Fundamentos de Arquitectura de Ordenadores Ingeniería Técnica en Informática Sistemas

PRÁCTICA IV
Procesador DLX. Camino de datos segmentado

Objetivos

- Comprender la terminología y los conceptos manejados en el estudio de los computadores segmentados: dependencias de datos, de control, tiempos de riesgos, adelantamiento, saltos retardados, etc.
- Realizar programas en ensamblador del DLX y probar distintas técnicas de planificación estática de instrucciones.
- Comprobar cómo el rendimiento obtenido por una determinada arquitectura segmentada tiene una dependencia importante del código ejecutado, y cómo se puede optimizar el rendimiento de una arquitectura dada modificando adecuadamente el software que la utiliza.

Introducción

Al ejecutar un programa, uno de los objetivos que generalmente perseguimos es que dicho programa se ejecute lo mas rápidamente posible, obteniendo el máximo provecho del procesador, el máximo rendimiento. Para medir este rendimiento, son varios los parámetros que podemos tener en cuenta:

- **NC:** Número de ciclos de reloj de CPU para ejecutar un programa.
- **NI:** Número de instrucciones de las que se compone el programa.
- **TC:** Duración de un ciclo de reloj.
- **Tcpu=**$NC * TC$**:** Tiempo requerido en ejecutar un programa.
- **CPI=**NC/NI**:** Número medio de ciclos por instrucción.

De aquí,

$$Tcpu = CPI * TC * NI$$

Si queremos hacer este tiempo lo más pequeño posible, tendríamos que disminuir alguno de los tres factores (CPI, TC, NI). No obstante, esta tarea no es tan simple, ya que los factores interactúan mutuamente y cuando disminuye alguno de ellos los otros aumentan. El ciclo de

reloj depende de la tecnología usada en el hardware, el CPI de la organización y repertorio de las instrucciones, y el NI de el repertorio de instrucciones y de la tecnología del compilador.

La idea de segmentar la ejecución de las instrucciones pretende acercar el valor de CPI a su valor más pequeño posible (1), valor ideal en el que todas las instrucciones tardan un ciclo de reloj.

En este tema nos centraremos en la simulación del procesador RISC segmentado DLX, procesador que presenta un gran número de similaridades con el procesador MIPS.

El procesador DLX

A continuación resumimos las características más importantes de la arquitectura del procesador DLX:

- **Arquitectura de carga y almacenamiento**: sólo las instrucciones del tipo load/store acceden a memoria (las instrucciones de operación son sólo entre registros).
- **Unidades de operaciones aritméticas separadas**: Una de números enteros y varias de punto flotante (suma, multiplicación y división).
- **Tamaño de palabra de 32 bits** (1 word = 32 bits, 1 halfword = 16 bits).
- **32 registros de propósito general (GPR) de 32 bits para enteros**. Se denotan: R0,R1,R2,...,R31. El registro R0 siempre tiene el valor 0.
- **32 registros de números en punto flotante** de simple precisión (32 bits), a los que se refiere como: F0,F1,...F31. Los registros Fx se pueden agrupan en pares consecutivos para operaciones de punto flotante en doble precisión (64 bits); a estos registros de doble precisión nos referiremos como D0,D1,...D30. (El registro Di está constituído por los registros Fi+1:Fi).
- **Estructura segmentada**: La ejecución de una instrucción se divide en 5 etapas. Dichas etapas (pipeline) son las siguientes:

  1. **Etapa IF**: Búsqueda de instrución.
  2. **Etapa ID**: Decodificación de instrucción.
  3. **Etapa EX**: Ejecución (operación o cálculo de dirección efectiva). Esta etapa puede realizarse en diferentes unidades, según la instrucción sea una operación de enteros (intEX), o flotantes (y estos últimos: suma, producto o división, denominándose faddEX, fmulEX, fdivEX (las unidades fmulEX y fdivEX también realizan la multiplicación y división de enteros).
  4. **Etapa MEM**: Carga o almacenamiento de datos en memoria.

5. **Etapa WB** (write back): Almacenamiento de los resultados de la instrucción en los registros correspondientes. Se realiza en la primera mitad del ciclo de reloj, de forma que una instrucción que simultáneamente esté en la etapa ID puede tomar el valor de los registros en la segunda mitad del ciclo, sin necesidad de cortocircuito (forwarding).

Todas las etapas ocupan un ciclo de reloj, salvo las de ejecución de punto flotante (faddEX, fmulEX, fdivEX) que tardan más de un ciclo en ser realizadas.

- **Conjunto reducido de instrucciones**. Dicho conjunto contiene los siguientes 4 tipos de instrucciones:
  - Transferencia de datos (carga, almacenamiento...).
  - Instrucciones aritméticas y lógicas sobre enteros.
  - Instrucciones de control.
  - Instrucciones de punto flotante.
- **Modos de direccionamiento** soportados:
  - Inmediato. Ejemplo:
    ```
    addi r1,r0,#1 ; r1=r0+1
    ```
  - Directo a memoria. Ejemplo:
    ```
    lb r1,0x10 ; r1=[0x10]
    lb r1,label ; r1=[label]
    ```
    Los números en hexadecimal se expresan como `0xN`.
  - Indirecto: Se expresa como `c(Rx)`, donde `c` es un desplazamiento de 16 bits y `Rx` es un registro que contiene una dirección de memoria. Ejemplo:
    ```
    addi r1,r0,label ; r1=label (dirección de memoria)
    lb r2,8(r1) ; r2=[label+8]
    ```

Para mayor detalle acerca de las instrucciones es conveniente consultar el contenido de la ayuda del programa simulador.

Riesgos en la segmentación

Aunque hemos pensado en la segmentación como un método para mejorar la eficiencia del procesador, la aparición de dependencias entre las instrucciones limita dicha mejora. La dependencia entre instrucciones provoca que la instrucción que sucede a aquella con la cual posee dependencias, no pueda ejecutarse en el ciclo de reloj que le corresponde, ya que ha de esperar algún resultado para poder efectuar su ejecución. Denominamos riesgo (*hazard*) a esta situación que impide a una instrucción la ejecución de sus etapas al depender de otra anterior. Los riesgos se traducen en una parada (*stall*) en el flujo del *pipeline*.

La causa de los riesgos puede ser variada. Los que podemos encontrar en el procesador DLX son los siguientes:

1. Riesgos de datos: son aquellos en los que una instrucción necesita de un resultado obtenido en una instrucción previa. Podemos distinguir:

   - **Riesgo RAW** (read after write): una instrucciń intenta leer un valor calculado en una instrucción previa. Por ejemplo:

     ```
     add r3,r2,r1 ; r3=r2+r1
     add r4,r3,r3 ; r4=r3+r3
     ```

   - **Riesgo WAW** (write after write): una instrucción intenta escribir un operando antes que una instrucción previa que también lo modifica. Este riesgo deriva en las instrucciones de punto flotante ya que su ejecución tarda un número de ciclos de reloj mayor al resto de instrucciones. En este tipo de instrucciones multiciclo pueden terminar instrucciones en un orden diferente al que se emitieron, por ejemplo en la siguiente secuencia la segunda instrucción es más rápida que la primera e intenta escribir f2 antes cuando no debería de ser así.

     ```
     multf f2,f1,f0 ; f2=f1*f0
     movf f2,f0 ; f2=f0
     ```

2. Riesgos estructurales: derivan de la imposibilidad del hardware en realizar a la vez dos actividades. Así, si tenemos solamente una unidad para multiplicar flotantes, mientras que se realiza una multiplicación es imposible realizar otra. En el siguiente ejemplo, la segunda instrucción ha de esperar a que la primera acabe la multiplicación para poder efectuar su multiplicación. Por ejemplo:

   ```
   multf f4,f1,f0 ; como ocupa mas de un ciclo
   multf f5,f2,f0 ; la unidad fmulEX esta ocupada
   ```

3. Riesgos de control: en aquellas instrucciones en las que se modifica el contador del programa (PC), como saltos, llamadas a subrutinas, interrupciones .... En DLX, a diferencia de MIPS, la dirección de salto es conocida en la etapa ID, con lo que la cancelación de la siguiente instrucción en caso de fallo es menos costosa.

Como se ha dicho, los riesgos se traducen en una parada (stall) en el flujo de las instrucciones dentro de la etapas de la segmentación, a la espera de que la dependencia causante se resuelva.

La forma más intuitiva de visualizar estos conceptos es la representación de un cronograma en el que representamos la evolución del tiempo en el eje horizontal y las diferentes instrucciones el eje vertical. El simulador winDLX, permite la visualización de dicho cronograma.

En el cronograma nos pueden aparecer diferentes tipos de paradas:

- **R-Stall**: causada por un riesgo RAW.
- **W-Stall**: causada por un riesgo WAW.
- **S-Stall**: causada por un riesgo estructural.
- **T-Stall**: causada por una interrupción (denominada excepción o *TRAP*). Espera a que se ejecute por completo la instrucción anterior.

- **Stall**: causada por una parada de una instrucción anterior.

Las dependencias RAW y WAW se muestran mediante una flecha roja.

Para paliar la posibilidad de riesgos, se introduce una mejora en la segmentación, que se denomina anticipación (*forwarding* o *short-circuit*). Dicha mejora consiste en facilitar el operando que produce la dependencia a la siguiente instrucción, en la etapa en que está disponible, sin esperar a que se llegue en la última etapa cuando es escrito en los registros. Así logramos una mayor eficiencia. El simulador WinDLX permite habilitar o deshabilitar la opción de *forwarding*. En caso de que esté activa, una flecha verde muestra las etapas a través de las cuales se anticipa el valor del operando. Por ejemplo en las instrucciones:

```
add r3,r2,r1 ; r3=r2+r1

add r4,r3,r3 ; r4=r3+r3
```

el valor de `r3` está disponible en la ALU a la salida de la etapa de ejecución (EX), con lo cual se puede pasar directamente a entrada de la etapa de ejecución (EX) de la siguiente instrucción sin necesidad de esperar a que sea guardado en `r3` (etapa WB).

Ejemplo

En este apartado se presenta un pequeño código ensamblador para DLX y se explica el proceso a seguir para su simulación con el simulador WINDLX.

El código ensamblador se carga desde la opción *FILE*. Primero se selecciona el nombre del archivo (*.s) (*SELECT*) y luego se carga (*LOAD*). Con ello cargamos en la memoria simulada el código. Desde la opción *FILE* podemos hacer un reset, bien del procesador (sin borrar la memoria) o de todo el sistema (incluyendo la memoria, borrándose por tanto el programa cargado). Como ejemplo cargaremos el siguiente código:

```
                .data 0
                ;comienzo de los datos
                .global dato1
dato1:          .word 1, 6
                .global dato2
dato2:          .float  17.0, 26.0

                .text 0x100
                ;comienzo del codigo
                .global start

start:          addi    r8,r0,#12       ; inicializa r8 a 12
                addi    r6,r0,#8        ; inizializa r6 a 8
```

```
                add     r10,r0,r0       ; inizializa r10 a 0
                lw      r1,0(r10)       ; r1 <- [r10]=[0]=1
                addi    r3,r1,#2        ; inizializa r3 a 2

lazo1:          add     r2,r3,r0
                subi    r3,r3,#1
                bnez    r3,lazo1        ; saltar si r3 no es cero
                add     r3,r3,r0
                addi    r11,r0,#4
                lw      r3,0(r11)       ; r3 <- [r11]=[4]=6
                addi    r1,r3,#4
                add     r4,r0,r0
                addi    r5,r0,#7

                lf      f3,0(r8)        ; f3 <- [r8]=[12]=26.0
                lf      f2,0(r6)        ; f2 <- [r6]=[8]=17.0
                multf   f4,f3,f2
                divf    f5,f4,f2
                addf    f5,f3,f2
                sf      0(r8),f5        ; Mem[r8]=Mem[12] <- f5
                j       fin             ; saltar al final del prog.
                nop
                nop
                nop
fin:            trap 0                  ; finalizar el programa
```

El código (`ejemplo.s`) consta de dos partes: datos y código. Las líneas que comienzan con un punto (p.e. `.data 0`) son *directivas*. La parte de datos empieza en la dirección 0, y se define con la directiva `.data 0x0`. La parte de código empieza en la dirección 0x100, y se indica con la directiva `.text 0x100`. Para que el simulador funcione correctamente hemos de inicializar estas posiciones (comienzo de los datos y código). Para ello en la opción *MEMORY SYMBOLS* del simulador (donde se muestran los valores de las etiquetas), hemos de cambiar los valores de las etiquetas *TEXT* y *DATA*, a los valores 0x100 y 0x0 respectivamente. De esta forma indicamos al simulador donde comienzan los datos y la primera línea ejecutable.

La última línea del programa es una llamada a la interrupción (excepción 0) que finaliza la ejecución del programa.

En la opción *CONFIGURATION* debemos de asegurarnos que la configuración de punto flotante (*FLOAT POINT STAGES*) está puesta a los siguientes valores (aconsejados, no obligatorio):

	count	delay
addition units	1	2
multiplication units	1	5
division units	1	19

(con esto decimos el número de unidades de punto flotante y el número de ciclos necesarios para la ejecución de estas etapas). En *MEMORY SIZE* nos aseguraremos que la memoria tiene un tamaño de 0x8000 bytes.

En la opción *CONFIGURATION* también habilitamos o deshabilitamos la anticipación con *ENABLE FORWARDING*.

El manejo del simulador es bastante intuitivo, constando de diferentes ventanas en las que se muestra: el cronograma de ejecución, el código cargado, contenido de los registros, estado del pipeline, puntos de ruptura y estadísticas de ejecución. Es posible también ver el contenido de la memoria en la opción *DISPLAY* de *MEMORY*.

Para la ejecución del programa, dentro de la opción *EXECUTE*, podemos ejecutar el código en su totalidad (*RUN*), ejecutar un número de ciclos (*MULTIPLE CYCLES*), ejecutar hasta una posición dada (*RUN TO*), o ejecutar un sólo ciclo de reloj (*SINGLE CYCLE*). Cuando se ejecuten varios ciclos de reloj de una vez, se debe tener en cuenta que el simulador solo conserva los últimos, por lo que, si se ejecutan muchos de una vez, puede que se pierda la información de los primeros.

Ejercicio 1

1. **Ejecutar el código ejemplo1.s en el simulador DLX. Explica en pocas palabras que es lo que hace el código.**

2. **Realizar la ejecución completa del código y calcular el número de ciclos que consume en el caso de que la anticipación esté activada y en el caso de que no lo esté. Calcular el CPI en ambos casos.**

	Activada	**No activada**
Ciclos		
CPI		

3. **Con la anticipación *forwarding* activa ¿Qué tipos de riesgos existen en el programa?**

Riesgos:	**Estructurales**	**Control**	**Datos**
Número:			

4. **En el caso de que sean dependencias de datos. ¿Cuáles son los registros causantes y de qué tipo son estas dependencias?.**

(Expresar las dependencias con la nomenclatura vista en clase.
Ejemplo: EX / MEM.RegistroRd = ID / EX.RegistroRs)

5. **Analizar las anticipaciones implementadas (entre qué etapas se dan).**

6. **Analizar los saltos ¿En qué etapa se efectúa el salto.**

7. **¿Es posible reducir el número de ciclos empleados en la ejecución del código mediante una reordenación del mismo?. En caso afirmativo indicar cómo quedaría el nuevo código y calcular el número de ciclos empleados en la ejecución y el CPI resultante. (Marca en el código original ”Primer programa” las modificaciones que creas necesarias)**

Ejercicio 2

Cargar el código ensamblador "Segundo programa" en el simulador WinDLX, prepare el simulador para su ejecución y responda a las cuestiones:

1. **Realice la ejecución completa del código y calcular el número de ciclos que consume en el caso de que la anticipación esté activada y no lo esté. Calcule el CPI en ambos casos.**

$$CPI = \frac{Ciclos\_consumidos}{instrucciones\_ejecutadas}$$

2. **Con la anticipación *forwarding* activa, ¿Qué tipo de riesgos se dan hasta que instrucción *sf* se ejecuta completamente por primera vez?. En el caso de que sean de datos, ¿Cuáles son los registros causantes de las dependencias? (Expresar las dependencias con la nomenclatura vista en clase).**

3. **¿Existe algún riesgo estructural (S-Stall) en la ejecución?. ¿Por qué?**

4. **En qué etapa del *pipeline*** de la instrucción de control ***bnez*** **se conoce la dirección del salto?. ¿Qué ocurre con la siguiente instrucción que sigue en memoria al salto?**

5. **¿Entre qué etapas del *pipeline* se produce la anticipación desde la instrucción** ***lf f0, 0(r1)*** **y la siguiente?**

6. **Configure la etapa de punto flotante de forma que el tiempo en ejecutar la multiplicación (fmulEX) sea el mismo que el de la suma (faddEX, 2 unidades de reloj). ¿Qué tipo de riesgo se elimina con esta modificación en la arquitectura del procesador?**

7. **Proponer una modificación (reordenación) que pueda mejorar el rendimiento del código "Segundo programa".**

```
;Primer programa
        .data 0x0 ;comienzo de los datos

        .global dato1
dato1:   .word 1, 6

        .global dato2
dato2:   .float 17.0, 26.0


        .text 0x100 ;comienzo del código fuente

        .global start

start: addi r8, r0, #12 ; Inicializa r8 a 12
       addi r6, r0, #8; Inicializa r6 a 8
       add r10, r0, r0; Inicializa r10 a 0
       lw  r1,0(r10)  ; r1<-[r10]=[r0]=1
       addi r3,r1,#2  ; Inicializa r3 a 2

lazo1: add r2,r3,r0
       subi r3,r3,#1
       bnez r3,lazo1  ; Saltar si r3 no es cero.
       add r3,r3,r0
       addi r11,r0,#4
       lw r3,0(r11)   ; r3<-[r11]=[4]=6.
       addi r1,r3,#4
       add r4,r0,r0
       addi r5,r0,#7
       lf f3,0(r8)    ; f3 <- [r8]=[12]=26.0
       lf f2,0(r6)    ; f2 <- [r6]=[8]=17.0
       multf f4,f3,f2
       divf f5,f4,f2
       addf f5,f3,f2
       sf 0(r8),f5    ; Mem(r8) = Mem[12] <- f5
       j fin          ; Saltar final del programa
       nop
       nop
       nop
fin:   trap 0         ; Finalizar el programa.


;Segundo programa
        .data 0x0

        .global a
a:      .float 3.1416

        .global b
b:      .float 2.1727

        .global x
x:      .float 1,2,3

        .global xtop
xtop:   .float 4

        .text 0x100 ; comienzo del código fuente
        .global start

start:add r3,r0,r0
       addi r2,r0,#4
       addi r1,r0,xtop
                                          lf f2,a
       lf f6,b
loop:  add r3,r3,r2
       sub r2,r2,#1
       lf f0,0(r1)
       multf f4,f0,f2
                                       addf f4,f4,f6
       sf 0(r1), f4
       sub r1,r1,#4
       bnez r1,loop
end:   nop
       trap #0
```