Proyecto de Ingeniería de Computadores (PEC)

Objetivo del curso

El objetivo de esta asignatura es que el alumno aprenda a desarrollar un prototipo de un computador o un SoC (System on Chip) en un chip programable sobre una placa base para crear un mini-ordenador. Se pondrán en práctica algunos de los conocimientos adquiridos en asignaturas anteriores sobre el diseño de la microarquitectura de un procesador, sobre el diseño e implementación de software de sistema, y sobre el diseño de sistemas digitales.

Fases principales del proyecto.

1) Aprendizaje de las herramientas de desarrollo para los chips programables (FPGA) y práctica del lenguaje de descripción del hardware VHDL.
2) Implementar pequeños componentes o dispositivos en el chip programable de la placa base.
3) Implementar una primera versión simplificada del procesador en una FPGA (sin memoria externa, ni soporte para el sistema operativo o dispositivos externos)
4) Implementar una versión completa del procesador.
5) Programar un sistema de arranque (BIOS) para el Sistema Operativo en el procesador.
6) Evaluar el rendimiento de varias aplicaciones sobre la plataforma que se ha diseñado.

Objetivo de las sesiones

En estas sesiones vamos a implementar físicamente un procesador sencillo en la FPGA. Lo haremos por etapas. En cada etapa cogeremos el trabajo realizado en la anterior y le añadiremos o modificaremos algún componente.

Etapa 7: Interrupciones, Excepciones y el Modo Sistema.

En esta etapa añadiremos soporte para manejar interrupciones y excepciones. Hasta este momento todos los accesos a los dispositivos de entrada/salida se han realizado mediante encuesta (*polling*) pero ello requiere que la iniciativa la lleve el programa que se está ejecutando en el procesador y que constantemente esté consultando si se ha producido un evento. Estas constantes consultas degradan el rendimiento del equipo, en un sistema no multitarea esto no es relativamente importante pero en uno multitarea si lo es. Además, el acceso a los dispositivos por encuesta, no es el modo habitual de funcionamiento de un procesador.

Para implementar y gestionar las interrupciones, excepciones y el modo sistema en el SISA es necesario un banco de registros especial adicional, un conjunto nuevo de instrucciones para gestionar este nuevo banco de registros (**`RDS`** y **`WRS`**) y un conjunto de instrucciones para gestionar las interrupciones (**`EI, DI, RETI`** y **`GETIID`**). Además, deberemos modificar los dispositivos que tenemos implementados para que puedan usarse mediante interrupciones.

Si no recordáis el concepto de interrupción y/o excepción, sus fases y su programación, en el anexo de este documento podéis encontrar un breve resumen.

Banco de Registros de Sistema

Veamos cómo es la estructura del Banco de Registros de Sistema (en la documentación del procesador tenéis más información) para poder dar soporte a las interrupciones y excepciones. El Banco de registros deberá continuar teniendo los 8 registros generales como hasta ahora (R0, R1, …, R7) pero además deberá tener 8 registros especiales (S0, S1, …, S7) cada uno con un uso predeterminado.

Los registros especiales solamente se pueden acceder explícitamente mediante las instrucciones **`RDS`** y **`WRS`**. La instrucción **`RDS`** lee un registro especial S_i y lo escribe en un registro general R_i y la instrucción **`WRS`** lee un registro general y lo escribe en uno especial. También las instrucciones **`EI`**, **`DI`** y **`RETI`** modifican el registro S7, aunque no

aparezca como operando explícito de la instrucción. Por otro lado, como resultado de la ejecución de otras instrucciones se pueden modificar algunos registros especiales, por ejemplo, si se produce una excepción. También se modifican cuando ocurre una interrupción externa.

Los registros especiales son 8 registros de 16 bits: S0, S1, …, S7. Su uso general es el siguiente (aunque en este estado de diseño del procesador aún haya algunas cosas que no hemos definido aún, o que no implementaremos como todo lo relacionado con la coma flotante):

S0. Contiene la palabra de estado que tenía el sistema cuando se produjo una excepción (interna), o interrupción (externa). Cuando se produce uno de estos eventos, se salva en S0 la palabra de estado actual, que se encuentra en S7, ya que acto seguido se cambia la palabra de estado actual, para inhibir las interrupciones (externas).

S1. Contiene la dirección de retorno después de una interrupción (externa), una excepción (interna). Cuando se produce uno de estos eventos, se salva en S1 el PC actualizado, ya que acto seguido se copia en el PC el registro S5, que contiene la dirección del código de entrada de las rutinas de atención de interrupciones y excepciones.

S2. Contiene, en sus cuatro bits de menor peso, un código binario que puede tomar valores de 0 a 15 que identifica el tipo de evento que se ha producido: tipo concreto de excepción (interna) o interrupción (externa). El resto de bits de S2 están siempre a 0. El código es:

- 0: Excepción de tipo "Instrucción ilegal".
- 1: Excepción de tipo "Acceso a memoria mal alineado".
- 2: Excepción de tipo "Overflow en operación de coma flotante".
- 3: Excepción de tipo "División por cero en coma flotante"
- 4: Excepción de tipo "División por cero en enteros o naturales"
- 5,...,14: No usadas en esta versión. Reservados para futuras ampliaciones.
- 15: Interrupción (externa)

S3. Contiene la dirección efectiva usada por una instrucción de acceso a memoria, cuando ésta provoca una excepción de acceso a memoria mal alineado.

S4. Puede ser usado por el programador como variable temporal.

S5. Contiene la dirección de memoria donde se encuentra la siguiente instrucción a ejecutar después de producirse una excepción (interna) o interrupción (externa). En esta dirección de memoria comienza el código de la rutina de servicio genérica (RSG) encargado de identificar la causa de excepción o interrupción y en cada caso llamar a la rutina de servicio de excepción (RSE) o a la rutina de servicio de interrupción (RSI) específica correspondiente.

S6. Su uso se reserva para ampliaciones futuras del lenguaje.

S7. Es el PSW (*Processor Status Word*). Contiene el estado del procesador en todo momento. Cada bit tiene el siguiente significado:

- Bit M (PSW<0>): Se reserva para ampliaciones futuras del lenguaje.
- Bit I (PSW<1>): Bit que indica si las interrupciones están permitidas (en cuyo caso vale 1) o si están inhibidas (en cuyo caso vale 0). Este bit se modifica con las instrucciones **`EI`** (Enable Interrupt) y **`DI`** (Disable Interrupt), que lo ponen a 1 o a 0 respectivamente y la instrucción **`RETI`** que restaura la palabra de estado y retorna de una excepción o interrupción. También, se pone a 1 cuando se produce una excepción o interrupción.
- Bit V (PSW<2>): Bit que indica si la excepción de overflow en operación de coma flotante está permitida (en cuyo caso vale 1) o si está inhibida (en cuyo caso vale 0).

Interrupciones y Excepciones

Veamos cómo trata las interrupciones y las excepciones el procesador SISA. Cuando se produce una petición de interrupción (externa) ésta será tratada si las interrupciones están globalmente permitidas: el bit I de la palabra de estado vale 1. Cuando se produce una excepción (interna), ésta será normalmente tratada, excepto en el caso de que sea enmascarable (sólo lo es la excepción de overflow de coma flotante) y no esté permitida: si el bit V de la palabra de estado vale 0. Si la excepción o interrupción está permitida, entonces

1. se guarda en S0 la palabra de estado, que se encuentra en S7,
2. se guarda en S1 la dirección del PC actualizado, *PCup*, que es la dirección de retorno, ya que se debe retornar a la siguiente instrucción que provoco el fallo,
3. se escribe en S2 el código del tipo de evento ocurrido, para informar a la rutina de servicio (RSG) de lo ocurrido,
4. se escribe en S3, en el caso de una excepción de acceso mal alineado a memoria, la dirección efectiva impar utilizada por la instrucción que provocó la excepción,
5. se carga en el PC el contenido del registro S5, que tiene la dirección del único punto de entrada a la rutina de servicio de excepciones/interrupciones genérica (RSG),
6. se modifica la palabra de estado, registro S7, inhibiendo las interrupciones: PSW<1>=0, y
7. se pasa a ejecutar la instrucción que indica el PC, que es la primera instrucción de entrada a la RSG.

Para permitir que la RSG pueda ser interrumpida en caso de recibirse una petición de interrupción (externa), habría que permitir las interrupciones de nuevo, ejecutando la instrucción **`EI`**, puesto que éstas se inhiben cada vez que se invoca la RSG. La correcta programación para admitir interrupciones anidadas requiere un diseño cuidadoso de la RSG que no se va a explicar aquí. Por otro lado, el código de las rutinas RSG, RSE y RSI debería estar siempre libre de excepciones, de modo que tampoco se pueda interrumpir su ejecución a causa de una excepción, lo cual invocaría de nuevo la RSG. No obstante, puesto que esa excepción podría ser causada de modo imprevisto por un código incorrecto, es conveniente que la RSG salve y restaure no sólo los registros Ri sino también S0, S1 y S3 antes de llamar a una rutina RSE/RSI susceptible de producir excepciones. En resumen, la rutina RSG debería seguir la siguiente secuencia de pasos al ser invocada:

1. Salvar en la pila todos los registros Ri que vayan a ser modificados por la rutina, sin excepción.
2. Salvar en la pila los registros S0 (donde se guardó la palabra de estado de retorno) y S1 (donde se guardó la dirección de retorno), y S3 (donde se guarda la dirección efectiva mal alineada en caso de ser una excepción de este tipo).
3. Copiar S2 (el tipo de evento que activó la ejecución de la rutina) en un registro R para examinarlo. Si S2 contiene un número entre 0 y 14, se trata de una excepción, y en tal caso se indexará con S2 en una tabla - el vector de excepciones - para obtener la dirección de la RSE específica correspondiente. Si S2 vale 15, entonces se trata de una interrupción, y debe de ejecutarse la instrucción **`GETIID`** para solicitar y cargar en un registro el identificador del dispositivo (al ejecutarse **`GETIID`**, el dispositivo se entera de que se ha aceptado su interrupción y envía su identificador por el bus). Después hay que indexar con ese identificador en una tabla - el vector de interrupciones – para obtener la dirección de la RSI específica correspondiente.
4. Saltar a la dirección de la rutina específica de excepción (RSE) o de interrupción (RSI), y ejecutarla.

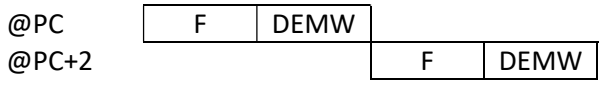
Al regresar de la RSE o RSI (aunque no siempre retornan), la RSG debe de efectuar la siguiente secuencia de pasos:

1. Restaurar los valores de S0, S1 y S3 que se salvaron en la pila
2. Restaurar los registros salvados en la pila
3. Ejecutar la instrucción **`RETI`** la cual restaura en el PSW, esto es en S7, la palabra de estado que tenía el programa cuando se produjo el evento, y que se encuentra en S0; y carga en el PC la dirección de retorno al código interrumpido, y que se encuentra en S1.

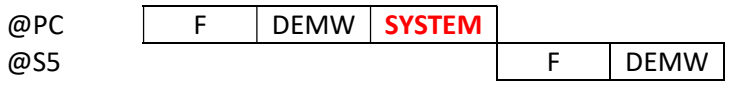
Subetapa 7.1: Implementación de las interrupciones

Si nos fijamos en las tareas a realizar, descritas en el apartado anterior, cada vez que se produce una interrupción veremos que hay que escribir diversos valores en varios registros de sistema. Hay varias posibilidades correctas de hacerlo. Básicamente haciendo todas las escrituras en un único ciclo o una en cada ciclo. Si escogemos está última, aparte de ser más ineficiente, la unidad de control se complica un poco más.

Los estados que ejecuta la unidad de control cuando no hay ninguna interrupción, ni excepción ni llamadas al sistema es el siguiente:



En cambio, si se produce una interrupción, hay que añadir un estado extra y la secuencia de estados es la siguiente.



Aparece un ciclo (o varios) nuevo al que hemos llamado SYSTEM que se encarga de realizar todas las tareas necesarias para poder ejecutar la primera instrucción correspondiente a la rutina de servicio genérica (RSG). Dicha dirección estaba almacenada en S5. En resumen, debe realizar las siguientes tareas:

- S0 ← S7 (se copia la palabra de estado PSW)
- S1 ← PCup (hace un backup del PC de retorno)
- S2 ← 0x000F (indica que el evento producido es una interrupción)
- PC ← S5 (rutina de sistema a la que se llama)
- S7<1> ← 0 (se cambia la PSW inhibiendo las interrupciones)

Finalmente, recordad que el procesador debe inicializarse con las interrupciones inhibidas ya que hasta que no se haya almacenado en el registro S5 la dirección de la RSG, si llegase alguna interrupción el comportamiento del procesador seria impredecible.

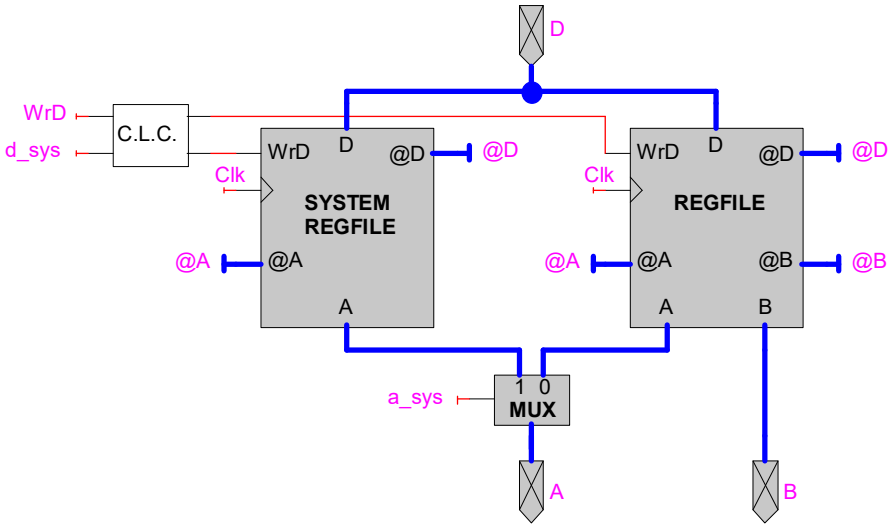
Banco de registros

El banco de registros debe permitirnos hacer operaciones de lectura y escritura sobre los registros de sistema cuando las instrucciones lo requieran. Únicamente tenemos dos instrucciones que acceden al Banco de Sistema que son las siguientes.

RDS Rd, Sa (Rd←Sa)

WRS Sd, Ra (Sd←Ra)

Además, podemos observar que nunca se hacen escrituras o lecturas simultáneas en ambos bancos. De modo que simplemente añadiendo un bloque de registros nuevo y una señal de control sería suficiente. La estructura básica del Banco de registros podría ser parecida a la siguiente.



Necesitaremos una señal para indicarnos si el registro que deseamos leer corresponde al banco de registros generales o al de sistema. También necesitaremos una señal para indicar en que banco queremos escribir un valor en caso de que estemos ejecutando una instrucción cuyo resultado se escriba en el banco de registros.

Además, para poder implementar las acciones correspondientes al ciclo en que se prepara la interrupción, el que anteriormente hemos llamado SYSTEM, necesitaremos una señal de *boot* para la inicialización, una señal que indique si ese ciclo se trata de una interrupción, el valor *PCup* proveniente de la unidad de control y un bus de salida para enviar el valor del registro S5 hasta el PC. Este último bus nos lo podemos ahorrar si usamos el mismo que para los saltos (Ra).

A parte de las instrucciones que acceden directamente a los valores del banco de registros de sistema (**RDS** y **WRS**), hay otras instrucciones que modifican algún valor de este banco. Las instrucciones **EI** y **DI** permiten o inhiben respectivamente las interrupciones en el procesador. Simplemente modifican el valor del bit S7<1> (PSW.I).

- EI → S7<1> ← 1 (se cambia la PSW permitiendo las interrupciones)
- DI → S7<1> ← 0 (se cambia la PSW inhibiendo las interrupciones)

La instrucción **RETI** realiza dos tareas. Restaurar la palabra de estado y enviar la dirección de retorno de la ejecución de la rutina RSG que estaba almacenada en S1.

- S7 ← S0 (se restaura la palabra de estado)
- PC ← S1 (restaura el PC para volver al código que ha sido interrumpido)

Para implementar estas tres instrucciones deberemos hacer varias modificaciones más al Banco de registros para que tenga en cuenta estos casos particulares. Necesitaremos señales nuevas para indicarnos si se trata de cada una de estas instrucciones concretas.

Unidad de control

La unidad de control debe generar unas cuantas señales nuevas y adaptar, en caso de que sea necesario, las ya existentes (todo dependerá de cómo hayáis implementado el banco de registros de sistema y otros módulos).

Veamos las señales nuevas. Se ha de generar una señal para indicar si en una lectura de un registro del Banco de registros, el registro a leer es general o de sistema. Igualmente, otra para indicar lo mismo en el caso de una escritura. El valor de estas señales dependerá básicamente de la ejecución de instrucciones **RDS** y **WRS**.

Para poder decodificar correctamente las instrucciones es necesario saber cómo se codifican. La siguiente tabla muestra la codificación de las instrucciones **RDS** y **WRS**.

15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0		
1 1 1 1	Rd	Sa	1 0 1 1 0 0	**ReaD S**ystem privileged register	RDS
1 1 1 1	Sd	Ra	1 1 0 0 0 0	**WR**ite **S**ystem privileged register	WRS

También hay que generar unas señales nuevas para indicar si estamos ejecutando alguna instrucción **EI**, **DI** o **RETI**, ya que estas afectan al contenido del banco de registros.

Para poder decodificar correctamente las instrucciones es necesario saber cómo se codifican. La siguiente tabla muestra la codificación de las instrucciones **EI**, **DI** y **RETI**.

15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0		
1 1 1 1	0 0 0	0 0 0	1 0 0 0 0 0	**E**nable **I**nterrupts	EI
1 1 1 1	0 0 0	0 0 0	1 0 0 0 0 1	**D**isable **I**nterrupts	DI
1 1 1 1	0 0 0	0 0 0	1 0 0 1 0 0	**RET**urn from **I**nterrupt	RETI

El valor de las señales puede depender del ciclo en que estemos, así que los valores los generaremos en el módulo *control_logic*, como hasta ahora, y aquellas que sean dependientes del ciclo de ejecución las filtraremos en el módulo *multi*. Además, deberemos adaptar el grafo de estados del módulo *multi* para que tenga en cuenta el nuevo ciclo, al que hemos llamado SYSTEM, en caso que deba ejecutarse.

También en el módulo *unidad_control* deberemos revisar cómo se actualiza el PC, ya que ahora deberemos tener en cuenta si se ha producido una interrupción y se va a ejecutar la rutina RSG o se finaliza el servicio de una interrupción mediante la instrucción **RETI**.

Una vez añadido el banco de registros de sistema, modificado el módulo *control_logic* y el módulo *multi* para que generen los valores de las nuevas señales, modificado el módulo *unidad_control* para que actualice el PC correctamente si la instrucción es un **RETI**; y modificados los módulos *proc* y *datapath* para que lleven las señales de donde se generan hasta donde son necesarias, ya tenemos los principales cambios en el procesador para poder gestionar las interrupciones. Aún no podemos controlar los dispositivos mediante interrupciones ya que aún no se ha mostrado como se conectarán ni controlarán los dispositivos. Pero antes de hacer esto es interesante hacer una comprobación para ver si estas nuevas instrucciones funcionan.

Juegos de prueba

Para probar estos cambios, utilizaremos el esquema tradicional de escribir en memoria y volcar su contenido. Así que haremos un pequeño programa que escriba en los registros de sistema unos valores y los lea a continuación copiándolos en otros registros. Para verificar que todo se ha efectuado correctamente, tomaremos estos resultados y los escribiremos en memoria para luego verificar su contenido.

Para probar las instrucciones **RDS** y **WRS** usaremos el siguiente código como juego de pruebas.

```
movi r0, 32
movi r1, 34
movi r2, -52
wrs  s3, r1
wrs  s6, r2
rds  r5, s3
rds  r6, s6
st   0(r0), r5
st   2(r0), r6
halt
```

Este programa inicializa la dirección base para el *store* final en la línea 1. La prueba la haremos poniendo unos valores cualesquiera en los registros r1 y r2 tal y como muestran las líneas 2 y 3. Uno de los dos valores es negativo para ver como también se escribe y se lee la parte alta del registro. Estos valores seguidamente los escribimos en dos registros de sistema (el S3 y S6) con la instrucción **WRS**, tal y como muestra el programa en las líneas 4 y 5.

Ahora podemos leer estos registros utilizando la instrucción **RDS** y colocar los nuevos valores a unos nuevos registros generales r5 y r6, tal y como muestran las líneas 6 y 7. Finalmente, para verificar que la escritura se ha producido correctamente, escribiremos estos dos registros r5 y r6 a las direcciones 32 y 33 respectivamente. El resultado de la memoria debería ser el siguiente:

```
Mem [21:20] = 0022
Mem [23:22] = ffcc
```

El siguiente juego de prueba comprueba el funcionamiento de las instrucciones **EI** y **DI**.

```
movi r0, 32
rds  r5, s7
st   0(r0), r5
movi r1,255
wrs  s7, r1
rds  r5, s7
st   2(r0), r5
di
rds  r5, s7
st   4(r0), r5
ei
rds  r5, s7
st   6(r0), r5
halt
```

Este programa comprueba cómo se ha inicializado la PSW del procesador (al menos el bit S7<1> debe estar a 0) y realiza un par de cambios de ese bit mediante las instrucciones **EI** y **DI**. El resultado de la memoria debería ser el siguiente:

```
Mem [21:20] = 0000
Mem [23:22] = ffff
Mem [25:24] = fffd
Mem [27:26] = ffff
```

Este último juego de pruebas comprueba el funcionamiento de la instrucción **RETI**.

```
movi  r0, 32
movi  r1, 0xAB
wrs   s0, r1   ; una palabra de estado inventada
movi  r1, 0x10
movhi r1, 0xC0
wrs   s1, r1
reti
halt          ; este halt no deberia ejecutarse si reti funciona
rds   r5, s7
st    0(r0), r5
halt
```

Este programa inicializa el registro S0 con una palabra de estado aleatoria (0xFFAB) y el registro S1 con la dirección de retorno (0xC010) para que sean restaurados por la instrucción **RETI**. El resultado de la memoria debería ser el siguiente:

```
Mem [21:20] = ffab
```

Estos juegos de prueba son extremadamente simples. Sólo comprueban que se modifica lo que ordenan las nuevas instrucciones que hemos implementado pero ninguno ha comprobado que la ejecución de estas instrucciones no haya alterado accidentalmente el contenido de algún otro registro que no debiera. Esta tarea os queda para vosotros.

Gestión de las interrupciones de los dispositivos

Una de las maneras más habituales de conectar y gestionar los dispositivos por interrupción cuando desconocemos el número de dispositivos que vamos a soportar o es muy elevado es hacerlo mediante una conexión "Daisy-chain". Como en nuestro caso sólo vamos a tener 4 dispositivos que generen interrupciones vamos a hacer un dispositivo concreto que controle las interrupciones: el controlador de interrupciones.

Ahora, al trabajar con dispositivos que envían y reciben interrupciones necesitaremos nuevas entradas en el procesador. Utilizaremos la nomenclatura estándar en estos casos. Utilizaremos una señal de entrada procedente del módulo de entrada/salida llamada ***intr*** que responde al *interrupt request* por parte de un dispositivo. Y una vez el procesador ha aceptado esta interrupción, añadiremos una salida al procesador con el nombre de ***inta*** (de *interrupt acknowledge*) para que el dispositivo se entere de que la interrupción ha sido atendida. Por tanto, tendremos una entrada nueva (***intr***) y una salida nueva (***inta***) en el procesador.

Cambios en los dispositivos

Los dispositivos que tendremos conectados por interrupciones son el teclado PS/2, los pulsadores (KEY) y los interruptores (SWITCH). Además, añadiremos un nuevo dispositivo reloj/temporizador (*Interval Timer*) que generará una interrupción cada veinteavo de segundo.

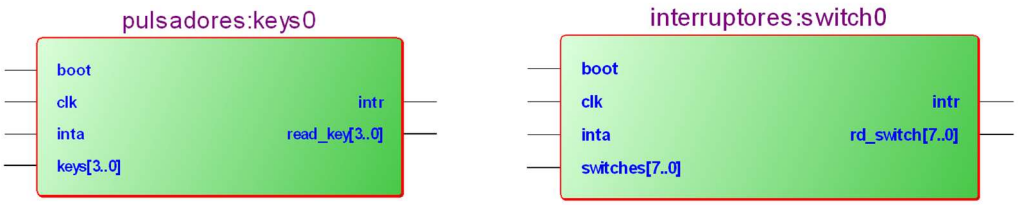
Veamos cual es el funcionamiento de estos dispositivos. Por claridad, puede ser recomendable que creéis un módulo para cada uno de ellos que controle la gestión de las señales de interrupción y sus valores.

Los pulsadores (KEY) y los interruptores (SWITCH) deben generar una interrupción cada vez que haya un cambio. El grupo de pulsadores activan su señal ***intr*** cada vez que se presiona o se deja de presionar un pulsador. El grupo de interruptores activan su señal ***intr*** cada vez que uno de ellos cambie su posición (ON/OFF).

Estos "pseudocontroladores" para pulsadores e interruptores deben tener un registro que memoriza el "estado actual". Cuando el controlador detecta que hay un cambio (por ejemplo, que lo que lee de los pulsadores no coincide con lo que tiene memorizado), memoriza el nuevo estado (y lo pone en el puerto que toque) y genera el ***intr*** correspondiente.

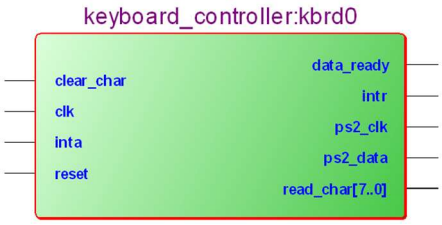
Aunque haya cambios en los pulsadores (o en los interruptores en su caso), si se ha generado el ***intr*** se ignoran los cambios (no se almacenan en el registro) hasta que no haya recibido el ***inta***. Cuando se recibe el ***inta*** se desactiva el ***intr*** hasta que se detecte un nuevo cambio. Si ha habido cambios en el intervalo de tiempo entre el ***intr*** y el ***inta*** será ahora que mediante el procedimiento normal se generará una nueva interrupción. Tenéis que implementar estos controladores para que funcionen como se ha descrito.

Un posible esquema de las señales de entrada y salida de estos dos bloques sería el siguiente:



El controlador de teclado que os hemos suministrado para esta etapa ya está preparado para funcionar por interrupciones. La señal *data_ready* hace las veces de *interrupt* y el *interrupt acknoledge* se hace a través de la señal *clear_char*. Ahora tiene dos señales nuevas (***intr*** e ***inta***). Este controlador sólo produce una interrupción cuando se presiona una tecla (o se mantiene pulsada con sus correspondientes repeticiones) y no cuando se suelta. Esto es debido a que el valor que devuelve es el código ASCII de la tecla pulsada (¿y cuál sería el valor ASCII de no pulsar ninguna tecla?) y no el valor del *scan code* del teclado.

Un posible esquema de las señales de entrada y salida de este bloque sería el siguiente:



Por último, debemos crear un nuevo dispositivo temporizador (*Interval Timer*) que enviará interrupciones cada cierto tiempo de forma regular. Este nuevo dispositivo es vital para que un sistema operativo pueda multiplexar tareas (y sea, pues, multitarea). Además, tal y como se va a desarrollar el sistema, sólo el sistema operativo recibirá estas interrupciones y los programas de usuario no podrán modificar el vector de interrupciones y por lo tanto el sistema funcionará sin admitir "buena voluntad" por parte de los programas de usuario. Por tanto, este dispositivo es el primero en enviar interrupciones y que nos permite desarrollar un sistema operativo multitarea.

Este temporizador debe enviar una interrupción cada 1/20 de segundo, es decir, cada 50 milisegundos. Este tiempo es bastante similar al temporizador del IBM PC 8086 y por ello se ha seleccionado este valor. Si mientras se está "sirviendo" una interrupción del temporizador (se ha generado la señal ***intr*** pero aún no ha llegado la ***inta***) llega el momento de generar otra, esta última se ignora y se pierde.

Un posible esquema de las señales de entrada y salida de este bloque sería el siguiente:

Controlador de interrupciones

Deberemos crear un módulo nuevo para gestionar las interrupciones de los dispositivos. Cada dispositivo tiene sus propias señales ***intr*** e ***inta***. Pero el procesador sólo tiene una pareja de señales ***intr***/***inta*** de modo que este módulo se encargará de priorizar las interrupciones y determinar que dispositivo va a servir su interrupción en primer lugar.

Las interrupciones estarán priorizadas. La del *Interval Timer* será la más prioritaria. Luego vendrá la de los pulsadores, seguida de la de los interruptores y finalmente la menos prioritaria será la del teclado PS/2.

Puesto que al procesador sólo le llega una señal ***intr***, necesita algún mecanismo para averiguar que dispositivo ha generado esta interrupción. Tal y como indica la documentación del procesador, disponemos de una instrucción llamada **`GETIID`** para identificar que dispositivo ha generado la interrupción. Esta instrucción es la que activa, en su etapa de ejecución, la señal ***inta*** que genera el procesador.

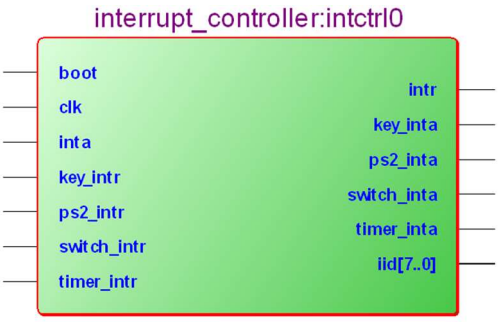
Cuando el controlador de interrupciones reciba la señal ***inta*** deberá mirar de todas las interrupciones de los dispositivos que tenga activadas cual es la más prioritaria. Devolverá el identificador de interrupción (**iid**) que se va a atender y activará la señal ***inta*** del dispositivo correspondiente. Los identificadores de interrupción serán los siguientes (debemos estandarizarlos para que cuando implementemos el Sistema Operativo no nos volvamos locos):

iid	Dispositivo
0	Temporizador (*Interval Timer*)
1	Pulsadores (*KEYs*)
2	Interruptores (*SWITCHs*)
3	Controlador Teclado PS/2

Depende de cómo implementéis el controlador de interrupciones deberéis ser cuidadosos en qué momento actualizáis el bus **iid** para que no lo hagáis justo en el momento que lo pueda estar leyendo el procesador. Puede ser útil sincronizar esta señal con el reloj del procesador para evitar que se realice alguna modificación mientras el procesador lo está consultando

El ***inta*** del procesador llega cuando se ejecuta la instrucción **`GETIID`** y dura un ciclo de reloj del procesador. El tiempo que deberá estar activa para cada uno de vuestros dispositivos deberá ser la adecuada en función de cómo los hayáis implementado.

Un posible esquema de las señales de entrada y salida de este bloque sería el siguiente:



Controlador de entrada/salida

El controlador de interrupciones lo integraremos en el módulo de entrada/salida. Por lo tanto deberemos modificar este módulo para integrar todo el sistema de interrupciones. Por un lado instanciará a todos los dispositivos para leer su estado y mantener el valor de los puertos correctamente para que puedan seguir siendo usados por encuesta.

En este módulo las modificaciones son pocas, simplemente las señales de interrupción de los dispositivos se envían al controlador de interrupciones y este ya se encargará de gestionarlas. Cuando se ejecute la instrucción **`GETIID`**, el valor del identificador del dispositivo que se atiende su interrupción (**iid**) lo enviaremos al procesador a través del bus de datos **rd_io** (si no le habéis cambiado el nombre al bus) del controlador de entrada/salida para no añadir un bus nuevo. Por tanto, el controlador de entrada/salida debe saber cuándo se está ejecutando una instrucción **`GETIID`**. Para ello podemos añadir una nueva señal procedente del procesador que le indique al controlador de entrada/salida que se está ejecutando la instrucción **`GETIID`**, podemos usar la señal ***inta*** o podemos evitarnos la señal prefijando un número de puerto de entrada/salida, de los que no están siendo usados, para indicar que se está ejecutando la instrucción **`GETIID`**. A vuestra elección.

Control de las interrupciones

Para controlar las interrupciones deberemos realizar diversas modificaciones en el procesador. Deberemos adaptar los módulos *unidad_control, control_logic* y *multi* para que reciban la señal ***intr***, la traten adecuadamente y generen la señal de salida ***inta***. Además deberemos añadir el control y la descodificación de la nueva instrucción **`GETIID`**.

También deberemos modificar el banco de registros para que cada vez que se vaya a tratar una interrupción realice todas las tareas descritas al principio del documento como: copiar la palabra de estado PSW, hacer la copia de seguridad del PC

de retorno, indica que el evento producido es una interrupción, cambiar la PSW inhibiendo las interrupciones y actualizar el PC con la dirección de la primera instrucción de la rutina de servicio a las interrupciones (la RSG).

Por todo ello aparecerán nuevas señales de control para el banco de registros, el datapah, etc., que se deberán generar en la parte de control de procesador.

Banco de registros

Cuando se vaya a atender una interrupción, el banco de registros debe realizar las siguientes tareas:

- S0 ← S7 (se copia la palabra de estado PSW)
- S1 ← PCup (hace un backup del PC de retorno)
- S2 ← 0x000F (indica que el evento producido es una interrupción)
- PC ← S5 (rutina de sistema a la que se llama)
- S7<1> ← 0 (se cambia la PSW inhibiendo las interrupciones)

Por lo tanto hay que realizar las modificaciones necesarias para que el Banco de registros realice estas tareas en el ciclo SYSTEM. Necesitará una señal nueva procedente de la unidad de control que le indique que se está preparando para atender una interrupción.

También necesitará un bus de entrada con el *PCup* proveniente de la unidad de control y un bus de salida con el valor del registro S5 para poder actualizar el PC correctamente. Se podría evitar añadir dos buses nuevos usando los que ya hay en el Banco de registros. Para enviar el *PCup* desde la unidad de control hasta el Banco de registros se podría usar el puerto D del banco de registros como ya hace la instrucción JAL y para enviar el contenido del registro S5 hasta el *Program Counter* en la unidad de control podríamos usar el puerto A del banco de registros como ya hacen algunas de las instrucciones de salto. Ello implica ciertos cambios en las señales de control para estos buses.

Para que la unidad de control sepa si debe atender una interrupción o no debe tener conocimiento de si las interrupciones están habilitadas o deshabilitadas. Esta información esta almacenada en el bit S7<1> del banco de registros. Por lo tanto el banco de registros deberá tener un bit de salida para enviar constantemente esa información a la unidad de control.

Unidad de control

Como ya se ha comentado al principio del documento, cuando se produce una interrupción esta se tratará al final de la ejecución de la instrucción en curso. Por tanto deberemos modificar el módulo *multi* para que una vez se está finalizando la ejecución del ciclo de DEWM, si hay una interrupción pendiente (***intr***=1) y las interrupciones están habilitadas, en vez de ir al ciclo de Fetch de la siguiente instrucción, vaya al ciclo que hemos llamado SYSTEM.

Es en este módulo *multi*, que cuando se esté en el ciclo SYSTEM hay que generar la señal para indicarle al banco de registros que se va a atender una interrupción. Además las señales con control de los buses y de los diversos multiplexores deberán tener el valor adecuado cuando se esté ejecutando ese ciclo.

Este grupo de módulos (*unidad_control*, *control_logic* y *multi*) se deben encargar de generar algunas señales nuevas que dependerán de como los hayáis implementado. Probablemente serán similares a estas:

- Una señal que le indique al módulo *multi* que debe atender una interrupción (ejecutar el ciclo SYSTEM). Esta señal deberá tener en cuenta si hay una petición de interrupción y además las interrupciones están permitidas.
- Una señal procedente del módulo *multi* que indique si se está ejecutando el ciclo SYSTEM para que la unidad de control sepa si en el PC debe cargar el valor siguiendo la ejecución del programa o el valor procedente del Banco de registros con la dirección de la rutina de servicio de interrupción (registro S5).
- Una señal para indicar al banco de registros que se está atendiendo a una interrupción y este realice sus tareas.

- Si habéis usado buses nuevos todas sus señales de control.

También deberéis modificar los valores de algunas señales para adecuar su comportamiento a las nuevas necesidades. Probablemente deberéis modificar las siguientes señales:

- Las que controlan los buses que se usen para traer el registro S5 del banco de registros al PC. Dependerán de si habéis usado un bus propio o el puerto A.
- Las que controlan los buses que se usen para llevar el *PCup* hasta banco de registros. También dependerán de si habéis usado un bus propio o el bus que llega al puerto D.
- La de selección del valor (*tknbr*) y/o la de carga del PC (*ldpc*) si aún usáis esta señal.

Además, estos tres módulos deberán encargarse de interpretar la nueva instrucción **`GETIID`** y generar los valores de todas las señales necesarias. La instrucción **`GETIID`** activa la señal ***inta*** para notificar al dispositivo que generó la petición que ésta va a ser atendida; y obtiene del bus de datos el identificador de interrupción enviado por el dispositivo, almacenándolo en el registro Rd. En la siguiente tabla podéis ver la codificación de la instrucción **`GETIID`**.

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0		
1 1 1 1	Rd	0 0 0	1 0 1	0 0 0	**GET I**nterrupt **ID**entification	GETIID

Esta instrucción devuelve el identificador del dispositivo más prioritario que ha generado la interrupción que se está atendiendo y se almacena en un registro. Este identificador (**iid**) se encuentra en el módulo controlador de interrupciones que está integrado en el módulo de entrada/salida. Una sencilla forma de implementar esta instrucción sería que el **iid** del controlador de interrupciones salga por el bus *rd_io* del controlador de entrada/salida siempre que ***inta***='1'. Así la tarea de grabar el contenido del **iid** en el banco de registros será muy similar a la instrucción **`IN`** ya implementada.

Juegos de prueba

Para probar todos estos cambios os hemos dejado un par de juegos de prueba. El primero, llamado "*5-test_interrupt.s*", simplemente inicializa los registros necesarios, activa las interrupciones y hace un bucle infinito a la espera de que le lleguen interrupciones. Cada vez que le llega una interrupción muestra el identificador de la interrupción que se está atendiendo por los visores siete segmentos de la placa. Para este juego de pruebas es recomendable desactivar (o ralentizar mucho) las interrupciones generadas por el *Interval Timer* si no veréis siempre el valor 0000 por los visores (el identificador de interrupción de reloj). El segundo juego de pruebas, llamado "*6.test2_interrupt.s*", muestra por la pantalla un reloj y diversa información como la tecla pulsada y sus repeticiones, el estado de los pulsadores e interruptores, etc.

Anexo A: Interrupciones y Excepciones

Interrupciones

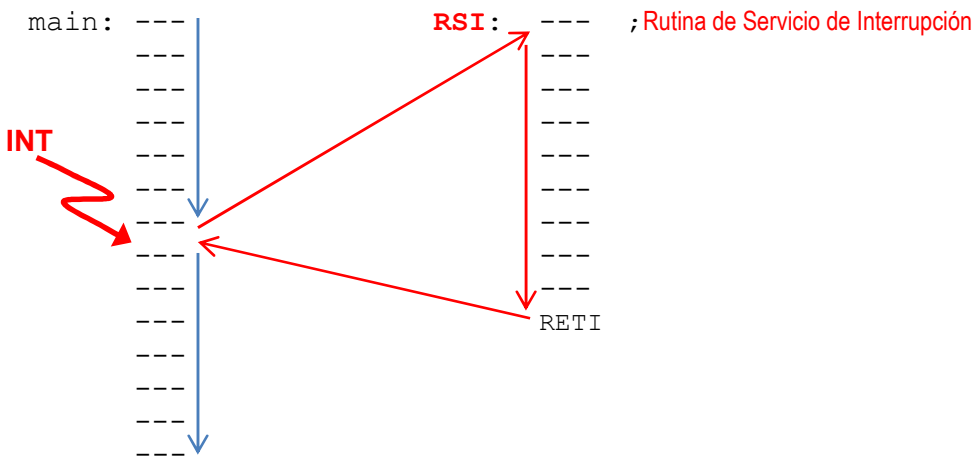
Podemos comunicar los dispositivos con el procesador de diversas maneras. Las más habituales son:

- Por sincronización: el procesador y el dispositivo se ponen de acuerdo en el momento de la transferencia.
- Por encuesta: el procesador pregunta al dispositivo repetidamente si está preparado (bucle de espera activa).
- Por interrupciones: el dispositivo avisa al procesador cuando está preparado (el procesador puede estar ejecutando otro proceso)

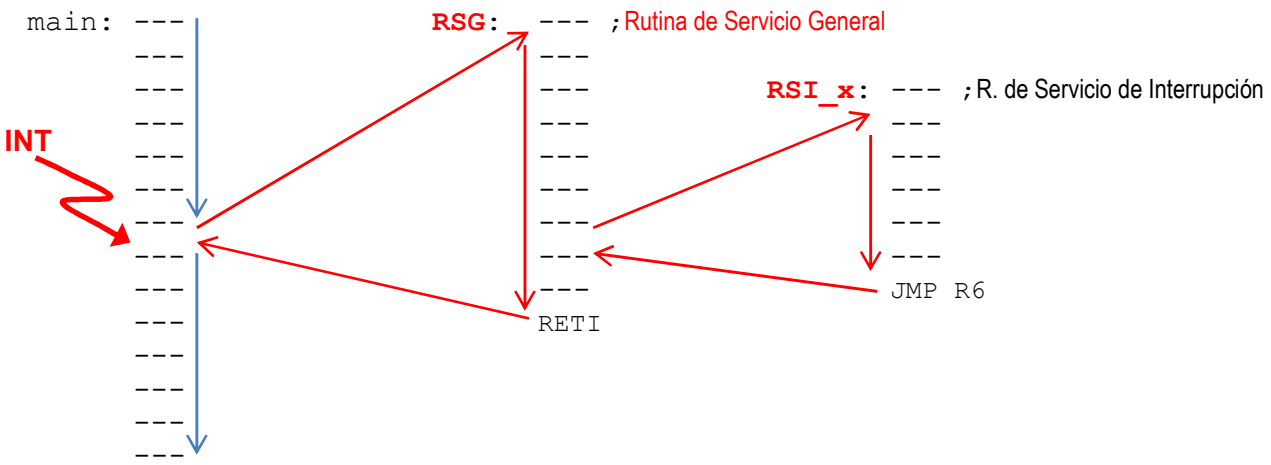
En el caso de las interrupciones, el procesador dispone de una única señal de interrupción y será mediante un mecanismo de arbitraje el que determinará que dispositivo es el que interrumpe en cada momento al procesador.

Cuando el procesador recibe una interrupción se ejecuta una rutina de servicio de interrupción (RSI) que está en una posición de memoria preestablecida.

En la ruptura de secuenciamiento el programa interrumpido no debe enterarse. Por tanto después de ejecutar la RSI todos los registros deben quedar exactamente como estaban antes de la ruptura.



Como puede haber varios dispositivos que compartan la señal de entrada de interrupción del procesador, cuando se genera una interrupción en vez de ejecutarse la rutina RSI, primero se ejecuta una rutina de servicio general (RSG) que se encarga de averiguar que dispositivo es el que ha hecho la interrupción y luego llamar a la RSI específica para ese dispositivo.



El mecanismo de interrupción se compone de diversas fases:

1. Generación de la petición
2. Detección de la petición
3. Aceptación de la petición
4. Salvar el contexto y saltar a la RSG
5. Salvar el estado
6. Identificar el dispositivo
7. Eliminar la petición de interrupción
8. Ejecutar la RSI específica
9. Restaurar el estado
10. Retornar al programa que fue interrumpido

Fase 1: el procesador (programador) debe permitir o no la generación de interrupciones por parte del controlador. En algunos controladores la generación o inhibición de interrupciones se puede hacer individualmente para cada dispositivo. En el caso del SISA simplemente el procesador ignora o no la petición de interrupción procedente del controlador de interrupciones en función del bit de la palabra de estado del computador (PSW).

Fase 2: No se atiende la interrupción hasta que la instrucción en curso ha terminado su ejecución (estado seguro).

Fase 3: Se comprueba es estado del permiso de las interrupciones en la palabra de estado (PSW)

Fase 4: Las tareas de esta fase son específicas de cada procesador. En el caso del SISA se hacen todas por hardware y son las siguientes:

- S0 ← S7 (salva palabra de estado PSW)
- S1 ← PCup (salva la dirección de retorno)
- S2 ← 15 (carga el código de evento: Interrupción)
- S7<1> ← 0 (inhibe las interrupciones)
- PC ← S5 (salta a ejecutar la rutina RSG)

Fase 5: La RSG salva el estado en la pila para que el programa que fue interrumpido no se entere. ¿Qué registros salvamos en la pila? Para ello debemos decidir si permitimos interrupciones/excepciones anidadas. En el SISA las excepciones están activadas por defecto y no las podemos inhibir. Las interrupciones se podrían permitir (mediante la ejecución de la instrucción EI) dentro de la RSG pero no es nada aconsejable.

Con interrupciones/excepciones anidadas necesitamos un diseño muy cuidadoso de la RSG, pero en cualquier caso las rutinas RSG, RSE y RSI deberían estar libres de excepciones. Pero los accidentes ocurren y las excepciones se podrían producir (por ejemplo un fallo de página en el TLB).

Por ello, a parte de los registros generales, necesitamos que la RSG salve en la pila los siguientes registros:

- S0: la palabra de estado que provocó la excepción/interrupción que se está tratando. Si no lo hacemos la macharemos con la llamada a RSG que se va a realizar.
- S1: La dirección de retorno de la instrucción donde se detectó excepción/interrupción. Si no lo hacemos, lo macharemos y no sabríamos donde volver.
- S3: Dirección efectiva de acceso no alineado. Es posible que provoquemos un acceso no alineado dentro de la RSE o RSI. Así podremos contemplar más de un acceso no alineado consecutivo.

Así, un posible fragmento de código SISA para salvar y restaurar el estado en la RSG podría ser el siguiente (usando la macro *$push* para acceder a la pila a salvar los registros):

```
RSG:  $push R0, R1, R2, R3, R4, R5, R6 ;Salvar el estado
      rds   R1, S0
      rds   R2, S1
      rds   R3, S3
      $push R1, R2, R3
      ...
```

Fase 6: Una vez dentro de la RSG debemos identificar el dispositivo concreto que ha generado la interrupción. El problema es que varios dispositivos pueden generar interrupciones y sólo hay una señal INT de entrada. En el SISA, para identificar al dispositivo que se va a atender la interrupción el procesador ejecuta la instrucción especial **`GETIID`** (GET Interrupt IDentifier). Esta instrucción genera la señal de salida INTA (aceptación de la interrupción) que cuando llega al dispositivo concreto este devuelve por un bus de datos el identificador del dispositivo.

El identificador que obtenemos del dispositivo que ha generado la interrupción es un número entero. Normalmente se usa este identificador para indexar un vector que contiene todas las direcciones de inicio de todas las RSI específicas para cada dispositivo. Este vector, conocido como vector de interrupciones, está en memoria y primero se accede al vector para obtener la dirección de la RSI que se va a ejecutar y luego ya se salta a la RSI para ejecutar el código.

Básicamente el fragmento de código para SISA que realiza esta tarea seria parecido al siguiente:

```
      ...
__interrupcion:
     GETIID R1                        ;(R1←id) obtenemos el identificador de dispositivo
     add    R1, R1, R1                ;lo multiplicamos por 2 ya que las @ almacenadas en el vector son words
     movi   R2, lo(interrupts_vector)
     movhi  R2, hi(interrupts_vector) ;cargamos la dirección de inicio del vector de interrupciones
     add    R2, R2, R1                ;le sumamos a la dirección de inicio el desplazamiento correspondiente al id del dispositivo
     ld     R2, 0(R2)                 ;obtenemos del vector de interrupciones la dirección de la RSI específica
     jal    R6, R2                    ;saltamos a la rutina RSI
      ...
```

Para evitar contratiempos, normalmente se inicializa el vector de interrupciones completamente, incluso para aquellos identificadores de dispositivos que no existen, con una rutina RSI por defecto que no realiza ninguna acción. En el siguiente ejemplo podéis ver como se definiría el vector de interrupciones en lenguaje ensamblador SISA (suponiendo que el identificador de dispositivo máximo es 3):

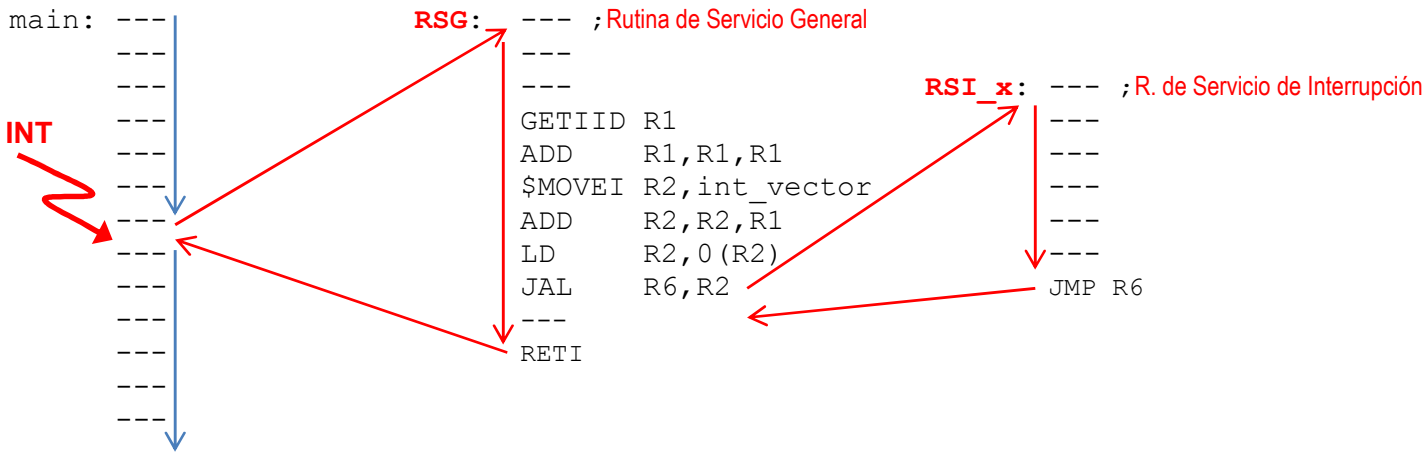
```
interrupts_vector:
    .word RSI_default_resume ;  0 Interval Timer
    .word RSI_default_resume ;  1 Pulsadores (KEY)
    .word RSI_default_resume ;  2 Interruptores (SWITCH)
    .word RSI_default_resume ;  3 Teclado PS/2

RSI_default_resume: JMP R6
```

El contenido del vector de interrupciones se puede cambiar en cualquier momento reprogramando las direcciones de inicio de las RSI.

```
main:
      ...
      movi   R0, lo(interrupts_vector)
      movhi  R0, hi(interrupts_vector)
      movi   R1, lo(mi_RSI_de_teclado)
      movhi  R1, hi(mi_RSI_de_teclado)
      ST     3*2(R0), R1              ;3 es el id de teclado PS/2
```

Fase 7: En el SISA, la eliminación de la petición de interrupción se hace por hardware y el controlador de interrupciones se encarga de ello.

Fase 8: Una vez hemos obtenido del vector de interrupciones la dirección de inicio de la RSI específica, simplemente con la instrucción **JAL** la ejecutamos. La RSI, como una subrutina cualquiera, no salva el estado (ya lo ha hecho la RSG). Al finalizar la ejecución de esta RSI debemos volver a la RSG mediante la ejecución de una instrucción **JMP**.



Fase 9: La RSG restaura el estado de la pila para que el programa que fue interrumpido no se entere. Así, un posible fragmento de código SISA para salvar y restaurar el estado en la RSG podría ser el siguiente (usando la macro *$pop* para acceder a la pila a restaurar los registros):

```
        ...
__finRSG:    ;Restaurar el estado
        $pop  R3, R2, R1
        wrs   S3, R3
        wrs   S1, R2
        wrs   S0, R1
        $pop  R6, R5, R4, R3, R2, R1, R0
        RETI
```

Fase 10: En esta última fase simplemente la RSG debe retornar al programa que fue interrumpido. Para ello simplemente debe ejecutar la instrucción **RETI**. Esta instrucción se encarga de restaurar la palabra de estado (PSW) y el *Program Counter* (PC)

La RSG completa en SISA quedaría más o menos de la siguiente forma

```
RSG: ; Salvar el estado
     $push  R0, R1, R2, R3, R4, R5, R6
     rds    R1, S0
     rds    R2, S1
     rds    R3, S3
     $push  R1, R2, R3
__interrupcion:
     getiid R1
     add    R1, R1, R1
     movi   R2, lo(interrupts_vector)
     movhi  R2, hi(interrupts_vector)
     add    R2, R2, R1
     ld     R2, 0(R2)
     jal    R6, R2
__finRSG:    ;Restaurar el estado
     $pop  R3, R2, R1
     wrs   S3, R3
     wrs   S1, R2
     wrs   S0, R1
     $pop  R6, R5, R4, R3, R2, R1, R0
     reti
```

Excepciones

Aparte de las interrupciones, en el procesador SISA también pueden producirse excepciones. Las excepciones son un suceso no esperado ("excepcional") que se produce internamente en el procesador.

Los sucesos que provocan una excepción en nuestro procesador son los siguientes:

- Ejecución de una instrucción ilegal
- Acceso no alineado a memoria
- Desbordamiento (overflow) en una operación de coma flotante
- División por cero en coma flotante
- División por cero en enteros o naturales
- Acceso a memoria protegida
- Instrucción protegida
- Instrucción CALLS

Las excepciones son tratadas por el procesador casi de la misma forma que las interrupciones. Cuando se produce una excepción se realiza una ruptura del secuenciamiento al igual que una interrupción y se llama a la rutina RSG. La rutina RSG detectará si se trata de una interrupción o de una excepción y en este último caso de que excepción se trata. Una vez identificada se llama a la RSE (Rutina de Servicio de Excepción) de la excepción que corresponda con el mismo modo de funcionamiento que las interrupciones.

Las excepciones pueden ser enmascarables o no. Las enmascarables son aquellas que el programador puede impedir que se produzcan (o puede hacer que el procesador las ignore). Según la documentación del SISA la única excepción enmascarable es la excepción de overflow para operaciones de coma flotante. Para indicar si están enmascaradas o no el procesador dispone de un bit en la palabra de estado (PSW<2>). El resto de excepciones en el SISA no pueden ser enmascaradas.

Para que la RSG pueda identificar si se trata de una interrupción o de una excepción, y en este caso de que excepción se trata, debe consultar el valor del registro S2. Ahora el registro S2 no sólo contiene el valor 15 si no que contiene el identificador de excepción. Si su valor es 15 se trata de una interrupción y si su valor es menor entonces es una excepción y este valor indica la excepción concreta de que se trata.

Al igual que las interrupciones, tendremos un vector de excepciones y un comportamiento por defecto. Los comportamientos por defecto habituales pueden ser no hacer nada o parar el procesador. Igualmente que con el vector de interrupciones, el contenido del vector de interrupciones se puede cambiar en cualquier momento reprogramando las direcciones de inicio de las RSE.

En el siguiente ejemplo podéis ver como se definiría el vector de excepciones en lenguaje ensamblador SISA (suponiendo que solo existen 4 excepciones):

```
exceptions_vector:
    .word RSE_default_halt   ; 0 Instrucción ilegal
    .word RSE_default_halt   ; 1 Acceso a memoria no alineado
    .word RSE_default_resume ; 2 Overflow en coma flotante
    .word RSE_default_halt   ; 3 División por cero

RSE_default_halt:   HALT
RSE_default_resume: JMP R6
```

Las fases de ejecución son exactamente las mismas que las de las interrupciones exceptuando la primera que es la generación de la excepción. Y la fase 6 donde hay que detectar de qué excepción se trata.

La RSG completa en SISA, con interrupciones y excepciones, quedaría más o menos de la siguiente forma:

```
RSG: ; Salvar el estado
     $push  R0, R1, R2, R3, R4, R5, R6
     rds    R1, S0
     rds    R2, S1
     rds    R3, S3
     $push  R1, R2, R3
     rds    R1, S2                    ;consultamos el contenido de S2
     movi   R2, 15
     cmplt  R3, R1, R2                ;si es menor a 15 es una excepción
     bz     R3, __interrupcion        ;saltamos a las interrupciones si S2 es igual a 15
__excepcion:
     movi   R2, lo(exceptions_vector)
     movhi  R2, hi(exceptions_vector)
     add    R1, R1, R1                ;R1 contiene el identificador de excepción
     add    R2, R2, R1
     ld     R2, 0(R2)
     jal    R6, R2
     bnz    R3, __finRSG
__interrupcion:
     getiid R1
     add    R1, R1, R1
     movi   R2, lo(interrupts_vector)
     movhi  R2, hi(interrupts_vector)
     add    R2, R2, R1
     ld     R2, 0(R2)
     jal    R6, R2
__finRSG:    ;Restaurar el estado
     $pop  R3, R2, R1
     wrs   S3, R3
     wrs   S1, R2
     wrs   S0, R1
     $pop  R6, R5, R4, R3, R2, R1, R0
     reti
```