Apéndice 1.3 Mejora de prestaciones de los programas en Procesadores Segmentados

Existen una serie de técnicas que permiten mejorar el comportamiento de los programas en una arquitectura determinada. Algunas de estas técnicas se pueden aplicar a los programas escritos en lenguaje de alto nivel, y otras a la hora de generar el código ejecutable. En este último caso, el compilador es el encargado de aplicar estas técnicas de optimización, cuya extensión puede seleccionar el usuario a partir de las correspondientes opciones de compilación (-O1, -O2, ... en gcc, por ejemplo). Sin embargo, en algunos casos, el compilador no es capaz de descubrir la posibilidad de utilizar ciertas transformaciones. Entonces, el análisis de los programas en ensamblador generados por el compilador, puede permitir a un programador experimentado y conocedor de la arquitectura del procesador, obtener los niveles de prestaciones requeridos. En este Apéndice presentaremos algunas de las optimizaciones posibles a nivel de lenguaje ensamblador, de cara a mejorar el rendimiento de los programas en procesadores segmentados.

En concreto, consideraremos el desenrollado de bucles (Sección A.1.3.1), la reducción de dependencias de datos mediante la reorganización de código (Sección A.1.3.2), y la segmentación software (Sección A.1.3.3).

A.1.3.1 Desenrollado de Bucles

Esta técnica consiste en reducir el número de iteraciones de un bucle aumentando el número de instrucciones que se realizan en cada iteración. La mejora en el tiempo de ejecución que se consigue con esta técnica se debe a las siguientes razones:

- Se reduce el número de instrucciones de actualización del índice del bucle, de comprobación de final del bucle, y de salto condicional. En procesadores segmentados, la reducción de las instrucciones de salto condicional reduce los riesgos de control y evita las posibles penalizaciones que éstas instrucciones ocasionan en un procesador segmentado si no se predice bien el resultado del salto. En algunos procesadores siempre que hay una instrucción de salto se pierde un ciclo, ya que siempre se aborta la instrucción que se capta a continuación de la de salto (debe ser una instrucción que no tenga efecto en el programa: por ejemplo, una instrucción nop) .
- Al aumentar el número de instrucciones en cada iteración del bucle, se facilita la aplicación de técnicas de reorganización del código para evitar dependencias (Sección A.1.3.2).

Un ejemplo de desenrollado de bucle se muestra en la Figura A.1.3.1 para el caso de un bucle escrito en lenguaje de alto nivel.

Por otra parte, en la Figura1.6 se muestra el programa DLX *inic2.s* obtenido al desenrollar el programa *inic1.s* de la Figura 1.3. En lugar de tener que realizar seis iteraciones, el programa *inic2.s* realiza sólo tres. Mientras que *inic1.s* necesitaba 223 ciclos para ejecutarse en la configuración de procesador seleccionada, *inic2.s* necesitaba 211. Como se puede comprobar, al ejecutarse más instrucciones por iteración (se dobla en el caso de *inic2.s*), el tamaño del código aumenta (necesita más memoria para almacenarse), aunque el número de instrucciones que ejecuta el procesador disminuye: en *inic1.s* se ejecutan 88 instrucciones y en *inic2.s* se ejecutan 82. Esta situación tendría

que tenerse en cuenta desde el punto de vista de la inclusión de todas las intrucciones de cada iteración en cache o no. Como se puede comprobar, en *inic1.s* se tiene que CPI= 223/88 = 2.53 ciclos por instrucción, mientras que en *inic2.s* el valor de CPI es mayor (CPI = 2.57), a pesar de que las mejores prestaciones se obtienen con *inic2.s*.

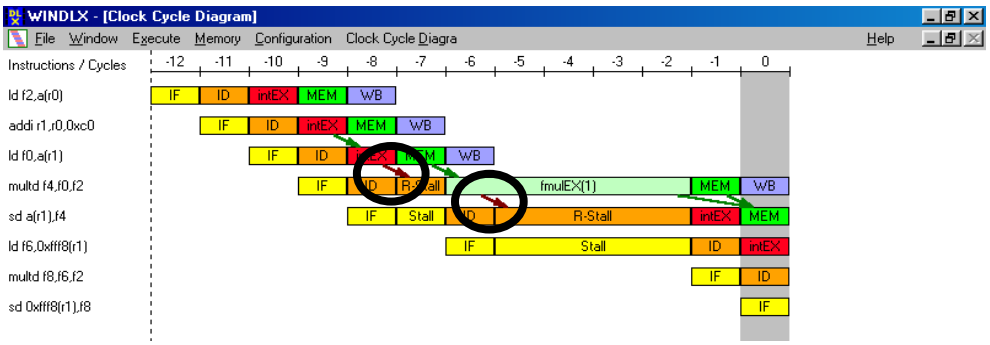
```
for (i=1; i<=N; i++)
{
      A[i] = A[i] + B[i];          BUCLE SIN DESENROLLAR
}


for (i=1; i<=N; i+4)
{
      A[i] = A[i] + B[i];          BUCLE DESENROLLADO
      A[i+1] = A[i+1] + B[i+1];
      A[i+2] = A[i+2] + B[i+2];
      A[i+3] = A[i+3] + B[i+3];
}
```

Figura A.1.3.1 Desenrollado de bucle

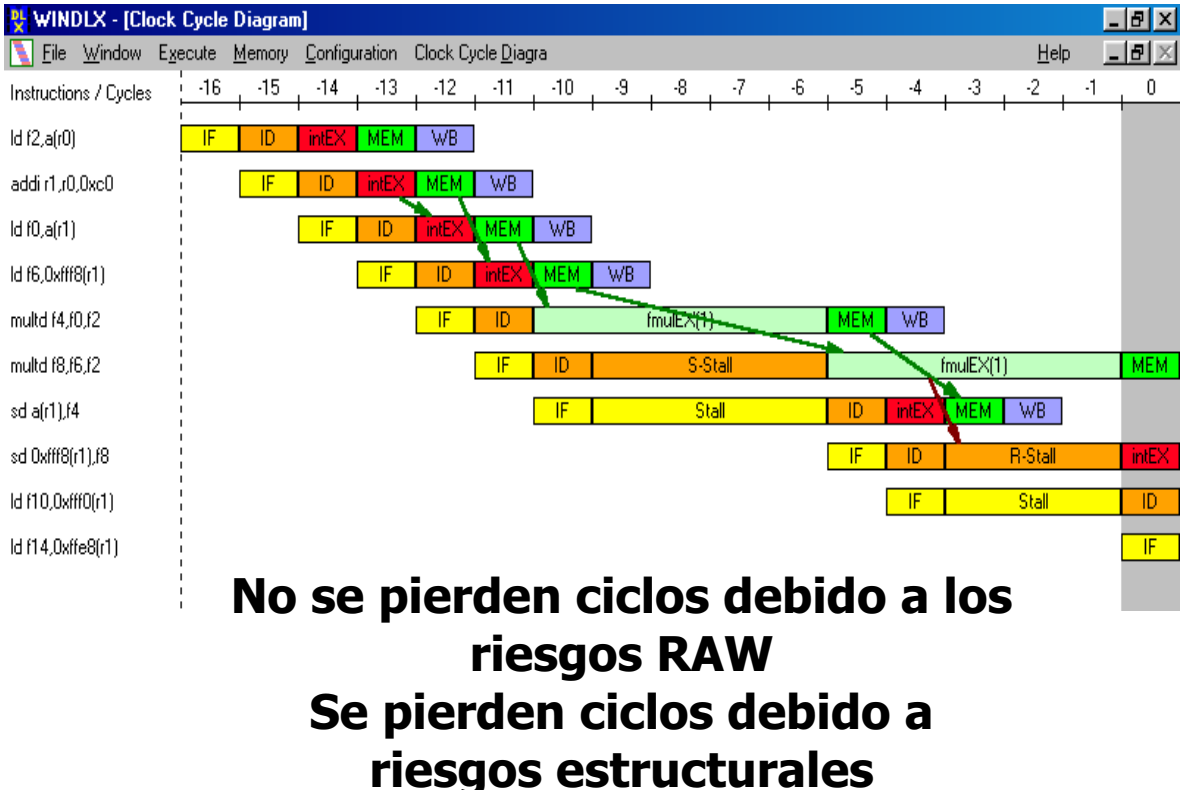
A.1.3.2 Reorganización de Código

Esta técnica permite reducir el número de ciclos de espera en el cauce (atascos o *stalls*) debido a dependencias de datos, saltos, o dependencias estructurales. Consiste en cambiar el orden en que las instrucciones aparecen en el código del programa, y por tanto el orden en que se ejecutan en el procesador segmentado, sin afectar a los resultados que proporciona dicho programa.



Dependencias de tipo RAW que ocasionan pérdidas de ciclos

Figura A.1.3.2. Ejecución de código (*inic1.s*) con WinDLX mostrando dependencias



No se pierden ciclos debido a los riesgos RAW
Se pierden ciclos debido a riesgos estructurales

Figura A.1.3.3 Ejecución de código (*inic3.s*) en WinDLX mostrando dependencias

En la figura A.1.3.2 se muestra la ejecución de un trozo del programa *inic1.s* en el que se producen pérdidas de ciclos debido a las dependencias RAW entre la instrucción ld f0,a(r1) y multd f4,f0,f2 a causa de f0, y entre multd f4,f0,f2 y sd f4,a(r1) a causa de f4. Para evitar estas pérdidas de ciclos, en la Figura A.1.3.3 se muestra como se retarda la ejecución de multd f4,f0,f2 con respecto a ld f0,a(r1), introduciendo entre ambas la instrucción ld f6,0xfff8(r1), y se retarda sd f4,a(r1) con respecto a multd f4,f0,f2 introduciendo entre ambas la instrucción multd f8,f6,f2. De esta forma, también se consigue que, gracias a multd f4,f0,f2, la instrucción multd f8,f6,f2 se retrase respecto a ld f6,0xfff8(r1), ya que hay un riesgo RAW entre ambas debido a f6.

No obstante, hay que tener en cuenta que, con el cambio realizado, pueden tener efecto (como así se observa en la Figura A.1.3.3) los riesgos estructurales entre las instrucciones de multiplicación multd f4,f0,f2 y multd f8,f6,f2, si sólo hay un multiplicador, y éste consume más de un ciclo (como se supone en el caso del ejemplo).

A.1.3.3. Segmentación Software

Esta técnica persigue eliminar las dependencias entre las instrucciones que se ejecutan en las iteraciones de un bucle. Para ello se reorganizan las instrucciones del bucle de forma que las secuencias de instrucciones (con dependencias RAW):

carga (i) / operación (i) / almacenamiento (i)

que existan en una iteración i, se distribuyen en iteraciones distintas. Por ejemplo, en la iteración i se produciría:

almacenamiento (i) / operación (i+1) / carga (i+2)

El efecto de las dependencias se evita, por tanto, reordenando la situación de las instrucciones en el código para introducir instrucciones entre las que presenten dependencias. A continuación se muestra un ejemplo más explícito de como funcionaría la técnica, a partir del código:

```
bucle:    ld    f0,0(r1)
          addd  f4,f0,f2
          sd    0(r1),f4
          subi  r1,r1,#8
          bnez  r1, bucle
```

En la Figura A.1.3.4 se muestra el efecto de aplicar la segmentación software al bucle anterior

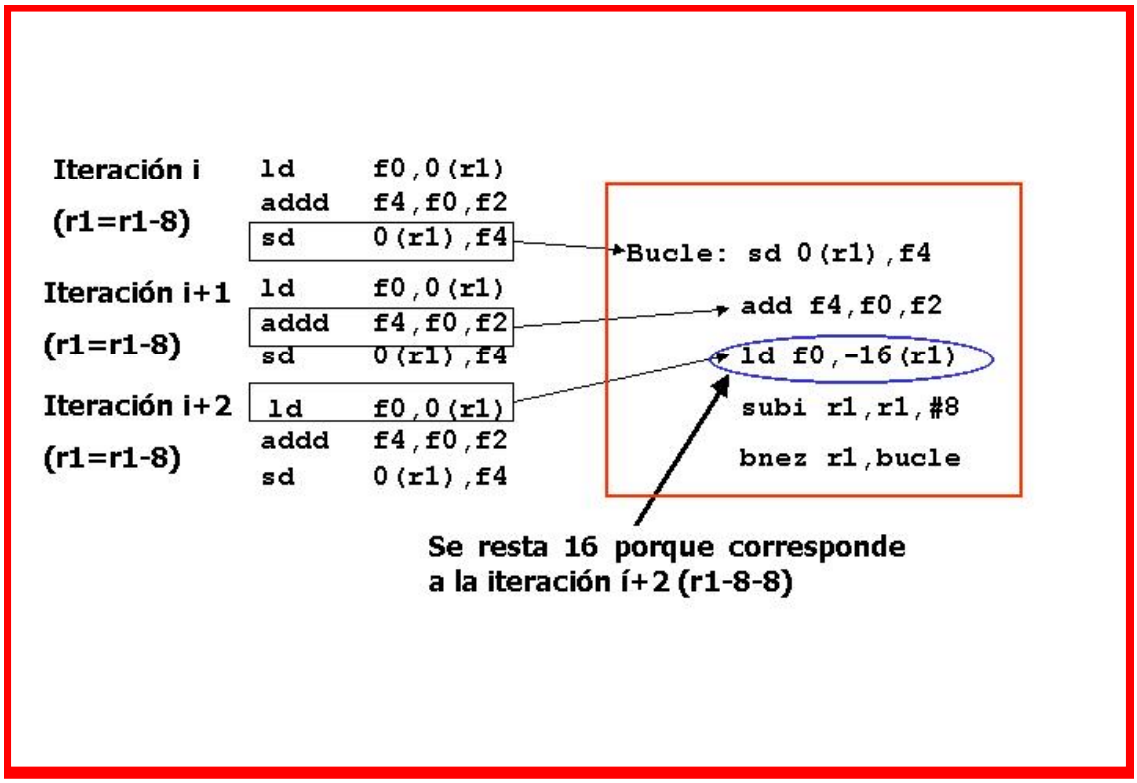


Figura A.1.3.4 Aplicación de la técnica de segmentación software

Como se puede observar en la Figura A.1.3.4, en la iteración i del bucle resultante, se almacena el valor del registro f4 en la posición a la que apunta r1; se realiza la suma de los registros f0 y f2 en f4, teniendo en cuenta que en la iteración anterior (i-1) se había cargado el valor de f0 con el correspondiente a dos iteraciones posteriores (es decir i-1+2= i+1); y se carga en f0 el dato correspondiente a dos iteraciones posteriores (i+2). El valor de r1 se sigue decrementando de 8 en 8 correspondiente al paso de la iteración i a la i+1. Como se puede ver, en el código final no hay riesgos de dependencias de datos de tipo RAW. Obviamente, para que el programa funcione correctamente, hay que añadir las operaciones que quedarían fuera de las iteraciones del bucle, correspondientes a las primeras iteraciones.

En la Figura A.1.3.5 se muestra el programa *inic5.s,* que utiliza segmentación software para mejorar las prestaciones del programa inic1.s descrito anteriormente. En la figura, las instrucciones (3) – (6) corresponden a las instrucciones de carga de los registros para las multiplicaciones de la primera iteración (instrucciones (17)-(20)), a las que se pasa con la instrucción de salto (7). Después se inician las siguientes iteraciones, almacenando mediante las instrucciones (8)-(11), y se modifica r1 restándole 32, mediante la instrucción (12) para que apunte a los datos a cargar para la iteración siguiente. Hay que restar 32 debido a que se realizan 4 operaciones y cada una utiliza datos de 8 bytes (4x8=32). Finalmente, las instrucciones (23) – (26) corresponden a las instrucciones de almacenamiento de la última iteración.

De esta forma, en la iteración i se almacenan los datos de la iteración i-1 y se cargan y multiplican los de la iteración i. No hace falta utilizar almacenamiento, multiplicación y carga de tres iteraciones distintas (i-2, i-1, i) debido a que, al haber varias instrucciones de multiplicación y varias de carga, se pueden evitar las dependencias entre cargas y multiplicaciones con dependencias RAW juntando todas las de multiplicación y todas las de carga. No quiere esto decir que sea la mejor solución, puesto que pueden ocasionarse pérdida de ciclos debido a los riesgos estructurales.

```
      .text   256

start:ld      f2,a                    (1)
      add     r1,r0,xtop              (2)
      ld      f0,0(r1)                (3)
      ld      f6,-8(r1)               (4)
      ld      f10,-16(r1)             (5)
      ld      f14,-24(r1)             (6)
      j       input                   (7)

loop:sd       0(r1),f4                (8)
      sd      -8(r1),f8               (9)
      sd      -16(r1),f12             (10)
      sd      -24(r1),f16             (11)
      sub     r1,r1,#32               (12)
      ld      f0,0(r1)                (13)
      ld      f6,-8(r1)               (14)
      ld      f10,-16(r1)             (15)
      ld      f14,-24(r1)             (16)

input:multd   f4,f0,f2                (17)
      multd   f8,f6,f2                (18)
      multd   f12,f10,f2              (19)
      multd   f16,f14,f2              (20)

      sub     r4,r1,#32               (21)
      bnez    r4,loop                 (22)

      sd      0(r1),f4                (23)
      sd      -8(r1),f8               (24)
      sd      -16(r1),f12             (25)
      sd      -24(r1),f16             (26)

      nop                             (27)
      trap    #0                      (28)
```

Figura A.1.3.5 Código del programa *inic5* donde se aplica segmentación software

Referencias

[GRU92] Grünbacher, H.: “WinDLX V1.2. A DLX-Simulator for MS-Windows”. Vienna University of Technology. Enero, 1992.

[HEN96] Hennesy, J.L.; Patterson, D.A.:”Computer Architecture. A Quantitative Approach” (Segunda Edición). Morgan Kaufmann Pub. Inc., 1996.