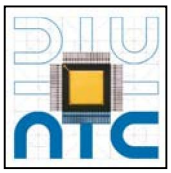
PRÁCTICAS DE PROCESADORES SEGMENTADOS con WinDLX

Julio Ortega Lopera
Departamento de Arquitectura y Tecnología de Computadores



PRÁCTICAS DE PROCESADORES SEGMENTADOS CON WinDLX

INTRODUCCIÓN

El objetivo de estas prácticas es el estudio experimental de un procesador segmentado para ilustrar la mejora de prestaciones que se puede conseguir con un cauce segmentado, analizando las dificultades y las limitaciones existentes para alcanzar dicha mejora. Para realizar el trabajo experimental correspondiente a esta práctica se utilizará el simulador WinDLX.

WinDLX ha sido desarrollado en la Universidad de Viena [GRU92], y permite simular el procesador segmentado DLX en el sistema Windows de MicroSoft. Posee un interfaz bastante amigable y permite analizar los problemas que plantea la ejecución, en el cauce segmentado del procesador, de un programa escrito en el lenguaje ensamblador DLX. Para ello, el simulador representa las dependencias que existen entre las instrucciones, y ofrece información acerca del número de ciclos que ha necesitado la ejecución del procesador, los ciclos que se han perdido por dependencias (de datos, de control, o estructurales) entre instrucciones, etc.

En esta introducción, se describirá en primer lugar el procesador segmentado DLX, para pasar a describir a continuación las características y opciones del simulador. Finalmente, se proporcionarán algunos ejemplos de su utilización.

1.1 El procesador segmentado DLX

El procesador DLX [HEN96] implementa un repertorio de instrucciones de tipo RISC, que presenta características típicas de los repertorios de procesadores recientes como MIPS, Power PC, PA (*Precision Architecture*), y SPARC. En el Apéndice 1.1 se proporcionan algunos detalles del repertorio DLX. En la Figura 1.1 se muestran las cinco etapas que constituyen el cauce del procesador DLX:

- **IF** (Captación de Instrucción, *Instruction Fetch*). Cada ciclo de reloj se capta una instrucción de memoria que se introduce en el registro de instrucción (IR). El contador de programa (PC) se incrementa de forma que apunta a la siguiente instrucción.
- **ID** (Decodificación de Instrucción, *Instruction Decode*). Se decodifica la instrucción almacenada en el registro IR, también se accede en esta etapa al banco de registros para cargar los operandos que se utilizarán en la etapa de ejecución en los registros A y B sobre los que actúan las unidades de la etapa de ejecución, EX (que se explica a continuación). En esta etapa ID también se procesan los saltos, tal y como veremos, para reducir el costo asociado a los riesgos de control.
- **EX** (Ejecución, *Execution*). Se ejecuta la operación correspondiente en la unidad funcional seleccionada. En la etapa EX existen varias unidades funcionales que pueden necesitar más de un ciclo de reloj para completar su operación. Las unidades funcionales incluidas en esta etapa son: **intEX** (realiza las operaciones de aritmética entera excepto la multiplicación y la división, y también realiza el cálculo de las direcciones en el acceso a memoria y los saltos), **faddEX** (implementa la suma y la resta de números en coma flotante en simple o doble precisión), **fmulEX** (implementa la multiplicación de números en coma flotante en simple y doble precisión y de enteros, con y sin signo), **fdivEX** (implementa la división de números en coma flotante en simple y doble precisión y de

enteros, con y sin signo). El simulador permite fijar el número de unidades funcionales **faddEX**, **fmulEX**, y **fdivEX** y el número de ciclos que requieren cada una de ellas.

- **MEM** (Acceso a Memoria). Sólo las instrucciones de carga (*load*) o escritura en memoria (*store*) están activas en esta etapa, en la que se realiza la lectura de los datos de memoria (en el caso de las cargas) o la escritura de los datos en memoria (en el caso de las escrituras). Las direcciones de memoria utilizadas por las instrucciones se calculan en el ciclo previo (en la unidad de ejecución para enteros intEX).
- **WB** (Escritura de Resultados, *Write Back*). Los resultados, que provienen de la memoria o de la ALU (etapa EX), se escriben en el banco de registros del procesador. Las operaciones correspondientes a esta etapa se realizan en la primera mitad del ciclo de reloj para que la instrucción que se está ejecutando en la etapa ID pueda leer los mismos datos en la segunda mitad del ciclo de reloj en el caso de que los necesitase. De esta manera se evita tener que incluir un camino de *bypass* para evitar al penalización asociada a una dependencia de datos de tipo RAW.

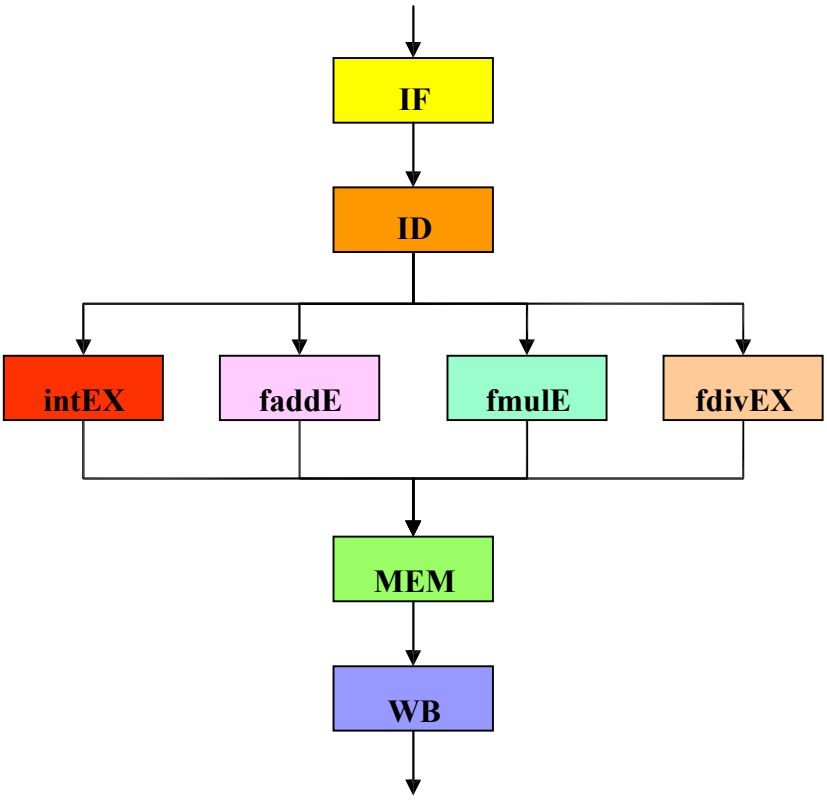


Figura 1.1 Estructura de etapas del cauce del Procesador DLX

Para establecer comparaciones con los resultados que se obtendrían al ejecutar un programa DLX en un procesador no segmentado, pero que utilizase una tecnología similar (frecuencias de reloj, y ciclos en las unidades funcionales de la ALU –etapa EX- similares en los dos procesadores), se definirá un procesador que denominaremos procesador DLX sin segmentar de referencia (**ssrDLX**). En este procesador cada instrucción van pasando por distintas fases que se suceden una tras otra según sean utilizadas por la correspondiente instrucción. Así, una instrucción que no accede a

memoria no tiene que pasar por la fase de acceso a memoria (MEM). De esta forma las instrucciones tienen distinto tiempo de ejecución según sean de un tipo u otro.

En el procesador ssrDLX, consideraremos que las distintas fases tienen los siguientes tiempos:

- **IF, intEX,** y **MEM** duran un ciclo de reloj cada una (con esto se está despreciando el retardo asociado a los registros de desacoplo en el procesador segmentado).
- **ID** y **WB** duran un **80%** del tiempo de ciclo de reloj.
- **faddEX, fmulEX,** y **fdivEX** duran el mismo número de ciclos que en el caso del procesador segmentado.

De esta forma, una instrucción de suma de enteros (operandos y resultado en el banco de registros) necesitará 3.6 ciclos en el procesador sin segmentar ya que pasa por las fases de captación (IF), decodificación y lectura de operandos (ID), ejecución en la unidad de enteros (intEX) y escritura del resultado en el banco de registros (WB). En cambio, en el caso del procesador segmentado necesitará 5 ciclos puesto que tiene que pasar por todas las etapas (IF, ID, intEX, MEM, WB), y todas duran un ciclo en el cauce segmentado.

1.2 El Simulador WinDLX

En esta sección se indicarán las posibilidades que ofrece el simulador WinDLX y la forma de acceder a las misma. En general, el simulador WinDLX permite simular (por completo o paso a paso) programas escritos en el ensamblador DLX, visualizando la forma en que las instrucciones se están ejecutando en las distintas etapas del cauce y las dependencias entre instrucciones. Mediante WinDLX es posible cambiar algunas de las características de alguna de las etapas del cauce, y tener información de prestaciones ya que proporciona el número de ciclos que ha tardado la ejecución del programa, los ciclos que se han perdido por dependencias entre instrucciones (*stalls*), etc.

Al ejecutar el simulador aparece el Menú Principal de WinDLX, en la línea superior de la ventana, y una serie de iconos correspondientes a subventanas minimizadas dentro de la propia ventana principal. El Menú Principal ofrece las siguientes opciones:

- **File**. Permite borrar e inicializar simulaciones (opciones ***Reset DLX*** y ***Reset All***), cargar los ficheros con los programas en ensamblador y datos (opción ***Load Code or Data***) y salir del programa (***Quit WINDLX***). En WinDLX se pueden cargar varios ficheros con programas en ensamblador, que se sitúan en memoria por orden alfabético. No obstante, el lenguaje ensamblador (Apéndice 1.1) proporciona una serie de directivas que afectan a la forma de cargar los programas en memoria.
- **Window**. Es la misma que en cualquier aplicación Windows. Permite abrir y modificar las subventanas, que también se pueden manejar directamente sobre los correspondientes iconos de la ventana principal.
- **Execution**. Permite controlar la ejecución de los programas por parte del simulador y mostrar la ventana de E/S de WinDLX donde aparecen los mensajes correspondientes a la simulación (opción ***Display DLX-I/O***). Los programas se pueden ejecutar hasta el final (opción ***Run***), o hasta un Breakpoint que se haya insertado (opción ***Run to***), ciclo a ciclo (opción ***Single Cycle***), un número de ciclos prefijado (opción ***Multiple Cycles***).
- **Memory**. Permite crear ventanas para visualizar el contenido de la memoria (opción ***Display***), ver y cambiar el contenido de ciertas direcciones de memoria (opción ***Change***) y manipular los símbolos de las variables (opción ***Symbols***).
- **Configuration**. Permite cambiar ciertas características del cauce, concretamente el número de unidades funcionales y sus latencias (opción ***Floating Point Stages***) y la

posibilidad de utilizar caminos de *bypass* o no (opción ***Enable Forwarding***); el tamaño de la memoria (opción ***Memory Size***); y otras características propias del simulador como la utilización de nombres simbólicos o no (opción ***Symbolic Addresses***), la utilización de tiempos absolutos o relativos al ciclo de reloj actual (opción ***Absolute Cycle Count***), el almacenamiento de las características de la configuración en un fichero –con la extensión por defecto *.wdc* - (opción ***Store***), o el establecimiento de una configuración almacenada en un fichero (opción ***Load***).

- **Help**. Permite acceder a una ayuda en inglés bastante completa del simulador WinDLX, las características del cauce, el repertorio de instrucciones, etc.

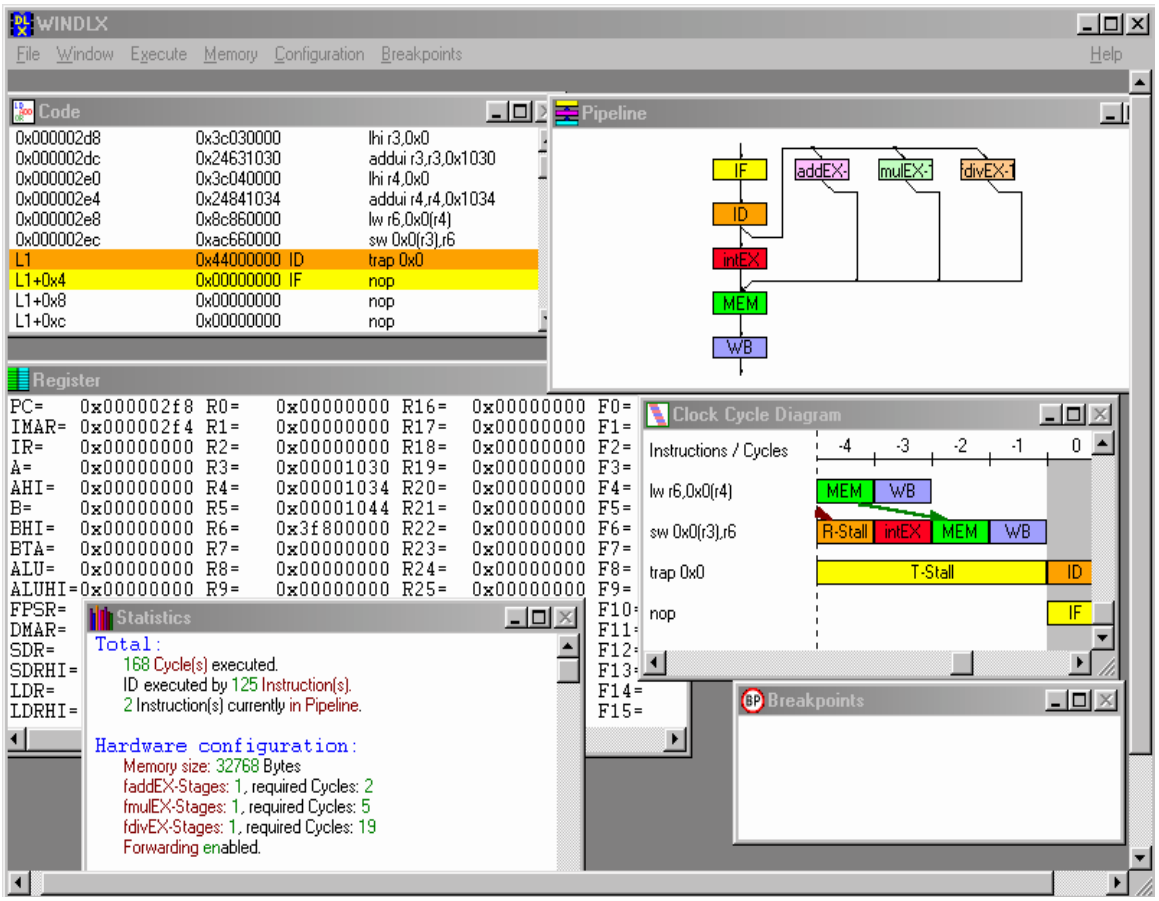


Figura 1.2. Ventana principal de WinDLX con subventanas abiertas

En la ventana principal existen una serie de iconos que permiten abrir una serie de subventanas con información acerca de diversos aspectos de la simulación. A continuación se describen estas subventanas:

- **Venta de Registros (Register)**. Permite ver los contenidos de los registros del procesador. En el Apéndice 1.1 se describen los registros que tiene el procesador DLX y su utilidad.
- **Ventana de Código (Code)**. Permite ver las instrucciones del programa DLX que se ha cargado en memoria. Una descripción del repertorio de instrucciones de DLX, las directivas del ensamblador, etc. se encuentra en el Apéndice 1.1.

- **Ventana de Cauce (Pipeline)**. Visualiza las etapas del cauce con las instrucción que se está ejecutando en cada una (si la ventana se abre con el suficiente tamaño).
- **Ventana de Ciclos de Reloj (Clock Cycle Diagram)**. Permite comprobar lo que se está realizando en cada ciclo y en cada etapa. También muestra las dependencias entre instrucciones, ocasionen detenciones (*stalls*) en el cauce (flechas rojas) o no (flechas verdes).
- **Ventana de Estadísticas (Statistics)**. Proporciona los datos de la ejecución del progrmas: número de ciclos, configuración del cauce con la que se ha hecho la simulación, ciclos perdidos y las causas de esas detenciones (*stalls*), saltos, etc.
- **Ventana de Breakpoint (Breakpoint)**. Permite insertar, borrar, o ver los *Breakpoints*.

La Figura 1.2 muestra la ventana principal de WinDLX incluyendo las seis subventanas desplegadas y las opciones del simulador en el menú principal, situado en la parte superior de la ventana.

1.3 Ejemplo de utilización de WinDLX

A continuación, se ilustra el uso del simulador a través de un programa de ejemplo que se muestra en la Figura 1.3. También se estudian las prestaciones que ofrece el cauce del procesador DLX frente al procesador sin cauce ssrDLX, y se ilustran posibles mejoras del código.

```
            .data   0
            .global a
a:          .double 3.14159265358979

            .global x
x:          .double 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
            .double 17,18,19,20,21,22,23

            .global xtop
xtop:       .double 24
            .text   256

start:      ld      f2,a            (1)
            add     r1,r0,xtop      (2)
loop:       ld      f0,0(r1)        (3)
            multd   f4,f0,f2        (4)
            sd      0(r1),f4        (5)
            ld      f6,-8(r1)       (6)
            multd   f8,f6,f2        (7)
            sd      -8(r1),f8       (8)
            ld      f10,-16(r1)     (9)
            multd   f12,f10,f2      (10)
            sd      -16(r1),f12     (11)
            ld      f14,-24(r1)     (12)
            multd   f16,f14,f2      (13)
            sd      -24(r1),f16     (14)
            sub     r1,r1,#32       (15)
            bnez    r1,loop         (16)
            nop                     (17)
            trap    #0              (18)
```

Figura 1.3 Programa ejemplo (*inic1.s*) en ensamblador de DLX

Supondremos que el programa en ensamblador de la Figura 1.3 se encuentra en el fichero *inic1.s*. Para hacer referencia al programa lo llamaremos *inic1*. En primer lugar analizaremos qué hace este programa.

Análisis del código de *inic1*

La primitiva **.data 0** indica que los datos se pondrán a partir de la posición de memoria **0x0**. A partir de esa posición se encuentra una variable de tipo doble (64 bits = 8 bytes a partir de la posición 0x0), que se designará con la etiqueta **a** (al utilizar **.global** se declara la etiqueta para que puedan utilizarla las instrucciones). A partir de la posición **a** se almacena el número *pi* como un número en coma flotante de doble precisión (ocupando 8 bytes).

Después se declara la etiqueta **x**, que hace referencia a la dirección inmediatamente posterior a la última ocupada por el dato almacenado a partir de la posición **a**. A partir de la dirección a la que apunta **x** se almacenan 23 números (precisamente del 1 al 23) declarados también como de tipo **.double**. Eso significa que la lista ocupa 184 bytes (8*23) y que, si sumamos los 8 bytes del número *pi*, la zona de datos ocupa, por ahora 192 bytes.

A continuación se declara la etiqueta **xtop** que apunta al byte 192 (el primer byte ocupado es el 0). Ahí es donde se almacena otro número, también de doble precisión. Este número es igual a 24, y es precisamente el último número de la lista de números almacenada.

Tenemos por tanto 200 bytes de datos. Los 8 primeros bytes están ocupados por el número *pi*, que empieza en la dirección cuya estiqueta es **a**. Luego se tiene una lista de 24 números, también de 8 bytes cada uno, y en la que la etiqueta **x** corresponde a la dirección del primer número, y la etiqueta **xtop** al último de la lista.

Después está la directiva **.text** con la dirección 256 (**0x100**). Por tanto el código del programa se situará a partir de dicha dirección.

La instrucción (1) carga el par de registros **(f2,f3)** con 8 bytes (ya que es una instrucción **ld**), que están almacenados a partir de la dirección de memoria cuya etiqueta es **a**. Por lo tanto en (f2,f3) se almacena el número *pi*. Después, la instrucción (2), que es una instrucción de suma de registro y dato inmediato, suma la dirección **xtop** al contenido del registro **r0**, que siempre es cero. Por lo tanto, el efecto de esta instrucción es cargar en **r1** la dirección **xtop**, y con ello hacer que r1 apunte a la dirección donde empieza el último número de la lista.

La instrucción (3) está marcada con la etiqueta **loop**. A partir de aquí empieza un bucle, que se extiende hasta la instrucción (16), que comprueba si el valor del registro r1 es igual a 0. Mientras r1 no sea igual a 0 se producirá un salto a loop, y se realizará otra iteración del bucle.

Dentro del bucle hay 4 secuencias de tres instrucciones consecutivas ((3)(4)(5); (6)(7)(8); (9)(10)(11); (12)(13)(14)), cada una de las cuales hace lo mismo. Por ejemplo, la instrucción (3) carga en **(f0,f1)** el dato al que apunta **r1**; la (6) carga en **(f6,f7)** el dato al que apunta **r1-8** (el dato anterior en la lista); la (9) carga en **(f10,f11)** el dato al que apunta **r1-16** (el dato anterior en la lista); y la (12) carga en **(f14,f15)** el dato al que apunta **r1-28** (el dato anterior en la lista). Después de cada una de las anteriores, respectivamente, la instrucción (4) multiplica el dato de la lista que ha leído por el número *pi*, almacenado en **(f2,f3)** y pone el resultado en **(f4,f5)**, y las instrucciones (7), (10) y (13) hacen lo mismo, pero introduciendo los resultados en **(f8,f9), (f12,f13),** y **(f16,f17)**, respectivamente. A continuación de cada una de ellas, las instrucciones (8), (11), y (14) almacenan los contenidos de los registros que tienen los resultados obtenidos por sus respectivas predecesoras. Cada resultado se almacena en la posición

de memoria de la que se había leído el dato utilizado en la multiplicación: **r1, r1-8, r1-16,** y **r1-24.**

Después la instrucción (15) carga **r1** con **r1-32** para que en la siguiente iteración empiece apuntando al número anterior al último que se ha multiplicado en la iteración previa: en cada iteración se multiplican 4 datos de 8 bytes (4x8=32 bytes). Cuando r1 se haga 0, después de completar 6 iteraciones (6x4=24 datos), la instrucción (16) no dará lugar a un salto y el programa termina. La instrucción (17) es una instrucción que no hace ninguna operación (veremos su utilidad posteriormente) y la instrucción (18) es la excepción que pasa el control al sistema y se utiliza para terminar los programas.

Por tanto el programa se encarga de multiplicar todos los números de la lista almacenada por el número *pi*, dejando los resultados almacenados en las mismas posiciones.

Simulación de inic1

A continuación se describe un posible proceso de simulación. Usualmente, en el proceso de simulación y análisis de resultados se pasará por las siuientes etapas:

(1) Cargar el fichero donde se encuentra el programa a simular.

Se selecciona **File** en el menú principal y después, en la ventana que se abre se selecciona **Load Code or Data**. A continuación se escribe el nombre del fichero o bien se selecciona con el cursor entre los ficheros .s que están en el directorio de trabajo. Después se selecciona **Select** y después **Load**.

Tanto si el fichero se carga sin problemas como si hay algún error aparece una ventana indicándolo. Para visualizar el código cargado se puede desplegar la ventana **Code** donde aparecen tres columnas (Figura 1.4). En la de la izquierda está la dirección de memoria donde se carga la instrucción que aparece en la columna de la derecha (en ensamblador). En el centro, aparecen (en hexadecimal) los contenidos de los 32 bits (4 bytes) que ocupa la instrucción.

(2) Fijar la configuración que se quiere simular para el procesador.

A continuación se puede comprobar si la configuración del procesador es la adecuada para la simulación que se quiere hacer o no. Para ello se selecciona la opción **Configuration**, y aparecerán una serie de alternativas.

Mediante **Floating Point Stages** se puede cambiar el número de sumadores, multiplicadores y divisores de coma flotante que hay en el cauce y también se pueden modificar sus retardos. El número de unidades de cada tipo puede ser 8 como máximo y 1 como mínimo, y los retardos pueden ir desde 1 a 50. Los valores que aparecen por defecto corresponden a un sumador con dos ciclos de retardo, un multiplicador con cinco ciclos de retardo, y un divisor con diecinueve ciclos de retardo. Para la simulación que vamos a realizar dejaremos los valores por defecto.

Con **Memory Size** se puede cambiar la cantidad de memoria (entre 512 Bytes y 16 MBytes). Dejaremos también la que hay por defecto (32 KBytes).

También hay tres opciones que se pueden activar o desactivar. Si **Symbolic Addresses** está activa en la ventana de código, las direcciones de memoria que aparecen en la columna de la izquierda se expresan con sus nombre simbólicos (interesa de cara a relacionar el cargado con el listado del fichero .s correspondiente), como en la Figura 1.4. Si no está activa las direcciones se expresan en hexadecimal.

Mediante **Absolute Cycle Count** se puede seleccionar que, en la ventana **Clock Cycle Count** aparezca el número de ciclos desde que empezó la simulación (si la opción está activa) o el número de ciclos relativo al ciclo que se está simulando.

Con **Enable Forwarding** activa se considera en la simulación que el procesador tiene caminos de *bypass* que evitan ciclos de espera en el cauce cuando hay ciertas dependencias entre instrucciones que están en el cauce.

La opción **Store** permite guardar la configuración que se ha seleccionado en un fichero. Esa configuración se puede cargar luego mediante la opción **Load** y seleccionando el fichero en cuestión para que se cargue.

(3) *Asegurarse que está seleccionada la primera instrucción que se quiere simular.*
Antes de empezar a simular hay que asegurarse que está seleccionada la instrucción con la que se quiere empezar la ejecución del programa. Esta instrucción se puede fijar seleccionándola sobre la ventana de código. En la Figura 1.4 está seleccionada la primera instrucción del programa, cuya dirección es precisamente **text**

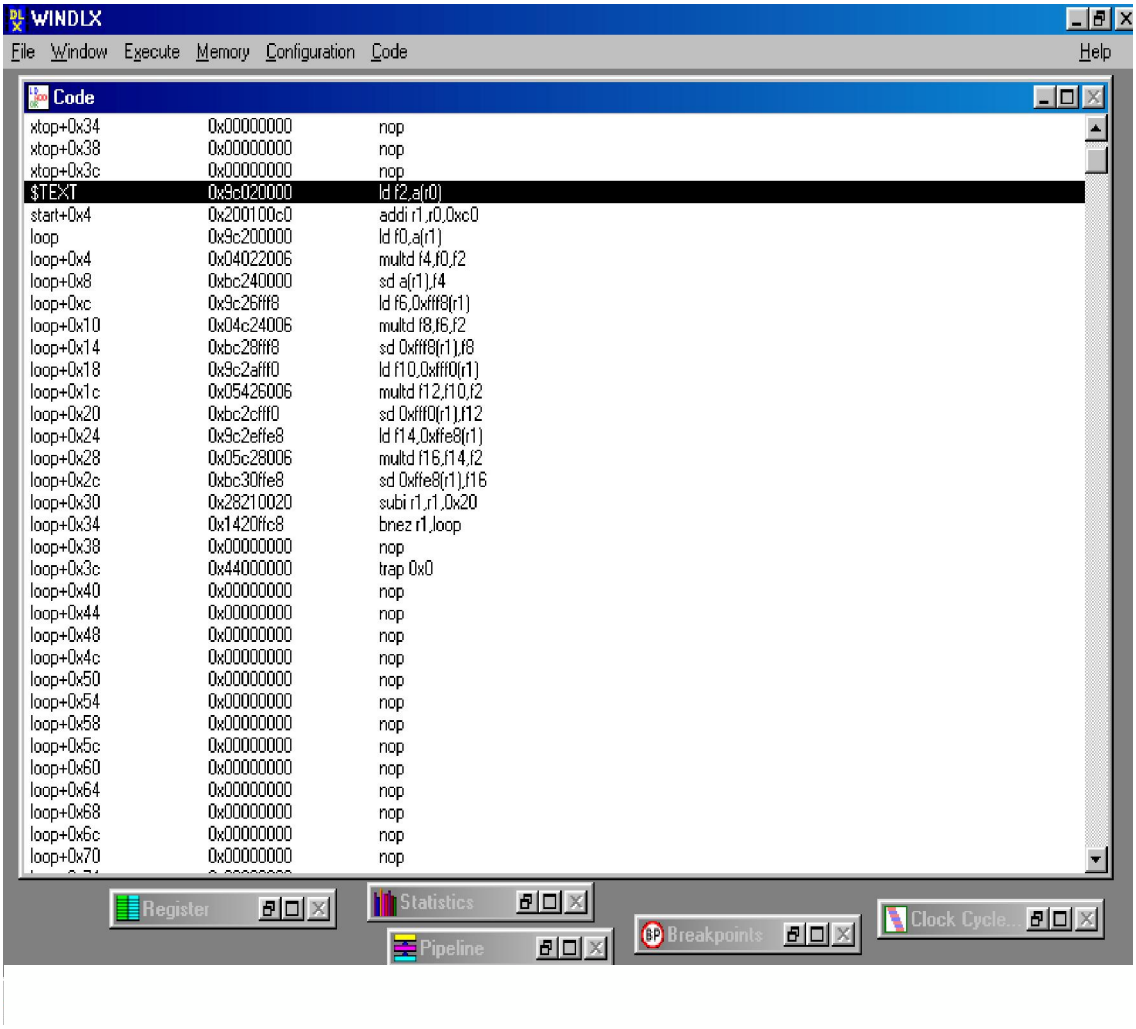


Figura 1.4 Ventana de código desplegada con el programa inic1 cargado y una instrucción seleccionada para empezar la simulación a partir de ella

(4) Realizar la simulación

Para elegir el tipo de simulación que se quiere realizar se utiliza la opción **Execute**. Se puede realizar una ejecución completa del programa (seleccionando **Run**); una simulación hasta que una instrucción que se indica se esté ejecutando en una etapa que también se puede indicar (seleccionando **Run to**); la simulación de un ciclo de reloj a partir del estado presente (seleccionando **Single Cycle**); o el número de ciclos que se desee a partir del ciclo actual (seleccionando **Multiple Cycles**).

Mientras se está realizando una simulación ciclo a ciclo (o de varios ciclos) se puede visualizar la evolución de las instrucciones en el cauce abriendo la ventana **Clock Cycle Diagram**, que permite visualizar un diagrama espacio-temporal en el que se aparece la situación de las instrucciones en las distintas etapas del cauce en varios ciclos anteriores al presente. En esta ventana también aparecen flechas indicando las dependencias que existen entre las instrucciones. Las flechas verdes corresponden a dependencias que no dan lugar a ciclos perdidos en el cauce (*stall*) debido al uso de caminos de bypass o la interposición de instrucciones entre las instrucciones dependientes, etc. Si se visualiza esta ventana después de realizar la simulación completa del programa se visualizan sólo la información del cauce correspondiente a los últimos 40 ciclos del programa. En la Figura 1.5 se muestra el aspecto de la ventana de diagrama de ciclos de reloj en un momento de la simulación del programa, indicándose las dependencias entre las instrucciones.

Mediante la ventana Pipeline se puede ver la ocupación de las distintas etapas del cauce en el ciclo que se esté simulando (siempre que la ventana se abra con un tamaño suficientemente grande.

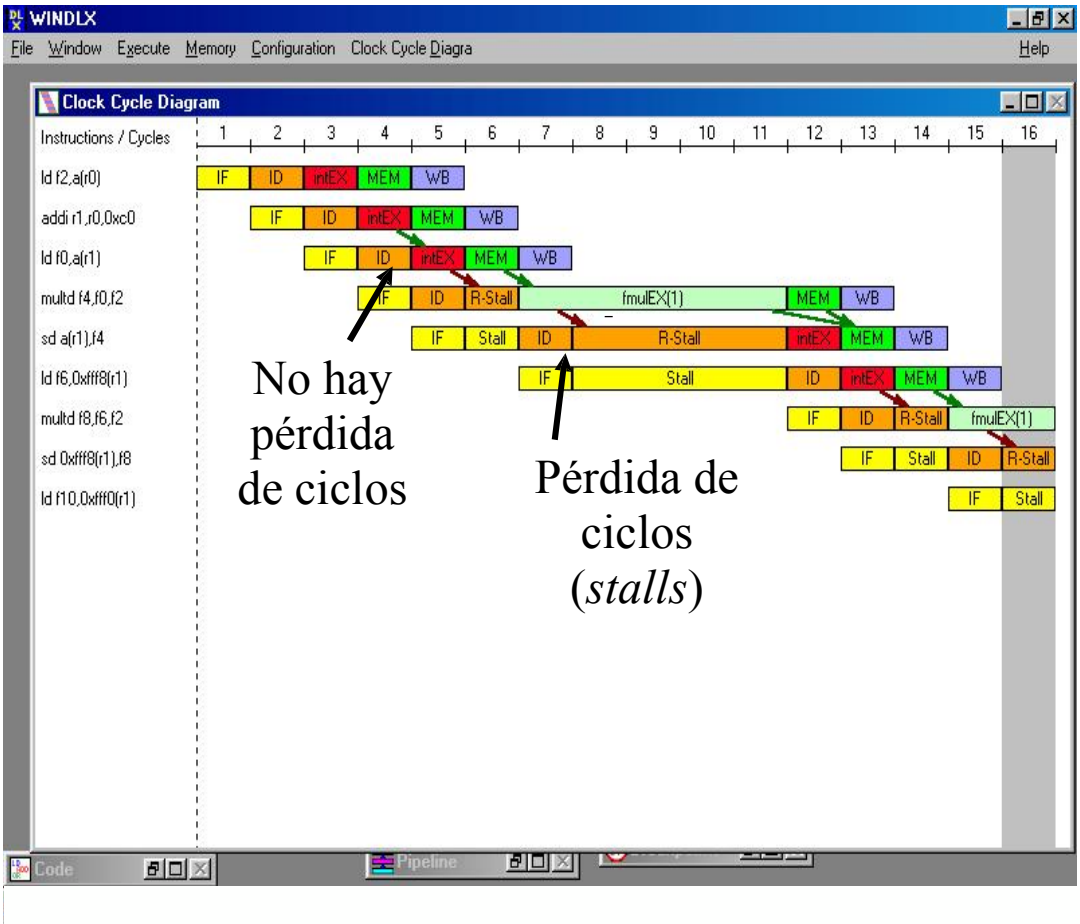


Figura 1.5 Diagrama de ciclos después de 16 ciclos simulados

(5) *Analizar los resultados y actuar en consecuencia (cambiar el programa y/o la configuración con la que se ha realizado la simulación y volver a simular, etc.)*

Cuando se termina la simulación se pueden analizar las prestaciones y la forma en que el programa ha utilizado los recursos del procesador a partir de la ventana **Statistics**. En esa ventana, la primera información que aparece corresponde al número de ciclos que ha tardado en ejecutarse el programa en el procesador, el número de instrucciones que se han decodificado (han pasado por la etapa ID), y el número de instrucciones que quedan en el cauce en el momento que se produce el *trap* hace que se ceda el control al sistema y termina la simulación. A continuación se da información de la configuración del procesador con la que se ha hecho la simulación.

Después se informa de las detenciones (*stalls*) que se han producido en el cauce durante la simulación. Se indican cuantos ciclos se han perdido y los tipos de dependencias que han dado lugar a esas detenciones. De esta manera se pueden analizar las características de paralelismo a nivel de instrucciones que presenta el código y se pueden extraer conclusiones respecto a posibles mejoras. También se informa del número de instrucciones de salto condicional que se han ejecutado indicando cuantas veces se ha producido salto.

Además, se indican el número de instrucciones de acceso a memoria, y de coma flotante. Por último se indica el número de traps que se han producido.

Por otra parte, los valores de las variables en memoria y los registros también se pueden visualizar para comprobar que el programa se ha ejecutado (o se está ejecutando) correctamente. Los valores de los registros se pueden ver abriendo la ventana **Register**, y los contenidos de las variables en memoria seleccionando la opción **Memory**.

La opción **Memory** permite, a su vez: (1) visualizar zonas de memoria (mediante **Display**) indicando si se quiere que se visualicen bytes, palabras, etc. y si se quiere que se muestren en hexadecimal, decimal,...; (2) cambiar los valores de una posición de memoria (mediante **Change**) indicando la dirección correspondiente; y (3) consultar la lista de etiquetas del programa y los valores que tienen asociados, cambiarlas, etc. (mediante **Symbols**).

Una vez terminada una simulación se puede iniciar otra con el mismo fichero si se selecciona la opción **Reset DLX** en **File**, o bien realizar otra con otro programa nuevo si se selecciona **Reset All**. Para salir del simulador se puede seleccionar **Quit WINDLX**, tambien en **File**.

Evaluación de Prestaciones

Tras la simulación del programa *inic1* se puede conocer el número de ciclos que tardaría en ejecutarse el programa en el procesador segmentado. En este caso se han consumido **223 ciclos** de reloj.

Se puede determinar la ganancia de velocidad que se consigue con respecto al procesador sin segmentar **ssrDLX** que hemos definido al final de la Sección 1.1. Para realizar ese cálculo hay que estimar el tiempo de ejecución del programa inic1 en **ssrDLX** teniendo en cuenta el tiempo que tardaría cada una de sus instrucciones.

El tiempo de una instrucción en el procesador sin segmentar se puede obtener partiendo de las etapas por las que pasa esa instrucción. Al no tratarse de un procesador segmentado, cada instrucción tardaría únicamente el tiempo estricto asociado al procesamiento de cada fase, y únicamente se tienen en cuenta las fases que necesita esa instrucción.

Por ejemplo, en *inic1* la primera instrucción, **ld f2,a**, pasaría por todas las etapas ya que al ser una instrucción de acceso a memoria en la que el dato que se lee de memoria (etapa MEM) debe escribirse después en el registro (f2,f3) en la etapa WB. Por supuesto, esta instrucción ha tenido que captarse (IF), decodificarse (ID), y se ha tenido que determinar la dirección de acceso a memoria (intEX). Tal y como se indicó al final de la sección 1.1, cada una de las etapas ID y WB tardan un 80% del tiempo de la etapa IF, o ID, o intEX, que consumen un tiempo igual a un ciclo de reloj cada una. Por lo tanto, el tiempo de la instrucción lf f2,a en el procesador ssrDLX es **1+0.8+1+1+0.8=4.6 ciclos**.

Así habría que hacer con cada una de las instrucciones del programa. Se pueden conocer las etapas por las que pasaría cada instrucción seleccionando dos veces la correspondiente instrucción en la ventana **Clock Cycle Diagram**. En ese caso, aparece información de las transferencias entre registros que se producen al ejecutar la instrucción en cada una de las etapas y se pueden ver las etapas en las que el cauce no hace nada al procesar la instrucción correspondiente.

No obstante, en general se puede anticipar de forma razonada qué etapas utilizará cada instrucción. Así, las instrucciones de almacenamiento en memoria (por ejemplo las instrucciones **sd** de *inic1*) pasarían por las etapas IF, ID, intEX, y MEM, pero no por WB, ya que no tendrían que almacenar nada en los registros de DLX. Eso significa que tardwarían en ejecutarse **1+0.8+1+1=3.8 ciclos**. Las instrucciones de lectura de memoria (como las instrucciones **ld** de inic) ya hemos visto que necesitan **4.6 ciclos**.

En cuanto a las instrucciones que realizan una operación aritmética o lógica, al tratarse de una arquitectura con modelo de operación registro-registro, no necesitarían ejecutar las operaciones correspondientes a la etapa MEM. Por lo tanto realizarían las operaciones correspondientes a las etapas IF, ID, EX, WB. Hay que tener en cuenta que la duración de la etapa EX dependería del tipo de instrucción: si es una instrucción que realiza una operación lógica o aritmética con enteros (excepto la multiplicación y la división) utilizaría intEX y necesitaría sólo un ciclo; si es una suma en coma flotante utilizaría faddEX y tardaría el tiempo asignado en la configuración del procesador a esa unidad (en esta simulación 2 ciclos); si es una multiplicación tardaría el tiempo que consume fmulEX (en esta simulación 5 ciclos); y si es una división tardaría el tiempo que consume fdivEX (en esta simulación 19 ciclos). En el programa *inic1* hay dos tipos de instrucciones aritméticas. Por un lado están las multiplicaciones en doble precisión (instrucciones del tipo **multd fd,fa,fb**) que tardarán **1+0.8+5+0.8=7.6 ciclos** en el procesador **ssrDLX**. Por otro lado están dos instrucciones de aritmética entera, que corresponden a la inicialización de r1 a 0 (**add r1,r0,xtop**) y a la actualización de r1 para determinar si se debe realizar una iteración más (**sub r1,r1,#32**). Cada una de estas instrucciones tarda **1+0.8+1+0.8=3.6 ciclos**.

Finalmente, quedan las instrucciones **bnez r1,loop** que controla el final del bucle, y la instrucción **trap #0** que cede el control al sistema al final del programa. A partir de la ventana Clock Cycle Diagram se puede ver que ambas instrucciones sólo realizan operaciones correspondientes a las etapa IF e ID, y que por lo tanto tardarían **1+0.8=1.8 ciclos** en ssrDLX.

Hay que tener en cuenta que la instrucción **nop** no se utilizaría en una programa para un procesador no segmentado ya que aparece en *inic1* para retrasar la captación de la instrucción que tiene que ejecutarse después de la instrucción de salto (**bnez r1,loop**). Esto debe hacerse así debido a que, tal y como están implementadas las instrucciones de salto condicional en DLX, el procesador siempre aborta la ejecución de cualquier

instrucción que se capte tras una instrucción de este tipo para evitar que se introduzcan en el cauce instrucciones que no deben ejecutarse si se produce el salto.

Así pues, una vez conocidos los tiempos de ejecución en ssrDLX de cada una de las instrucciones de *inic1*, sólo falta sumar los tiempos de todas las instrucciones del programa:

4.6 + 3.6 + 6*[(4.6 + 7.6 + 3.8) * 4 + 3.6 +1.8] + 1.8 = 426.4

donde **[(4.6 + 7.6 + 3.8) * 4 + 3.6 +1.8]** es el tiempo de una iteración del bucle incluyendo los cuatro grupos de instrucciones **ld – multd – sd**, la instrucción de actualización de r1, y la de salto condicional.

De esta forma, la ganancia de velocidad que se consigue al ejecutar el programa inic1 en el cauce es de

Ganancia(inic1) = 426.4 / 223 = 1.9

Se trata de un valor relativamente pequeño ya que, si se considera que se ha utilizado un cauce con 5 etapas, la eficiencia sería sólo de **E(inic1) = 1.9 / 5 = 0.38**. Además, esta simulación se ha realizado con la opción de **Enable Forwarding** activa, con lo que se supone que hay caminos de bypass que evitan los retardos que podrían ocasionar ciertas dependencias. Si se hace la simulación desactivando esta opción se obtendría un tiempo de **303 ciclos** en el procesador segmentado, con lo que la ganancia sería **Ganancia(inic1_nf)= 426.4 / 303 = 1.41** y la eficiencia **E(inic1_nf)= / 5 = 0.28**.

Si se analizan los resultados de la ejecución de inic1 (con la opción **Enable Forwarding** activa) que se muestran en la ventana **Statistics**, se puede ver que de los **223** ciclos que tarda el programa en ejecutarse, se han producido detenciones del cauce en un total de **134 ciclos**: más de la mitad del tiempo de ejecución. La cuestión que se plantea ahora es la de ver si es posible realizar transformaciones en el código de forma que se reduzca este número de ciclos desperdiciados. Esta es una de las funciones del compilador que, en su fase de optimización debe ser capaz de disponer las instrucciones de programa de forma que se saque el mayor partido posible de la arquitectura del procesador.

Mejora de Prestaciones del programa inic1

En las Figuras 1.6, 1.7, 1.8, y 1.9 se muestran los fragmentos de código correspondientes a cuatro posibles modificaciones del programa *inic1*, denominados respectivamente *inic2*, *inic3*, *inic4*, e *inic5*. Estos programas tienen un tiempo de ejecución menor que *inic1*. Concretamente inic2 se ejecuta en 211 ciclos, inic3 en 187 ciclos, inic4 en 181 ciclos, e inic5 en 183 ciclos.

El código *inic2* se ha transformado aplicando una técnica que recibe el nombre de *desenrollado de bucle* que consiste en realizar los cálculos de varias iteraciones en una sóla para reducir así el número de instrucciones de salto que se tienen que ejecutar, con lo que además se evitan riesgos de control (en este caso se evitan los ciclos perdidos por las instrucciones **nop** que se ponen detrás de la instrucción de salto condicional).

Los códigos *inic3* e *inic4* se obtienen a partir de distintas reorganizaciones de las instrucciones para reducir el efecto de las dependencias entre ellas. Por ejemplo, en *inic3* se ha situado la instrucción **ld f6,-8(r1)** entre las instrucciones **ld f0,0(r1)** y **multd f4,f0,f2** que en inic1 estaban una a continuación de otra. De esta forma, al interponer la instrucción de carga **ld f6,-8(r1)** con la que no tienen dependencias las otras, se retrasa el instante en que la multiplicación necesita el contenido de **(f0,f1)** que carga **ld f0,0(r1)**.

La transformación que se ha aplicado para obtener *inic5* recibe el nombre de *segmentación software (sofware pipelining)*, que consiste en organizar el bucle de forma que en cada iteración se carguen los datos que se procesarán en la siguiente iteración, se procesan los que se cargaron en la anterior, y se almacenan los que se calcularon en la iteración anterior. De esta forma se consigue reducir las dependencias entre las instrucciones que se ejecutan en cada iteración del bucle.

***Cuestión*:** ¿Cuáles son las ganancias que se obtienen en inic2 inic3, inic4, e inic5 con la segmentación con respecto al cauce sin segmentar de ssrDLX? ¿Cuál de los cuatro códigos es el mejor?

```
      .text   256

start:ld      f2,a                (1)
      add     r1,r0,xtop          (2)
loop:ld       f0,0(r1)            (3)
      multd   f4,f0,f2            (4)
      sd      0(r1),f4            (5)
      ld      f6,-8(r1)           (6)
      multd   f8,f6,f2            (7)
      sd      -8(r1),f8           (8)
      ld      f10,-16(r1)         (9)
      multd   f12,f10,f2          (10)
      sd      -16(r1),f12         (11)
      ld      f14,-24(r1)         (12)
      multd   f16,f14,f2          (13)
      sd      -24(r1),f16         (14)
      ld      f18,-32(r1)         (15)
      multd   f20,f18,f2          (16)
      sd      -32(r1),f20         (17)
      ld      f22,-40(r1)         (18)
      multd   f24,f22,f2          (19)
      sd      -40(r1),f24         (20)
      ld      f26,-48(r1)         (21)
      multd   f28,f26,f2          (22)
      sd      -48(r1),f28         (23)
      ld      f30,-56(r1)         (24)
      multd   f16,f30,f2          (25)
      sd      -56(r1),f16         (26)
      sub     r1,r1,#64           (27)
      bnez    r1,loop             (28)
      nop                         (29)
      trap    #0                  (30)
```

Figura 1.6 Código del programa *inic2*

```
      .text   256

start:ld      f2,a            (1)
      add     r1,r0,xtop      (2)
loop:ld       f0,0(r1)        (3)
      ld      f6,-8(r1)       (4)
      multd   f4,f0,f2        (5)
      multd   f8,f6,f2        (6)
      sd      0(r1),f4        (7)
      sd      -8(r1),f8       (8)
      ld      f10,-16(r1)     (9)
      ld      f14,-24(r1)     (10)
      multd   f12,f10,f2      (11)
      multd   f16,f14,f2      (12)
      sd      -16(r1),f12     (13)
      sd      -24(r1),f16     (14)
      sub     r1,r1,#32       (15)
      bnez    r1,loop         (16)
      nop                     (17)
      trap    #0              (18)
```

Figura 1.7 Código del programa *inic3*

```
      .text   256

start:ld      f2,a              (1)
      add     r1,r0,xtop        (2)
loop:ld       f0,0(r1)          (3)
      ld      f6,-8(r1)         (4)
      ld      f10,-16(r1)       (5)
      ld      f14,-24(r1)       (6)
      multd   f4,f0,f2          (7)
      multd   f8,f6,f2          (8)
      multd   f12,f10,f2        (9)
      multd   f16,f14,f2        (10)
      sd      0(r1),f4          (11)
      sd      -8(r1),f8         (12)
      sd      -16(r1),f12       (13)
      sd      -24(r1),f16       (14)
      sub     r1,r1,#32         (15)
      bnez    r1,loop           (16)
      nop                       (17)
      trap    #0                (18)
```

Figura 1.8 Código del programa *inic4*

```
      .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 1.9 Código del programa *inic5*

En el Apéndice 1.3 se describen, además de las técnicas utilizadas en los programas de ejemplo *inic2* a *inic5*, algunas de las técnicas que pueden utilizarse para mejorar el rendimiento de los programas en un procesador segmentado.

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.

Ejercicios Prácticos con WinDLX

El diagrama de la Figura 1.1 muestra una serie de cálculos a realizar con cuatro variables reales A,B,C,D.

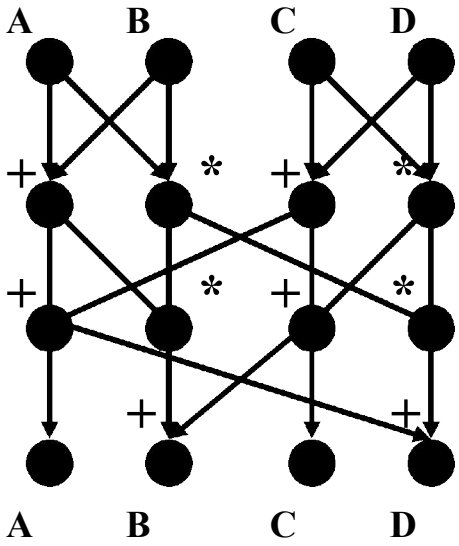


Figura 1.1 Diagrama de cálculo

El programa en ensamblador contenido en el fichero ***pr1_1.s*** (cuyo contenido se muestra en el Apéndice 1.2) ha sido generado con el compilador **`dlxcc`** a partir de un programa en C que implementaba el cálculo de la Figura 1.1.

1. Ejecute el programa incluido en ***pr1_1.s*** con el simulador WinDLX y determine el tiempo de procesamiento, el número de ciclos en los que el cauce ha estado detenido (*stall*) y las causas de las detenciones. Determine el número de instrucciones por ciclo que ejecuta el procesador y calcule la ganancia de velocidad con respecto al procesador sin segmentar de referencia ssrDLX. (Nota: el número de ciclos en las unidades funcionales ha de ser el mismo que en el procesador ssrDLX).

2. Teniendo en cuenta las causas de las detenciones en la ejecución del programa incluido en ***pr1_1.s***, y las posibles ineficiencias en el uso de los recursos del procesador, optimice las prestaciones obtenidas: (a) modificando manualmente ***pr1_1.s*** mediante transformaciones del tipo de movimiento de instrucciones, renombrado de registros, etc.; y (b) realizando un programa en ensamblador que implemente directamente el cálculo descrito en la Figura 1.1.

4. Utilizando la mejor y la peor versión del programa del fichero ***pr1_1.s*** determine la influencia que tiene en las prestaciones el cambio en los ciclos del multiplicador y del sumador del cauce. Para ello, aumente 10 veces en paralelo la latencia de estas dos unidades, partiendo de los valores por defecto (2,5) hasta (20,50).

5. Suponga que la secuencia de operaciones de la Figura 1.1 se encuentra dentro de un bucle que repite un número de veces N. Evalúe las prestaciones que se obtienen en función de N para el código del programa ***pr1_1.s***. Para ello, varíe el valor de N desde 1 hasta 100 de 10 en 10. Optimice el programa obtenido (a) mediante técnicas como desenrollado de bucles, salto retardado, reordenación de instrucciones, etc.; y (b) realizando directamente el programa en ensamblador que realice el mismo cálculo.

Documentación a presentar:

1. Programas en C y en DLX realizados para cada uno de los apartados.
2. Resultados experimentales obtenidos en las ejecuciones efectuadas en cada apartado.
3. Comentarios relativos a las modificaciones realizadas en los códigos, justificación de los resultados obtenidos, con las correspondientes tablas y gráficas de apoyo.

Apéndice 1.1. Arquitectura del procesador DLX

Repertorio de Instrucciones. El repertorio de instrucciones DLX corresponde a un modelo de ejecución registro-registro en el que las operaciones se ejecutan sobre datos en registros y almacenan los resultados en registros. El acceso a memoria se produce sólo a través de instrucciones de carga (load) y almacenamiento (store). A continuación se proporciona el repertorio de instrucciones con sus correspondientes significados.

Instrucciones de Transferencia de Datos

LB Rd,Adr	Cargar un byte (con extensión para signo) en Rd
LBU Rd,Adr	Cargar un byte (sin signo)
LH Rd,Adr	Cargar media palabra (con extensión para signo)
LHU Rd,Adr	Cargar media palabra (sin signo)
LW Rd,Adr	Cargar una palabra
LF Fd,Adr	Cargar dato en coma flotante y simple precisión
LD Dd,Adr	Cargar dato en coma flotante y doble precisión
SB Adr,Rs	Almacenar un byte
SH Adr,Rs	Almacenar media palabra
SW Adr,Rs	Almacenar una palabra
SF Adr,Fs	Almacenar dato en coma flotante y simple precisión
SD Adr,Fs	Almacenar dato en coma flotante y doble precisión
MOVI2FP Fd,Rs	Mover 32 bits desde uno de los registros enteros (Rs) a uno (Fd) de los los registros en coma flotante (FP)
MOVI2FP Rd,Fs	Mover 32 bits desde los registros FP a los registros enteros
MOVF Fd,Fs	Mover desde un registro FP a otro registro FP
MOVD Dd,Ds	Mover desde un dato de doble precisión desde un par de registros a otro par
MOVI2S SR,Rs	Mover desde un registro a un registro especial (no está implementada en el simulador)
MOVS2I Rs,SR	Mover desde un registro especial a otro registro GPR (no está implementada en el simulador)

Instrucciones Aritméticas y Lógicas

ADD Rd,Ra,Rb	Suma con signo de Ra y Rb con el resultado en Rd
ADDI Rd,Ra,Imm	Suma de Ra con un dato inmediato (todos los datos inmediatos son de 16 bits).
ADDU Rd,Ra,Rb	Suma sin signo
ADDUI Rd,Ra,Imm	Suma sin signo de un registro y un dato inmediato.
SUB Rd,Ra,Rb	Resta con signo (Ra menos Rb y el resultado se guarda en Rd)
SUBI Rd,Ra,Imm	Resta con signo de un dato inmediato
SUBU Rd,Ra,Rb	Resta sin signo de registros
SUBUI Rd,Ra,Imm	Resta sin signo de un dato inmediato
MULT Rd,Ra,Rb	Multiplicación con signo de Ra y Rb con resultado en Rd
MULTU Rd,Ra,Rb	Multiplicación sin signo de registros
DIV Rd,Ra,Rb	División con signo de Ra entre Rb con el resultado en Rd
DIVU Rd,Ra,Rb	División sin signo
AND Rd,Ra,Rb	And bit a bit de Ra y Rb con el resultado en Rd
ANDI Rd,Ra,Imm	And bit a bit con un dato inmediato
OR Rd,Ra,Rb	Or bit a bit de Ra con Rb y el resultado en Rd
ORI Rd,Ra,Imm	Or bit a bit con un dato inmediato

XOR Rd,Ra,Rb	Xor bit a bit de Ra y Rb con el resultado en Rd
XORI Rd,Ra,Imm	Xor bit a bit con un dato inmediato
LHI Rd,Imm	Carga de un dato inmediato (16 bits) en la mitad superior (más significativa) de un registro (32 bits)
SLL Rd,Rs,Rc	Desplazamiento lógico a la izquierda de un dato en Rs, el número de bits indicado en Rc, y con el resultado en Rd
SRL Rd,Rs,Rc	Desplazamiento lógico a la derecha
SRA Rd,Rs,Rc	Desplazamiento aritmético a la derecha
SLLI Rd,Rs,Imm	Desplazamiento lógico a la izquierda con el número de bits indicado en un dato inmediato
SRLI Rd,Rs,Imm	Desplazamiento lógico a la derecha con el número de bits indicado en un dato inmediato
SRAI Rd,Rs,Imm	Desplazamiento aritmético a la derecha con el número de bits indicado en un dato inmediato
S__ Rd,Ra,Rb	Pone Rd a 1 si entre Ra y Rb se cumple la condición indicada en "__", que puede ser: EQ, NE, LT, GT, LE o GE
S__I Rd,Ra,Imm	Pone a Rd a 1 si entre Ra y un dato inmediato se cumple la condición indicada en "__", que puede ser: EQ, NE, LT, GT, LE o GE
S__U Rd,Ra,Rb	Pone Rd a 1 si entre Ra y Rb (sin signo) se cumple la condición indicada en "__", que puede ser: EQ, NE, LT, GT, LE o GE
S__UI Rd,Ra,Imm	Pone a Rd a 1 si entre Ra y un dato inmediato (sin signo) se cumple la condición indicada en "__", que puede ser: EQ, NE, LT, GT, LE o GE
NOP	No operar

Instrucciones de Control

BEQZ Rt,Dest	Salta si el registro Rt (del conjunto GPR) es igual a cero. El desplazamiento del salto (desde PC) se indica en Dest como un dato de 16 bit.
BNEZ Rt,Dest	Salta si el registro Rt (del conjunto GPR) no es igual a cero. El desplazamiento del salto (desde PC) se indica en Dest como un dato de 16 bit.
BFPT Dest	Comprueba si el bit de comparación en el registro de estado para coma flotante (FPSR) está a 1 y salta en ese caso. El desplazamiento del salto (desde PC) se indica en Dest como un dato de 16 bit.
BFPF Dest	Comprueba si el bit de comparación en el registro de estado para coma flotante (FPSR) está a 0 y salta en ese caso. El desplazamiento del salto (desde PC) se indica en Dest como un dato de 16 bit.
J Dest	Salta a una dirección obtenida sumando a PC un desplazamiento de 26 bits dado en Dest.
JR Rx	Salta a una dirección de destino dada en Rx
JAL Dest	Salta a una dirección obtenida sumando a PC el desplazamiento dado en Dest y almacena en R31 el contenido de PC+4.
JALR Rx	Salta a una dirección contenida en Rx y almacena en R31 el contenido de PC+4.

TRAP Imm	Transfiere el control al sistema operativo, a una dirección vectorizada proporcionada como dato inmediato (ver la sección correspondiente a Excepciones - TRAPs -).
RFE Dest	Retorno al código de usuario (y al modo de usuario) tras una excepción (no está implementada en el simulador).

Instrucciones de Coma Flotante

ADDD Dd,Da,Db	Suma de números en doble precisión almacenados en los pares de registros Da y Db y alamcena el resultado en el par de registros Dd
ADDF Fd,Fa,Fb	Suma números en simple precisión almacenados en Fa y Fb y almacena el resultado en Fd
SUBD Dd,Da,Db	Resta números en doble precisión (Da menos Db con el resultado en Dd)
SUBF Fd,Fa,Fb	Resta números en simple precisión
MULTD Dd,Da,Db	Multiplica números en doble precisión contenidos en los pares de registros Da y Db y pone el resultado en el par de registros Dd
MULTF Fd,Fa,Fb	Multiplica números en simple precisión.
DIVD Dd,Da,Db	Divide números en doble precisión (el par de registros Da entre el par de registros Db y pone el resultado en Db).
DIVF Fd,Fa,Fb	Divide números en simple precisión
CVTF2D Dd,Fs	Convierte un número de simple precisión (contenido en Fs) a doble precisión (en el par de registros Dd)
CVTD2F Fd,Ds	Convierte un número en doble precisión a simple precisión
CVTF2I Fd,Fs	Convierte un número de simple precisión a entero
CVTI2F Fd,Fs	Convierte un número entero a número en simple precisión
CVTD2I Fd,Ds	Convierte un número en doble precisión a número entero
CVTI2D Dd,Fs	Convierte un número entero a número en doble precisión.
__D Da,Db	Comparación de doble precisión, que pone a 1 el bit de comparación del registro de estado de coma flotante (FPSR) si se verifica la condición indicada en "__": EQ, NE, LT, GT, LE o GE
__F Fa,Fb	Comparación de doble precisión, que pone a 1 el bit de comparación del registro de estado de coma flotante (FPSR) si no se verifica la condición indicada en "__": EQ, NE, LT, GT, LE o GE

Tipos de Instrucciones. Todas las instrucciones del repertorio DLX tienen 32 bits en los que 6 bits corresponden al código de operación (primario). Atendiendo a las distintas formas de codificación, las instrucciones se clasifican en tres tipos cuyas estructuras se muestran en la Figura A1.1.1.

Tipo I
Codifica las siguientes instrucciones:

- Cargas (loads) y Almacenamientos (stores): **Rd <- Mem[Rs+Inmediato]**
- Operaciones con Datos Inmediatos: **Rd <- Rs op Immediato**
- Salto condicional: **Rs es el registro usado en la instrucción, Rd no se usa**
- JR (salto con registro) y JALR (salto y enlace con registro): **Rd=0, Rs=Rx, Inmediato=0**

Tipo R
Codifica las siguientes instrucciones:

- Operaciones con la ALU registro-registro: **Rd <- Ra op Rb**
- Instrucciones MOVxxx: **Rd <- Ra**

Tipo J
Codifica las instrucciones:

- J (Salto, *Jump*) y JAL (Salto y Enlace)
- TRAP
- RFE (Retorno de una excepción)

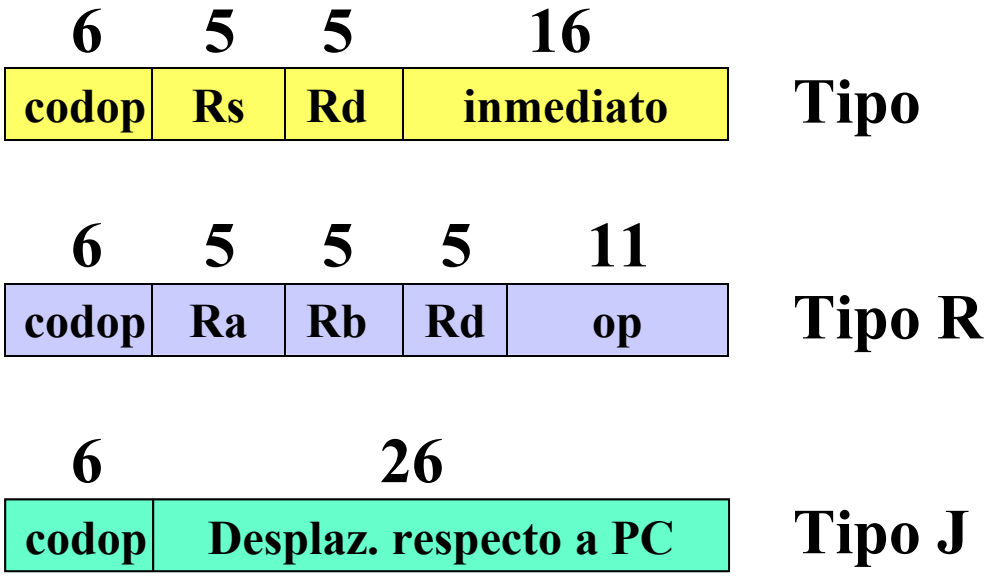


Figura A1.1.1. Formatos de los distintos tipos de instrucciones

Conjunto de Registros. DLX tiene una serie de registros que son visibles para el programador y otros registros internos que no lo son. A continuación se enumeran todos los registros y se explica su uso.

Registros visibles a los programas:

- **GPRs (R0..R31):** Registros (32) de 32 bits de propósito general. R0=0 siempre.
- **FPRs (F0..F31):** Registros de 32 bits para datos en coma flotante de simple precisión. Se pueden utilizar por parejas (par-impar) para almacenar datos de coma flotante en doble precisión: D0 – D30 (16 registros de 64 bits) con D0 = (F1-F0) ....
- **FPSR:** Registro de estado para las operaciones en coma flotante. Se utiliza en las comparaciones y en las excepciones de coma flotante. Todos los movimientos a y desde el registro FPSR se realizan a través de los registros GPRs.
- **PC:** Contador de Programa que contiene la dirección de la siguiente instrucción a ser captada (se incremente de 4 en 4 ya que tiene 32 bits = 4 bytes).

Registros internos:

- **IMAR:** Registro inicializado con el contenido del contador de programa de la instrucción en la etapa IF.
- **IR:** Registro en la etapa IF que se carga con la dirección de la siguiente instrucción a captar.
- **A, B:** Registros que se cargan en la etapa ID con los contenidos de los registros que se operan. Estos registros constituyen las entradas a la ALU. También existen dos registros AHI y BHI que contienen los 32 bits de la parte superior de los datos en coma flotante de doble precisión.
- **BTA:** En este registro se escribe la dirección de salto en las instrucciones branch y jump. Esta dirección se calcula en la etapa ID, y si se produce el salto, el contenido de BTA se carga en PC.
- **ALU:** Registro de 32 bits donde se cargan los resultados de la ALU. Existe un registro ALUHI que contiene los 32 bits más altos de un datos de doble precisión.
- **DMAR:** Registro donde se carga la dirección de memoria para un acceso a la misma para una carga o almacenamiento en la etapa MEM.
- **SDR:** Registro que contiene el dato que se va a escribir en memoria en un almacenamiento (*store*). Existe un registro SDRHI para los 32 bits superiores de los datos de doble precisión.
- **LDR:** Registro donde se almacena el dato leido de memoria en una carga (*load*). Existe un registro LDRHI para los 32 bits superiores de los datos de doble precisión.

Ensamblador para DLX. A continuación se describe brevemente la sintaxis de las expresiones en ensamblador para DLX, y se enumeran las directivas junto con su significado.

La sintaxis de las expresiones es similar a la de C, debiendo estar el valor de una expresión definido en el rango apropiado para el tipo de expresión:

Bytes:	-128 .. +255 (8 Bit)
Halfwords:	-32768 .. +65535 (16-Bit)
Words:	-2147483648 ..+2147483647 o inferior a 0xffffffff (32-Bit)

El ensamblador acepta números en notación decimal, hexadecimal (los dos primeros caracteres del número son **0x**), u octal (el primer carácter del número es **0**).

Respecto a las expresiones de una dirección, la forma más sencilla de expresar una dirección es utilizar un número (que se interpreta como una dirección). No obstante, se pueden utilizar símbolos para indicar direcciones (estos símbolos deben estar definidos en el fichero ensamblador que se carga. Los operadores *. /, +, -, <<, >>, &, |, y ^ tienen el mismo significado que en C y las reglas de agrupación y paréntesis son también las mismas que en C. Las cadenas de caracteres deben indicarse entre comillas (“ “).

El programa, inicialmente, se carga en el segmento *text*. Por defecto, el código se carga en la posición $CODE (inicialmente es 0x100) y los datos en la posición $DATA (inicialmente en 0x1000).

Cuando el programa ensamblador está procesando un fichero los datos y las instrucciones que va ensamblando se sitúan en memoria utilizando un puntero a texto (código) o a datos. El puntero que se utiliza no depende del tipo de información (instrucciones o datos) sino en si la directiva más reciente ha sido **.data** o **.text**. A continuación se describe el conjunto de directivas disponible.

Directivas del Ensamblador

.align *n*	Hace que el siguiente dato o la siguiente instrucción se cargue en la siguiente dirección con los n bits menos significativos iguales a 0 (por ejemplo **.align** *2* indica la dirección de palabra siguiente – 32 bits = 4 bytes y **. align** *2* apunta a una dirección X...XXXX00 -).
.ascii *“string1”,”..”*	Almacena en memoria las cadenas indicadas en la línea como una lista de caracteres. Las cadenas no terminan con un byte igual a 0
.asciiz *“string1”,”..”*	Igual que **.ascii** excepto que cada cadena termina con un byte igual a 0
.byte *byte1,byte2,..*	Almacena los bytes indicados en la directiva secuencialmente en memoria.
.data *[dirección]*	Hace que el código o los datos que la siguen se almacenen en el área de datos: si se indica una dirección, la carga empieza por esa dirección, y si no se indica dirección se

	utiliza el último valor del puntero de datos. Si se estuvieran leyendo a partir del puntero de texto, hay que almacenar esa dirección para poder continuar desde ahí en otro momento (mediante una directiva **.text**)
.double *número1,..*	Almacena los números indicados en la directiva secuencialmente en memoria como números en coma flotante de doble precisión.
.global *etiqueta*	Hace que la etiqueta esté disponible para ser utilizada por las instrucciones correspondientes al código cargado después de esta directiva.
.space *tamaño*	Mueve hacia direcciones superiores el puntero actual utilizado para almacenar (datos o código). El número de bytes que se dejan vacíos por esta directiva es igual a *tamaño*.
.text *[dirección]*	Hace que las siguientes instrucciones o datos se almacenen en el área de texto (código): si se indica una dirección la carga se produce a partir de ella, y si no se indica se utiliza el último valor del puntero de texto. Si se estuviera leyendo utilizando el puntero de datos habría que almacenar esta dirección para poder continuar desde ahí después (mediante una directiva ***.data***)
.word *palabra1, palabra2,..*	Almacena las palabras indicadas en la línea de la directiva secuencialmente en memoria

Excepciones (TRAPs). La interfaz entre los programas DLX y el sistema de entradas/salidas se realiza a través del uso de excepciones (TRAPs). En WinDLX hay cinco TRAPs definidos

- **Trap #1:** Abrir fichero
- **Trap #2:** Cerrar fichero
- **Trap #3:** Leer un bloque del fichero
- **Trap #4:** Escribir un bloque en el fichero
- **Trap #5:** Salida formateada a través dela salida estándard.

El valor 0 para un TRAP (TRAP #0) no está permitido, y se utiliza para terminar un programa. En la propia ayuda de WinDLX se pueden consultar más detalles acerca del uso de los TRAPs.

Apéndice 1.2 Contenido del Fichero pr1_1.s

```
.data
.global _a
        .align 4
_a:
        .float 0.000000000000
.global _b
        .align 4
_b:
        .float 1.000000000000
.global _c
        .align 4
_c:
        .float 1.000000000000
.global _d
        .align 4
_d:
        .float -1.000000000000
.global _n
_n:     .space 8
.global _m
_m:     .space 8
.global _l
_l:     .space 8
.global _k
_k:     .space 8
.global _j
_j:     .space 8
.global _i
_i:     .space 8
.global _h
_h:     .space 8
.global _g
_g:     .space 8
.global _f
_f:     .space 8
.global _e
_e:     .space 8

        .align 4
        .text
.global _main

_main:
        lhi r3,(_e>>16)&0xffff
        addui r3,r3,(_e&0xffff)
        lhi r4,(_a>>16)&0xffff
        addui r4,r4,(_a&0xffff)
        lhi r5,(_b>>16)&0xffff
        addui r5,r5,(_b&0xffff)
        lf f4,0(r4)
        lf f5,0(r5)
        addf f4,f4,f5
        sf 0(r3),f4

        lhi r3,(_f>>16)&0xffff
        addui r3,r3,(_f&0xffff)
        lhi r4,(_a>>16)&0xffff
        addui r4,r4,(_a&0xffff)
        lhi r5,(_b>>16)&0xffff
        addui r5,r5,(_b&0xffff)
        lf f4,0(r4)
        lf f5,0(r5)
        multf f4,f4,f5
        sf 0(r3),f4

        lhi r3,(_g>>16)&0xffff
        addui r3,r3,(_g&0xffff)
        lhi r4,(_c>>16)&0xffff
        addui r4,r4,(_c&0xffff)
        lhi r5,(_d>>16)&0xffff
        addui r5,r5,(_d&0xffff)
        lf f4,0(r4)
```

```
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4

lhi r3,(_h>>16)&0xffff
addui r3,r3,(_h&0xffff)
lhi r4,(_c>>16)&0xffff
addui r4,r4,(_c&0xffff)
lhi r5,(_d>>16)&0xffff
addui r5,r5,(_d&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4

lhi r3,(_i>>16)&0xffff
addui r3,r3,(_i&0xffff)
lhi r4,(_e>>16)&0xffff
addui r4,r4,(_e&0xffff)
lhi r5,(_g>>16)&0xffff
addui r5,r5,(_g&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4

lhi r3,(_j>>16)&0xffff
addui r3,r3,(_j&0xffff)
lhi r4,(_e>>16)&0xffff
addui r4,r4,(_e&0xffff)
lhi r5,(_f>>16)&0xffff
addui r5,r5,(_f&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4

lhi r3,(_k>>16)&0xffff
addui r3,r3,(_k&0xffff)
lhi r4,(_g>>16)&0xffff
addui r4,r4,(_g&0xffff)
lhi r5,(_h>>16)&0xffff
addui r5,r5,(_h&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4

lhi r3,(_l>>16)&0xffff
addui r3,r3,(_l&0xffff)
lhi r4,(_f>>16)&0xffff
addui r4,r4,(_f&0xffff)
lhi r5,(_h>>16)&0xffff
addui r5,r5,(_h&0xffff)
lf f4,0(r4)
lf f5,0(r5)
multf f4,f4,f5
sf 0(r3),f4

lhi r3,(_m>>16)&0xffff
addui r3,r3,(_m&0xffff)
lhi r4,(_j>>16)&0xffff
addui r4,r4,(_j&0xffff)
lhi r5,(_k>>16)&0xffff
addui r5,r5,(_k&0xffff)
lf f4,0(r4)
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4

lhi r3,(_n>>16)&0xffff
addui r3,r3,(_n&0xffff)
lhi r4,(_i>>16)&0xffff
addui r4,r4,(_i&0xffff)
lhi r5,(_l>>16)&0xffff
addui r5,r5,(_l&0xffff)
lf f4,0(r4)
```

```
lf f5,0(r5)
addf f4,f4,f5
sf 0(r3),f4

lhi r3,(_a>>16)&0xffff
addui r3,r3,(_a&0xffff)
lhi r4,(_i>>16)&0xffff
addui r4,r4,(_i&0xffff)
lw r6,0(r4)
sw 0(r3),r6

lhi r3,(_b>>16)&0xffff
addui r3,r3,(_b&0xffff)
lhi r4,(_m>>16)&0xffff
addui r4,r4,(_m&0xffff)
lw r6,0(r4)
sw 0(r3),r6
```