Capítulo 2

Arquitecturas Convencionales Avanzadas

2.1 Procesadores superescalares

Una implementación superescalar de la arquitectura de un procesador es una implementación en la que las instrucciones comunes —aritmética entera y en punto flotante, cargas, almacenamientos y bifurcaciones condicionales— pueden iniciar su ejecución simultáneamente y ejecutarse de manera independiente. Estas implementaciones plantean problemas complejos de diseño relacionados con el cauce de instrucciones.

En la mayoría de las aplicaciones, la mayor parte de las operaciones se realizan con cantidades escalares. Así pues, la aproximación superescalar representa el siguiente paso en la evolución de los procesadores de uso general de altas prestaciones. La figura 2.1 muestra la temporización de un procesador superescalar con dos cauces de cinco etapas. Cada etapa del cauce ha sido *ensanchado* para acomodar dos instrucciones.

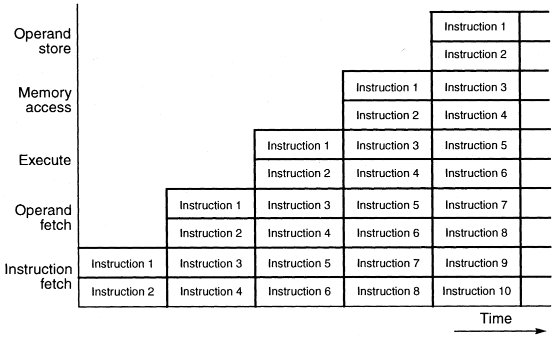

Figura 2.1: Ejecución en un procesador superescalar de dos cauces.

2.1.1 Limitaciones al paralelismo a nivel de instrucción

La temporización de la figura 2.1 sugiere que cada pareja de instrucciones se ejecutan simultáneamente, y el procesador puede operar dos veces más rápido que la versión escalar. Sin embargo, este rendimiento no puede alcanzarse siempre, ya que pueden existir dependencias en la secuencia de instrucciones. A menudo será necesario ejecutar completamente una instrucción antes de una instrucción posterior. La expresión *paralelismo a nivel de instrucciones* se refiere al grado en el que, en promedio, las instrucciones de un programa se pueden ejecutar en paralelo. Para maximizar el paralelismo a nivel de instrucciones, se puede usar una combinación de optimizaciones realizadas por el compilador y de técnicas hardware. Antes de examinar las técnicas de diseño utilizadas en las máquinas superescalares para aumentar el paralelismo a nivel de instrucciones, debemos estudiar las limitaciones fundamentales del paralelismo a las que el sistema tiene que enfrentarse. Es posible distinguir cinco limitaciones:

- Dependencia de datos verdadera
- Dependencia relativa al procedimiento
- Conflictos en los recursos
- Dependencia de salida
- Antidependencia

En la figura 2.2 se muestra el efecto de algunas dependencias en un cauce superescalar de grado 2.

Dependencia de datos verdadera

Consideremos la siguiente secuencia:

```
add  r1,r2
move r3,r1
```

La segunda instrucción se puede captar y decodificar, pero no se puede ejecutar hasta que finalice la ejecución de la primera instrucción. El motivo es que la segunda instrucción necesita un dato producido por la primera. A esta situación se le denomina dependencia de datos verdadera (también llamada *dependencia de flujo* o *dependencia escritura-lectura*).

Dependencias relativas al procedimiento

La presencia de bifurcaciones en una secuencia de instrucciones complica el funcionamiento del cauce. Las instrucciones que siguen a una bifurcación (se puede saltar o no saltar) tienen una dependencia relativa al procedimiento en la bifurcación y no se pueden ejecutar hasta que se ejecute la bifurcación.

Conflicto en los recursos

Un conflicto en un recurso es una competición de dos o más instrucciones por el mismo recurso al mismo tiempo. Ejemplos de recursos son las memorias, las cachés, los buses, los puertos del banco de registros, y las unidades funcionales.

Desde el punto de vista del cauce, un conflicto en los recursos presenta el mismo comportamiento que la dependencia de datos. Sin embargo hay algunas diferencias. En primer lugar, los conflictos en los recursos pueden superarse duplicando éstos, mientras que una dependencia de datos verdadera no se puede eliminar. Además, cuando una operación tarda mucho en finalizar, los conflictos en recursos se pueden minimizar segmentando la unidad funcional correspondiente.

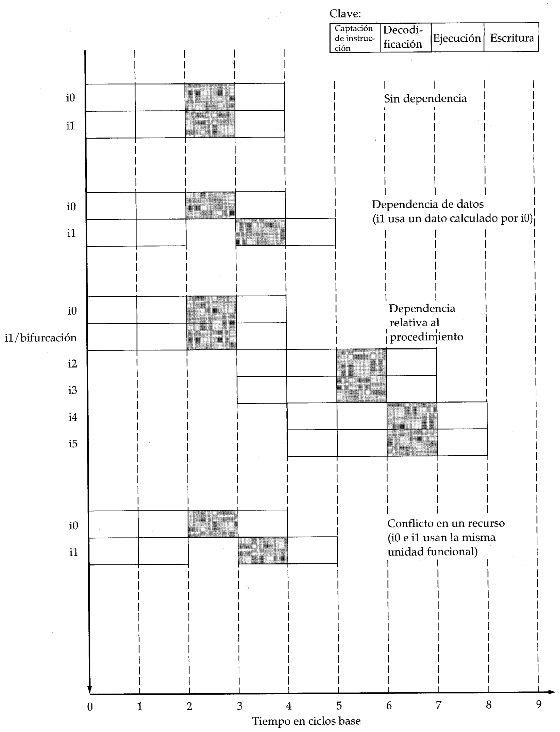

Figura 2.2: Efecto de las dependencias.

Dependencia de salida

Consideremos el siguiente fragmento de código que ilustra esta dependencia (también llamada *dependencia escritura-escritura*):

```
(I1) r3 ← r3 op r5
(I2) r4 ← r3 + 1
(I3) r3 ← r5 + 1
(I4) r7 ← r3 op r4
```

La instrucción I3 asigna un nuevo valor a r3 después de que I1 le haya dado otro valor. Si I3 se ejecuta hasta el final antes que I1, se captará un valor incorrecto del contenido de R3 para la ejecución de I4. Por consiguiente, I3 debe terminar después de I1 para producir el valor correcto de salida. Para asegurar esto, la emisión de la tercera instrucción debe detenerse si su resultado puede ser sobreescrito más tarde por una instrucción anterior que tarda más en finalizar.

Antidependencia

Consideremos el anterior fragmento de código. La instrucción I3 no puede finalizar antes de que la instrucción I2 comience a ejecutarse y haya captado sus operandos. Esto es así debido a que I3 actualiza el registro R3, que es un operando fuente de I2. El término *antidependencia* (también llamada *dependencia lectura-escritura*) se usa porque la restricción es similar a la de la dependencia verdadera, pero a la inversa.

Las antidependencias pueden no ser un problema en un procesador escalar con cauce único si las instrucciones se ejecutan en el orden dado por el programa y las operaciones de escritura siempre ocurren después de las operaciones de lectura en el cauce. Sin embargo, son un problema si se cambia la secuencia de ejecución, permitiendo, por ejemplo, que las instrucciones sobrepasen a una instrucción parada.

2.1.2 Políticas de emisión de instrucciones

Normalmente se utiliza el término *emisión de instrucciones* para referirse al proceso de iniciar la ejecución de instrucciones en las unidades funcionales del procesador y el término *política de emisión de instrucciones* para referirse al protocolo usado para emitir instrucciones.

Fundamentalmente el procesador intenta localizar instrucciones, más allá del punto de ejecución en curso, que puedan introducirse en el cauce y ejecutarse. Con respecto a esto son importantes tres ordenaciones:

- El orden en que se captan las instrucciones.
- El orden en que se ejecutan las instrucciones.
- El orden en que las instrucciones alteran los registros y las posiciones de memoria.

La única restricción que tiene el procesador es que el resultado debe ser correcto. Por tanto, el procesador debe ser capaz de acomodar las dependencias y conflictos discutidos antes.

En términos generales, las políticas de emisión de instrucciones en procesadores superescalares se pueden clasificar en las siguientes categorías:

- Emisión en orden y finalización en orden.
- Emisión en orden y finalización desordenada.
- Emisión desordenada y finalización desordenada.

La política de emisión de instrucciones más sencilla es la primera. En ella las instrucciones son emitidas en el orden que lo haría una ejecución secuencial y los resultados son escritos en ese mismo orden. Ni siquiera los cauces escalares siguen esta política, centrándose en la finalización desordenada para mejorar la velocidad de instrucciones que necesitan muchos ciclos. En este caso hay que tener en cuenta las dependencias de salida existentes entre las distintas instrucciones. Nosotros nos centraremos en la implementación de la emisión desordenada de instrucciones.

2.1.3 Implementación de la emisión desordenada

Con la emisión en orden, el procesador sólo decodificará instrucciones hasta el punto de dependencia o conflicto. No se decodifican instrucciones adicionales hasta que el conflicto se resuelve. Por consiguiente, el procesador no puede buscar más allá del punto de conflicto instrucciones que podrían ser independientes. Para mostrar cómo funciona el manejo de la emisión desordenada y las dependencias asociadas, consideraremos el procesador mostrado en la figura 2.3.

Para permitir la emisión desordenada, es necesario desacoplar las etapas del cauce de decodificación y ejecución. Esto se hace mediante un buffer llamado *ventana de instrucciones*. Con esta organización, cuando un procesador termina de decodificar una instrucción, la coloca en la ventana de instrucciones con lo que se indica que está lista para se ejecutada. Las instrucciones son emitidas desde la ventana siempre que sea posible su ejecución, lo cual ocurre cuando los operandos de la instrucción están disponibles y la unidad funcional requerida para la operación está libre. La ventana de instrucción puede implementarse de dos maneras: centralizada o distribuida.

Ventana de instrucciones centralizada

La figura 2.4 muestra la posición de una ventana de instrucciones centralizada. Dicha ventana mantiene todas la instrucciones pendientes sin importar su tipo. Cuando una instrucción tiene sus operandos y puede ser emitida, se encamina a la unidad funcional apropiada para su ejecución. Si un operando no está disponible (todavía no ha sido actualizado por la operación anterior), el bit de valido se podrá a cero. En ese caso, el identificador de registro se almacena en la ventana de instrucción en lugar del valor del operando. Por lo tanto, la ventana de instrucción guardará las instrucciones con los operandos o identificadores de registros como se muestra en la figura 2.5. Cuando una instrucción actualiza el valor de un registro, también será necesario acceder a la ventana de instrucción para actualizar los identificadores de registros por el valor correspondiente.

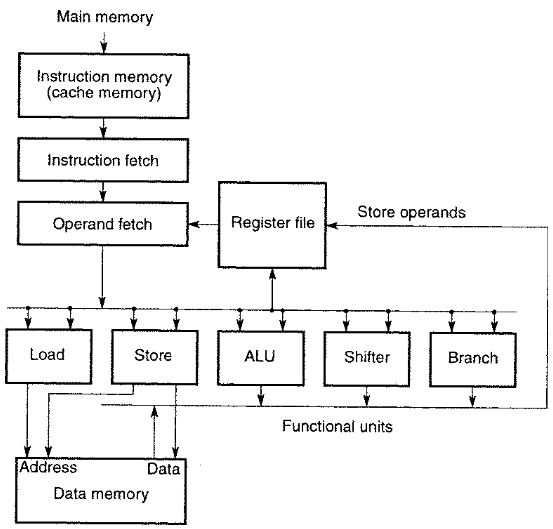

Figura 2.3: Modelo de procesador superescalar.

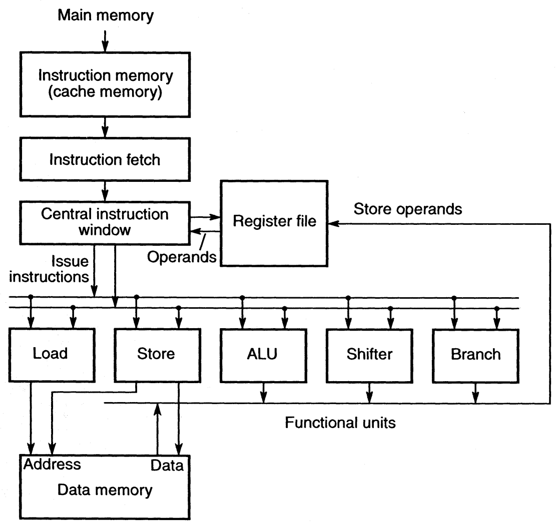

Figura 2.4: Procesador superescalar con ventana de instrucciones centralizada.

Instruction	Opcode	Destination register	Operand 1	Operand 1 register	Operand 2	Operand 2 register
1	Operation	ID	Operand value			ID
2	Operation	ID	Operand value			ID
3	Operation	ID		ID	Operand value	ID
4	Operation	ID	Operand value		Operand value	
5	Operation	ID		ID	Operand value	ID
	⋮	⋮	⋮	⋮	⋮	⋮

Figura 2.5: Contenido de la ventana de instrucciones.

Es posible que más de una instrucción esté lista después de la actualización de un valor, y todas las instrucciones listas deben ser emitidas si ello es posible. La ventana de instrucción centralizada debe también seleccionar una unidad funcional apropiada si existe más de una. Cuando una instrucción ha sido emitida desde la ventana de instrucción, la entrada se libera y puede ser usada por una instrucción entrante, aunque es necesario seguir manteniendo el orden del programa.

Ventana de instrucción distribuida

En la aproximación de ventana de instrucción distribuida, los buffers de instrucción, denominados puestos de reserva (*reservation stations*), se colocan antes de cada unidad funcional. Las instrucciones decodificadas son enviadas al puesto de reserva apropiado y posteriormente emitida a la unidad funcional cuando se reciben todos los operandos. La figura 2.6 muestra la posición de la ventana de instrucción en la aproximación distribuida.

Las conexiones entre la ventana centralizada y las unidades funcionales requiere un mayor ancho de banda que usando una ventana de instrucción distribuida. Sin embargo, una ventana de instrucción centralizada es más eficiente en cuanto al almacenamiento ya que los buffers pueden ser usados por cualquier tipo de instrucción.

El tamaño de cada puesto de reserva no necesita ser el mismo. Cada unidad funcional puede tener un tamaño de puesto de reserva diferente dependiendo del número esperado de instrucciones decodificadas para él.

2.1.4 Renombramiento de registros

Hemos visto que permitir la emisión desordenada de instrucciones y/o la finalización desordenada puede dar origen a dependencias de salida y antidependencias. Estas dependencias son distintas de las dependencias de datos verdaderas y de los conflictos de recursos, que reflejan el flujo de datos a través de un programa y la secuencia de ejecución. Las dependencias de salida y las antidependencias, por otra parte, surgen porque los valores de los registros no pueden reflejar ya la secuencia de valores dictada por el flujo del programa.

Hay un método para hacer frente a estos tipos de conflictos de almacenamiento, que se basa en la duplicación de recursos. En este contexto, la técnica se conoce como *renombramiento de registros*.

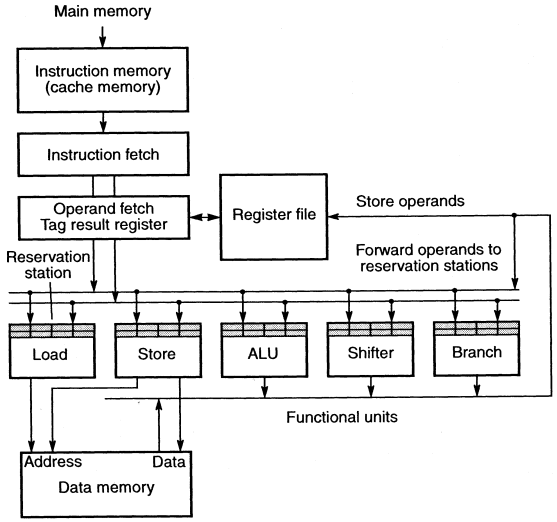

Figura 2.6: Puestos de reserva (sin renombramiento, ver después).

Fundamentalmente, el hardware del procesador asigna dinámicamente los registros, que están asociados con los valores que necesitan las instrucciones en diversos instantes de tiempo. Supongamos la siguiente secuencia:

$$r1 \leftarrow r2 + r4$$
$$r2 \leftarrow r1 + 1$$
$$r1 \leftarrow r4 + r5$$

que tiene varias dependencias. Introduciendo los nuevos registros r1' y r2', tenemos:

$$r1 \leftarrow r2 + r4$$
$$r2' \leftarrow r1 + 1$$
$$r1' \leftarrow r4 + r5$$

lo que elimina la antidependencia y la dependencia de salida.

El renombramiento de registros se puede conseguir mediante un *buffer de reordenación*. Un buffer de reordenación es una cola cuyas entradas se asignan de modo dinámico a los registros resultados de las instrucciones. La posición de dicho buffer es la indicada en la figura 2.7.

Cuando una instrucción que escribe en un registro se decodifica, se le asigna una entrada en el buffer de reordenación[1]. Los resultados de la instrucción se escriben en la entrada correspondiente del buffer

[1] Las instrucciones que no escriben a un registro no necesitan una entrada en dicho buffer ya que no existe dependencias de salida o antidependencias.

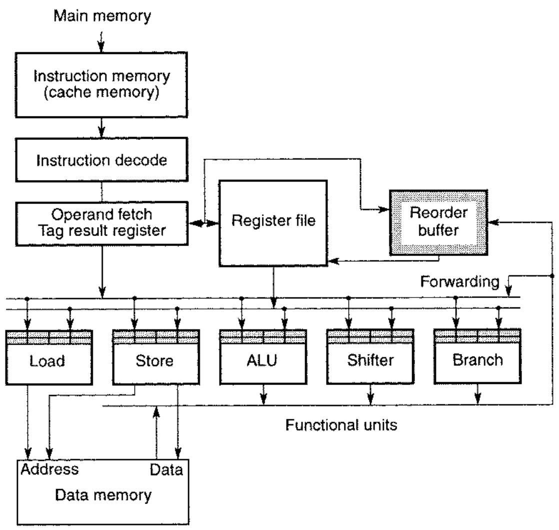

Figura 2.7: Buffer de reordenación con puestos de reserva.

de reordenación. Cuando la entrada alcanza la cabeza de la cola, y el valor ha sido escrito, se transfiere al banco de registros. Si el valor no ha sido escrito en ese momento, la cola no avanza hasta que el valor haya sido escrito. Dado que el buffer de reordenación asocia entradas a registros, el identificador del registro debe almacenarse en las entradas. También se genera una etiqueta única para cada entrada que todavía no ha sido actualizada. El formato de las entradas de un buffer de reordenación podría ser como el que se indica en la figura 2.8.

Cuando se decodifica una instrucción y se necesitan los registros operandos, se realiza una busqueda en el buffer de reordenación. Si no se encuentra ningún registro coincidente, se accede al banco de registros. Si existe alguna entrada en el buffer de reordenación, el valor es leído en el caso de que esté presente y se carga el puesto de reserva correspondiente, en caso contrario se almacena la etiqueta. Cuando el resultado se almacena en el buffer de reordenación reemplazando la etiqueta, también se reemplaza al mismo tiempo las etiquetas por los valores en los puestos de reserva. El algoritmo general para procesar las instrucciones se muestra en la figura 2.9.

2.1.5 Ejemplo de ejecución de un programa en un procesador superescalar

Para terminar con los procesadores superescalares veremos la aplicación de todo lo visto para el programa ejemplo que se muestra en la figura 2.10.

La ejecución de este código en un procesador segmentado sería como la que se muestra en la figura 2.11. En esta ejecución hemos supuesto que las operaciones de lectura y escritura necesitan dos ciclos de reloj y que las operaciones en punto flotante tardan en realizarse un ciclo de reloj.

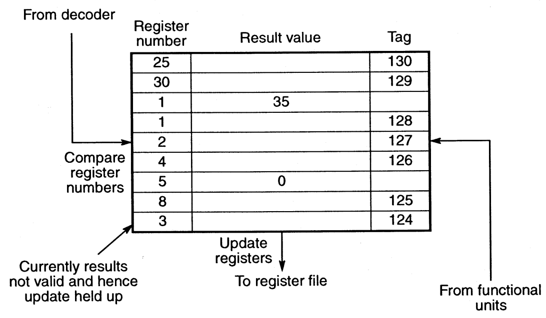

Figura 2.8: Organización del buffer de reordenación.

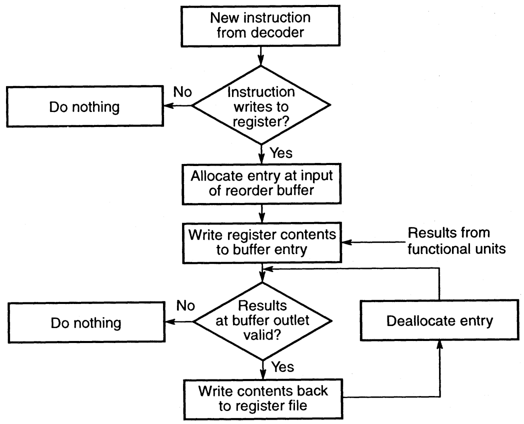

Figura 2.9: Algoritmo de actualización del buffer de reordenación.

```
     Do 10 i=1,looplength
        a(i) = b(i) * x + c(i)
 10  continue


 a1          <- looplength
 s2          <- x            .load constant
 a2          <- 0            .index value
 a4          <- addr(b)      .load
 a5          <- addr(c)      . address
 a6          <- addr(a)      .  pointers
loop: s3     <- mem(a4+a2)   .load b(i)
 s4          <- mem(a5+a2)   .load c(i)
 s7          <- s3 * s2      .b(i)*x
 a1          <- a1 - 1       .decr. counter
 s9          <- s7 + s4      .c(i)+b(i)*x
 mem(a6+a2)<- s9             .store a(i)
 a2          <- a2 + 8       .bump index
 BNEZ        a1,loop         .branch on A1 NEZ
```

Figura 2.10: Código del programa ejemplo y su transcripción a lenguaje ensamblador.

IF	ID	EX	ME	WB
s3<-mem				
s4<-mem	s3<-mem			
s7<-s3*s2	s4<-mem	s3<-mem		
s7<-s3*s2		s4<-mem	s3<-mem	
a1<-a1-1	s7<-s3*s2		s4<-mem	s3<-mem
s9<-s7+s4	a1<-a1-1	s7<-s3*s2		s4<-mem
mem<-s9	s9<-s7+s4	a1<-a1-1	s7<-s3*s2	
a2<-a2+8	mem<-s9	s9<-s7+s4	a1<-a1-1	s7<-s3*s2
	etc.			

Figura 2.11: Ejecución del código en un procesador segmentado.

En el caso de una ejecución en un procesador superescalar, la figura 2.12a muestra el tiempo de captación de las distintas instrucciones, en donde se ha supuesto que es posible captar y decodificar dos instrucciones por ciclo de reloj. El primer paso en la mejora en el tiempo de ejecución se debe a la técnica de predicción de saltos, lo que permite la captación anticipada de instrucciones antes de que se haya determinado la siguiente instrucción a realizar. El resultado de esta optimización se muestra en la parte b de la figura 2.12.

```
time
0    s3           <- mem(a4+a2)
     s4           <- mem(a5+a2)
1    s7           <- s3 * s2
     a1           <- a1 - 1
2    s9           <- s7 + s4
     mem(a6+a2)   <- s9
3    a2           <- a2 + 8
     BNEZ          a1,loop
4    s3           <- mem(a4+a2)
     s4           <- mem(a5+a2)
5    s7           <- s3 * s2
     a1           <- a1 - 1
6    s9           <- s7 + s4
     mem(a6+a2)   <- s9
7    a2           <- a2 + 8
     BNEZ          a1,loop
8    s3           <- mem(a4+a2)
     s4           <- mem(a5+a2)
9    s7           <- s3 * s2
     a1           <- a1 - 1
10   s9           <- s7 + s4
     mem(a6+a2)   <- s9
11   a2           <- a2 + 8
     BNEZ          a1,loop
```

(a)

```
time
0  s3           <- mem(a4+a2)
   s4           <- mem(a5+a2)
1  s7           <- s3 * s2
   a1           <- a1 - 1
2  s9           <- s7 + s4
   mem(a6+a2)   <- s9
3  a2           <- a2 + 8
4  s3           <- mem(a4+a2)
   s4           <- mem(a5+a2)
5  s7           <- s3 * s2
   a1           <- a1 - 1
6  s9           <- s7 + s4
   mem(a6+a2)   <- s9
7  a2           <- a2 + 8
8  s3           <- mem(a4+a2)
   s4           <- mem(a5+a2)
9  s7           <- s3 * s2
   a1           <- a1 - 1
10 s9           <- s7 + s4
   mem(a6+a2)   <- s9
11 a2           <- a2 + 8
```

(b)

Figura 2.12: Código del programa ejemplo y su transcripción a lenguaje ensamblador. (a) Tiempo de captación de las instrucciones. (b) Tiempo de captación después utilizar la técnica de predicción de saltos.

En la figura 2.13a se muestra como queda el código después de producirse el renombramiento de los registros. En la parte b de dicha figura se muestra el instante en el que las distintas instrucciones se copian en los puestos de reserva para ser emitidas a las distintas unidades funcionales en cuanto éstas estén libres. Finalmente, en la figura 2.14 se muestra la ejecución del código suponiendo los siguiente parámetros: 2 ciclos en un acierto de caché, 5 ciclos en un fallo de caché y 3 ciclos para ejecutar las instrucciones en punto flotante.

2.2 Procesadores supersegmentados

Una aproximación alternativa para aumentar el rendimiento del procesador es utilizar más etapas, y de grano más fino, en el cauce. Con más etapas, más instrucciones pueden estar en el cauce al mismo tiempo, aumentando el paralelismo. La reducción en la complejidad de cada etapa permite un aumento en la frecuencia del reloj con lo que es posible conseguir rendimientos similares al diseño

```
time
1   S3          <- mem(A4+A2)
    S4          <- mem(A5+A2)
2   S7          <- s3 * S2
    A6          <- A1 - 1
3   S9          <- S7 + S4
    mem(A6+A2)  <- S9
4   A7          <- A2 + 8
5   S10         <- mem(A4+A7)
    S11         <- mem(A5+A7)
6   S12         <- S10 * S2
    A8          <- A6 - 1
7   S13         <- S12 + S11
    mem(A6+A7)  <- S13
8   A9          <- A7 + 8
9   S14         <- mem(A4+A9)
    S15         <- mem(A5+A9)
10  S16         <- S14 * S2
    A10         <- A8 - 1
11  S17         <- S16 + S15
    mem(A6+A9)  <- S17
12  A11         <- A9 + 8
```

```
time  integer/memory          floating point:
2     S3  <- mem(A4+A2)
      S4  <- mem(A5+A2)
3     A6  <- A1 - 1           S7  <- s3 * S2
4     mem(A6+A2)<- S9         S9  <- S7 + S4
5     A7  <- A2 + 8
6     S10 <- mem(A4+A7)
      S11 <- mem(A5+A7)
7     A8  <- A6 - 1           S12 <- S10 * S2
8     mem(A6+A7)<- S13        S13 <- S12 + S11
9     A9  <- A7 + 8
10    S14 <- mem(A4+A9)
      S15 <- mem(A5+A9)
11    A10 <- A8 - 1           S16 <- S14 * S2
12    mem(A6+A9)<- S17        S17 <- S16 + S15
13    A11 <- A9 + 8
```

Figura 2.13: (a) Situación después del renombramiento de registros. (b) Tiempo de emisión de las distintas instrucciones a los puestos de reserva.

superescalar. Además, esta técnica puede implementar de forma conjunta con técnicas superescalares.

La figura 2.15 compara ambas aproximaciones. La parte superior del diagrama ilustra un cauce normal como base de la comparación. El cauce base emite una instrucción por ciclo de reloj y puede ejecutar una etapa en cada ciclo. La segunda parte muestra una implementación supersegmentada que es capaz de ejecutar dos etapas del cauce por ciclo de reloj. Una forma alternativa de enfocar esto consiste en que las funciones realizadas en cada etapa se pueden dividir en dos partes no solapadas y que cada una se ejecuta en medio ciclo de reloj. Se dice que una implementación de un cauce supersegmentado que se comporta de esta forma es de grado 2. Por último, la parte inferior del diagrama muestra una implementación superescalar capaz de ejecutar dos instrucciones en paralelo dentro de cada etapa.

2.3 Procesadores VLIW

La aproximación VLIW (*Very Long Instruction Word*) consiste en incorporar varias pequeñas operaciones independientes en una palabra de instrucción larga. Haciendo un uso intensivo del compilador para generar estas instrucciones se consigue que sean lo suficientemente grandes para proporcionar, en paralelo, suficientes bits de control sobre varias unidades funcionales. En otras palabras, una arquitectura VLIW proporciona muchas más unidades funcionales que un diseño de procesador típico, junto con un compilador que busca paralelismo a través de operaciones básicas para mantener las unidades funcionales tan ocupadas como sea posible. El compilador compacta código secuencial ordinario en palabras de instrucción largas que hacen mejor uso de los recursos. Durante la ejecución, la unidad de control emite una instrucción larga por ciclo. La instrucción emitida inicia varias operaciones independientes de forma simultánea. En la figura 2.16 se muestra como el compilador agruparía varias instrucciones sencillas para formar una instrucción larga en un procesador VLIW.

Una comparación de las tres aproximaciones vistas en este capítulo muestran algunas diferencias


```
time integer/memory               floating point:
3    S3   <- mem(A4+A2)
          (cache miss)
4    S4   <- mem(A5+A2)
5    A6   <- A1 - 1
6    mem(A6+A2)<- S9
7    A7   <- A2 + 8
8    S10  <- mem(A4+A7)  S7    <- s3 * S2
9    S11  <- mem(A5+A7)
10   A8   <- A6 - 1      S12   <- S10 * S2
11   mem(A6+A7)<- S13    S9    <- S7 + S4
12   A9   <- A7 + 8
13   S14   <- mem(A4+A9)S13   <- S12 + S11
14   S15   <- mem(A5+A9
15   A10   <- A8 - 1     S16   <- S14 * S2
16   mem(A6+A9)<- S17
17   A11   <- A9 + 8
18                       S17   <- S16 + S15
```

(a)

```
time   integer/memory      floating point:   reorder/update
       S3  <- mem(A4+A2)
       S4  <- mem(A5+A2)
       A6  <- A1 - 1
       mem(A6+A2)<- S9
       A7  <- A2 + 8
8      S10 <- mem(A4+A7) S7 <- s3*S2     s3=S3
9      S11 <- mem(A5+A7)                 s4=S4
10     A8  <- A6 - 1     S12<- S10*S2
11     mem(A6+A7)<- S13  S9 <- S7+S4     s7=S7  a1=A6
12     A9  <- A7 + 8
13     S14  <- mem(A4+A9)S13<- S12+S11
14     S15  <- mem(A5+A9)                s9=S9  a2=A7
15     A10  <- A8 - 1    S16<- S14*S2    s3=S10
16     mem(A6+A9)<- S17                  s4=S11
17     A11  <- A9 + 8                    s7=S12  a1=A8
18                       S17<-S16+S15    s9=S13  a2=A9
19                                       s3=S14
20                                       s4=S15
21                                       s7=S16  a1=A10
22                                       s9=S17  a2=A11
```

(b)

Figura 2.14: (a) Ejecución de las instrucciones. (b) Actualización de los registros.

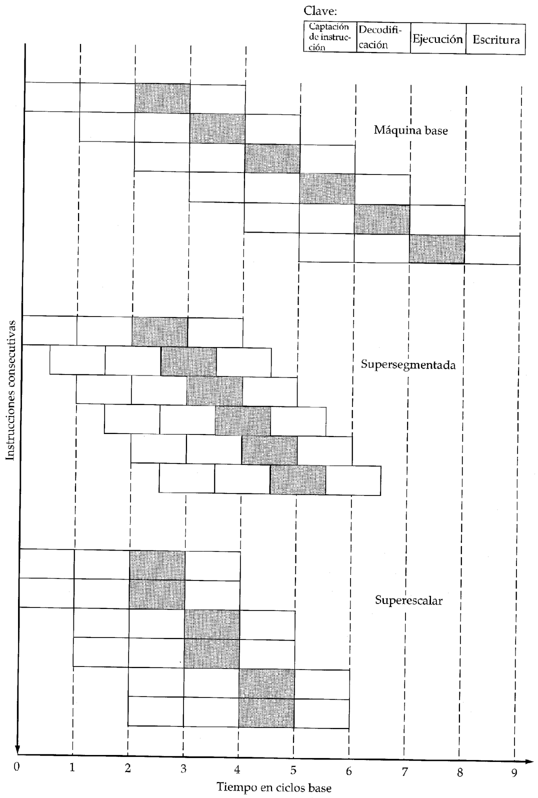

Figura 2.15: Comparación entre el diseño superescalar y supersegmentado.


```
          S3  <- mem(A4)        |
          S5  <- mem(A5)        |
          S13 <- mem(A4+8)      |  S7 <- S3 * S2    |
          S15 <- mem(A5+8)      |                   |
          S23 <- mem(A4+16)     | S17 <- S13 * S2   |
          S25 <- mem(A5+16)     | S9  <- S7  + S5   | A1 <- A1-4
          S33 <- mem(A4+24)     | S27 <- S23 * S2   | A4 <- A4+32
          S35 <- mem(A5+24)     | S19 <- S17 + S15  | A5 <- A5+32
loop:     S3  <- mem(A4)        | S37 <- S33 * S2   | A6 <- A6+32
          S5  <- mem(A5)        | S29 <- S27 + S25|
          mem(A6) <- S19        |                   |
          S13 <- mem(A4+8)      | S7  <- S3  * S2   |
          S15 <- mem(A5+8)      | S39 <- S37 + S35  |
          mem(A6+8) <- S29      |                   |
          S23 <- mem(A4+16)     | S17 <- S13 * S2   |
          S25 <- mem(A5+16)     | S9  <- S7  + S5   |
          mem(A6+16) <- S39     |                   |
          S33 <- mem(A4+24)     | S27 <- S23  * S2  | A4 <- A4+32
          S35 <- mem(A5+24)     | S19 <- S17 + S15  | A5 <- A5+32
          mem(A6+24) <- S9      |                   | BNEZ  A1, loop
```

Figura 2.16: (a) Ejecución de las instrucciones. (b) Actualización de los registros.

interesantes. Por ejemplo, las aproximaciones superescalar y VLIW son más sensibles a conflictos de recursos que la aproximación supersegmentada. En un procesador superescalar o VLIW, un recurso debe duplicarse para reducir la posibilidad de conflictos, mientras que el diseño supersegmentado evita cualquier conflicto de recursos.

Para evitar que el procesador supersegmentado sea más lento que el superescalar, la tecnología usada en la supersegmentación debe reducir el retraso del un cauce de instrucción más largo. Por lo tanto, en general, los diseños supersegmentados requieren una tecnología de transistores más rápida como la GaAs (galio arsénico), mientras que los diseños superescalares usan a menudo tecnología CMOS, dado que dicha tecnología proporciona una buena densidad de integración. Aunque la supersegmentación parece una solución más inmediata que el diseño superescalar, la tecnología existente favorece el aumento en la densidad de integración sobre el aumento en la velocidad del circuito. Históricamente, la densidad ha aumentado a mayor ratio que la velocidad de los transistores. Este precedente histórico sugiere una conclusión general: la aproximación superescalar tiene un coste de implementación más eficiente para la industria.

Los avances tecnológicos han permitido combinar las técnicas superescalar y supersegmentada, proporcionando buenas soluciones a muchos de los problemas de eficiencia existentes en la industria de ordenadores actual. Dichas soluciones, que intentan aprovechar las características positivas de cada diseño, pueden verse en los procesadores existentes actualmente como el microprocesador Alpha AXP 21064.

2.4 Ejemplos de máquinas reales

2.4.1 PowerPC 620

El micro procesador Power PC 620 es una implementación de la arquitectura Power PC de 64-bits; esta implementación incluye la captación y decodificación de cuatro instrucciones por ciclo, la ejecución especulativa y desordenada, etc.

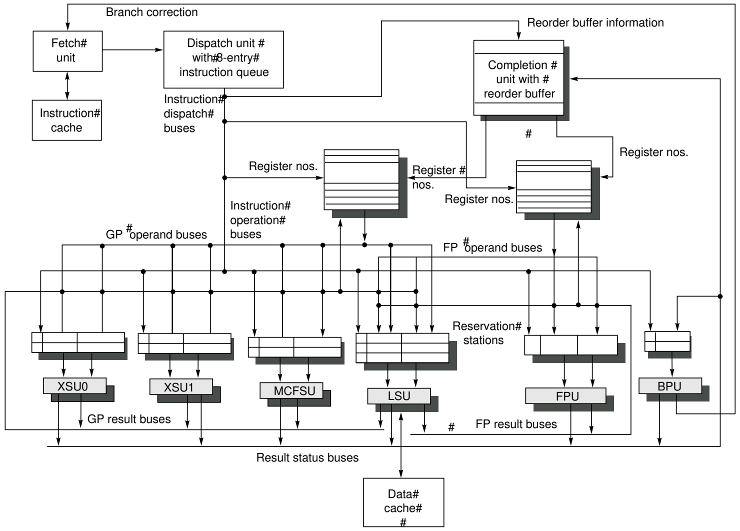

Figura 2.17: Estructura del Power PC 620.

La figura 2.17 muestra la estructura del Power PC 620, muy similar a otros procesadores que proporcionan estas características como el MIPS R1000 y el HP PA 8000. El 620 puede captar, emitir y completar hasta cuatro instrucciones por ciclo. Consta de seis unidades de ejecución:

- Dos unidades de enteros sencillas, XSU0 y XSU1, que se encargan de las operaciones simples, tales como suma, resta y operaciones lógicas elementales. Estas operaciones se realizan en un ciclo de reloj.
- Una unidad de enteros compleja, MCFXU, que se encarga de la multiplicación y división entera. Las operaciones de esta unidad tienen una latencia de 3 a 20 ciclos de reloj y proporciona un mecanismo de disponibilidad anticipada para para valores enteros cortos. Las operaciones de esta unidad varían desde totalmente encauzada (para multiplicaciones con enteros cortos) ha no encauzada (para división entera).
- Una unidad de lectura/escritura, LSU, que se ocupa de las lecturas y escrituras a memoria y que tiene una latencia de ejecución de 1 ciclo para las lecturas de enteros y dos ciclos para las de FP. La LSU está totalmente encauzada y tiene su propio sumador de dirección efectiva. Esta unidad contiene además buffers de lectura y escritura que permiten que las lecturas pasen a las escrituras pendientes chequeando los posibles conflictos de dirección una vez obtenidas las direcciones efectivas de ambas instrucciones.

- Una unidad de punto flotante, FPU, que tiene una latencia de 2 ciclos para la multiplicación, suma o multiplicación-suma, y 31 ciclos de reloj para la división. Esta unidad está totalmente encauzada excepto para la división.
- Una unidad de saltos, BPU, que completa los saltos e informa a la unidad de captación de los fallos en las predicciones.

Las instrucciones fluyen a lo largo de un cauce que varía de cinco a siete ciclos de reloj en los casos típicos y muchas más en operaciones tales como la división, donde no existe encauzamiento. Todas las instrucciones pasan a través de las siguientes etapas:

1. *Captación*. Inserta en la cola de decodificación las instrucciones obtenidas de la caché y determina la dirección de la siguiente instrucción. Consta además de un buffer con 2048 entradas para la predicción de saltos y un buffer de destino del salto con 256 entradas que se utiliza como entrada para predecir la dirección de la siguiente captación.
2. *Decodificación de la instrucción*. Las instrucciones son decodificadas y preparadas para su emisión. Todas las porciones críticas de la decodificación se realizan aquí. Las siguientes cuatro instrucciones se pasan a la siguiente etapa del cauce.
3. *Emisión de instrucciones*. Emite las instrucciones al puesto de reserva apropiada. Los operandos son leídos del fichero de registros en esta etapa, ya sea a la unidad funcional o a los puestos de reserva. Se le asigna un registro en donde se guardará el resultado de la instrucción y una entrada en el buffer de reordenación para asegurar una terminación ordenada.
4. *Ejecución*. Se pasa a este estado cuando todos los operandos están disponibles en el puesto de reserva. Una de las seis unidades funcionales ejecuta la instrucción. Al final de la ejecución, el resultado se escribe en el bus apropiado, y desde allí a cualquier puesto de reserva que esté esperando el resultado al igual que en la entrada reservada por la instrucción en el buffer de reordenación. La unidad de terminación es informada de que la instrucción ha finalizado. Si la instrucción era un salto, y se falló en la predicción, se le notifica este hecho a las unidades de captación y de terminación, con lo que la unidad de captación comienza de nuevo en la dirección correcta, haciendo que la unidad de terminación descarte las instrucciones especulativas y libere las entradas asociadas en el buffer de reordenación.
5. *Finalización (Commit)*. Ocurre cuando todas las instrucciones anteriores se han comprometido. Pueden finalizar hasta cuatro instrucciones por ciclo. Los resultados del buffer de renombramiento se escriben en el fichero de registros y se libera la entrada en el buffer. Cuando finaliza una instrucción de almacenamiento, también se le notifica el hecho a la LSU, para que la correspondiente entrada en el buffer de almacenamiento pueda ser enviada a la caché. A esta etapa también se le denomina completación de la instrucción (*instruction completion*).

La figura 2.18 muestra la estructura básica del cauce del Power PC 620 y como las etapas están conectadas mediante buffers, permitiendo el desacoplamiento de las etapas. Cuando se completa una instrucción, toda la información sobre esa instrucción se elimina y el resultado de la misma se escribe en el estado visible de la arquitectura (registros, PC o memoria).

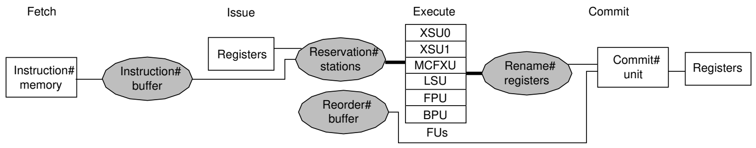

Figura 2.18: Los estados del cauce del Power PC 620.

Rendimiento del cauce del Power PC 620

En este apartado estudiaremos el rendimiento de l Power PC 620 utilizando siete programas del banco de pruebas SPEC92: compress, eqntott, espresso, li, alvinn, hydro2d y tomcatv.

Para determinar el rendimiento del 620, nos centraremos en el *throughput* de la etapa de emisión como la medida crítica del rendimiento. Centrarse en la emisión tiene sentido por dos razones. En primer lugar, es una buena medida del rendimiento en un estado estacionario, dado que cuando el procesador se encuentra en equilibrio las instrucciones no pueden emitirse más rápidamente de lo que son ejecutadas o comprometidas. En segundo lugar, en la etapa de emisión se encuentran algunos de los cuellos de botella claves que son comunes a muchas máquinas con planificación dinámica de instrucciones. Aunque nos centraremos en la etapa de emisión, tanto la etapa de captación como la de ejecución afectan al rendimiento de la emisión de instrucciones ya que son responsables de llenar los buffers de entrada y vaciar los buffers de salida, respectivamente, de la etapa de emisión. La figura 2.19 muestra una visión del rendimiento del cauce, mostrando la diferencia entre el IPC ideal (4) y el real (1.2-1.3).

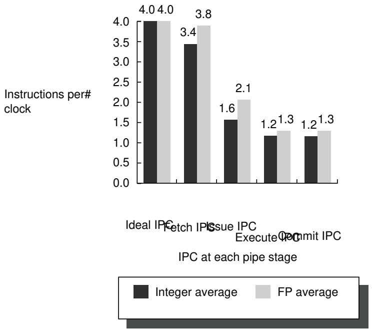

Figura 2.19: Visión general del rendimiento del cauce del Power PC 620.

Rendimiento de la etapa de captación

La etapa de captación de instrucciones es capaz de captar hasta cuatro instrucciones por ciclo e introducirlas en el buffer de instrucciones de ocho entradas. Esta etapa se puede convertir en un cuello de botella si no es capaz de mantener al menos cuatro instrucciones en el buffer de instrucción.

Por término medio, la etapa de captación es capaz de mantener 5.2 instrucciones en el buffer, no limitando el rendimiento. La captación limita el rendimiento cuando no es capaz de proporcionar cuatro instrucciones para su emisión. Existen tres circunstancias bajo las cuales la unidad de captación puede fallar en su misión de mantener el buffer de instrucciones lleno:

1. *Un fallo en la predicción de un salto*. No se añaden instrucciones *útiles* al buffer; la unidad de captación está parada. En realidad, se añaden instrucciones al buffer, pero dado que las instrucciones provienen del camino equivocado, cuentan como una parada, tratándose como si no se hubiesen puesto instrucciones en el buffer.

2. *Un fallo en la caché de instrucciones*. No se añaden instrucciones al buffer; la unidad de captación está totalmente parada.

3. *Línea de caché parcialmente llena*. El siguiente grupo de cuatro instrucciones cruza una línea de caché, y únicamente se captan las instrucciones de la misma línea de caché, que tiene una longitud de 8 palabras.

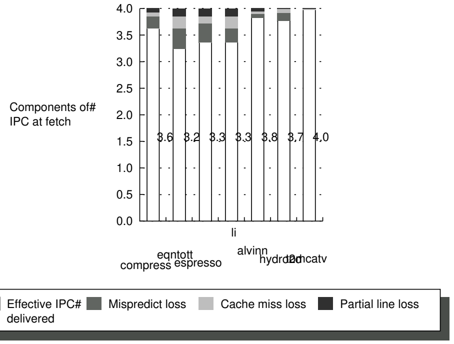

Figura 2.20: Número medio de instrucciones captadas por el Power PC 620.

La figura 2.20 muestra la contribución de estos factores a la pérdida de rendimiento pico. La pérdida en los programas enteros es del 15% (0.6 de 4.0), mientras que en los programas de punto flotante es del 5% (0.2 de 4.0).

Emisión de instrucciones

La emisión de instrucciones intenta enviar cuatro instrucciones a las unidades de reserva en cada ciclo de reloj. Las razones principales por las que puede no conseguirse esta meta son dos. En primer lugar, existen limitaciones en la etapa de emisión de instrucciones que hacen que ciertas combinaciones de instrucciones no puedan emitirse simultáneamente. En segundo lugar, la falta de progreso en las etapas de ejecución y finalización dan lugar a una falta de buffers necesarios para emitir una instrucción. Dado que las instrucciones se emiten ordenadamente, estos conflictos pueden hacer que el número medio de emisiones caiga rápidamente como se muestra en la tabla 2.1.

Prob. de no emisión = (1-p)	Prob. de emisión = p	Número medio de instr. emitidas
0.1	0.9	3.1
0.2	0.8	2.4
0.3	0.7	1.8
0.4	0.6	1.3
0.5	0.5	0.9

Tabla 2.1: Número de instrucciones emitidas para un total de cuatro posibles.

Existen cinco posibles causas de conflicto a la hora de emitir una instrucción:

1. No existe un puesto de reserva disponible.
2. No hay registros disponibles.
3. El buffer de reordenación está lleno.
4. Existe dos operaciones para la misma unidad funcional.
5. Otros conflictos.

Los tres primeros tipos de conflictos se deben a que las etapas de ejecución y finalización no han procesado las instrucciones que fueron previamente emitidas; los dos últimos conflictos se deben a limitaciones internas de la implementación. La figura 2.21 muestra la reducción en el IPC debida a estas causas.

La tabla 2.2 muestra los mismos datos que la figura 2.21 pero organizados para mostrar los eventos responsables de que se emitan menos de cuatro instrucciones por ciclo. Es posible que varios eventos ocurran de manera simultánea para una instrucción dada en un instante determinado. En los datos de la tabla 2.2 se ha supuesto que la causa a la que se le asocia la parada es el primer evento que produce la parada en el orden mostrado anteriormente.

Rendimiento de la etapa de ejecución

Una vez que las instrucciones han sido emitidas, deben esperar hasta que la unidad funcional y los operandos están disponibles. Existen seis clases diferentes de unidades funcionales, lo que permite hasta 6 iniciaciones por ciclo. Las causas principales de retraso en esta etapa son:

1. *Operando fuente no disponible*. Uno de los operandos fuentes todavía no está disponible.

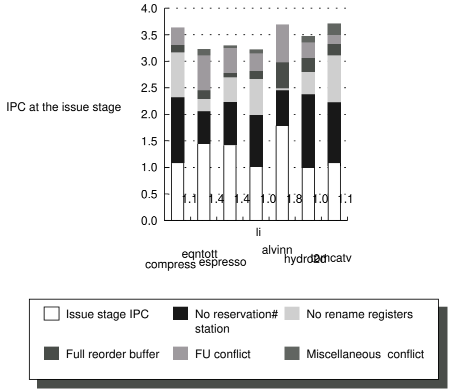

Figura 2.21: IPC obtenido para la etapa de emisión del Power PC 620.

Benchmark	Sin paradas	Parada				Parada		
		no res. stat.	no ren. buff.	no reorder buff.	total	2 inst. a FU	misc.	total
compress	24%	36%	24%	6%	66%	10%	0%	10%
eqntott	41%	22%	8%	4%	34%	21%	4%	25
espresso	33%	32%	14%	2%	48%	18%	1%	19
li	31%	34%	17%	4%	55%	11%	3%	14
alvinn	31%	23%	1%	21%	45%	24%	0%	24
hydro2d	17%	43%	17%	8%	68%	12%	3%	15
tomcatv	6%	37%	34%	9%	80%	7%	7%	14%
Media Ent.	32%	31%	16%	4%	51%	15%	2%	17%
Media FP	34%	28%	10%	8%	46%	19%	2%	21%
Media Total	28%	33%	12%	9%	54%	16%	2%	18%

Tabla 2.2: Causas de las paradas en la unidad de emisión del Power PC 620.

2. *Unidad funcional no disponible*. Otra instrucción está utilizando la unidad funcional. Para unidades totalmente encauzadas, esto ocurre únicamente cuando dos instrucciones están disponibles para iniciarse en el mismo ciclo de reloj, pero para unidades no encauzadas (multiplicación entera/división, división en punto flotante), la unidad funcional estará bloqueada hasta que finalice la operación.

3. *Iniciación fuera de orden no permitida*. La unidad de salto y la de punto flotante necesitan que las instrucciones se inicien ordenadamente. En consecuencia, una instrucción puede quedar bloqueada hasta que las instrucciones que le preceden se inicien. Esta es una limitación de la unidad de ejecución.

4. *Serialización*. Unas pocas instrucciones requieren una ejecución totalmente ordenada. Por ejemplo, las instrucciones que acceden a registros especiales que no son renombrables deben ejecutarse ordenadamente. Estas instrucciones esperan en el puesto de reserva hasta que las instrucciones anteriores finalicen.

La figura 2.22 muestra como puede atribuirse las paradas en la etapa de emisión por el llenado de buffers a una falta de progreso en la unidad de ejecución debido a alguno de los cuatro componentes anteriormente indicados.

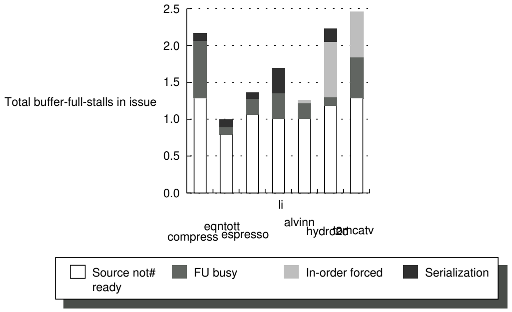

Figura 2.22: Causas de las paradas en la etapa de emisión atribuibles a la unidad de ejecución del Power PC 620.

Cuando la etapa de emisión se bloquea y una instrucción en un puesto de reserva no puede ejecutarse por alguno de los cuatro motivos arriba indicados, un diseñador puede contemplar una de las tres siguientes razones:

1. Si el operando fuente no esta disponible y la emisión se detiene porque los buffers están llenos, esto indica que el paralelismo existente es insuficiente para el tamaño de ventana dado por los buffers. Si el código tiene mas paralelismo, entonces menos puestos de reserva deberán esperar otros resultados, dando lugar a más iniciaciones y a una necesidad de menos buffers. Alternativamente, el diseñador puede aumentar el número de buffers, dando lugar a una mayor ventana y a un posible incremento del ILP.

2. Si la instrucción no se ha iniciado porque la unidad funcional está en uso, el problema básico es que la capacidad de las unidades funcionales no es suficiente para manejar la distribución dinámica de instrucciones, al menos en ese punto de ejecución. La solución en este caso es incrementar el número de unidades funcionales o incrementar la segmentación de aquellas que no lo son.

3. Las dos últimas causas de no iniciación de la ejecución (ejecución fuera de orden no permitida y serialización) son debidas a elecciones de implementación de la etapa de ejecución; y pueden ser eliminadas reorganizando la unidad de ejecución.

Rendimiento de la finalización de instrucciones

La completación de instrucciones puede detenerse únicamente cuando la instrucción a la cabeza del buffer de reordenación no ha terminado su ejecución. Este parada puede dar lugar a que se llene el buffer de reordenación, lo que haría que se bloquease la emisión de instrucciones. Las paradas de esta etapa son muy infrecuentes: en promedio, la ejecución se detiene siete veces más frecuentemente para los programas de punto flotante y 100 veces más frecuentemente para los programa enteros. Como resultado, esta etapa no es un cuello de botella.

Resumen: Rendimiento total

De las figuras anteriores, podemos determinar que el IPC para estos programas de pruebas van desde el 1 hasta el 1.8. Esta diferencia entre el IPC ideal y el real se debe a tres factores:

1. *Limitaciones debidas a las unidades funcionales*. Esta limitación se debe a que el 620 no tiene cuatro copias de cada unidad funcional. Para estos programas de prueba el cuello de botella está en la unidad de carga y almacenamiento.

2. *Pérdidas en etapas específicas*. Las etapas de captación, emisión y ejecución presentan pérdidas asociadas específicamente a esa etapa.

3. *ILP limitado y buffers finitos*. Paradas que se producen por la falta de paralelismo o insuficientes buffers. También se incluyen aquí los fallos de caché que dan lugar a ciclos de parada.

La figura muestra el IPC ideal de 4 dividido entre el IPC real (1.0 a 1.8) y los diferentes tipos de parada.

2.4.2 MIPS R10000

El microprocesador R1000 es un ejemplo de arquitectura superescalar que permite la captación y decodificación de cuatro instrucciones por ciclo. Las instrucciones son captadas y decodificadas de forma ordenada, añadiéndose a una de las tres colas de instrucción, pero se permite la emisión desordenada de las mismas con lo que se pueden llegar a tener en un momento dado 32 instrucciones en varios estados de ejecución. En la figura 2.24 se muestra un diagrama del R10000.

El microprocesador R10000 tiene tres colas de instrucción. Estas colas pueden emitir instrucciones hacia las diferentes unidades de ejecución en cuanto los recursos estén disponibles. Cada cola utiliza

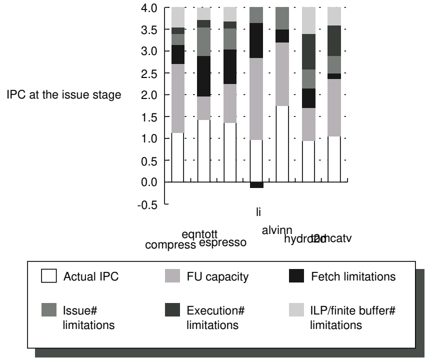

Figura 2.23: División del IPC ideal de 4.0 en sus componentes.

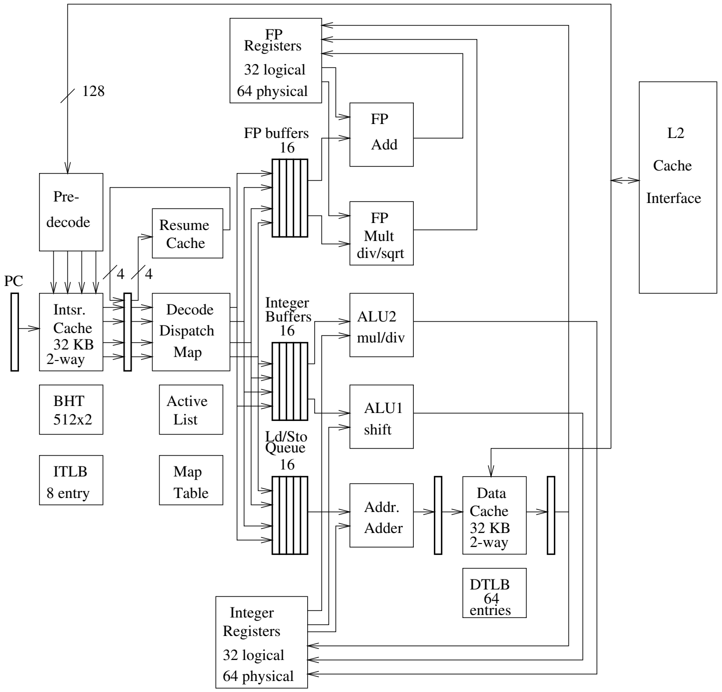

Figura 2.24: Diagrama de bloques del microprocesador R10000.

etiquetas en las instrucciones para seguir el rastro de las instrucciones en cada etapa del cauce de ejecución.

La cola de instrucciones de enteros consta de 16 entradas y emite las instrucciones a las dos ALUs. En cada ciclo es posible escribir hasta cuatro instrucciones de enteros en las entradas libres que existan en esta cola, en la cual permanecerán hasta que sean emitidas a una ALU.

La cola de punto flotante consta de 16 entradas que pueden ser emitidas al sumador o multiplicador de punto flotante. Al igual que en el caso anterior, el ritmo de entradas en esta cola es de hasta cuatro instrucciones por ciclo.

En la cola de direcciones se almacenan instrucciones de lectura/escritura. Al igual que las otras dos colas, consta de 16 entradas organizada como un buffer circular FIFO. Las instrucciones pueden ser emitidas en cualquier orden, pero deben escribirse y eliminarse de la cola en orden secuencial. La FIFO mantiene la secuencia de instrucciones del programa original de tal manera que las dependencias entre instrucciones de memoria sean sencillas de manejar. Una instrucción emitida puede no completarse debido a una dependencia de memoria, un fallo de caché o un conflicto de recursos. En estos casos la cola debe volver a emitir la instrucción hasta que esta se complete.

El microprocesador R10000 permite la eliminación de dependencias entre las instrucciones mediante la utilización del renombramiento de registros. Cuando se *renombra* un registro, el registro lógico que se referencia en la instrucción se liga a un registro físico usando una tabla de mapeo. La asignación de un registro lógico a un nuevo registro físico se realiza siempre que el registro sea el registro destino de una instrucción. Así, cuando una instrucción pone un valor nuevo en un registro lógico, el registro lógico se renombra para usar un nuevo registro físico. Sin embargo, el valor anterior permanece en el antiguo registro físico. El esquema implementado en el R10000 consta de una tabla de mapeo, una lista de activos y una lista de libres como se muestra en la figura 2.25.

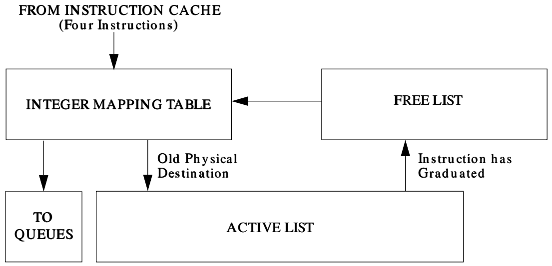

Figura 2.25: Diagrama de bloques del esquema de mapeo de las instrucciones enteras.

El microprocesador R10000 contiene 64 registros físicos. En un momento dado el valor del registro físico se encuentra en alguna de las listas mencionadas anteriormente. La lista de activos mantiene una lista de las 32 instrucciones que en un momento dado pueden estar en el cauce. Esta lista siempre está ordenada. Las instrucciones de las colas pueden ejecutarse de forma desordenada pero los resultados deben almacenarse en el orden determinado por la lista activa. Una vez que el valor se almacena se convierte en obsoleto y el registro lógico vuelve a la lista de libres.

Cada instrucción puede ser identificada de forma única por su localización en la lista activa. Un valor de 5 bits sirve como etiqueta de cada instrucción y permite localizar dentro de la lista de instrucciones activas a aquellas instrucciones que han finalizado su ejecución.

Cuando se elige un registro de la lista de libres se pasa a la tabla de mapeo y esta es actualiza. El valor de ese registro particular será ahora el valor actual de un operando. El valor antiguo que estaba en la tabla de mapeo se pone en la lista activa. El valor permanece en dicha lista hasta que la instrucción finaliza en el orden indicado en el programa. Una instrucción se considera finalizada según el orden del programa si todas las instrucciones anteriores también han finalizado. Una vez que la instrucción finaliza, los valores anteriores se descartan.

El microprocesador R10000 consta de 64 registros físicos y 32 registros lógicos. La lista activa contiene un máximo de 32 valores. La lista de libres puede también contener un máximo de 32 valores. Si la lista activa esta llena podrían existir 32 valores comprometidos y 32 valores temporales, de ahí la necesidad de 64 registros.

Instruction	Latency	Repeat Rate
Integer Add, Subtract, Logical Op's, branches	1	1
Integer Load/ Store (primary cache hit)	2	1
Integer Multiply (single precision)	5 (Lo) - 6 (Hi)	6
Integer Multiply (double precision)	9 (Lo) -10 (Hi)	10
Integer Divide (single precision)	34 (Lo) - 35 (Hi)	35
Integer Divide (double precision)	66 (Lo) - 67 (Hi)	67
Integer to FP conversion (single precision)	4	1
Floating Point Add, Subtract, Conversion, Logical operations	21	1
Floating Point Load/ Store	3	1
Floating Point Multiply (double precision)	2	1
Floating Point Multiply-Add	2/ 4	1
Floating Point Divide (single precision)	12	14
Floating Point Divide (double precision)	19	21
Floating Point Square Root (single precision)	18	20
Floating Point Square Root (double precision)	33	35
Floating Point Reciprocal Square Root (single prec.)	30	20
Floating Point Reciprocal Square Root (double prec.)	52	35

Figura 2.26: Latencia de las instrucciones y ratios de repetición.

En la figura 2.26 se muestra la latencia de las distintas instrucciones en el R10000, así como cuántos ciclos de reloj es necesario esperar para poder introducir otra instrucción en el *pipeline*.