Practica 3. Gestión básica de interrupciones, excepciones y traps.

1. Objetivo

En esta práctica se pretende que el alumno comprenda los mecanismos hardware que dan soporte a interrupciones, excepciones y traps, de forma que sea capaz de definir y utilizar rutinas básicas para su gestión. El dominio de estás rutinas permitirá al alumno entender como, a partir de ellas, se pueden definir otros servicios de más alto nivel propios de los sistemas software empotrados (drivers de dispositivos o llamadas al sistema), y como tratar las situaciones excepcionales que se pueden producir durante su ejecución.

2. Introducción

Los mecanismos de atención a los eventos (interrupciones, excepciones y traps) que proporcionan los procesadores facilitan el acceso a los recursos del sistema de forma protegida, asegurando su integridad frente a los usos inadecuados.

Por una parte, mediante instrucciones tipo *TRAP* (también llamadas interrupciones software) el usuario puede solicitar servicios que fueron configurados durante la etapa de inicialización del sistema. Estos servicios permiten gestionar abstracciones tales como los sistemas de archivos, los procesos o el acceso a puertos de comunicación.

El mecanismo de las interrupciones, por su parte, permite a los dispositivos externos solicitar la atención del procesador, con el fin de que se ejecute una rutina como respuesta a su petición. Gracias a este mecanismo es posible evitar el control por sondeo de los trabajos asignados a los dispositivos.

Las excepciones, finalmente, permiten definir qué hacer cuando el procesador se encuentra en un estado no estable como cuando ejecuta una instrucción cuyo código no es válido, o se produce una división por cero o un overflow en las instrucciones de operación aritmética.

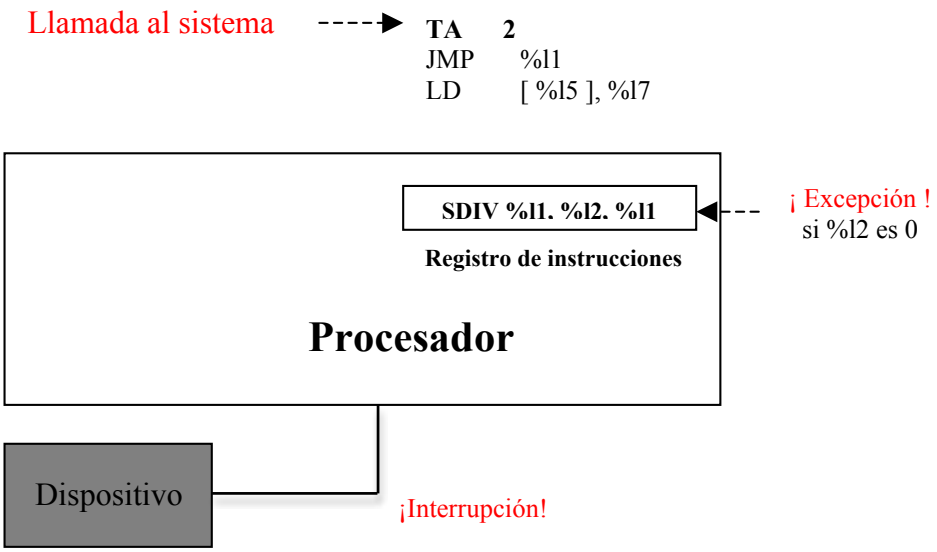

Figura 1. Mecanismos de atención a eventos proporcionados por el procesador.

© Óscar Rodríguez Polo 2012-2013

3. Tabla de Vectores de Interrupción

La tabla de vectores de interrupción es una zona de memoria en la que el procesador localiza el código a ejecutar cuando se produce una interrupción, una excepción o se ejecuta una instrucción de tipo *TRAP*. La estructura de esta tabla depende de cada procesador, y puede tener tamaños muy diferentes en función de si se trata de un procesador de propósito general o de una pequeña CPU de 8 bits integrada en un microcontrolador. La figura 2 muestra una estructura de tabla de vectores de interrupción genérica donde cada evento es antendido por el procesador ejecutando el manejador cuya dirección se encuentra almacenada en el propio vector.

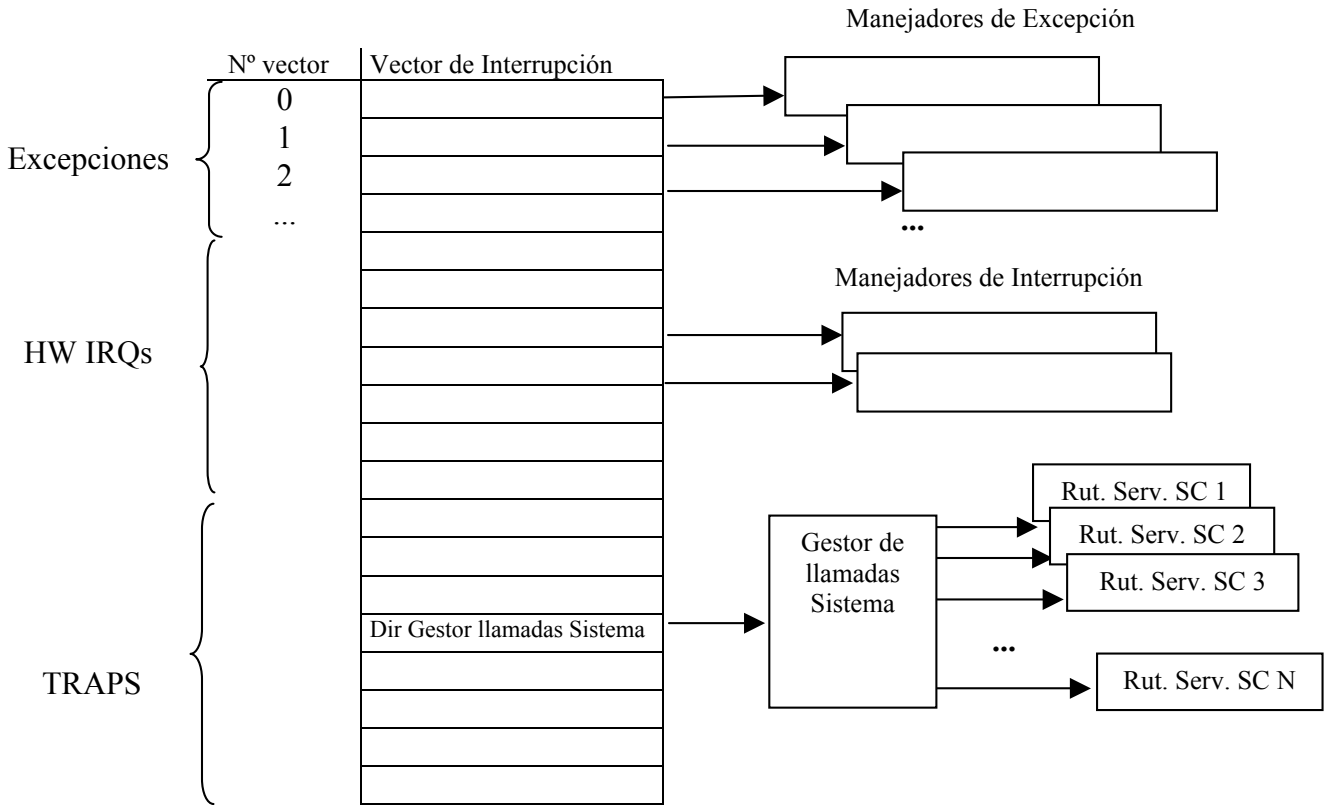

Figura 2. Tabla genérica de vectores de interrupción.

Esta tabla debe ser configurada durante la inicialiación del sistema (en el caso de contar con un Sistema Operativo, será éste el que se encarge de fijar la configuración inicial) . De esta forma, al ejecutarse posteriormente las aplicaciones el sistema ya será estable, y tendrá una respuesta controlada a cualquier evento que pueda ocurrir.

Una configuración que los sistemas operativos emplean habitualmente es la de utilizar un único vector de la tabla para gestionar todas las llamadas al sistema. Una rutina, denominada *gestor de llamadas al sistema,* centraliza todas las peticiones, y en función de un parámetro recibido (bien a través de un registro predeterminado, bien a través de la pila) identifica la llamada al sistema solicitada, e invoca a su rutina de servicio.

4. Tabla de Vectores de Interrupción del procesador LEON3

El procesador LEON3 presenta una estructura de tabla de vectores de interrupción en la que cada elemento de la tabla no es una dirección a la que saltar, sino un conjunto de 4 instrucciones que el procesador ejecuta diréctamente cada vez que se produce el evento.

Nº vector	Vector de Interrupción			
X	Instrucción-1	Instrucción-2	Instrucción-3	Instrucción-4

Esta estructura permite definir una respuesta rápida al evento. Sin embargo, cuando la respuesta es compleja, es necesario saltar a una rutina para poder definir la respuesta. La siguiente configuración de un vector, define cuatro instrucciones que permiten saltar a una función denominada `handler`, guardando en el registro `%l0` el registro de estado `%psr`, y en el registro `%l3` el número del vector. Ambos registros podrán ser utilizados por la rutina `handler` para realizar un tratamiento adecuado al evento.

Vector de Interrupción			
`rd %psr,%l0`	`sethi %hi(handler), %l4`	`jmp %l4 + %lo(handler)`	`mov vector,%l3`

Para poder manejar mejor la gestión de esta tabla de vectores de interrupción, es de gran utilidad definirse funciones que, a partir de el número del vector, y de un puntero a la rutina, gestione la forma en la que esta tabla debe completarse. Un prototipo de función con estas caractarísticas sería el siguiente:

```
void leon3_install_handler(   uint32_t vector_num ,
                              void (* handler) (void))
```

El parámetro `vector_num` corresponde al número de vector, mientras que `handler` es el puntero a la función que queremos que se invoque cuando se atiende al evento.

La siguiente figura muestra la tabla de vectores de interrupción del procesador LEON3, donde **TT** indica el número de vector, **Trap** el evento al que está asociado el vector (puede ser una excepción, una interrupción hardware o un trap provocado por una instrucción) y **Pri** la prioridad con la que se atiende al evento. Las primeras 15 entradas corresponden a excepciones del procesador, las 15 siguientes son interrupciones externas, mientras que el rango definido por 0x80-0xFF está reservado para ser invocado a través de la instrucción *TRAPS*

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error during data store
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Hardware breakpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
cp_exception	0x28	11	Co-processor exception
data_access_exception	0x09	13	Access error during data load, MMU page fault
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6
interrupt_level_7	0x17	25	Asynchronous interrupt 7
interrupt_level_8	0x18	24	Asynchronous interrupt 8
interrupt_level_9	0x19	23	Asynchronous interrupt 9
interrupt_level_10	0x1A	22	Asynchronous interrupt 10
interrupt_level_11	0x1B	21	Asynchronous interrupt 11
interrupt_level_12	0x1C	20	Asynchronous interrupt 12
interrupt_level_13	0x1D	19	Asynchronous interrupt 13
interrupt_level_14	0x1E	18	Asynchronous interrupt 14
interrupt_level_15	0x1F	17	Asynchronous interrupt 15
trap_instruction	0x80 - 0xFF	16	Software trap instruction (TA)

5. Creación de proyecto para LEON3

Creación de un nuevo proyecto denominado prac3 cuyo ejecutable sea para la plataforma Sparc Bare C. En ese proyecto crear dos subdirectorios **include** y **src**.
En el directorio **src** añadir los archivos *leon3_bprint.c y leon_uart.c* de la práctica anterior e igualmente los archivos *leon3_irqs_asm.S leon3_traps.c y leon3_irqs.c* que encontrarás también en el archivo enlazado como *prac3_fuentes* en la página web. El contenido de cada uno de estos archivos se determina a continuación:

- *leon3_traps.c.* Implementa la función `leon3_install_handler` que permite instalar una rutina de atención a un evento, independientemente de si es una interrupción, un trap o una excepción. El prototipo de la función es el siguiente, donde `vector_num` es el número de vector y `hanlder` la rutina a instalar:

```
void leon3_install_handler(   uint32_t vector_num ,
                              void (* handler) (void))
```

- *leon3_irqs.c.* Gestiona solamente interrupciones (no excepciones, ni traps). Implementa parcialmente las siguientes funciones:
  - `leon3_mask_irq(int32_t irq_level)` permite enmascarar uno de los 15 niveles de interrupción externa poniendo a 0 en el registro registro IMASK el bit correspondiente al nivel (este registro está ubicado en la dirección 0x80000240). El número de nivel debe suministrarse mediante el parámetro `irq_level.`
  - `leon3_unmask_irq(int32_t irq_level)` permite desenmascarar uno de los 15 niveles de interrupción externa poniendo a 1 en el registro registro IMASK el bit correspondiente al nivel. El número de nivel debe suministrarse mediante el parámetro `irq_level.`
  - `leon3_force_irq(int32_t irq_level)` permite forzar el disparo de uno de los 15 niveles de interrupción externa poniendo a 1 en el registro IFORDE el bit correspondiente al nivel (este registro está ubicado en la dirección 0x80000240). El número de nivel debe suministrarse mediante el parámetro `irq_level.`

- *leon3_irqs_asm.S.* Archivo en ensamblador que implementa las siguientes funciones:

  - `leon3_trap_handler_enable_irqs` rutina de atención a un trap que permite habilitar todas las interrupciones que no estén enmascaradas en el registro IMASK.
  - `leon3_trap_handler_disable_irqs` rutina de atención a un trap que permite deshabilitar todas las interrupciones, independientemente de cómo esté configurada su máscara en el registro IMASK.
  - `leon3_sys_call_enable_irqs` llamada al sistema, efectuada a través de un *TRAP*, que permite llamar a la rutina de atención `leon3_trap_handler_enable_irqs.`
  - `leon3_sys_call_disable_irqs(void)` llamada al sistema, efectuada a través de un *TRAP*, que permite llamar a la rutina de atención `leon3_trap_handler_disable_irqs.`

En el directorio **include** añadir los archivos *leon3_bprint.h y leon_uart.h* de la práctica anterior. Añadir también los archivos *leon3_asm.h leon3_traps.h y leon3_irqs.h* que igualmente encontrarás en el archivo enlazado como *prac3_fuentes* en la página web. El contenido de estos tres últimos archivos es el siguiente:

- *leon3_asm.h* Macros en ensamblador de utilidad para la definición de las funciones implementadas en *leon3_irqs_asm.S*
- *leon3_irqs.h* Archivo para la declaración de las funciones definidas en *leon3_irqs_asm.S y leon3_irqs.c*
- *leon3_traps.h* Archivo para la declaración de las funciones definidas en *leon3_traps.c*

6. Tarea a realizar.

1. Completar las funciones `leon3_mask_irq, leon3_unmask_irq y leon3_force_irq` para que se gestionen de forma adecuada las máscaras y el registro que fuerza el disparo de interrupciones.
2. Implementar en el archivo *leon3_bprint.c* la función `leon3_print_uint32` con el siguiente prototipo. Esta función implementa una función análoga a la función `leon3_print_uint8` que se realizó en la práctica anterior, pero trabajando con un entero de 32 bits. (Añade esta declaración a *leon3_bprint.h* para que se pueda usar la función)

```
int8_t leon3_print_uint32(   uint32_t i);
```

3. Comprobar la valided de la implementación con el siguiente programa principal completando las partes que faltan:

```
#include "leon3_uart.h"
#include "leon3_bprint.h"
#include "leon3_traps.h"
#include "leon3_irqs.h"

void hw_irq_vector_0x11_handler(void)

      leon3_print_string("handler hw irq vector 0x11\n");
}

int main()
{

      //Instalar como manejador del trap 0x83 la rutina
      // que habilita las interrupciones
      leon3_install_handler(0x83,leon3_trap_handler_enable_irqs);

      //Instalar el manejador del trap que 0x83 la rutina
      // que deshabilita las interrupciones
      leon3_install_handler(0x84,
                              leon3_trap_handler_disable_irqs);
```

```
      //Instalar la función hw_irq_vector_0x11_handler como
      // manejador del trap el manejador del trap

      //COMPLETAR
      //
      //

      //Habilitar las interrupciones
      leon3_sys_call_enable_irqs();

      //Desenmascarar la interrupcion de nivel 1 (correspondiente
      //al vector 0x11)
      leon3_unmask_irq(1);

      //Fuerza la interrupción
      leon3_sparc_force_irq(1);

      return 0;

}
```

Comprobar que la salida que se da por pantalla es la siguiente:

```
hw irq level 0
handler hw irq vector 11
```

4. Repetir la ejecución poniendo un breakpoint en las función `leon3_sys_call_enable_irqs` definida en *leon3_irqs_asm.S*. Ejecutar paso a paso para comprobar cual es el comportamiento. ¿A qué función salta tras la instrucción en ensamblador **ta** ?
5. Repetir la ejecución enmascarando la interrupción (usa `leon3_mask_irq`) antes de `leon3_sparc_force_irq(1)` y comprobar que no se genera ningún mensaje por pantalla.
6. Hacer lo mismo pero llamando a `leon3_sys_call_disable_irqs()` en vez de `a leon3_sys_call_enable_irqs() y` comprobando que tampoco se genera ningún mensaje.
7. Ejecutar el siguiente código. ¿Qué ocurre? Sabiendo que cuando se produce una división por 0 la rutina que lo gestiona llama al trap 0x82, ¿cómo utilizarías la función `leon3_install_handler` para conseguir que el programa no se cuelgue y en su lugar imprima un mensaje que diga “error, división por cero”?

```
      int main()
          uint8_t i;
          uint8_t j;

          for(i=10; i>0; i--)
            j=j/(i-9);
      }
```