PRÁCTICA 3:
"UNIDAD DE INSTRUCCIÓN SEGMENTADA (II)"

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

Objetivos:

- Implementar la lógica para resolver los riegos de datos y de control en un procesador segmentado.

Desarrollo:

Para el desarrollo de la práctica se partirá de un simulador del MIPS. El simulador interpreta código ensamblador del MIPS, incluyendo tanto instrucciones enteras como de coma flotante. En esta práctica, trabajaremos con programas que utilizan únicamente instrucciones enteras. Las instrucciones enteras se simulan en una unidad de instrucción segmentada en 5 etapas (IF, ID, EX, MEM y WB). Sin embargo, el código que resuelve los riesgos de datos y de control no está incluido, y su realización constituye el objetivo de la práctica.

El presente boletín se organiza como sigue: descripción de los ficheros que componen el simulador, las instrucciones implementadas, estructuras de datos utilizadas, estructura del simulador del MIPS segmentado y, finalmente, los ejercicios a realizar.

Estructura del simulador

El simulador del MIPS está escrito en el lenguaje de programación C. Se compone de diversos módulos, entre los que resaltaremos los siguientes:

main.c Programa principal del simulador. Encargado de leer el ensamblador y ejecutar las distintas fases de la unidad de instrucción segmentada.

main.h Contiene todas las variables compartidas del simulador: memoria de instrucciones y datos, registros de uso general, registros inter-etapa, señales de control, etc.

tipos.h Contiene las definiciones de todas las estructuras de datos utilizadas en el simulador: memoria de instrucciones y datos, banco de registros, registros inter-etapa, formatos de instrucción, etc.

instrucciones.h Contiene los códigos de operación de las instrucciones implementadas y algunas macros de utilidad.

mips.c Contiene la implementación de las fases de la unidad de instrucción. *Este fichero lo modificaremos en esta práctica.*

mips_int.c Contiene la implementación de la unidad aritmética de enteros, la lógica de detección de riesgos de datos y la lógica para aplicar cortocircuitos entre instrucciones enteras. *Este fichero lo modificaremos en esta práctica.*

Instrucciones implementadas

El simulador soporta todas las instrucciones enteras del MIPS, tanto en versión registro–registro como registro–inmediato, con la excepción de la instrucción de salto incondicional. En particular, es capaz de ejecutar el siguiente repertorio de instrucciones:

- 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	bc1t desp
beqz Ry, desp	bc1f desp

- Otras

nop
trap

Estructuras de datos

A continuación, se describirán las estructuras de datos utilizadas (que se encuentran en el fichero `tipos.h`) y su utilización.

Tipos básicos

Los tipos básicos utilizados son:

```
typedef unsigned char  byte;  /* Un byte: 8 bits */
typedef short          half;  /* Media palabra: 16 bits */
typedef int32_t        word;  /* Una palabra: 32 bits */
typedef int64_t        dword; /* Una doble palabra: 64 bits */
typedef enum {NO=0, SI=1} boolean; /* Valor lógico */
```

Formatos de instrucción

Los formatos de instrucción se representan mediante un tipo enumerado:

```
/* Formatos de instruccion */
typedef enum {FormatoR, FormatoI, FormatoJ} formato_t;
```

Las instrucciones se representan mediante la siguiente estructura de datos:

```
typedef struct {
  codop_t       codop;          /* Código de operación */
  formato_t     tipo;           /* Formato */
  byte          Rfuente1,       /* Registro fuente 1 */
                Rfuente2;       /* Registro fuente 2 */
  byte          Rdestino;       /* Registro destino */
  half          inmediato;      /* Valor Inmediato */
} instruccion_t;
```

Banco de registros

El banco de registros es un vector compuesto por elementos del tipo `reg_int_t`, con un único campo, valor.

```
typedef struct {
  dword          valor;         /* Valor del registro */
} reg_int_t;
```

Registros inter-etapa

Los registros inter-etapa se representan mediante una estructura que contiene cada uno de los campos necesarios:

- Registro IF/ID:

```
typedef struct {
  instruccion_t IR;          /* IR */
  dword         NPC;         /* PC+4 */
} IF_ID_t;
```

- Registro ID/EX:

```
typedef struct {
  instruccion_t IR;          /* IR */
  dword         NPC;         /* PC+4 */
  dword         Ra,          /* Valores de los registros*/
                Rb;
  dword         Imm;         /* Inmediato con signo extendido */
} ID_EX_t;
```

- Registro EX/MEM:

```
typedef struct {
  instruccion_t IR;          /* IR */
  dword         ALUout;      /* Resultado ALU */
  dword         data;        /* Dato a escribir */
  boolean       cond;        /* Resultado condicion de salto */
} EX_MEM_t;
```

- Registro MEM/WB:

```
typedef struct {
  instruccion_t IR;          /* IR */
  dword         ALUout;      /* Resultado ALU */
  dword         MEMout;      /* Resultado Memoria */
} MEM_WB_t;
```

Otras estructuras de datos

- Define cómo se resuelven los riesgos de datos:

```
typedef enum {
  parada,         /* Inserta ciclos de parada */
  cortocircuito, /* Aplica cortocircuito+ciclos de parada */
  ninguno
} riesgos_d_t;
```

- Define cómo se resuelven los riesgos de control:

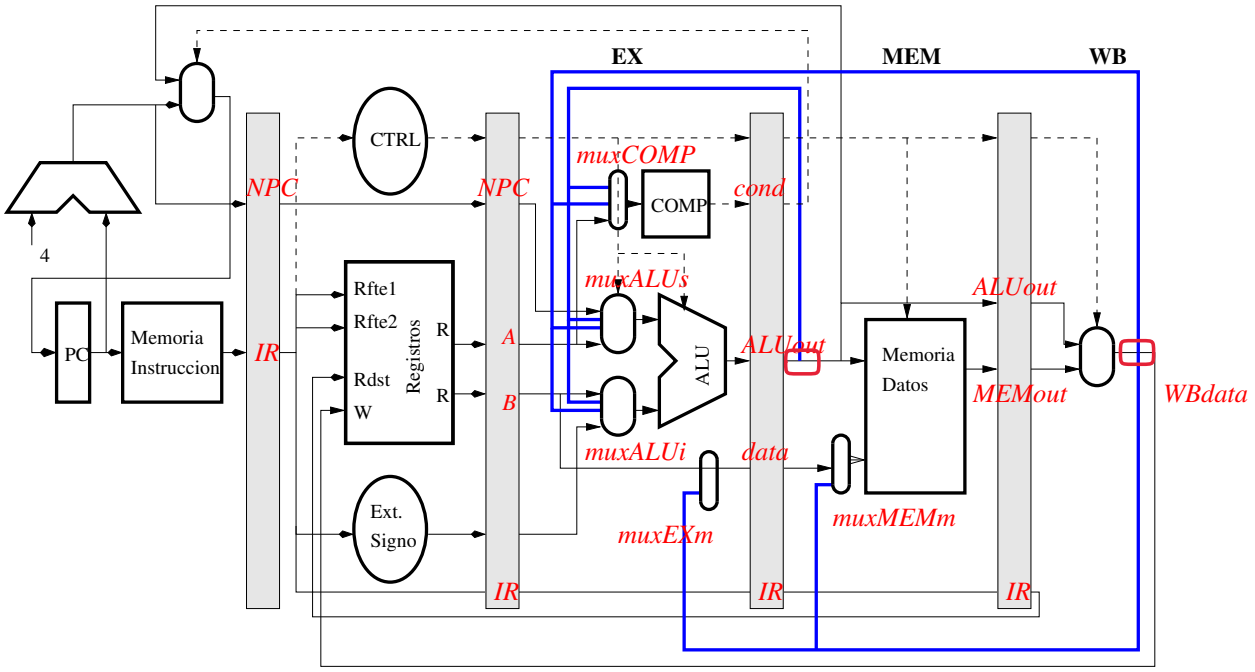
```
typedef enum {
  stall3,    /* Inserta 3 ciclos de parada */
  stall2,    /* Inserta 3 ciclos de parada */
  stall1,    /* Inserta 3 ciclos de parada */
  pnt3,      /* Predict-not-taken, 3 ciclos */
  pnt2,      /* Predict-not-taken, 2 ciclos */
  pnt1,      /* Predict-not-taken, 1 ciclos */
  ds3,       /* Salto retardado, con ds=3 */
  ds2,       /* Salto retardado, con ds=2 */
  ds1,       /* Salto retardado, con ds=1 */
} riesgos_c_t;
```

Estructura de unidad de instrucción segmentada

La unidad de instrucción segmentada está compuesta por los siguientes elementos (ver figura):

Memoria de instrucciones. Almacena el programa a ejecutar. Es direccionable al byte[1]. Está representada por la variable `MI` (`main.h`), del tipo `memoria_instruciones_t` (`tipos.h`). El tamaño de la memoria viene indicado por la constante `TAM_MEMO_INSTRUC` (`tipos.h`).

[1]bytes sucesivos tienen asignadas direcciones consecutivas y palabras sucesivas tienen asignadas direcciones que difieren en 4



Banco de Registros Enteros. Contiene los registros enteros. Está representada por la variable `Rint` (`main.h`), del tipo `reg_int_t []` (`tipos.h`). El número de registros viene indicado por la constante `TAM_REGISTROS` (`main.h`).

Operador aritmético. Realiza las operaciones aritméticas correspondientes a la fase EX. Está representada por la función `operacion_ALU (codop,in1,in2)` (`mips_int.c`), donde `codop`, `in1` e `in2` representan la instrucción aritmética a ejecutar y los dos datos a operar, respectivamente. La función devuelve el resultado de la operación correspondiente.

Evaluación de la condición de salto. Calcula la condición de salto. Está representada por la función `operacion_COMP (codop,in1,in2)` (`mips_int.c`), donde `codop`, `in1` e `in2` representan la instrucción de salto a ejecutar y los operandos a evaluar, respectivamente. La función devuelve si se debe o no saltar.

Multiplexor de la entrada superior del operador aritmético. Está representado por la función `mux_ALUsup (npc,ra,mem,wb)` (`mips_int.c`), donde `npc`, `ra`, `mem` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Multiplexor de la entrada inferior del operador aritmético. Está representado por la función `mux_ALUinf (rb,imm,mem,wb)` (`mips_int.c`), donde `rb`, `imm`, `mem` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Multiplexor de la entrada del comparador (saltos). Está representado por la función `mux_COMP (ra,mem,wb)` (`mips_int.c`), donde `ra`, `mem` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Multiplexor de los datos a escribir en memoria (etapa EX). Está representado por la función `mux_EXmem (rb,wb)` (`mips_int.c`), donde `rb` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Memoria de datos. Almacena los datos del programa a ejecutar. Es direccionable al byte. Está representada por la variable `MD` (`main.h`), del tipo `memoria_datos_t` (`tipos.h`). El tamaño de la memoria viene indicado por la constante `TAM_MEMO_DATOS` (`tipos.h`).

Multiplexor de los datos a escribir en memoria (etapa MEM). Está representado por la función `mux_MEMmem (rb,wb)` (`mips_int.c`), donde `rb` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Salida del multiplexor de la fase WB. Representa el dato que se va a escribir en el banco de registros en la fase WB. Está representado por la variable `WBdata` (`main.h`), de tipo `dword`.

Registros inter–etapa. Su nombre viene de las etapas que interconectan. Son los siguientes:

- `IF_ID`, del tipo `IF_ID_t` (`tipos.h`).
- `ID_EX`, del tipo `ID_EX_t` (`tipos.h`).
- `EX_MEM`, del tipo `EX_MEM_t` (`tipos.h`).
- `MEM_WB`, del tipo `MEM_WB_t` (`tipos.h`).

Por ejemplo, si queremos conocer el identificador del registro fuente1 de la instrucción que está en la etapa ID, utilizaremos `IF_ID.IR.Rfuente1`.

Valor actual y nuevo valor del PC. Lo representa las variables `PC` y `PCn`, del tipo `dword`. La siguiente instrucción se buscará en la dirección indicada en `PCn`.

Señales de control El simulador dispone de las siguientes señales de control, todas ellas del tipo `boolean`:

- `IFstall`: Al activarla (`IFstall=SI`) mantiene la instrucción en la fase IF al siguiente ciclo de reloj, entregando además una instrucción `nop` a la etapa ID.
- `IDstall`: Al activarla mantiene la instrucción en la fase ID al siguiente ciclo de reloj, entregando además una instrucción `nop` a la etapa EX.
- `IFnop`: Al activarla pasará una instrucción `nop` a la fase ID al siguiente ciclo de reloj.
- `IDnop`: Al activarla pasará una instrucción `nop` a la fase EX al siguiente ciclo de reloj.
- `EXnop`: Al activarla pasará una instrucción `nop` a la fase MEM al siguiente ciclo de reloj.

Comprobación del tipo de instrucción Las siguientes macros o funciones permiten analizar determinados campos de una instrucción:

- `es_load(inst).` Indica que la instrucción inst es de carga.
- `lee_Rfte1(inst).` Indica que la instrucción inst utiliza un registro fuente 1 válido.
- `lee_Rfte2(inst).` Indica que la instrucción inst utiliza un registro fuente 2 válido.
- `escribe_Rdst(inst).` Indica que la instrucción inst utiliza un registro destino válido.

A modo de ejemplo, se muestra el contenido de una de ellas:

```
boolean lee_Rfte1(instruccion_t inst) {
    // Hay Rfte1 si no es R0, no es NOP, no es TRAP
    return (
            (inst.Rfuente1 != 0) &&
            (inst.codop != OP_NOP) && (inst.codop != OP_TRAP)
           );
```

Por ejemplo, si queremos saber si la instrucción que está en la etapa ID necesita (va a leer) el campo registro fuente1, utilizaremos la llamada `lee_Rfte1(IF_ID.IR)`. De igual manera, si queremos saber si la instrucción que está en la etapa EX utiliza (va a escribir) el campo registro destino, utilizaremos la llamada `escribe_Rdst(ID_EX.IR)`. Por otra parte, si queremos saber si la instrucción que está en la etapa ID es de carga, utilizaremos la llamada `es_load(IF_ID.IR)`.

Pseudo-código del simulador del MIPS

El programa principal del simulador, tras inicializar las estructuras de datos, carga el fichero que contiene el programa a ejecutar, lo ensambla y ejecuta el bucle principal del simulador, cuyo pseudo-código se muestra seguidamente:

```
/* Bucle principal del simulador MIPS */

  /*** Fase: WB **************/
  fase_escritura ():
    -escribir registro

  /*** Fase: MEM **************/
  fase_memoria ():
    -detectar riesgos de control
    -aplicar cortocircuitos
    -acceso a memoria, en su caso

  /*** Fase: EX *************/
  fase_ejecucion ():
    -detectar riesgos de control
    -aplicar cortociruitos
    -operacion en la ALU/COMP
```

```
/*** Fase: ID **************/
fase_decodificacion():
  -detectar riesgos de datos
  -detectar riesgos de control
  -leer registros

/*** Fase: IF ***********/
fase_busqueda();
  -buscar instrucción
  -actualizar PC

Ciclo++;
imprimir_estado;
impulso_reloj();
```

Ejercicios a realizar

En primer lugar, probaremos el simulador con un sencillo fragmento de código que no tenga riesgos de datos, el cual está almacenado en el fichero `ejemplo.s`

```
        ; suma las componentes del vector hasta que encuentra
        ; una componente a 0
        ; almacena el resultado en a
        .data
a:      .dword  0
y:      .dword  1,2,3,0,4,5,6,7,8
        .text
        dadd r1,r0,r0 ; r1=0
        dadd r3,r0,r0 ; r3=0
        dadd r2,r0,y  ; r2 recorre y
        nop

bucle: dadd r1,r3,r1
        ld r3,0(r2)   ; r3 es y[i]
        dadd r2,r2,#8
        nop
        nop
        bnez r3,bucle ; si r3<>0
        sd r1,a(r0)
final:
        ; el resultado debe ser a=6
        trap #0
```

Para invocar la ejecución del simulador se utilizará la orden `mips-m`. El simulador acepta varios parámetros:

```
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-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-control*: indica cómo se resuelven los riesgos de control. Hay nueve 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.

En este caso, lo ejecutaremos sín lógica de detección de riesgos de datos y resolviendo los riesgos de control insertando (3) ciclos de parada:

```
mips-m -d n -c s3 -f ejemplo.s
```

Esta orden generará un fichero en formato **html** por cada ciclo con la información sobre el estado de la máquina, más los ficheros inicial `index.html` y de resultados `final.html`. Estos ficheros pueden visualizarse mediante un navegador. Por defecto, el simulador borra los archivos html al iniciar una nueva simulación, salvo que se le pase el parámetro "-n".

Comprobar el correcto funcionamiento del simulador, y que el resultado de la ejecución es el esperado.

Sin embargo, el simulador suministrado sólo incluye el código necesario para resolver los **riesgos de control insertando 3 ciclos de parada** o mediante la técnica del **salto retardado con *delay slot=3***. Sin embargo, **no es capaz de detectar ni de resolver los riesgos de datos**.

La labor a realizar en esta práctica consiste en añadir al simulador nuevas estrategias para detectar y resolver los riesgos. Para ello, deberán modificarse los ficheros `mips.c` y `mips_int.c`, añadiendo el código necesario para realizar las acciones indicadas.

Para la edición de los ficheros se puede utilizar cualquiera de los editores disponibles: `vi`, `emacs`, `[gk]edit` o `kate`.

Para la compilación del simulador `mips-m` se debe ejecutar la orden `make` en el directorio donde se encuentren los fuentes del simulador.

Para comprobar el correcto funcionamiento de las modificaciones realizadas, se utilizaran los ficheros de prueba suministrados, tal y como se explica a continuación.

1. Modificar el simulador del MIPS para detectar y resolver los riesgos de datos **insertando ciclos de parada**. En particular, pretendemos resolver el riesgo de datos provocado por la siguiente secuencia de instrucciones (almacenado en el fichero `datos1.s`):

   ```
   .ireg 10,20,0,0,0 ; r1=10, r2=20
   .text
   dadd r3,r1,r2
   dsub r4,r3,#5
   dadd r5,r3,#5
   ; el resultado debe ser r3=30, r4=25 y r5=35
   ```

   Para ello, se debe modificar la función que realiza la detección de riesgos de datos en la fase de decodificación de las instrucciones (función `detectar_riesgos_datos` en el archivo `mips_int.c`), escribiendo el código que activa las señales de control necesarias (`IFstall`, `IDstall`).

   Comprobar el correcto funcionamiento del simulador modificado ejecutando:

   ```
   mips-m -d p -c s3 -f datos1.s
   ```

2. Modificar el simulador MIPS para detectar y resolver los riesgos de datos **aplicando cortocircuitos**.

- En primer lugar, pretendemos resolver el riesgo de datos provocado por la misma secuencia de código del apartado anterior (fichero `datos1.s`), la cual no requiere insertar ciclos de parada.

  Para ello, se debe modificar las funciones que implementan los multiplexores superior (operando fuente1) e inferior (operando fuente2) ubicados a la entrada del operador aritmético lógico. Como en el código de prueba el riesgo de datos únicamente afecta únicamente al operando fuente1 de las instrucciones, solo es necesario modificar la función `mux_ALUsup` en el fichero `mips_int.c`.

  Comprobar el correcto funcionamiento del simulador modificado ejecutando:

  ```
  mips-m -d c -c s3 -f datos1.s
  ```

- Ahora pretendemos resolver el riesgo de datos provocado por una **instrucción de carga seguida por una instrucción aritmética**. El código de prueba está almacenado en el fichero `datos2.s`:

  ```
          .ireg 10,20,0,0,0
          .data
  a:      .dword 10
          .text
          ld r3,a(r0)
          dsub r4,r3,#5
          dadd r5,r3,#5
          ; el resultado debe ser r3=10, r4=5 y r5=15
  ```

  En este caso, además de activar el cortocircuito correspondiente, se debe insertar un ciclo de parada en la fase de decodificación. Por lo tanto, además de la modificación realizada en la función que implementa el multiplexor superior asociado a la entrada del operador aritmético lógico (`mux_ALUsup` en `mips_int.c`), debe modificarse la función que realiza la detección de riesgos en la fase de decodificación de las instrucciones (función `detectar_riesgos_datos` en el fichero `mips_int.c`).

  Comprobar el correcto funcionamiento del simulador modificado ejecutando:

  ```
  mips-m -d c -c s3 -f datos2.s
  ```

3. Modificar el simulador MIPS para resolver los riesgos de control mediante la estrategia *predict-not-taken*.

   Para ello, se debe modificar la función que realiza la fase de búsqueda de las instrucciones (función `fase_busqueda` en el fichero `mips.c`).

   Para probar esta modificación, puede emplearse el código siguiente, almacenado en el fichero `suma.s`:

   ```
           ; suma las componentes del vector hasta que encuentra
           ; una componente a 0
           ; almacena el resultado en a
           .data
   a:      .dword  0
   ```

```
y:      .dword  1,2,3,0,4,5,6,7,8
        .text
        dadd r1,r0,r0 ; r1=0
        dadd r3,r0,r0 ; r3=0
        dadd r2,r0,y  ; r2 recorre y
bucle:  dadd r1,r3,r1
        ld r3,0(r2)   ; r3 es y[i]
        dadd r2,r2,#8
        bnez r3,bucle ; si r3<>0
        sd r1,a(r0)
final:
        ; el resultado debe ser a=6
```

Comprobar el correcto funcionamiento del simulador modificado, resolviendo los riesgos de datos con ciclos de parada:

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

Observa que ahora se cancelan las instrucciones sólo cuando el salto es efectivo.