline* a la entrada de la ALU. La instrucción OR recibe este resultado a través del registro, lo cual se logra fácilmente por la lectura de los registros en la segunda mitad del ciclo y escribiendo en la primera mitad, tal como lo muestran las líneas punteadas.

Desafortunadamente, no todas las dependencias de datos pueden ser manipuladas por medio del *forwarding*. Consideremos la siguiente secuencia de instrucciones:

```
LD      R1, 0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

La secuencia en las unidades funcionales, para éste segmento de código, se muestra en la figura 13.

Figura 13. Dependencia de datos de una instrucción de carga



Fuente: Elsevier Science (USA) 2003

Este caso es diferente, ya que la instrucción LD (*load)*, no tiene el dato hasta el final del ciclo de reloj 4 (siendo éste el ciclo de la etapa MEM), mientras que la instrucción DSUB lo necesita antes.

Esta dependencia no puede ser completamente eliminada con simple *hardware*, la instrucción de carga (LD), provoca un retardo o latencia que no puede ser eliminado solamente con *forwarding.* En cambio, necesitamos agregar hardware, llamado *pipeline interlock*, que detecta la dependencia y para el *pipeline*, hasta que la dependencia sea eliminada.

En este caso el *interlock* para el pipeline, hasta que la fuente produzca el dato que se necesita, por lo que el CPI incrementa debido al la parada efectuada, tal como se muestra en la siguiente tabla:

Tabla VI. Paradas causadas por la dependencia de datos de una instrucción de carga

	Número de ciclos de reloj								
Instrucción	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
LD R1, 0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	*stall*	EX	MEM	WB		
AND R6,R1,R7			IF	*stall*	ID	EX	MEM	WB	
OR R8,R1,R9				*stall*	IF	ID	EX	MEM	WB

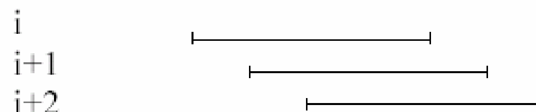
La figura 16 muestra la afección en el *pipeline* debido a la necesidad de la realización de paradas a causa de las dependencias, usando los nombres de los diferentes estados.

2.3.4. Dependencias de control (*Branch Hazards)*

Este tipo de dependencias pueden causar grandes pérdidas en el rendimiento del procesamiento de instrucciones.

Hasta ahora se ha considerado el secuenciamiento implícito, tal como se muestra en la figura 14.

Figura 14. Secuenciamiento implícito.



Por defecto en los programas el secuenciamiento es implícito. Lo que hace incrementar automáticamente el contador del programa según el tamaño de la instrucción. Nuestro procesador sólo necesita un sumador en la etapa de búsqueda para sumar una constante.

¿Qué pasa cuando la instrucción buscada es una instrucción de ruptura de la secuencia? (Salto incondicional, condicional, llamada a subrutina, retorno a subrutina). Los saltos modifican la secuencia implícita de las instrucciones; esto hace perder rendimiento ya que es necesario algunos ciclos para calcular la dirección efectiva de la siguiente instrucción (y saber si hay que saltar o no).

Si la instrucción *i* es una toma de ramificación, entonces el PC (Contador de Programa), no cambia hasta el final del segmento ID (*Instruction Decode)*. En la tabla VII, se muestra una simple técnica, para proceder con ramificación el cual consiste en rehacer el ciclo de IF (*Instruction Fetch*) de la instrucción siguiente a la ramificación, una vez es detectada la ramificación durante el estado ID.

Tabla VII. Parada de un ciclo debido a una ramificación

	Número de ciclos de reloj									
Instucción	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción de ramificación	I F	ID	EX	MEM	WB					
Sucesor de la ramificación		IF	IF	ID	EX	MEM	WB			
Sucesor de la ramificación + 1				IF	ID	EX	MEM	WB		
Sucesor de la ramificación + 2					IF	ID	EX	MEM	WB	
Suecsor de la ramificación + 3						IF	ID	EX	MEM	WB

El primer ciclo de IF señalado en la tabla VII, es esencialmente una parada, ya que no tiene funcionalidad. Podemos darnos cuenta, que si la ramificación no es tomada, la repetición del estado IF es innecesaria ya que la correcta instrucción fue tomada.

Una parada de un ciclo para cada ramificación podrá producir una pérdida de rendimiento del 10% al 30% dependiendo de la frecuencia con que se presenten.

2.3.4.1 Reducción de las dependencias de ramificación.

Existen varias técnicas para tratar las paradas en el *pipeline* causadas por el retardo generado al procesar una instrucción de ramificación, discutiremos cuatro esquemas en tiempo de compilación. El *software* puede tratar de minimizar la paradas, teniendo conocimiento del esquema de *hardware* y del comportamiento de la ramificación.

Un esquema simple para tomar las ramificaciones, es bloqueando ó llenando el *pipeline*, manteniendo ó borrando cualquier instrucción después de la ramificación hasta que el destino de la ramificación sea conocido, tal como el esquema que se mostró en la tabla VII. En este caso es reparado el riesgo generado por el retardo generado, pero la pérdida de un ciclo no puede ser reducida.

Otro esquema, y levemente más complejo, es tratar todas las ramificaciones como no tomadas, simplemente permitiendo que el *hardware* continúe como si la ramificación no fuera ejecutada.

Aquí, se debe tener cuidado en no cambiar el estado del procesador hasta que el resultado de la ramificación, sea definitivamente conocido. La complejidad de este esquema surge desde tener que saber cuando el estado deberá ser cambiado por una instrucción y como retroceder y eliminar lo ya trabajado.

En un *pipeline* básico de cinco estados, el esquema de tomar o no tomar la ramificación, es implementado con la continua toma de instrucciones como si la ramificación fuera una instrucción normal, el *pipeline* trabaja como que si nada fuera de lo normal estuviera pasando. Si la ramificación es tomada necesitamos llevar la instrucción tomada a un estado de no operación y reiniciar la búsqueda a la dirección destino. La tabla VIII muestra ambas situaciones.

Tabla VIII. Esquema de toma de ramificación

	Número de ciclos de reloj								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrucción de ramificación no tomada	IF	ID	EX	MEM	WB				
Instrucción i+1		IF	ID	EX	MEM	WB			
Instrucción i+2			IF	ID	EX	MEM	WB		
Instrucción i+3				IF	ID	EX	MEM	WB	
Instrucción i+4					IF	ID	EX	MEM	WB
Instrucción de ramificación tomada	IF	ID	EX	MEM	WB				
Instrucción i+1		IF	**Idle**	**idle**	**idle**	**idle**			
Ramificación destino			IF	ID	EX	MEM	WB		
Ramificación destino + 1				IF	ID	EX	MEM	WB	
Ramificación destino + 2					IF	ID	EX	MEM	WB

Una alternativa a este esquema es tratar todas las ramificaciones como tomadas, tan pronto como la instrucción de ramificación es decodificada y la dirección destino es calculada, asumiendo que la ramificación sea tomada y empezando a traer y ejecutar la ramificación destino. En nuestro *pipeline* de cinco etapas no conocemos la dirección destino antes de que conozcamos el resultado de comparación de nuestra instrucción de ramificación, por lo que esta alternativa no es una ventaja en este enfoque de *pipeline*. En algunos procesadores la ramificación destino es conocida antes que el resultado de la instrucción y una predicción de ramificación tomada, tiene sentido hacerla.

Un cuarto esquema en uso en algunos procesadores es llamado *delayed branch*. En un *delayed branch*, los elementos del ciclo de ejecución son:

- Instrucción de ramificación
- Sucesor de secuencia
- Ramificación destino, si ésta es tomada

El sucesor de secuencia está localizado en el *branch delay slot*. El *branch delay slot*, es un espacio destinado para una instrucción. La instrucción es ejecutada independientemente si la ramificación es tomada o no. El comportamiento de un *pipeline* de cinco estados con este esquema es mostrado en la tabla IX.

Tabla IX. Comportamiento del *pipeline* con esquema *delayed branch*

	Número de ciclos de reloj								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrucción de ramificación no tomada	IF	ID	EX	MEM	WB				
Instrucción *Branch Delay* (i+1)		IF	ID	EX	MEM	WB			
Instrucción i+2			IF	ID	EX	MEM	WB		
Instrucción i+3				IF	ID	EX	MEM	WB	
Instrucción i+4					IF	ID	EX	MEM	WB
Instrucción de ramificación tomada	IF	ID	EX	MEM	WB				
Instrucción *Branch Delay* (i+1)		IF	ID	EX	MEM	WB			
Ramificación destino			IF	ID	EX	MEM	WB		
Ramificación destino + 1				IF	ID	EX	MEM	WB	
Ramificación destino + 2					IF	ID	EX	MEM	WB

A pesar que es posible tener un *branch delay* más largo que uno, en la práctica, casi todos los procesadores con *delayed branch* tienen solo una instrucción de retardo.

El trabajo del compilador es hacer que el sucesor de instrucciones, sea válido para todo uso. La figura 15 muestra las tres maneras en las cuales el *branch delay* puede ser planificado.

Figura 15. Planificación del *delay slot*

Fuente: Elsevier Science (USA) 2003

En la figura 15 cada bloque en la parte superior de cada uno de los tres pares, se muestra el código antes de planificar, el bloque de abajo muestra el código ya planificado. En (a) el *delay slot* está planificado con una instrucción independiente situada antes de la ramificación, ésta es la mejor opción. Las estrategias (b) y (c) son usadas cuando (a) no es posible. En las secuencias de código para (b) y (c), el uso de R1 en la condición de la ramificación impide que la instrucción DADD (donde su destino es R1) sea trasladado después de la ramificación. La estrategia (b) es preferida cuando la ramificación es tomada debido a su configuración, mientras que la estrategia (c) es preferible cuando existe una alta probabilidad de que la ramificación no sea tomada.

Las limitaciones de la planificación mediante *delayed branch* surgen de:

1. Las restricciones sobre las instrucciones que son planificadas dentro los *delay slots.*
2. Nuestra habilidad de predecir en tiempo de compilación si la ramificación es tomada o no.

Para perfeccionar la habilidad del compilador de llenar los *delay slots*, muchos procesadores con ramificaciones condicionales tienen introducido un cancelador o anulador de ramificación. Por lo que cuando la predicción de la ramificación es incorrecta, la instrucción en el *branch delay slot* es simplemente puesta en un estado de no operación.

2.4. Manejo de excepciones (Interrupciones y Traps)

Cuando ocurre una excepción, es bastante difícil la manipulación en una CPU con *pipeline* implementado, debido al traslape de instrucciones, esto hace más dificultoso conocer si una instrucción puede cambiar en una forma segura el estado de la CPU. En un CPU con *pipeline*, una instrucción es ejecutada parte por parte y es completada en varios ciclos de reloj. Desafortunadamente, en el *pipeline* pueden surgir excepciones que pueden forzar la CPU a abortar las instrucciones antes de su finalización.

Las excepciones están clasificadas en cinco categorías semi-independientes:

1. Síncronas / asíncronas. Si el evento ocurre en el mismo lugar donde el programa es ejecutado con la misma localidad de datos y memoria, el evento es síncrono. Cuando los eventos son causados por mal funcionamiento de dispositivos externos a la CPU y memoria son asíncronos. Los eventos asíncronos usualmente pueden ser manipulados después de la finalización de la instrucción presente, lo que hace más fácil su manejo.

2. Requeridas por usuario / forzadas. Las excepciones requeridas por el usuario son predecibles, ya que se tiene el conocimiento del acceso del usuario, por lo que estas pueden ser tomadas después de la finalización de la instrucción presente. Las excepciones forzadas son causadas por algunos eventos de *hardware* que no están sobre el control del usuario del programa. Las excepciones forzadas son difíciles de implementar ya que no son predecibles.

3. Enmascarables por usuario / no enmascarables por usuario. Si un evento puede ser enmascarado o deshabilitado por el usuario, esta es una excepción enmascarable. Esta máscara simplemente controla si el hardware responde a la excepción o no.

4. Dentro de instrucciones / entre instrucciones. Esta clasificación depende si el evento impide la finalización de la instrucción por ocurrir en medio de la ejecución o si éste es reconocido entre instrucciones. Excepciones que ocurren dentro de instrucciones son usualmente síncronas, donde la instrucción presente deberá ser parada y reiniciada. Si una excepción asíncrona ocurre dentro de una instrucción surge una situación catastrófica y siempre causa que el programa termine.

5. Continuar / finalizar la ejecución. Esta clasificación depende si la ejecución del programa siempre termina después de la interrupción, ó si la ejecución del programa continúa después de la interrupción.

La Tabla X clasifica diferentes tipos de excepciones de acuerdo a las cinco categorías mencionadas anteriormente. Cuando una interrupción ocurre dentro de una instrucción donde la instrucción debe continuar, nos vemos en la necesidad de recurrir a otro programa que deberá ser invocado para guardar el estado de la ejecución del programa correctamente para luego reestablecer el estado del programa antes de la excepción. Este proceso debe de ser efectivamente invisible al momento de ejecutar el programa. Si un *pipeline* provee la habilidad que el procesador tome la excepción, guarde el estado y reinicie sin afectar la ejecución del programa, el *pipeline* o el procesador es llamado "*restartable"* o procesadores "rearrancables".

Tabla X. Acciones necesitadas para diferentes tipos de excepciones

Tipo de Excepción	Sincrona / Asíncrona	Requerida por el usuario/ forzada	Enmascarable/ No Enmascarable	Dentro de instrucciones / Entre instrucciones	Continuar / Finalizar Ejecución
I/O device request	Asíncrona	Forzada	No Enmascarable	Entre	Continuar
Invoke operating system	Síncrona	Requerida por el usuario	No Enmascarable	Entre	Continuar
Tracing instruction execution	Síncrona	Requerida por el usuario	Enmascarable	Entre	Continuar
Breakpoint	Síncrona	Requerida por el usuario	Enmascarable	Entre	Continuar
Integer arithmetic overflow	Síncrona	Forzada	Enmascarable	Dentro	Continuar
Floating point arithmetic overflow or underflow	Síncrona	Forzada	Enmascarable	Dentro	Continuar
Page Fault	Síncrona	Forzada	No Enmascarable	Dentro	Continuar
Misaligned memory acces	Síncrona	Forzada	Enmascarable	Dentro	Continuar
Memory protection violations	Síncrona	Forzada	No Enmascarable	Dentro	Continuar
Using undefined instructions	Síncrona	Forzada	No Enmascarable	Dentro	Finalizar
Hardware malfunctions	Síncrona	Forzada	No Enmascarable	Dentro	Finalizar
Power failure	Asíncrona	Forzada	No Enmascarable	Dentro	Finalizar

Fuente: Hennesy John y David Patterson.Computer Architecture: A Quantitative Approach. 3ra. Edición. Apéndice A. Pag. A-42. Traducción libre del autor.

Hasta aquí vemos, dos propiedades que hacen difícil el manejo de excepciones:

1. Cuando la excepción ocurre dentro de la instrucciones (que es en medio de la ejecución correspondiendo a los estados EX o MEM del *pipeline*).
2. Se deberá poder rearrancar.

Al ocurrir la excepción en el *pipeline* deberá ser cerrado el proceso y su estado guardado sin ningún peligro para poder ser reestablecido de manera correcta. El reestablecimiento es usualmente implementado guardando el contenido del contador de programa (PC) de la instrucción a la cual se va a reiniciar. Si la instrucción que se reestablece no es una ramificación entonces se podrá continuar y ejecutar en un modo normal. Si la instrucción reestablecida es de ramificación, entonces deberá ser reevaluada la condición de ramificación y empezar a tomar instrucciones desde la ramificación destino. El control del *pipeline* puede tomar los siguientes pasos para guardar el estado de éste adecuadamente:

1. Forzar una instrucción de *trap* en el siguiente IF.
2. Hasta que entre el trap, eliminar las escrituras de la instrucción que causó el fallo y de las siguientes.
3. Guardar el PC de la instrucción que causó el fallo.

Cuando utilizamos la técnica de *delayed branch* mencionada en el inciso anterior, no es posible recrear el estado del procesador con un solo dato del contador de programa (PC) ya que las instrucciones del *pipeline* pueden no estar secuencialmente relacionadas, por lo que es necesario en el paso 3 restaurar varios valores del PC, tantos como el largo del *branch delay* mas uno.

Cuando se completan las instrucciones anteriores a la excepción y las posteriores se reinician desde el principio, con el propósito de restaurar el estado del programa, el proceso es llamado, tratamiento preciso de excepciones, tomando la excepción el nombre de "*precise exceptions"* ó excepciones precisas, tratando de esta manera las excepciones en el mismo orden que las instrucciones.

2.5. Extensión del *pipeline* para manipulación de operaciones multiciclo

Cuando se habla de operaciones multiciclo, nos referimos a las operaciones de punto flotante. Las operaciones de punto flotante no se pueden finalizar en uno o dos ciclos por lo que nos enfrentamos a dos cambios importantes:

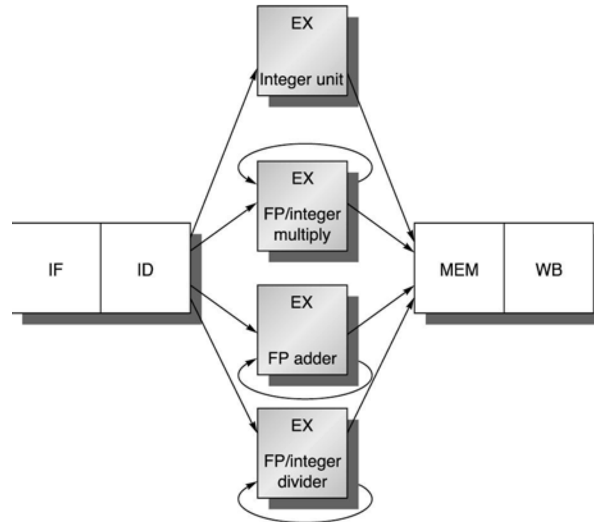
1. El ciclo de ejecución (EX) puede ser repetido tanto como las veces que lo necesitemos para completar la operación.
2. Es necesario añadir varias unidades funcionales.

Una configuración básica, consta de cuatro unidades funcionales separadas como sigue:

1. Unidad de enteros (*Integer unit)*: Esta es utilizada en cargas y almacenamientos, operaciones de la ALU y en ramificaciones.
2. Multiplicador de enteros y punto flotante *(FP/Interger multiply).*
3. Sumador de punto flotante: sumas, restas y conversión (*FP adder).*
4. Divisor de enteros y punto flotante. *(FP/Interger divider).*

Por lo que la estructura de pipeline estaría formada tal como la figura 16.

Figura 16. Estructura del *pipeline* con 3 unidades funcionales de punto flotante.



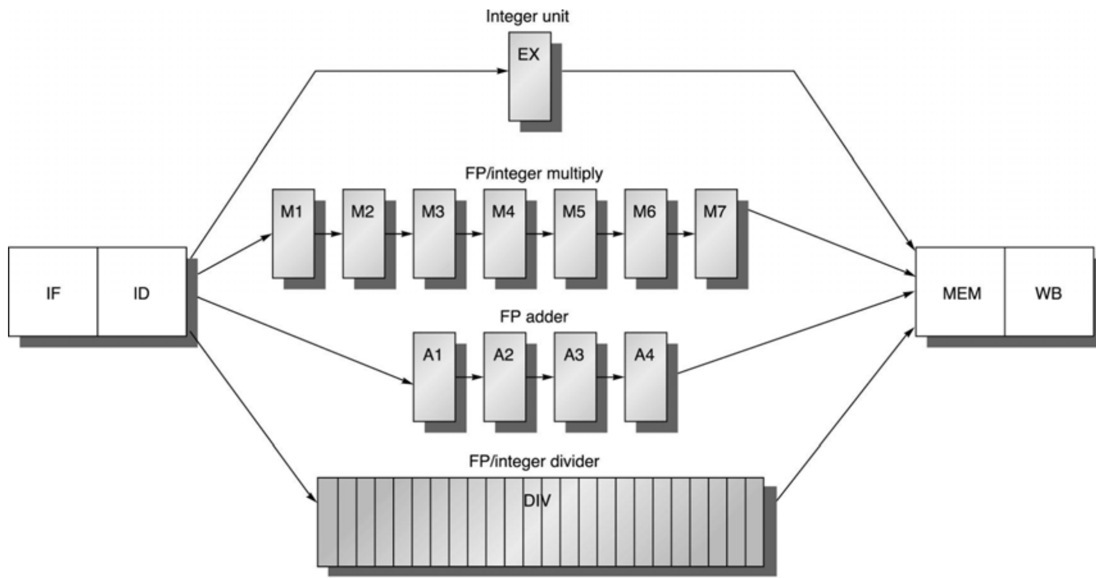
Fuente: Elsevier Science (USA) 2003

Algunas unidades necesitan varios ciclos de ejecución, por lo que para cada unidad son importantes dos características:

- **Latencia:** corresponde al número de ciclos vacíos que hay que esperar entre la emisión de la instrucción que produce el resultado y la siguiente que lo utiliza.
- **Intervalo de repetición:** corresponde al número de ciclos entre la emisión de dos operaciones del mismo tipo.

Por lo que se hace necesita la introducción de registros adicionales al *pipeline* y la modificación de las conexiones a estos registros. Tal como se muestra en la siguiente figura:

Figura 17. Etapa de ejecución multiciclo parcialmente segmentada



Fuente: Elsevier Science (USA) 2003

El registro ID/EX deberá ser expandido para conectar ID a EX, DIV, M1, y A1; asociando cada uno de ellos a los estados con la siguiente notación ID/EX, ID/DIV, ID/M1, y ID/A1. La latencia usualmente es el número de etapas de EXE para producir el resultado, por lo que para instrucciones de carga la latencia es mínima.

2.5.1. Dependencias y anticipación en *pipelines* de gran latencia.

Existen varios aspectos que se deben tomar en consideración en la detección de dependencias y anticipación para un pipeline como el de la figura 17.

1. Existencia de unidades no segmentadas: dos instrucciones que quieren utilizar la misma unidad, provocando una dependencia del tipo estructural, teniendo así la necesidad de detectar y emitir instrucciones de parada (stalls).

2. Escritura simultánea en el banco de registros, al haber instrucciones con diferente número de ciclos.

3. Dependencias de salida, al alterarse el orden de escritura en el banco de registros.

4. Problemas con las excepciones por finalización fuera de orden, ya que las son completadas en diferente orden en que son emitidas.

5. Parones debido a dependencias verdaderas (RAW).

Tomando en consideración los problemas anteriores y asumiendo que el *pipeline* tendrá la tarea de detección de dependencias en el estado ID, se está en la obligación de revisar y actuar antes de que una instrucción pueda emitirse de la siguiente manera:

- **Revisión de conflictos estructurales:** al tener unidades funcionales no segmentadas, se debe llevar el *pipeline* a un estado de espera por medio de paradas (*stalls)* hasta que la unidad funcional quede libre y asegurar de esta manera la disponibilidad del puerto de escritura del banco de registros.

- **Revisión de dependencias reales (RAW):** esperar hasta que los registros fuentes no estén listados como destinos pendientes de una instrucción que no haya finalizado cuando se necesite el resultado. El registro fuente de la instrucción en ID no puede ser el destino de ID/A1, A1/A2, A2/A3, A3/A4; ID/M1, M1/M2,..., M6/M7, ID/DIV.

- **Revisión de dependencias de salida:** parar en el estado ID, si cualquier instrucción en A1..A4, M1..M7, DIV, tiene el mismo registro destino que la instrucción presente.

Con lo anterior, se puede observar que la detección de dependencias es más compleja con operaciones de multiciclo de punto flotante, pero los conceptos e implementación es la misma que en los enteros. De esta misma manera es implementada la técnica de *forwarding*, comprobando si el registro destino en EX/MEM, A4/MEM, M7/MEM, DIV/MEM o MEM/WB coincide con el fuente ID, seleccionando por medio de la multiplexación el dato anticipado.

2.5.2. Tratamiento preciso de excepciones.

Otro problema causado por algunas instrucciones de gran latencia, puede ser ilustrado con la siguiente secuencia de código:

```
DIV.D     F0, F2, F4
ADD.D     F10, F10, F8
SUB.D     F12,  F12, F14
```

En estas instrucciones la extensión ".D" en el mnemónico de la instrucción indica doble precisión en operaciones de punto flotante. En esta secuencia de código es posible la anticipación, debido a la inexistencia de dependencias, pero el problema surge debido a que una de las instrucciones puede ser emitida antes y ser completada después de la última instrucción emitida, por lo que en este ejemplo podemos esperar que ADD.D y SUB.D sean completadas antes que DIV.D. Esto es llamado "Completación fuera de orden". Supongamos que la instrucción SUB.D causa una excepción de punto flotante, al momento en que ADD.D fue completada, pero DIV.D no ha sido completada, el resultado será una excepción imprecisa.

El problema surge porque las instrucciones son completadas en diferente orden en que son emitidas, por lo que hay cuatro maneras de proceder ante esta situación:

1. La primera forma consiste en ignorar el problema y tratarlo como una instrucción imprecisa. Este enfoque fue utilizado en los años 60's y a principio de los 70´s y aún es utilizado en algunas supercomputadoras donde cierta clase de excepciones no son permitidas o tomadas por el hardware sin parar el *pipeline*. Esto es dificultoso en el enfoque de los procesadores construídos el día de hoy por las características tales como la memoria virtual y el estándar de IEEE de punto flotante, los cuales requieren excepciones precisas a través de la combinación de *software* y *hardware*.

Algunos procesadores como el DEC Alpha 21064 y 21164, el IBM Power 1 y Power 2, y el MIPS R8000, tienden a resolver este problema introduciendo dos modos de ejecución: uno rápido, con modo impreciso y uno modo lento con excepciones precisas. El modo preciso puede ser tratado por la inserción de instrucciones explícitas que revisan las excepciones de punto flotante.

2. Un segundo enfoque consiste en almacenar los resultados de una operación hasta que todas las operaciones que son emitidas anticipadamente sean completadas. Algunos CPU utilizan esta solución que llega a ser costosa cuando la diferencia de tiempo de ejecución entre operaciones es grande, así el número de resultados a almacenar también será grande. Además, los resultados desde la cola deberán ser multiplexados para continuar emitiendo instrucciones mientras se espera la instrucción más larga. Existen dos variaciones viables para éste enfoque, la primera consiste en el uso de un historial (*history* file) usado en el CYBER 180/990.

   El historial, almacena los valores originales de los registros. Cuando una excepción ocurre el estado de la CPU deberá ser retornado antes de que las instrucciones fuera de orden sean completadas, el estado de los registros puede ser reestablecido mediante el historial, una técnica similar es utilizada en procesadores como VAX.

La segunda variante es llamada *future file*, propuesta por Smith y Pleszkun en 1988, la cual consiste en almacenar el más reciente valor de un registro; cuando todas las instrucciones emitidas anticipadamente sean completadas siendo actualizados los registros con valores precisos para el estado de interrupción, extensiones de esta idea son las técnicas utilizadas en procesadores como el PowerPC 620 y el MIPS R10000.

3. La tercer técnica en uso es permitir que las excepciones lleguen a ser un poco imprecisas, pero guardando la suficiente información para que las rutinas de reestablecimiento puedan crear una secuencia precisa de la excepción. Esto significa conocer que operaciones estaban en el *pipeline* y sus PCs. Entonces después de ello tomamos la excepción, donde el *software* termina cualquier instrucción que precede a la última instrucción completada y la secuencia puede ser reestablecida.

4. La técnica final es un esquema híbrido que permite que la instrucción emitida continúe solamente si todas las instrucciones antes de la emisión de la instrucción fueron completadas sin causar una excepción. Esto garantiza que cuando una excepción ocurre, no serán completadas instrucciones después de la interrupción y todas estas instrucciones antes de la interrupción pueden ser completadas.

3. CONSIDERACIONES DE DISEÑO EN LA PLANIFICACIÓN DINÁMICA DE INSTRUCCIONES

En una planificación dinámica, el *hardware* aumenta el ILP reordenando las instrucciones en tiempo de ejecución, de esta manera:

- Las instrucciones independientes se ejecutan simultáneamente en la unidad segmentada.
- Las instrucciones dependientes se ejecutan secuencialmente.

En el diseño de *pipeline* vimos que si una instrucción se queda parada, ninguna instrucción posterior puede continuar, incluso si es independiente de las que están en ejecución, y el operador que necesita está libre.

La idea clave de la planificación dinámica, consiste en que el *hardware* debe poder lanzar a ejecución instrucciones posteriores a la que se ha parado alterando de ésta manera dinámicamente el orden de la ejecución, y evita que el hecho de parar una instrucción afecte a las que le siguen. Por lo que ganamos las siguientes ventajas:

- Simplifica el diseño del compilador.
- Soluciona eficientemente dependencias en tiempo de compilación.
- Permite ejecutar eficientemente cualquier código.

Pero como en cualquier diseño, además de ventajas poseemos ciertos inconvenientes:

- El *hardware* es bastante más complejo
- Se pueden dar las dependencias WAR y WAW
- Hay que tener cuidado en el manejo de excepciones.

3.1. Planificación dinámica del *pipeline* (*scoreboarding*)

En el *Scoreboarding* se pretende que una instrucción con sus operandos disponibles no se quede paralizada si tiene posibilidad de ejecutarse (porque otra anterior esté bloqueada), esto se logra mediante la división de la fase ID del *pipeline*, en dos partes:

- **Lanzamiento o emisión (*issue)*:** decodificación y comprobación de dependencias estructurales.

- **Lectura de operandos (*Read Operands)*:** espera por operandos; cuando están listos, se leen y se pasa a ejecutar (no necesariamente respetando el orden del programa)

Todas las instrucciones pasan a través del estado de lanzamiento (*issue*) en orden, no obstante pueden ser paradas o desviadas al siguiente estado (*read operands)* y así entrar a una ejecución fuera de orden; de esta forma, una instrucción puede adelantar a otra que esté bloqueada por dependencias, tomando la responsabilidad de ello el marcador (*scoreboard*). Para ello se hacen necesarias varias unidades funcionales (UF), quedando determinado por el marcador, cuándo se ejecuta una instrucción y cuándo se escribe a un registro determinado.

De esta manera el flujo del *pipeline* de instrucciones, estará estructurado de la siguiente manera:

- **Búsqueda (*Instruction Fetch*):** como en el *pipeline* básico.
- **Lanzamiento (*issue*):** si su UF está libre, y no hay dependencias WAW con otra instrucción, la instrucción se envía a la UF; en caso contrario, el lanzamiento se bloquea.
- **Lectura de operandos (*read operands*):** si los datos están listos, el marcador (*scoreboard*) indica a la UF correspondiente que los lea. Esto permite la ejecución fuera de orden, garantizando que se respetan las dependencias RAW.
- **Ejecución (*execute*):** puede durar varios ciclos. Cuando termina, la UF avisa al marcador (*scoreboard*).
- **Almacenamiento (*write result*):** el marcador comprueba que no hay dependencias WAR, y si no los hay, indica a la UF que actualice el banco de registros.

Para llevar a cabo las tareas anteriores, el marcador posee los siguientes componentes e información:

- **Estado de las instrucciones:** indica en qué fase o etapa se encuentran las instrucciones, tal como que si fuese llenada la tabla XI.

Tabla XI. Marcador de estado de las instrucciones

		Estado de la instrucción		
Instrución	**Lanzamiento**	**Lectura de operandos**	**Ejecución**	**Almacenamiento**

- **Estado de las unidades funcionales:** indica la situación en la que se encuentra la unidad funcional en diferentes aspectos listados a continuación y mostrados en la tabla XII.
  - Ocupada / no ocupada
  - Operación que tiene que realizar
  - Operandos fuente para la operación (qué registros)(fj,fk)
  - Operando destino (qué registro) (fi)
  - UF que genera cada operando fuente (qj,qk)
  - Estado de los operandos fuente (listo / no listo) (rj,rk)

Tabla XII. Marcador de estado de las unidades funcionales.

	Estado de la unidad funcional								
Nombre	**Ocupada**	**Op**	**Fi**	**Fj**	**Fk**	**Qj**	**Qk**	**Rj**	**Rk**

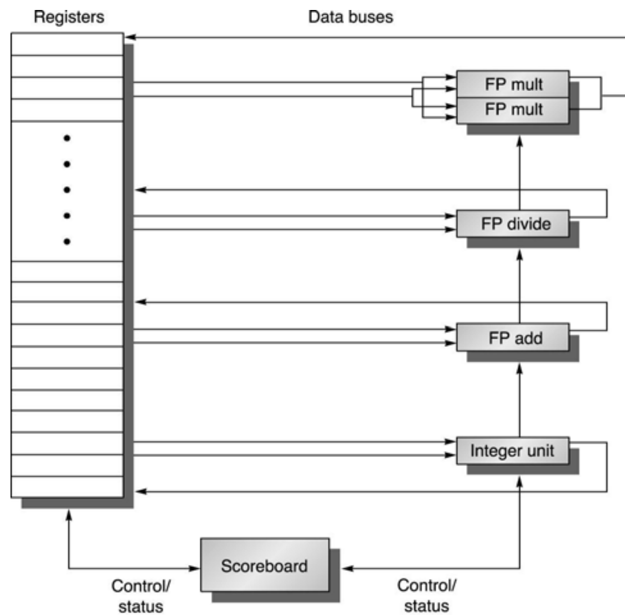
- **Estado de los registros:** indica la UF que va a generar el siguiente valor del registro (en blanco si no usado), tal como que si se llenara la tabla XIII.

Tabla XIII. Marcador de estado de escritura a registros.

	Estado de escritura a registros					
Registro	**F0**	**F1**	**F2**	**F3**	**. . .**	**F30**
Unidad funcional						

Un esquema básico de un procesador con *scoreboard* lo podemos observar en la figura 18, donde la función escencial de éste radica en el control de la ejecución de instrucciones y el flujo de datos.

Figura 18. Estructura básica de un procesador con *scoreboard*.



Fuente: Elsevier Science (USA) 2003

Las líneas verticales son las encargadas de controlar la ejecución de las instrucciones y todo el flujo de datos entre registros y unidades funcionales está representado por las líneas horizontales.

Las limitantes en el *scorebording* radican en la capacidad de eliminar paradas, donde se ve afectado por los siguientes factores:

- **El paralelismo existente en la aplicación:** si no hay instrucciones independientes, no se pueden eliminar las paradas.

- **El número de entradas en el marcador:** es decir, con cuántas instrucciones se puede trabajar a la vez.

- **El número y tipo de unidades funcionales:** influye sobre la cantidad de paradas estructurales.

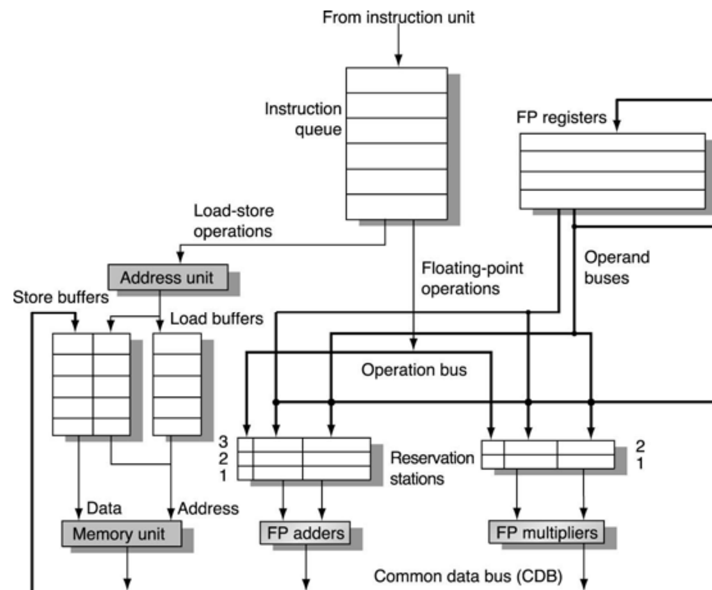
- La existencia de antidependencias (WAR) y dependencias de salida (WAW)

3.2. Planificación dinámica usando el algoritmo de Tomasulo

El algoritmo de Tomasulo, inventado por Robert Tomasulo fue implementado por primera vez en el procesador IBM 360/91 en su unidad de punto flotante. El algoritmo está basado en el esquema de *scoreboarding*, incorporando el concepto de renombramiento de registros (que elimina las dependencias WAR y WAW) en el estado de lanzamiento (*issue*).

Este esquema, utiliza *estaciones de reserva*, que almacenan las instrucciones que están pendientes de entrar a una UF (cada UF tiene la suya), donde cada estación de reserva controla cuándo pueden ejecutarse sus instrucciones pasando los resultados directamente de las estaciones a las UF sin pasar por los registros, a través de un bus común de datos o CDB. La estructura de una unidad de punto flotante usando el algoritmo de Tomasulo se puede observar en la figura 19.

Figura 19. Estructura de unidad de punto flotante utilizando el algoritmo de Tomasulo



Fuente: Elsevier Science (USA) 2003

De acuerdo con la estructura de la unidad, las fases en las que debe pasar una instrucción son las siguientes:

- **Lanzamiento:**
  - Se toma una instrucción de la cola.
  - Si hay espacio en su *estación de reserva*, se coloca ahí.
  - Si los operandos están en el banco de registros, se envían a la estación de reserva.
  - Si no hay espacio en la *estación de reserva*, hay una parada estructural.

- **Ejecución:**
  - Si falta algún operando, se vigila el CDB
  - Cuando están listos los operandos, se ejecuta la operación.

- **Almacenamiento:**
  - Cuando termina la ejecución, los resultados se ponen en el CDB. De ahí van a los registros y a las UF que los esperen.

Hasta aquí, podemos observar varias diferencias con el *scoreboarding*:

- No se comprueban las dependencias WAW y WAR: se eliminan al renombrar en la fase de lanzamiento

- No se espera que los datos lleguen a los registros: el CDB los lleva a las UF que los necesitan

- Los búfer de carga (*load buffer*) y almacenamiento (*store buffer*) se tratan como unidades funcionales básicas.

La implementación del renombrado de registros en este caso, se lleva a cabo mediante la asignación de un identificador (de 4 bits) a cada operando en la fase de lanzamiento, indicando así qué fuente proporciona el dato, donde las fuentes de datos corresponden a las 6 entradas del búfer de carga (*load buffer)*, 3 entradas de la estación de reserva de suma y 2 en la de multiplicación, extendiendo de esta manera el número de registros de 4 a 11.

La información que se almacena en la estación de reserva está asignada a diferentes campos:

- **Op:** operación que se tiene que ejecutar.
- **Qj,Qk:** *estación de reserva* que produce los operandos (un valor de cero indica que el operando fuente esta disponible en Vj y Vk o es innecesario).
- **Vj,Vk:** valor de los operandos.
- **Ocupado (*Busy*):** indica si la entrada está ocupada.

De esta misma manera la información que se almacena en cada registro y en los búfers de *stores* es la siguiente:

- Qi: el número de la *estación de reserva* que genera el valor que hay que enviar a memoria.
- V: valor que hay que enviar a memoria.

Por lo que el algoritmo de Tomasulo puede ser descrito tal como se presenta en la figura 20.

Figura 20. Detalle del algoritmo de Tomasulo.

Instruction state	Wait until	Action or bookkeeping

Issue
FP Operation
	Station r empty	if (RegisterStat[rs].Qi≠0)
		{RS[r].Qj ← RegisterStat[rs].Qi}
		else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};
		if (RegisterStat[rt].Qi≠0)
		{RS[r].Qk ← RegisterStat[rt].Qi
		else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
		RS[r].Busy ← yes; RegisterStat[rd].Qi=r;

Load or Store	Buffer r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Load only		RegisterStat[rt].Qi=r;
Store only		if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
Execute FP Operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load-Store step 1	RS[r].Qj = 0 & r is head of load-store queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write result FP Operation or Load	Execution complete at r & CDB available	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
Store	Execution complete at r & RS[r].Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

Fuente: Hennesy, John L. y David A. Patterson. Computer Architecture: A Quantitative Aproach. Pág. 193

En la figura 20 son mostrados los requerimientos necesarios para cada paso del algoritmo de Tomasulo, señalizando el estado de la instrucción (*instruction state)*, describiendo las diferentes situaciones que se pueden dar (*wait until*) y mostrando la acción tomada (*action or bookkeeping)*. Los operandos “rs” y “rt” indican el número de registro fuente, “rd” es el destino, “r” es la estación de reserva, “imm” es el *sign-extended inmediate field*. Mientras que el valor retornado por una unidad FP o por una unidad de carga es llamado “*result.RegisterStat*”.

Existen muchas variaciones sobre este esquema en procesadores superescalares modernos; sin embargo, el concepto clave de rastrear y procesar las dependencias en instrucciones para permitir la ejecución tan pronto como los operandos estén disponibles y el renombramiento de registros para eliminar las dependencias WAW y WAR son características comunes.

3.3. Reduciendo los costos de ramificación con predicción dinámica

En la sección anterior describimos técnicas para tratar las dependencias de datos. La frecuencia de ramificaciones y saltos hace necesario también estudiar las técnicas para el tratamiento de las dependencias de control. Las técnicas son usadas con dos propósitos:

- Predecir si una ramificación será tomada.
- Encontrar el destino lo más rápido posible, con el objeto de eliminar paradas.

En el capítulo 2 se examinó una variedad de esquemas básicos (como por ejemplo, parar el cauce, no tomar la ramificación, suponer que se toma y *delayed branch*) para el tratamiento con ramificaciones, estos esquemas eran del orden estático: la acción de tomar o no tomar la ramificación no depende del comportamiento dinámico de la ramificación. Esta sección se enfoca sobre la utilización de *hardware* para predecir dinámicamente el resultado ó desenlace de una ramificación; la predicción dependerá de el comportamiento de la ramificación al tiempo de corrida y cambiará si la ramificación cambia éste comportamiento durante la ejecución.

3.3.1. Búfer de predicción

El objetivo del búfer de predicción consiste en predecir en qué sentido va a ir cada salto. El problema de los esquemas estáticos es que siempre hacen la misma predicción sin embargo los saltos no se comportan así, no todos saltan en el mismo sentido.

Para efectuar la predicción, la información sobre la que se trabaja está enfocada en el comportamiento anterior del salto (del salto concreto, o de todos los saltos, según el esquema que se tome)

Un esquema básico consiste en una tabla de bits a la que se accede con una porción de la dirección de la instrucción de salto por ejemplo: si hay 1024 bits, se accede con los 10 bits inferiores de la dirección del salto, tal como se muestra en la figura 21.

Figura 21. Acceso mediante una porción del PC al Buffer de predicción

Con el esquema de la figura 21, se presenta el problema de que a varios saltos les corresponde la misma entrada en la tabla, pero no es un problema potencial, ya que la predicción es una sugerencia que es asumida como correcta; una de las soluciones a esto es considerar más bits del PC en el índice de la tabla, tomando ésta mayor tamaño.

En una unidad segmentada que calcula la dirección de destino en ID y la condición de salto en EX, añade los siguientes eventos en las fases del ciclo de instrucción:

IF: Búsqueda de la instrucción

ID: Decodificación de la instrucción

Si es un salto:

- Se calcula la dirección destino del salto.
- Si el bit de predicción indica “destino entonces, se comienza la búsqueda de instrucciones en la dirección destino del salto.

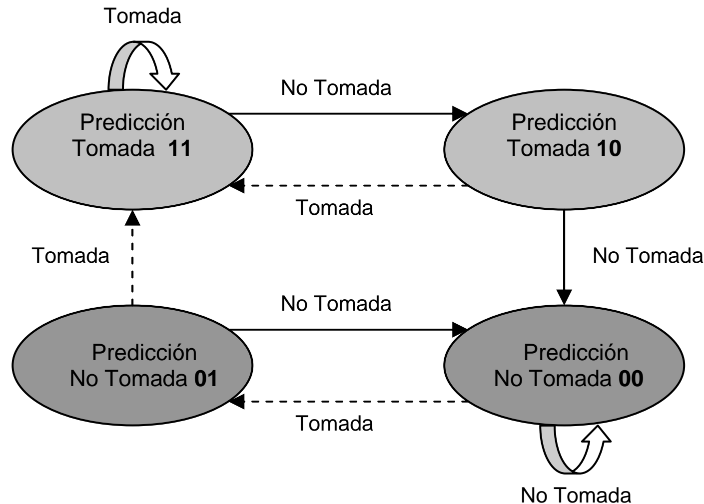
EX: Cálculo de la condición de salto.

Si la predicción es incorrecta:

- Se terminan las instrucciones incorrectamente lanzadas.
- Se actualiza el bit de predicción.
- Se comienza la búsqueda de instrucciones en la dirección correcta.

Si tomamos un esquema con un bit de predicción éste corresponderá a 1 si la última vez el salto fue tomado y a 0 si no se tomó, la predicción se basa en el comportamiento del salto en la última vez que fue ejecutado. Con 1 bit de predicción, si una ramificación tiende a ser casi siempre tomada, probablemente se hará la predicción incorrecta dos veces en lugar de una, cuando la ramificación no es tomada. Una solución a esto es la utilización de 2 bits de predicción donde las entradas de la tabla son contadores 2 bits con saturación, tal como se muestra en la figura 21.

Figura 22. Estados en un esquema con 2 bits de predicción



Según el esquema de la figura 22, cuando un salto es tomado el contador se incrementa (sin pasar de 11) y cuando no se toma se decrementa (sin pasar de 00); el acceso a la tabla de de la misma manera, es decir con los bits inferiores de la dirección de salto. La predicción es tomada de la siguiente manera:

- Si el contador vale 10 u 11, el salto se predice tomado.
- Si el contador vale 00 o 01, se predice el salto como no tomado.

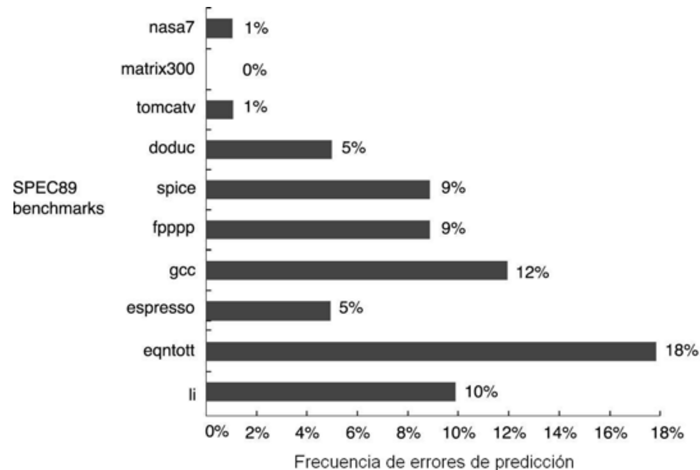
Cuando el salto se ejecuta, el valor del contador se actualiza según la figura 22. Generalizando lo anterior, podemos citar un predictor de *n* bits donde cada entrada tiene un valor de n bits (0 a 2^{n-1}), por lo que:

- Si el valor es igual o mayor a 2^{n-1} se predice la ramificación como "tomada".
- Si el valor es menor a 2^{n-1} se predice la ramificación como "no tomada".

La predicción debe fallar 2^{n-1} veces antes de modificarla. En la práctica, un predictor de dos bits es suficiente.

Una de las preguntas interesantes sería: ¿Cuánta precisión puede esperarse de un búfer de predicción usando 2 bits por entrada en aplicaciones reales?, para el SPEC89 (*benchmarking software*), un buffer de predicción con 4096 entradas resulta tener una precisión del 99% a 82%, o mejor dicho una falta predicción del 1% al 18%, al ser puesto a prueba en diferente tipo de software, tal como se muestra en la siguiente figura:

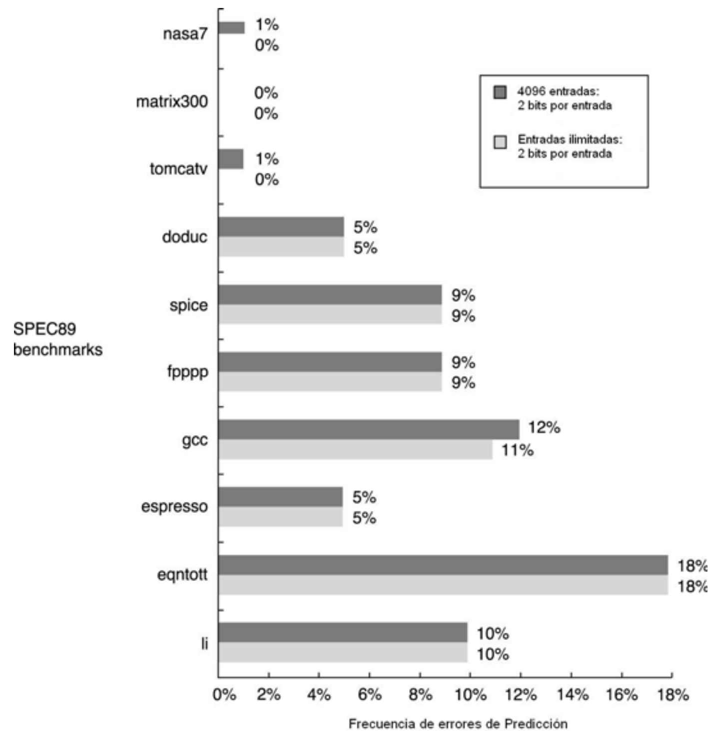
Figura 23. Precisión de predicción para un búfer de 4096 entradas y dos bits por entrada



Fuente: Elsevier Science (USA) 2003
Traducción libre del autor

Si comparamos esta configuración con un búfer de entradas ilimitadas y 2 bits de predicción, nos daremos cuenta de que el cambio en la estructura del predictor tiene un pequeño impacto, tal como es mostrado en la figura 24. Por lo que se hace necesario la búsqueda de más soluciones.

Figura 24. Comparación de un búfer de predicción de 2 bits con 4096 entradas y uno de entradas ilimitadas



Fuente: Elsevier Science (USA) 2003
Traducción libre del autor

3.3.1.1 Predictores correlacionados o de varios niveles

Este tipo de predictores toman en cuenta no sólo los bits de predicción asociados a la instrucción de salto actual, sino también el comportamiento de las últimas instrucciones de salto ejecutadas. Consideremos el siguiente fragmento de código:

```
if  (aa = =2)
aa = 0;
if (bb = = 2)
bb=0;
if (aa! = bb) {
...
```

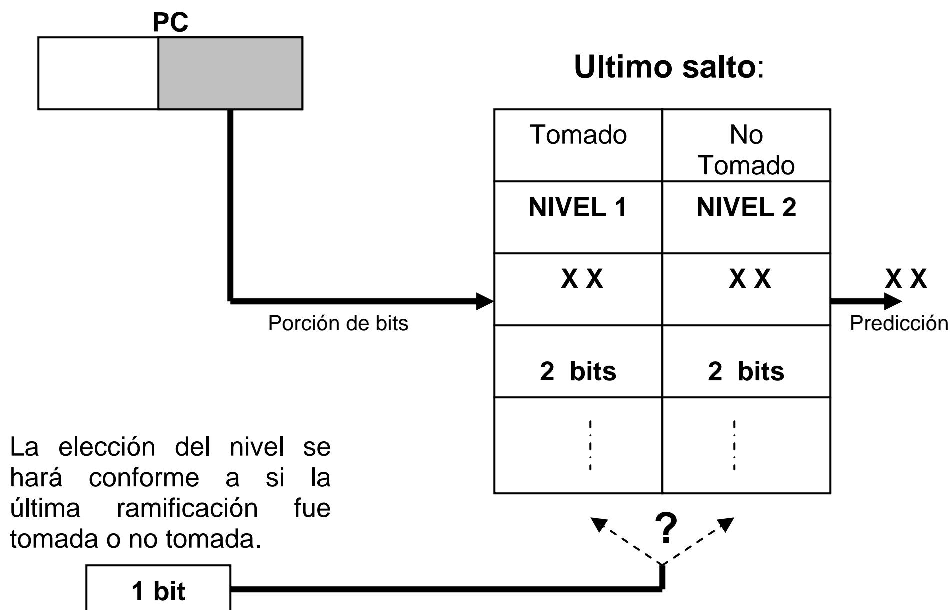
En lenguaje ensamblador, asumiendo que “aa” y “bb” son asignados a los registros R1 y R2:

```
dsubi r3,r1,#2
bnez r3,L1              ; ramificación b1 (aa!= 2), bnez= branch is not equal zero
dadd r1,r0,r0           ; aa=0
L1: dsubi r3,r2,#2
bnez r3,L2                  ; ramificación b2 (bb!=2)
dadd r2,r0,r0           ; bb=0
L2: dsub r3,r1,r2                ; R3 = aa-bb
beqz r3,L3                  ; ramificación b3 (aa = = bb), beq = branch is equal zero
```

Si los saltos *b1* y *b2* son “no tomados” (entonces *aa* y *bb* serán 0), el salto *b3* será obviamente “tomado”. Un predictor que usa solamente el comportamiento de una sola ramificación para predecir la ramificación, nunca podrá capturar este comportamiento. Por lo que es posible perfeccionar la precisión de la predicción si se usa el comportamiento reciente de otras ramificaciones en lugar de utilizar sólo la que deseamos predecir.

El comportamiento asociado a la última ramificación es descrito mediante la utilización de bits de correlación (*m)*; en el caso general (m,n) se utilizan el comportamiento de las últimas *m* ramificaciones para escoger entre 2^m predictores de ramificación, teniendo cada entrada de la tabla de predicción n bits. El búfer de predicción puede ser seleccionado usando una concatenación de la porción de la dirección de la instrucción de ramificación con los m-bits del historial global. Este tipo de implementación puede ser comprendido observando la figura 25.

Figura 25. Predictor (1,2) que utiliza el comportamiento del último salto.

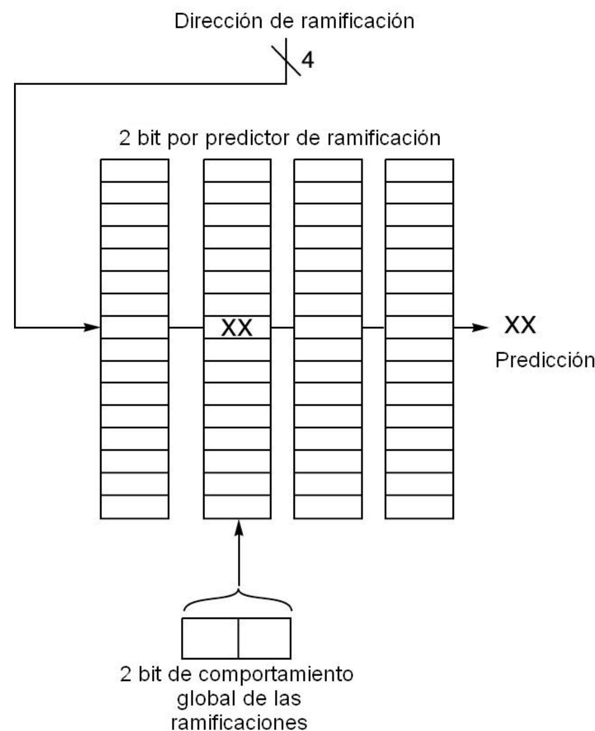


El predictor de la figura 25, con m=1 y n=2 eligirá el búfer de predicción conforme al nivel elegido mediante el bit de correlación correspondiente al comportamiento de la última ramificación, localizando por medio de la porción

de bits tomadas de la dirección de la instrucción de ramificación la posición en el búfer de predicción.

La figura 26 nos muestra un predictor (2,2) con un total de 64 entradas, localizadas por medio de una porción de 4 bits de direccionamiento y 2 bits de comportamiento global.

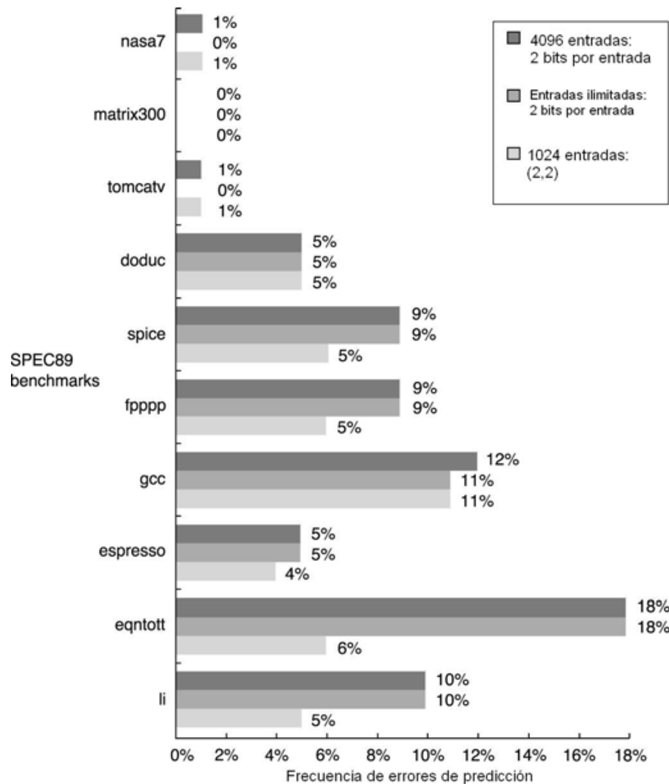
Figura 26. Predictor Correlacionado utilizando 2 bits de historia global para escoger 4 predictores para cada dirección de ramificación



Fuente: Elsevier Science (USA) 2003
Traducción libre del autor

Es notable la minimización en la frecuencia de errores de predicción al utilizar un predictor (2,2) de 1024 entradas, comparado con un predictor sin correlación de 4096 entradas de dos bits, lo que puede ser observado en la figura 27, conociéndose de esta manera la efectividad de la correlación.

Figura 27. Comparación entre predictores de 2 bits



Fuente: Elsevier Science (USA) 2003
Traducción libre del autor

3.4. Entrega de instrucciones

En un *pipeline* de alto rendimiento, especialmente uno con múltiple emisión, la predicción de ramificaciones no es suficiente. Nuestro objetivo actualmente consiste en que nuestro diseño sea capaz de entregar un flujo de alto ancho de banda de instrucciones. Para lograr ésto es necesario tomar a consideración tres conceptos importantes: el concepto de *branch-target buffer*, unidades de búsqueda y el tratamiento de ramificaciones indirectas mediante un *return address predictor*.

3.4.1. Branch-target Buffers

Entre las consideraciones llevadas hasta el momento, no hemos tomado en cuenta el hecho de que no es sino que hasta el fin de la fase de decodificación que se sabe si la instrucción tomada es un salto, donde para entonces, ya podemos saber si el salto es tomado o no tomado en muchos casos. Por lo tanto, necesitamos identificar los saltos con mayor anticipación, para ello utilizamos otra tabla, que no es más que el búfer de destinos de saltos o mejor conocido como branch-target buffer. Esta tabla se va a acceder en la fase de búsqueda con el PC (de forma simultánea a la búsqueda de la instrucción), el acceso nos dirá si la instrucción que buscamos es un salto, y si lo es, la dirección a la que salta (si se toma).

El *branch- target buffer* (BTB), está constituído de una tabla de dos columnas (tal como se muestra en la figura 28), donde en la primera de ellas se almacenan las direcciones de las instrucciones de salto (memoria asociativa), y en la segunda lo sitios a los que se salta en esos saltos si son tomados. Existiendo además de éstas dos columnas, una tercera, la cual es opcional y puede ser usada para un estado extra de bits de predicción.

Figura 28. Estructura de un *branch target buffer* (BTB)

PC

Número de entradas en el BTB

Dirección de la instrucción de salto

Dirección de la predicción

=

NO

SI

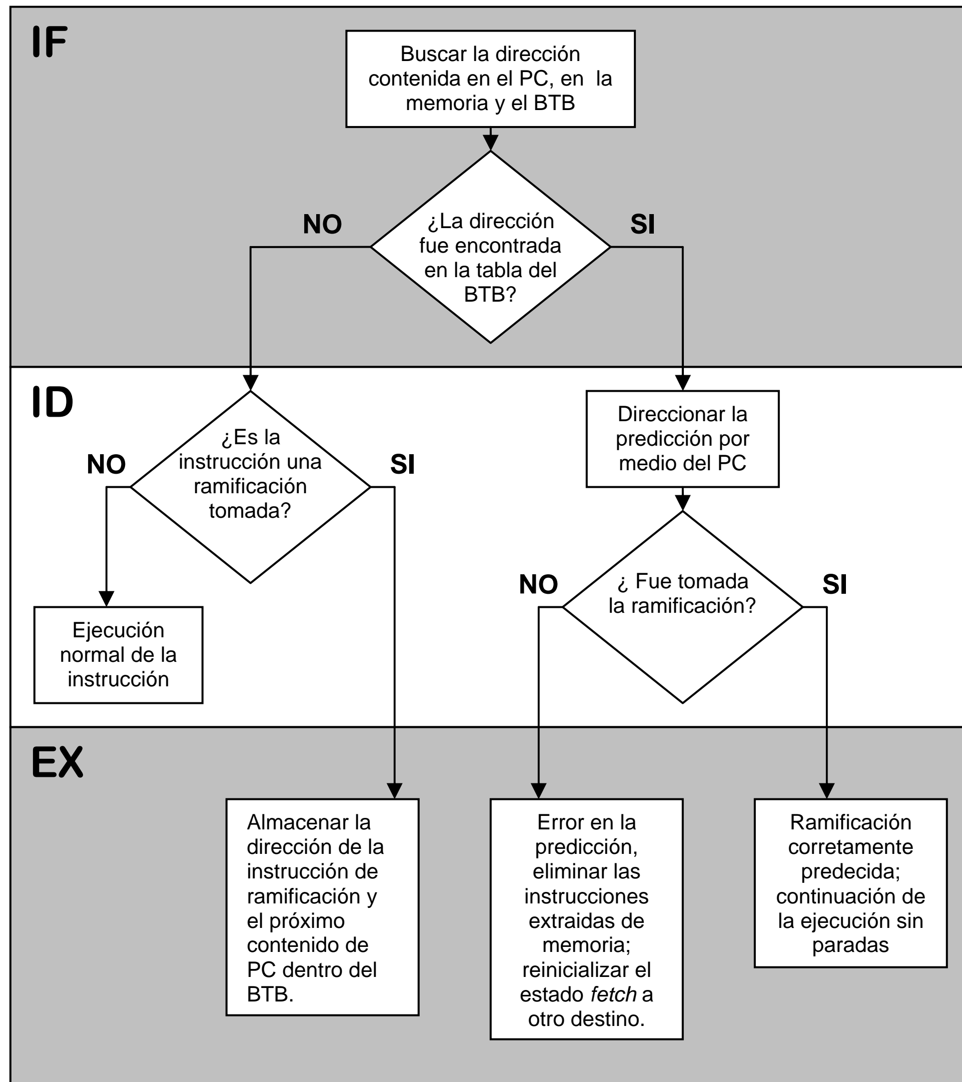
Si la instrucción es ramificación y tiene la dirección de la predicción correcta ésta es ejecutada sin paradas, sino el proceso es reinicializado.

Si la instrucción no es una ramificación se procede de manera normal y si es pero no está en la tabla ésta se almacena

bits de predicción, salto tomado o no tomado

En una unidad segmentada, con el uso de un *branch-target buffer* (BTB) debemos de considerar que los pasos en la manipulación de la instrucción se ven afectados, ya que se debe de realizar operaciones y comparaciones dentro de las etapas, por lo que las podemos ubicar en el siguiente diagrama de flujo, que presenta la siguiente figura:

Figura 29. Pasos involucrados en la manipulación de una instrucción con un BTB



3.4.2. Unidades IF integradas

Para enfrentarse con las demandas de múltiple emisión de instrucciones en procesadores, muchos diseñadores recientes han optado por implementar una *integrated instruction fetch unit*, como una unidad autónoma separada que alimenta de instrucciones al resto del pipeline. Esencialmente, esta unidad no puede ser constituida como un simple estado del pipeline, debido a la complejidad necesaria para realizar una emisión múltiple.

Una unidad IF integrada, agrega varios de los diseños vistos con anterioridad, integrando así varias funciones:

1. **Predicción de ramificaciones.** El predictor de ramificaciones llega a ser parte de la unidad IF y está constantemente prediciendo ramificaciones, manejando el estado de *fetch* en el *pipeline.*

2. **P*refetch* de instrucciones.** Para entregar múltiples instrucciones por ciclo de reloj, la unidad IF necesitará tomar instrucciones adelantadas con la ayuda del predictor de ramificaciones.

3. **Acceso de instrucciones a memoria y memoria intermedia (*buffering*).** Cuando tomamos múltiples instrucciones por ciclo, una variedad de complicaciones son encontradas pudiendo requerir el acceso múltiples líneas de *cache.* La IF provee una memoria intermedia actuando como una unidad de demanda suministrando instrucciones al estado de emisión tanto como es necesario y la cantidad necesaria.

Como diseñadores debemos intentar incrementar el número de instrucciones ejecutadas por ciclo de reloj, por lo que debemos de considerar que el estado IF llegará a ser un cuello de botella conforme incrementemos el flujo, lo que conduce a la necesidad de implementar nuevas ideas para que la entrega de instrucciones se realice al *rate* necesario.

3.4.3. *Return address predictors*

Otro de los métodos que los diseñadores han estudiado e incluido en muchos procesadores recientes es la técnica de predecir saltos indirectos, es decir, saltos cuya dirección del destino varía en momento de corrida. A través de lenguajes de alto nivel generamos saltos indirectos, tal es el caso de llamadas a procedimientos (*calls)*, los *gotos,* instrucciones de *select* ó *case* en algunos programas y la gran bastedad de saltos indirectos provenientes de retornos a procedimientos.

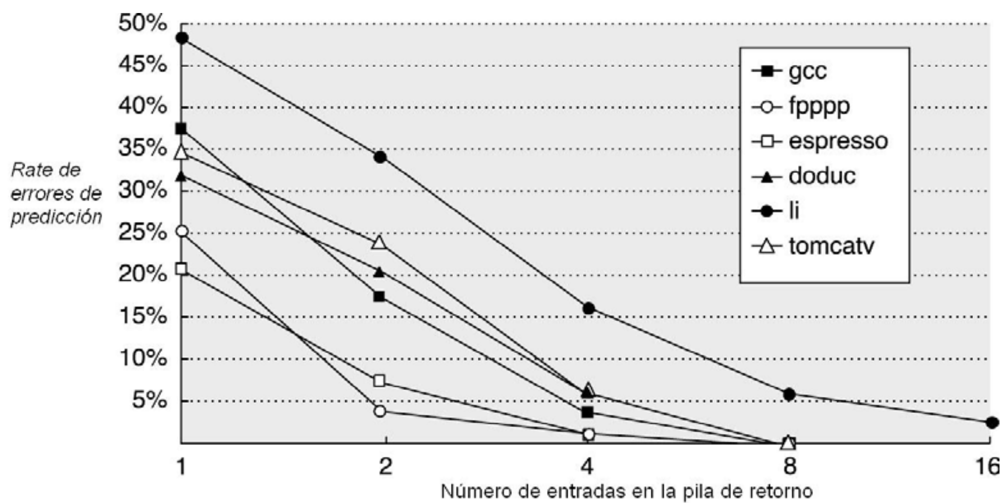
El problema consiste en que el destino no va a ser siempre el mismo por lo que el procedimiento puede ser llamado de varios puntos. Para superar este problema es introducido un pequeño búfer de retorno de direcciones, operando como una pila (*stack,* almacenada en la CPU), tomando una estructura LIFO (*last in, first out*), es decir el último en entrar es el primero en salir, almacenando las direcciones de retorno de la siguiente manera:

- Se introduce una nueva dirección cuando se ejecuta una instrucción de llamada.

- La dirección es retirada en el retorno.

En la figura 30 podemos observar la precisión de predicción de un típico *return address buffer*, donde la precisión varía conforme al tipo de aplicación y la número de entradas.

Figura 30. Precisión de predicción para un *return address predictor* operando como pila



Fuente: Elsevier Science (USA) 2003
Traducción libre del autor

3.5. Ejecución especulativa basada en *hardware*

Para lograr una ejecución especulativa basada en *hardware*, en la cual podamos aprovechar al máximo el ILP, es necesario combinar tres ideas claves:

1. **Predicción dinámica de ramificaciones:** con el objeto de elegir cuales instrucciones serán ejecutadas.

2. **Especulación:** para permitir la ejecución de instrucciones, antes que la dependencia de control sea resuelta (con la habilidad de deshacer los efectos de una secuencia especulada incorrectamente).

3. **Planificación dinámica de instrucciones:** el enfoque integrado en varios procesadores (Power PC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II/III/4, Alpha 21264 y AMD K5/K6/Athlon), consiste en implementar la ejecución especulativa basándose en la planificación dinámica del algoritmo de Tomasulo.

Para implementar el algoritmo de Tomasulo especulativo, se debe tomar en consideración la separaración de los resultados entre instrucciones para ejecutar una instrucción especulativamente desde la actual completación de ésta. Haciendo la separación podemos permitir que una instrucción sea ejecutada y desvíe el resultado a otra instrucción, sin permitir que la instrucción ejecute cualquier actualización que no pueda deshacerse hasta conocer que la instrucción ya no es especulativa. Cuando una instrucción ya no es especulativa, con toda libertad se puede permitir que determinado registro o la memoria sean actualizados. Este paso adicional en la secuencia de la ejecución de la instrucción es llamado *instruction commit*.

La idea clave detrás de la implementación de la especulación, es permitir que las instrucciones sean ejecutadas fuera de orden pero forzar a que sean entregadas en orden y prevenir cualquier acción irrevocable (tal como la actualización de un estado o la toma de una excepción) hasta que se ejecute el proceso de *instruction commit.*

Cuando es adherida especulación al algoritmo de Tomasulo, es necesario separar el proceso de completación de la ejecución de la instrucción del proceso de *instruction commit*, ya que instrucciones especuladas, pueden finalizar la ejecución antes de que estén listas para el proceso de *commit*. Agregando ésta fase de *commit* a la secuencia de ejecución de la instrucción son requeridos algunos cambios de *hardware (hardware buffers)* que mantengan el resultado de las instrucciones que han finalizado su ejecución pero no han llevado a cabo el proceso de *commit*. Este búfer implementado mediante *hardware* lleva el nombre de *reorder buffer* (ROB).

El *reorder buffer* (ROB), es necesario también para transmitir resultados entre instrucciones especuladas, éste provee registros adicionales de la misma manera que las estaciones de reserva en el algoritmo de Tomasulo, extendiendo el conjunto de registros. El ROB mantiene el resultado de una instrucción entre el tiempo de operación asociado al completamiento de la instrucción y el tiempo en que la instrucción lleva el proceso de *commit*. El ROB es similar al búfer de almacenamiento (*store buffer)* descrito en el algoritmo de Tomasulo por lo que podemos integrar la función de éste dentro del ROB por simplicidad.

Cada entrada en el ROB debe de poseer al menos cuatro campos:

1. **El tipo de instrucción:** este campo indica si la instrucción es una ramificación (y no tiene ningún resultado del destino), un almacenamiento (la cual tiene una direccion de memoria como destino), o una operación de registros (operación de ALU ó carga, la cual tiene registros como destino).

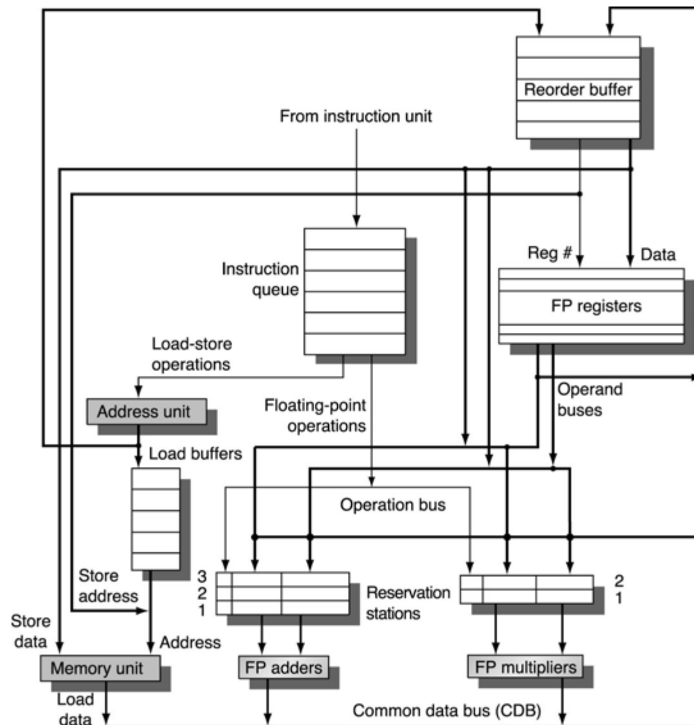
2. **Destino:** este campo proporciona el número de registro (para cargas y operaciones de ALU) o la dirección de memoria (para almacenamientos) donde el resultado de la instrucción deberá ser escrito.

3. **Valor:** este campo es utilizado para mantener el valor del resultado de la instrucción hasta que la instrucción lleve el proceso de *commit.*

4. ***Ready field*:** este campo indica que la instrucción a completado la ejecución, por lo que el valor está listo.

La figura 24, nos muestra una estructura de *hardware* de un procesador incluyendo el ROB. El ROB completamente reemplaza los búfers de almacenamiento. A pesar de que la función de renombramiento de las estaciones de reserva es reemplazada por el ROB, todavía necesitamos un lugar intermedio (búfer) para las operaciones (y operandos) entre el tiempo en que ellas son emitidas y el tiempo en que empieza la ejecución; esta función es todavía proporcionada por las estaciones de reserva. De aquí, todas las instrucciones tendrán una posición en el ROB hasta el proceso de *commit*, teniendo como opción usar el número entrada al ROB en lugar de usar el número de la estación de reserva.

Figura 31. Estructura básica de una unidad de punto flotante usando el algoritmo de Tomasulo especulativo.



Fuente: Elsevier Science (USA) 2003
Traducción libre del autor

Los pasos involucrados en el algoritmo de Tomasulo especulativo son los siguientes:

- ***Issue*****:** toma la instrucción de la cola. Emite la instrucción si existe una estación de reserva y ROB disponibles. Actualiza el control de entradas, para indicar que búfers están en uso. Este estado es algunas veces llamado *dispatch* en algunos procesadores planificados dinámicamente.

- **Ejecución:** opera sobre los operandos disponibles, si no están disponibles monitorea el CDB hasta que estén disponibles. Este paso revisa las dependencias RAW.

- **Escritura de resultados:** cuando el resultado está disponible, escribe sobre el CDB y desde el CDB dentro el ROB, así como también a cualquier estación de reserva que esté esperando el resultado.

- ***Commit:*** en esta etapa se pueden dar tres diferentes situaciones, dependiendiendo de ellas, la acción tomada. En el caso normal, cuando una instrucción alcanza la cabeza del ROB, y el resultado está presente en el búfer, el procesador actualiza el registro con el resultado y remueve la instrucción desde el ROB. Cuando la instrucción es un almacenamiento, el proceso es similar excepto que la actualización es hecha a la memoria en lugar que a un registro. Cuando una una ramificación con predicción incorrecta alcanza la cabeza del ROB, esta indica que la predicción fue incorrecta, el ROB es vaciado y la ejecución es reiniciada al destino correcto de la ramificación; si la ramificación fue correctamente predecida, la ramificación es finalizada. En algunos procesadores esta fase toma el nombre de *completion* o *graduation*. Si una instrucción paso por el proceso de *commit*, la entrada en el ROB es regenerada y el destino (memoria o registro) es actualizado, eliminando la necesidad de la entrada de ROB. Si el ROB se llena, la emisión es parada hasta que exista una entrada libre.

En la práctica, esquemas que utilizan especulación, intentan recuperarse lo más pronto posible después de que se haya ejecutado una predicción incorrecta.

Esta recuperación puede realizarse por medio de un borrado del ROB para todas las entradas que aparecieron después del fallo, permitiendo así que todas las instrucciones anteriores continúen y reinicializando el *fetch* al destino correcto. En procesadores especulativos, el rendimiento es mas sensible a los mecanismos de predicción de ramificaciones, así el impacto de una predicción errónea en cierta manera, será alto.

Las excepciones son tomadas pero no reconocidas hasta que se está listo para el proceso de *commit*. Si en una instrucción especulada, surge una excepción, la excepción es guardada en el ROB. Si surge una predicción de ramificación errónea, la instrucción no deberá ser ejecutada, por lo que la excepción será borrada con la instrucción cuando el ROB sea borrado.

3.5.1. Consideraciones de diseño

3.5.1.1 Renombre de registros

El renombre de registros es la técnica que se utiliza para eliminar los riesgos WAW y WAR y aumentar el paralelismo en la ejecución de los programas. En el código estático de un programa, las instrucciones almacenan sus resultados en los lugares *lógicos* de la arquitectura (registro destino o memoria). Pero durante la ejecución dinámica del programa se utilizan otros elementos de almacenamiento que ya ocupan lugares *físicos* y que pueden ser distintos de los lógicos. Las instrucciones utilizan esos lugares físicos para transmitirse sus resultados. Por tanto, es posible la existencia simultánea de varias versiones especulativas del mismo registro lógico.

La técnica del renombre *asigna* a todo registro lógico destino, un elemento de almacenamiento físico para guardar de forma temporal su valor (versión, instancia). A dicho elemento de almacenamiento se le denomina *registro de renombre*. Las posteriores referencias al registro lógico se redireccionarán al registro físico durante el tiempo que éste sea válido.

En los años 60, fueron el CDC 6600 y el IBM 360/91 los primeros que incluyeron en su implementación técnicas de procesado paralelo de instrucciones. De ellos, el IBM 360/91 es el que incluye el concepto del renombre de registros en la forma en que lo describe el algoritmo de Tomasulo y que coincide con la que conocemos ahora. El diseño del renombre desarrollado en el 360/91 solo se incluyó en su unidad de punto flotante, pero era de una complejidad tal que hizo que no tuviese éxito en el mercado.

En los años 70, autores como Tjaden y Flynn, y Riseman y Foster demuestran que para extraer un paralelismo razonable entre las instrucciones, es necesario invertir grandes cantidades de *hardware*. Más tarde Keller vuelve a introducir el renombre describiendo como implementarlo y extendiéndolo a todo tipo de instrucciones con registro destino, pero esta técnica no se utiliza hasta comienzos de los 90. Los primeros procesadores superescalares como HP-PA 7100, Sun SuperSparc, DEC Alpha 21064, MIPS R8K e Intel Pentium, inician la ejecución de instrucciones en orden y no utilizan renombre. Apareció luego de forma restringida (aplicado solo a determinados tipos de intrucciones) en los Power1, Power2, PowerPC 601 y en NextGen Nx586. A comienzos de 1992 aparece tal y como lo conocemos ahora en los últimos modelos de IBM ES/9000 y luego en el PowerPC 603. En la actualidad, el renombre de registros se considera una característica estándar que aparece en muchos procesadores superescalares a excepción de la línea Sun UltraSparc y los procesadores de Alpha que preceden al DEC Alpha 21264.

Una característica de las técnicas de renombre es el elemento de almacenamiento físico utilizado para almacenar las versiones consolidadas y especulativas. Las grandes opciones de diseño se diferencian por utilizar almacenes más o menos distribuidos. En la opción más distribuida, el almacén de renombre se divide en *estaciones de reserva* asociadas a las unidades funcionales. Las estaciones de reserva contienen los operandos fuente (consolidados o especulativos). A su vez los valores consolidados suelen estar centralizados en un banco de registros que se actualiza al consolidar cada instrucción. Ejemplos de esta opción es la familia Pentium Pro de Intel (Pentium II y Pentium III).

En la opción centralizada el almacén de renombre puede contener únicamente versiones especulativas o puede incorporar también las versiones consolidadas. En el primer caso, las versiones especulativas pueden organizarse en orden programa en un ROB (*Reorder Buffer*) con capacidad para alimentar a las unidades funcionales; ejemplos de esta opción son la familia HP-8000 y la familia K6 de AMD. De nuevo, los valores consolidados suelen estar centralizados en un banco de registros que se actualiza en *commit*.

En el segundo caso, los dos conjuntos de registros (lógicos y de renombre) se juntan en un *banco de registros mezclado*. Ejemplos de este caso son IBM ES/9000, IBM Power1 y Power2, MIPS R10K/12K, Compaq Alpha 21264 e Intel Pentium 4. Está opción de diseño parece tomar fuerza en procesadores recientes, porque elimina la necesidad de efectuar copia al consolidar.

Un procesador superescalar con banco de registros mezclado, asigna un registro físico a cada registro lógico destino en decodificación y guarda dicha asignación en una tabla de mapeo. Esta actividad a veces se aísla de la decodificación y se concentra en una etapa separada que se denomina Renombre.

La tabla de mapeo posee tantas entradas como número de registros lógicos tiene la arquitectura y guarda el identificador del registro físico asignado a cada lógico.

Esta opción está implementada en procesadores superescalares actuales como MIPS R10K/.../18K, DEC Alpha 21264 e Intel Pentium 4 (aquí se llama *Frontend Register Alias Table*). En los procesadores citados el proceso de renombre se extiende tanto al banco de registros entero como al de coma flotante, duplicándose todos los recursos necesarios (tabla de mapeo, control de asignación/liberación, etc.).

Para *liberar* los registros de renombre cuando ya no se necesitan es necesario conocer con seguridad que ninguna instrucción en el procesador va a utilizar más el valor almacenado en el registro. La especulación y la necesidad de cumplir un modelo de interrupciones preciso, hace que sea especialmente complejo el detectar de forma segura cuando se realiza el último uso de un registro. Los procesadores superescalares actuales han optado por implementar lo anterior de la forma más sencilla que consiste en esperar a que otra instrucción que escribe al mismo registro lógico alcance su etapa de *commit*. En este momento es seguro que ninguna instrucción va a utilizar más el valor del anterior registro de renombre. Las instrucciones durante su ejecución permanecen en una determinada *estructura de reordenación* en espera de recuperar el orden secuencial original del programa. Una implementación para liberar registros consiste en guardar junto con la instrucción el identificador de su anterior registro físico en esa estructura de reordenación (*History Buffer indirecto*). Esta opción se utiliza en los procesadores MIPS R10K/12K, DEC Alpha 21264 e Intel Pentium 4.

En la *etapa de commit* se modifica el estado arquitectural de la máquina y también se recupera el orden secuencial de las instrucciones. El conjunto de acciones que se llevan a cabo dependen de la técnica que se utilice para recuperar el estado preciso de la máquina. Existen distintas técnicas que se emplean para resolver este problema. Para bancos de registros mezclados, una posibilidad de diseño consiste en utilizar *tablas de mapeo en orden* que sirven para restaurar la asignación original de los registros en orden de programa. El procesador Intel Pentium 4 implementa esta idea con el nombre de *Retirement Register Alias Table.*

3.5.1.2 Especulación a través de múltiples ramificaciones

Tres situaciones se pueden beneficiar respecto al rendimiento de nuestro procesador, especulando múltiples ramificaciones simultánemente:

- Gran frecuencia de ramificaciones en un programa
- Una cantidad significativa de saltos dentro de una ramificación
- Grandes retardos en unidades funcionales

Desarrollar un rendimiento alto en los en los primeros dos casos puede significar incluso la necesidad de ejecución de más de una ramificación en un ciclo de reloj.

Programas de bases de datos, y otros programas de cálculo de enteros menos estructurados, a menudo exhiben estas propiedades, haciendo el tema de la especulación de ramificaciones múltiples bastante importante. Igualmente, los retardos largos en las unidades funcionales pueden levantar la importancia de especular en las ramas múltiples como una manera de evitar paradas a lo largo de una estructura de *pipeline.*

Especulando ligeramente en las ramas múltiples complica el proceso de recuperación de la especulación; pero, por otra, parte provocaría una anticipación eficiente. Siendo esta manera de procesar ramificaciones, una técnica que dependerá mucho de las circunstancias y aplicaciones a las que este sometido nuestro procesador, y que hagan necesaria su implementación, debido a la cantidad de *hardware* involucrada.

4. CONSIDERACIONES DE DISEÑO GENERALES EN BASE A LIMITACIONES DEL ILP EN *HARDWARE*

Las técnicas de paralelismo a nivel de instrucción realizadas mediante *hardware*, han logrado un alto perfeccionamiento del rendimiento en procesadores a través de los años, siendo en algunos de los diseños en la actualidad combinadas con algunas técnicas de *software*.

Varias de las limitaciones del ILP están intrínsecamente relacionadas con el modelo utilizado y las suposiciones tomadas en el desarrollo del diseño, tanto la cantidad como la cualidad de los elementos utilizados, determinarán la calidad de desempeño que se presente y marcaran las limitaciones, debido a los aspectos realizables en nuestro diseño.

4.1. Consideraciones respecto al modelo de *hardware*

Una de las maneras de conocer realmente las limitaciones a las que nos debemos enfrentar en el momento de diseñar, consiste en pensar en el modelo ideal de nuestros objetivos y partir de allí para determinar que asunciones son realizables. Tal es el caso del procesador con ILP ideal, en el cual podemos tomar las siguientes suposiciones:

1. Existencia de un gran número de registros disponibles para el renombramiento, eliminando de esta manera toda dependencia WAW y WAR por lo que un ilimitado número de instrucciones pueden ser ejecutadas simultáneamente.

2. Predicción perfecta de todas las ramificaciones

3. Especulación perfecta y un ilimitado búfer de instrucciones disponible para la ejecución

4. Que todas las direcciones de memoria sean exactamente conocidas y que una carga pueda ser movida antes que un almacenamiento tal que las direcciones no sean idénticas (memoria sin ambigüedades)

Juntando las disposiciones anteriores, nos damos cuenta de que en este procesador, cualquier instrucción en la ejecución del programa puede ser planificada sobre el ciclo inmediato siguiente a la ejecución del predecesor sobre el cual ésta depende, y cualquier especulación se podrá realizar sin ningún problema, tomando en cuenta también un número ilimitado de cargas y almacenamiento emitidos en un ciclo de reloj, latencias de un ciclo de reloj para todas las unidades funcionales y finalmente la suposición de *caches* perfectas lo que asegura que todas las cargas y almacenamientos se completan en un ciclo.

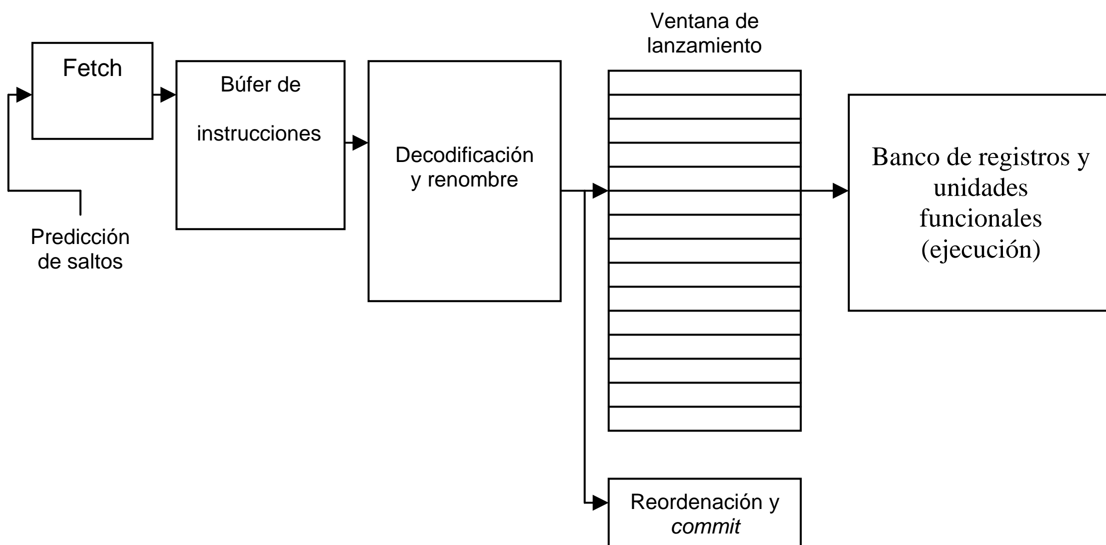
Sin embargo, varias de estas asunciones se ven limitadas en los procesadores realizables en diferentes aspectos, por lo que dependerá del modelo de *hardware* tomado que tan limitado estará nuestro ILP con determinada aplicación.

4.2. Limitaciones en el tamaño de ventana

En procesadores reales, la emisión de instrucciones ocurre en orden, donde la búsqueda de instrucciones y la predicción de saltos es realizada en la etapa de *fetch*, luego las dependencias de datos falsas son tomadas y eliminadas por el proceso de renombramiento, una vez las instrucciones son emitidas el conjunto de instrucciones es examinado para su ejecución simultánea (fuera de orden) en la ventana de lanzamiento, para luego realizar la ejecución de las operaciones de memoria y reordenación y *commit* de las instrucciones.

De ésta manera el tamaño de ventana quedará determinado por el conjunto de instrucciones examinadas como candidatas para ser ejecutadas. Ocupando así la ventana de lanzamiento un lugar importante en la ejecución de instrucciones, tal como es mostrado en el ejemplo de la siguiente figura.

Figura 32. Diagrama de bloques general del proceso de emisión y ejecución de instrucciones.



Al eliminar parte de las restricciones de precedencia entre las instrucciones, se consigue aumentar el número de instrucciones de la ventana de lanzamiento que pueden ejecutarse en paralelo. Sin embargo, a pesar de poder ejecutar varias instrucciones, se presentan al planificador una serie de restricciones que debe de tomar a consideración en el diseño del tamaño de ventana, tales como:

- El espacio físico requerido para almacenar instrucciones
- El número de comparaciones necesarias, para determinar dependencias
- El límite de emisiones por ciclo, debido a dependencias estructurales

A simple vista, el tamaño de ventana directamente limita el número de instrucciones que empezaran a ser ejecutadas en un ciclo dado. Sin embargo, este punto de vista en la realidad es confrontado de diferente manera, ya que los procesadores reales están limitados por el número utilizado de unidades funcionales (un procesador puede tomar mas de dos referencias a memoria por ciclo de reloj o más de dos operaciones de punto flotante), también está limitado por el número de buses y registros de acceso a puertos, lo cual reduce el número de instrucciones que empezarán a realizarse en el mismo ciclo de reloj, causando así, que el tamaño de ventana venga a ser de menos ayuda, ya que se hace necesaria la implementación de nuevo *hardware*, llevando esto de antemano otras limitaciones.

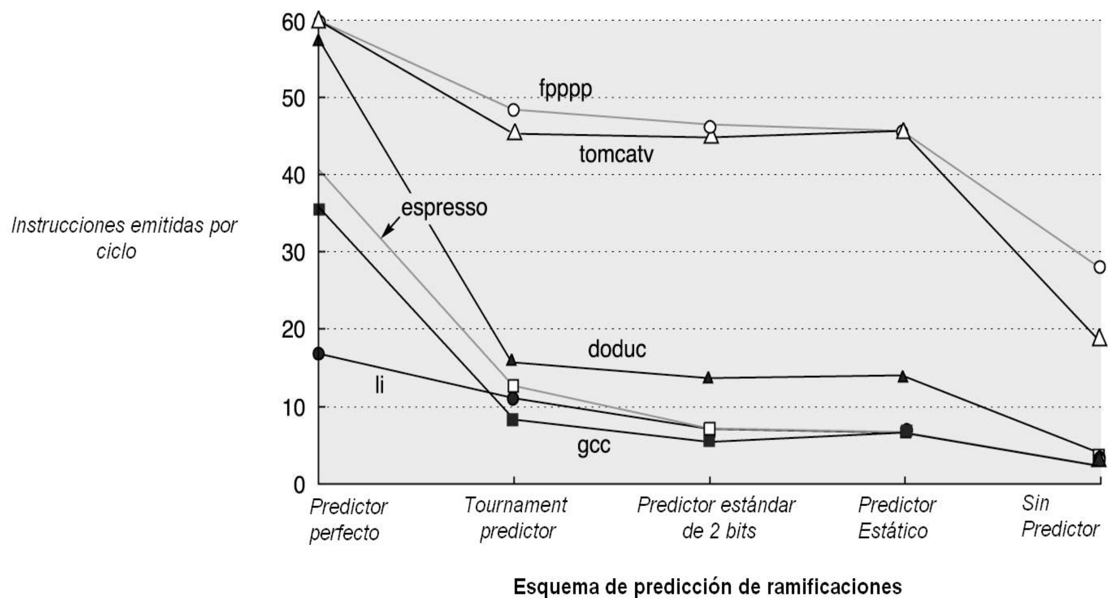
De esta manera, el máximo número de instrucciones que pueden ser emitidas, empezar su ejecución, o llevar a cabo el proceso de *commit* en el mismo ciclo de reloj es usualmente más pequeño que el tamaño de ventana.

Así, el número de restricciones para una posible implementación de una múltiple emisión es bastante grande, por lo que el límite del tamaño de ventana estará siempre directamente afectado por los obstáculos encontrados en el desarrollo del ILP.

4.3. Efectos en la predicción de saltos y ramificaciones reales.

En un procesador ideal se asume que las ramificaciones son perfectamente predecidas, es decir, que la salida de cualquier ramificación en el programa es conocida antes que la primer instrucción sea ejecutada, por supuesto ningún procesador real puede llevar a cabo esto. La figura 33 nos muestra la comparación de diferentes esquemas de predicción, variando desde el predictor perfecto hasta la emisión de instrucciones sin predictor.

Figura 33. Efecto de diferentes esquemas de predicción



Fuente: Elseiver Science (USA), 2003

Traducción libre del autor

Los cinco niveles de ramificación mostrados en la figura 33 son los siguientes:

1. **Predictor perfecto:** todas las ramificaciones y saltos son perfectamente predecidos al inicio de la ejecución.

2. ***Tournament-based predictor*:** este esquema utiliza juntamente un predictor correlacionado de 2 bits y un predictor no correlacionado de 2 bits, con un selector el cual escoge el mejor predictor para cada ramificación. El búfer de predicción contiene 2^{13} (8K) entradas, donde cada una consiste de 3 campos de 2 bits, dos de los cuales son predictores y el tercero es el selector. El predictor correlacionado es indexado usando la conjunción lógica XOR de la dirección de la ramificación y la información del historial de la ramificación. El predictor no correlacionado es un predictor estándar de 2 bits indexado por la dirección de la ramificación.

   La tabla selectora es también indexada por la dirección de la instrucción de ramificación especificando cual predictor va a ser utilizado. El selector es incrementado o decrementado tal como se hace en un predictor de 2 bits estándar. Este predictor, el cual utiliza un total de 48 Kbits, lleva a cabo una precisión del 97% para éstos seis SPEC *benchmarks*; este predictor es compatible en estrategia, con los utilizados en la actualidad. El predictor para retornos esta hecho con un par de predictores de 2000 (2K) entradas y organizado como un búfer circular para predecir los retornos, organizado como un predictor estándar (en caso de una instrucción *case* o un cálculo de *gotos*).

3. **Predictor estándar de dos bits con 512 entradas:** en adición asumimos un búfer de 16 bits para predecir retornos.

4. **Predictor estático:** Un predictor estático utiliza el perfil histórico del programa y predice que la ramificación es siempre tomana o siempre no tomada en base al perfil.

5. **Sin predictor:** en este caso ningún predictor es usado, aunque los saltos son aún predecidos. El paralelismo es muy limitado en este esquema.

La figura 33 a pesar de que asume varios aspectos del procesador ideal, nos muestra una perspectiva de la cantidad de ILP que se ejecuta con diferentes esquemas de predictores, nos permite visualizar la importancia de la utilización de éstos en el diseño de procesadores para alcanzar un mejor nivel de ILP en base a instrucciones emitidas por ciclo y nos permite observar la dependencia existente respecto a la naturaleza de la aplicación, tal es el caso de los *benchmarks*: fpppp y tomcatv, los cuales tienen operaciones de punto flotante intesivas con un número reducido de ramificaciones permitiendo un mayor rendimiento, prediciendo en buena manera pocas que existen

4.4. Los efectos de registros finitos

La tendencia actual de explotar cada vez más ILP de los programas, provoca un incremento de la complejidad en el diseño de las arquitecturas fuera de orden. Un *hardware* más complejo tiende a aumentar el camino crítico del procesador y por lo tanto a limitar su velocidad. Son varias las partes de la arquitectura cuya complejidad aumenta con el incremento del ILP, por lo que procesadores con grandes ventanas de lanzamiento también van a precisar de un banco de registros enorme, donde el banco de registros contiene tanto las versiones que ya representan el estado preciso de la máquina (versiones no especuladas) como las versiones que todavía son especuladas.

El banco de registros mezclado en un procesador que utiliza renombramiento, contiene un número determinado de registros (P) y de puertos (T), por lo que dependiendo del tamaño del banco de registro, nos confrontamos a las siguientes consideraciones en de diseño:

- El tiempo de acceso al banco de registros puede afectar al tiempo de ciclo del procesador. Tal es el caso del procesador Intel Pentium 4, que dedica dos ciclos a la lectura del banco de registros.

- El tamaño del banco de registros (P registros) y su número de puertos (T puertos en total, los de lectura más los de escritura) determinan tanto el área que ocupa en el procesador como su tiempo de acceso y consumo de energía.

Una de las soluciones más directas para tratar las consideraciones expuestas, consiste en la reducción de P,T o los dos, limitándonos a un número finito de registros y puertos, buscando que exista un equilibrio entre la disminución de *IPC* (instrucciones que hacen el *commit* por ciclo) y el incremento de *IPS* (instrucciones por segundo) obtenido, causando una limitante en el rendimiento requerido.

4.5. Limitaciones del ILP para procesadores realizables

El éxito de varias técnicas utilizadas en la planificación de instrucciones depende de la habilidad de ejecutar varias instrucciones de forma simultánea. De éste modo se aprovecha el paralelismo a nivel de instrucciones (ILP).

Una de las limitaciones más importantes, es el paralelismo de la máquina que es una medida de la capacidad del microprocesador para aprovechar el ILP. De acuerdo a todas las asunciones para el procesador ideal, podemos considerar que en determinado momento algunas de estas pueden ser cumplidas, dependiendo en cierta manera de los aspectos siguientes:

- La naturaleza de la aplicación que se este realizando ya que algunas aplicaciones tienden a tener gran nivel de paralelismo, dependiendo del contenido estructural de éstas (cantidad de ramificaciones, de operaciones de punto flotante, operaciones de enteros y de la cantidad de dependencia de datos que se presente)

- La precisión en el momento de la especulación y sofisticación de los algoritmos utilizados en la planificación

- La optimización en el uso de los registros

- El *Hardware* disponible

El paralelismo puede ser visto como una alternativa en la mejora en el rendimiento de un procesador, tomando el principio de que si las tareas no se pueden hacer más rápidas es necesario hacer más tareas a la vez, solventando también de esta manera el problema causado por dependencias verdaderas ya que durante se resuelve la dependencia es posible ejecutar otras instrucciones independientes, logrando ejecutar más instrucciones por ciclo de reloj.

Independientemente, del método usado para explotar el ILP, existen limitaciones potenciales que surgen de limitantes físicas, tal es el hecho de:

- Retardos entre interconexiones
- Consumo y disipación de energía

En el proceso de miniaturización, aunque los transistores decrezcan cada vez más de tamaño, las líneas conductoras en un circuito integrado no decrecen en gran manera. En particular, el retardo de una señal en una línea conductiva (*wire delay*) incrementa en proporción al producto de su resistencia y su capacitancia, siendo esta relación de bastante complejidad debido a que depende de la geometría de la línea, la carga sobre ésta e incluso la adyacencia a otras estructuras.

En los últimos años el *wire delay* a llegado a ser una de las mayores limitantes en el diseño de circuitos integrados a gran escala y es a menudo más crítico que el retardo en la conmutación en transistores, causando que grandes fracciones del ciclo de reloj tengan que ser consumidas por el retardo en la propagación de señales, creando retos adicionales al diseñador. Tal es el caso del Pentium IV el cual asigna 2 estados de su *pipeline* para propagar señales a través del chip, segmentando de ésta manera dicho retardo.

La potencia consumida también crea grandes retos en la miniaturización, en procesadores modernos, el mayor consumo de energía se encuentra en la conmutación de transistores, siendo la energía requerida por transistor, proporcional al producto de la carga capacitiva del transistor, la frecuencia de conmutación y al cuadrado del voltaje. El incremento de transistores que conmutan y su frecuencia, llevan en conjunto al crecimiento de la energía consumida. Soluciones a ello, han llevado a la implementación de un control dinámico de los recursos de *hardware* con el procesamiento de estructuras adaptivas de *hardware*, con el objeto de administrar la energía en el procesador, tomando en la actualidad un papel importante la potencia estática provocada por corrientes de fuga en el estado de apagado de los transistores.

A pesar de las limitaciones presentadas con anterioridad, con el desarrollo a través de los años, se han logrado admirables características, tal es el caso de los procesadores mostrados en la tabla XIV.

Tabla XIV. Características de hardware de procesadores de los últimos años

Procesador	Año	Frecuencia De reloj (MHz)	Potencia (Watts)	Cantidad de Transitores (Millones)	Tamaño de Ventana	Registros para renombre enteros/FP	Capacidad de emission: máxima/memoria/ enteros / FP/ ramificaciones	Capacidad del Buffer de predicción de ramificaciones	Cantidad de Estados del *Pipeline* (enteros/load)
MIPS R14000	2000	400	25	7	48	32/32	4/1/2/2/1	2K x 2	6
**UltraSPARC III	2001	900	65	29	No Aparece	Ninguno	4/1/4/3/1	16K x 2	14/15
Pentium III	2000	1000	30	24	40	Total: 40	3/2/2/1/1	512 entradas	12/14
Pentium 4	2001	1700	64	42	126	Total: 128	3/2/3/2/1	4K x 2	22/24
**HP PA 8600	2001	552	60	130	56	Total: 56	4/2/2/2/1	2K x 2	7/9
Alpha 21264B	2001	833	75	15	80	41/41	4/2/4/2/1	multinivel	7/9
Power PC 7400 (G4)	2000	450	5	7	5	6/6	3/1/2/1/1	512 x 2	4/5
AMD Athlon	2001	1330	76	37	72	36/36	3/2/3/3/1	4K x 9	9/11
** IBM Power 3-II	2000	450	36	23	32	16/24	4/2/2/2/2	2K x 2	7/8

** Estos procesadores poseen emisión dinámica pero no soportan especulación

Fuente: Hennesy, John L. y David A. Patterson. Computer Architecture: A Quantitative Aproach. Pág. 284. *Traducción libre del autor.*

En la tabla XIV son mostradas varias características claves que caracterizan a un procesador como tal, permitiendo observar cuantitativamente y tomar una concepción de acuerdo con la cantidad, de los límites a los que han estado sometidos procesadores en los últimos años, teniendo de esta manera una panorámica del asunto.

Además de lo anterior, existen varios factores externos al procesador, relacionados íntimamente con el desarrollo del ILP, entre los que podemos mencionar:

- Desarrollo del principio de localidad (los programas tienden a reusar los datos y las instrucciones que fueron usados recientemente), mediante el establecimiento de diferentes niveles y jerarquías de memoria, evitando ambigüedades en los mismos.

- Utilización de multiprocesadores, para una mayor explotación del paralelismo.

Son éstas un grupo más de las técnicas utilizadas para el incremento del rendimiento del sistema como tal, las cuales al igual que las consideraciones internas de *hardware*, también presentan varias consideraciones y limitantes de diseño, convirtiéndose en éste caso de esta manera otro tema de estudio.

5. HERRAMIENTAS DE SIMULACIÓN COMO BASE PARA EL DISEÑO

La simulación por computador es una herramienta básica en muchos campos de la ciencia y la ingeniería, para complementar (y a veces, sustituir) a otras técnicas como el estudio analítico y el desarrollo de prototipos.

Involucra la generación de una historia artificial del comportamiento del sistema y a partir de dicha historia se efectúan inferencias relativas a las características operacionales del sistema real que representa. Permitiendo así, describir y analizar el comportamiento del sistema real, y responder ciertas interrogantes para apoyar el diseño de sistemas reales.

5.1. Conceptos generales de simuladores utilizados

La simulación es una metodología que permite apoyar la toma de decisiones, de dos diferentes maneras:

- En el diseño de sistemas, antes de que sean construidos.
- Probando políticas de operación, antes que estas sean implantadas.

Por si misma, la simulación, no resuelve los problemas, sino que ayuda a:

- Identificar los problemas relevantes.
- Evaluar cuantitativamente las soluciones alternativas.

Llegando así a ser un modelo que representa acciones que interaccionan a lo largo del tiempo.

5.1.1. Tipos de simuladores

Por muchos años, la simulación a diferentes niveles de abstracción ha jugado un papel clave en el diseño de sistemas de computadoras. Son numerosas las razones compeliendo para implementar los simuladores. Es indiscutible, el uso de simuladores a lo largo de todas las fases del ciclo de un diseño determinado, como una herramienta útil, con el objeto de facilitar muchos aspectos, tal como es descrito a continuación:

1. Inicialmente, durante un diseño de alto nivel la simulación se usa para delimitar el diseño y establecer alternativas realizables y factibles, encontrando de ésta manera la actuación competitiva de los objetivos.

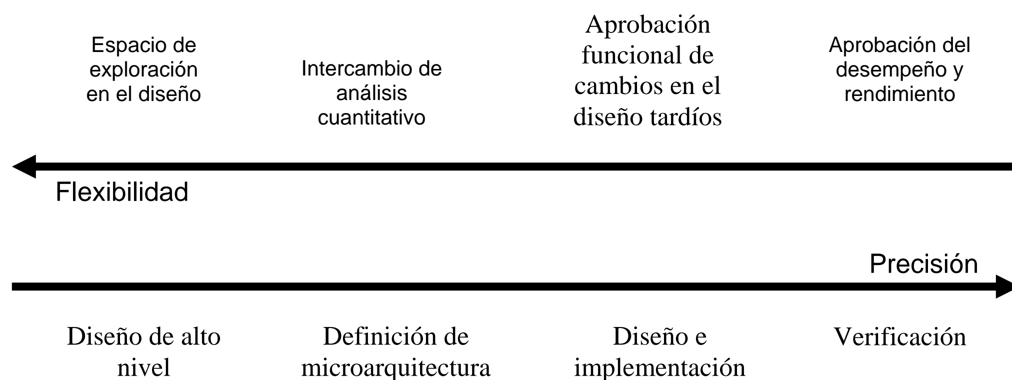
2. Después, durante la definición de la microarquitectura, un simulador proyecta una guía de expectativas, habilitando la comparación cuantitativa de varias alternativas.

3. Durante la implementación del diseño, los simuladores son empleados para poner a prueba varios aspectos y mostrar una expectativa funcional.

4. Finalmente, los simuladores proporcionan una referencia útil del comportamiento y rendimiento, una vez el *hardware* este disponible.

A través del avance en las fases de diseño, la precisión como una medida de fidelidad del simulador incrementa como efecto natural de las necesidades de los diseñadores, también como consecuencia de ello la flexibilidad del simulador tiende a decrecer debido a que más y más características son modeladas específicamente delimitando el campo de exploración abarcado por el simulador, tal como es mostrado en la figura 34.

Figura 34. Utilización de la simulación durante el proceso de diseño



Además de la flexibilidad y la precisión, varios atributos importantes son necesarios para caracterizar un simulador o un enfoque de simulación. Estos incluyen la velocidad de simulación, la funcionalidad, su uso, y la exactitud de simulación. Donde la exactitud de simulación está estrechamente relacionada, con la semenjanza al comportamiento del diseño real y como el modelo es manejado (*driven simulation*).

Fuera del ciclo del plan industrial, los simuladores son también ampliamente utilizados en el plano académico-investigativo de arquitectura de computadoras, dentro de este contexto, los simuladores son principalmente usados como un vehículo para demostrar o comparar la utilidad de nuevos rasgos arquitectónicos, técnicas de la compilación, o técnicas en la microarquitectura, en lugar de ayudar a guiar un proyecto de un diseño real. Como resultado, los simuladores académicos raramente se usan para un diseño funcional, pero estrictamente son utilizados para la prueba de concepto, diseño de exploración espacial, o para presentar una proyección o comparación respecto a un análisis cuantitativo, presentando una serie de consideraciones importantes de diseño.

5.1.1.1 Simuladores de modelo estático (*static modeling)*

Esta es una técnica de simulación destacada por su simplicidad y su bajo costo, la cual se basa mediante perfiles que son modelados estáticamente. En esta técnica el perfil dinámico de la ejecución de un programa, el cual indica que tan a menudo cada instrucción es ejecutada, es obtenido por cualquiera de los siguientes tres métodos:

1. Usando contadores sobre el procesador, los cuales son periódicamente modificados. Esta técnica a menudo da un perfil aproximado, pero con poco porcentaje de exactitud.
2. Utilizando ejecución instrumentada, en la cual un código de instrumentación es compilado dentro del programa. Este código es usado para incrementar contadores, produciendo un perfil exacto. Esta técnica, también puede ser usada para crear una traza de direcciones de memoria accesadas, la cual es útil para otras técnicas de simulación.

3. Interpretando el programa a nivel de *set* instrucciones, compilando los conteos de instrucción en el proceso.

Una vez el perfil es obtenido, este es usado para analizar el programa en una manera estática. Obviamente, con el perfil, el conteo total de instrucciones es fácil obtener, así como conseguir sobre que tipos de instrucciones fueron ejecutados y con que frecuencia, logrando de esta manera, para procesadores sencillos obtener una aproximación del CPI. Esta aproximación es calculada modelando y analizando la ejecución de cada bloque básico de instrucciones y entonces computando una estimación global del CPI. A pesar de que este modelo simple, ignora el comportamiento de la memoria y tiene severas limitaciones para modelar *pipelines* complejos, esta es una razonable y muy rápida técnica para modelar el rendimiento de *pipelines* enteros cortos, ignorando el comportamiento del sistema de memoria.

5.1.1.2 Simuladores dirigidos mediante trazas *(trace-driven simulation)*

Esta es una técnica más sofisticada técnica par modelar el rendimiento y es particularmente útil para modelar el rendimiento de sistemas de memoria. En la simulación dirigida mediante trazas, una traza de una referencia de memoria ejecutada es creada, usualmente ya sea por simulación o por una ejecución instrumentada. La traza incluye que instrucciones fueron ejecutadas (dado por la dirección de la instrucción), también como las direcciones de datos accesados.

La simulación dirigida mediante trazas, puede ser usada en diferentes maneras. El uso más común es para modelar el rendimiento del sistema de memoria, el cual puede ser hecho por la simulación del sistema de memoria, incluyendo las *caches* y cualquier administración de memoria en *hardware*, usando la dirección trazada. La simulación del sistema de memoria puede ser combinada con un análisis estático del rendimiento del *pipeline* para obtener una precisión razonable del modelo para procesadores con un *pipeline* simple. Para *pipelines* más complejos, la traza de los datos puede ser usada para llevar a cabo un análisis más detallado del desempeño del *pipeline* por medio de la simulación del procesador. De aquí la traza de datos permite una simulación del orden exacto de las instrucciones con lo que se alcanza más alta precisión que con el enfoque estático.

Esta técnica de simulación, típicamente aisla la simulación de cualquier comportamiento del *pipeline* del sistema de memoria. En particular, esta técnica asume que la traza es completamente independiente del comportamiento del sistema de memoria.

Este tipo de simuladores, son usados por microarquitecturas durante la fase temprana de diseño con el objeto de explorar y comparar varios aspectos en un modo cuantitativo, siendo utilizados los resultados para confirmar o corregir las intuiciones del diseñador.

5.1.1.3 *Execution-driven simulators*

Esta tercer técnica es la más precisa y más costosa. En este tipo de simuladores, una detallada simulación del sistema de memoria y el *pipeline* del procesador son hechas simultáneamente, permitiendo el modelo exacto de la interacción entre los dos, la cual se hace necesaria en el diseño avanzado de procesadores.

Existe una variante de este tipo de simuladores, donde la técnica lleva el nombre de *full system execution-driven simulation*, donde adicionalmente son tomados en cuenta aspectos de *software* como lo es el sistema operativo, por lo que hace posible incluir los mecanismos de memoria virtual, tornando de éste modo la simulación de una manera más compleja y un tiempo de simulación superior.

5.1.1.4 Análisis comparativo de las técnicas de simulación

Cada una de las técnicas de simulación mencionadas, presentan una serie de beneficios y desventajas, lo que torna una dependencia con el propósito de utilización y de la etapa de diseño en la que se esté, presentándose las diferentes consideraciones en la tabla XV que sigue a continuación

Tabla XV. Atributos de varias técnicas de simulación

Técnica de Simulación	Entradas	Beneficios	Desventajas
Modelo Estático	CPI básico Fracasos en la caché	Flexible, simple, rápido, precisión razonable	Falta de Precisión, No modela simultáneamente
Dirigido mediante trazas (trace-driven simulation)	Trazas de *hardware* y trazas de *software.*	Detallado y preciso	Disputa en la recopilación de trazas, Falta de efectos especulativos, Implementación compleja
Execution-driven Simulation	Programas, parámetros	Detallado, preciso y rutas especulativas	Implementación compleja, Tiempo de simulación superior, Requerimiento de exactitud, Falta de los efectos del sistema operativo (OS).
*Full-system, execution-*driven simulation	Sistema Operativo, programas, parámetros, imágenes de disco.	Detallado, preciso y exacto	Implementación compleja, Tiempo de simulación superior, Requerimiento de exatitud.

5.2. Descripción de algunos simuladores

En esta sección se hará la descripción de varios simuladores comúnmente utilizados con propósitos investigativos y conceptuales, los cuales poseen una descarga gratuita en la *internet*. La mayoría de éstos simuladores nos permiten modificar varios parámetros de diseño y con ello poder observar las diferentes limitantes y ventajas que ofrece el ILP en el momento de ejecución, así como también poder tomar un mayor criterio de consideración en una etapa temprana del diseño y experimentar los beneficios de la simulación para el diseñador como tal.

Muchos de éstos simuladores utilizan el *set* de instrucciones DLX, donde los programas pueden ser desarrollados directamente o mediante la utilización de un compilador de C a DLX (por ejemplo dlxcc).

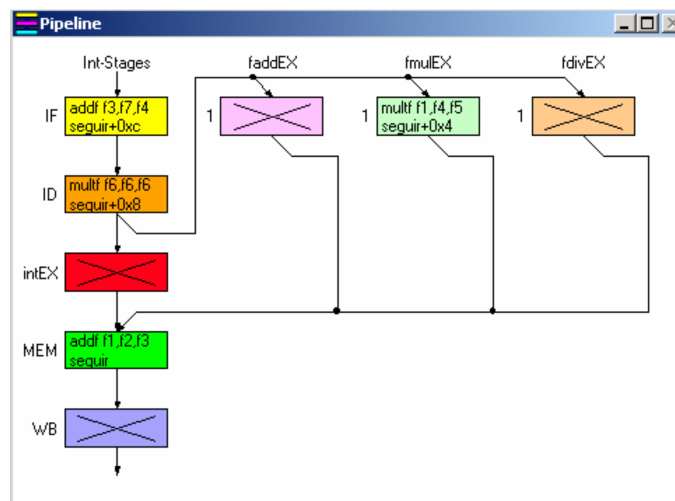
5.2.1. Windlx (Windows Deluxe Simulator)

Este es un simulador que utiliza el *set* de instrucciones DLX. Es una aplicación desarrollada para entornos Windows de Microsoft. Este programa fue elaborado por la Universidad de Tecnología de Viena y, en concreto, esta versión salió a la luz en mayo de 1992.

El Windlx, está compuesto por 6 subventanas (*child window*), las cuales nos permiten visualizar varios de los estados del procesador, tales como: *Pipeline* (segmentación de flujo), *Clock Cycle Diagram* (diagramas de ciclo de reloj), *Code* (código), *Breakpoints* (puntos de interrupción), *Register* (registro) y *Statistics* (estadísticas).

En primer lugar, se encuentra la ventana de *pipeline* la cual está compuesta de un *pipeline* básico de cinco etapas, tal como se muestra en la figura 35.

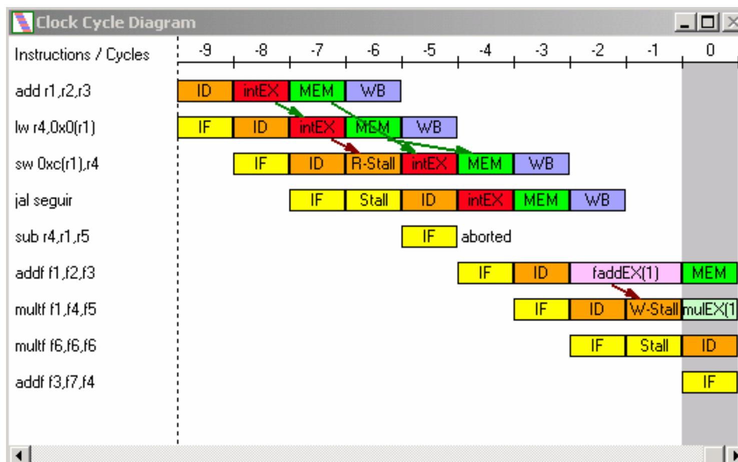
Figura 35. Ventana de *pipeline* en el simulador Windlx



Donde Se puede observar el detalle que el estado de ejecución esta ramificado en cuatro unidades: intEX (*INTeger EXecution* o ejecución con operadores Enteros); faddEX (*single and doble precision Folating point ADD EXecution* o Ejecución de Suma con operadores de Punto Flotante de simple o doble precisión); fmulEX (*single and doble precision Folating point MULtiply EXecution* o Ejecución de Multiplicación con operadores de Punto Flotante de simple o doble precisión) y fdivEX (*single and doble precision Folating point DIVide EXecution* o Ejecución de División con operadores de Punto Flotante de simple o doble precisión). Esto se debe a que el tradicional módulo de ejecución "EX" ha sido substituido por 4 etapas paralelas de ámbito más especifico.

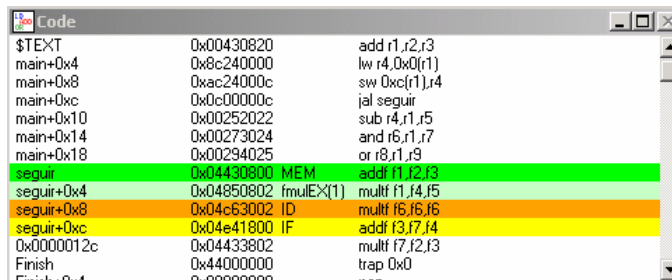
En la ventana llamada diagrama de ciclos de reloj permite ver, de forma sencilla, la evolución cronológicamente de la ejecución del código. Esta compuesta por una primera columna donde se mostrarán las instrucciones a las cuales pertenece esa ejecución. Y a su derecha cómo las diferentes ejecuciones han trascurrido por el *Pipeline*, encabezadas por el número de ciclo al cual pertenecen los estados. La zona más oscurecida representa la ejecución actual del procesador, tal como se muestra en la figura 36.

Figura 36. Ventana del diagrama de ciclos de reloj del simulador Windlx



En la ventana de código, está representa la memoria mediante tres columnas: la dirección (simbólica o numérica), la representación máquina (en hexadecimal) de la instrucción y la instrucción en ensamblador. A medida que se ejecuta la el código se va coloreando, línea a línea, el fondo de las distintas instrucciones de la memoria con el color de los distintos pasos del procesador representado en la ventana *pipeline*, tal como se muestra en la figura 37.

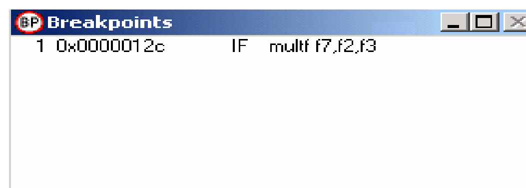
Figura 37. Ventana de código del simulador Windlx



Desde esta ventana se pueden insertar y eliminar puntos de interrupción (que quedarán reflejados en la ventana “Breakpoints”). En el código aparecerán junto a la dirección una etiqueta indicando que en esa instrucción hay una interrupción.

La ventana de *breakpoints* simplemente nos informa de los distintos puntos de interrupción insertados en el código, tal como es mostrado en la figura 38.

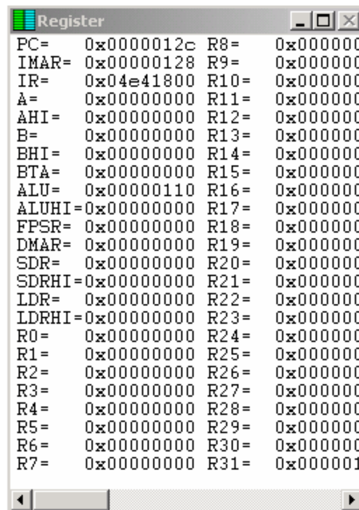
Figura 38. Ventana de *breakpoints* en el simulador Windlx.



Esta ventana estará formada por cuatro columnas: número de punto de interrupción, dirección, ciclo donde se parará la ejecución e instrucción.

En la ventana de registros, tal y como indica su nombre, se pueden observar los valores que contienen los registros del computador durante la ejecución. Mostrándose el nombre del registro y a continuación el valor que tiene, tal como se muestra en la figura 39.

Figura 39. Ventana de registros del simulador Windlx

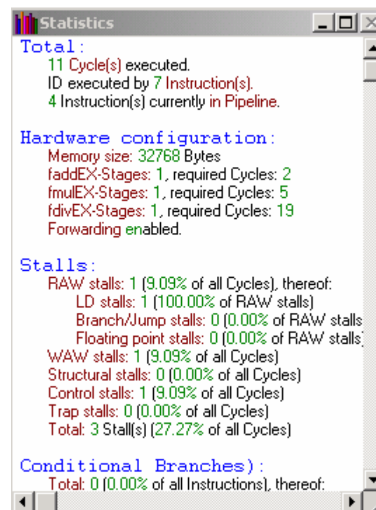


Esta ventana *Register* lista el nombre y contenido de todos los registros de la arquitectura DLX. En concreto estos los registros accesibles para los programas: GPRs (*General Propouse Register* o registro de propósito general), treinta y dos registros (R0..R31) de 32 bits (para almacenar datos enteros) dónde R0 siempre tiene valor “0”; FPRs (*Floating Point Register* o registro de punto flotante), conjunto de registros que se puede utilizar como treinta y dos registros (F0..F31) de 32 bits, es decir de precisión simple, o bien dieciséis registros (D0..D30 contando de 2 en 2) de precisión doble, 64 bits, agrupando los registros de dos en dos con el objetivo de obtener mayor precisión; FPRS (*Floating Point Status Register* o registro de estado de punto flotante), utilizado para comparar y para excepciones de punto flotante. Todos los movimientos desde o hacia el FPRS van a través del GPRs, existe un salto que prueba el bit de comparación en el FPRS.

Y por último, está el PC (*Program Counter* o contador de programa), que contiene la dirección de la próxima instrucción a ser buscada; este registro puede ser modificado por las instrucciones de salto (*branches/jumps*).

Cabe mencionar la ventana de estadísticas que nos muestra información de diferentes aspectos de una simulación como son: la configuración *harware* que se ha utilizado en la simulación (decidida por el usuario), las detenciones que se han realizado y sus causas (estructurales, de control, por interrupción, *RAW*, *WAW*), los saltos que han sido efectivos, el número de instrucciones de carga y almacenamiento en memoria, instrucciones con punto flotante y los mensajes mostrados por pantalla. Además de los números de ciclos simulados, instrucciones que pasaron por el estado ID y las instrucciones que se encuentran en el *pipeline* en ese momento, tal como es mostrado en la figura 40.

Figura 40. Ventana de estadísticas del Windlx



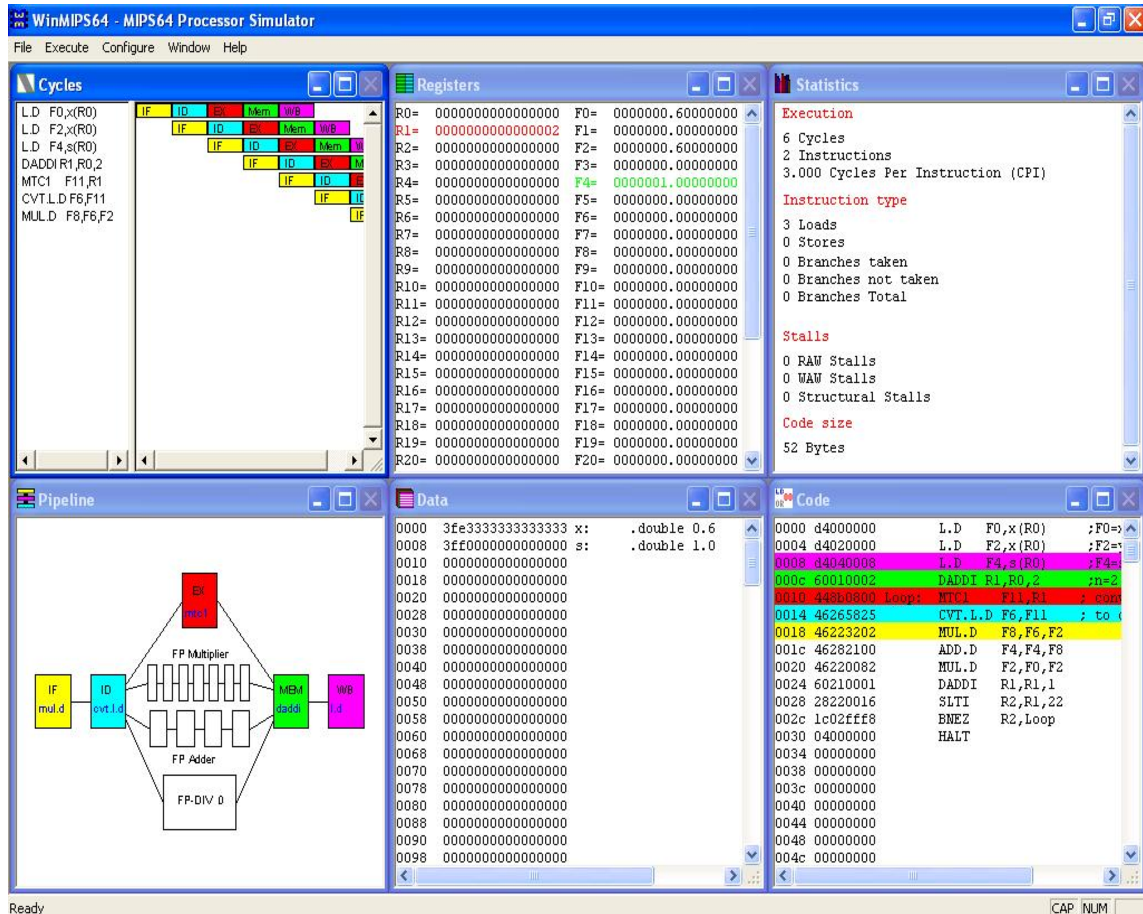
Toda la información que encontraremos en esta ventana de estadísticas resulta muy útil a la ahora de realizar comparaciones entre distintas simulaciones cambiando distintos parámetros de ejecución. Puede ser muy interesante comparar cómo afectan los cambios en la configuración del DLX: habilitar o deshabilitar *Forwarding*, número de unidades de punto flotante, latencias, etcétera.

Este simulador, nos permite observar varias de las consideraciones desarrolladas en el capítulo 2, permitiendo de esta manera una ampliación de criterio para un diseño futuro.

5.2.2. Winmips64

Winmips 64 es un simulador diseñado como un reemplazo al popular Windlx, migrando de una arquitectura DLX de 32 bits a una arquitectura MIPS de 64 bits, desarrollada en el año 2003 por la Universidad de Dublín, Irlanda. El Winmips 64 muestra una interfaz gráfica similar a la del Windlx. Tal como es mostrado en la figura 41.

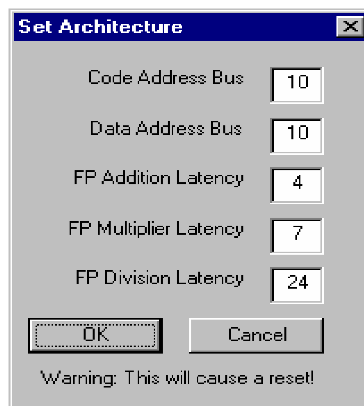
Figura 41. Interfaz gráfica del simulador Winmips64



Si se conoce como utilizar el Windlx, entonces el WinMips64, nos parecerá muy familiar, ya que como podemos observar, las subventanas que utiliza son muy similares, tales como: *Pipeline* (segmentación de 5 estados y sus unidades de FP), *Cycle* (diagramas de ciclo de reloj), *Code* (código en memoria), *Data* (datos en memoria de 64 bits), *Register* (contenido de registros) y *Statistics* (estadísticas de ejecución, de tipo de instrucción, y de paradas).

Winmips64 puede ser configurado de varias maneras. Puede ser cambiada la estructura y el tiempo requerido de las unidades de punto flotante y el tamaño de memoria. Tal como se muestra en la siguiente figura:

Figura 42. Ventana de configuración de arquitectura del simulador



Cualquier cambio en las latencias de las unidades de punto flotante serán reflejadas en la ventana de *pipeline.* El *Code Address Bus* y el *Data Address Bus* se refiere al número actual de líneas en el bus de direcciones, así que para un valor de 10 significa 2^{10}=1024 bytes serán desplegados en las ventanas respectivas.

Además de estas opciones de configuración nos permite la habilitación de la técnica de *forwarding* y en adición con respecto a Windlx, permite la habilitación de la técnica de *branch delay slot.*

5.2.3. Dlxview

DLXview es un simulador interactivo de *pipeline* que utiliza el set de instrucciones DLX. La meta principal de Dlxview es proporcionar un ambiente visual que sirva como una herramienta útil para la comprensión, depuración, y evaluación de actuación de procesador, existiendo una versión ejecutable en plataforma Linux (presentada en agosto 1997) y una versión adaptada para plataforma Windows (bajo prueba y con algunas limitantes presentada en el 2003).

Este simulador del procesador DLX, tiene algunas diferencias con el Windlx anteriormente mencionado:

- Tiene tres modos posibles de encadenar instrucciones, que pueden elegirse en la configuración:

  1. Basic Pipeline (cadena básica, sin planificación dinámica pero con todos los *bypasses*). Es similar a la del WINDLX, mostrando el detalle electrónico del flujo de datos de la unidad de datos entera, aunque no informa en el cronograma de la causa de un bloqueo.
  2. Algoritmo de Tomasulo
  3. Marcador centralizado o ScoreBoard.

- En el *pipeline* básico, al ser un simulador orientado a instrucciones FP, DLXview permite que existan varias instrucciones en las fases MEM y WB, es decir supone que la unidad de control soporta dos fases MEM y que hay un puerto de escritura para el fichero de registros entero y otro para el de flotantes.

- Aunque algunas directivas varíen, las que se necesitan para programas simples son idénticas al WINDLX, y los programas de uno sirven para el otro.

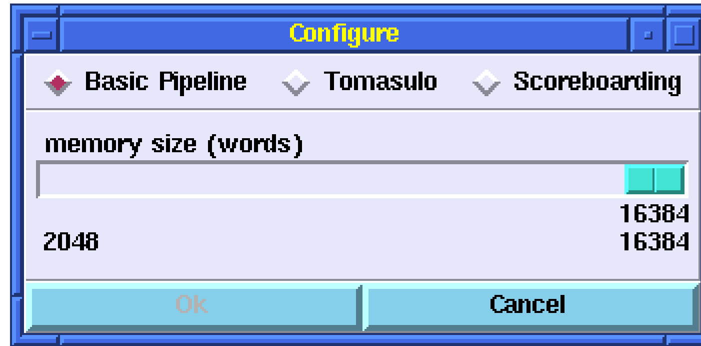
- Todas las instrucciones *mult y div* requieren que sus operandos sean registros FP como está definido en el DLX original.

- DLXview utiliza siempre saltos condicionales del tipo *delayed branch* con un hueco de una sola instrucción (aunque el mnemónico para las instrucciones de ramificación sea *beqz, bnez*, la *'d'* de *"delayed"* delante, o sea ser *dbnez y dbeqz*). Esto tiene la consecuencia inmediata de que la instrucción tras el salto del bucle siempre se ejecuta. Además, el simulador trabaja con un "supuesto sistema de predicción", que siempre acierta (el simuladores capaz de "predecir" con 100% de acierto los saltos, ya que conoce *a priori* los resultados de las comparaciones, o sea que presenta ésta situación de manera ideal).

- Las unidades funcionales (U.F.) del DLXview son configurables, en cuanto a segmentación, cantidad y latencia.

Al utilizar este simulador, es necesario especificar, en que modo se desea operar y que cantidad de memoria se desea utilizar mediante la ventana de configuración, mostrada en la figura 43.

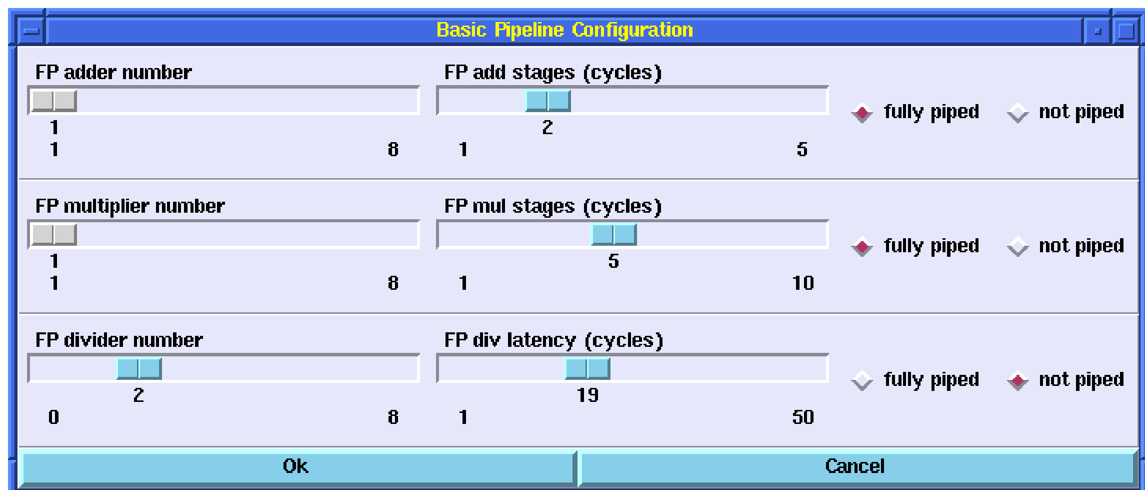
Figura 43. Ventana de configuración de memoria y modo de operación en el simulador DlxView 0.9



5.2.3.1 Simulación del *pipeline* básico.

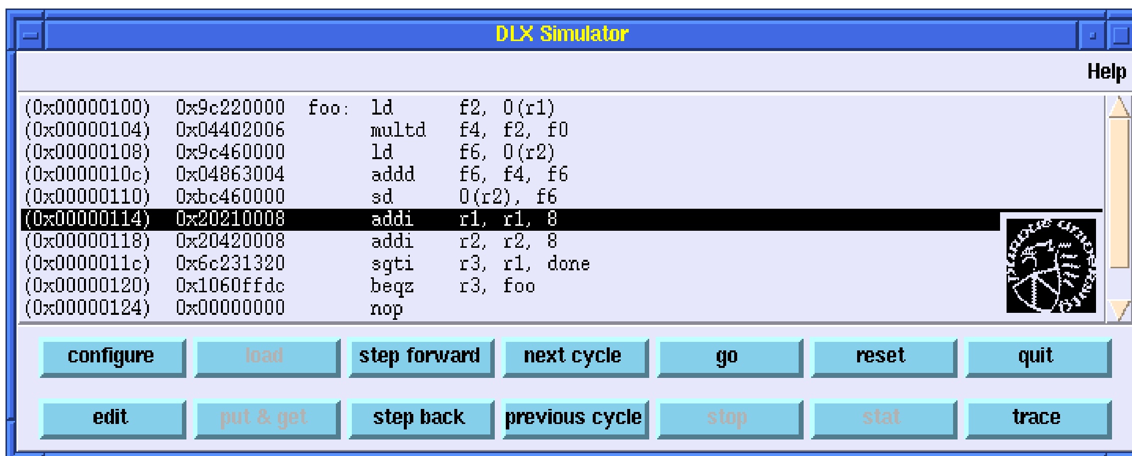
Al elegir el modo de *pipeline* básico, este simulador permite variar los valores de varios parámetros, tal como se muestra en la figura 44.

Figura 44. Ventana de configuración del pipeline básico en el simulador Dlxview



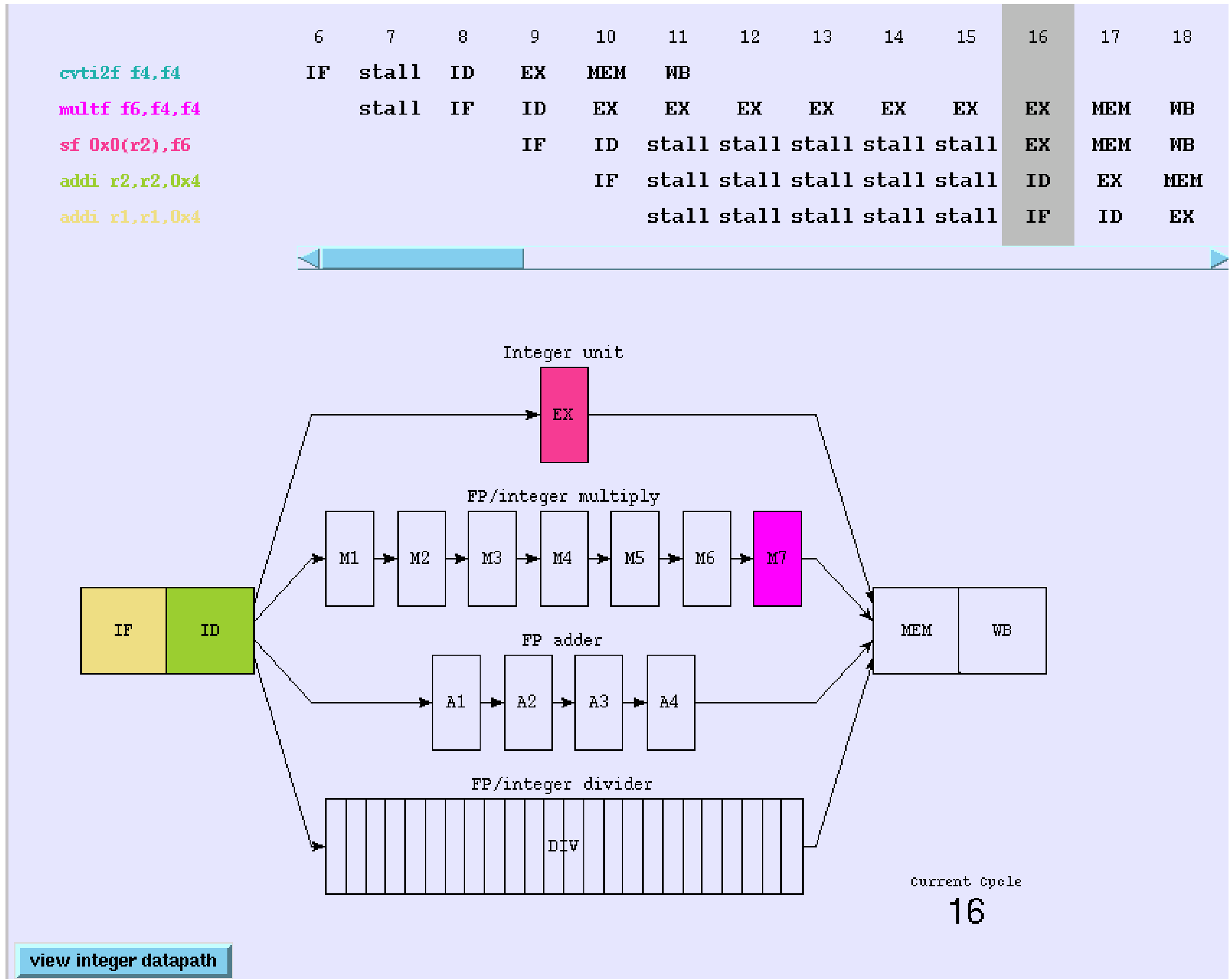
Luego de configurar, número de unidades funcionales, latencia y segmentación del las unidades, después de haber sido cargado el programa, es posible controlar el flujo de instrucciones mediante la ventana de control del simulador.

Figura 45. Ventana de control del simulador dlxview.



En este *pipeline* básico, las instrucciones son tomadas y decodificadas secuencialmente, las ramificaciónes son resueltas en el estado de ID por medio de la técnica de branch *delay,* al igual que los otros simuladores mencionados, permite mostrar el flujo de instrucciones, tal como se muestra en la figura 46.

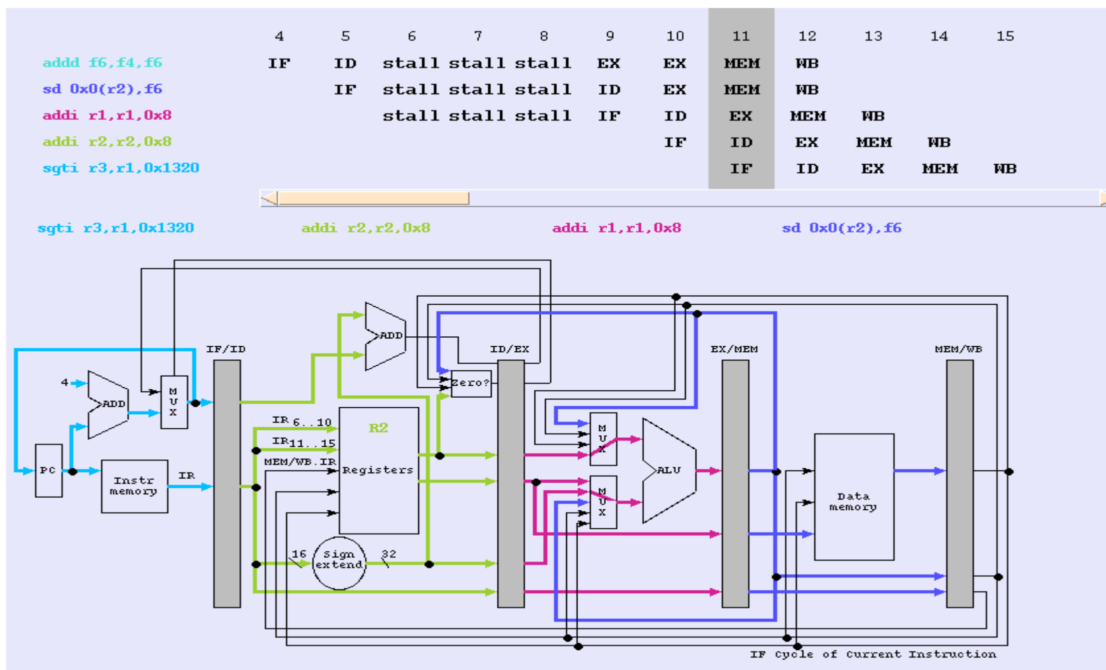
Figura 46. Ventana del *pipeline* básico en el simulador Dlxview



Al empezar la simulación con los botones de la ventana de control (botón *next cycle*, *step forward* o *go*), al ejecutar el primer paso, irán saliendo instrucciones y su cronograma de forma similar a los del WinDLX. Los colores indicarán en que fase está cada instrucción.

Al presionar el botón *view integer data path* podrá verse el flujo de datos de la unidad de datos entera, con un detalle perfecto de las fases de ejecución, los caminos de desvío (*bypass*), los multiplexores de entrada a la ALU y otras unidades, etc. Allí se muestran con distintos colores, como se encuentran todas las fases ejecutando hasta cinco instrucciones a la vez, tal como es mostrado en la figura 47.

Figura 47. Ventana del flujos de datos en la unidad entera del *pipeline* del simulador Dlxview



5.2.3.2 Simulación de planificación dinámica de instrucciones mediante el algoritmo de Tomasulo.

La simulación desarrolla el algoritmo de Tomasulo básico, tomando en cuenta algunos aspectos importantes:

- La búsqueda de la instrucción (IF), su decodificación (ID) y su emisión (IS) se hacen todas en una sola fase, la IS. Las instrucciones NOP ocupan una estación de reserva (R.S.) entera y pueden contribuir a bloquear la máquina.

- Aquí el bus de escritura CDB es único.

- Se le dota a las unidades funcionales de varias estaciones de reserva para emitir instrucciones enteras, para las de F.P. MULT, para las F.P. ADD, y además de R.S. (o búfferes de *load y store*) para los accesos a memoria. El número de R.S. es configurable.

- Las instrucciones NOP ocupan una R.S. (pueden ayudar a bloquear la máquina si se agotan las R. S. enteras).

- El nombre de las etiquetas de los registros coincide con el de la R.S. que ocupa. Por ejemplo en la figura, el buffer de Store está esperando por el registro F6, que en realidad es la etiqueta Add1, o sea el registro de destino de la primera R.S. de la U.F. de ADD.

La configuración, se logra mediante la ventana mostrada en la figura 48 y el corrimiento puede ser observado en la ventana de la figura 49.

Figura 48. Ventana de configuración del algoritmo de Tomasulo en el simulador Dlxview

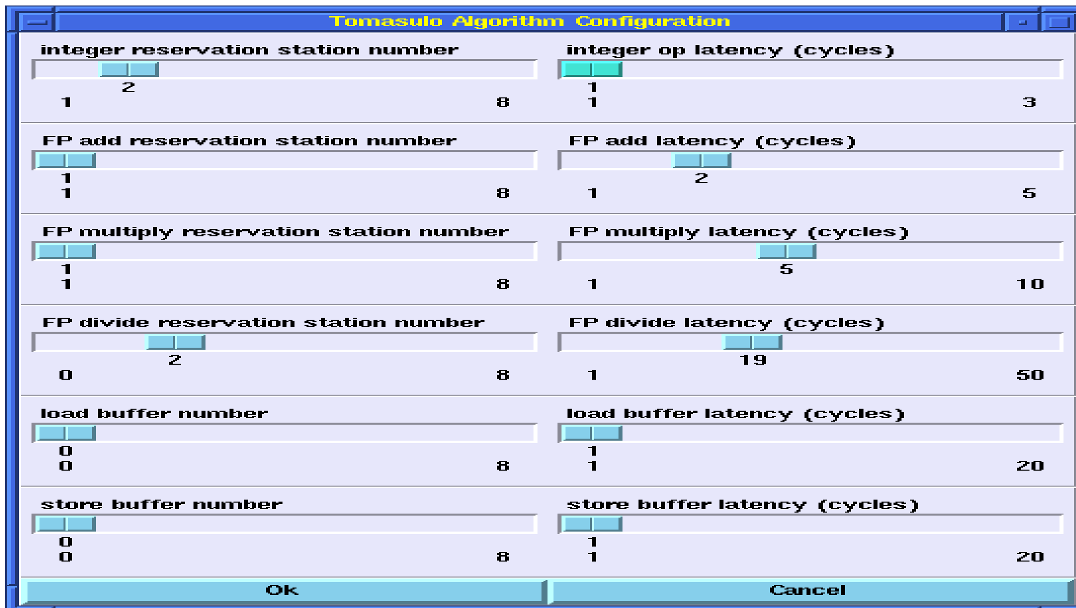
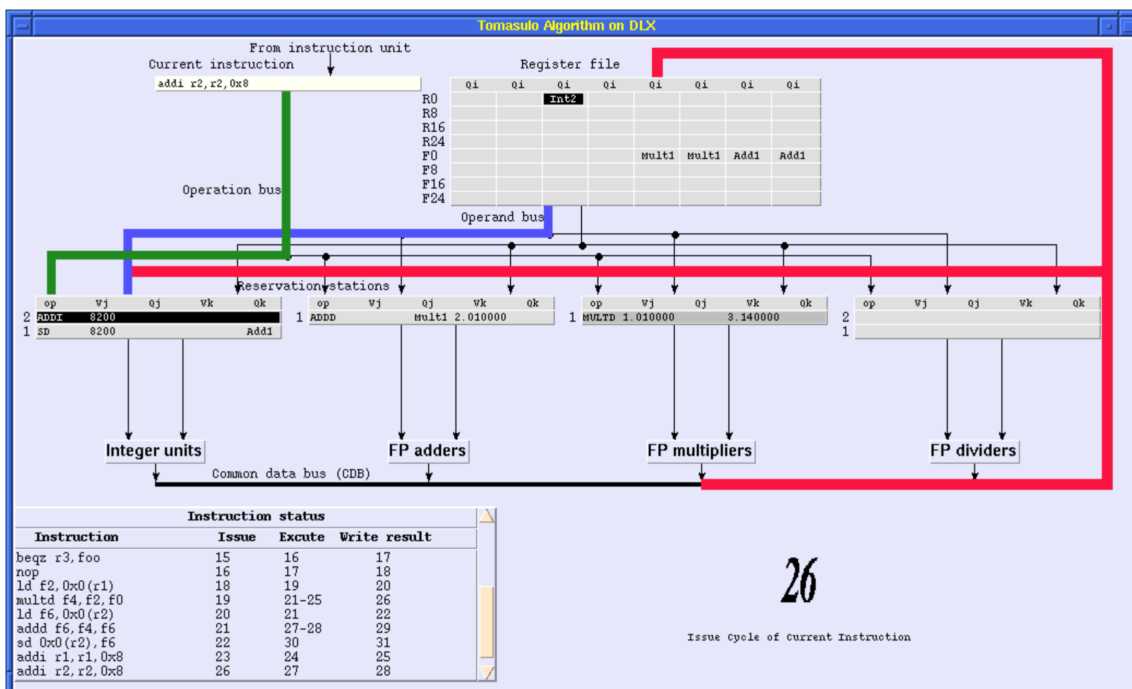


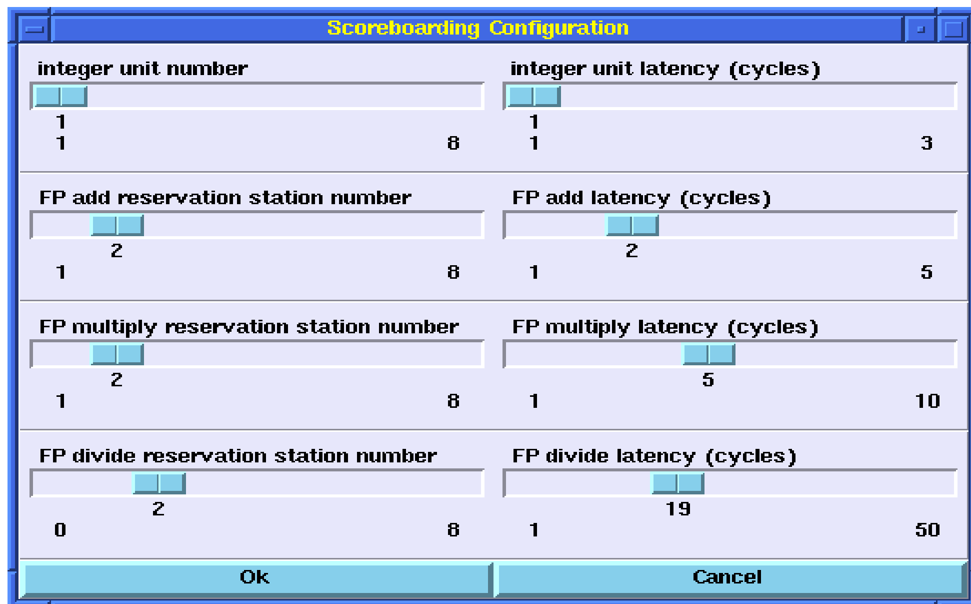
Figura 49. Ventana de flujo del algoritmo de Tomasulo, en el simulador Dlxview



5.2.3.3 Simulación de *scoreboarding*

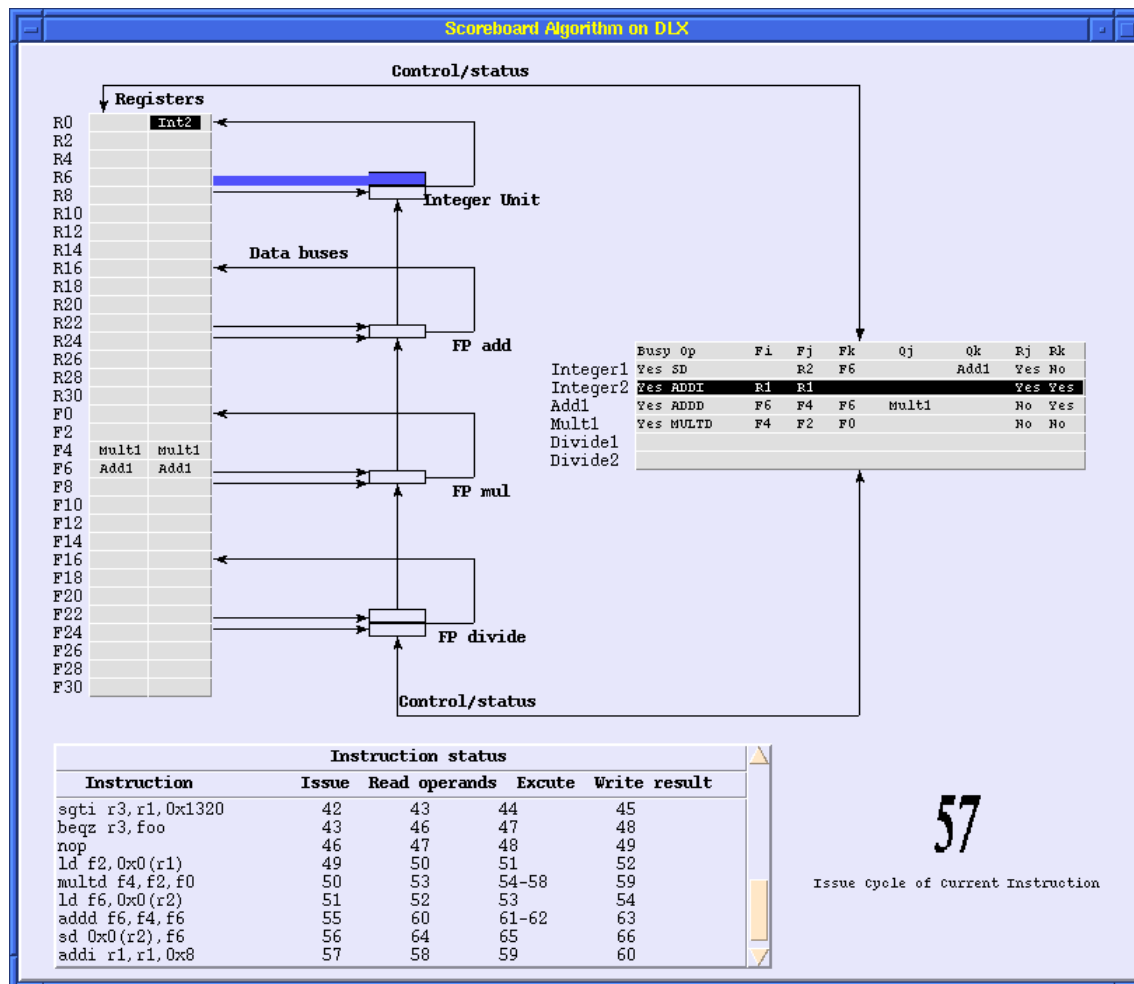
Al igual que los otros modos de simulación, el *scoreboarding* puede ser configurado en diferentes parámetros: número de unidades enteras (*integer units*), número de estaciones de reserva de punto flotante tanto para suma (FP *add*), multiplicación (FP *multiply*) y división (FP *divide*), con sus latencias respectivas, tal como se muestra en la figura 50.

Figura 50. Ventanta de configuración del *scoreboarding* de Dlxview.



Luego de haber configurado nuestro *scoreboard*, es posible visualizar como se lleva a cabo la ejecución de nuestro programa, mediante la ventana mostrada en la figura 51, en la cual se a configurado un *scoreboard* con dos unidades de enteros, una unidad de punto flotante de suma, una de multiplicación, y dos de división.

Figura 51. Ventana de ejecución del algoritmo del *scoreboarding* de Dlxview

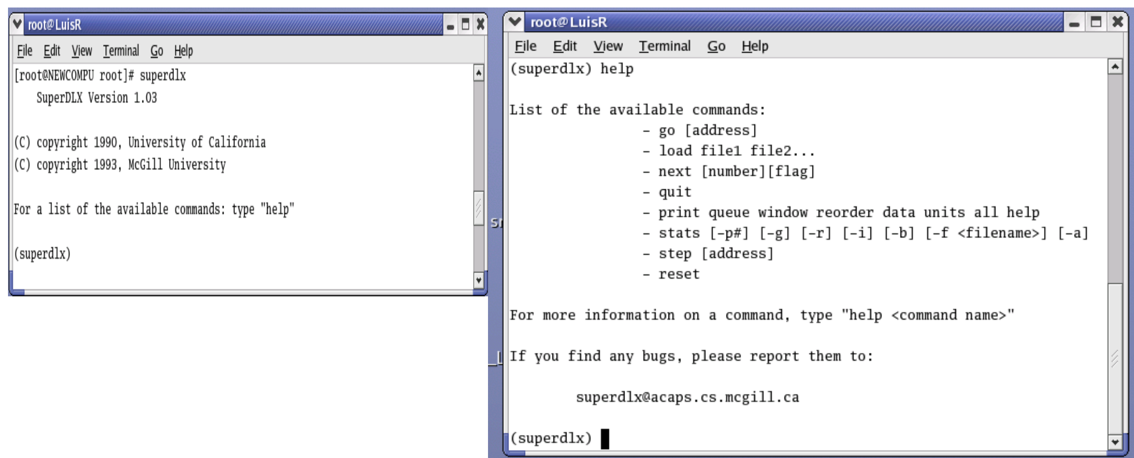


En esta ventana es posible observar la diferente información que se mueve a través del *scoreboard*, tal es el caso del estado de las instrucciones, indicando en qué fase o etapa se encuentran las instrucciones; estado de las unidades funcionales, indicando la situación en la que se encuentra la unidad funcional en diferentes aspectos; y el estado de los registros, indicando la unidad funcional que va a generar el siguiente valor del registro.

5.2.3.4 Superdlx (superescalar)

Este es un simulador que no presenta entorno gráfico, todo se lleva a cabo mediante una línea de comandos, teniendo la opción para poderse trabajar en una plataforma Linux (tal como es mostrado en la figura 52). Sin embargo su potencial lo hace calificar como un simulador del tipo *execution-driven* a diferencia de los simuladores anteriormente mencionados, catalogados del tipo *trace-driven simulators*.

Figura 52. Ambiente en el que se desarrolla Superdlx



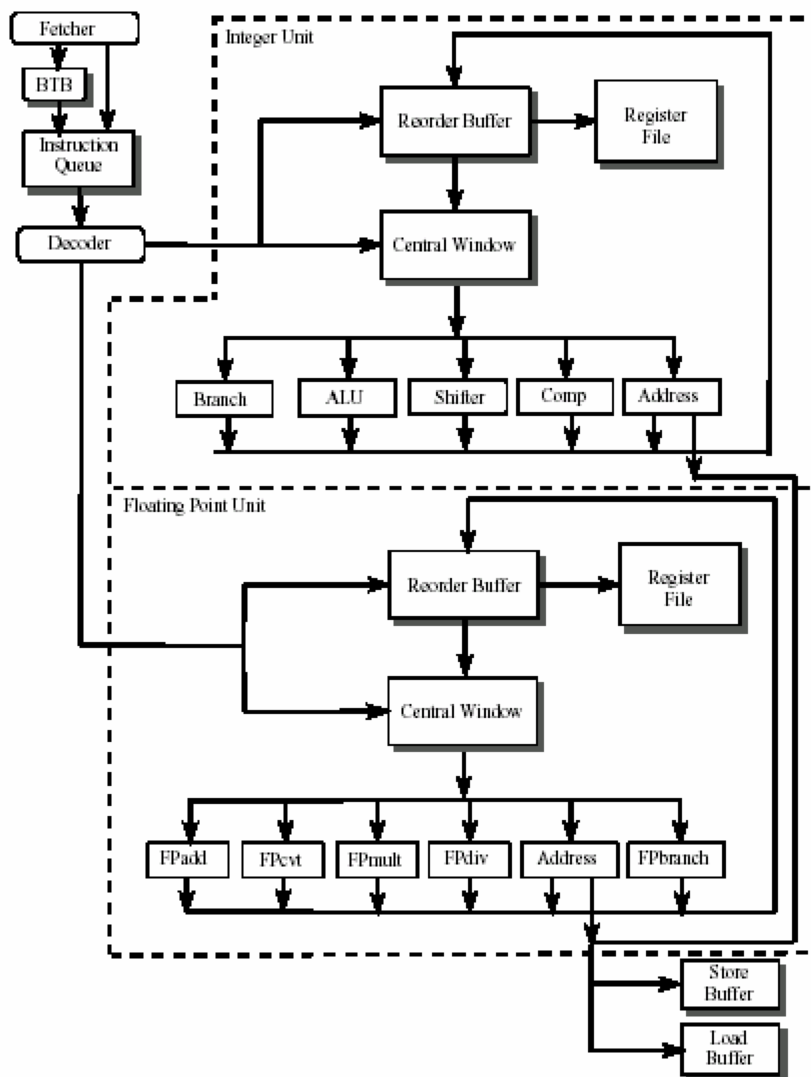
Este simulador de una arquitectura superescalar es capaz de simular el proceso de una emisión y completación fuera de orden de las instrucciones, para implementar tal proceso de instrucciones, varios mecanismos de *hardware* son seleccionados:

- Ventana de instrucciones: desde donde las instrucciones fuera de orden son emitidas a múltiples unidades funcionales.
- Renombre de registros.
- Búfer de reordenamiento (ROB): donde los conflictos de almacenamiento son resueltos

- Predicción dinámica de ramificaciones, haciendo uso de 2 bits de predicción.
- Acceso a memoria optimizado, con el uso de búfers de carga (*load*) y almacenamiento (*store)*.

Llevando todo ello, a la representación del modelo mostrada en la figura 53.

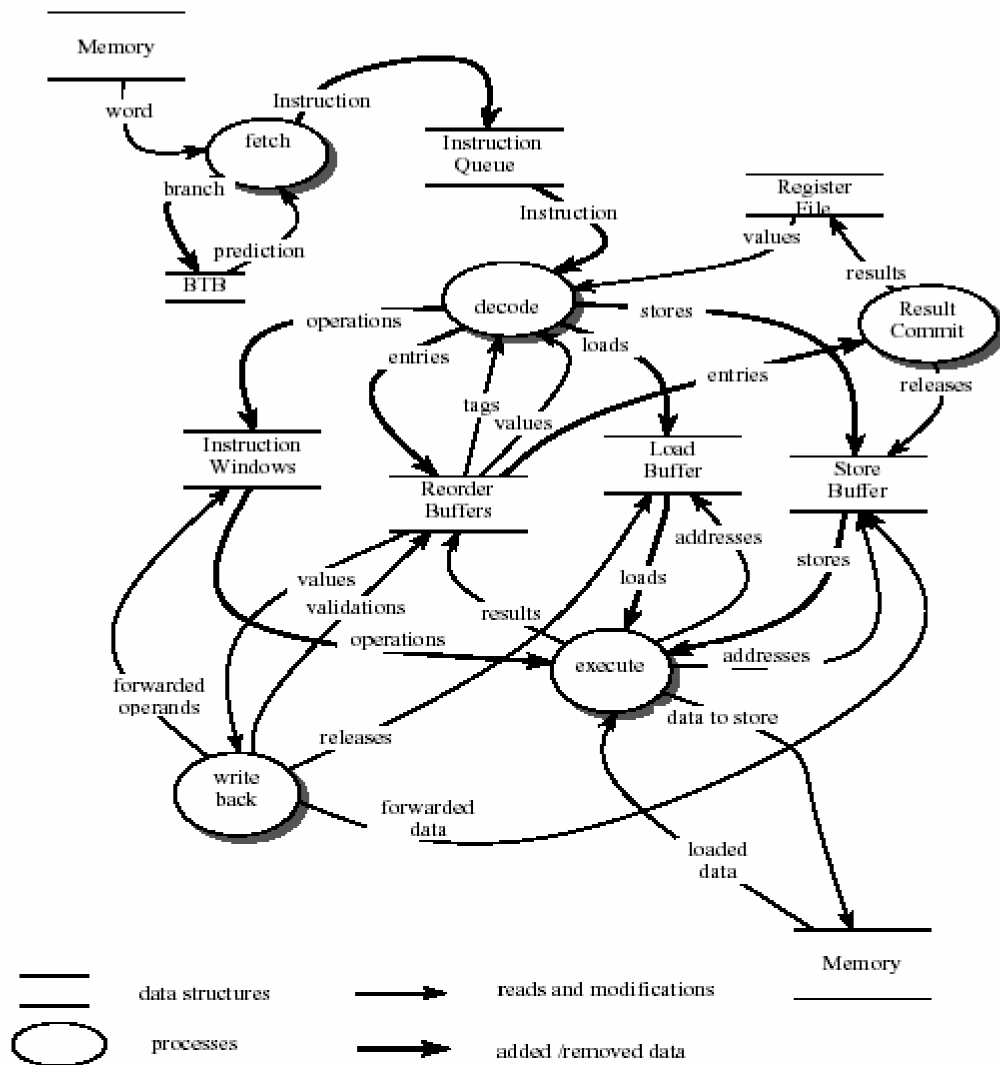
Figura 53. Modelo de procesador superescalar utilizado por Superdlx



Fuente: Moura, Cecile. Superdlx:a generic superscalar simulator. Pag. 8

De esta manera, el funciona del modelo representado en la figura 53, puede ser descrito por el diagrama global representado en la figura 54.

Figura 54. Funcionamiento global del manejo de instrucciones de Superdlx.



Fuente: Moura, Cecile. Superdlx:a generic superscalar simulator. Pag. 25

Este simulador nos permite configurar una serie de parámetros, propios de un procesador superescalar, siendo estos ingresados por medio de la edición o sustitución de un archivo llamado *machinefile*; los siguientes parámetros se pueden modificar:

- Instrucciones procesadas por ciclo, en la etapa fetch, en la etapa de decodificación y en la etapa de *commit*.
- Memoria, en tamaño, latencia y accesos.
- Tamaño del búfer de reordenamiento, para instrucciones de enteros y de punto flotante.
- Tamaño de ventana de instrucciones, para instrucciones de enteros y de punto flotante.
- Tamaño de búfer de datos para carga (*load*) y almacenamiento (*store*).
- Tamaño de la cola de instrucciones.
- Número y latencia de unidades funcionales enteras.
- Número y latencia de unidades de punto flotante.

Luego de cargar nuestro programa (con el comando load) y simularlo con cualquiera de los comandos determinados (por ejemplo: *step*, *next*, o *go*), el simulador es capaz de presentarnos por medio del comando *print* el estado actual de cualquier unidad del procesador modelado, para luego con el comando *stats* poder mostrar las estadísticas realizadas durante el programa.

Las estadísticas mostradas por este simulador nos ofrecen datos importantes del desarrollo obtenido mediante el corrimiento del programa en nuestro diseño de *hardware*, tanto en cantidad, como porcentajes según sea el caso; éstos son los siguientes:

- Número de ciclos.
- Número Instrucciones traídas de memoria (*fetched*)
- Número de Instrucciones decodificas y porcentaje respecto a instrucciones *fetched*.
- Número de Instrucciones emitidas (*issued*) con porcentaje respecto al total de *fetched*, especificando el número de instrucciones de enteros y de punto flotante, con porcentaje respecto al total de instrucciones emitidas.
- Número de instrucciones que llevaron el proceso de *commit* con porcentaje respecto al total de *fetched*, especificando el número de instrucciones de enteros, de punto flotante, de escritura a registros y de instrucciones sin escritura, con porcentaje respecto al total de instrucciones que llevaron el proceso de *commit.*
- Número de cargas bloqueadas por almacenamientos con porcentaje respecto al total de cargas.
- Instrucciones por ciclo (*rates*) de los diferentes estados: *fetch, decode, issue* y *commit.*
- Número de ramificaciones, con el porcentaje de las tomadas y no tomadas, el porcentaje de acierto para cada uno de los casos y el porcentaje de ciclos perdidos.
- Número y porcentaje de paradas de *fetch* y paradas de *fetch* debidas a búfers llenos.

- Porcentaje de operandos renombrados, especificando enteros y de punto flotante, también el porcentaje de ciclos de reloj de éstos.
- Número de operandos búscados con el número y porcentaje de ciclos de reloj.
- Número y porcentaje de distribución de instrucciones emitidas (*issue*), distribución de retardo de emisión y distribución en *commit*, entre instrucciones de enteros y punto flotante.
- Cantidad y porcentaje de ocupación de los búfers de enteros y punto flotante respecto a la ventana de instrucciones y el búfer de reordenamiento.

Si se tienen presentes las estadísticas mencionadas con anterioridad, este simulador permite observar, respecto a la aplicación que se ponga en marcha, la incidencia existente en cada una de las partes de la arquitectura diseñada conforme a los parámetros prediseñados en búsqueda de una mayor eficiencia, formulando de esta manera la simulación diferentes consideraciones de diseño del paralelismo en hardware a nivel de instrucción en microprocesadores.

CONCLUSIONES

1. El paralelismo es una herramienta utilizada ante los diferentes obstáculos que se han presentado en el diseño de microprocesadores, soluciona el hecho de que si una tarea en base a sus limitantes no se puede realizar con mayor rapidez, se considera realizar más tareas en el mismo tiempo de ejecución tal como sea posible.

2. Los límites de un ILP ideal radican en la existencia de dependencias entre instrucciones (datos y control), dependencias estructurales y limitantes tecnológicas; éstas últimas llevan a un círculo vicioso debido a la interdependencia con las demás.

3. El CPI de un procesador dependerá de la organización y del *set* de instrucciones utilizado, ya que un *set* de instrucciones menos complejo, permitirá una más fácil planificación del código y conducirá a una ejecución más eficiente.

4. La organización del microprocesador y el uso efectivo de cada segmento de instrucción designado en su ILP, marcarán la diferencia en la eficiencia del procesamiento de instrucciones, tomando a consideración entonces que procesadores con bajos CPIs no siempre serán más rápidos, al igual que procesadores con altas velocidades de reloj no siempre lo serán.

5. La administración adecuada en el acceso y lectura a banco de registros dará como resultado una mejor implementación de la técnica o algoritmo a utilizar maximizando de esta manera la asignación y liberación de éstos.

6. La planificación dinámica no es más que la versión de *hardware* del ILP, la cual está basada en la ejecución paralela de instrucciones sin importar el orden original del programa, por ello utiliza técnicas como el renombramiento de registros, especulación y predicción.

7. El renombramiento de registros es una técnica que permite la eliminación de dependencias del tipo WAR y WAW disminuyendo de ésta manera los obstáculos en el ILP, pero con limitantes respecto al número de registros físicos y la capacidad de acceso a ellos.

8. La especulación es una técnica basada en el principio de “ejecución fuera de orden”, su principio fundamental, es no detener la ejecución debido a una dependencia; sino seguir ejecutando todas las instrucciones posibles. Su limitante es el tamaño de ventana, la adición de nuevo *hardware* de control que soporte cualquier especulación mal habida, la actualización de registros y memoria mediante el proceso de *commit* utilizando un búfer de reordenamiento, y soporte a excepciones.

9. La técnica de predicción es la base de la especulación de ramificaciones, donde se han desarrollado diferentes técnicas utilizando predictores de varios niveles; de esta manera se puede ejecutar el contenido de la ramificación antes de que se sepa si ésta es tomada o no. Las limitantes de esta técnica radican en la búsqueda de un predictor perfecto debido a los atrasos causados en el proceso de reestablecimiento cuando una especulación es realizada incorrectamente.

10. La simulación permite describir y analizar el comportamiento de un sistema real y responder ciertas interrogantes para apoyar el diseño de sistemas reales, mediante el proceso de modelación. Permite conocer oportunamente hechos relevantes disminuyendo inversiones y gastos de operaciones reduciendo en algunas ocasiones el tiempo de desarrollo del sistema, caracterizando al simulador su flexibilidad y precisión.

11. Las estadísticas obtenidas mediante la simulación, permiten observar respecto a la aplicación que se ponga en marcha, la incidencia existente en cada una de las partes de la arquitectura diseñada conforme a los parámetros prediseñados en búsqueda de una mayor eficiencia.

RECOMENDACIONES

1. Al diseñar un microprocesador independientemente del diseño del ILP interno de éste, se debe considerar sus capacidades externas de comunicación (buses, memoria, etc.), para que sea explotado de la mejor manera el paralelismo interno.

2. En el arduo diseño del ILP es importante considerar la existencia de un elemento de *hardware* que se encargue de mantener el estado preciso de la máquina, en la presencia de cualquier excepción.

3. Al utilizar múltiples estados en el pipeline se incrementan latencias y se imponen altas penalizaciones debido a fallos de predicción o de especulación; sin embargo, para obtener el efecto esencial del pipeline (varias instrucciones por ciclo), es necesario incrementar la velocidad de reloj y someter nuestro diseño a un análisis exhaustivo de costo-rendimiento.

4. En la elección de un *set* de instrucciones es necesario que éste sea lo más simple posible, para fácil implementación de diferentes técnicas de paralelismo; si por razones de compatibilidad utilizamos un set de instrucciones complejo, nos veremos en la necesidad de traducir las instrucciones a unas más simples y planificar partiendo de ese punto.

5. En el proceso de diseño, si se desea utilizar la modelación por medio de la simulación en cada una de sus etapas, es necesario considerar factores como: costo y tiempo inicial, el no pasar por alto otras soluciones, poner demasiada confianza en los resultados de la simulación, así como todo factor humano y tecnológico.

6. Enfatizar en los cursos del área digital de ingeniería electrónica la utilización de simuladores y temas correspondientes al ILP para formar criterios de diseño correspondientes a una planificación dinámica.

BIBLIOGRAFÍA

1. Abonesi, David H. y otros. **“Dynamically tuning processor resources with adaptive procesing”**. IEEE Computer (U.S.A.) 36(12):49-57.2003.

2. **AMD.** http://www.amd.com, 2003.

3. Austin, Tood y otros. **“Making typical silicon matter with razor”**. IEEE Computer (U.S.A.) 37(3):57-65.2004.

4. Black, Bryan y otros. **Can trace-driven simulators accurately predict superscalar performance?**. Estados Unidos de Norte América*: Department of Electrical and Computer Engineering* Carnegie Mellon University,1996.

5. Burger, Doug y Goodman, James R. **“Billion-Transistor Architectures: there and back again”**. IEEE Computer (U.S.A.) 37(3):22-27.2004.

6. Cain, Harold W. y otros. **Precise and Accurate processor simulator**. Estados Unidos de Norte América: *Computer Science and Electrical engineering University of Wisconsin*, 2003.

7. Carnigie Mellon University: Computer Science Department. http://www.csd.cs.cmu.edu/research/publications.htm, 2004

8. **Colorado University**. http://ece-www.colorado.edu/~dconnors/CAR/links.html, 2003

9. Colwell, Bob. **“We may need a new box”**. IEEE Computer (U.S.A.) 37(3):40-41.2004.

10. Dasgupta, Subrata. **“A Hierarchical Taxonomic System for Computer Architectures”**. IEEE Computer (U.S.A.) 23(3):64-74.1990.

11. **Elsevier Science:** Morgan Kaufmann Publishers. http://www.mkp.com/CA3/, 2003.

12. González, Antonio**. "Tendencia en la microarquitectura de los procesadores"**. Novatica (España). (3):88-91.2000.

13. Hennesy, John L. y David A. Patterson.**Computer Architecture: A Quantitative Approach**. 3a ed. U.S.A.: Elsevier Science. 2003. 1108 pp.

14. **IBM Research.** http://www.research.ibm.com/MET, 2003.

15. **IEEE Computer Society**. http://www.computer.org, 2004

16. **Intel Corporation**. http://www.intel.com/espanol/labs/index.htm, 2004

17. Kim, Nam Sung y otros. **"Leakage current: Moore's law meets static power"**. IEEE Computer (U.S.A.) 36(12):68-74.2003.

18. Lu, Shin-Lien. **"Speeding Up processing with approximation circuits"**. IEEE Computer (U.S.A.) 37(3):67-73.2004.

19. **Massachussets Institute Technology:** Laboratory for computer Science. http://cgs.lcs.mit.edu/pubs/, 2003.

20. Moura, Cecile. **Superdlx:a generic superscalar simulator**. Canadá: *Advanced Compilers Architectures and Parallel Systems Group McGill University*, 1993. 84 pp.

21. Patterson, David A. y John L. Hennesy. **Computer Organization & Design: the hardware/software interface.** 2a ed. U.S.A. Morgan Kaufmann Publishers. 1998. 756 pp.

22. Schlansker, Michael S. **“EPIC: Explicitly Parallel Instruction Computing”**. IEEE Computer (U.S.A.) 33(2):37-45.2000.

23. Techonline University. http://www.techonline.com,2004

24. **Winsconsin University**. http://www.cs.wisc.edu/~arch/www/tools.html, 2003.

25. Wolf, Wayne**. “A Decade of Hardware/Software Codesign”**. IEEE Computer (U.S.A.) 36(4):38-43.2003.

26. Wolf, Wayne. **“How many systems Architectures?”**. IEEE Computer (U.S.A.) 36(3):93-95.2003.