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

Arquitectura e Ingeniería de Computadores
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 RISC V.

Desarrollo:

El simulador del computador RISC V de 5 etapas.

El simulador es capaz de simular ciclo a ciclo la ejecución de instrucciones del RISC V, así como el avance de las mismas por la ruta de datos de la máquina. Soporta el conjunto rv64i de las instrucciones del RISC V. Tiene cache de instrucciones y de datos separadas (arquitectura *Harvard*). El banco de registros tiene dos puertos de lectura y uno de escritura Los riesgos de control pueden resolverse mediante la inserción de ciclos de parada (*stalls*) y *predict-not-taken* con tres valores de latencia de salto (uno, dos y tres ciclos). Los riesgos de datos pueden resolverse insertando ciclos de parada o aplicando la técnica de los cortocircuitos (*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:

```
./riscv-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.

    Si se añade la opción `-j` se genera un único fichero **index.htm** que incluye todos los resultados.

    - **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 seis 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* con latencia de salto de 3 ciclos (tres ciclos de penalización si el salto es efectivo).
    - **pnt2**: Se resuelven los riesgos de control mediante *predict-not-taken* con latencia de salto de 2 ciclos (dos ciclos de penalización si el salto es efectivo).
    - **pnt1**: Se resuelven los riesgos de control mediante *predict-not-taken* con latencia de salto de 1 ciclo (un ciclo de penalización si el salto es efectivo).
- *archivo.s*: es el nombre del archivo que contiene el código en ensamblador.

(*): opción por defecto.

Ejemplo de programa para RISC V.

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

y:  .dword 100,100,100,100,100,100,100,100,100,100
    .dword 100,100,100,100,100,100

# Vector z

z:  .space 128

# Escalar a
a:  .dword -10

# El codigo
    .text
```

```
start:
    addi t0,gp,x
    addi t3,gp,y
    addi t1,gp,y
    addi t2,gp,z
    ld t4,a(gp)

loop:
    ld a0,0(t0)
    add a0,t4,a0
    ld a1,0(t1)
    add a1,a0,a1
    sd a1,0(t2)
    addi t0,t0,8
    addi t1,t1,8
    addi t2,t2,8
    slt t5,t0,t3
    bnez t5,loop   # bne t5,zero,loop

    ori a7,zero,10   # Fin de programa
    ecall
```

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

   ```
   riscv-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 (pulsando sobre el enlace FINAL) 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.

⇒ Comprueba que se ha almacenado en la dirección definida por la etiqueta *z* el vector con el resultado del programa.

⇒ Considera el cronograma de la primera iteración del bucle y responde a las siguientes preguntas:

- La contribución de las instrucciones de la primera iteración al tiempo de ejecución comprende desde el ciclo __6__ al ciclo __26__. (16-6)+1
- El número de ciclos de reloj consumido por una iteración del bucle es de __21__ ciclos cuando el salto es efectivo. → 21 - nº instrucciones ⇒ 21 - 10 = 11
- Los ciclos de penalización son __11__ ciclos, de los cuales por riesgos de datos son __8__ ciclos y por riesgos de control son __3__ ciclos.
- El bucle ejecuta __10__ instrucciones.
- El CPI alcanzado es de: __2,1__. $\frac{T_{ej}}{Instrucciones} = 1 + \frac{CP}{instrucciones} = 2.1$

⇒ Considera el cronograma de la última iteración del bucle y responde a las siguientes preguntas:

- El número de ciclos de reloj consumido por una iteración del bucle es de __21__ ciclos cuando el salto no es efectivo.

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

```
riscv-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 buscan las instrucciones tras la instrucción salto y se abortan en la etapa MEM del salto, puesto que éste es efectivo.

⇒ Considera el cronograma de la primera iteración del bucle y responde a las siguientes preguntas:

- La contribución de las instrucciones de la primera iteración al tiempo de ejecución comprende desde el ciclo __6__ al ciclo __26__.
- El número de ciclos de reloj consumido por una iteración del bucle es de __21__ ciclos cuando el salto es efectivo.
- Los ciclos de penalización son __11__ ciclos, de los cuales por riesgos de datos son __8__ ciclos y por riesgos de control son __3__ ciclos.
- El bucle ejecuta __10__ instrucciones.
- El CPI alcanzado es de: __2.1__.

⇒ Considera el cronograma de la última iteración del bucle y responde a las siguientes preguntas:

- El número de ciclos de reloj consumido por una iteración del bucle es de 18 ciclos cuando el salto no es efectivo.

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:

```
riscv-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.

⇒ Considera el cronograma de la primera iteración del bucle y responde a las siguientes preguntas:

- La contribución de las instrucciones de la primera iteración al tiempo de ejecución comprende desde el ciclo 6 al ciclo 20.
- El número de ciclos de reloj consumido por una iteración del bucle es de 15 ciclos cuando el salto es efectivo. 15 - 10 = 5
- Los ciclos de penalización son 5 ciclos, de los cuales por riesgos de datos son 2 ciclos y por riesgos de control son 3 ciclos.
- El bucle ejecuta 10 instrucciones.
- El CPI alcanzado es de: 1,5. $1 + \frac{CP}{Inst.} = \frac{5}{10} + 1 = 1,5$

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 cortocircuitos, copia el código a otro archivo (por ejemplo apxpy-p3.s) y modifícalo para reducir la penalización por riesgos de datos. Ten 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:

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

⇒ Accediendo a los resultados finales de la ejecución, comprueba que el resultado es el correcto. En particular, verifica que el el vector $\vec{Z}$ contiene los valores esperados.

⇒ Considera el cronograma de la primera iteración del bucle y responde a las siguientes preguntas:

- La contribución de las instrucciones de la primera iteración al tiempo de ejecución comprende desde el ciclo 6 al ciclo 18.
- El número de ciclos de reloj consumido por una iteración del bucle es de 13 ciclos cuando el salto es efectivo.

- Los ciclos de penalización son 3 ciclos, de los cuales por riesgos de datos son 0 ciclos y por riesgos de control son 3 ciclos.
- El bucle ejecuta 10 instrucciones.
- El CPI alcanzado es de: 1,3. $1 + \frac{3}{10} = 1,3$

2. Manteniendo los cortocircuitos para resolver los riesgos de datos, selecciona 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).

   Ejecuta el programa mediante la orden:

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

   ⇒ Accediendo a los resultados finales, comprueba que el resultado es el correcto.

   ⇒ Considera el cronograma de la primera iteración del bucle y responde a las siguientes preguntas:

   - La contribución de las instrucciones de la primera iteración al tiempo de ejecución comprende desde el ciclo 6 al ciclo 16.
   - El número de ciclos de reloj consumido por una iteración del bucle es de 11 ciclos cuando el salto es efectivo.
   - Los ciclos de penalización son 1 ciclos, de los cuales por riesgos de datos son 0 ciclos y por riesgos de control son 1 ciclos.
   - El bucle ejecuta 10 instrucciones.
   - El CPI alcanzado es de: 1,11.

Desarrollo de un nuevo programa.

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

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

A continuación, se muestra el esqueleto del código ensamblador para resolver la tarea indicada, almacenado en el archivo `search.s`. El vector está almacenado a partir de la posición de memoria `a`, su tamaño está definido en la posición de memoria `tam` y el número de componentes nulas del vector se almacenará en la posición de memoria `cont`.

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

        .text
start:  addi t0,gp,a       ; Puntero
        ld t1,tam(gp)      ; Tamaño lista
        add t2,zero,zero   ; Contador de ceros

loop:
        ...

        ori x17,x0,10      ; Fin de programa
        ecall
```

⇒ Se pide:

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

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

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

   Analiza su tiempo de ejecución y CPI. → 1,49

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

   Analiza su tiempo de ejecución y CPI.

→ CP

3