Práctica 13: Sincronización por interrupciones

1. Introducción y objetivos

El simulador PCSpim, además de sus periféricos, incluye parte de los registros e instrucciones MIPS para tratar las excepciones, por lo que puede utilizarse para realizar operaciones de entrada y salida sincronizadas por interrupción.

El objetivo final de la práctica es tratar dos periféricos mediante interrupciones. Para eso, se tendrá que diseñar el **código de inicialización** y el **manejador del sistema**.

Esta práctica está organizada en 7 pasos sucesivos, con la intención de resolver en cada uno un problema de diseño concreto. Los problemas de diseño son los siguientes:

- En el código de inicialización: la habilitación global de las interrupciones y la habilitación selectiva de las interrupciones del teclado y del reloj.
- En el manejador: la preservación del contexto del programa interrumpido, y el retorno al punto adecuado para que continúe su ejecución.
- En el manejador: aplicación de un tratamiento concreto al teclado y otro tratamiento al reloj.
- En el manejador: el diseño del árbol de decisiones que identifica la fuente de cada interrupción y escoge el tratamiento adecuado

Material

- La versión del simulador PCSpim-ES.
- Los archivos fuente *nada.asm*, *nada.handler* y *bucles.asm*.
- Archivos anexos: llamadas al sistema, el coprocesador y periféricos en PCSpim.

2. Las excepciones en el simulador PCSpim

En el procesador MIPS real hay diversas causas de excepción (resumidas en el documento "*Apéndice 2. Coprocesador.pdf*"). Con el simulador PCSpim que se va a utilizar sólo se pueden tratar dos de ellas: las interrupciones de periférico y las llamadas al sistema mediante la instrucción `syscall`. En esta práctica sólo se va a trabajar con interrupciones. La configuración apropiada del simulador, mediante *Simulator>Settings...* aparece en la figura 1.

Tarea 1. Observación del sistema con el simulador.

A lo largo de esta práctica, se trabajará al mismo tiempo con dos archivos:

- El *Trap file*, con la extensión *.handler*, que define los segmentos *.kdata* y *.ktext* del manejador de excepciones y un fragmento del segmento *.text* para el código de inicio y terminación que haga falta.
- El programa de usuario, con extensiones *.s* o *.asm*, describe el resto de los segmentos *.data* y *.text*.

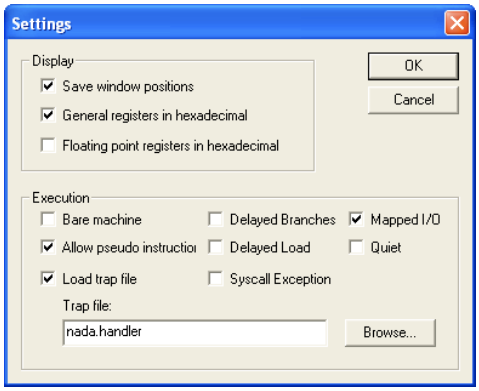

Figura 1. Menú de configuración para procesado de las interrupciones. El manejador se encuentra en nada.handler

Cada vez que se abre (*File>Open*) o recarga (*Simulator>Reload*) un archivo de usuario, el simulador carga también el archivo identificado como *Trap File* en el cuadro de configuración de la figura 1 (*Simulator>Settings...*).

La combinación de ambos archivos formará un **sistema** compuesto de un programa de usuario, un manejador de excepciones y un código de inicio. La figura 2 muestra el sistema completo.

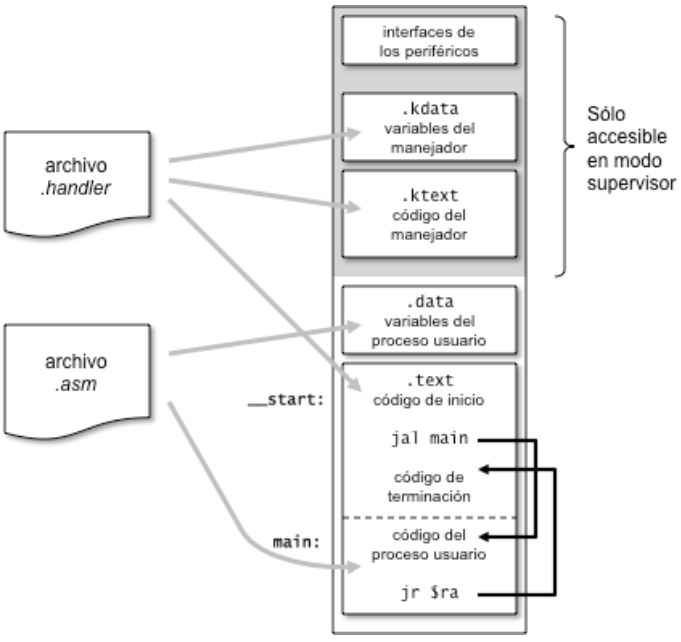

Figura 2. Los dos archivos fuente y su contribución a los distintos segmentos del sistema.

► Observe la figura 3 o abra con un editor de textos los archivos *nada.handler* y *nada.asm* para comprobar los siguientes detalles:

- En el archivo *nada.handle* se proporciona el contenido de los segmentos `.kdata`, `.ktext` y `.text`.
- El punto de comienzo de la ejecución (etiqueta `__start`) está localizado en el segmento `.text` del archivo *nada.handle*. Por tanto, a continuación de esa etiqueta va el código de inicio (en *nada.handler*, esta zona está vacía).

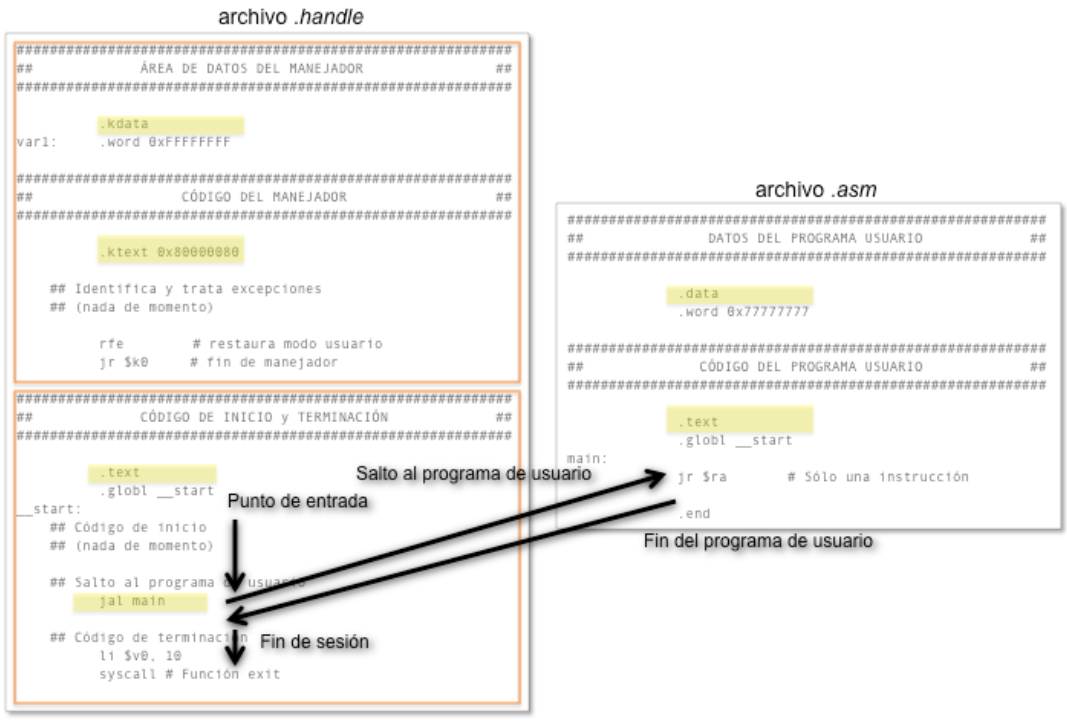

Figura 3. Flujo de ejecución del sistema

- En *nada.handler* la inicialización va seguida de un salto a la etiqueta `main`. El punto de entrada del programa usuario está marcado por esta etiqueta en el archivo *nada.asm*.
- El programa usuario debe acabar con una instrucción `jr $ra`, para que el control vuelva al código de terminación que se encuentra en *nada.handler*.
- El código de terminación consiste en la llamada `exit`.
- El código del manejador está fuera del flujo de ejecución aparente del sistema. El manejador se ejecutará sólo cuando se produzca una interrupción.

►Especifique el archivo *nada.handler* como *Trap File* y cargue *nada.asm*. Busque en las ventanas de estado, código, variables y mensajes los elementos marcados en la Figura 4 y que se enumeran a continuación:

- En la ventana de estado: los registros del coprocesador de excepciones.
- En la ventana de código: el contenido de los segmentos de código `.text` y `.ktext`. separados por la marca `KERNEL`. Nota que el código de inicialización va seguido

físicamente del programa de usuario porque ambos códigos se ubican en el mismo segmento de código `.text`.

- En la ventana de variables: las variables del segmento de datos de usuario van seguidas de la pila (marcada como `STACK`) y las variables del manejador (`KERNEL DATA`). Observe que *nada.handler* define una variable `var1` en el segmento `.kdata` cuyo valor inicial es `0xFFFFFFFF`, mientras que el programa usuario define `var2` en el segmento `.data` con el contenido `0x77777777`; en la ventana de variables aparece cada una en su segmento.
- En la ventana de mensajes: cuando se carga o recarga un archivo *.asm*, se indica qué archivo *.handler* se ha cargado. Si el código tiene errores, se indica aquí en el momento de la carga.

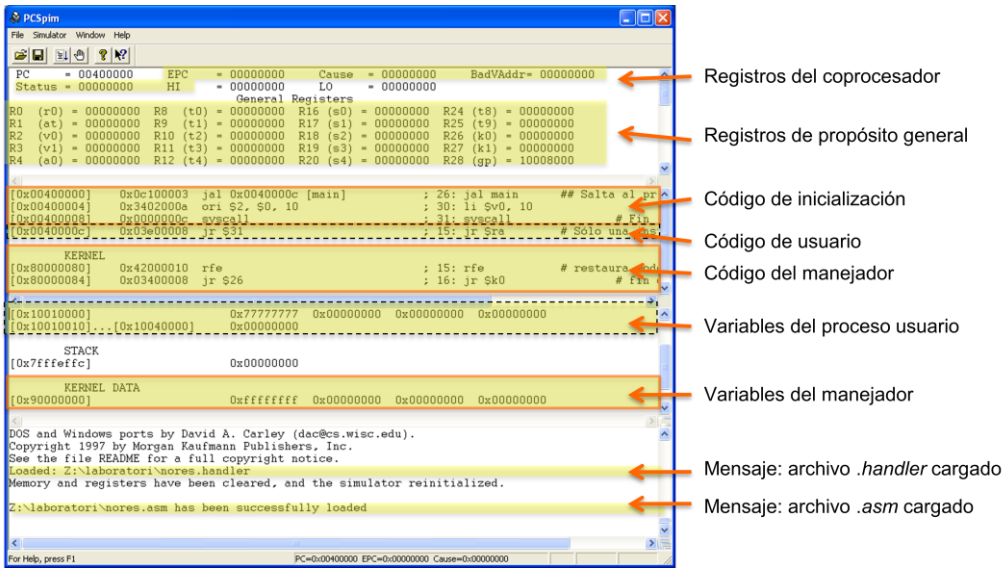

Figura 4. El simulador permite inspeccionar los registros del coprocesador de excepciones, el código y las variables del manejador junto con el código y las variables de usuario

3. Tratamiento de las interrupciones.

PCSpim dispone de tres fuentes de interrupción, correspondientes a los tres periféricos definidos en el simulador. Dos de ellos (teclado y consola) se estudiaron en la práctica anterior; el tercer dispositivo es un reloj. La descripción de las interfaces se muestra en el archivo "*Apéndice 3. Periféricos del PCSPIM.pdf*". Cada uno de estos periféricos está conectado a una línea de petición de interrupción, tal como se muestra en la Figura 5.

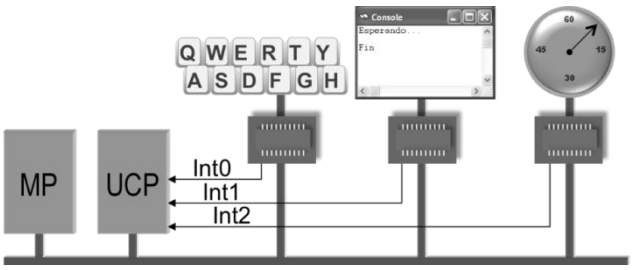

Figura 5. Conexión del teclado, consola y reloj a las líneas de interrupción de MIPS. La línea de interrupción de la consola (Int1) no se utilizará en esta práctica.

Tarea 2. Preparación de un programa de usuario para pruebas.

► Abre el archivo *bucle.asm* con un editor de textos. Observa que coniene un programa que consiste en un par de bucles anidados cuyo único propósito es mantener la CPU ocupada durante unos 5 o 10 segundos.

► Cárgalo con el simulador (se puede mantener el archivo *nada.handler* como *Trap File*) y ejecútalo. Si se ejecuta demasiado rápido o demasiado lento, modifica el valor inicial que regula el número de iteraciones del bucle exterior (línea `li $t0, 10`). **No olvides que siempre puedes detener la ejecución del programa pulsando *Control-C*.**

Tarea 3. Inicialización del sistema.

Para poder tratar las interrupciones de un periférico conectado a una línea de interrupción dada, es necesario hacer que:

- El periférico tenga habilitadas las interrupciones en su interfaz.
- La línea de interrupción esté desenmascarada en la palabra de estado del coprocesador.
- Las interrupciones estén habilitadas en la palabra de estado del procesador.

Es por ello que, en general, la inicialización del sistema requiere tres pasos:

1. Configurar los periféricos disponibles, habilitando o inhibiendo las interrupciones a través de sus respectivas interfaces, dependiendo de si se van a sincronizar por interrupción o no.
2. Configurar el registro de estado del coprocesador de excepciones (CP0). En concreto, se debe fijar la máscara de interrupciones (bits IM_i), la habilitación global de interrupciones (bit IE_c) y el modo de funcionamiento del procesador (bit KU_c).
3. Transferir el control al programa usuario.

Paso 1: Habilitación de interrupciones y modo de funcionamiento

► Con el editor de textos: Modifique *nada.handler* y guarde el archivo resultante como *teclado.handler*. Ha de crear el código de inicio apropiado (antes de `jal main`) para que el manejador trate **solamente** la interrupción de teclado.

La inicialización supone dos cosas:

- Habilitar las interrupciones en el registro de estado/órdenes del teclado, y
- Escribir el valor adecuado en la palabra de estado del coprocesador para que la línea de interrupción *Int0** quede desenmascarada, las interrupciones habilitadas y el modo usuario seleccionado. Para ello serán útiles las instrucciones `mfc0` y `mtc0`.

► Con el simulador: configure *teclado.handler* como *Trap File* en *Simulator>Settings...*. Cargue y ejecute *bucle.asm*. Si se ha habilitado y desenmascarado correctamente la interrupción de teclado, el simulador comenzará a dar mensajes del tipo "`Bad address in data/stack read`" nada más se pulse una tecla. En caso contrario, la simulación acabará normalmente.

► **Cuestión 1.** Escribe aquí el código de inicio correcto.

```
li $t0, 0xffff0000 # registro de estado/órdenes del teclado
li $t1, 0x2
sw $t1, 0($t0) # Habilita las interrupciones en el registro de estado/órdenes del teclado

mfc0 $t0, $12
ori $t0, $t0, 0x103
mtc0 $t0, $12
```

► **Cuestión 2**. ¿Por qué se producen los mensajes de error? **Pista**: busca en el código de retorno al programa, que está al final del manejador (hay un comentario `## Vuelve al programa`). ¿Es correcta la vuelta al programa?

No es correcta ya que al ejecutar "jr $k0" no regresa realmente al programa, porque para regresar al programa se debe cargar la dirección de la instrucción en la que se produjo la excepción, y esta dirección se encuentra en el registro $14 del coprocesador.

Tarea 4. Comprensión de la estructura del manejador

En los siguientes pasos se va a escribir el código completo de un manejador sencillo que trate las interrupciones. El manejador disponible en *nada.handler* sólo contiene comentarios marcando zonas para escribir código ejecutable, que se irá completando a lo largo de la práctica. De momento, **la única fuente de interrupciones es el teclado**.

Paso 2: Obtención de la dirección de retorno al programa de usuario

► Con el editor de textos: Añade una instrucción que escriba en *$k0* la dirección de retorno correcta al final del manejador.

► **Cuestión 3.** Copia aquí la línea de código que has escrito.

mfc0 $k0, $14

► Con el simulador: Carga y ejecuta de nuevo *bucles.asm*. Si todo va bien, el programa acabará normalmente aunque se pulse una tecla.

Paso 3: Tratamiento provisional de la interrupción de teclado

Se aplicará el siguiente **tratamiento incorrecto** que escribe un asterisco (carácter "*") en la consola cada vez que se produce una excepción. Esto es:

```
li $v0,11
li $a0, '*'
syscall
```

► Con el editor de textos: Escribe este tratamiento en el manejador, (donde dice "`identifica y trata excepciones`").

► Con el simulador: carga y ejecuta el sistema. Pulsa una tecla. Si todo va bien, comenzarán a aparecer asteriscos en la consola sin parar.

► **Cuestión 4.** ¿Por qué se escriben tantos asteriscos al pulsar una tecla?

Esto se debe porque una vez se escribe un asterisco debido a que se ha detectado una interrupción, se vuelve al programa, el procesador comprueba las peticiones pendientes tras la ejecución de cada instrucción y como la petición de interrupción del teclado no se cancela, se trata la interrupción sin parar.

Paso 4: Cancelación de la interrupción

Se ha de corregir el tratamiento anterior para que cancele la petición de interrupción del teclado durante el tratamiento de la interrupción.

► Con el editor de textos: Escribe el código que sirve para cancelar la interrupción. Escribe algo como esto:

```
li $t0, dirección base de la interfaz del teclado
lw $a0, registro de datos del teclado en la dirección ($t0 + 4)
```

► **Cuestión 5.** Copia aquí la/s línea/s de código que has escrito para cancelar la interrupción.

```
li $t0, 0xffff0000
lw $a0, 4($t0)
```

► Con el simulador: carga y ejecuta el sistema. El programa de usuario acabará en cuanto se pulse una tecla.

► **Cuestión 6.** ¿Por qué acaba el programa de usuario antes de lo esperado cuando se pulsa una tecla? **Pista:** el manejador modifica el registro *$t0* que utiliza el programa *bucles.asm*.

Esto se debe a que el manejador modifica el contenido del registro $t0 por 0xffff0000, y al ejecutarse la instrucción "bgtz $t0, bucleE" del programa bucles.asm, se interpreta el contenido de este registro como un número negativo, por lo que no se salta a la etiqueta bucleE, por lo que sale del bucle y el programa finaliza ejecutando sus últimas instrucciones.

Paso 5: Gestión del contexto

El manejador debe preservar el contexto de ejecución del programa de usuario detenido para no interferir con él. Por contexto se entiende el banco de registros, el contador de programa y cualquier otro dato contenido en el procesador que defina el estado de ejecución del programa. En nuestro caso, bastará con que el manejador, antes de hacer nada, almacene en memoria el contenido previo de **los registros que utilice**. A esta operación la llamaremos *salvar contexto*. Después, justo antes de volver al programa interrumpido, habrá que *restaurar* el valor de los registros preservados.

Figura 6.. Estructura del manejador que preserva el contexto del programa interrumpido

► Tome nota de qué registros utiliza el manejador por ahora. A primera vista son tres: *$t0*, *$a0* y *$v0*, pero se ha de contar también con el registro *$at* que utilizan de forma implícita pseudoinstrucciones como `li` o `sw`. Aunque el programa *bucles.asm* sólo utiliza unos pocos registros, conviene pensar en general: para que cualquier programa pueda funcionar correctamente con *teclado.handler*, el manejador debe preservar los cuatro registros que utiliza.

*Una salvedad importante: evitaremos que los programas de usuario utilicen los registros **$k0** y **$k1**. Utilizaremos $k1 como dirección de la zona donde se guardará el contexto, dentro del segmento de datos del manejador (.kdata), como se muestra en la figura 7. El registro $k0 lo reservamos para guardar la dirección de retorno al programa de usuario, de modo que el retorno desde el manejador se obtenga mediante `jr $k0` (que será la última instrucción ejecutada por el manejador).*

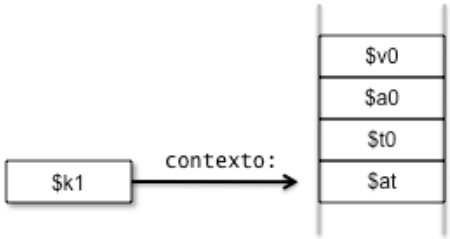

Figura 7. El registro *$k1* apunta de forma permanente a la zona de memoria donde se guarda el contexto.

► Se ha de añadir el código apropiado para preservar el contexto. Hay un problema adicional con el que se tiene que contar: normalmente, el ensamblador no permite utilizar el registro *$at* en las instrucciones. Si se escribe (tenga o no sentido) la instrucción

```
add    $t1, $t0, $at
```

el ensamblador emitirá el mensaje "`Register 1 is reserved for assembler on line` *xx* `of file` *yy*". Para evitar este error, una instrucción como esta tiene que aparecer entre dos directivas **`.set noat`** y **`.set at`**:

```
.set   noat
add    $t1, $t0, $at
.set   at
```

► **Cuestión 7.** Con el editor de textos, abre *teclado.handler* y haz y anota las siguientes modificaciones:

- En el segmento de datos del manejador: utiliza una etiqueta **`contexto`** a partir de la cual se reservará espacio para almacenar cuatro palabras.

```
.kdata
contexto: .word 0,0,0,0 # para salvar $at,$t0,$a0 y $v0
```

- En el código de inicio del sistema: guarda en *$k1* la dirección que corresponde a la etiqueta `contexto`.

```
la $k1, contexto
```

- Al principio del código del manejador, donde dice "preserva el contexto del programa usuario", escribe las instrucciones que guarden los cuatro registros (*$at*, *$t0*, *$a0*, *$v0*) a partir de la dirección correspondiente a `contexto`, tal como muestra la figura 7.

```
## Preserva el contexto del programa de usuario

.set noat
sw $at, 0($k1) # Salvo $at
.set at
sw $t0, 4($k1) # Salvo $t0
sw $a0, 8($k1) # Salvo $a0
sw $v0, 12($k1) # Salvo $v0
```

- Al final del código del manejador, donde dice “restaura el contexto”, escribe las instrucciones necesarias para restaurar el valor original de los registros de contexto.

```
## Restaura el contexto

.set noat
lw $at, 0($k1) # Resturo $at
.set at
lw $t0, 4($k1) # Restauro $t0
lw $a0, 8($k1) # Restauro $a0
lw $v0, 12($k1) # Restauro $v0
```

► Con el simulador: comprueba que el funcionamiento del sistema es correcto.

Resumen

Se ha escrito todo el código necesario para que el sistema responda a una sola fuente de interrupciones: el teclado.

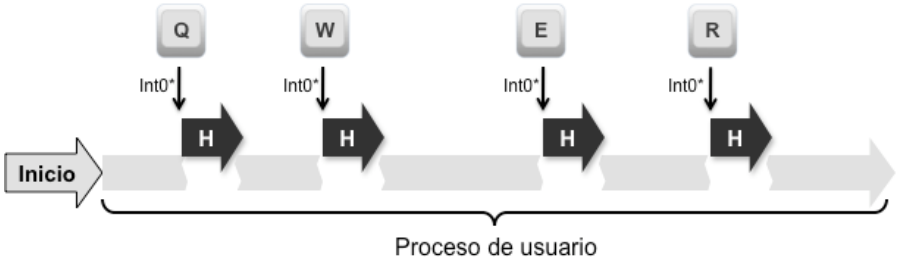

Figura 8. El manejador *H* se ejecuta cada vez que el teclado está preparado.

Tarea 5. Tratamiento de varias interrupciones

Cada vez que se añade una fuente de interrupción (y, en general, al tratar una nueva causa de excepción), es necesario:

- En el segmento de datos del manejador: definir las **variables** que sean necesarias para el tratamiento.
- En el código del manejador: escribir (1) las instrucciones que **identifican** la nueva interrupción y (2) el **tratamiento** indicado.
- En el código de inicio del sistema: **inicializar** el periférico y las variables necesarias para gestionarlo.

Como hasta ahora sólo se ha tratado una única fuente de interrupción, no ha sido necesario implementar en el manejador nada que la discrimine: siempre que hay una excepción, es la del teclado; por lo que no es necesario averiguar qué dispositivo ha provocado la interrupción.

Paso 6: Habilitación de las interrupciones del reloj

Añadiremos ahora el dispositivo reloj al sistema. En este paso, se habilitará la interrupción del reloj sin modificar el manejador.

► **Cuestión 8.** Con el editor de textos: abre el archivo *teclado.handler* y guárdalo con el nombre *teclado-y-reloj.handler*.

- En el código de inicio, añade las instrucciones que habilitan la interrupción del reloj.

```
## Habilitar interrupciones para el reloj
li $t0, 0xffff0010 # registro de estado/órdenes del reloj
li $t1, 0x1
sw $t1, 0($t0) # Habilita las interrupciones en el registro de estado/órdenes del reloj
```

- Modifica la palabra de estado del coprocesador para que la línea de interrupción *int2** esté desenmascarada.

```
mfc0 $t0, $12
ori $t0, $t0, 0x503
mtc0 $t0, $12
De esta manera queda desenmascarada int2* e int0* (reloj y teclado)
```

► Con el simulador: prueba el sistema. Si has habilitado correctamente la interrupción de reloj, la consola mostrará asteriscos sin parar.

► **Cuestión 9.** Explica por qué el sistema se comporta así. **Pista**: ¿Cancela el manejador la interrupción del reloj?

Ocurre algo similar a lo que sucedía con el teclado, cuando se produce una interrupción por el reloj, se escribe un asterisco, luego se vuelve al programa, el procesador comprueba las peticiones pendientes tras la ejecución de cada instrucción y como la petición de interrupción del reloj no se cancela, se trata la interrupción sin parar, mostrando asteriscos sin parar.

Paso 7: Análisis de la causa de excepción

Cada interrupción (y, en general, cada causa de excepción) ha de recibir su tratamiento específico. En este paso, se va a añadir un pequeño árbol de decisión en el manejador para asociar a cada fuente de interrupción su tratamiento e impedir que cualquier otra causa de excepción o fuente de interrupciones provoque una respuesta inadecuada. El objetivo es conseguir que el flujo de ejecución dentro del manejador se corresponda con la figura 9.

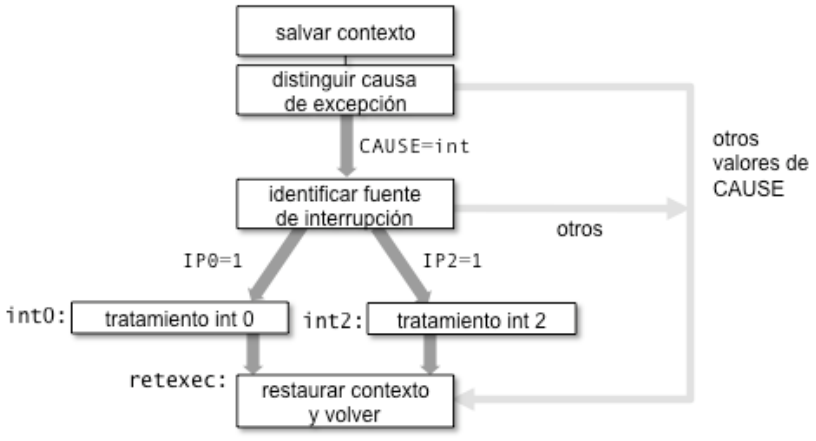

Figura 9. Estructura del manejador que atiende dos interrupciones

► **Cuestión 10**. Con el editor de textos: abre el archivo *teclado-y-reloj.handler*.

- Crea una etiqueta que marque el punto del manejador que restaura el contexto y devuelve el control al programa interrumpido (en la figura 9 y en las transparencias de teoría aparece `retexc`, pero se puede escoger la que se quiera). A partir de ahora, saltar a esa etiqueta desde dentro del manejador será equivalente a *finalizar*. Copia aquí el código a partir de `retexc`:

```
retexc:
   ## Restaura el contexto
   .set noat  .....
```

El resto de codigo se encuentra en la cuestión 7 apartado 4

- Crea una etiqueta `int0` para el tratamiento que desde los pasos 4 y 5 se aplica a la interrupción del teclado. Al final del tratamiento, también se ha de añadir el salto para *finalizar*.

```
int0:  li $v0,11
       li $a0, '*'
       syscall

       li $t0, 0xffff0000
       lw $a0, 4($t0)

       b retexc   # Salto para finalizar
```

- Crea una etiquea `int2` seguida del tratamiento del reloj. Como tratamiento, limítate a cancelar la interrupción en la interfaz, asegurándote de no inhabilitarla. No olvides *finalizar*.

```
int2:
   li $t0, 0xffff0010
   li $t1, 0x1
   sw $t1, 0($t0)
   # Cancela la interrupción pero deja habilitadas las interrupciones del reloj
   b retexc
```

- Añade las instrucciones que lean y aíslen el código de causa de excepción del coprocesador en la sección rotulada "`## Identifica y trata excepciones`" (la tabla de códigos aparece en el archivo "*Apéndice 1. Llamadas al sistema.pdf*"). Si el código es de interrupción, seguirá el análisis; en otro caso ha de saltar a la etiqueta que vale para *finalizar*.

```
## Identifica y trata excepciones
mfc0 $k0, $13 # Lee registro de Causa
andi $t0, $k0, 0x003c # Aísla los bits 2,3,4,5
bne $t0, $zero, retexc # Compara con 0 y salta si no son iguales
```

EC = 0, el motivo de excepción es una interrupción

- Una vez se ha establecido que la causa de la excepción es una interrupción, queda añadir las instrucciones que analicen los bits *IP0* e *IP2* de la palabra de estado del coprocesador de excepciones, y que salten a las etiquetas `int0` o `int2`. Si ninguno de estos bits está activo, habrá que *finalizar*.

```
## Causa por interrupción
andi $t0, $k0, 0x400 # miro int0
bne $t0, $zero, int0
andi $t0, $k0, 0x1000 # miro int2
bne $t0, $zero, int2
b retexc
```

►En todo momento, ten cuidado con los registros que se utilizan en el manejador. En el paso 5 se escribió el código que guarda y restaura los registros *$at*, *$t0*, *$v0* y *$a0*. Puedes limitarte a usar esos cuatro registros en el manejador, o bien utilizar otros registros (como el *$t1*) tomando

la precaución de ampliar la zona de `contexto` para guardarlos y restaurarlos junto con los demás.

► Con el simulador, comprueba el sistema. Si todo va bien, en la consola se mostrará un asterisco "*****" cada vez que pulses una tecla y nada más; el programa de usuario terminará una vez transcurrido el tiempo habitual.

ANEXO

Ensamblador para las excepciones

Nuevos segmentos

`.kdata` Describe datos que se encuentran en la zona restringida de la memoria (direcciones 80000000_h a $FFFFFFFF_h$). Estos datos sólo son accesibles en modo supervisor, así que los programas de usuario no pueden leer o escribir en ellos. Esta es la directiva adecuada para definir variables del manejador.

`.ktext` Describe instrucciones que se encuentran en la zona restringida de memoria. Sólo se pueden ejecutar en modo supervisor, así que los programas de usuario no pueden saltar a ellas. Esta es la directiva adecuada para definir el código ejecutable del manejador.

Nuevas instrucciones

`mfc0` *rt, rs* Transfiere el contenido del registro *rs* del coprocesador de excepciones al registro *rt* del banco de propósito general.

`mtc0` *rt, rs* Transfiere el contenido del registro *rt* del banco de propósito general al registro *rs* del coprocesador de excepciones.

`rfe` Realiza un desplazamiento a la derecha de dos posiciones de los bits de modo y habilitación del registro de estado (los seis bits menos significativos de este registro), deshaciendo el efecto de la última excepción. Habitualmente, esta instrucción sirve para volver a modo usuario al final de la ejecución del manejador de excepciones.