ESTRUCTURA DE COMPUTADORES

Práctica 4: Segmentación del procesador

1. Objetivos

- Conocer el manejo del simulador de computador DLX segmentado.
- Analizar la influencia de los conflictos de datos y de control en las prestaciones de la ruta de datos segmentada.

2. Desarrollo

En esta práctica vamos a realizar experimentos sobre una ruta de datos segmentada, muy similar a la ruta de datos del MIPS vista en clase. Principalmente, vamos a ejecutar código sobre dicha ruta habilitando diferentes técnicas de resolución de conflictos de datos. Esto nos permitirá entender mejor y profundizar en los conflictos de datos y las distintas alternativas en su posible resolución, como la inserción de ciclos de parada, la inserción de instrucciones `nop` o el empleo de cortocircuitos. También vamos a tratar en esta práctica los conflictos de control, generados por el uso de instrucciones de salto, ya sean de salto incondicional o de salto condicional, ensayando algunas de las técnicas de resolución disponibles, como la predicción de saltos.

Vamos a utilizar el ensamblador DLX (muy similar al MIPS, con ligeras diferencias de sintaxis). La ruta de datos está compuesta de cinco etapas que se corresponden con la búsqueda de la instrucción, su descodificación y búsqueda de operandos en el banco de registros, la ejecución en la UAL, el acceso a la memoria de datos tanto en lectura como en escritura, y la escritura del resultado en el bando de registros. Como resultados obtendremos el tiempo de ejecución de los programas sobre la ruta de datos, medido en ciclos, así como el número de instrucciones ejecutadas, el número de ciclos de parada del procesador y el CPI obtenido. En todos los casos deberemos obtener el CPI que se obtiene restando 4 (número de etapas menos uno) al número de ciclos totales de ejecución.

Las diferencias entre DLX y MIPS que se pueden apreciar en esta práctica son las siguientes:

- Los registros se denominan r1, r2,... en DLX, a diferencia de $1, $2,... en el MIPS. El registro r0 continúa estando cableado a 0.
- Los valores inmediatos para las instrucciones aritméticas se expresan mediante el prefijo "#". Por ejemplo, `addi r4,r1,#64` en DLX equivale a `addi $4,$1,64` en MIPS. Además se puede utilizar una etiqueta (que lógicamente tendrá asignado un valor) para especificar un valor inmediato. Por ejemplo, si x=0 entonces `addi r1,r0,x` carga en r1 el valor 0.
- La instrucción DLX `seq r5,r4,r1` (*set if equal*) escribe un 1 en r5 si r4=r1; en caso contrario, escribe un cero en r5.
- La instrucción DLX `beqz r5,loop` (*branch on equal to zero*) salta a la etiqueta `loop` si y solo si el registro r5 vale 0.

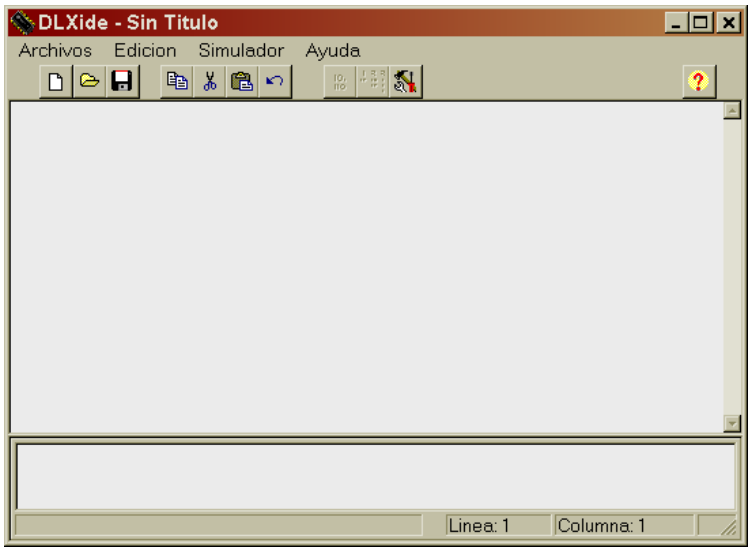
- La instrucción de almacenamiento en el DLX cambia el orden de los parámetros con respecto a la sintaxis MIPS. Por ejemplo, la instrucción MIPS `sw $14,0($3)` debe escribirse `sw 0(r3),r14` en el DLX.
- Finalmente, el programa DLX termina cuando se alcanza la instrucción `trap #0`.

3. Simulador DLXide

Vamos a utilizar el simulador DLXide, contenido en la carpeta de la práctica (en poliformaT). El simulador es un único ejecutable, el cual no necesita instalación.

El simulador es capaz de simular ciclo a ciclo la ejecución de instrucciones del DLX, así como visualizar el avance de las mismas por la ruta de datos de la máquina. Soporta todas las instrucciones del DLX que operan sobre el banco de registros de enteros. Hay memoria de instrucciones y de datos separadas (arquitectura Harvard). Los registros se escriben y leen en el primer y segundo semiciclo de reloj, respectivamente.

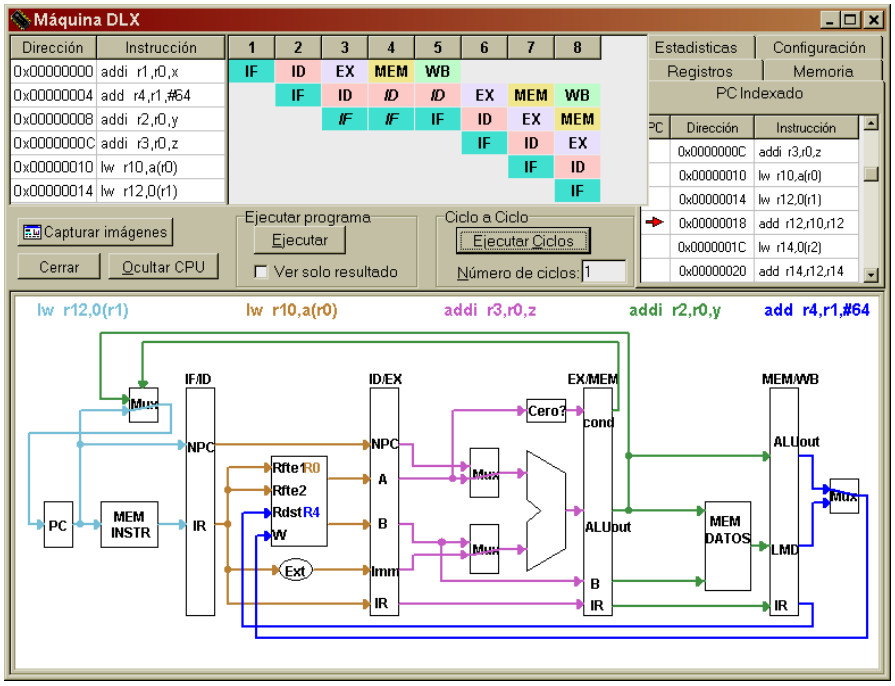
Al iniciar DLXide se mostrará la ventana del programa, con los menús disponibles. La siguiente figura muestra el aspecto de esta ventana.



Para configurar el simulador, acceded al menú *Simulador*, opción *Configuración DLX*. El programa abrirá una ventana de diálogo mostrando las estrategias disponibles para resolver los riesgos de datos y de control. Por defecto, se insertarán ciclos de parada (primera de las opciones del menú *Configuración DLX)*. La configuración debe quedar pues como se muestra a continuación:

El siguiente paso es **cargar el fichero con el programa**. Para ello, acceded al menú *Archivos*, opción *Abrir*, y seleccionad el fichero adecuado. Tras cargar un programa, éste debe ensamblarse (menú *Simulador*, opción *Ensamblar*). En el caso de que hubieran errores, se mostrarían en la parte inferior de la ventana del programa. Tras corregir los errores hay que volver a ensamblarlo. Cuando el programa se ensambla sin errores, éste se almacena en la memoria de la máquina simulada y se informa de ello al usuario.

Para **iniciar la simulación** debe utilizarse el menú *Simulador*, opción *Ejecutar*. Se abrirá una nueva ventana en la que se visualizará el diagrama instrucciones-tiempo, la ruta de datos, así como una ventana con múltiples fichas que permiten inspeccionar el estado de la máquina. La siguiente figura muestra la ventana de simulación tras la ejecución de 8 pasos de simulación:



La ficha *Configuración* permite recordar cuáles son las estrategias seleccionadas para resolver los conflictos. La ficha *Registros* permite visualizar el contenido de los registros de uso general. Haciendo doble click en el campo "valor" se puede alterar el valor de los mismos. Pulsando el botón derecho se puede cambiar la base de numeración. Las fichas *PC Indexado* y *Memoria*

permiten visualizar el contenido de la memoria de instrucciones y de datos, respectivamente. Finalmente, la ficha *Estadísticas* indica el número de ciclos consumidos, las instrucciones ejecutadas, los ciclos de parada introducidos y los cortocircuitos aplicados. Los cortocircuitos son una solución hardware para solucionar los conflictos producidos por dependencia de datos con el objetivo de mejorar el rendimiento del procesador. Esta solución no se estudiará en ETC, por tanto no utilizaremos esta opción al simular.

El simulador permite la ejecución del programa ciclo a ciclo o avanzar varios ciclos (botón Ejecutar Ciclos) o ejecutarlo completamente hasta encontrar una instrucción `trap #0` (botón Ejecutar). Tras cada ciclo de reloj se actualiza el diagrama instrucciones–tiempo. Cuando se inserta un ciclo de parada, las fases en las que se mantienen las mismas instrucciones se reflejan en cursiva. También se actualiza la ruta de datos de la máquina. A cada nueva instrucción buscada se le asigna un nuevo color que se utiliza durante todo el recorrido de la misma por la unidad de instrucción. Cuando en una de las fases no hay ninguna instrucción, se visualiza el texto –nop–. **No debe confundirse esta indicación con la auténtica instrucción `nop`.**

4. Conflicto de datos resueltos con ciclos de parada

Vamos a empezar por ejecutar un programa inicial. El programa en cuestión es `aritm1.s` que se muestra a continuación:

```
; Varias operaciones aritmeticas
      .text

start:
1)    add  r1,r0,r0     ; r1 = 0
2)    addi r2,r0,#64    ; r2 = 64
3)    addi r3,r2,#10    ; r3 = r2 + 10 = 74
4)    sub  r4,r3,r2     ; r4 = r3 – r2 = 10
5)    trap #0          ; Fin del programa
```

Como podemos apreciar, el código no realiza ninguna operación global útil, tan solo inicializa algunos registros. Ahora bien, nos vendrá bien para ejercitar los conflictos de datos.

Ejercicio 1: Anotad en la siguiente tabla las dependencias de datos que existen en el programa anterior que ocasionan conflicto en la ruta de datos.

	Registro	Número de instrucción en que se escribe	Número de instrucción en que se lee
Dependencia nº			
Dependencia nº			
Dependencia nº			
Dependencia nº			
…			

Ejercicio 2: Rellenad la siguiente tabla asumiendo la ejecución del código en una ruta de datos donde los conflictos de datos se resuelven con la introducción de ciclos de parada por parte del procesador.

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

Ejercicio 3: Cargad, ensamblad, y ejecutad paso a paso el código del fichero aritm1.s. Recordad que la configuración del DLX debe ser la anterior (insertar ciclos de parada para conflictos de datos, y la opción forwarding deshabilitada). Dibujad el diagrama instrucciones/ciclo resultado de la ejecución.

Instrucción	1	2	3	4	5	6	7	8	9	10	11	12	13

5. Conflicto de datos resuelto por instrucciones nop

Como hemos visto, aparecen ciertos conflictos de datos que afectan a las prestaciones que obtenemos. En concreto, durante algunos ciclos las instrucciones se bloquean. Una alternativa es la introducción de instrucciones `nop`, las cuales permiten que las instrucciones están a una distancia "segura" para que la dependencia de datos no se convierta en un conflicto. Se deberán introducir tantas instrucciones `nop` como ciclos de parada se hayan obtenido, y de tal forma que las instrucciones con dependencias de datos están a una distancia mínima de 3 instrucciones.

Ejercicio 4: Modificad el código `aritm1.s`, generando un nuevo fichero denominado `aritm1_nop.s`, el cuál tenga las instrucciones `nop` adecuadas para no generar ningún conflicto de datos. Trascribid el código resultante en el siguiente cuadro:

Ejercicio 5: Rellenad la siguiente tabla asumiendo la ejecución del código `aritm1_nop.s` en una ruta de datos donde los conflictos de datos se resuelven con la introducción de instrucciones `nop`.

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

Ejercicio 6: ¿Has obtenido un tiempo de ejecución distinto? Tanto en caso afirmativo, como en caso negativo, razona la respuesta.

6. Conflictos de datos con memoria

En los apartados anteriores hemos estado trabajando con conflictos de datos generados en el acceso a la ALU. Ahora bien, podemos tener tambien conflictos de datos en el acceso a la memoria de datos. Estos conflictos ocurren cuando queremos escribir un dato en memoria que todavía no se ha consolidado en el banco de registros. También tenemos un conflicto de datos cuando queremos acceder a un valor que todavía no ha llegado de memoria de datos (producido por una instrucción lw).

Vamos, en este apartado, a ejercitar este tipo de dependencias. Para ello disponemos del código en el fichero `mem.s`, el cuál se muestra a continuación:

```
; Varias operaciones con memoria

      .data
A:    .word 0
B:    .word 20
C:    .word 30
D:    .word 0

      .text
start:
1)    addi r1, r0, #10  ; r1 = 10
2)    sw A(r0), r1      ; almacenar 10 en A
3)    lw r1, B(r0)      ; r1 = 20
4)    lw r2, C(r0)      ; r2 = 30
5)    add r3, r1, r2    ; r3 = r1 + r2 = 50
6)    sw D(r0), r3      ; almacenar 50 en D
7)    trap #0           ; Fin del programa
```

Ejercicio 7: Cargad, ensamblad, y ejecutad el código en el simulador DLXide. Obtened los resultados para la siguiente tabla.

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

Ejercicio 8: A partir del fichero `mem.s`, modificad el fichero, generando `mem_nop.s`, de tal forma que los ciclos de parada se eliminen, utilizando para ello instrucciones `nop`. Transcribid el código resultante en el cuadro siguiente:

Ejercicio 9: Utilizad el simulador DLXide para obtener los resultados solicitados en la tabla siguiente.

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

7. Conflicto de control

Los conflictos de control son debidos a cambios en el flujo de ejecución del programa. El código del fichero `bucle1.s` realiza una operación con vectores (C[i] = 2*A[i] + B[i] + 1). El código es el siguiente:

```
; C[i] = 2*A[i] + B[i]+1
      .data
A:    .word 0,1,2,3,4,5,6,7,8,9
B:    .word 10,11,12,13,14,15,16,17,18,19
C:    .space 40

.text

start:
1)    addi r1, r0, #10        ; r1 = numero de iteraciones
2)    addi r2, r0, #0         ; r2 = desplazamiento a elemento de A
3)    addi r3, r0, #0         ; r3 = desplazamiento a elemento de B
4)    addi r4, r0, #0         ; r4 = desplazamiento a elemento de C
bucle:
5)    lw r6, A(r2)            ; lectura A[i]
6)    add r6, r6, r6          ; r6 = 2*A[i]
7)    lw r7, B(r3)            ; lectura B[i]
8)    addi r7, r7, #1         ; r7 = B[i]+1
9)    add r8, r6, r7          ; r8 = 2*A[i]+B[i]+1
10)   sw C(r4), r8            ; C[i] = r8
11)   addi r1,r1,#-1          ; r1 = r1 – 1
12)   addi r2, r2, #4         ; r2 = r2 + 4
13)   addi r3, r3, #4         ; r3 = r3 + 4
14)   addi r4, r4, #4         ; r4 = r4 + 4
15)   seq r5, r1, r0          ; r5 = (r1 == 0)
16)   beqz r5, bucle          ; salta si r5 == 0
17)   trap #0                 ; Fin del programa
```

Ejercicio 10: Anotad en la siguiente tabla las dependencias de datos que existen en el programa anterior que ocasionan conflicto en la ruta de datos.

	Registro	Número de instrucción en que se escribe	Número de instrucción en que se lee
Dependencia nº			
Dependencia nº			
Dependencia nº			
Dependencia nº			
Dependencia nº			
Dependencia nº			
Dependencia nº			

Vamos ahora a ejecutar el código. La configuración que debemos poner es la de resolución de conflictos de control mediante ciclos de parada (3 ciclos), y resolución de conflictos de datos mediante ciclos de parada también. La configuración es la siguiente:

Ejercicio 11: Cargad, ensamblad, y ejecutad paso a paso el código del fichero `bucle1.s`. Obtened los siguientes resultados:

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

Ejercicio 12: De todos los ciclos de parada del procesador, calculad cuántos son debidos a conflictos de datos y cuántos a conflictos de control. Ejecutad el código con el simulador DLXide ciclo a ciclo, identificando cada uno de los ciclos de parada en una iteración.

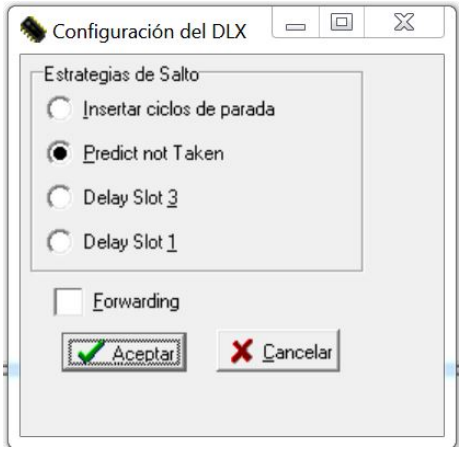
Ciclos de parada por conflictos de datos total:

Ciclos de parada por conflictos de control total:

8. Predicción de saltos para los conflictos de control

Anteriormente los conflictos de control los hemos resuelto mediante inserción de ciclos de parada, penalizando con ello las prestaciones. Una alternativa para solucionarlos es el empleo de técnicas de predicción de saltos. La única técnica de predicción disponible en DLX es la de *Predict not Taken*.

Vamos ahora a ejecutar el código. La configuración que debemos poner es la de resolución de conflictos de control mediante *Predict not Taken*. La configuración es la siguiente:



Ejercicio 13: Cargad, ensamblad, y ejecutad el código del fichero `bucle1.s`. Obtened los siguientes resultados:

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

Ejercicio 14: Indica si se han logrado eliminar todos los conflictos de control (en el caso de que el número de ciclos de parada sea superior a los que quedaron sin resolver anteriormente a causa de los conflictos de datos). **Justifica** por qué la técnica de predicción empleada no resulta eficaz en este caso **¿Qué alternativa de solución se podría haber ensayado?**

Ejercicios de ampliación (opcional) para conocer más

1. Conflictos de datos resueltos por reordenación de código

Una alternativa eficaz de diseño, para mitigar los conflictos de datos, es la reordenación de código. En vez de incluir instrucciones `nop` entre dos instrucciones que introducen un conflicto, con la reordenación se incluyen instrucciones del propio código, evitando así la penalización de prestaciones por incremento del número de instrucciones a ejecutar. Asimismo, la reordenación de código, a diferencia de la técnica de cortocircuitos, es una alternativa sin coste hardware alguno. Ahora bien, no todas las instrucciones del código pueden ser reordenadas ya que no pueden adelantar a las instrucciones que producen sus datos ni ser adelantadas por las instrucciones que consumen los datos que éstas producen. Tampoco instrucciones que producen sobre el mismo registro (o posición de memoria) pueden desordenarse. En estos casos no queda otra solución que insertar instrucciones nop, o emplear cualquiera de las técnicas hardware (ciclos de parada o cortocircuitos) vistas anteriormente en caso que el procesador las soporte. La tarea de reordenar el código para evitar los conflictos corre habitualmente a cargo del compilador.

Vamos a ejercitar la reordenación con un código nuevo, ubicado en el fichero `aritm2.s`. El código es el siguiente:

```
; Varias operaciones aritmeticas
; R5 = R4 – R3 + R2
; R6 = R4 + R1 – R3

      .text
start:
1)    addi r1, r0, #10  ; r1 = 10
2)    addi r2, r0, #20  ; r2 = 20
3)    addi r3, r0, #30  ; r3 = 30
4)    addi r4, r0, #40  ; r4 = 40
5)    sub r5, r4, r3    ; r5 = r4 – r3
6)    add r5, r5, r2    ; r5 = r5 + r2
7)    add r6, r4, r1    ; r6 = r4 + r1
8)    sub r6, r6, r3    ; r6 = r6 – r3
9)    trap #0           ; Fin del programa
```

El código realiza dos operaciones aritméticas obteniendo r5 y r6, a partir de los valores de los registros r1 a r4.

Ejercicio 15: Utilizando el simulador DLXide, cargad, ensamblad, y ejecutad el código aritm2.s, obteniendo los datos de la tabla siguiente:

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	

Ejercicio 16: A partir del fichero aritm2.s, modificad el fichero, generando aritm2_reord.s, de tal forma que, utilizando la reordenación del código, se obtenga el mínimo tiempo de ejecución. Se pueden utilizar instrucciones `nop` en caso de no resolver todos los conflictos de datos al reordenar. Ello significa que el procesador no soporta el empleo de cortocircuitos ni la inserción de ciclos de parada para la resolución de conflictos de datos. Transcribid el código resultante en el cuadro siguiente:

Ejercicio 17: Utilizad el simulador DLXide para obtener los resultados solicitados en la tabla siguiente. NOTA IMPORTANTE: Una vez ejecutado el código debeis comprobar que la ejecución del código ha sido correcta (los registros r5 y r6 deben tener los valores finales 30 y 20, respectivamente).

	Valor
Número de instrucciones ejecutadas	
Número de ciclos de parada del procesador	
Número de ciclos totales	
CPI	