PRÁCTICA 3ª
"PLANIFICACIÓN ESTÁTICA DE INSTRUCCIONES"

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

Objetivos:

- Conocer, comprender y aplicar algunas técnicas de gestión estática de instrucciones.

Desarrollo:

El simulador del procesador MIPS con instrucciones multiciclo

El simulador **mips-m** permite ejecutar programas escritos en lenguaje ensamblador del MIPS. Soporta un subconjunto del juego de instrucciones, incluyendo instrucciones enteras y de coma flotante. En el anexo se detallan las instrucciones soportadas.

El procesador simulado no incorpora planificación dinámica de instrucciones. Para resolver los riesgos de datos inserta ciclos de parada o bien aplica la técnica de la anticipación o cortocircuito, con inserción de ciclos de parada en caso necesario. Los riesgos de control se pueden resolver insertando ciclos de parada, con *predict-not-taken* o bien mediante salto retardado, con *delay-slot* de una, dos o tres instrucciones. Para la ejecución de operaciones multiciclo, dispone de una unidad de carga/almacenamiento, un operador de multiplicación, un operador de suma y un operador de comparación. Todos ellos están segmentados. Se puede configurar la latencia de cada uno de los operadores.

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:

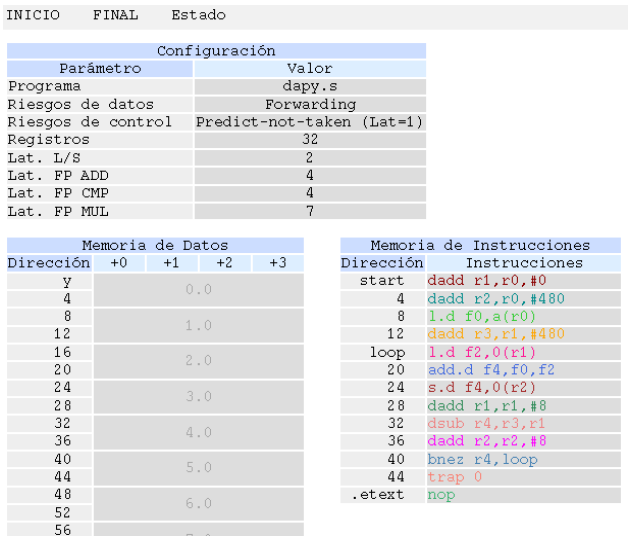
- *resultados*: indica cómo se ofrecerá el resultado de la simulación. Hay varias opciones:
  - **tiempo**: Muestra el tiempo de ejecución en el terminal.
  - **final**: Muestra el tiempo de ejecución, los registros y el contenido de la memoria tras la ejecución en el terminal.
  - **html**(*): Genera 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.

Ejemplo de programa para MIPS

A continuación, se muestra el código ensamblador correspondiente a un bucle que realiza la suma de un valor escalar a un vector almacenado en la memoria ($\vec{Z} = a + \vec{Y}$, bucle DAPY):

```
start:
        dadd r1,r0,y     ; r1 contiene la direccion de y
        dadd r2,r0,z     ; r2 contiene la direccion de z
        l.d f0,a(r0)     ; f0 contiene a
        dadd r3,r1,#480  ; 60 elementos son 480 bytes
loop:
        l.d f2,0(r1)
        add.d f4,f0,f2
        s.d f4, 0(r2)
        dadd r1,r1,#8
        dsub r4,r3,r1
```


INICIO FINAL Estado

Configuración

Parámetro	Valor
Programa	dapy.s
Riesgos de datos	Forwarding
Riesgos de control	Predict-not-taken (Lat=1)
Registros	32
Lat. L/S	2
Lat. FP ADD	4
Lat. FP CMP	4
Lat. FP MUL	7

Memoria de Datos

Dirección	+0	+1	+2	+3
y	0.0			
4				
8	1.0			
12				
16	2.0			
20				
24	3.0			
28				
32	4.0			
36				
40	5.0			
44				
48	6.0			
52				
56				

Memoria de Instrucciones

Dirección	Instrucciones
start	dadd r1,r0,#0
4	dadd r2,r0,#480
8	l.d f0,a(r0)
12	dadd r3,r1,#480
loop	l.d f2,0(r1)
20	add.d f4,f0,f2
24	s.d f4,0(r2)
28	dadd r1,r1,#8
32	dsub r4,r3,r1
36	dadd r2,r2,#8
40	bnez r4,loop
44	trap 0
.etext	nop

Figura 1: Contenido del archivo *index.html*

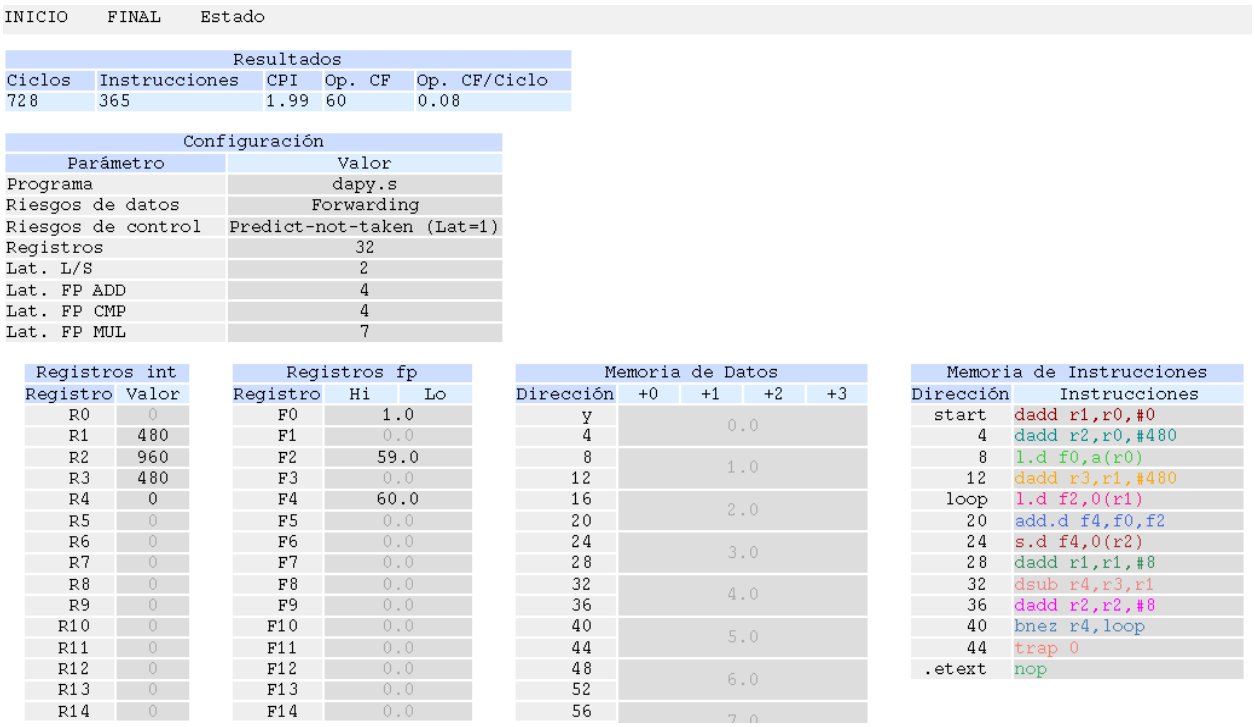
```
        dadd r2,r2,#8
        bnez r4,loop

        trap #0          ; Fin de programa
```

Este programa está almacenado en el fichero `dapy.s`. Lo lanzaríamos a ejecución mostrando resultados con archivos html y resolviendo riesgos de datos y de control mediante cortocircuito y *predict-not-taken* con la orden siguiente:

```
mips-m -s html -d c -c pnt1 -f dapy.s
```

Seguidamente, abriremos el archivo index.html mediante el navegador, el cual muestra la configuración del procesador y el contenido de la memoria inicialmente, 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. Comprobar el contenido final de la memoria permite verificar que el programa funciona correctamente.
- 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.



INICIO FINAL Estado

Resultados

Ciclos	Instrucciones	CPI	Op. CF	Op. CF/Ciclo
728	365	1.99	60	0.08

Configuración

Parámetro	Valor
Programa	dapy.s
Riesgos de datos	Forwarding
Riesgos de control	Predict-not-taken (Lat=1)
Registros	32
Lat. L/S	2
Lat. FP ADD	4
Lat. FP CMP	4
Lat. FP MUL	7

Registros int

Registro	Valor
R0	0
R1	480
R2	960
R3	480
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
R13	0
R14	0

Registros fp

Registro	Hi	Lo
F0	1.0	
F1	0.0	
F2	59.0	
F3	0.0	
F4	60.0	
F5	0.0	
F6	0.0	
F7	0.0	
F8	0.0	
F9	0.0	
F10	0.0	
F11	0.0	
F12	0.0	
F13	0.0	
F14	0.0	

Memoria de Datos

Dirección	+0	+1	+2	+3
y	0.0			
4				
8	1.0			
12				
16	2.0			
20				
24	3.0			
28				
32	4.0			
36				
40	5.0			
44				
48	6.0			
52				
56	7.0			

Memoria de Instrucciones

Dirección	Instrucciones
start	dadd r1,r0,#0
4	dadd r2,r0,#480
8	l.d f0,a(r0)
12	dadd r3,r1,#480
loop	l.d f2,0(r1)
20	add.d f4,f0,f2
24	s.d f4,0(r2)
28	dadd r1,r1,#8
32	dsub r4,r3,r1
36	dadd r2,r2,#8
40	bnez r4,loop
44	trap 0
.etext	nop

Figura 2: Contenido del archivo *final.html*

La Figura 1 muestra el contenido del archivo *index.html* generado. Se muestran los parámetros de configuración de procesador y el contenido inicial de la zona de datos y de instrucciones de la memoria, respectivamente.

Si seguimos el enlace FINAL se abrirá el archivo *final.html*. La Figura 2 muestra su contenido para nuestro ejemplo. En primer lugar, se muestran los resultados de prestaciones de la ejecución: tiempo de ejecución, instrucciones ejecutadas, CPI, operaciones en coma flotante y operaciones en coma flotante por ciclo. Seguidamente, se recuerda la configuración del procesador y se muestra el contenido final de los registros y la zona de datos y de instrucciones de la memoria

Si seguimos el enlace Estado se abrirá el archivo *resultXXX.html*, donde XXX representa el ciclo de ejecución, comenzando en "001". La figura 3 muestra su contenido para el ciclo 21 de nuestro ejemplo. En primer lugar, se muestran los enlaces a las páginas *index.html* y *final.html*, así como enlaces a los archivos con el estado existente hace 10 ([-10]), ([-5]) 5 ciclos , el ciclo anterior ([-1]), al ciclo siguiente ([+1]), dentro de 5 ciclos ([+5]) y dentro de 10 ciclos ([+5]). También se muestra el diagrama instrucciones–tiempo hasta el ciclo actual (Crono). A continuación se muestran las etapas de la unidad de ejecución, indicando qué instrucción ocupa cada una de ellas. Las etapas vacías contienen el equivalente a una instrucción que no hace nada (-nop-). Como hay banco de registros entero y de coma flotante separados, en un ciclo dado puede haber hasta una instrucción entera y otra de coma flotante en la etapa WB. También se muestran las señales de control que se activarían como consecuencia de la detección de riesgos y aplicación de cortocircuitos. A continuación se muestra el contenido de los registros enteros (R0 a R31) y de coma flotante (F0 a F31), respectivamente, más el registro de estado de coma flotante (FPSR). Finalmente, se muestra el contenido de la memoria de datos. Los archivos con el estado del procesador permiten ejecutar paso a paso el programa, permitiendo, de esta

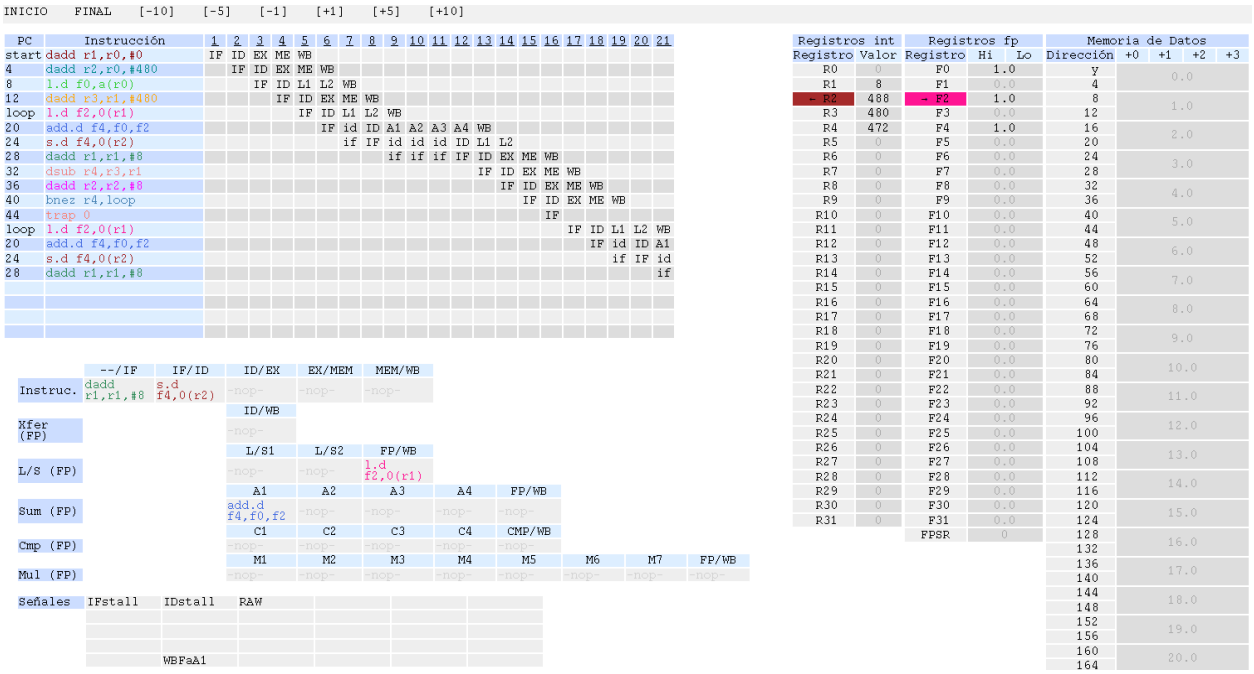


Figura 3: Contenido del archivo *result021.html*

forma, depurar el código cuando escribamos programas en ensamblador MIPS.

Tras ejecutar el programa, comprobar que se ha almacenado en la dirección definida por la etiqueta `z` un vector de 60 datos, con el contenido esperado. Anotar el tiempo de ejecución del programa y los CPI obtenidos.

Modificación del programa aplicando planificación estática de instrucciones

1. *Loop unrolling*

   Básicamente, el *loop unrolling* replica el código base del bucle varias veces, disminuyendo el número de iteraciones realizadas.

   En nuestro ejemplo, como el máximo número de ciclos de parada necesario para resolver el riego RAW producido es de tres ciclos, el código del bucle $\vec{Z} = a + \vec{Y}$ debe ser replicado 4 veces, y se muestra seguidamente. Nótese que se han renombrado algunos registros para eliminar dependencias de nombre:

```
start:
        dadd r1,r0,y     ; r1 contiene la direccion de y
        dadd r2,r0,z     ; r2 contiene la direccion de z
        l.d f0,a(r0)     ; f0 contiene a
        dadd r3,r1,#480  ; 60 elementos son 480 bytes
loop:
        l.d f2,0(r1)
        add.d f4,f0,f2
        s.d f4,0(r2)
        l.d f6,8(r1)
        add.d f8,f0,f6
        s.d f8,8(r2)
        l.d f10,16(r1)
        add.d f12,f0,f10
        s.d f12,16(r2)
        l.d f14,24(r1)
        add.d f16,f0,f14
        s.d f16,24(r2)
        dadd r1,r1,#32
        dsub r4,r3,r1
        dadd r2,r2,#32
        bnez r4,loop

        trap #0          ; Fin de programa
```

   Este programa está almacenado en el fichero `dapyu.s`. Ejecutamos este nuevo programa:

```
mips-m -s html -d c -c pnt1 -f dapyu.s
```

   Comprueba la corrección del resultado obtenido y anotando su tiempo de ejecución. Calcular los CPI obtenidos. Cuantifica la mejora frente al programa original.

   Ese código puede modificarse fácilmente para eliminar todos los riesgos de datos:

```
start:
        dadd r1,r0,y     ; r1 contiene la direccion de y
```

```
        dadd r2,r0,z     ; r2 contiene la direccion de z
        l.d f0,a(r0)     ; f0 contiene a
        dadd r3,r1,#480  ; 60 elementos son 480 bytes
loop:
        l.d f2,0(r1)
        l.d f6,8(r1)
        l.d f10,16(r1)
        l.d f14,24(r1)
        add.d f4,f0,f2
        add.d f8,f0,f6
        add.d f12,f0,f10
        add.d f16,f0,f14
        s.d f4,0(r2)
        s.d f8,8(r2)
        s.d f12,16(r2)
        s.d f16,24(r2)
        dadd r1,r1,#32
        dsub r4,r3,r1
        dadd r2,r2,#32
        bnez r4,loop

        trap #0         ; Fin de programa
```

Este programa está almacenado en el fichero `dapyuo.s`. Lo ejecutamos:

```
mips-m -s html -d c -c pnt1 -f dapyuo.s
```

Comprueba la corrección del resultado obtenido y anotando su tiempo de ejecución. Calcular los CPI obtenidos. Cuantifica la mejora frente al programa original.

2. *Software pipelining.*

   Básicamente, el *software pipelining* sustituye el bucle original por otro nuevo en el que las instrucciones que se ejecutan pertenecen a iteraciones distintas del bucle original, con lo que se eliminan los riesgos de datos.

   El código del bucle $\vec{Z} = a + \vec{Y}$ modificado es el siguiente:

```
start:
        dadd r1,r0,y     ; r1 contiene la direccion de y
        dadd r2,r0,z     ; r2 contiene la direccion de z
        l.d f0,a(r0)     ; f0 contiene a
        dadd r3,r1,#480  ; 60 elementos son 480 bytes
prepara:
        l.d f2,0(r1)
        add.d f4,f0,f2
        l.d f2,8(r1)
        dadd r1,r1,#16

loop:
```

```
        s.d f4, 0(r2)
        add.d f4,f0,f2
        l.d f2,0(r1)
        dadd r1,r1,#8
        dsub r4,r3,r1
        dadd r2,r2,#8
        bnez r4,loop

resto:
        s.d f4, 0(r2)
        add.d f4,f0,f2
        s.d f4, 8(r2)

        trap #0         ; Fin de programa
```

Este programa está almacenado en el fichero `dapysp.s`. Lo ejecutamos:

```
mips-m -s html -d c -c pnt1 -f dapysp.s
```

Comprueba la corrección del resultado obtenido y anotando su tiempo de ejecución. Calcular los CPI obtenidos. Cuantifica la mejora frente al programa original.

Desarrollo de un nuevo programa.

En esta parte de la práctica supondremos que las latencias del sumador y multiplicador son de 2 y 4 ciclos, respectivamente (opciones `-a 2 -m 4` al lanzar el simulador).

1. Escribe el código MIPS convencional para ejecutar la operación $\vec{Z} = a * \vec{X} + \vec{Y}$ (bucle DAXPY), siendo el tamaño de los vectores a procesar de 60 números en coma flotante de doble precisión.

   Puedes partir del programa almacenado en el fichero `daxpy.s`.

   IMPORTANTE: Si hay algún error del tipo "etiqueta indefinida o syntax error", lo más probable es que se deba a alguno de los errores mencionados en el Anexo A de este boletín.

   Ejecuta el programa en el simulador. Evalúa las prestaciones alcanzadas.

   ```
   mips-m -s html -d c -c pnt1 -a 2 -m 4 -f daxpy.s
   ```

2. Aplica la técnica del *loop unrolling* al código desarrollado, reorganizándolo, en su caso, también para reducir los ciclos de parada insertados.

   Puedes partir del programa realizado en el apartado a), copiándolo previamente a otro archivo (por ejemplo `daxpyu.s`). Escribe el nuevo código y ejecútalo. Evalúa las prestaciones alcanzadas, comparándolas con las de la versión base.

   ```
   mips-m -s html -d c -c pnt1 -a 2 -m 4 -f daxpyu.s
   ```

   ¿Se han eliminado todos los ciclos de parada? En caso negativo, explica cuál es la causa.

Subconjunto de instrucciones MIPS que soporta el simulador

- 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

- Carga/almacenamiento en coma flotante

l.d Fx, desp(Ry)
s.d Fz, desp(Ry)

- Aritmética en coma flotante

add.d Fx, Fy, Fz
sub.d Fx, Fy, Fz
mul.d Fx, Fy, Fz
div.d Fx, Fy, Fz

- Comparacion en coma flotante

c.eq.d Fy, Fz
c.ne.d Fy, Fz
c.lt.d Fy, Fz
c.le.d Fy, Fz
c.gt.d Fy, Fz
c.ge.d Fy, Fz

- 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 cargar datos en coma flotante de doble precisión, la instrucción correcta es *l.d* (por ejemplo *ld* es incorrecta en este caso).
- Para sumar enteros del tipo *dword*, la instrucción correcta es *dadd* (por ejemplo *add* es incorrecta en este caso).
- Para sumar datos en coma flotante de doble precisión, la instrucción correcta es *add.d* (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).
- Para multiplicar datos en coma flotante de doble precisión, la instrucción correcta es *mul.d* (por ejemplo *mult* es incorrecta en este caso).
- Para almacenar datos en coma flotante de doble precisión, la instrucción correcta es *s.d* (por ejemplo *sd* es incorrecta en este caso).