1 Simulador de Procesador Segmentado: WinDLX

1.1. 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.
- **N:** Número de instrucciones de las que se compone el programa.
- **Tck:** Duración de un ciclo de reloj.
- **Tcpu=**$Nc * Tck$**:** Tiempo requerido en ejecutar un programa.
- **CPI=**Nc/N**:** Número medio de ciclos por instrucción.

De aquí,

$$Tcpu = N * CPI * Tck$$

Si queremos hacer este tiempo lo más pequeño posible, tendríamos que disminuir alguno de los tres factores (N, CPI, Tck). 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 nº de instrucciones del 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.

1.2. 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 con desplazamiento: 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.

1.3. 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.

Los riesgos 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).

1.4. 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 (ejemplo.s):

```
                .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        ; inicializa r6 a 8
                add     r10,r0,r0       ; inicializa r10 a 0
                lw      r1,0(r10)       ; r1 <- [r10]=[0]=1
                addi    r3,r1,#2        ; inicializa r3 a 3

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.

1.5. Práctica 1

1. Ejecutar el código (`ejemplo.s`) en el simulador DLX. ¿El resultado de este código es el mismo al ejecutarlo en el DLX y en una máquina escalar (no segmentada) o por el contrario hace falta que el compilador/programador introduzca intrucciones `nop` entre algunas instrucciones para que esto sea así?

2. Realizar la ejecución completa del código y calcular el número de ciclos que consume en el caso de que la anticipación (forwarding o cortocircuitos) 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, especifica cuántos ciclos se pierden por paradas (stall o burbujas) en cada uno de los posibles tipos de riesgos. En los riesgos de datos, desglosa el nº de ciclos que se pierden en cada uno de los riesgos posibles (RAW o WAW).

Riegos:	Estructurales	Control	Datos
Ciclos:			

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

   - RAW:
   - WAW:

5. ¿Que tipo de estrategia se emplea para los saltos (detener siempre, salto retardado, predicción estática, etc.) ? ¿Cómo aparece reflejado en el diagrama de ciclo de reloj (clock cycle diagram)?

6. Analizar los cortocircuitos implementados (entre qué etapas se dan).

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 nuevo número de ciclos empleados en la ejecución y el CPI resultante.

   Marca con flechas en el codigo original las modificaciones que crees necesarias:

```
                .text 0x100
                ;comienzo del codigo
                .global start

start:          addi    r8,r0,#12
                addi    r6,r0,#8
                add     r10,r0,r0
                lw      r1,0(r10)
                addi    r3,r1,#2

lazo1:          add     r2,r3,r0
                subi    r3,r3,#1
                bnez    r3,lazo1
                add     r3,r3,r0
                addi    r11,r0,#4
                lw      r3,0(r11)
                addi    r1,r3,#4
                add     r4,r0,r0
                addi    r5,r0,#7

                lf      f3,0(r8)
                lf      f2,0(r6)
                multf   f4,f3,f2
                divf    f5,f4,f2
                addf    f5,f3,f2
                sf      0(r8),f5
                j       fin
                nop
                nop
                nop
fin:            trap 0
```

CPI resultante: ☐

Apéndice A

Directivas e instrucciones del ensamblador DLX

A.1. Directivas

While the assembler is processing an assembly file, the data and instructions it assembles are placed in memory based on either a text (code) or data pointer. Which pointer is used is not selected by the type of information, but whether the most recent directive was .data or .text. The program initially loads into the text segment. By default, the code is loaded at location CODE (initially set to 0x100) and the data are loaded at location DATA (initially set to 0x1000).

- `.align n` Cause the next data/code loaded to be at the next higher address with the lower `n` bits zeroed (the next closest address greater than or equal to the current address that is a multiple of 2^n (e.g. `.align 2` means the next word begin)
- `.ascii string1,...` Store the `strings` listed on the line in memory as a list of characters. The strings are not terminated by a 0 byte.
- `.asciiz string1,...` Similar to .ascii, except each string is terminated by a 0 byte.
- `.byte byte1,byte2,...` Store the bytes listed on the line sequentially in memory.
- `.data [address]` Cause the following code and data to be stored in the data area. If an address was supplied, the data will be loaded starting at that address, otherwise, the last value for the data pointer will be used. If we were just reading data based on the text (code) pointer, store that address so that we can continue from there later (on a .text directive).
- `.double number1,...` Store the `numbers` listed on the line sequentially in memory as double-precision Floating point numbers.
- `.global label` Make the label available for reference by code found in files lo aded after this file.
- `.space size` Move the current storage pointer forward size bytes (to leave some empty space in memory).
- `.text [address]` Cause the following code and data to be stored in the text (code) area. If an address was supplied, the data will be loaded starting at that address, otherwise, the last value for the text pointer will be used. If we were just reading data based on the data pointer, store that address so that we can continue from there later (on a .data directive).

- `.word word1,word2,...` Store the word listed on the line sequentially in memory.

A.2. Conjunto de instrucciones

Instructions for Data Transfer

Move data between registers and memory, or between integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR:

```
LB Rd,Adr       Load byte (sign extension)
LBU Rd,Adr      Load byte (unsigned)
LH Rd,Adr       Load halfword (sign extension)
LHU Rd,Adr      Load halfword (unsigned)
LW Rd,Adr       Load word
LF Fd,Adr       Load single-precision Floating point
LD Dd,Adr       Load double-precision Floating point
SB Adr,Rs       Store byte
SH Adr,Rs       Store halfword
SW Adr,Rs       Store word
SF Adr,Fs       Store single-precision Floating point
SD Adr,Fs       Store double-precision Floating point
MOVI2FP Fd,Rs   Move 32 bits from integer registers to FP registers
MOVI2FP Rd,Fs   Move 32 bits from FP registers to integer registers
MOVF Fd,Fs      Copy one Floating point register to another register
MOVD Dd,Ds      Copy a double-precision pair to another pair
MOVI2S SR,Rs    Copy a register to a special register (not
                implemented!)
MOVS2I Rs,SR    Copy a special register to a GPR (not implemented!)
```

Integer arithmetic and logical instructions

Operations on integer or logical data in GPRs; overflows by signed arithmetics are not reported.

```
ADD Rd,Ra,Rb    Add
ADDI Rd,Ra,Imm  Add immediate (all immediates are 16 bits)
ADDU Rd,Ra,Rb   Add unsigned
ADDUI Rd,Ra,Imm Add unsigned immediate
SUB Rd,Ra,Rb    Subtract
SUBI Rd,Ra,Imm  Subtract immediate
SUBU Rd,Ra,Rb   Subtract unsigned
SUBUI Rd,Ra,Imm Subtract unsigned immediate
MULT Rd,Ra,Rb   Multiply signed
MULTU Rd,Ra,Rb  Multiply unsigned
DIV Rd,Ra,Rb    Divide signed
DIVU Rd,Ra,Rb   Divide unsigned
AND Rd,Ra,Rb    And
ANDI Rd,Ra,Imm  And immediate
```

```
OR Rd,Ra,Rb      Or
ORI Rd,Ra,Imm    Or immediate
XOR Rd,Ra,Rb     Xor
XORI Rd,Ra,Imm   Xor immediate
LHI Rd,Imm       Load high immediate - loads upper half of register
                 with immediate
SLL Rd,Rs,Rc     Shift left logical
SRL Rd,Rs,Rc     Shift right logical
SRA Rd,Rs,Rc     Shift right arithmetic
SLLI Rd,Rs,Imm   Shift left logical 'immediate' bits
SRLI Rd,Rs,Imm   Shift right logical 'immediate' bits
SRAI Rd,Rs,Imm   Shift right arithmetic 'immediate' bits
S__ Rd,Ra,Rb     Set conditional: "__" may be EQ, NE, LT, GT, LE
                 or GE
S__I Rd,Ra,Imm   Set conditional immediate: "__" may be EQ, NE, LT,
                 GT, LE or GE
S__U Rd,Ra,Rb    Set conditional unsigned: "__" may be EQ, NE, LT,
                 GT, LE or GE
S__UI Rd,Ra,Imm Set conditional unsigned immediate: "__" may be
                 EQ, NE, LT, GT, LE or GE
NOP              No operation
```

Control instructions

Control is handled through a set of jumps and a set of branches. The four jump instructions are differentiated by the two ways of specifying the destination address and by wether or not a link is made. Two jumps use a 26-bit offset added to the program counter to determine the destination address; the other two specify a register that is the destination address.

There are two flavors of jumps: plain jump, and jump and link (used for procedure call). The latter places the return address in R31. All branches are conditional. The branch condition is specified by the instruction, which may test the register source for zero or nonzero; this may be a data value or the result of a compare. The branch target address is specified with a 16-bit signed offset that is added to the program counter.

In WINDLX, the jumps and the branches are completed by the end of the ID cycle to reduce the number of control stalls. In the DLX pipeline, the predict-not-taken scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, we neeed to stop the pipeline and restart the fetch.

```
BEQZ Rt,Dest     Branch if GPR equal to zero; 16-bit offset from PC
BNEZ Rt,Dest     Branch if GPR not equal to zero; 16-bit off. from PC
BFPT Dest        Test comparison bit in the FP status register
                 (true) and branch;  16-bit offset from PC
BFPF Dest        Test comparison bit in the FP status register
                 (false) and branch;  16-bit offset from PC
J Dest           Jump: 26-bit offset from PC
JR Rx            Jump: target in register
JAL Dest         Jump and link: save PC+4 to R31; target is
                 PC-relative
```



```
JALR Rx          Jump and link: save PC+4 to R31; target is
                 a register
TRAP Imm         Transfer to operating system at a vectored address;
```

Floating Point Instructions

Floating point instructions manipulate the floating point registers and indicate whether the operation to be performed is single or double precision.

Single-precision operations can be applied to any of the registers, while double-precision operations apply only to even-odd pair (e.g. F4,F5), which is designated by the even register number.

```
ADDD Dd,Da,Db    Add double-precision numbers
ADDF Fd,Fa,Fb    Add single-precision numbers
SUBD Dd,Da,Db    Subtract double-precision numbers
SUBF Fd,Fa,Fb    Subtract single-precision numbers.
MULTD Dd,Da,Db   Multiply double-precision Floating point numbers
MULTF Fd,Fa,Fb   Multiply single-precision Floating point numbers
DIVD Dd,Da,Db    Divide double-precision Floating point numbers
DIVF Fd,Fa,Fb    Divide single-precision Floating point numbers
CVTF2D Dd,Fs     Converts from type single-precision to type
                 double-precision
CVTD2F Fd,Ds     Converts from type double-precision to type
                 single-precision
CVTF2I Fd,Fs     Converts from type single-precision to type integer
CVTI2F Fd,Fs     Converts from type integer to type single-precision
CVTD2I Fd,Ds     Converts from type double-precision to type integer
CVTI2D Dd,Fs     Converts from type integer to type double-precision
```