TEMA 4. EL PARALELISMO INTERNO EN LOS SISTEMAS COMPUTADORES

4.1 INTRODUCIÓN. LA ARQUITECTURA DLX

El procesador DLX (DeLuXe) constituye una *sencilla arquitectura de carga/almacenamiento*. Es el resultado de una serie de experimentos y máquinas comerciales muy similares en filosofía a DLX. El nombre es consecuencia de un promedio expresado en números romanos: (AMD 29K, DEC STATION 3100, HP 850, IBM 801, Intel i860, MPS M/120 A, MIPS M/1000, MOTOROLA 88K, RISC I, SGI 4D/60, SPARC STATION-1, SUN-4/110, SUN-4/260) / 13 = 560 = DLX.

La arquitectura DLX se eligió tomando como base observaciones sobre las primitivas usadas en los programas más frecuentes.

Al igual que el resto de máquinas de carga/almacenamiento, DLX cumple:

1. Tiene un *sencillo repertorio de instrucciones de carga/almacenamiento*.
2. Tiene *segmentación* (*pipelining*) *eficiente*.
3. Tiene un repertorio de instrucciones *fácilmente decodificable*.
4. Tiene como objeto una *compilación eficiente*.

En definitiva, DLX es un buen modelo arquitectónico para el estudio de los sistemas computadores, y además es fácil de comprender.

4.1.1 Definición de la arquitectura DLX

El procesador DLX consta de:

- *REGISTROS*:
  - 32 *Registros de Propósito General* de una longitud de 32 bits → GPR (R0 – R31). El registro R0 siempre tiene el valor 0.
  - 32 *Registros de Punto Flotante* de una longitud de 32 bits → FPR (F0 – F31). Permiten el almacenamiento de datos según el estándar IEEE 754 de simple precisión. También permiten el almacenamiento de datos en doble precisión según el mismo estándar, utilizándose un registro par e impar consecutivos (F0-F1, F2-F3, …, F30-F31). Permiten realizar operaciones en simple y doble precisión. También existe un *Registro de Estado de FP*, que se utiliza para comparaciones y excepciones de FP; todas las transferencias a y desde el Registro de Estado FP se deben realizar a través de los GPR.
- *MEMORIA*: Está direccionada por palabras de tamaño igual a 1 byte. Tiene una capacidad de direccionamiento de 4 Gbytes, empleando 32 líneas de

dirección (A31 – A0). En el DLX se emplea el modo BIG ENDIAN[1] (byte de mayor peso en la dirección de memoria más baja). Todas las referencias a memoria se realizan a través de instrucciones de cargas y almacenamientos entre memoria y los registros GPR o FPR:

- Accesos que involucran a los registros GPR → Pueden implicar a distintos tamaños de datos: un BYTE, una MEDIA PALABRA, una PALABRA.
- Accesos que involucran a los registros FPR → Pueden realizarse con datos tanto en SIMPLE como en DOBLE PRECISIÓN.

Todos los accesos a memoria deben estar *alineados*.

Existen instrucciones de transferencia entre un registro FPR y un registro GPR.

- *INSTRUCCIONES*: Todas las instrucciones tienen un formato de 32 bits y deben estar *alineadas* en la memoria.
- *REGISTROS ESPECIALES*: Los registros especiales pueden transferirse a o desde los registros GPR; por ejemplo, el Registro de Estado FP (que se utiliza para almacenar la información sobre el resultado de la operación en FP).

4.1.2 Operaciones del procesador DLX

Las instrucciones del procesador DLX se clasifican según los cuatro grupos siguientes:

1. *Cargas* y *almacenamientos*.
2. *Operaciones ALU*.
3. *Saltos* y *Bifurcaciones*.
4. *Operaciones en Punto Flotante*.

Veamos a continuación en detalle cada una de ellas.

1. *CARGAS Y ALMACENAMIENTOS*. Estas instrucciones se caracterizan porque:

- Se pueden cargar o almacenar cualquiera de los registros GPR o FPR. En el registro R0 la carga no tiene efecto puesto que su valor va a ser siempre ‘0’.
- Las instrucciones de carga y almacenamiento tienen un único modo de direccionamiento: REGISTRO BASE + DESPLAZAMIENTO (de 16 bits; y puede ser positivo o negativo).

[1] Además del modo *Big Endian*, otros computadores pueden tener los modos *Little Endian* (Byte de menos peso en la dirección más baja) y *Bi Endian* (contempla tanto el modo Big Endian como el Little Endian)

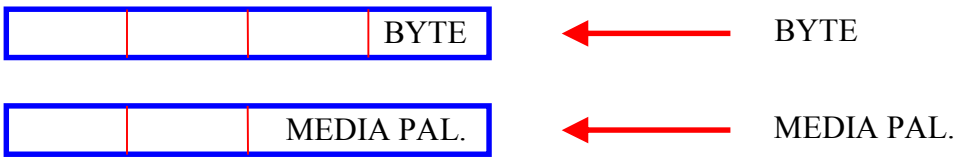


Figura 4.1 Carga de registros con datos de tamaño menor que la palabra.

- Las cargas de los datos de tamaño BYTE y MEDIA PALABRA ubican el objeto en la parte inferior del registro (ver Figura 4.1); el resto de los bits se rellenan, según el código de operación, con la extensión de signo o con ceros.
- El formato de coma flotante es según el estándar IEEE 754.

Instrucción	**Nemónico**	**Descripción de operación**
Cargar Palabra	LW R1, 30(R2)	$R1 \leftarrow_{32} M[30+R2]$
Cargar Palabra	LW R1, 10(R0)	$R1 \leftarrow_{32} M[10+0]$
Cargar Byte	LB R1, 40(R3)	$R1 \leftarrow_{32} (M[40+R3]_0)^{24} \#\# M[40+R3]$
Cargar Byte Sin Signo	LBU R1, 40(R3)	$R1 \leftarrow_{32} 0^{24} \#\# M[40+R3]$
Cargar Media Palabra	LH R1, 40(R3)	$R1 \leftarrow_{32} (M[40+R3]_0)^{16} \#\# M[40+R3] \#\# M[41+R3]$
Cargar Flotante	LF F0, 50(R2)	$F0 \leftarrow_{32} M[50+R2]$
Cargar Doble	LD F0, 50(R2)	$F0 \#\# F1 \leftarrow_{64} M[50+R2]$
Almacenar Palabra	SW 50(R4), R3	$M[50+R4] \leftarrow_{32} R3$
Almacenar Media Palabra	SH 52(R4), R3	$M[52+R4] \leftarrow_{16} R3_{16L\ 31}$
Almacenar Byte	SB 40(R4), R3	$M[40+R4] \leftarrow_{8} R3_{24L\ 31}$
Almacenar Flotante	SF 40(R3), F0	$M[40+R3] \leftarrow_{32} F0$
Almacenar Doble	SD 40(R3), F0	$M[40+R3] \leftarrow_{32} F0 \quad ; \quad M[44+R3] \leftarrow_{32} F1$

Tabla 4.1 Ejemplos de instrucciones de Carga / Almacenamiento.

- En la Tabla 4.1 se muestran ejemplos de instrucciones de carga y almacenamiento.

2. *OPERACIONES ALU*. Las instrucciones ALU se caracterizan porque:

- Todas tienen el modelo de ejecución REG – REG.
- Las operaciones que incluyen las instrucciones ALU son:
  - *Aritméticas*: Suma, Resta, Multiplicación y División.
  - *Lógicas*: AND, OR y XOR.
  - *De Desplazamiento* (*Shift*): Desplazamientos Lógicos y Desplazamientos Aritméticos, tanto a derecha como a izquierda.
  - *De Comparación*: $=, \neq, <, >, \leq, \geq$. Estas instrucciones tienen como efecto que si se cumple la condición definen como '1' el registro destino; en otro caso, lo definen como '0'. Puesto que el efecto de estas instrucciones no es más que inicializar con el valor '0' o '1' un registro, se nombran como Inicializa - *Set*: *Inicializa Igual*, *Inicializa No Igual*, *Inicializa Menor Que*, …

Todas estas instrucciones (excepto Multiplicación y División) tienen el modo de direccionamiento *inmediato* (inmediato con extensión de signo a 16 bits).

Instrucción	**Nemónico**	**Descripción de operación**
Suma	ADD R1, R2, R3	$R1 \leftarrow R2 + R3$
Suma Inmediato	ADDI R1, R2, #3	$R1 \leftarrow R2 + 3$
Cargar Alto Inmediato	LHI R1, #42	$R1 \leftarrow 42 \# \# 0^{16}$
Desplaz. Lógico a Izq.	SLL R1, R2, #5	$R1 \leftarrow R2 << 5$
Inicializa Menor Que	SLT R1, R2, R3	$IF\left(R2 < R3\right): \quad R1 \leftarrow 1 \; else \; R1 \leftarrow 0$

Tabla 4.2 Ejemplos de instrucciones aritméticas / lógicas en DLX, con y sin direccionamiento inmediato.

La instrucción *LHI* (*Carga Inmediato Más Significativo*) carga la mitad superior del registro con inmediato y coloca '0' en la parte menos significativa del registro. Por ejemplo, la instrucción LHI R1, #42, según la nomenclatura empleada en la Tabla 4.1, realiza: $R1 \leftarrow 42 \# \# 0^{16}$.

Una carga inmediata de un registro (abreviadamente LI R1, #3) se realizaría en DLX como ADDI R1, R0, #3 ($R1 \leftarrow 0+3$). De la misma forma, la instrucción para la arquitectura DLX de suma ADD R1, R0, R2 ($R1 \leftarrow 0+R2$) se puede especificar abreviadamente como MOV R1, R2.

- En la Tabla 4.2 se muestran ejemplos de instrucciones ALU.
- Para las operaciones de multiplicación y división de enteros MULT (con signo), MULTU (sin signo), DIV (con signo) y DIVU (sin signo), sus operandos deben estar en registros FPR.

3. *SALTOS Y BIFURCACIONES*. En DLX, los *Saltos* son rupturas de secuencia programadas condicionales y las *Bifurcaciones* son rupturas de secuencia programadas incondicionales.

- Hay dos tipos de *Saltos*:
  - *Salto* simple (sin retorno). Los modos de direccionamiento que admiten son:
    - *Desplazamiento con signo*: Desplazamiento de 26 bits que se suma al Contador de Programa (PC).
    - *Registro Diferido*: La dirección de bifurcación está definida en un registro.
  - *Salto Con Enlace* (con retorno). Son las que normalmente se conocen como *llamadas a subrutinas*. La dirección de retorno se coloca en el registro R31. Admite los mismos modos de direccionamiento que admite el Salto simple.
- En las *Bifurcaciones* se especifica la condición de ruptura de secuencia en el propio código de operación de la instrucción; y la dirección de destino se especifica con un desplazamiento con signo (de 16 bits), que se suma al PC. La condición define si la comparación del registro fuente es con '0' ó 'NO 0'.

  En la Tabla 4.3 se muestran ejemplos de saltos y bifurcaciones.
- Otras instrucciones que provocan rupturas de secuencia son:
  - TRAP. Transfiere el control a una rutina del Sistema Operativo. El desplazamiento es de 26 bits. TRAP #*Inm*. Las operaciones que realiza se definen por las transferencias: $IAR \leftarrow_{32} PC+4$; $PC \leftarrow_{32} 0^6 \# \# Inm$. (*IAR* es el Registro de Dirección de Interrupción – Dirección de Vuelta)

- RFE (*Retorno desde una Excepción*). Volver al código del usuario desde una excepción; se restaura el modo de usuario. La operación que realiza se define por la transferencia: $PC \leftarrow_{32} IAR$.

Instrucción	**Nemónico**	**Descripción de operación**
Salto	J *nombre*1	$PC \leftarrow nombre1$
Salto a Registro	JR R3	$PC \leftarrow R3$
Salto con Enlace	JAL *nombre*1	$R31 \leftarrow PC + 4 \quad ; \quad PC \leftarrow nombre1$
Salto Con Enlace a Registro	JALR R2	$R31 \leftarrow PC + 4 \quad ; \quad PC \leftarrow R2$
Bfurcación Si Igual a Cero	BEQZ R4, *nombre*2	$IF\left(R4 == 0\right): \quad PC \leftarrow nombre2$
Bifurcación si No Igual a Cero	BNEZ R4, *nombre*2	$IF\left(R4 \neq 0\right): \quad PC \leftarrow nombre2$

Tabla 4.3 Instrucciones típicas de flujo de control para el procesador DLX. *nombre*1 y *nombre*2 cumplen: $\left(PC + 4 - 2^{25}\right) \leq nombre1 < \left(PC + 4 + 2^{25}\right)$ y $\left(PC + 4 - 2^{15}\right) \leq nombre2 < \left(PC + 4 + 2^{15}\right)$.

4. *OPERACIONES EN PUNTO FLOTANTE*. Las instrucciones de punto flotante manipulan los registros FPR e indican si la operación a realizar es en simple o en doble precisión. Las operaciones de simple precisión se pueden aplicar a cualquiera de los 32 registros de punto flotante FPR (F0, …, F31), mientras que las operaciones en doble precisión se aplican únicamente a una pareja par-impar consecutiva que se designa por el número de registro par (F0, F2, F4, …, F30).

- *Instrucciones de Carga y Almacenamiento en Coma Flotante*. Estas instrucciones transfieren datos entre los registros de punto flotante y memoria en simple y doble precisión.
  - *Cargas*: LF (en simple precisión) y LD (en doble precisión).
  - *Almacenamientos*: SF (en simple precisión) y SD (en doble precisión)
- *Transferencias entre Registros FPR*. Para mover datos entre registros FPR:
  - MOVF (en simple precisión).

- MOVD (en doble precisión)

- *Transferencias entre Registros FPR y GPR*. Estas instrucciones permiten mover datos entre registros del tipo FPR y GPR:
  - MOVFP2I (de un registro FPR a otro GPR).
  - MOVI2FP (de un registro GPR a otro FPR).

  Para transferir un valor en doble precisión a dos registros enteros es necesario definir dos instrucciones del tipo MOVFP2I.

- *Operaciones de Conversión entre Coma Flotante y Entero*. Permiten transformar un dato entero a su representación en coma flotante y viceversa:
  - CVTF2D (Coma flotante en simple precisión a doble precisión).
  - CVTF2I (Coma flotante en simple precisión a entero).
  - CVTD2F (Coma flotante en doble precisión a simple precisión).
  - CVTD2I (Coma flotante de doble precisión a entero).
  - CVTI2F (Entero a coma flotante en simple precisión).
  - CVTI2D (Entero a coma flotante en doble precisión).

- *Operaciones en Coma Flotante*. Con estas instrucciones se pueden realizar operaciones de suma, resta, multiplicación y división tanto en simple como doble precisión. Se emplea el sufijo F para la simple precisión y el sufijo D para la doble precisión: ADDF, ADDD, SUBF, SUBD, MULTF, MULTD, DIVF y DIVD.

- *Comparación de Datos en Simple y en Doble Precisión*. La comparación de datos que están definidos en los registros de coma flotante, se puede hacer tanto para simple como para doble precisión. Para simple precisión se emplean: LTF, GTF, LEF, EQF y NEF; y para doble precisión: LTD, GTD, LED, EQD y NED.

  En estas instrucciones, el resultado de la comparación se almacena en el *Registro de Estado FP*.

- *Bifurcaciones*. Estas instrucciones comprueban el valor del bit del *Registro de Estado FP*:
  - BFPT (Bifurcación en Punto Flotante Si Cierto - *True*).
  - BFPF (Bifurcación en Punto Flotante Si Falso - *False*).

4.1.3 Formato de las instrucciones DLX

Todas las instrucciones del procesador DLX son de 32 bits, con un código de operación de 6 bits. La Figura 4.2 muestra los tres tipos de formatos de instrucción posibles en DLX: *Instrucción Tipo I*, *Instrucción Tipo R* e *Instrucción Tipo J*.

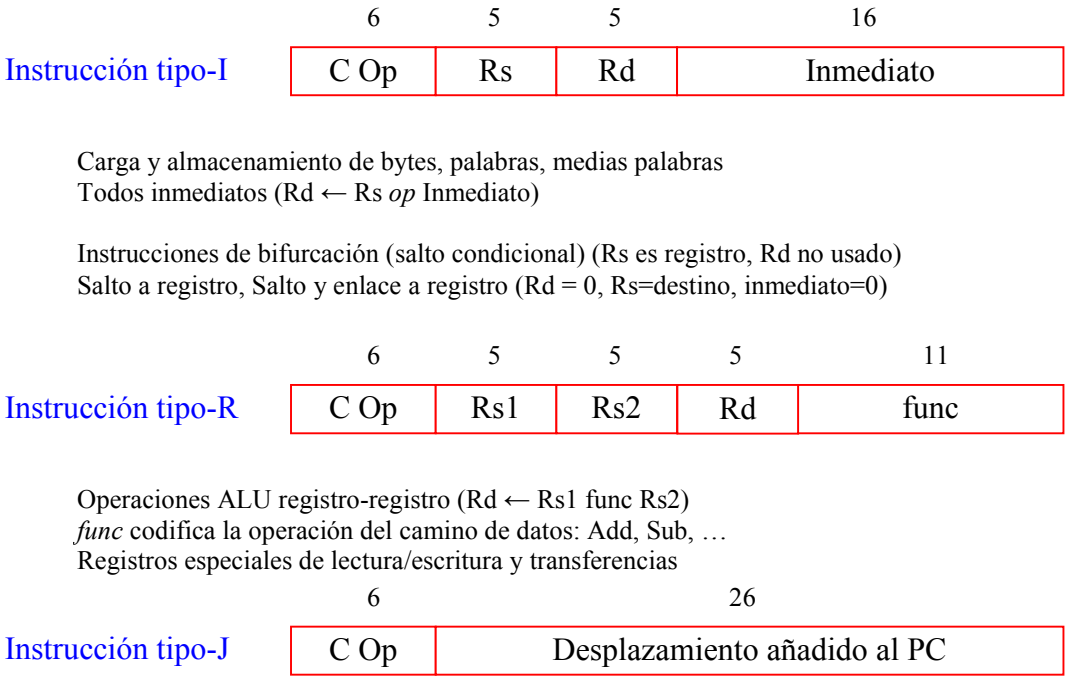


Figura 4.2 Formatos para las instrucciones DLX.

Todas las instrucciones pertenecientes al repertorio de instrucciones del procesador DLX se codifican según uno de los formatos definidos en la **Figura 4.2**.

A modo de resumen se muestra en Tabla 4.4 todas las instrucciones del procesador DLX, indicando su función y los tipos de formatos de instrucción que les corresponden.

Tipo de Instrucción	Formato	Significado de la Instrucción
Transferencia de datos		**Transfiere datos entre registros y memoria, o entre registros GPR, FPR o especiales; sólo el modo de direccionamiento de memoria es un desplazamiento de 16 bits + contenido de un GPR**
LB, LBU, SB	Tipo I	Carga byte, carga byte sin signo, almacena byte

LH, LHU, SH	Tipo I	Carga media palabra, carga media palabra sin signo, almacena media palabra
LW, SW	Tipo I	Carga palabra, almacena palabra (a/desde registros GPR)
LF, LD, SF, SD	Tipo I	Carga punto flotante SP, carga punto flotante DP, almacena punto flotante SP, almacena punto flotante DP
MOVI2S, MOVS2I	Tipo R	Transfiere desde/a GPR a/desde un Registro Especial
MOVF, MOVD	Tipo R	Copia uno o un par de registros FPR en DP en otro u otro par de registros FPR
MOVFP2I, MOVI2FP	Tipo R	Transfiere 32 bits desde/a registros FPR a/desde registros GPR
Aritmética / Lógica		**Operaciones lógicas o aritméticas sobre datos en registros GPR; la aritmética con signo causa trap si desbordamiento**
ADD, ADDI, ADDU, ADDUI	Tipo R Tipo I	Suma, suma inmediato (todos los inmediatos son de 16 bits); con signo y sin signo
SUB, SUBI, SUBU, SUBUI	Tipo R Tipo I	Resta, resta inmediato (todos los inmediatos son de 16 bits); con signo y sin signo
MULT, MULTU, DIV, DIVU	Tipo R	Multiplica y divide, con signo y sin signo; los operandos deben estar en registros de punto flotante; todas las operaciones tienen valores de 32 bits
AND, ANDI	Tipo R Tipo I	And, and inmediato
OR, ORI, XOR, XORI	Tipo R Tipo I	Or, or inmediato, or exclusiva, or exclusiva inmediato
LHI	Tipo I	Carga inmediato superior; carga la mitad superior del registro con inmediato
SLL, SRL, SRA,	Tipo R Tipo I	Desplazamientos: lógico a izquierda, lógico a derecha, aritmético a derecha; e inmediatos

SLLI, SRLI, SRAI		
S__, S__I	Tipo R Tipo I	Inicialización condicional: __ puede ser LT, GT, LE, GE, EQ, NE
Control		**Saltos y bifurcaciones relativos al PC (Contador de Programa) o mediante registros del GPR**
BEQZ, BNEZ	Tipo I	Bifurca si registro GPR es igual/no igual a cero; desplazamiento de 16 bits desde PC+4
BFPT, BFPF	Tipo I	Test de comparación de bit del registro FP y bifurcación; desplazamiento de 16 bits desde PC+4
J, JR	Tipo J Tipo I	Salto: desplazamiento de 26 bits desde PC (J) o destino en registro GPR (JR)
JAL, JALR	Tipo J Tipo I	Salto y enlace: guarda PC+4 en R31, el destino es relativo al PC (JAL) o está en un registro GPR (JALR)
TRAP	Tipo J	Transfiere el control a una rutina del sistema operativo; PC+4 se guarda en IAR y el desplazamiento de 26 bits define la dirección de bifurcación
RFE	Tipo J	Volver al código del usuario desde una excepción; restaurar modo de usuario
Punto Flotante		**Operaciones en coma flotante con formatos en doble y simple precisión**
ADDD, ADDF	Tipo R	Suma números coma flotante en DP, SP
SUBD, SUBF	Tipo R	Resta números coma flotante en DP, SP
MULTD, MULTF	Tipo R	Multiplica números en coma flotante DP, SP
DIVD, DIVF	Tipo R	Divide números en coma flotante DP, SP
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Tipo R	Instrucciones de conversión: CVTx2y convierte dato de tipo x a tipo y, donde x e y pueden ser de tipo I (entero), D (doble precisión) o F (simple precisión). Anbos operandos están en los registros FPR
__D, __F	Tipo R	Compara números en DP y SP: __ puede ser LT, GT,

		LE, EQ, NE; el bit de comparación lo pone en el Registro de Estado FP
NOP	Tipo R	No realiza ninguna operación

Tabla 4.4 Instrucciones del procesador DLX: Nemotécnico, tipo de formato y significado.

4.2 LA SEGMENTACIÓN Y LA DIVISIÓN FUNCIONAL

El innegable éxito comercial de la arquitectura de computador propuesta por John Von Neumann en el año 1946 se debe principalmente a su *versatilidad* y *sencillez*.

Al final de los años 80 aparecieron una amplia gama de máquinas que, buscaban con el *paralelismo* una forma de superar las limitaciones inherentes a dicha *arquitectura serie*.

El paralelismo en los sistemas computadores se justifica por dos razones:

- *La máxima potencia de cálculo alcanzable con la tecnología del momento*. La velocidad de transmisión en el silicio, teniendo en cuenta los retardos provocados por la conmutación, tiene un límite de $3 \cdot 10^7 m/s$. Si se considera una pastilla de $3\,cm$ de lado, el retardo longitudinal de las señales sería de $\frac{3 \cdot 10^{-2} m}{3 \cdot 10^7 m/s} = 10^{-9} s$. Por lo tanto, para un funcionamiento serie, la máxima potencia de cálculo alcanzable es de 10^9 operaciones de coma flotante por segundo (1 GFLOP). Y, como consecuencia, para satisfacer las cada vez mayores necesidades de potencia de cálculo son necesarias las máquinas paralelas.
- *El coste*. Si un circuito integrado se fabrica en grandes series tiene un coste muy bajo, siendo por tanto, desde el punto de vista del coste, más atractivo diseñar computadores con arquitecturas repetitivas en vez de mediante grandes monoprocesadores.

Por lo tanto, cuando se trata de resolver problemas específicos, y especialmente si éstos se pueden adaptar a la arquitectura paralela, con el paralelismo se consiguen unas inmejorables relaciones *potencia/coste*.

Existen distintas formas de llevar el paralelismo a los computadores, que pueden clasificarse como:

- *Paralelismo interno o implícito* (dentro de la CPU).
- *Paralelismo externo o explícito* (fuera de la CPU).

El *paralelismo interno* queda oculto a la arquitectura del computador; es un paralelismo a nivel de estructura y/o construcción del computador, que se utiliza para aumentar su velocidad sin modificar su funcionamiento básico. Aplicado a la arquitectura Von Neumann, permite construir computadores más rápidos pero que, a nivel de ejecución de instrucciones máquina, esto es, a nivel de usuario, siguen siendo serie. Las soluciones clásicas de este paralelismo son:

- *Segmentación o pipe-line*. Se aplica a la unidad de control.
- *División funcional*. Se aplica a la unidad operativa.

El *paralelismo explícito* queda visible al usuario, debiéndose encargar éste de explotarlo de forma adecuada.

Los conceptos de *segmentación* y *división funcional* coinciden en la idea pero se aplican a la unidad de control y a la unidad operativa respectivamente. La idea es similar al trabajo en cadena de la producción en serie, por ello nos referimos a la unidad segmentada como *cadena*.

La *segmentación* consiste en dividir la *función F* a realizar en una serie de *subfunciones* $F_1, F_2, \ldots, F_n$ que se pueden ejecutar de forma independiente. De esta forma, si se construyen unidades individuales para procesar cada *subfunción* F_i, se puede montar una *cadena* (ver **Figura 4.3**), que permite procesar simultáneamente n procesos; mientras que, una unidad que realice toda la *función F* sólo puede realizar un proceso al tiempo.

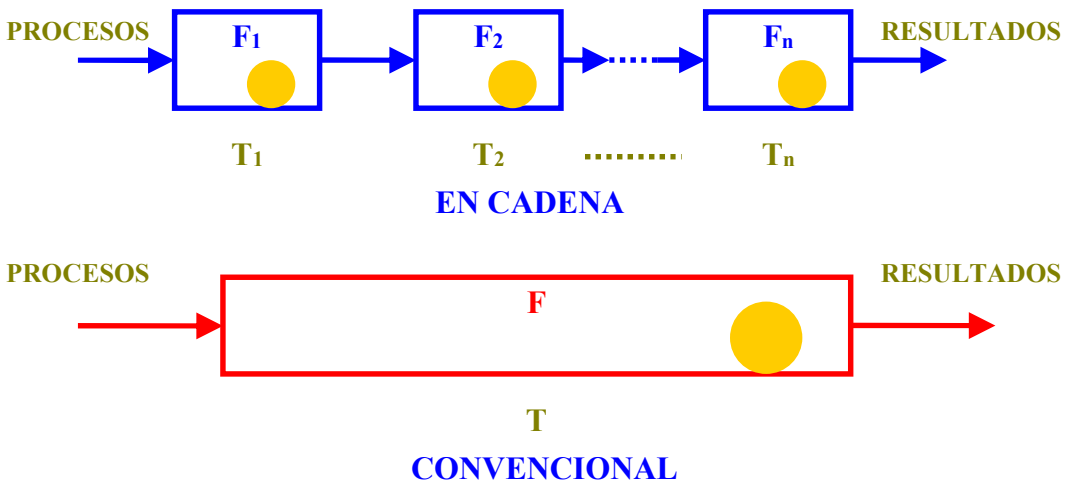


Figura 4.3 Proceso segmentado o pipe-line.

Dado que las subfunciones F_i son más sencillas que la total F, cada una de ellas se pueden realizar en una fracción T_i del tiempo T que se tarde en ejecutar toda la función F.

En caso de suponer que los tiempos de ejecución T_1, T_2, …, T_n, correspondientes a las subfunciones F_1, F_2, …, F_n respectivamente, sean todos iguales a T/n, la cadena permitiría resolver n procesos por cada uno de los que procesaría la unidad convencional. Sin embargo, para ese mismo supuesto, en ambos casos el tiempo que se tarda en resolver cada proceso sigue siendo T. Es decir, en ambos casos se tarda T en resolver un proceso pero, en la cadena el número de procesos resueltos por unidad de tiempo es n veces el número de procesos resueltos por unidad de tiempo en la unidad convencional.

La **Figura 4.4** representa el esquema de tiempos para el proceso segmentado. Obsérvese que, cuando el proceso P_1 pasa a la subfunción F_2, entra el proceso P_2 en la subfunción F_1, y así sucesivamente.

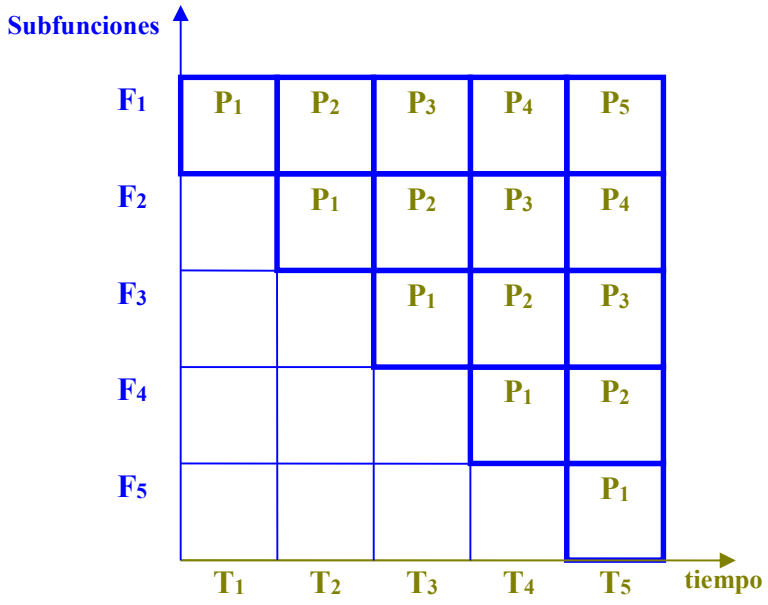


Figura 4.4 Esquema de tiempos del proceso segmentado.

Después del planteamiento realizado, nos preguntamos: ¿es siempre posible realizar una partición funcional como se propone de una función F?. La respuesta a esta pregunta es, que no es siempre posible; no es posible si no cumple alguna de las premisas que se plantean a continuación.

PREMISAS PARA PODER REALIZAR UNA PARTICIÓN FUNCIONAL DE F:

1. La evaluación de la función básica *F* debe poder descomponerse en la evaluación secuencial de una serie de subfunciones $F_1, F_2, \ldots, F_n$.
2. Las entradas de cada subfunción F_i deben venir determinadas exclusivamente por las salidas de la subfunción anterior en la secuencia de evaluación, esto es, de F_{i-1}.
3. No debe existir ninguna interrelación cruzada entre subfunciones, éstas sólo deben comunicarse de salida de F_i a entrada de F_{i+1}.
4. Cada subfunción debe ser calculable mediante un circuito específico y de forma más rápida que toda la función *F*. Cada uno de estos circuitos formará una etapa de la cadena.
5. Los tiempos requeridos para el cálculo de cada subfunción deben ser parecidos. Esta condición es muy importante ya que el tiempo de cálculo a asignar a cada subfunción debe ser el mismo, e igual al máximo de todos los tiempos T_i. La justificación de esta restricción es como consecuencia de que todos los datos deben cambiar de subfunción de forma simultánea; por todo ello, si los tiempos de cálculo T_i no son parecidos, se desaprovecha la velocidad de trabajo de las etapas rápidas.

La segmentación puede aplicarse a toda función compleja. En concreto, en los computadores se emplea en dos grandes módulos:

- *Unidad de Control*. La ejecución de instrucciones se divide en una serie de fases, cada una de las cuales se pueden realizar por circuitos específicos.
- *Unidad Operativa*. Los operadores complejos son también buenos candidatos para ser segmentados. Por ejemplo, los distintos pasos requeridos en las operaciones en coma flotante se pueden realizar por circuitos específicos, permitiendo construir una *cadena* con todos ellos.

También, en cierto modo, cuando una *Memoria Principal* está implementada de forma entrelazada, se le está aplicando segmentación. Aunque, en este caso, el concepto de la memoria entrelazada es ligeramente distinto puesto que los accesos consecutivos se realizan en módulos de memoria distintos, no existiendo interrelación ni comunicación entre estos módulos.

4.2.1 Segmentación de instrucciones

La *segmentación* (*pipelining*) es una técnica de implementación por la cual se solapa la ejecución de múltiples instrucciones. Actualmente, la segmentación se aplica de

forma generalizada en la CPUs. Cada uno de los pasos que definen la segmentación se define como *etapa de la segmentación* o *segmento*. Las etapas están conectadas, cada una a la siguiente, formando una especie de *cauce*: las instrucciones entran por un extremo, son procesadas a través de las etapas y salen por el otro extremo.

La *productividad de la segmentación* está determinada por la frecuencia con que una instrucción salga del cauce. Como las etapas están conectadas entre sí, todas las etapas deben estar listas para proceder al mismo tiempo. El tiempo requerido para desplazar una instrucción, un paso a lo largo del cauce, es un ***ciclo máquina***. La duración de un ciclo máquina está determinada por el tiempo que necesita la etapa más lenta. Con frecuencia el *ciclo máquina* es un *ciclo de reloj* (a veces dos o raramente más), aunque el reloj puede tener múltiples fases.

El *objetivo del diseñador* es equilibrar la duración de las etapas de la segmentación. Y, estar perfectamente equilibradas significa que el *tiempo por instrucción* de la máquina segmentada, suponiendo condiciones ideales (por ejemplo, sin detenciones) es igual a:

$$Tiempo\ por\ instrucción_{Máquina\ segmentada} = \frac{Tiempo\ por\ instrucción_{Máquina\ no\ segmentada}}{Nº\ etapas\ de\ la\ segmentación}$$

Bajo estas condiciones, la *mejora de la velocidad* debida a la segmentación es igual al número de etapas. Sin embargo, habitualmente, las etapas no están perfectamente equilibradas; además, la segmentación involucra algún gasto extra (por ejemplo *latchs* entre etapas). Así, el tiempo por instrucción en la máquina segmentada no tendrá su valor mínimo posible, aunque sí puede estar próximo (en un 10 % más por ejemplo).

La segmentación consigue una reducción en el *tiempo de ejecución medio por instrucción*. Esta reducción puede ser consecuencia de:

- Decrementar la duración del ciclo de reloj.
- Disminuir el número de ciclos de reloj medio por instrucción.
- Disminuir ambos.

La segmentación es una técnica de implementación que explota el paralelismo entre instrucciones de un flujo secuencial, teniendo como ventaja principal el *ser transparente al programador* (el programador sigue programando como en un computador serie).

4.3 SEGMENTACIÓN PARA DLX

Como ya se ha indicado anteriormente, DLX puede implementarse según cinco pasos básicos de ejecución:

1. ***IF*** → *BÚSQUEDA DE LA INSTRUCCIÓN*

$$\begin{aligned} MAR &\leftarrow PC \\ IR &\leftarrow M[MAR] \end{aligned}$$

2. ***ID*** → *DECODIFICACIÓN DE LA INSTRUCCIÓN / BÚSQUEDA DE REGISTROS*

$$\begin{aligned} &Decodificación \\ &PC \leftarrow PC + 4 \\ &A \leftarrow Rs1 \quad ; \quad B \leftarrow Rs2 \quad ; \quad Extensión\ de\ signo\ operando\ inmediato \end{aligned}$$

3. ***EX*** → *EJECUCIÓN / CÁLCULO DE DIRECCIÓN EFECTIVA*

Dependiendo del tipo de instrucción se realizan unas u otras operaciones de las que se muestran a continuación:

- **Instrucción de referencia a memoria (Carga o Almacenamiento)**

$$\begin{aligned} MAR &\leftarrow A + (IR_{16})^{16} \#\# IR_{16L\ 31}\ (Obtención\ dirección\ efectiva) \\ MDR &\leftarrow Rd \end{aligned}$$

- **Instrucción ALU**

$$ALU_{OUTPUT} \leftarrow A\ op\ \underbrace{\left(B\ o\ (IR_{16})^{16} \#\# IR_{16\cdots 31}\right)}_{\text{Extensión de signo del operando inmediato}}$$

- **Salto/Bifurcación**

$$ALU_{OUTPUT} \leftarrow PC + \left\{ \underbrace{(IR_{16})^{16} \#\# IR_{16\cdots 31}}_{\substack{\text{Extensión de signo del} \\ \text{desplazamiento en un Salto}}} \right\} ó \left\{ \underbrace{(IR_{6})^{6} \#\# IR_{6\cdots 31}}_{\substack{\text{Extensión de signo del} \\ \text{desplazamiento en una Bifurcación}}} \right\}$$

$$COND \leftarrow A \underbrace{op}_{\substack{\text{Según operador} \\ \text{relacional} \\ \text{considerado}}} \left(0\ ó\ \overline{0}\right)$$

$$si\ \ JAL: \quad R31 \leftarrow Dirección\ de\ retorno$$

$$si\ \ TRAP: \quad PC \leftarrow IAR(\text{Registro de Dirección de Interrupción})$$

4. ***MEM*** → *ACCESO A MEMORIA / COMPLETAR EL SALTO*

   Las únicas instrucciones activas en esta fase son: Cargas, Almacenamientos, Saltos y Bifurcaciones:

- **Cargas**

$$MDR \leftarrow M[MAR]$$

- **Almacenamientos**

$$M[MAR] \leftarrow MDR$$

- **Bifurcaciones**

$$si \quad COND: \quad PC \leftarrow ALU_{OUTPUT}$$

- **Saltos**

$$PC \leftarrow ALU_{OUTPUT}$$

5. ***WB*** → *WRITE-BACK – POST-ESCRITURA*

$$Rd \leftarrow \underbrace{ALU_{OUTPUT}}_{\text{Si viene de la ALU}} \quad \acute{o} \quad \underbrace{MDR}_{\text{Si viene de memoria}}$$

Se puede segmentar DLX buscando sencillamente una nueva instrucción en cada ciclo de reloj. En este caso, cada una de las fases anteriores se convertiría en una *etapa de la segmentación*, dando como resultado el patrón de acceso mostrado en la **Figura 4.5**. Aunque cada instrucción necesita cinco ciclos de reloj, durante cada ciclo de reloj el hardware está ejecutando alguna fase de cinco instrucciones diferentes.

NÚMERO DE INSTRUCCIÓN	CICLO DE RELOJ								
	1	2	3	4	5	6	7	8	9
i	**IF**	**ID**	**EX**	**MEM**	**WB**				
i+1		**IF**	**ID**	**EX**	**MEM**	**WB**			
i+2			**IF**	**ID**	**EX**	**MEM**	**WB**		
i+3				**IF**	**ID**	**EX**	**MEM**	**WB**	
i+4					**IF**	**ID**	**EX**	**MEM**	**WB**

Figura 4.5 Segmentación sencilla de DLX.

La segmentación incrementa la *productividad de instrucciones de la CPU* (número de instrucciones completadas por unidad de tiempo), pero no reduce el *tiempo de ejecución de una instrucción individual* (de hecho, habitualmente incrementa

ligeramente el tiempo de ejecución de cada instrucción debido al *gasto en el control de la segmentación*). Aumentar la productividad de instrucciones significa que un programa se ejecuta más rápidamente, teniendo menor tiempo total de ejecución.

El hecho de que el tiempo de ejecución de cada instrucción no se mejore pone límites a la *profundidad práctica de la segmentación* (esto se tratará en el siguiente apartado de este tema). Otras consideraciones de diseño limitan la frecuencia de reloj que puede alcanzarse mediante una segmentación más profunda. La consideración más importante es el efecto combinado del *retardo de los cerrojos* (*latchs*) y el *flanco de reloj* (*clock skew*). Los cerrojos son necesarios entre etapas del cauce, sumando al *tiempo de preparación* (*setup*) el retardo a través de los cerrojos en cada periodo de reloj. El flanco de reloj también contribuye al límite inferior del ciclo de reloj:

> *Una vez que el ciclo de reloj sea tan pequeño como la suma del flanco de reloj y el gasto de los cerrojos, no es útil aplicar una mayor segmentación.*

Ejemplo 4.1 Considerar una máquina no segmentada con cinco pasos de ejecución cuyas duraciones son 50 ns, 50 ns, 60 ns, 50 ns y 50 ns. Suponer que debido al tiempo de preparación (*setup*) y al flanco de reloj (*clock skew*), el segmentar la máquina añade 5 ns de gasto a cada etapa de ejecución. Ignorando cualquier impacto de latencia, ¿cuánta velocidad se ganará en la frecuencia de ejecución de las instrucciones con la segmentación?.

SOLUCIÓN: En la máquina no segmentada, el tiempo de ejecución por instrucción medio:

$$t_{\substack{Ejec.por\ Inst.Medio \\ Máquina\ No\ Segmentada}} = (50 + 50 + 60 + 50 + 50)\ ns = 260\ ns$$

En la implementación segmentada, el ciclo máquina debe ir a la velocidad de la etapa más lenta más el gasto de la segmentación, que será el tiempo de ejecución por instrucción medio de la máquina segmentada:

$T_{\text{ejecución medio\_máquina segmentada}} = T_{\text{total}}/\text{Número de Instrucciones} = 8 \text{ ciclos} * (60 + 5) / 4 = 130 \text{ ns}$

Por lo tanto, la *aceleración* obtenida por la segmentación es:

Aceleración = $T_{\text{ejecución medio\_máquina no segmentada}} / T_{\text{ejecución medio\_máquina segmentada}} = 2$

En la **Figura 4.6** se muestra el patrón de ejecución para cuatro instrucciones, tanto para la máquina no segmentada como para la segmentada.

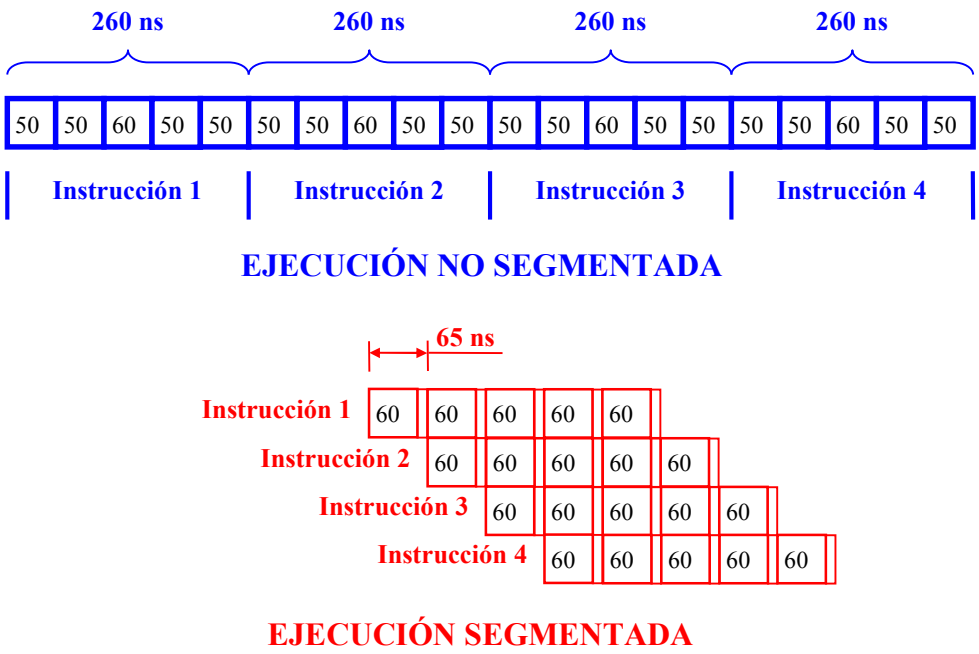


Figura 4.6 Patrón de ejecución para cuatro instrucciones, sin segmentación y con segmentación.

El gasto de la segmentación de los 5 ns establece esencialmente un límite en la efectividad de la segmentación. Si modificar el ciclo de reloj no afecta al gasto de la segmentación, la Ley de Amdahl nos indica que el gasto de la segmentación limita la velocidad.

Debido a que los cerrojos (*latchs*) de un diseño segmentado pueden tener un impacto significativo sobre la velocidad de reloj, los investigadores han diseñado cerrojos que permiten la frecuencia de reloj más alta posible (*Cerrojo de Earle* – 1965).

4.3.1 Modificación para que la segmentación DLX funcione

La segmentación no es tan sencilla como se ha planteado hasta ahora. Se van a tratar de resolver los problemas que introduce la segmentación.

En primer lugar, hay que determinar *qué ocurre en cada ciclo de reloj* y asegurarnos que el solapamiento de las instrucciones no utiliza los recursos más allá de sus posibilidades (por ejemplo, a una única ALU no se le puede pedir que calcule una dirección efectiva y realice una operación de resta a la vez).

Las operaciones que se han definido para las distintas fases de ejecución necesitan algunas modificaciones para que puedan ejecutarse de forma segmentada. La Tabla 4.5 lista las principales unidades funcionales de nuestra implementación de DLX (en el eje horizontal), las etapas de la segmentación (en el eje vertical) y lo que tiene que ocurrir en cada etapa.

Etapa	Unidad PC	Memoria	Camino de datos
IF	PC ← PC+4	IR ← M[PC]	
ID	PC1 ← PC	IR1 ← IR	A ← Rs1; B ← Rs2
EX			Instrucciones de Carga/Almacenamiento: DMAR ← A+$(IR1_{16})^{16}$##$IR1_{16..31}$ SMDR ← B Instrucciones ALU: ALU_{output} ← A op (B or $(IR1_{16})^{16}$##$IR1_{16..31}$) Instrucciones Bifurcación: ALU_{output} ← PC1+ $(IR1_{16})^{16}$##$IR1_{16..31}$) Cond ← (Rs1 op 0); Instrucciones Salto: ALU_{output} ← PC1+ $(IR1_{26})^{6}$##$IR1_{6..31}$)
MEM	If (cond): PC ← ALU_{output}	LMDR← M[MAR] o M[MAR]← SMDR	$ALU_{output}1$ ← ALU_{output}
WB			Rd ← $ALU_{output}1$ o LMDR

Tabla 4.5 Principales unidades funcionales y ocurrencias en cada etapa de la segmentación.

En la Tabla 4.6 se muestra información similar pero, en el eje horizontal se muestra el tipo de instrucción. Por supuesto, la combinación de instrucciones que pueden estar siendo ejecutadas en cualquier instante es arbitraria. Como la instrucción no está todavía codificada, las dos primeras etapas de la segmentación son idénticas. Al comienzo de la operación de la ALU, las entradas correctas son multiplexadas según el código de operación (que ya había sido decodificado en la etapa anterior).

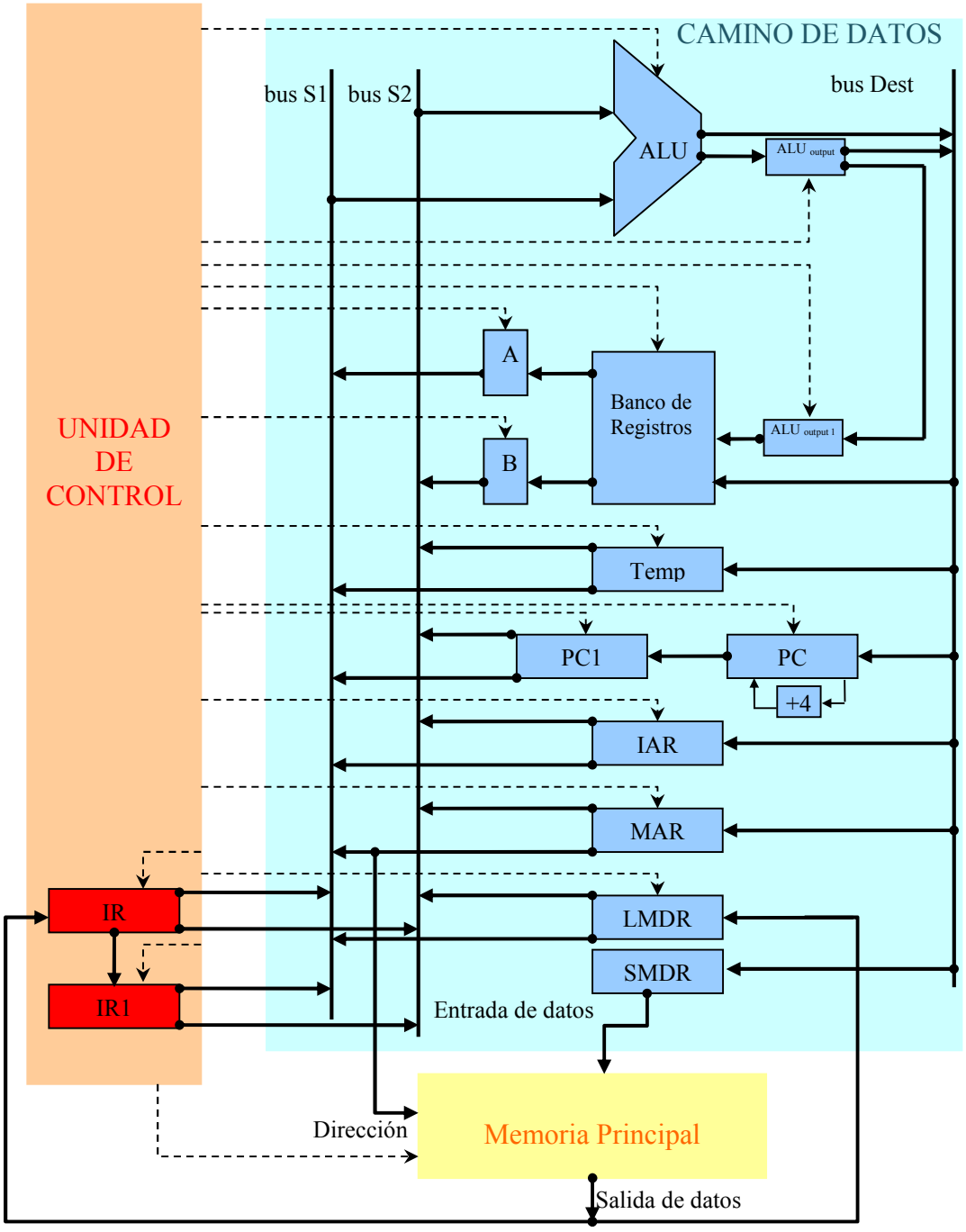
Cada etapa de la segmentación se activa en cada ciclo de reloj. Esto requiere que todas las operaciones asociadas a una etapa se completen en un ciclo de reloj y que cualquier combinación de operaciones se pueda presentar a la vez. A continuación se muestra todo lo que implica esto (**Figura 4.7**):

1. El PC se debe incrementar en cada ciclo de reloj, lo que debe hacerse en IF en vez de en ID. Como la ALU se utiliza en otras fases para otras operaciones, no se puede utilizar para incrementar PC; por lo tanto hay que incluir un *circuito incrementador del PC* aparte.
2. En cada ciclo de reloj se debe buscar una nueva instrucción, que también se hace en IF.

3. En cada ciclo de reloj se necesita una nueva palabra de datos, que se hace en MEM.
4. Debe haber un *MDR doble*; uno para cargas (LMDR) y otro para almacenamientos (SMDR), ya que cuando son instrucciones consecutivas se solapan en el tiempo.
5. Se necesitan *tres cerrrojos adicionales* que contengan los valores que se necesitarán más tarde, pero que se pueden modificar con una instrucción posterior. Los valores almacenados son: la instrucción (en IR1), la salida de la ALU (en $ALU_{output}1$) y el siguiente PC (en PC1).

Etapa	Instrucción ALU	Carga/Almacenamiento	Salto/Bifurcación
IF	IR ← M[PC] PC ← PC+4	IR ← M[PC] PC ← PC+4	IR ← M[PC] PC ← PC+4
ID	A ← Rs1; B ← Rs2 PC1 ← PC IR1 ← IR	A ← Rs1; B ← Rs2 PC1 ← PC IR1 ← IR	A ← Rs1; B ← Rs2 PC1 ← PC IR1 ← IR
EX	ALU_{output} ← A op (B or $(IR1_{16})^{16}$##$IR1_{16..31}$)	MAR ← A+$(IR1_{16})^{16}$##$IR1_{16..31}$ SMDR ← B	Bifurcación ALU_{output} ← PC1+ $(IR1_{16})^{16}$##$IR1_{16..31}$ Cond ← (Rs1 op 0); Salto ALU_{output} ← PC1+ $(IR1_{26})^{6}$##$IR1_{6..31}$
MEM	$ALU_{output}1$ ← ALU_{output}	LMDR ← M[MAR] o M[MAR] ← SMDR	If (cond): PC ← ALU_{output}
WB	Rd ← $ALU_{output}1$	Rd ← LMDR	

Tabla 4.6 Eventos en cada etapa de la segmentación de DLX diferenciados según el tipo de instrucción.



Opciones de salida de la ALU:

- S1+S2
- S1-S2
- S1&S2
- S1|S2
- S1^S2
- S1<<S2
- S1>>S2
- $S1>>_a S2$
- S1
- S2
- 0
- 1

IAR → Registro de dirección de interrupción
MAR → Registro de direcciones de memoria
LMDR → Registro de datos de memoria para Carga
SMDR → Registro de datos de memoria para Almacenamiento
IR → Registro de instrucción
IR1 → Registro de instrucción adicional
PC→ Contador de programa
PC1→ Contador de programa adicional
$ALU_{output1}$ → Registro de salida de ALU adicional

Figura 4.7 Modificación a la arquitectura para que la segmentación DLX funcione.

Probablemente, el impacto más grande de la segmentación sobre los recursos del sistema computador está en el sistema de memoria. Aunque el tiempo de acceso a memoria no haya cambiado, el ancho de banda máximo (*peak*) de memoria se debe incrementar cinco veces con respecto a la máquina no segmentada (mientras que en la máquina segmentada se requieren dos accesos a memoria en cada ciclo de reloj, en la máquina no segmentada con el mismo número de pasos por instrucción se requieren dos accesos a memoria cada cinco ciclos de reloj). Para que el sistema de memoria pueda proporcionar dos accesos en cada ciclo de reloj, la mayoría de los computadores utilizan memorias cachés separadas (una para instrucciones y otra para datos).

En la etapa EX, aunque la ALU puede utilizarse para tres funciones diferentes (cálculo efectivo de una dirección de dato, cálculo de una dirección de salto o una operación ALU), para una instrucción concreta únicamente se realizará una de ellas; de esa forma no hay conflictos.

La segmentación planteada para DLX funcionaría bastante bien si cada instrucción fuese independiente del resto de instrucciones que se ejecutan al mismo tiempo pero, en realidad las instrucciones pueden ser dependientes unas de otras (esto es lo que se trata en el siguiente apartado.

4.4 LOS RIESGOS DE LA SEGMENTACIÓN

Hay situaciones, llamadas *riesgos* (*hazards*), que impiden que se ejecute la siguiente instrucción del flujo de instrucciones durante su ciclo de reloj designado. Los riesgos reducen el rendimiento que se obtiene con la velocidad ideal de la segmentación. Hay tres clases de riesgos:

1. *Riesgos estructurales*. Surgen como consecuencia del hardware no poder soportar todas las combinaciones posibles de las instrucciones en ejecuciones simultáneas.
2. *Riesgos por dependencias de datos*. Surgen cuando una instrucción depende de los resultados de una instrucción anterior, de forma que ambas podrían llegar a ejecutarse de forma solapada.
3. *Riesgos de control*. Surgen de la segmentación de los saltos y otras instrucciones que cambian el PC.

Los riesgos de la segmentación pueden hacer necesario detener el cauce de instrucciones. Una detención en una máquina segmentada requiere, con frecuencia, que continúen ejecutándose algunas instrucciones mientras se detienen otras (se retardan). Normalmente, cuando una instrucción está detenida, todas las instrucciones

posteriores a ésta también se detienen. Las instrucciones anteriores a la instrucción detenida pueden continuar, pero no se buscan instrucciones nuevas durante la detención. En esta sección se verán algunos ejemplos de estas detenciones.

Una detención hace que el rendimiento de la segmentación se degrade (se reduzca) con respecto al rendimiento ideal. La siguiente ecuación muestra la *aceleración real de la segmentación*:

$$Aceleración_{Segm.} = \frac{t_{\substack{ejec.inst.medio \\ SIN\ SEGM.}}}{t_{\substack{ejec.inst.medio \\ CON\ SEGM.}}} = \frac{CPI_{\substack{medio \\ SIN\ SEGM.}} \cdot t_{\substack{ciclo \\ SIN\ SEGM.}}}{CPI_{\substack{medio \\ CON\ SEGM.}} \cdot t_{\substack{ciclo \\ CON\ SEGM.}}} = \frac{t_{\substack{ciclo \\ SIN\ SEGM.}}}{t_{\substack{ciclo \\ CON\ SEGM.}}} \cdot \frac{CPI_{\substack{medio \\ SIN\ SEGM.}}}{CPI_{\substack{medio \\ CON\ SEGM.}}} \qquad (4.1)$$

Como ya se ha indicado anteriormente, la segmentación consigue una reducción en el $t_{ejec.inst.medio}$, siendo esta reducción consecuencia de:

- Disminución del CPI_{medio}
- Disminución del t_{ciclo}
- Disminución de ambos

Supongamos a partir de ahora que es como consecuencia del CPI_{medio}. El $CPI_{medio\ IDEAL}$ en una máquina segementada es habitualmente:

$$CPI_{medio\ IDEAL} = \frac{CPI_{\substack{medio \\ SIN\ SEGM.}}}{Profundidad\ SEGM}$$

Sustituyendo en la ecuación de la segmentación (4.1), se obtiene:

$$Aceleración_{Segm.} = \frac{t_{\substack{ciclo \\ SIN\ SEGM.}}}{t_{\substack{ciclo \\ CON\ SEGM.}}} \cdot \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{\substack{medio \\ CON\ SEGM.}}} \qquad (4.2)$$

que, si nos limitamos a las detenciones de la segmentación, es decir:

$$CPI_{\substack{medio \\ CON\ SEGM.}} = CPI_{medio\ IDEAL} + C_{\substack{detencion \\ SEGM.}} PI_{medio}$$

se obtiene la siguiente expresión para la aceleración:

$$Aceleración_{Segm.} = \frac{t_{\substack{ciclo \\ SIN\ SEGM.}}}{t_{\substack{ciclo \\ CON\ SEGM.}}} \cdot \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}} \qquad (4.3)$$

Aunque esto da una fórmula general para la aceleración de la segmentación (en la que se han ignorado detenciones distintas a las de la segmentación), se puede utilizar una

expresión más simple aún si se ignora el decremento potencial de la frecuencia de reloj debido al gasto de la segmentación:

$$Aceleración_{Segm.} = \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}} \quad (4.4)$$

Pero, aunque se utilice esta forma más simple de evaluar la aceleración de la segmentación de DLX, un diseñador, al evaluar las estrategias de segmentación, debe tener cuidado de no descontar el impacto potencial sobre la frecuencia de reloj.

4.4.1 Riesgos estructurales

Cuando se segmenta una máquina, la ejecución solapada de instrucciones requiere la segmentación de unidades funcionales y duplicar los recursos para permitir todas las posibles combinaciones de instrucciones. Si alguna combinación de instrucciones no se puede acomodar debido a conflictos de recursos, se dice que la máquina tiene un *riesgo estructural*. Esto suele ocurrir normalmente:

- Cuando alguna unidad funcional no está completamente segmentada; no pudiendo iniciar secuencialmente el procesador ninguna secuencia de instrucciones en las que todas utilicen esa unidad funcional, o
- Cuando algunos recursos no se han duplicado lo suficiente, para permitir la ejecución de todas las combinaciones de instrucciones (por ejemplo, una máquina puede tener solamente un puerto de escritura en el banco de registros, pero, bajo ciertas circunstancias, puede existir la necesidad de realizar dos escrituras en un ciclo de reloj).

Cuando una secuencia de instrucciones encuentre riesgo, el procesador detendrá una de las instrucciones hasta que esté disponible el recurso requerido.

Si en una máquina segmentada hay una única memoria para datos y para instrucciones, y un único puerto de acceso a ella, cuando una instrucción contenga una referencia para datos, la segmentación debe detenerse durante un ciclo de reloj (la máquina no puede buscar la siguiente instrucción). La Figura 4.7 muestra que una segmentación con un solo puerto de memoria implica una *detención* durante una búsqueda de instrucción. Veremos otro tipo de detenciones cuando hablemos de otros tipos de riesgos.

Ejemplo 4.2 Supongamos que las referencias a datos constituyen el 30 % de la mezcla y que el $CPI_{medio\ IDEAL}$ de la máquina segmentada, ignorando los riesgos estructurales, es 1.2. Sin considerar ninguna otra pérdida de rendimiento, ¿cuántas veces es más

rápida la máquina ideal sin los riesgos estructurales de memoria, frente a la máquina con dichos riesgos?

SOLUCIÓN: La máquina ideal será más rápida según la relación de la aceleración de la máquina ideal sobre la máquina real. Como las frecuencias de reloj no están afectadas, se puede utilizar la expresión (4.4) para la aceleración de la segmentación:

$$Aceleración_{Segm.} = \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}}$$

Como la máquina ideal no tiene detenciones, la aceleración de la segmentación es sencillamente:

$$Aceleración_{\substack{Segm. \\ SIN\ RIESGOS}} = \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}} = \frac{1.2 \cdot Profundidad\ SEGM.}{1.2 + 0}$$

La aceleración de la segmentación en la máquina real es:

$$Aceleración_{\substack{Segm. \\ CON\ RIESGOS}} = \frac{1.2 \cdot Profundidad\ SEGM.}{1.2 + 0.3 \cdot 1}$$

Por lo tanto, la máquina ideal sin riesgos estructurales es más rápida que la que tiene riesgos estructurales el siguiente número de veces:

$$\frac{Aceleración_{\substack{Segm. \\ SIN\ RIESGOS}}}{Aceleración_{\substack{Segm. \\ CON\ RIESGOS}}} = \frac{\frac{1.2 \cdot Profundidad\ SEGM.}{1.2}}{\frac{1.2 \cdot Profundidad\ SEGM.}{1.5}} = \frac{1.5}{1.2} = 1.25$$

Por lo tanto, la máquina sin riesgos estructurales sería el 25 % más rápida.

Instrucción	**Número de ciclo de reloj**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrucción de carga	IF	ID	EX	MEM	WB				
Instrucción $i+1$		IF	ID	EX	MEM	WB			
Instrucción $i+2$			IF	ID	EX	MEM	WB		
Instrucción $i+3$				**Det.**	IF	ID	EX	MEM	WB
Instrucción $i+4$						IF	ID	EX	MEM

Figura 4.8 Detención durante un riesgo estructural, una instrucción de carga con una memoria que sólo tiene un puerto de acceso.

Si los demás factores son iguales, una máquina sin riesgos estructurales tendrá siempre un CPI_{medio} más bajo. Entonces, ¿por qué permite riesgos estructurales un diseñador? Existen dos razones:

- Para reducir costes. La segmentación de todas las unidades funcionales puede ser muy costosa (una memoria que soporte referencias a memoria de un ciclo de reloj requiere un ancho de banda total de dos veces ese valor; y, por ejemplo, para segmentar completamente un multiplicador de punto flotante se necesitan muchas puertas). Por lo tanto, si los riesgos estructurales no se presentan con frecuencia, puede no merecer la pena el coste de evitarlos.
- Para reducir la latencia de la unidad. Es posible, habitualmente, diseñar una unidad no segmentada, o que no esté segmentada completamente, con un retardo total menor que el de una unidad completamente segmentada (por ejemplo, los diseñadores de las unidades de punto flotante del CDC 7600 y del MIPS R2010 prefirieron una latencia menor (menos ciclos por operación) que una segmentación completa. Además, reducir la latencia tiene otros beneficios en el rendimiento que, frecuentemente, pueden superar las desventajas de los riesgos estructurales.

4.4.2 Riesgos por dependencias de datos

Un efecto importante de la segmentación es cambiar la temporización relativa de las instrucciones al solapar su ejecución; esto, introduce riesgos por dependencias de datos y riesgos de control. Los *riesgos por dependencias de datos* se presentan cuando el orden de acceso a los operandos los cambia la segmentación, con relación al orden normal que se sigue en las instrucciones que se ejecutan secuencialmente. Por ejemplo, considérese la ejecución encauzada de las instrucciones:

$ADD \quad R1, R2, R3$

$SUB \quad R4, R1, R5$

Instrucción	**Número de ciclo de reloj**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrucción ADD		IF	ID	EX	MEM	$WB_{Escrito}$			
Instrucción SUB			IF	$ID_{Leído}$	EX	MEM	WB		

Figura 4.9 La instrucción ADD escribe en un registro que es un operando fuente para la instrucción SUB. Pero ADD no termina de escribir el dato en el banco de registros hasta tres ciclos de reloj después de que SUB comienza a leerlo.

La instrucción *SUB* tiene una fuente, *R1*, que es el destino de la instrucción *ADD*. Como muestra la **Figura 4.9**, la instrucción *ADD* escribe el nuevo valor de *R1* en la etapa WB, pero la instrucción *SUB* lee el valor durante su etapa ID. Por lo tanto, a menos que se tomen precauciones para prevenirlo, la instrucción *SUB* leerá el valor erróneo y lo utilizará. Si por casualidad, se presentara una interrupción entre las instrucciones *ADD* y *SUB*, se completaría la etapa WB de *ADD*, y el valor de *R1* sería correcto para *SUB*. Por supuesto, este comportamiento impredecible es inaceptable obviamente.

El problema que se ha planteado en este ejemplo puede resolverse con una simple técnica hardware llamada *adelantamiento* (*forwarding*) (también llamada *desvío* (*bypassing*) y a veces *cortocircuito*). La técnica consiste en:

El resultado de la ALU siempre realimenta sus cerrojos de entrada; si el hardware de adelantamiento detecta que la operación previa de la ALU ha escrito en un registro que es fuente para la operación actual de la ALU, la lógica de control selecciona el resultado adelantado como entrada de la ALU en lugar del valor leído del banco de registros.

Observar que con el adelantamiento, si *SUB* es detenida, le dará tiempo a completarse a *ADD* y no se activará el desvío, haciendo que se utilice el valor del registro que corresponda. Esto también es cierto para el caso de una interrupción entre las dos instrucciones.

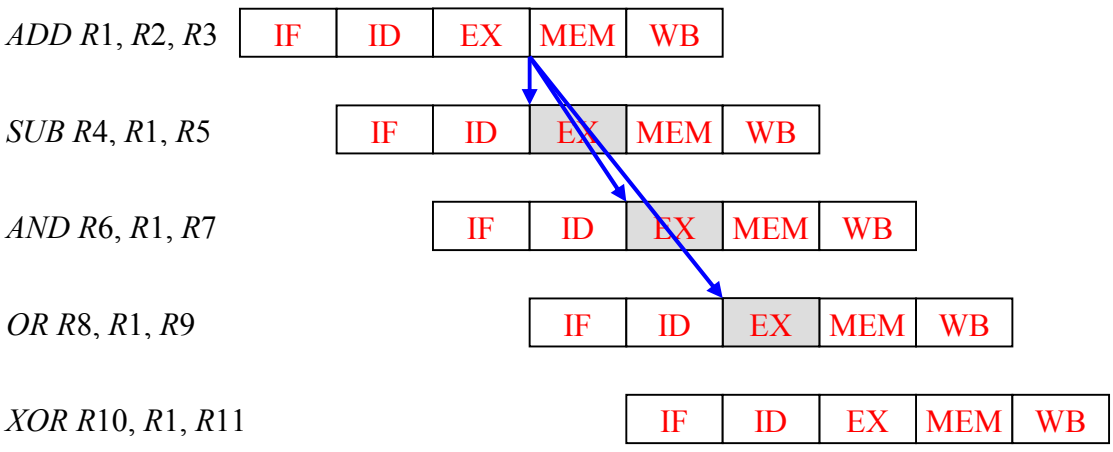


Figura 4.10 Ejemplo de conjunto de instrucciones para el procesador DLX, que necesitan se aplique adelantamientos por dependencias de datos.

Refiriéndonos a la segmentación planteada para DLX, no sólo hay que pasar resultados a la instrucción inmediatamente siguiente, sino también a la instrucción que

venga después de esa. Incluso para una instrucción tres líneas más abajo, se solapan las etapas ID y WB; por lo tanto, como la escritura no se termina hasta el fin de WB, hay que continuar adelantando el resultado. La **Figura 4.10**, muestra un ejemplo de repertorio de instrucciones y las operaciones de adelantamiento que se pueden presentar: la instrucción ADD inicializa $R1$, y las cuatro instrucciones siguientes lo utilizan; el valor de $R1$ debe ser desviado para las instrucciones SUB, AND y OR, pero, en el instante en que la instrucción XOR va a leer R1 en la fase ID, la instrucción ADD ha completado WB, con lo cual no hay que desviar.

Es deseable reducir el número de instrucciones que deban ser desviadas ya que, cada nivel requiere hardware especial. Para ello, como el banco de registros es accedido dos veces en un ciclo de reloj, es posible hacer las escrituras en el registro en la primera mitad de WB y las lecturas en la segunda mitad de ID; y de esta forma se elimina la necesidad de desviación para una tercera instrucción, como se muestra en la **Figura 4.11**: las instrucciones SUB y AND necesitan todavía que el valor de $R1$ sea desviado, y esto ocurre cuando entran en su etapa EX; sin embargo, durante la instrucción OR, que también usa $R1$, se ha completado la escritura de $R1$, no necesitándose adelantamiento (aunque XOR depende de ADD, el valor de $R1$ obtenido en ADD se escribe siempre, en el mismo ciclo, antes que XOR alcance su etapa ID y lo lea.

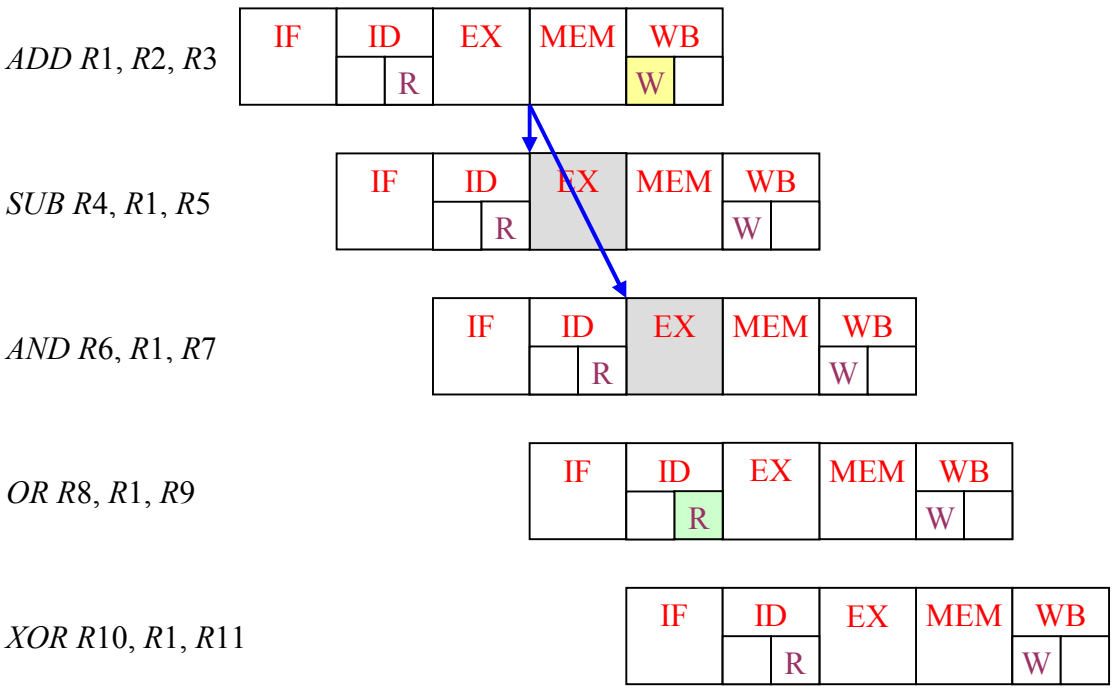


Figura 4.11 Ejemplo de conjunto de instrucciones para el procesador DLX (las mismas que las del ejemplo 4.7) con lecturas y escrituras de registros en las mitades opuestas de las etapas ID y WB.

Como se ha indicado anteriormente, cada nivel de desvío requiere de un hardware especial. Cada uno de estos niveles requiere un *cerrojo* y un *par de comparadores* que examinen si instrucciones adyacentes comparten un destino y una fuente. En la **Figura 4.12** se muestra la estructura de la ALU y su unidad de desvío, así como los valores que están en los registros de desvío según la secuencia de instrucciones dada en la **Figura 4.10**: los contenidos del buffer se muestran en el punto donde la instrucción AND de la secuencia de código de la **Figura 4.11** está aproximadamente al comienzo de la etapa EX; la instrucción ADD que calculó $R1$ (en el segundo buffer) está en su etapa WB, y la entrada izquierda del multiplexor se inicializa para pasar el valor exactamente calculado de $R1$ (no el valor leído del banco de registros) como primer operando de la instrucción AND; el resultado de la resta, $R4$, está en el primer buffer. Estos buffers corresponden a las variables ALU_{output} y $ALU_{output1}$ indicadas en las Tablas 4.5 y 4.6.

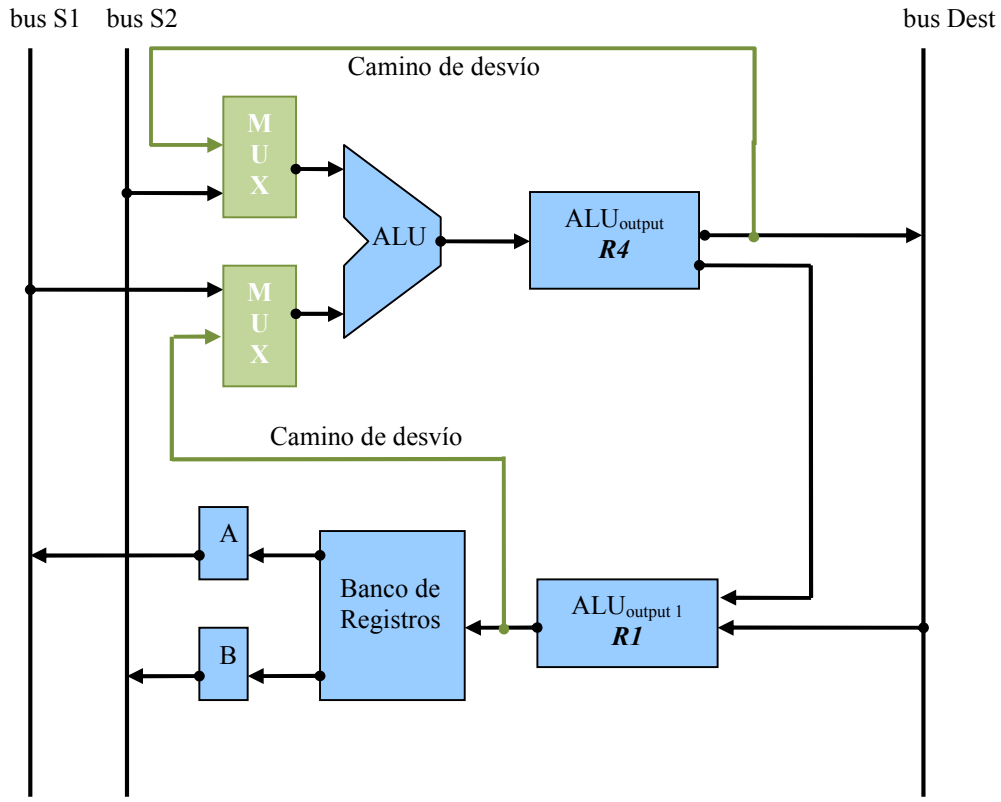


Figura 4.12 ALU con sus caminos de desvío.

Para operaciones de la ALU, el resultado se adelanta siempre que la instrucción que utiliza el resultado como fuente entra en su etapa EX. (La instrucción que calculó el valor que se va a adelantar, puede estar en sus etapas MEM o WB). El resultado del buffers puede ser entrada a uno de los puertos de entrada a la ALU, vía dos multiplexores. El control de los multiplexores puede hacerse de dos formas:

- Por la *Unidad de Control* de la CPU. En cuyo caso, ésta debe rastrear los destinos y fuentes de todas las operaciones en curso.
- Localmente, por la *lógica asociada el desvío*. En cuyo caso, el buffers de desvío contendrá etiquetas que especifique el registro para el que esté destinados el valores de $ALU_{output1}$.

Sea de una forma o de otra, la lógica debe examinar si alguna de las dos instrucciones anteriores escribió en un registro que sea entrada a la instrucción actual. En caso de ser así, el multiplexor selecciona el registro de resultado apropiado, en vez del valor que viene del bus S1 o del bus S2. Una vez se haya implementado el desvío y, puesto que la ALU únicamente opera en una etapa de la segmentación, no hay necesidad de ninguna detención por ninguna combinación de instrucciones de la ALU.

Como se ha comentado, se crea un riesgo siempre que haya una dependencia entre instrucciones y estén tan próximas que el solapamiento causado por la segmentación pueda cambiar el orden para acceder a un operando. Los ejemplos de riesgos por dependencias de datos mostrados anteriormente, han sido con operandos en registros, pero, también es posible crear, para un par de instrucciones, una *dependencia escribiendo y leyendo la misma posición de memoria*. Sin embargo, en la segmentación propuesta para DLX, las referencias a memoria se mantienen siempre en orden, evitando que surja este tipo de riesgos. Los *fallos de la caché* podrían desordenar las referencias a memoria si se permitiera que el procesador continuase trabajando con instrucciones posteriores, mientras que estuviese accediendo a memoria una instrucción anterior que falló en la caché. En el procesador DLX vamos a suponer que se para la segmentación por completo, haciendo que la ejecución de la instrucción que causó el fallo se prolongue durante múltiples ciclos de reloj. Con la *planificación dinámica* (el hardware reorganiza la ejecución de la instrucción para reducir las detenciones), que no estudiaremos en este curso, se permite que cargas y almacenamientos se ejecuten en un orden distinto del que tenían en el programa.

El *adelantamiento* se puede generalizar para que permita el paso de un resultado directamente a la unidad funcional que lo requiera: un resultado que se comunica desde la salida de una unidad a la entrada de otra, en lugar de permitir únicamente desde el resultado de una unidad a la entrada de la misma unidad. Por ejemplo, en la secuencia:

$$ADD \quad R1, R2, R3$$

$$SW \quad 25(R1), R1$$

para prevenir una detención, se necesita adelantar el valor de $R1$ desde la ALU a la ALU, para que se pueda utilizar en el cálculo de la dirección efectiva, y al SMDR (registro de datos de memoria), para que se pueda almacenar sin ciclos de detención.

Los *riesgos por dependencias de datos* pueden clasificarse según tres tipos distintos, dependiendo del orden de los accesos de lectura y escritura en las instrucciones. (Por convenio, los riesgos se definen por el orden del programa). Si se consideran dos instrucciones i y j, presentándose en el programa i antes que j; los posibles tipos de riesgos por dependencias de datos son:

- RAW (*lectura después de escritura – read after write*) → La instrucción j trata de leer una fuente antes de que la escriba la instrucción i; y, de esa forma, la instrucción j tomaría incorrectamente el valor antiguo. Este es el tipo de riesgo más común (el presentado en las **Figuras 4.9** y **4.10**).
- WAR (*escritura después de lectura – write after read*) → La instrucción j intenta escribir un destino antes que sea leído por la instrucción i; de esa forma, la instrucción i tomaría incorrectamente el nuevo valor. Esto no puede ocurrir en la segmentación planteada para el procesador DLX puesto que todas las lecturas se hacen antes (en ID) y todas las escrituras después (en WB). Este riesgo se presenta cuando hay instrucciones que escriben anticipadamente los resultados en el curso de la instrucción, e instrucciones que leen una fuente después que una instrucción posterior haya realizado una escritura. Por ejemplo, autoincrementar en el direccionamiento puede crear un riesgo WAR.
- WAW (*escritura después de escritura – write after write*) → La instrucción j intenta escribir un operando antes que sea escrito por la instrucción i; las escrituras se están realizando en orden incorrecto, dejando en el destino el valor escrito por la instrucción i en lugar del escrito por la instrucción j. Este riesgo se presenta solamente en segmentaciones que escriben en más de una etapa (o permiten que proceda una instrucción aún cuando se encuentre detenida una instrucción anterior). La segmentación propuesta para el procesador DLX solamente escribe un registro en WB, evitando esta clase de riesgos.

Hay que hacer notar que el caso RAR (*lectura después de lectura – read after read*) no es un riesgo.

No todos los riesgos por dependencias de datos se pueden manipular sin que tengan efecto en el rendimiento. Por ejemplo, considérese la siguiente secuencia de instrucciones:

$LW \quad R1,32(R6)$

$ADD \quad R4,R1,R7$

$SUB \quad R5,R1,R8$

$AND \quad R6,R1,R7$

Este caso es diferente al de operaciones consecutivas de la ALU que se ha planteado en los ejemplos anteriores. La instrucción LW no tiene el dato hasta el final del ciclo MEM, mientras que la instrucción ADD necesita el dato al comienzo de ese ciclo de reloj. Por tanto, el riesgo de utilizar el resultado de una instrucción de carga, no se puede eliminar completamente por hardware. Pero, para la instrucción SUB (que comienza dos ciclos después de cuando lo hace LW), sí se puede adelantar el resultado a la ALU directamente desde el LMDR (el resultado llega a tiempo, como muestra la **Figura 4.13**); sin embargo, para la instrucción ADD, el resultado adelantado llega demasiado tarde (al final del ciclo de reloj en el que lo necesita desde el comienzo), teniéndose que provocar por tanto una detención. Si nos referimos a la última instrucción de la secuencia planteada, $AND \quad R6,R1,R7$, puesto que la lectura en la etapa ID se realiza en la segunda mitad del ciclo, siendo escrito el valor en $R1$ en la primera mitad del ciclo correspondiente a WB, no hay ningún problema (en la Figura **4.13** se han marcado en color amarillo las etapas de una y otra instrucción).

Instrucción	Número de ciclo de reloj								
	1	2	3	4	5	6	7	8	9
$LW \quad R1,32(R6)$	IF	ID	EX	MEM	WB				
$ADD \quad R4,R1,R7$		IF	ID	EX	MEM	WB			
$SUB \quad R5,R1,R8$			IF	ID	EX	MEM	WB		
$AND \quad R6,R1,R7$				IF	ID	EX	MEM	WB	

Figura 4.13 Riesgos que se presentan cuando el resultado de una instrucción de carga lo utiliza la siguiente instrucción como operando fuente y se adelanta la operación con él.

Como se ha indicado, la instrucción de carga LW tiene un retardo o latencia que no se puede eliminar sólo por adelantamiento; para poder hacer eso, se requeriría que el tiempo de acceso al dato fuese cero (que es imposible). Para provocar la detención se emplea como solución, añadir un hardware denominado *interbloqueo de la segmentación*. En general un *interbloqueo de la segmentación* detecta un riesgo y

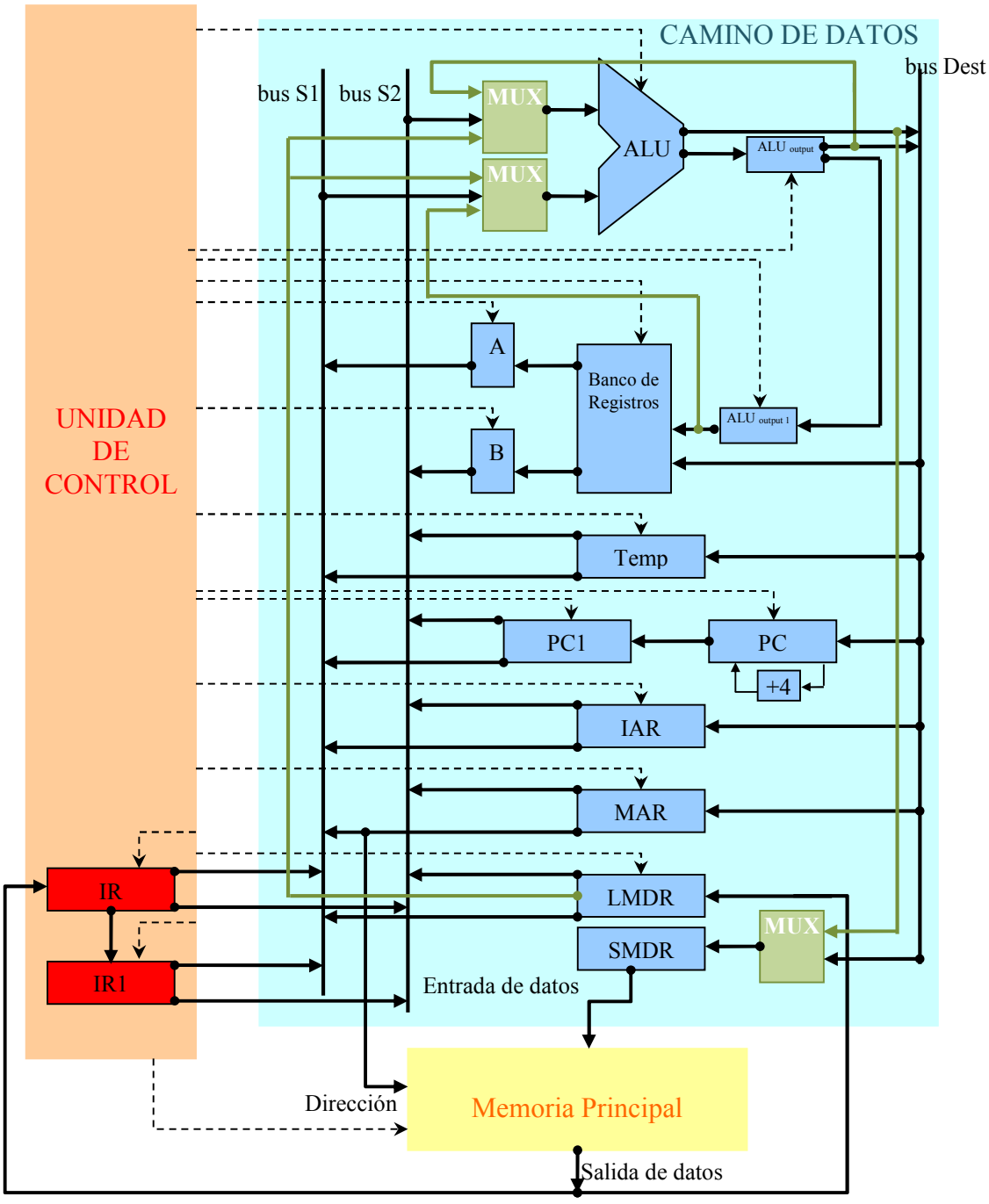
detiene la segmentación hasta que el riesgo desaparece. En este caso, el interbloqueo detiene la segmentación, continuando nuevamente cuando la instrucción que define el dato necesario lo produzca. Este ciclo de retardo, llamado *burbuja* o *detención del cauce* (*pipeline stall* or *bubble*), permite que el dato de carga llegue desde memoria; y a partir de aquí puede ser adelantado por el hardware (con la conexión directa desde LMDR a la ALU). El CPI para la instrucción detenida aumenta según la duración de la detención (un ciclo de reloj en este caso). La segmentación detenida para el ejemplo que se está tratando se muestra en la **Figura 4.14**: todas las instrucciones a partir de la instrucción que tiene la dependencia están retardadas. Con este retardo, el valor de la carga que está disponible al final de la etapa MEM ahora se puede adelantar a la etapa EX de la instrucción ADD. Debido a la detención, la instrucción SUB leerá ahora el valor de los registros durante su etapa ID en lugar de tener que llevarlo directamente desde el registro LMDR.

Instrucción	**Número de ciclo de reloj**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Cualquier inst : IF	ID	EX	MEM	WB					
$LW \quad R1,32(R6)$	IF	ID	EX	MEM	WB				
$ADD \quad R4,R1,R7$		IF	ID	**Det.**	EX	MEM	WB		
$SUB \quad R5,R1,R8$			IF	**Det.**	ID	EX	MEM	WB	
$AND \quad R6,R1,R7$				**Det.**	IF	ID	EX	MEM	WB

Figura 4.14 Efecto de la detención en la segmentación por un riesgo por dependencia de datos.

El proceso de permitir que una instrucción se desplace desde la etapa de decodificación de la instrucción, ID, a la de ejecución, EX, de ese mismo cauce, habitualmente, se denomina *emisión de la instrucción* (*instruction issue*); y una instrucción que haya realizado este paso se dice que ha sido *emitida* (*issued*). Para la segmentación de instrucciones sobre enteros del procesador DLX, todos los riesgos de dependencias de datos se pueden comprobar durante la fase ID; si existe un riesgo, la instrucción es detenida antes que sea emitida. Más adelante, en este capítulo se estudiarán situaciones donde la emisión de instrucciones es mucho más compleja. Detectar con antelación interbloqueos, reduce la complejidad del hardware porque éste no tiene que suspender ninguna instrucción que haya actualizado el estado de la máquina, a menos que la máquina completa esté detenida.

La **Figura 4.15** muestra el esquema contemplando *adelantamiento generalizado*.



Opciones de salida de la ALU:

- S1+S2
- S1-S2
- S1&S2
- S1|S2
- S1^S2
- S1<<S2
- S1>>S2
- $S1>>_a S2$
- S1
- S2
- 0
- 1

IAR → Registro de dirección de interrupción
MAR → Registro de direcciones de memoria
LMDR → Registro de datos de memoria para Carga
SMDR → Registro de datos de memoria para Almacenamiento
IR → Registro de instrucción
IR1 → Registro de instrucción adicional
PC→ Contador de programa
PC1→ Contador de programa adicional
$ALU_{output1}$ → Registro de salida de ALU adicional

Figura 4.15 Arquitectura DLX con adelantamiento generalizado.

Ejemplo 4.3 Suponer que el 20 % de las instrucciones son cargas, y que la mitad del tiempo la instrucción que sigue a una instrucción de carga depende del resultado de la carga. Si este riesgo crea un retardo de un solo ciclo, ¿cuántas veces es más rápida la máquina ideal segmentada (con un $CPI_{medio\ IDEAL}$ de 1) que no retarda la segmentación, si se compara con una segmentación más realista? Ignorar cualquier otro tipo de detenciones.

SOLUCIÓN: La máquina ideal será más rápida según la relación de la aceleración de la máquina ideal sobre la máquina real. Como las frecuencias de reloj no están afectadas, se puede utilizar la expresión (4.4) para la aceleración de la segmentación:

$$Aceleración_{Segm.} = \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}}$$

Como la máquina ideal no tiene detenciones, la aceleración de la segmentación es sencillamente:

$$Aceleración_{\substack{Segm. \\ SIN\ RIESGOS}} = \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}} = \frac{1 \cdot Profundidad\ SEGM.}{1+0}$$

La aceleración de la segmentación en la máquina real es:

$$Aceleración_{\substack{Segm. \\ CON\ RIESGOS}} = \frac{1 \cdot Profundidad\ SEGM.}{1+0.2 \cdot 0.5 \cdot 1} = \frac{1 \cdot Profundidad\ SEGM.}{1.1}$$

Por lo tanto, la máquina ideal sin riesgos estructurales es más rápida que la que tiene riesgos estructurales el siguiente número de veces:

$$\frac{Aceleración_{\substack{Segm. \\ SIN\ RIESGOS}}}{Aceleración_{\substack{Segm. \\ CON\ RIESGOS}}} = \frac{\dfrac{1 \cdot Profundidad\ SEGM.}{1}}{\dfrac{1 \cdot Profundidad\ SEGM.}{1.1}} = \frac{1.1}{1} = 1.1$$

Por lo tanto, la máquina sin riesgos estructurales sería el 10 % más rápida.

Muchos tipos de detenciones son bastante frecuentes. Por ejemplo, el patrón normal de generación de código para una sentencia como $A = B + C$:

$LW \quad R1, B$

$LW \quad R2, C$

$ADD \quad R3, R1, R2$

$SW \quad A, R3$

produce una detención para la carga del valor del segundo dato, C. Tal como se muestra en la **Figura 4.16**, la instrucción ADD debe ser detenida para permitir que se complete la carga de C. El almacenamiento SW no requiere que se retarde más debido a que el adelantamiento hardware pasa el resultado de la ALU directamente a SMDR para almacenarlo. Las máquinas donde los operandos pueden provenir de memoria para operaciones aritméticas, es decir, que tienen el modelo de ejecución REG – MEM. (El procesador DLX no lo tiene), necesitarían detener la segmentación en la mitad de la instrucción para esperar a que se completara el acceso a memoria.

Instrucción	Número de ciclo de reloj								
	1	2	3	4	5	6	7	8	9
$LW \quad R1, B$	IF	ID	EX	MEM	WB				
$LW \quad R2, C$		IF	ID	EX	MEM	WB			
$ADD \quad R3, R1, R2$			IF	ID	**Det.**	EX	MEM	WB	
$SW \quad A, R3$				IF	**Det.**	ID	EX	MEM	WB

Figura 4.16 Segmentación y detenciones en el procesador DLX para la secuencia de código correspondiente a la operación $A = B + C$.

En lugar de permitir que se detenga la segmentación, el compilador podría intentar realizar una planificación que evitara estas paradas, reorganizando la secuencia de código para eliminar el riesgo por dependencia de datos que se produce. Por ejemplo, el compilador trataría de evitar la generación de código con una instrucción de carga seguida por un uso inmediato del registro destino de la carga. Esta técnica, denominada *planificación de la segmentación* o *planificación de instrucciones*, se utilizó por primera vez en los años sesenta; llegando a ser de mucho interés en los años ochenta, cuando estuvieron más extendidas las máquinas segmentadas.

Ejemplo 4.4 Generar el código para el procesador DLX que evite detenciones del cauce para la siguiente secuencia de operaciones:

$$a = b + c$$

$$d = e - f$$

Suponer que las cargas tienen una latencia de un ciclo de reloj.

SOLUCIÓN: Tal como se muestra en la **Figura 4.16**, si se sigue ese patrón de acceso, existirían dos interbloqueos: entre las instrucciones $LW \quad Rc, c$ y $ADD \quad Ra, Rb, Rc$, y entre $LW \quad Rf, f$ y $SUB \quad Rd, Re, Rf$. El código planificado para eliminar esos

problemas se muestra más adelante. En dicho código, esos dos interbloqueos se han eliminado. Aunque en el código propuesto, hay una dependencia entre la instrucción de la ALU $SUB \quad Rd,Re,Rf$ y la de almacenamiento $SW \quad d,Rd$, como se ha comentado anteriormente, la estructura de la segmentación propuesta para DLX permite que se adelante el resultado (desde la salida de la ALU directamente a SMDR).

Código Planificado para DLX:

$LW \quad Rb,b$

$LW \quad Rc,c$

$LW \quad Re,e$; Intercambiada con la siguiente instrucción para evitar la detención

$ADD \quad Ra,Rb,Rc$; en ADD

$LW \quad Rf,f$; Intercambiada con la siguiente instrucción para evitar la detención

$SW \quad a,Ra$; en SUB

$SUB \quad Rd,Re,Rf$

$SW \quad d,Rd$

Hay que hacer notar que, el uso de registros diferentes para las operaciones primera y segunda ha sido crítico para que esta planificación sea posible. En concreto, si se cargase la variable e en los mismos registros que b o c, la planificación no sería posible. En general, la planificación de la segmentación, puede incrementar el número de registros requeridos; y, para las máquinas que pueden emitir múltiples instrucciones en un ciclo, este incremento es sustancial.

La técnica que se ha empleado para evitar detenciones, funciona tan bien que algunas máquinas dejan en manos del software evitar este tipo de riesgos. Una carga que no se necesite para la instrucción inmediatamente siguiente se puede denominar *carga anticipada* (*forwarded load*). Cuando el compilador no puede planificar el interbloqueo, se puede insertar una instrucción de no operación; pero, además de no mejorar el rendimiento (se sigue teniendo necesitando el mismo tiempo que con el interbloqueo), incrementa el espacio de código con respecto a una máquina con interbloqueo.

Tanto si el hardware detecta este interbloqueo y detiene la segmentación como si no (en cuyo caso habría que incluir instrucciones de no operación), el rendimiento mejorará si el compilador planifica las instrucciones.

IMPLEMENTACIÓN DE LA DETECCIÓN DE RIESGOS POR DEPENDENCIAS DE DATOS EN SEGMENTACIONES SIMPLES

La forma de cómo implementar los interbloqueos de la segmentación depende mucho de la longitud y complejidad de la segmentación. Para la segmentación de instrucciones sobre enteros propuesta para el procesador DLX, el único interbloqueo que se necesita aplicar es a una carga (*load*) seguida de una instrucción que tiene que hacer uso de forma inmediata de ella. Esto puede hacerse con un simple comparador que busque este patrón de fuente y destino de la carga. El hardware requerido para detectar y controlar los riesgos por dependencias de datos de la carga y adelantar el resultado de la carga consiste en:

- Multiplexores adicionales en las entradas a la ALU (ya se incluyeron para el hardware de desvío para las instrucciones REG. - REG.).
- Caminos extras desde el MDR a las entradas de los multiplexores a la ALU.
- Un buffer para guardar los números del registro destino de las dos instrucciones anteriores (igual que para el adelantamiento de la instrucciones REG. – REG.)
- Cuatro comparadores para comparar los dos posibles campos del registro fuente con los campos destino de las instrucciones anteriores, y así poder encontrar coincidencias. Estos comparadores comprueban un interbloqueo de carga al principio del ciclo EX. Las cuatro posibilidades y las acciones requeridas se muestran en la Tabla 4.7.

Para el procesador DLX, la detección de riesgos y el hardware de adelantamiento son razonablemente simples, pero para procesadores segmentados con un mayor grado de segmentación son mucho más complicados.

Situación	**Ejemplos de secuencias de códigos**	**Acción a realizar**
No dependencia	$LW\ R1,45(R2)$ $ADD\ R5,R6,R7$ $SUB\ R8,R6,R7$ $OR\ R9,R6,R7$	No hay riesgo porque no existe dependencia sobre $R1$ en las tres instrucciones que están inmediatamente a continuación
Dependencia que requiere detención	$LW\ R1,45(R2)$ $ADD\ R5,R1,R7$ $SUB\ R8,R6,R7$ $OR\ R9,R6,R7$	Los comparadores detectan el uso de $R1$ en la instrucción ADD, y detienen ADD (y por ende SUB y OR también); antes de que ADD comience la etapa EX

Dependencia resuelta con adelantamiento	$LW\ R1,45(R2)$ $ADD\ R5,R6,R7$ $SUB\ R8,R1,R7$ $OR\ R9,R6,R7$	Los comparadores detectan el uso de $R1$ en SUB y adelantan el resultado de la carga a la ALU en el instante en que SUB comienza EX (gracias al camino de desvío)
Dependencia con accesos en orden	$LW\ R1,45(R2)$ $ADD\ R5,R6,R7$ $SUB\ R8,R6,R7$ $OR\ R9,R1,R7$	No se requiere acción alguna porque la lectura de $R1$ en la instrucción OR se presenta en la segunda mitad de la etapa ID de dicha instrucción, mientras que la escritura del dato cargado se presentó en la primera mitad de la fase WB de la instrucción LW (ver Figura 4.10)

Tabla 4.7 Situaciones que el hardware de detección de riesgos de la segmentación detecta comparando los destinos y fuentes de instrucciones adyacentes.

4.4.3 Riesgos de control

Los riesgos de control pueden provocar mayor pérdida de rendimiento para la segmentación del procesador DLX que los riesgos por dependencias de datos.

Cuando se ejecuta un salto (condicional), puede cambiar o no cambiar el contenido de PC; si lo cambia se dice que el salto es efectivo. Si la instrucción i es un salto efectivo, como el PC no cambia hasta el final de la etapa MEM (después de que se complete el cálculo de la dirección y la comparación), como muestra la Tabla 4.6; lo que significa detención durante tres ciclos de reloj, al final de los cuales el nuevo PC es conocido y se puede buscar la instrucción adecuada. Este efecto se denomina *riesgo de salto* o *riesgo de control*. La **Figura 4.17** muestra una detención de tres ciclos para un riesgo de control.

El esquema presentado en la **Figura 4.17** no es realmente así puesto que no se sabe si la instrucción es un salto o no hasta que no se termina de decodificar (etapa ID) y ya se comenzó la búsqueda de la anterior instrucción durante la etapa ID de la instrucción de salto. La **Figura 4.18** muestra lo que realmente ocurre en el procesador DLX según la situación planteada; simplemente se vuelve a realizar una nueva etapa de búsqueda, IF; se busca la instrucción $i+1$, aunque se ignora y se vuelve a comenzar la búsqueda una vez que se conoce el destino del salto. Es obvio que si el salto no es efectivo, es redundante la segunda búsqueda IF; esto se tratará en breve.

Instrucción	**Número de ciclo de reloj**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción salto	IF	ID	EX	MEM	WB					
Instrucción $i+1$		**Det.**	**Det.**	**Det.**	IF	ID	EX	MEM	WB	

Instrucción $i+2$	**Det.**	**Det.**	**Det.**	IF	ID	EX	MEM	WB
Instrucción $i+3$		**Det.**	**Det.**	**Det.**	IF	ID	EX	MEM
Instrucción $i+4$			**Det.**	**Det.**	**Det.**	IF	ID	EX
Instrucción $i+5$				**Det.**	**Det.**	**Det.**	IF	ID
Instrucción $i+6$					**Det.**	**Det.**	**Det.**	IF

Figura 4.17 Detención ideal en el procesador DLX después de un riesgo de control.

Instrucción	**Número de ciclo de reloj**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción salto	IF	ID	EX	MEM	WB					
Instrucción $i+1$		IF	**Det.**	**Det.**	IF	ID	EX	MEM	WB	
Instrucción $i+2$			**Det.**	**Det.**	**Det.**	IF	ID	EX	MEM	WB
Instrucción $i+3$				**Det.**	**Det.**	**Det.**	IF	ID	EX	MEM
Instrucción $i+4$					**Det.**	**Det.**	**Det.**	IF	ID	EX
Instrucción $i+5$						**Det.**	**Det.**	**Det.**	IF	ID
Instrucción $i+6$							**Det.**	**Det.**	**Det.**	IF

Figura 4.18 Detención real en el procesador DLX después de un riesgo de control.

De todas formas, emplear tres ciclos de reloj en cada salto es una pérdida significativa. Haciendo los cálculos, con una frecuencia de saltos de un 30 % y un $CPI_{medio\ IDEAL}$ de 1, la máquina con detenciones de salto logra aproximadamente la mitad de la velocidad ideal de la segmentación. Por ello, reducir la penalización de los saltos llega a ser crítico.

El número de ciclos de reloj de una detención de salto puede reducirse con las dos medidas siguientes:

1. Averiguar si el salto es efectivo o no anteriormente en la segmentación.
2. Calcular anteriormente el PC efectivo (dirección destino del salto).

Para optimizar el comportamiento del salto, deben tomarse las dos medidas anteriores; y, ambas medidas se deben tomar lo antes posible en la segmentación.

En el procesador DLX, los saltos ($BEQZ$ y $BNEZ$) requieren únicamente examinar la igualdad a cero; siendo posible completar esta decisión al final de la etapa ID utilizando una lógica especial dedicada a esta comprobación. Para tomar una decisión lo antes posible, ambos valores posibles para el PC se deben calcular cuanto antes. El cálculo de la dirección del destino del salto requiere un sumador separado, que pueda sumar durante la etapa ID. En este caso por tanto, con el sumador separado y con una decisión de salto tomada durante la etapa ID, solamente habría una detención de un ciclo de reloj en los saltos. La Tabla 4.8 muestra la última columna de la Tabla 4.6 con los nuevos eventos que ocurren en cada etapa de la segmentación de una instrucción de salto definida para el procesador DLX con nuevos recursos. Debido a que la suma de la dirección destino del salto (BTA) ocurre durante la etapa ID, ésta operación se llevará a cabo para todas las instrucciones; también la condición de salto (Rs1 op 0) también se hará para todas las instrucciones. La última operación de la etapa ID es sustituir el PC; por lo tanto, se deberá saber que la instrucción es un salto antes de realizar este paso. Esto requiere que la decodificación de la instrucción se realice antes del final de la etapa ID, o hacer esta operación al comienzo de la etapa EX, cuando se envíe el PC. Las etapas EX, MEM y WB no se usan para los saltos.

Si nos referimos a las bifurcaciones, surge una complicación adicional puesto que tienen un desplazamiento mayor que los saltos; esto puede resolverse utilizando un sumador adicional que sume el PC y los 26 bits de menos peso de la instrucción en el registro IR, o también, otra forma podría ser, con un esquema más inteligente, que haga una suma de 16 bits en la primera mitad del ciclo y determine si sumar posteriormente los 10 bits restantes del registro IR en la segunda mitad del ciclo, decodificando antes los códigos de operación de bifurcación.

Etapa	Instrucción de Bifurcación
IF	IR ← M[PC] PC ← PC+4
ID	A ← Rs1; B ← Rs2 BTA ← PC+ $(IR1_{16})^{16}$##$IR1_{16..31}$) If (Rs1 op 0): PC ← BTA
EX	
MEM	
WB	

Tabla 4.8 Eventos en cada etapa de la segmentación de DLX para las instrucciones de bifurcación, con la estructura revisada de la segmentación incluyendo un sumador separado para calcular la dirección de destino del salto.

En algunas máquinas los riesgos de los saltos son aún más caros en ciclos de reloj que en nuestro ejemplo, ya que el tiempo para evaluar la condición de bifurcación y calcular el destino puede ser aún mayor. En general, a mayor profundidad de la segmentación, peor penalización de salto. Por supuesto, el efecto en el rendimiento relativo de una mayor penalización de saltos depende del CPI_{GLOBAL} de la máquina; una máquina con CPI_{GLOBAL} alto puede soportar penalizaciones por saltos más caras porque el porcentaje del rendimiento de la máquina que se perdería por los saltos sería menor.

En lo que respecta al comportamiento dinámico de los saltos hay que decir que, para el procesador DLX, los estudios sobre dicho comportamiento, arrojan que el 11 % de las instrucciones son saltos condicionales y el 2 % son incondicionales, el 53 % de los saltos condicionales son efectivos y, el 75 % de los saltos ejecutados son saltos hacia delante.

REDUCCIÓN DE LA PENALIZACIÓN DE LOS SALTOS EN LA SEGMENTACIÓN

Existen varios métodos para tratar con las detenciones de la segmentación debidas a los saltos; se explican a continuación cuatro sencillos esquemas en tiempo de compilación, predicciones estáticas (son fijas para cada salto durante la ejecución completa). Las predicciones dinámicas para los saltos se tratan al final de este apartado.

Instrucción	Número de ciclo de reloj									
	1	2	3	4	5	6	7	8	9	10
Instrucción salto no efectivo	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 salto efectivo	IF	ID	EX	MEM	WB					
Instrucción $i+1$		IF	IF	ID	EX	MEM	WB			
Instrucción $i+2$			**Det.**	IF	ID	EX	MEM	WB		
Instrucción $i+3$				**Det.**	IF	ID	EX	MEM	WB	
Instrucción $i+4$					**Det.**	IF	ID	EX	MEM	WB

Figura 4.19 Esquema de predecir-no-efectivo y la secuencia de la segmentación cuando no es efectivo el salto (en la parte superior) y cuando es efectivo (en la parte inferior).

Los cuatro métodos que se plantean son:

1. *Congelar la segmentación*, reteniendo todas las instrucciones que están a continuación del salto hasta que se conozca el destino del salto → La ventaja principal de esta solución es su simplicidad. Esta es la solución que se ha planteado en las Figuras 4.17 y 4.18.

2. *Predecir el salto como no efectivo*, permitiendo que el hardware continúe como si el salto no se ejecutase → Se trata de un esquema algo mejor que el anterior y sólo ligeramente más complejo. Hay que tener cuidado de no cambiar el estado de la máquina hasta que no se conozca definitivamente el resultado del salto. La dificultad está en saber cuándo una instrucción puede cambiar el estado y en cómo deshacer un cambio; la solución más simple a considerar es la de *limpiar* (*flushing*) *la segmentación*. En la segmentación para DLX, este esquema de *predecir-no-efectivo* (*predict-not-taken*) se implementa continuando la búsqueda de las instrucciones, como si no ocurriese nada extraordinario. Sin embargo, si el salto es

efectivo, se necesita detener la segmentación y recomenzar la búsqueda, la Figura 4.19 muestra ambas situaciones: cuando el salto no es efectivo, que se determina en la etapa ID, ya se ha buscado la siguiente instrucción y, simplemente, se continúa; si el salto es efectivo, determinado en la etapa ID, se recomienza la búsqueda en el destino del salto, haciendo que todas las que sigan a la que está en la dirección de salto se detengan un ciclo de reloj.

3. *Predecir el salto como efectivo* y, una vez que se decodifica la instrucción y se calcula la dirección de destino, se supone que el salto se va a realizar y comienza la búsqueda y ejecución en el destino → Al igual que en la segmentación del procesador DLX, no se conoce la dirección del destino antes de que se conozca el resultado del salto, no habiendo ninguna ventaja con respecto al procedimiento anterior. Sin embargo, en algunas máquinas (especialmente las que tienen códigos de condición o condiciones de salto más potentes, y por consiguiente más lentas) el destino del salto se conoce antes del resultado del salto; teniendo sentido entonces este esquema.

4. *Salto retardado* → Este esquema se ha empleado en muchas unidades de control microprogramadas. En un salto retardado, el ciclo de ejecución con un retardo de salto de longitud n es:

instrucción de salto

sucesor secuencial$_1$
sucesor secuencial$_2$
.....
sucesor secuencial$_n$

destino del salto si efectivo

Los sucesores secuenciales están en lo que se conoce como *huecos de retardo del salto* (*branch-delay slots*). Como con los huecos de retardo de carga, la tarea del software es hacer que las instrucciones sucesoras sean válidas y útiles; normalmente se utilizan una serie de optimizaciones. La Figura 4.20 muestra las tres formas posibles de planificación del retardo de salto; y la Tabla 4.9 muestra las diferentes restricciones para cada uno de estos esquemas de planificación de saltos, así como las situaciones en las que se comportan mejor. El origen de la instrucción que se planifica en el hueco de retardo determina la estrategia de planificación. El compilador debe hacer cumplir los requerimientos cuando busque instrucciones para planificar el retardo. Cuando los huecos no se pueden planificar, se rellenan con instrucciones de no operación. En la estrategia (b), si el destino del salto también es accesible desde otro punto del programa (como ocurriría si fuese

la cabeza de un bucle) las instrucciones del destino deben ser copiadas y no transferidas.

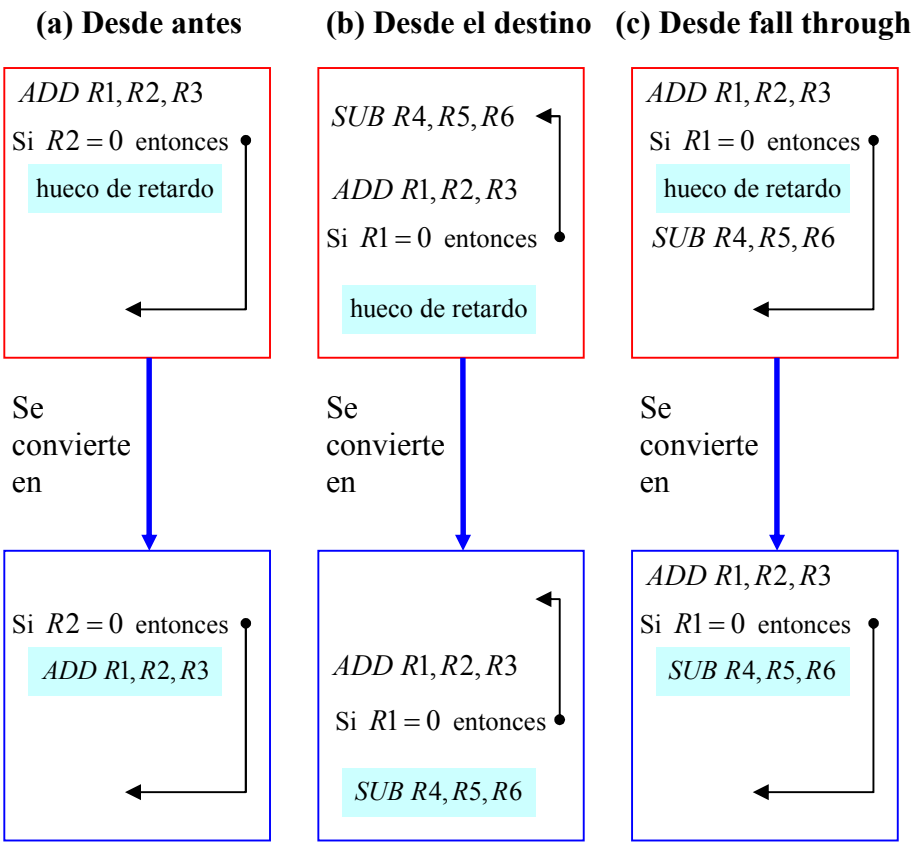


Figura 4.20 Planificación del hueco de retardo de los saltos. El cuadro superior de cada pareja muestra el código antes de la planificación y el cuadro inferior muestra el código planificado. En **(a)** el hueco de retardo se planifica con una instrucción independiente anterior al salto. Esta es la mejor elección. Las estrategias **(b)** y **(c)** se utilizan cuando no es posible la estrategia **(a)**. En las secuencias de código para **(b)** y **(c)**, el uso de $R1$ en la condición de salto impide que la instrucción ADD (cuyo destino es $R1$) sea transferida después del salto. En **(b)** el hueco de retardo de salto se planifica con el destino del salto; habitualmente, la instrucción del destino necesitará ser copiada porque puede ser alcanzada por otro camino. La estrategia **(b)** se prefiere cuando el salto se realiza con alta probabilidad, como, por ejemplo, un salto por bucle. Finalmente, en **(c)**, el salto puede ser planificado con la instrucción siguiente, secuencialmente. Para hacer esta optimización legal para **(b)** o **(c)**, debe ser <<OK>> ejecutar la instrucción SUB cuando el salto vaya en la dirección no esperada. Por <<OK>> se entiende que el trabajo se desperdicia, pero el programa todavía se ejecutará correctamente. Este es el caso, por ejemplo, si $R4$ fuese un registro temporal no utilizado cuando el salto vaya en la dirección no esperada.

Estrategia de planificación	**Requerimientos**	**¿Cuándo mejora el rendimiento?**
(a) Desde antes del salto	Los saltos no deben depender de las instrucciones replanificadas	Siempre
(b) Desde el destino	Debe ser <<OK>> ejecutar las instrucciones replanificadas si no es efectivo el salto. Puede ser necesario duplicar instrucciones	Cuando es efectivo el salto. Puede alargar el programa si las instrucciones están duplicadas
(c) Desde instrucciones siguientes	Debe ser <<OK>> ejecutar las instrucciones si es efectivo el salto	Cuando el salto no es efectivo

Tabla 4.9 Esquemas de planificación de salto retardado y sus requerimientos.

Las limitaciones principales en la planificación de saltos retardados surgen de las restricciones sobre las instrucciones que se planifican en los huecos de retardo y de la posibilidad de predecir en tiempo de compilación la probabilidad de que un salto va a ser efectivo.

Hay un pequeño coste adicional de hardware para los saltos retardados. Debido al efecto retardado de los saltos, se necesitan múltiples PC (uno más que la duración del retardo) para restaurar correctamente el estado cuando se presente una interrupción. Considerar que la interrupción se presenta después de que se completa una instrucción de salto efectivo, pero antes de que se completen todas las instrucciones de los huecos de retardo y del destino del salto. En este caso, los PC de los huecos de retardo y el PC del destino del salto se deben guardar, ya que no son secuenciales.

Ahora, es el momento de preguntarse: ¿cuál es el rendimiento efectivo de cada uno de los cuatro métodos para tratar las detenciones de la segmentación de los saltos en tiempo de compilación?

La aceleración efectiva de la segmentación con penalizaciones de salto es:

$$Aceleración_{Segm.} = \frac{CPI_{medio\ IDEAL} \cdot Profundidad\ SEGM.}{CPI_{medio\ IDEAL} + C_{\substack{detención \\ SEGM.}} PI_{medio}}$$

Si suponemos que el $CPI_{medio\ IDEAL} = 1$, entonces se puede simplificar a la expresión:

$$Aceleración_{Segm.} = \frac{Profundidad\ SEGM.}{1 + C_{\substack{detención \\ SEGM.}} PI_{medio}}$$

y, puesto que $C_{\substack{detención \\ SEGM.}} PI_{medio} = Frecuencia\ de\ saltos \cdot Penalización\ de\ salto$ se obtiene la siguiente expresión:

$$Aceleración_{Segm.} = \frac{Profundidad\ SEGM.}{\left(1 + Frecuencia\ de\ saltos \cdot Penalización\ de\ salto\right)} \qquad (4.5)$$

Utilizando las medidas dadas para el procesador DLX en este apartado y la expresión (4.5), se muestran en la Tabla 4.10 las distintas aceleraciones según los distintos métodos planteados para los saltos (suponiendo un $CPI_{medio\ IDEAL} = 1$).

Esquema de planificación	**Penalización de salto**	**CPI efectivo**	**Aceleración de la máquina segmentada respecto a la no segmentada**	**Aceleración respecto a la estrategia de detenerse en los saltos**
Detención	3	1.42	3.5	1.0
Predicción "efectivo"	1	1.14	4.4	1.26
Predicción "no efectivo"	1	1.09	4.5	1.29
Salto retardado	0.5	1.07	4.6	1.31

Tabla 4.10 Costes globales de una serie de esquemas de salto con la segmentación para el procesador DLX.

Hay que hacer notar que los datos numéricos que se han manejado en este apartado están muy afectados por la duración del retardo de salto y por el $CPI_{medio\ IDEAL}$ que se tome como base. Un retardo de salto mayor, provocará un incremento en la penalización y un porcentaje de tiempo desperdiciado mayor (considerar un retardo de un solo ciclo de reloj es pequeño). Con un $CPI_{medio\ IDEAL}$ bajo el retardo debe mantenerse pequeño, mientras que un $CPI_{medio\ IDEAL}$ mayor reduciría la penalización relativa de los saltos.

REDUCCIÓN DE LAS PENALIZACIONES DE LOS SALTOS CON PREDICCIÓN DINÁMICA HARDWARE

Antes se han examinado esquemas de hardware sencillos para tratar los saltos (suponiéndolos efectivos o no efectivos), y enfoques orientados al software (saltos retardados). Ahora nos centramos en utilizar el hardware para predecir dinámicamente el resultado de un salto (la predicción cambiará si los saltos cambian su comportamiento mientras se está ejecutando el programa).

El esquema dinámico de predicción de saltos más simple es un *buffer de predicción de saltos*. Se trata de una pequeña memoria indexada por la parte menos significativa de la dirección de la instrucción de salto. La memoria contiene un bit que indica si recientemente el salto fue efectivo o no. Este es el tipo más sencillo de buffer, no tiene etiquetas y es útil sólo para reducir el retardo de salto cuando es mayor que el tiempo para calcular los posibles PC del destino. De hecho, no se sabe si la predicción es correcta (puede haber sido puesto allí por otro salto que tenga iguales bits menos significativos de la dirección). Pero esto no importa; se supone que es correcta la predicción, y comienza la búsqueda en la dirección predicha. Si la predicción de salto resulta ser errónea, se invierte el bit de predicción.

Este esquema de predicción, de un solo bit, tiene un problema para el rendimiento: si un salto es efectivo casi siempre, cuando no lo sea, se predicirá incorrectamente dos veces, en lugar de una. Por ejemplo, si se considera un salto en un bucle cuyo comportamiento es: efectivo nueve veces seguidas, y no efectivo una vez; si la siguiente vez a la de no ser efectivo se predice como no efectivo, la predicción será errónea. Entonces, la precisión de la predicción será únicamente del 80 % aún en saltos el 90 % de efectivos. Para remediar esto, se utilizan con frecuencia esquemas de predicción de dos bits: una predicción debe errar dos veces antes de que se cambie. La **Figura 4.21** muestra la máquina de estados finitos para el esquema de predicción de dos bits. Un salto que la mayoría de las veces sea efectivo (o no efectivo), como ocurre con muchos saltos, estará mal predicho sólo una vez.

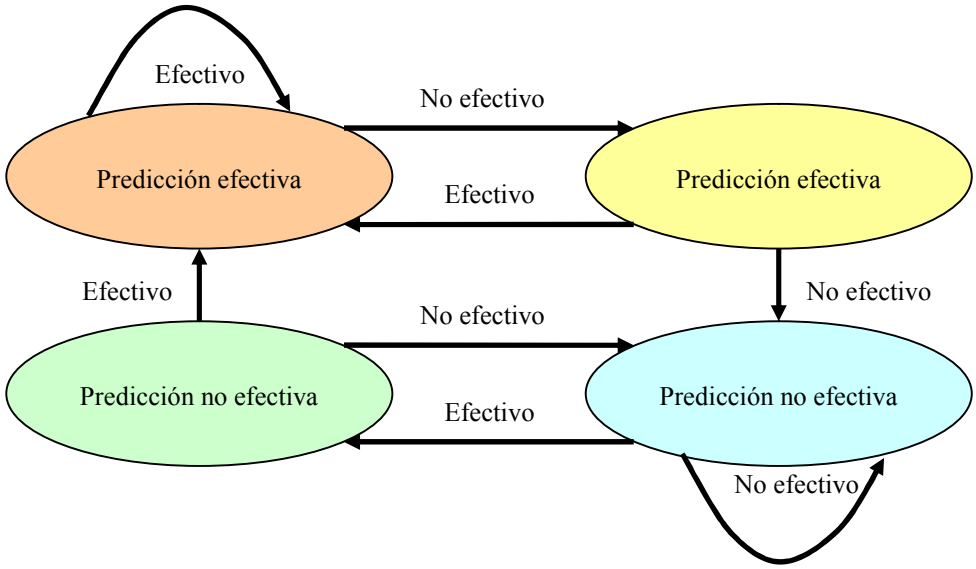


Figura 4.21 Esquema de predicción con dos bits de estado.

El buffer de predicción de saltos se puede implementar como una pequeña caché especial accedida por la dirección de la instrucción durante la etapa IF de la segmentación, o como un par de bits vinculados a cada bloque de la caché de instrucciones y buscados con la instrucción. Si la instrucción se predice como un salto y si el salto se predice como efectivo, la búsqueda del destino comienza tan pronto como se conozca el PC. En cualquier otro caso, continúa la búsqueda y la ejecución secuencial. Si la predicción resulta ser errónea, se cambian los bits de predicción, como muestra la **Figura 4.21**. Aunque este esquema es útil para muchas segmentaciones, la segmentación de DLX averigua al mismo tiempo si el salto es efectivo y el destino del salto. Por tanto, este esquema no ayuda a la sencilla segmentación de DLX; más adelante se explora un esquema que puede ser útil para DLX. En primer lugar se va a ver cómo funciona un buffer de predicción con una segmentación más larga.

La precisión de un esquema de dos bits está afectada por el número de veces que la predicción es correcta para cada salto, y por el número de veces que la entrada en el buffer de predicción coincide con el salto que se está ejecutando. Cuando la entrada no coincide, el bit de predicción se utiliza de todas formas porque no hay otra información mejor disponible; aún cuando la entrada fuese para otro salto, la suposición podría haber tenido éxito (aproximadamente hay un 50 % de probabilidad de que la predicción sea correcta). Estudios que se han realizado sobre esquemas de predicción de salto, han encontrado que la predicción de dos bits tiene una precisión de aproximadamente el 90 % cuando la entrada del buffer es la entrada del salto. Un

buffer de entre 500 y 1000 entradas tiene una frecuencia de aciertos del 90 %. La predicción global está dada por:

$$\begin{aligned} Precisión = \quad & (\%\ correctamente\ predicho \cdot \%\ predicción\ es\ para\ esta\ instrucción) + \\ & + (\%\ conjetura\ afortunada) \cdot (1 - \%\ predicción\ es\ para\ esta\ instrucción) \end{aligned} \tag{4.6}$$

Si se aplica (4.6) al caso que se está planteando, se obtiene:

$$Precisión = \quad (90\,\% \cdot 90\,\%\ ) + (50\,\% \cdot (1 - 90\,\%)) = 86\,\%$$

que da una frecuencia de aciertos mayor que la correspondiente a los saltos retardados, siendo útil en una segmentación con un retardo de salto mayor. A continuación se va a ver un esquema de predicción dinámica que es utilizable para DLX, y veremos la comparación con el esquema de retardo de salto.

Para reducir la penalización de saltos en DLX se necesita conocer qué dirección buscar al final de IF. Esto significa que si la instrucción todavía no decodificada es un salto y, si lo es, cuál será el siguiente PC. Si la instrucción es un salto y se sabe cuál es el siguiente PC, se tiene una penalización de salto de cero. Una caché de predicción de saltos que almacena la dirección predicha de la siguiente instrucción después del salto se denomina *buffer de destino de salto* (*branch-target buffer*). Como se está prediciendo la dirección de la siguiente instrucción y se envía antes de decodificar la instrucción, se debe saber si la instrucción buscada se predice como un salto efectivo. También se quiere saber si la predicción del buffer de destinos es para una predicción de salto efectivo o no efectivo, para que de esa forma se pueda reducir el tiempo para determinar un salto mal predicho. La **Figura 4.22** muestra el aspecto del buffer de destinos de saltos. Si el PC de la instrucción buscada coincide con un PC en el buffer, entonces el correspondiente PC predicho se utiliza como PC siguiente.

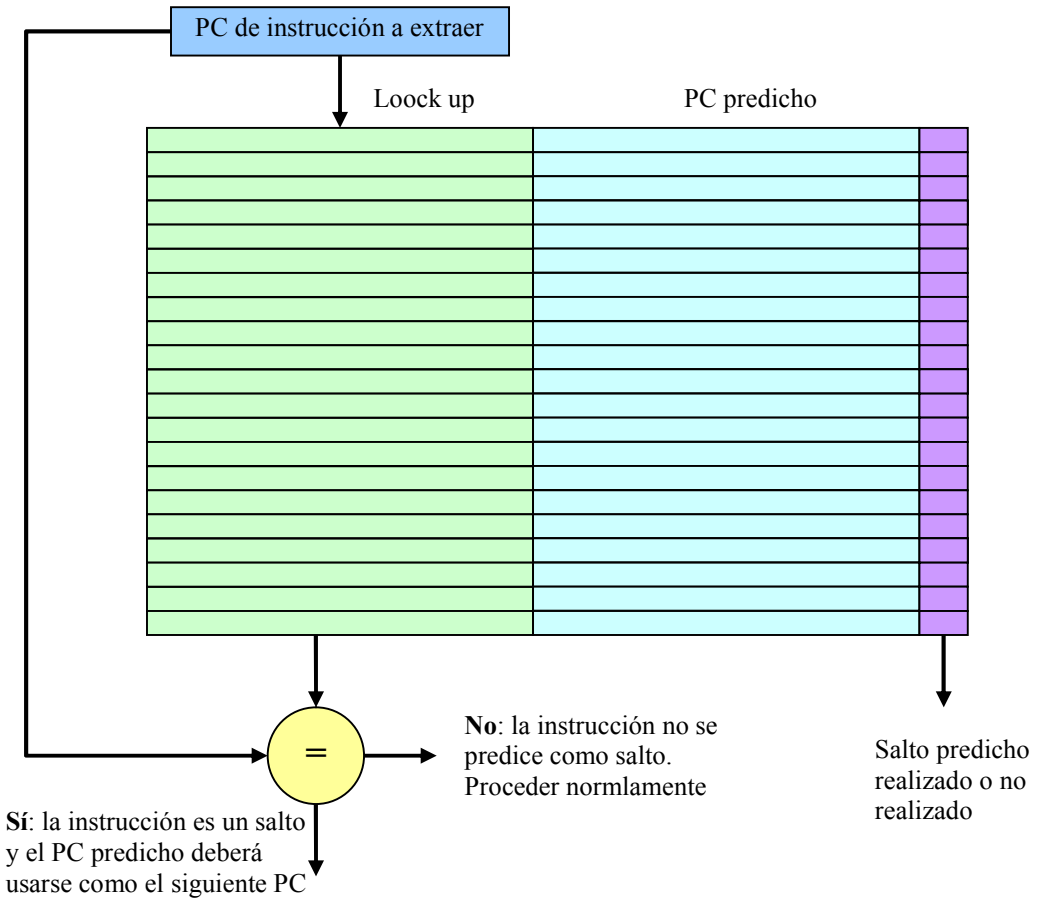


Figura 4.22 Buffer de destinos de saltos. El PC de la instrucción que se está buscando se compara con un conjunto de direcciones de instrucciones almacenadas en la primera columna; éstas representan direcciones de instrucciones de saltos conocidos. Si el PC coincide con una de estas entradas, entonces la instrucción que se está buscando es un salto. En caso de efectivamente ser un salto, el segundo campo (PC predicho) contiene la dirección para el siguiente PC después del salto. La búsqueda comienza inmediatamente en esa dirección. El tercer campo, informa exactamente de si el salto se predijo como efectivo o no efectivo y ayuda a mantener pequeña la penalización en caso de mala predicción.

Como se indica en la **Figura 4.22**, si en el buffer de destinos de saltos se encuentra una entrada que coincide, la búsqueda comienza inmediatamente en el PC predicho. Se observa que, de forma distinta a un buffer de predicción de saltos, la entrada debe ser para esta instrucción, porque el PC predicho se enviará fuera, incluso antes de que conozca si esta instrucción es un salto o no. Si no se comprobara si la entrada coincide con ese PC, el PC erróneo podría utilizarse por instrucciones que no fuesen saltos, en cuyo caso se obtendría una máquina más lenta. La **Figura 4.23** muestra los pasos seguidos cuando se utiliza un buffer de destinos de saltos, así como cuándo se presentan estos pasos en la segmentación DLX.

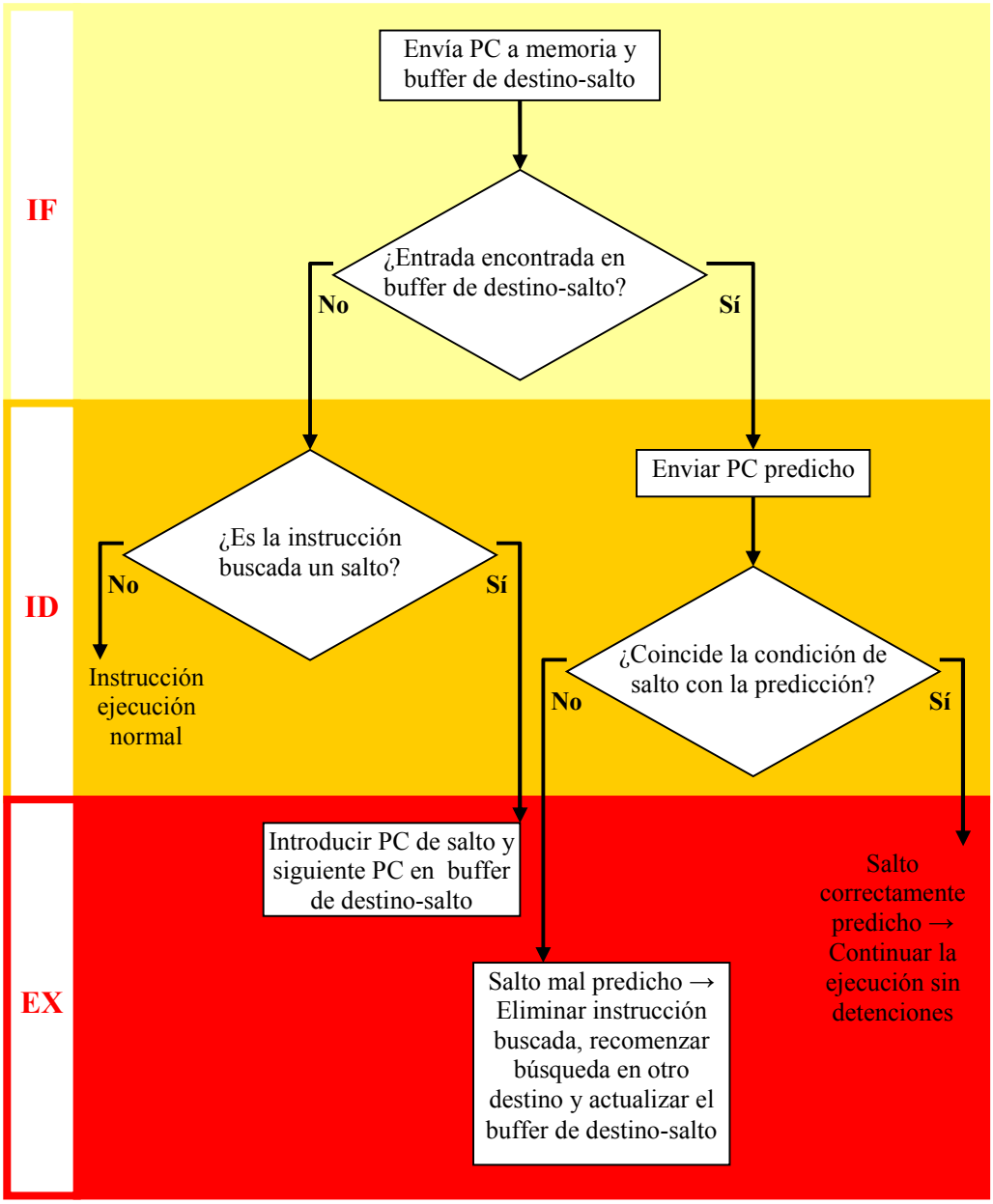


Figura 4.23 Los pasos involucrados en la manipulación de una instrucción con un buffer de destino-salto.

Según la **Figura 4.23**, si el PC de una instrucción se encuentra en el buffer, entonces la instrucción debe ser un salto, y la búsqueda comienza inmediatamente desde el PC predicho en la etapa ID. Si no se encuentra la entrada y posteriormente resulta ser un salto, se incluye en el buffer junto al destino, que es conocido al final de la etapa ID. Si la instrucción es un salto, se encuentra y se predice correctamente, entonces la ejecución procede sin retardos. Si la predicción es incorrecta, se sufre un retardo de un ciclo de reloj (buscando la instrucción errónea) y se recomienza la búsqueda un ciclo de reloj más tarde. Si el salto no se encuentra en el buffer y la instrucción resulta ser un salto, también se procede como si la instrucción fuese un salto y se puede

convertir esto en una estrategia de asumir - no efectivo; la penalización diferirá dependiendo de que el salto sea realmente efectivo o no.

Siguiendo esa estrategia, no hay retardo de salto si en el buffer se encuentra una entrada de predicción de salto y es correcta. En cualquier otro caso, hay una penalización de al menos un ciclo de reloj. En la práctica, hay una penalización de dos ciclos de reloj, porque se debe actualizar el buffer de destinos de saltos. También, se podría suponer que la instrucción que sigue a un salto o la del destino del salto no es un salto, y hacer la actualización durante el tiempo de esa instrucción; pero esta opción complicaría el control. Por lo tanto, se penalizarán dos ciclos de reloj cuando el salto no se haya predicho correctamente.

Instrucción en buffer	Predicción	Salto real	Ciclos de penalización
Sí	Efectivo	Efectivo	0
Sí	Efectivo	No efectivo	2
Sí	No efectivo	No efectivo	0
Sí	No efectivo	Efectivo	2
No	-	Efectivo	2
No	-	No efectivo	1

Tabla 4.11 Penalizaciones para todas las posibles combinaciones de si el salto está en el buffer, cómo se predijo, y lo que ocurre realmente.

La evaluación de un buffer de destinos de saltos obliga a que primero se determine qué penalizaciones hay en todas las situaciones posibles; en la Tabla 4.11 se encuentra dicha información: no hay penalización de salto si todo se predice correctamente y el salto se encuentra en el buffer de destino de saltos. Si el salto no se predice correctamente, la penalización es de un ciclo de reloj para actualizar el buffer con la información correcta (durante la cual no se puede buscar ninguna instrucción), y otro ciclo de reloj, si se necesita, para recomenzar la búsqueda de la siguiente instrucción correcta para el salto. Es decir, en caso de no haberse encontrado el salto en el buffer de destino y no ser efectivo, como se ha supuesto que se emplea una estrategia de asumir no – efectivo (cuando aún no estaba enterado el procesador de que la instrucción era un salto), la penalización es entonces de un sólo ciclo de reloj; otros fallos de coincidencia cuestan dos ciclos de reloj, puesto que se tiene que recomenzar la búsqueda y actualizar el buffer.

Si para evaluar el buffer de destinos de saltos se utilizan las mismas probabilidades que se usaron para un buffer de predicción de salto (90 % de probabilidad de encontra la entrada y 90 % de probabilidad de predicción correcta) y el porcentaje de efectivos/no efectivos tomado anteriormente (90/10), la predicción total del salto sería:

$$\begin{aligned} Penalización\ del\ salto_{medio} = \ & \%\ saltos\ encontrados\ en\ el\ buffer \cdot \%\ predicciones\ incorrectas \cdot 2 + \\ & + \left(1 - \%\ saltos\ encontrados\ en\ el\ buffer\right) \cdot \%\ saltos\ realizados \cdot 2 + \\ & + \left(1 - \%\ saltos\ encontrados\ en\ el\ buffer\right) \cdot \%\ saltos\ no\ realizados \cdot 1 \end{aligned}$$

Por lo tanto:

$$Penalización\ del\ salto_{medio} = 90\% \cdot 10\% \cdot 2 + 10\% \cdot 40\% \cdot 1 = 0.34\ ciclos\ de\ reloj$$

Comparando con la penalización para los saltos retardados, que es aproximadamente de $0.5\ ciclos\ de\ reloj$ por salto, se ve que hay una mejora y, como ya sabemos, la mejora de la predicción dinámica de saltos crece además al aumentar el retardo de los saltos.

Los esquemas de predicción de saltos están limitados por:

1. la precisión de la predicción y,
2. por la penalización de las predicciones erróneas.

Mejorar la precisión de la predicción de saltos por encima del 80 % es improbable. Sí se puede tratar de reducir la penalización para las predicciones erróneas; una forma es buscando tanto la dirección predicha como la no predicha, utilizando un sistema de memoria con dos puertos que tenga una caché intercalada. Aunque esto añade coste al sistema, puede ser la única forma de reducir las penalizaciones de los saltos por debajo de un cierto punto.

4.4.4 Las dificultades de la implementación de la segmentación

En este apartado se van a tratar algunas complicaciones que existen en la segmentación y que hasta ahora no se ha hablado de ellas. Si ya sabemos que las interrupciones complican el diseño de una máquina secuencial, cuando es segmentada las complicaciones son aún mayores.

COMPLICACIONES DE LAS INTERRUPCIONES

Las interrupciones son más difíciles de manejar en una máquina segmentada porque, al existir solapamiento en las instrucciones, se hace más difícil saber si una instrucción puede cambiar sin peligro el estado de la máquina.

Al igual que en las máquinas no segmentadas, las interrupciones hacen más difícil la implementación de la máquina segmentada por las siguientes características:

1. Se presentan mientras se están ejecutando las instrucciones (varias instrucciones a la vez).
2. Deben ser recomenzables.

Por ejemplo, en el procesador DLX, un fallo de página de memoria virtual no puede presentarse hasta la etapa MEM de la instrucción. Cuando se detecte ese fallo, se estarán ejecutando otras instrucciones. Como un fallo de página debe ser recomenzable y requiere la intervención de otro proceso (del sistema operativo), se deben detener las instrucciones en curso y guardar el estado para que las instrucciones se puedan reiniciar en su estado correcto. Cuando se presenta una interrupción de este tipo, se pueden realizar los siguientes pasos para guardar con seguridad el estado del procesador:

1. Forzar una instrucción de trap en la siguiente etapa IF.
2. Hasta que el trap sea efectivo, eliminar todas las escrituras para la instrucción que causó el fallo y para las siguientes. Esto previene cualquier cambio de estado para las instrucciones que no se hayan completado antes que sea tratada la interrupción.
3. Después de que la rutina de tratamiento de interrupciones del sistema operativo reciba el control, se guarda inmediatamente el PC de la instrucción que causó el fallo. Este valor se utilizará al retorno de la interrupción.

Si en una máquina se utilizan saltos retardados, no es posible restablecer el estado de la máquina con un solo PC, porque las instrucciones en curso pueden no estar relacionadas secuencialmente. En particular, cuando la instrucción que provoca la interrupción es un hueco de retardo de salto, y el salto fue efectivo, las instrucciones para recomenzar son las del hueco más la instrucción del destino del salto; el salto ha completado la ejecución y no se reinicia (las direcciones de las instrucciones del hueco del retardo del salto y del destino no son secuenciales, por ello será necesario guardar y restaurar un número de PC igual a la longitud del retardo de salto más uno → esto se hace en el tercero de los pasos anteriores).

Una vez se haya tratado la interrupción, instrucciones especiales devuelven a la máquina al estado anterior a la interrupción (la instrucción RFE en el procesador DLX). Si la segmentación se puede parar para que se completen las instrucciones anteriores a la del fallo y las posteriores se pueden reiniciar desde el principio, se dice que el procesador segmentado tiene *interrupciones precisas*.

El que un sistema soporte interrupciones precisas algunas veces es obligado y otras se considera algo valioso por simplificar la interfaz del sistema operativo. Las interrupciones precisas se pueden resolver con hardware o con software.

COMPLICACIONES DEL REPERTORIO DE INSTRUCCIONES

Otra serie de dificultades surge de bits de estado singulares, que pueden crear riesgos adicionales en la segmentación o pueden necesitar hardware extra para guardar y restaurar. Los códigos de condición son un buen ejemplo. Muchas máquinas modifican implícitamente los códigos de condición como parte de la instrucción; que, aunque inicialmente esto parece una buena idea porque los códigos de condición separan la evaluación de la condición del salto real, implícitamente pueden provocar dificultades para hacer los saltos más rápidos. Es decir, limitan la efectividad de la planificación de los saltos porque la mayoría de las operaciones modificarán el código de condición, haciendo difícil planificar instrucciones entre la inicilización del código de condición y el salto.

Un área espinosa final de la segmentación encauzada es la correspondiente a operaciones multiciclo. Cuando las instrucciones del repertorio de un sistema computador difieren enormemente en el número de ciclos que necesitan (podrían diferir desde uno hasta varios centenares) o incluso en el número de accesos a memoria que necesita (desde cero a incluso cientos de accesos a memoria), los riesgos por dependencias de datos son muy complejos y se presentan entre y en las instrucciones. Para este caso, la solución simple de hacer que todas las instrucciones se ejecuten durante el mismo número de ciclos de reloj es inaceptable, porque introduce un número enorme de riesgos y de condiciones de desvío, además de hacer una segmentación inmensamente larga. La solución (inicialmente propuesta y aplicada en el VAX 8800) consiste en segmentar a nivel de microinstrucción en lugar de hacerlo a nivel de instrucción; como éstas son sencillas, el control de la segmentación es mucho más fácil. Aunque no esté completamente que esta opción pueda conseguir un CPI tan bajo como con una segmentación a nivel de instrucción, este enfoque es mucho más sencillo y posiblemente obtiene una menor duración del ciclo de reloj.

Las máquinas de carga/almacenamiento que tienen instrucciones que realizan operaciones sencillas y con similares cantidades de trabajo se segmentan más fácilmente. Si los arquitectos de computadores son conscientes de la relación entre el diseño del repertorio de instrucciones y la segmentación, podrán diseñar arquitecturas para segmentaciones más eficientes. En el siguiente apartado se estudia cómo la segmentación DLX trata instrucciones de larga ejecución.

4.5 OPERACIONES MULTICICLO. EXTENSIÓN DE LA SEGMENTACIÓN DLX

En este apartado se va a ver cómo se puede ampliar la segmentación DLX para manipular operaciones de punto flotante, dando un enfoque básico y las alternativas de diseño.

No es práctico exigir que todas las operaciones de punto flotante de DLX se completen en un ciclo de reloj, puesto que, hacer eso significaría o aceptar un reloj lento o utilizar mucho hardware en las unidades de punto flotante, o ambas cosas. En cambio, permitir que se segmente la propia operación de coma flotante, y como consecuencia una mayor latencia para estas operaciones, tiene ventajas.

La idea planteada sobre segmentar la propia operación de coma flotante se entiende mejor si se supone que las instrucciones en punto flotante tienen la misma segmentación que la de las instrucciones de enteros, con dos cambios importantes:

1. La etapa EX se puede repetir tantas veces como sea necesario para completar la operación; el número de repeticiones puede variar para distintas operaciones.
2. Puede haber múltiples unidades funcionales de punto flotante. Se presentará una detención si la instrucción que se va a tratar puede provocar un riesgo estructural para la unidad funcional que utiliza o un riesgo por dependencias de datos.

Para abordar el estudio de la implementación DLX, se va a suponer que hay cuatro unidades funcionales separadas:

1. La unidad principal de enteros.
2. Un sumador/restador FP.
3. Un multiplicador FP y entero.
4. Un divisor FP y entero.

La unidad de enteros manipula todas las cargas y almacenamientos en ambos conjuntos de registros, todas las operaciones enteras (excepto la multiplicación y división), y los saltos. Por ahora se supondrá también que las etapas de ejecución de otras unidades funcionales no están segmentadas, para que ninguna otra instrucción que utilice la unidad funcional pueda aparecer hasta que la instrucción anterior deje su etapa EX. Además, si una instrucción no puede continuar a su etapa EX, se detendrá por completo la segmentación a partir de esa instrucción. La **Figura 4.24** muestra la estructura de la segmentación resultante.

Como la etapa EX se puede repetir muchas veces (de 30 a 50 repeticiones para una división en punto flotante no sería excesivo), se debe encontrar una forma de detectar largas dependencias potenciales y resolver riesgos que duran decenas de ciclos de reloj, en lugar de uno o dos como se ha visto hasta ahora. También hay que tratar el solapamiento entre instrucciones enteras y de punto flotante. Sin embargo, el solapamiento de instrucciones enteras y de punto flotante no complica la detección de riesgos, excepto en las referencias a memoria de punto flotante y las transferencias entre los conjuntos de registros; esto es como consecuencia de que, excepto para las referencias a memoria y transferencias, los registros enteros y de punto flotante son distintos y las instrucciones enteras operan sobre registros enteros, mientras que las operaciones en punto flotante lo hacen sólo sobre sus propios registros de punto flotante. Esta simplificación del control de la segmentación es la principal ventaja de tener bancos de registros separados para datos enteros y de punto flotante.

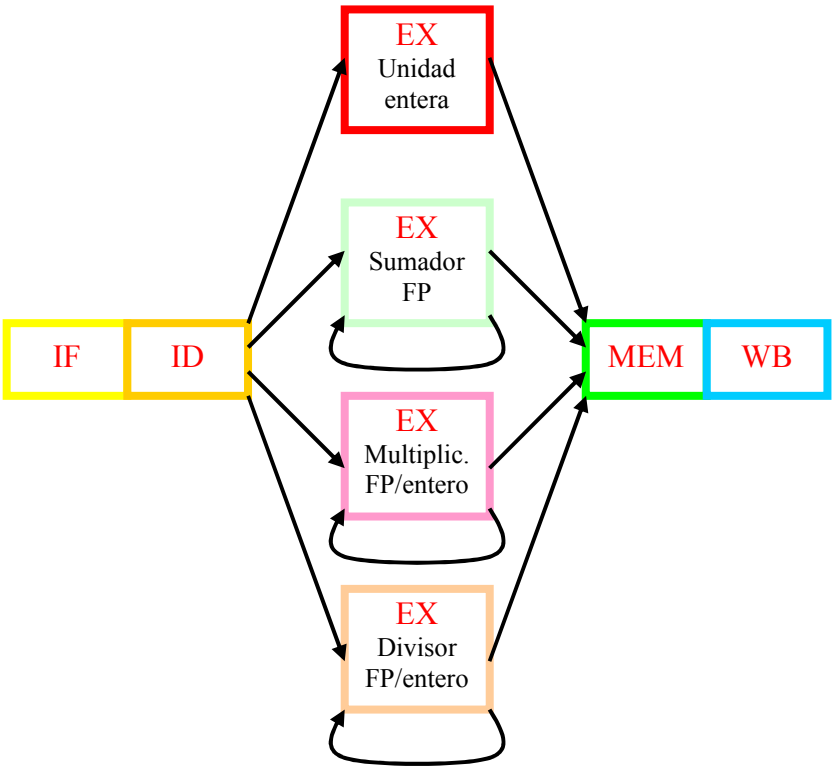


Figura 4.24 La segmentación del procesador DLX con tres unidades funcionales de punto flotante no segmentadas. Como se emite en cada ciclo de reloj sólo una instrucción, todas las instrucciones van a través de la segmentación estándar para operaciones con enteros. Las operaciones con datos en punto flotante, sencillamente forman un bucle cuando alcanzan la etapa EX, continúan en la etapa MEM y por último en la etapa WB.

Por ahora supóngase que todas las operaciones en punto flotante emplean el mismo número de ciclos de reloj (por ejemplo, 20 en la etapa EX). ¿Qué tipo de circuitería de detección de riesgos se necesitará?. Como todas las operaciones se suponen tardan la misma cantidad de tiempo, y las lecturas y escrituras de los registros siempre ocurren en la misma etapa, solamente son posibles riesgos RAW. Los riesgos WAR y WAW no se pueden presentar. Por lo tanto, en estas condiciones, todo lo que se necesita es conocer el registro destino de cada unidad funcional activa. Cuando se quiera emitir una nueva instrucción en punto flotante, se realizarán los siguientes pasos:

1. *Comprobar los riesgos estructurales* → Esperar hasta que la unidad funcional requerida deje de estar ocupada.
2. *Comprobar riesgos por dependencias de datos RAW* → Esperar hasta que los registros fuente no estén en la lista de destinos para cualquiera de las etapas EX, en las unidades funcionales.
3. *Comprobar adelantamientos* → Examinar si el registro destino de una instrucción, en la etapa MEM o en la etapa WB, es uno de los registros fuente de la instrucción de punto flotante; si es así, habilitar el multiplexor de entrada para utilizar ese resultado, en lugar del contenido del registro.

Surge un pequeño problema de conflictos entre las instrucciones de carga en punto flotante y las operaciones en punto flotante cuando ambas alcanzan simultáneamente la etapa WB. Esta situación se tratará a continuación de forma general.

En la discusión anterior, se ha supuesto que todos los tiempos de ejecución de las unidades funcionales de punto flotante son iguales; sin embargo esto es imposible de mantener en la práctica: las sumas de punto flotante pueden realizarse, normalmente, en menos de 5 ciclos de reloj; la multiplicación en menos de 10, y las divisiones en 20 como mínimo. Se pretende en definitiva, poder tener diferentes tiempos de ejecución para unidades funcionales diferentes, aún solapándose las ejecuciones. Esto no cambiaría la estructura básica de la segmentación para DLX propuesta en la Figura 4.24, aunque variaría el número de iteraciones en torno a los bucles. Sin embargo, el solapamiento en la ejecución de instrucciones cuyos tiempos de ejecución difieren, crea tres complicaciones:

1. Contención para accesos a registros al final de la segmentación.
2. La posibilidad de riesgos WAR y WAW.
3. Mayor dificultad para proporcionar interrupciones precisas.

Ya se ha visto que las instrucciones de carga y las operaciones de punto flotante pueden competir en las escrituras al banco de registros de punto flotante. Cuando las operaciones de punto flotante tengan distintos tiempos de ejecución, también pueden colisionar cuando traten de escribir resultados. Este problema puede resolverse estableciendo una prioridad estática para utilizar la etapa WB. Cuando múltiples instrucciones intenten entrar simultáneamente en la etapa MEM, todas, excepto la de mayor prioridad, son detenidas en su etapa EX. Una heurística sencilla, aunque a veces subóptima, es dar prioridad a la unidad con mayor latencia, ya que, probablemente, será la causante del cuello de botella. Aunque este esquema es razonablemente sencillo de implementar, este cambio de la segmentación de DLX es bastante significativo. En la segmentación de operaciones enteras, todos los riesgos se comprobaban antes de enviar la instrucción a la etapa EX. Con este esquema para determinar accesos al puerto de escritura de resultados, las instrucciones pueden detenerse después de que se emitan.

El solapamiento de instrucciones con tiempos de ejecución diferentes puede introducir riesgos WAR y WAW en el procesador DLX, porque el instante en el que las instrucciones escriben deja de ser fijo. Si todas las instrucciones todavía leen sus registros al mismo tiempo, no se introducirán riesgos WAR.

Los riesgos WAW se introducen porque las instrucciones pueden escribir sus resultados en un orden diferente en el que éstas aparecen. Por ejemplo:

$$DIVF \quad F0, F2, F4$$

$$SUBF \quad F0, F8, F10$$

Se presenta un riesgo WAW entre las operaciones de división y resta. La resta se completará primero, escribiendo su resultado antes que la división escriba el suyo → **Observar** que este riesgo solamente ocurre cuando se sobreescriba el resultado de la división, sin que lo utilice ninguna instrucción. Si hubiese una utilización de $F0$ entre $DIVF$ y $SUBF$, la segmentación se detendría debido a la dependencia de los datos, y $SUBF$ no se emitiría hasta que se completase $DIVF$. Hay dos formas posibles de tratar este riesgo WAW:

1. Primer enfoque → Consiste en retardar la emisión de la instrucción $SUBF$ hasta que $DIVF$ entre en la etapa MEM.
2. Segundo enfoque → Consiste en eliminar el resultado de la división detectando el riesgo, e indicando a la unidad de dividir que no escriba su resultado. Por tanto, $SUBF$ podría proseguir en seguida.

Como este riesgo es raro, ambos esquemas funcionarán bien (se puede escoger el que sea más sencillo de implementar). Sin embargo, cuando una segmentación se vuelve más compleja, se necesita dedicar mayores recursos para determinar cuándo puede emitirse una instrucción.

Otro problema causado por las instrucciones de larga ejecución se puede ilustrar con una secuencia de código muy similar:

$DIVF \quad F0, F2, F4$

$ADDF \quad F10, F10, F8$

$SUBF \quad F12, F12, F14$

Esta secuencia de código parece sencilla; no hay dependencias. El problema con el que nos enfrentamos surge porque una instrucción que aparezca antes en la secuencia de código, pueda completarse después de una instrucción que aparezca posteriormente en la misma secuencia. En este ejemplo, se puede esperar que $ADDF$ y $SUBF$ se completen antes que $DIVF$. A esto se le denomina *terminación fuera de orden* (*out-of-order completion*), y es común en los procesadores segmentados con operaciones de larga duración. Pero, puesto que la detección de riesgos impide que cualquier dependencia entre instrucciones sea violada, ¿por qué es un problema la terminación fuera de orden?. Suponer que $SUBF$ provoca una interrupción aritmética de punto flotante cuando se haya completado $ADDF$, pero $DIVF$ no. El resultado será una *interrupción imprecisa*; que se está tratando de evitar. Se podría pensar que se puede evitar el problema parando la segmentación de punto flotante, como hacíamos para las instrucciones enteras. Pero la interrupción puede estar en una posición donde esto no sea posible; por ejemplo, si $DIVF$ provoca una interrupción aritmética de punto flotante una vez que se completa la suma, se podría no tener una interrupción precisa a nivel hardware porque, como $ADDF$ destruye uno de sus operandos ($F10$) no se podría restaurar el estado en que anteriormente estaba $DIVF$, aún con la ayuda del software.

Este problema se ha creado porque las instrucciones se completan en un orden diferente del que aparecieron. Hay cuatro enfoques posibles para tratar la terminación fuera de orden:

1. *Ignorar el problema y conformarse con interrupciones imprecisas* → Este enfoque fue utilizado en los años sesenta y comienzos de los setenta. También se sigue utilizando por algunos supercomputadores, donde no se permiten ciertas clases de interrupciones o son tratadas por el hardware sin

detener la segmentación. Pero, en las máquinas actuales es difícil utilizar este enfoque, por las características de la memoria virtual y el estándar de coma flotante IEEE 754, que, esencialmente, requiere interrupciones precisas, a través de una combinación de hardware y software.

2. *Poner en cola los resultados de una operación, hasta que se completen todas las operaciones que comenzaron antes* → Algunos sistemas computadores utilizan realmente esta solución, pero llega a ser cara cuando la diferencia de tiempos de ejecución entre las operaciones es grande (el número de resultados puede hacer muy larga la cola). Además, los resultados de la cola deben ser desviados para continuar la ejecución de instrucciones mientras se espera que termine la instrucción más larga; esto requiere un número grande de comparadores y un multiplexor de muchas entradas y variables de selección. Hay dos variantes viables para este enfoque básico:

    a. Utilizar un *fichero de historia* (utilizado en el CYBER 180/990) → Este fichero mantiene los valores originales de los registros y, cuando se presente una interrupción y haya que volver al estado anterior antes que se complete alguna instrucción fuera de orden, el valor original del registro se puede restaurar a partir de dicho fichero. Una técnica similar se utiliza para el direccionamiento de autoincremento y autodecremento en los VAX.

    b. Utilizar un *fichero de futuro* (propuesto por J. Smith y Plezkun (1988)[2]) → Mantiene el valor más nuevo de un registro; cuando se haya completado todas las instrucciones anteriores, el banco de registros principal se actualiza a partir del fichero de futuro. De esta forma, una interrupción puede tener el banco de registros principal con los valores precisos del estado interrumpido.

3. *Permitir que las interrupciones lleguen a ser algo imprecisas,* pero manteniendo suficiente información para que las rutinas de manejo de traps puedan crear una secuencia precisa para la interrupción → Técnica en uso actualmente. Para que permita que las rutinas de manejo de traps puedan crear una secuencia precisa para la interrupción, se deben conocer las operaciones que estaban en el procesador y sus PC. Entonces, después de tratar un trap, el software termina cualquier instrucción que preceda a la última instrucción completada, y la secuencia puede recomenzar.

[2] Smith, J. E. and A. R. Plezkun [1988] 'Implementing precise interrupts in pipelined processors', *IEEE Trans. On Computers* 37: 5 (May) 562-573. (p. 364)

Considerar la secuencia de código siguiente, que plantea el peor caso:

$Instrucción_1$ → Instrucción de larga ejecución que interrumpe la ejecución

$Instrucción_2, \text{K}, Instrucción_{n-1}$ → Una serie de instrucciones que no se completan

$Instrucción_n$ → Una instrucción que se termina

Dados los PC de todas las instrucciones en curso y el PC de retorno de la interrupción, el software puede determinar el estado de la $Instrucción_1$ y de la $Instrucción_n$. Ya que la $Instrucción_n$ se ha completado, se quiere recomenzar la ejecución de la $Instrucción_{n+1}$. Después de tratar la interrupción, el software debe simular la ejecución de: $Instrucción_1, \text{K}, Instrucción_{n-1}$. Entonces, se puede volver de la interrupción y recomenzar la $Instrucción_{n+1}$. La complejidad de ejecutar estas instrucciones adecuadamente es la dificultad principal de este esquema. Hay una simplificación importante, si $Instrucción_2, \text{K}, Instrucción_n$ son instrucciones enteras, entonces se sabe que si se ha completado la $Instrucción_n$, también se ha completado toda la secuencia $Instrucción_2, \text{K}, Instrucción_{n-1}$; por tanto, sólo necesitan ser tratadas las operaciones de punto flotante. Para que este esquema sea tratable, se puede limitar el número de instrucciones de punto flotante cuya ejecución se pueda solapar; por ejemplo, si sólo se solapan dos instrucciones, entonces sólo hay que completar por software la instrucción interrumpida. Esta restricción puede reducir el rendimiento potencial si la segmentación de punto flotante es profunda o si hay un número significativo de unidades funcionales de punto flotante. Este enfoque se utiliza en la arquitectura SPARC para permitir el solapamiento de operaciones enteras y de punto flotante.

4. *Esquema híbrido que permite que continúe la emisión de las instrucciones sólo si es cierto que todas las instrucciones anteriormente emitidas terminarán sin provocar ninguna interrupción*[3] → Garantiza que cuando se presente una interrupción, no se complete ninguna instrucción posterior a la de la interrupción y que se puedan completar todas las instrucciones anteriores a la de la interrupción. Esto, a veces, significa detener la máquina para mantener interrupciones precisas. Para hacer que funcione este esquema, las unidades

[3] Este esquema se explica en detalle en la Sección A-7 del Apéndice A del libro 'Arquitectura de Computadores. Un enfoque cuantitativo' [1993]. Traducido de la primera edición en ingles: 'Computer Architecture. A Quantitative Approach'. McGraw-Hill/Interamericana de España, S.A.

funcionales de punto flotante deben determinar al principio de las etapas si es podible que se provoque una interrupción al principio de la etapa EX (en los tres primeros ciclos de la segmentación de DLX), para impedir que se completen las siguientes instrucciones. Este esquema se utiliza en las arquitecturas MIPS R2000/3000.

4.6 AUMENTO DEL PARALELISMO A NIVEL DE INSTRUCCIÓN

4.6.1 Introducción

El objetivo de las técnicas explicadas en este apartado es permitir *emitir múltiples instrucciones en un ciclo de reloj* $\Rightarrow$ CPI (ciclos de reloj por instrucción) menor que la unidad.

Para mantener el procesador a pleno rendimiento, se debe explotar el paralelismo entre las instrucciones buscando secuencia de instrucciones no relacionadas que se puedan solapar en la segmentación. Dos instrucciones relacionadas deben estar separadas por una distancia igual a la latencia de la segmentación de la primera de las instrucciones. Se supondrá:

- Latencias de operaciones utilizadas según la Tabla 4.12.

Instrucción que produce resultado	Instrucción destino	Latencia en ciclos
Op FP ALU	Otra op FP ALU	3
Op FP ALU	Almacenamiento doble	2
Carga doble	Op FP ALU	1
Carga doble	Almacenamiento doble	0

Tabla 4.12 Latencias de operaciones utilizadas en este apartado. La primera columna indica la instrucción que produce la latencia, en la segunda el tipo de instrucción que hace efectiva esa latencia; y, la tercera columna indica la separación en ciclos de reloj para evitar una detención.

- Los saltos tienen un retardo de un ciclo de reloj.
- Las unidades funcionales están completamente segmentadas o repetidas, y se puede emitir una operación cada ciclo de reloj.

Se tratarán tanto *técnicas software* como *técnicas hardware*:

Técnicas software:

- Desenrollado o extensión de bucles
- Segmentación software y planificación de trazas

Técnicas hardware:

- Escalar supersegmentada
- Superescalar

Técnicas hardware y software:

- Arquitectura VLIW

4.6.2 Desenrollado o extensión de bucles

Técnica sencilla de compilación que nos permite incrementar el paralelismo a nivel de instrucción.

Ejemplo 4.5 Se trata de un bucle que suma un valor escalar a un vector en memoria; ver **Figura 4.25**. Se supone que el array comienza en la posición de memoria 0; si no fuese así, habría que añadir una instrucción adicional entera. El registro **R1** contiene la dirección de memoria de comienzo del último elemento del array; **F2 F3** tienen cargado el escalar de 64 bits.

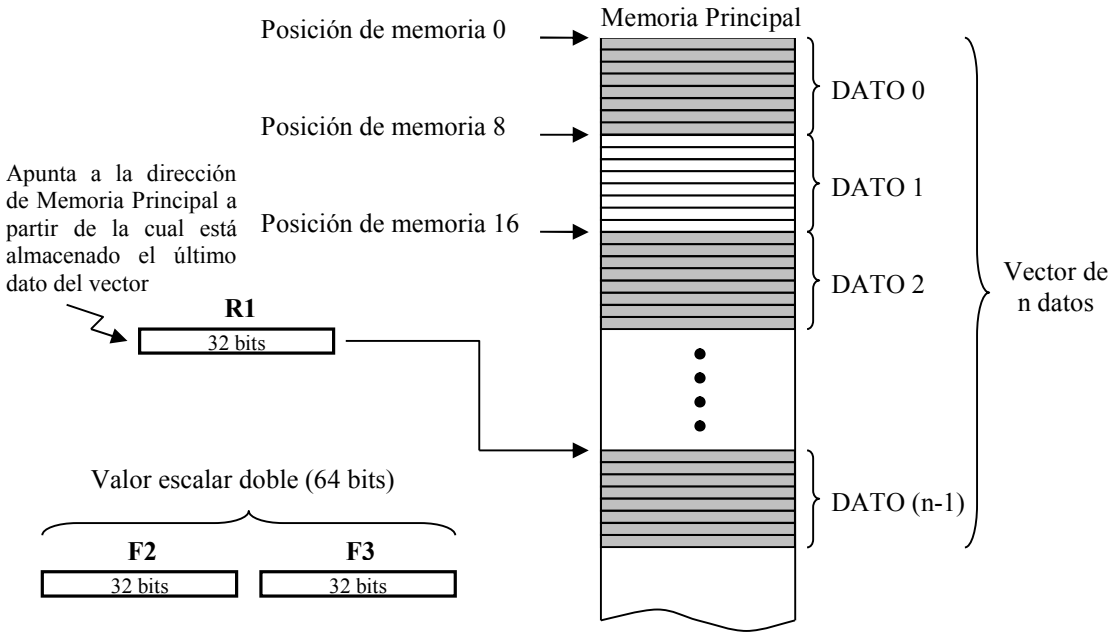


Figura 4.25 Definición del vector en la memoria.

El código de DLX, no teniendo en cuenta la segmentación, sería el mostrado en la Tabla 4.13.

```
LOOP:  LD      F0, 0(R1)     ; Carga el elemento del vector (carga doble)
                             ; empezando por el último
       ADDD    F4, F0, F2    ; Suma el escalar de F2 (suma en coma flotante de
                             ; doble precisión)
       SD      0(R1), F4     ; Almacena (almacenamiento doble) el elemento
                             ; del vector
       SUB     R1, R1, #8    ; Decrementa el puntero en 8 bytes (cada dato son
                             ; 64 bits por ser en doble precisión)
       BNEZ    R1, LOOP      ; Salta cuando no es cero
```

Tabla 4.13 Código para el procesador DLX, sin tener en cuenta la segmentación.

Cuando se ejecuta el bucle sin planificación, y teniendo en cuenta las latencias de operaciones indicadas anteriormente, se tiene el resultado expuesto en la Tabla 4.14.

				Ciclo de reloj en el que se emite
LOOP:	**LD**	**F0, 0(R1)**		1
	Detención			2
	ADDD	**F4, F0, F2**		3
	Detención			4
	Detención			5
	SD	**0(R1), F4**		6
	SUB	**R1, R1, #8**		7
	BNEZ	**R1, LOOP**		8
	Detención			9

Tabla 4.14 Ciclo de reloj en el que se emiten las distintas instrucciones. Ejecución sin planificación.

Planificando el bucle se reduce el número de ciclos por iteración, de 9 ciclos por iteración pasa a 6 ciclos por iteración (Tabla 4.15).

				Ciclo de reloj en el que se emite
LOOP:	**LD**	**F0, 0(R1)**		1
	Detención			2
	ADDD	**F4, F0, F2**		3
	SUB	**R1, R1, #8**		4
	SD	**0(R1), F4**	; Cambiado	5
	BNEZ	**R1, LOOP**	; Salto retardado	6

Tabla 4.15 Ciclo de reloj en el que se emiten las distintas instrucciones, una vez ha sido planificado el bucle (6 ciclos de reloj por iteración).

Para la mayoría de los compiladores, el intercambio que se hace de la instrucción SUB por la instrucción SD no es trivial, ya que verían que la instrucción SD depende de la de SUB. El compilador tendría que ser lo suficientemente "inteligente" para realizar dicho cambio [aparte de detectar la dependencia tiene que ser capaz de cambiar la dirección (su desplazamiento pasa de ser 0 a ser 8 por haberse reducido en 8 unidades el valor de R1)].

Con la planificación realizada, se termina un elemento del vector cada 6 ciclos de reloj, siendo el trabajo real de operación sobre el elemento del vector de tres ciclos (correspondientes a LD, $ADDD$ y SD); los tres ciclos de reloj restantes se emplean en gastos de bucles (SUB, $BNEZ$ y una detención). Para eliminar estos tres ciclos de reloj, necesitamos obtener más operaciones en el bucle $\Rightarrow$ Un sencillo esquema para incrementar el número de instrucciones entre las ejecuciones de los saltos del bucle es *desenrollar el bucle* (*loop unrolling*) (ver Tabla 4.16) $\Rightarrow$ Se replica múltiples veces el cuerpo del bucle, ajustando su código de terminación y planificando entonces el bucle desenrollado (ver Tabla 4.17). Para lograr una planificación efectiva, utilizaremos diferentes registros para cada iteración, incrementando así el número de registros.

```
LOOP:  LD      F0, 0(R1)      ; Carga el elemento del vector (carga doble)
                              ; empezando por el último
       Detención
       ADDD    F4, F0, F2     ; Suma el escalar de F2 (suma en coma flotante
                              ; de doble precisión)
       Detención
       Detención
       SD      0(R1), F4      ; Almacena (almacenamiento doble) el elemento
                              ; del vector
       LD      F6, -8(R1)     ; Se repite la secuencia de instrucciones y se han
                              ; eliminado SUB y BNEZ una vez
       Detención
       ADDD    F8, F6, F2
       Detención
       Detención
       SD      -8(R1), F8
       LD      F10, -16(R1)   ; Se repite la secuencia de instrucciones y se han
                              ; eliminado SUB y BNEZ una vez
       Detención
       ADDD    F12, F10, F2
       Detención
       Detención
       SD      -16(R1), F12
       LD      F14, -24(R1)   ; Se repite la secuencia de instrucciones y se han
                              ; eliminado SUB y BNEZ una vez
       Detención
       ADDD    F16, F14, F2
       Detención
       Detención
       SD      -24(R1), F16
       SUB     R1, R1, #32    ; Decrementa el puntero en 8*4 bytes (cada dato
                              ; son 64 bits por ser en doble precisión)
       BNEZ    R1, LOOP       ; Salta cuando no es cero
       Detención
```

Tabla 4.16 Bucle desenrollado tres veces (cuatro copias del cuerpo), **sin planificación**, y suponiendo que incialmente R1 es múltiplo de 4.

Nota: Aunque se han utilizado registros diferentes en cada iteración explicitada, en el programa propuesto no es necesario cambiar de registro ya que según se va realizando la suma se va almacenando el resultado. Sí es necesario hacer uso de diferentes registros cuando se agrupan todas las cargas, todas las sumas y todos los almacenamientos, que es lo que se hace con la planificación en el programa que se muestra a continuación.

Se han eliminado tres saltos y tres decrementos de R1. Se han ajustado las direcciones de las cargas y almacenamientos. Vemos que sin planificación, cada operación va seguida de una operación dependiente, se provocarán detenciones. Este bucle se ejecutará en 27 ciclos de reloj ó 6.75 ciclos de reloj por cada uno de los cuatro elementos del bucle.

```
LOOP:  LD      F0, 0(R1)      ; Carga el elemento del vector (carga doble)
                              ; empezando por el último
       LD      F6, -8(R1)
       LD      F10, -16(R1)
       LD      F14, -24(R1)
       ADDD    F4, F0, F2     ; Suma el escalar de F2 (suma en coma flotante
                              ; de doble precisión)
       ADDD    F8, F6, F2
```

ADDD	**F12, F10, F2**	
ADDD	**F16, F14, F2**	
SD	**0(R1), F4**	; Almacena (almacenamiento doble) el elemento ; del vector
SD	**-8(R1), F8**	
SD	**-16(R1), F12**	
SUB	**R1, R1, #32**	; Decrementa el puntero en 8*4 porque cada dato ; ocupa 8 posiciones (son 64 bits por cada dato)
BNEZ	**R1, LOOP**	; Salta cuando no es cero
SD	**8(R1), F16**	; 8 – 32 = -24

Tabla 4.17 Bucle desenrollado tres veces, **con planificación para DLX**, y suponiendo que incialmente R1 es múltiplo de 4.

Ahora el tiempo de ejecución del bucle desenrollado ha caído a un total de 14 ciclos de reloj ó 3.5 ciclos de reloj por cada uno de los cuatro elementos.

Conclusión: Desenrollar el bucle, aunque incrementa el tamaño de los fragmentos, es un método sencillo pero útil para incrementar el código lineal que puede ser planificado efectivamente, consiguiéndose reducir el número de ciclos por elemento. Esta transformación en tiempo de compilación es similar a la que hace el Algoritmo de Tomasulo con el renombramiento de registros y la ejecución fuera de orden.

4.6.3 Aumento del paralelismo a nivel de instrucción con segmentación software y planificación de trazas

Desenrollar bucles crea secuencias más largas de código, que se pueden utilizar para explotar más el paralelismo a nivel de instrucción. Para este mismo propósito se han desarrollado dos técnicas más generales:

- Segmentación software
- Planificación de trazas

SEGMENTACIÓN SOFTWARE

Es una técnica para reorganizar bucles, de tal forma que, cada iteración en el código segmentado por software se haga a partir de secuencias de instrucciones escogidas en diferentes iteraciones del segmento de código original.

En un bucle, con la ***planificación*** se intercalan instrucciones de diferentes iteraciones de bucles, juntando todas las cargas, después juntando todas las sumas, después juntando todos los almacenamientos. Un bucle, ***segmentado por software***, intercala las instrucciones de diferentes iteraciones sin desenrollar el bucle; el bucle segmentado por software contendrá una carga, una suma y un almacenamiento, cada uno de una iteración diferente. También hay código de arranque que se necesita antes que comience el bucle, así como código para concluir una vez que se complete el

bucle. Esta técnica es la contrapartida software a lo que el Algoritmo de Tomasulo hace en hardware.

Si se omiten los códigos de arranque y de finalización, una versión segmentada por software del bucle planteado puede ser la que se muestra a continuación en la Tabla 4.18.

```
LOOP:   SD      0(R1), F4       ; Almacena en M[i]
        ADDD    F4, F0, F2      ; Suma en M[i-1]
        LD      F0, -16(R1)     ; Carga M[i-2]
        SUB     R1, R1, #8
        BNEZ    R1, LOOP
```

Tabla 4.18 Versión segmentada por software del bucle planteado, sin considerar los códigos de arranque ni de finalización.

Si se ignoran las partes de arranque y terminación, este bucle se puede ejecutar a una velocidad de 5 ciclos por resultado.

Algunos de los algoritmos para segmentación software utilizan desenrollado de bucles previamente, para determinar cómo segmentar el bucle. Por esa razón puede considerarse la segmentación software como un desenrollado simbólico de bucles.

La principal ventaja de la *segmentación software* sobre el *desenrollado directo de bucles* es que consume menos espacio de código.

El *desenrollado de bucles* reduce gastos del bucle (código de saltos y actualización de contadores), la *segmentación software* reduce la porción de tiempo en la que el bucle no se ejecuta a la velocidad máxima a una única vez al comienzo y al final del bucle. El mejor rendimiento se logra utilizando ambas.

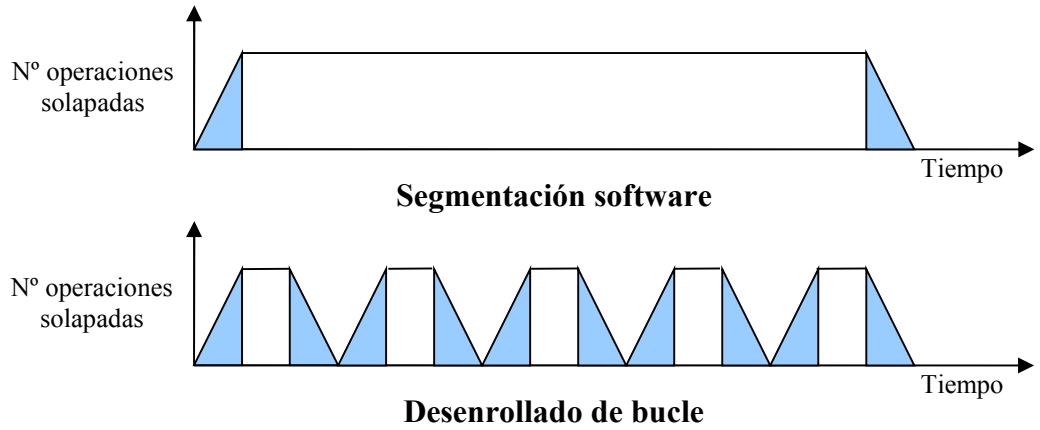


Figura 4.26 Patrón de ejecución para un bucle segmentado por software y un bucle desenrollado. (En área sombreada no se ejecuta con solapamiento o paralelismo máximo entre instrucciones. Esto ocurre una vez al comienzo del bucle y otra al final para el bucle segmentado por software. Para el bucle desenrollado ocurre m/n veces si el bucle tiene un total de m ejecuciones y se desenrolla n veces).

PLANIFICACIÓN DE TRAZAS

Esta técnica es particularmente útil para la arquitectura VLIW, para las que se desarrolló originalmente.

La *planificación de trazas* es una combinación de dos procesos separados:

- *Selección de trazas* ⇒ Trata de encontrar la secuencia más probable de operaciones para juntarlas en un pequeño número de instrucciones; esta secuencia se denomina *traza*. (El desenrollado de bucles se utiliza para generar trazas largas, ya que los saltos del bucle son efectivos con una probabilidad alta).
- *Compactación de trazas* ⇒ Trata de compactar la traza en un pequeño número de instrucciones anchas. La compactación de trazas intenta mover operaciones tan pronto como pueda en una secuencia (traza), empaquetando las operaciones en el mínimo número posible de instrucciones anchas. Dos consideraciones diferentes al compactar una traza:
    - *Las dependencias de los datos*, que fuerzan un orden parcial sobre las operaciones.
    - *Los puntos de salto*, que crean lugares a través de los cuales no se puede mover el código fácilmente. Los saltos son el principal impedimento para este proceso.

La ventaja principal de la ***planificación de trazas*** frente a las técnicas más sencillas de planificación de procesadores segmentados, es que incluye un método para mover código a través de los saltos.

Desenrollado de bucles, *planificación de trazas*, y *segmentación software*, todos ayudan a intentar incrementar la cantidad de paralelismo local de las instrucciones que se puede explotar por una máquina.

4.6.4 Técnicas hardware para hacer el CPI menor que la unidad

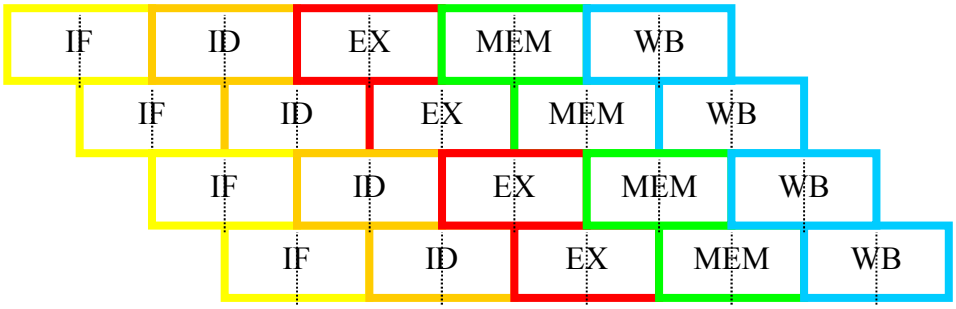
Tomando como referencia el computador segmentado (computador escalar) de cinco etapas estudiado hasta ahora (IF, ID, EX, MEM y WB), hay dos técnicas de tipo hardware que permiten mejorar el rendimiento de estos sistemas. Si con el computador segmentado se considera que se puede ejecutar una instrucción por ciclo, ambas técnicas reducen por debajo de uno el número de ciclos por instrucción. Las dos técnicas hardware son:

- *Escalar supersegmentada*

- *Superescalar*

COMPUTADORES SUPERSEGMENTADOS

La idea básica del concepto de *supersegmentación* consiste en realizar una ejecución segmentada de instrucciones con muchas etapas (**Figura 4.27**); permitiendo de esa forma solapar en mayor grado la segmentación. Es decir, trata de dar una mayor profundidad a la segmentación. Como ya se ha comentado anteriormente en este tema, cuando existe una profundidad grande de la segmentación, todos los problemas que se han venido planteando en la segmentación propuesta para el procesador DLX se ven agravados; el hardware necesario se complica enormemente. Por supuesto, con estructura que se plantea y considerando aparte las complicaciones que pueda implicar, el rendimiento del sistema se incrementa.



IF ≡ Búsqueda de la instrucción
ID ≡ Decodificación de la instrucción y búsqueda de registros
EX ≡ Ejecución y cálculo de direcciones efectivas
MEM ≡ Acceso a memoria
WB ≡ Postescritura (Write Back)

Figura 4.27 Supersegmentación comparada con la segmentación DLX.

COMPUTADORES SUPERESCALARES

El concepto *superescalar* consiste en realizar una ejecución segmentada de instrucciones en varios cauces de instrucciones (**Figura 4.28**). En una máquina superescalar, el hardware puede emitir un pequeño número (por ejemplo, de 2 a 4) de instrucciones independientes en un solo ciclo. Sin embargo, si las instrucciones del flujo de instrucciones del flujo dado son dependientes o no cumplen ciertos criterios, sólo se emitirá la primera instrucción de la secuencia. Una máquina donde el compilador tenga completa responsabilidad para crear un paquete de instrucciones que se puedan emitir simultáneamente, y el hardware no tome dinámicamente decisiones sobre múltiples emisiones, debería considerarse del tipo VLIW (se estudian en el siguiente apartado).

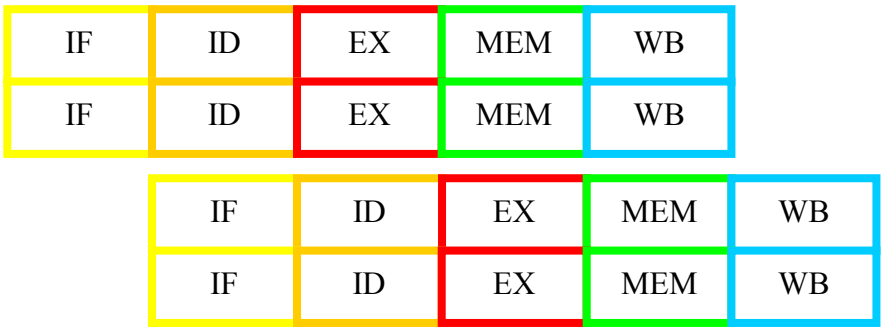


Figura 4.28 Superescalar para DLX con dos cauces de instrucciones.

Para el procesador DLX que estamos estudiando, para conseguir emitir múltiples instrucciones por ciclo, se puede plantear por ejemplo que una instrucción sea de carga, almacenamiento, salto u operación entera de la ALU, y otra cualquier operación de punto flotante. Está claro que emitir una operación entera en paralelo con una operación de punto flotante es menos exigente que emitir dos instrucciones cualesquiera.

Al emitir en paralelo una operación en punto flotante y otra entera, se minimiza la necesidad de hardware adicional (las operaciones enteras y de punto flotante utilizan diferentes conjuntos de registros y diferentes unidades funcionales). El único conflicto surge cuando la instrucción entera es una carga, almacenamiento o transferencia de punto flotante. Esto crea contención para los puertos de los registros de punto flotante, y también puede crear un riesgo si la operación de punto flotante utiliza el resultado de una carga de punto flotante emitida al mismo tiempo. Ambos problemas pueden resolverse detectando esta contención como un riesgo estructural y retrasando la emisión de la instrucción de punto flotante. La contención también puede eliminarse suministrando dos puertos adicionales, uno de lectura y otro de escritura, en el banco de registros de punto flotante. También sería necesario añadir algunos caminos de desvío adicionales para evitar pérdida de rendimiento.

Hay otra dificultad que puede limitar la efectividad de una segmentación superescalar. En la segmentación básica propuesta para el procesador DLX, las cargas tienen una latencia de un ciclo de reloj; impidiendo que una instrucción utilizase el resultado sin detención. En la segmentación superescalar, el resultado de una instrucción de carga no se puede utilizar en el mismo ciclo de reloj ni en el siguiente. Esto significa que las tres instrucciones siguientes no pueden utilizar el resultado de la carga sin detención; sin puertos extras, las transferencias entre los conjuntos de registros estarían afectadas de la misma forma. El retardo de los saltos también llega a ser de tres instrucciones. Para explotar efectivamente el paralelismo disponible en una máquina superescalar, se necesita implementar técnicas más ambiciosas de planificación por el

compilador, así como una decodificación de instrucciones más compleja. Desenrollar el bucle ayuda a generar mayores fragmentos lineales para su posterior planificación. No abordaremos esas técnicas de compilación más potentes en este curso.

4.6.5 Arquitectura VLIW (Very Long Instruction Word – Palabra de Instrucción Muy Larga)

Los computadores superescalares pueden emitir dos, tres, cuatro, o múltiples instrucciones por ciclo de reloj; pero resulta difícil de determinar si se pueden emitir tantas instrucciones simultáneamente sin saber en qué orden estarán las instrucciones cuando se busquen de memoria y qué dependencias pueden existir entre ellas. Una alternativa es la arquitectura VLIW, que utiliza múltiples unidades funcionales independientes, y empaqueta múltiples operaciones en una instrucción muy larga. La instrucción VLIW tiene un conjunto de campos, asociados cada uno de ellos a distintas unidades funcionales. Una instrucción VLIW puede incluir varias operaciones enteras, varias operaciones en punto flotante, varias referencias a memoria, etc.

La arquitectura VLIW incluye una combinación u otra de instrucciones según la arquitectura hardware existente. Por lo tanto, podemos decir que la arquitectura VLIW implica tanto a mecanismos software como a mecanismos hardware.

En principio, el enfoque presentado por la arquitectura VLIW resulta ideal, no encontrándose inicialmente limitaciones para la misma (si con cuatro unidades funcionales de enteros se pueden hacer cuatro operaciones a la vez, por qué no poner 40 unidades funcionales), aunque evidentemente sí existen.

¿Cuáles son las limitaciones de un enfoque VLIW?.

- *Paralelismo limitado* $\Rightarrow$ Hay una cantidad limitada de paralelismo disponible en las secuencias de instrucciones.
- *Recursos hardware limitados* $\Rightarrow$ Duplicar las unidades funcionales enteras de punto flotante es fácil y el coste aumenta linealmente. Sin embargo hay un gran incremento en el ancho de banda del fichero de registros y de la memoria. Por lo tanto, añadir unidades aritméticas solamente, no ayudaría ya que la máquina estaría limitada por el ancho de banda de memoria.
- *Explosión del tamaño del código* $\Rightarrow$ Hay dos elementos diferentes que se combinan para aumentar sustancialmente el tamaño del código:
    - Generar suficientes operaciones en un fragmento de código lineal requiere bucles ambiciosamente desenrollados, lo que incrementa el tamaño del código.

- Siempre que las instrucciones no sean completas, las unidades funcionales no utilizadas implican bits desaprovechados en la codificación de las instrucciones.

El reto más importante para estas máquinas es tratar de explotar grandes cantidades de paralelismo a nivel de instrucción. Cuando el paralelismo proviene de desenrollar bucles, el bucle original probablemente se habría ejecutado eficientemente en una máquina vectorial. No está claro que sea preferible una VLIW sobre una máquina vectorial para estas aplicaciones; los costes son similares y la máquina vectorial tiene normalmente la misma o mayor velocidad.