PRÁCTICA 2a:
"UNIDAD DE INSTRUCCIÓN SEGMENTADA (I)"

Arquitectura e Ingeniería de Computadores (3º curso)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Objetivos:

- Conocer el manejo de un simulador de computador segmentado.
- Analizar la influencia de los riesgos de control y datos en la prestaciones de la unidad de instrucción segmentada.
- Realizar programas en lenguaje ensamblador MIPS.

Desarrollo:

El simulador del computador MIPS de 5 etapas.

El simulador es capaz de simular ciclo a ciclo la ejecución de instrucciones del MIPS, así como el avance de las mismas por la ruta de datos de la máquina. Soporta un subconjunto de las instrucciones del MIPS que operan sobre el banco de registros entero. Tiene cache de instrucciones y de datos separadas (arquitectura *Harvard*). Los registros se escriben y leen en el primer y segundo semiciclo de reloj, respectivamente. Los riesgos de control pueden resolverse aplicando diversas estrategias: inserción de ciclos de parada (*stalls*), *predict-not-taken* y salto retardado, con tres posibilidades de *delay-slot* (uno, dos y tres, respectivamente). Los riesgos de datos pueden resolverse insertando ciclos de parada o aplicando la técnica de la anticipación (*forwarding*). Nótese que la ruta de datos simulada cambia en función de las estrategias empleadas para resolver los riesgos.

El simulador se ejecuta desde la línea de órdenes:

```
mips-m -s resultados -d riesgos-datos -c riesgos-control -f archivo.s
```

donde:

- *resultados*: indica cómo se ofrecerá el resultado de la simulación. Hay varias opciones:
  - **tiempo**: Muestra el tiempo de ejecución en el terminal.
  - **final**: Muestra el tiempo de ejecución, los registros y el contenido de la memoria tras la ejecución en el terminal.
  - **html**(*): Genera varios archivos html con el estado de la ejecución ciclo a ciclo así como los resultados finales. Los resultados se visualizan abriendo en un navegador el archivo **index.html**. Esta es la opción por defecto.
  - **html-final**: Genera un archivo html **final.html** con el resultado final de la ejecución.

- *riesgos de datos*: indica cómo se resuelven los riesgos de datos. Hay tres opciones:
  - **n**: No hay lógica para resolver los riesgos de datos.
  - **p** Se resuelven los riesgos de datos insertando ciclos de parada.
  - **c**: Se resuelven los riesgos de datos mediante la técnica de la anticipación o cortocircuito, insertando asímismo los ciclos de parada necesarios.
- *riesgos de control*: indica cómo se resuelven los riesgos de control. Hay cinco opciones:
  - **s3** Se resuelven los riesgos de control insertando tres ciclos de parada.
  - **s2** Se resuelven los riesgos de control insertando dos ciclos de parada.
  - **s1** Se resuelven los riesgos de control insertando un ciclo de parada.
  - **pnt3**: Se resuelven los riesgos de control mediante *predict-not-taken*, insertando tres ciclos de parada si el salto es efectivo.
  - **pnt2**: Se resuelven los riesgos de control mediante *predict-not-taken*, insertando dos ciclos de parada si el salto es efectivo.
  - **pnt1**: Se resuelven los riesgos de control mediante *predict-not-taken*, insertando un ciclos de parada si el salto es efectivo.
  - **ds3**: Se resuelven los riesgos de control mediante la técnica del salto retardado, con valor del *delay-slot*=3.
  - **ds2**: Se resuelven los riesgos de control mediante la técnica del salto retardado, con valor del *delay-slot*=2.
  - **ds1**: Se resuelven los riesgos de control mediante la técnica del salto retardado, con valor del *delay-slot*=1.
- *archivo.s*: es el nombre del archivo que contiene el código en ensamblador.

Ejemplo de programa para MIPS

A continuación, se muestra el código ensamblador correspondiente a un bucle que realiza la siguiente operación vectorial: $\vec{Z} = a + \vec{X} + \vec{Y}$:

```
    .data
    ; Vector x
x:  .dword 0,1,2,3,4,5,6,7,8,9
    .dword 10,11,12,13,14,15

    ; Vector y
y:  .dword 100,100,100,100,100,100,100,100,100,100
    .dword 100,100,100,100,100,100

    ; Vector z
    ; 16 elementos son 16*8=128 bytes
z:  .space 128

    ; Escalar a
```

```
a:  .dword -10

    ; Codigo
    .text

start:
    dadd r1,r0,x
    dadd r4,r1,#128 ; 16*8
    dadd r2,r0,y
    dadd r3,r0,z
    ld r10,a(r0)

loop:
    ld r12,0(r1)
    dadd r12,r10,r12
    ld r14,0(r2)
    dadd r14,r12,r14
    sd r14,0(r3)
    dadd r1,r1,#8
    dadd r2,r2,#8
    dadd r3,r3,#8
    seq r5,r4,r1
    beqz r5,loop
    trap #0         ; Fin de programa
```

1. Este programa está almacenado en el fichero `apxpy.s`. Se puede lanzar a ejecución mostrando resultados detallados y resolviendo riesgos de datos y de control mediante 3 ciclos de parada con la orden siguiente:

   ```
   mips-m -d p -c s3 -f apxpy.s
   ```

   Seguidamente, abriremos el archivo **index.html** mediante el navegador, el cual muestra la configuración del procesador y el contenido inicial de la memoria, así como unos enlaces que permiten navegar por los resultados:

   - INICIO. Muestra la configuración del procesador y el contenido inicial de la memoria.
   - FINAL. Muestra los resultados de prestaciones tras la ejecución, la configuración del procesador y el contenido final de la memoria.
   - Estado. Muestra el diagrama instrucciones–tiempo correspondiente a la ejecución del programa, así como el estado de la unidad de ejecución en un ciclo dado, indicando qué instrucción ocupa cada una de las etapas del procesador. Cada instrucción se muestra en diferente color. Finalmente, muestra el contenido de los registros y de la memoria al final del ciclo analizado. En caso de operaciones de lectura o escritura, se utiliza como color de fondo en el registro o posición de memoria accedido el correspondiente la instrucción implicada. En esta página tenemos enlaces a las páginas de estado correspondientes a 1, 5 o 10 ciclos anteriores o posteriores al actual.

   Navegando por los archivos de estado podemos observar el avance de las instrucciones a lo largo de la unidad de instrucción segmentada así como la inserción de

ciclos de parada cuando se detectan riesgos. En el archivo de resultados finales se pueden obtener los resultados de prestaciones así como analizar el contenido de los registros y la memoria para comprobar la correcta ejecución del programa. En este caso, se debe haber almacenado en la dirección definida por la etiqueta *z* el vector con el resultado del programa. Anotar el número de instrucciones ejecutadas, el tiempo transcurrido y el CPI obtenido.

2. Seguidamente, ejecutaremos de nuevo el programa cambiando a *predict-not-taken* de tres ciclos de penalización la resolución de riesgos de control:

   ```
   mips-m -d p -c pnt3 -f apxpy.s
   ```

   Si analizamos la ejecución ciclo a ciclo para la primera iteración del bucle, podemos observar cómo se abortan las instrucciones buscadas incorrectamente tras la instrucción salto (puesto que éste es efectivo). Accediendo a los resultados finales, anota el número de instrucciones ejecutadas, el tiempo transcurrido y el CPI obtenido.

3. A continuación, manteniendo la resolución de los riesgos de control mediante *predict-not-taken*, modificaremos la configuración del simulador para que los riesgos de datos se resuelvan mediante cortocircuitos:

   ```
   mips-m -d c -c pnt3 -f apxpy.s
   ```

   El análisis de la ejecución ciclo a ciclo de la primera iteración del bucle nos permite observar cómo se aplican los cortocircuitos adecuados. Accediendo a los resultados finales, anota el número de instrucciones ejecutadas, el tiempo transcurrido y el CPI obtenido.

Realización de modificaciones en el código.

El objetivo de esta parte de la práctica es efectuar cambios en el código a ejecutar de tal manera que se reduzca en lo posible el número de ciclos de parada.

1. Seleccionando las estrategias *pnt3* y anticipación, copia el código a otro archivo (por ejemplo apxpy-p3.s) y modifícalo para reducir la penalización por riesgos de datos. Téngase en cuenta que si se aplican cortocircuitos, las únicas instrucciones que insertan ciclos de parada para resolver los riesgos de datos son las de carga. Ejecuta el programa mediante la orden:

   ```
   mips-m -d c -c pnt3 -f apxpy-p3.s
   ```

   Accediendo a los resultados finales, anota el número de instrucciones ejecutadas, el tiempo transcurrido y el CPI obtenido.

2. Manteniendo la anticipación para resolver los riesgos de datos, seleccionar ahora *pnt1* como estrategia de resolución de los riesgos de control. En este caso, las instrucciones de salto podrían tener que insertar ciclos de parada para resolver los riesgos de datos. Partiendo del código obtenido en el apartado 1, modifícalo nuevamente para reducir la penalización por riesgos de datos (por ejemplo en el archivo apxpy-p1.s).

   Ejecutar el programa mediante la orden:

   ```
   mips-m -d c -c pnt1 -f apxpy-p1.s
   ```

   Accediendo a los resultados finales, anota el número de instrucciones ejecutadas, el tiempo transcurrido y el CPI obtenido.

3. Manteniendo la anticipación para resolver los riesgos de datos, seleccionar ahora *ds1* como estrategia de resolución de los riesgos de control. Partiendo del código obtenido en el apartado 2, modifícalo nuevamente (por ejemplo en el archivo apxpy-d1.s) para que se ejecute adecuadamente, intentando rellenar el *delay-slot* con instrucciones útiles.

   Ejecutar el programa mediante la orden:

   ```
   mips-m -d c -c ds1 -f apxpy-d1.s
   ```

   Accediendo a los resultados finales, anota el número de instrucciones ejecutadas, el tiempo transcurrido y el CPI obtenido.

Desarrollo de un nuevo programa.

El código en alto nivel que se muestra seguidamente cuenta el número de componentes nulas de un vector:

```
...
num=0;
for (i=0; i<n-1; i++) {
  if (a[i]==0)
  num = num+1;
}
...
```

A continuación, se muestra el esqueleto del código ensamblador para resolver la tarea indicada, almacenado en el archivo `search.s`.

```
        .data
a:      .dword  9,8,0,1,0,5,3,1,2,0
tam:    .dword 10 ; Tamaño del vector
cont0:  .dword 0      ; Número de componentes == 0

        .text
start:
        ...

        trap #0
```

1. Completa el código suministrado. El número de componentes iguales a cero se debe almacenar en la variable `cont0` al finalizar el programa.

2. Analiza el programa y comprueba su correcto funcionamiento ejecutándolo en el simulador. Utilizaremos la configuración de anticipación y *pnt1*:

   ```
   mips-m -d c -c pnt1 -f search.s
   ```

   Analizar su tiempo de ejecución y CPI.

3. Identifica, en su caso, lo ciclos de parada y su causa. A continuación, modifica el código para reducir dicha penalización.

   Analizar su tiempo de ejecución y CPI.

Conjunto de instrucciones soportadas

- Carga/almacenamiento

ld Rx, desp(Ry)
sd Rz, desp(Ry)

- Aritméticas, lógicas y de desplazamiento

dadd Rx, Ry, Rz	daddi Rx, Ry, imm
dsub Rx, Ry, Rz	dsubi Rx, Ry, imm
and Rx, Ry, Rz	andi Rx, Ry, imm
or Rx, Ry, Rz	ori Rx, Ry, imm
xor Rx, Ry, Rz	xori Rx, Ry, imm
dsra Rx, Ry, Rz	dsra Rx, Ry, imm
dsll Rx, Ry, Rz	dsll Rx, Ry, imm
dsrl Rx, Ry, Rz	dsrl Rx, Ry, imm

- Comparación:

seq Rx, Ry, Rz	seq Rx, Ry, imm
sne Rx, Ry, Rz	sne Rx, Ry, imm
sgt Rx, Ry, Rz	sgt Rx, Ry, imm
slt Rx, Ry, Rz	slt Rx, Ry, imm
sge Rx, Ry, Rz	sge Rx, Ry, imm
sle Rx, Ry, Rz	sle Rx, Ry, imm

- Salto condicional

bnez Ry, desp	bne Rx,Ry, desp
beqz Ry, desp	beq Rx,Ry, desp

- Otras

nop
trap

Anexo A

Los errores más comunes son:

- El fichero se ha editado en Windows e incluye retornos de carro '\r' que se pueden eliminar con el comando: tr -d "\r"<fichero_original>fichero_sin_r
- Falta .data en el código.
- Alguna etiqueta está duplicada.
- Para sumar enteros del tipo *dword*, la instrucción correcta es *dadd* (por ejemplo *add* es incorrecta en este caso).
- Para restar enteros del tipo *dword*, la instrucción correcta es *dsub* (por ejemplo *sub* es incorrecta en este caso).