Departamento de Arquitectura Instituto de Computación

Universidad de la República

Montevideo - Uruguay

Notas de Teórico

Interrupciones

Arquitectura de Computadoras

(Versión 4.3 - 2012)

16 INTERRUPCIONES

16.1 Introducción

En este capítulo veremos el último de los componentes esenciales de la arquitectura von Neuman: el manejo de interrupciones.

Este concepto es habitualmente difícil de entender en los cursos básicos de computación porque se aparta de lo que “intuitivamente” pensamos de la ejecución de un programa: siempre pensamos que la próxima instrucción (o sentencia) que se ejecuta de un programa es la que está escrita a continuación en el código, o siendo más precisos (para contemplar los saltos) la que está a continuación en la lógica de ejecución del programa.

Esto, que parece hasta de perogrullo, sin embargo no es cierto debido justamente al mecanismo de interrupciones que se utiliza en todos los sistemas que se basan en los conceptos de programa almacenado.

Primeramente veremos una justificación de porqué es conveniente utilizar un mecanismo de este tipo, para luego presentar una definición de lo que es una interrupción.

16.2 Justificación del Manejo de E/S por Interrupciones

16.2.1 Técnica de “Poleo”

Imaginemos la siguiente situación: tenemos un computador al que se le conecta una “terminal tonta” mediante un puerto de comunicaciones serial. Una terminal de este tipo consiste en una pantalla capaz de desplegar caracteres y un teclado. En la pantalla se despliegan los caracteres recibidos por la conexión serial, mientras que las teclas digitadas en el teclado son enviadas como caracteres por dicha conexión.

Complementemos el cuadro diciendo que en el computador se ejecuta un programa de edición básica de texto (que va mostrando en la pantalla de la terminal el texto que va digitando el usuario en el teclado de la misma).

Una forma de diseño de este programa implica que el programa esté en un “loop” (con una estructura tipo while) esperando por la aparición de un carácter en el puerto serie del computador al que está conectado el terminal. Cuando éste aparece lo procesa para desplegarlo en la pantalla, mueve el cursor un lugar hacia delante (eventualmente cambiando de línea) y vuelve al bucle de espera por un nuevo carácter.

Supongamos por un momento que se necesitan 1000 instrucciones de máquina para procesar un carácter recibido en el puerto serie. Si tenemos en cuenta que un hábil digitador es capaz de teclear del orden de 60 palabras por minuto y considerando que una palabra promedio tiene del orden de 5 letras, concluimos que la tasa de ingreso de caracteres es:

60 palabras / minuto * 5 letras / palabra = 300 letras / minuto = 5 letras / segundo

Esto significa que el tiempo entre que el usuario digita una letra y la siguiente es del orden de 0,2 segundos (200 ms).

Supongamos que el procesador de este computador tiene una capacidad de proceso de 1 MIPS (Millón de Instrucciones Por Segundo). Esto significa que en 200 ms puede ejecutar 200.000 instrucciones. De ellas requiere 1000 para procesar la tecla, mientras que las restantes 199.000 se desperdician en el “loop” de espera.

Una idea que surge casi enseguida es: ¿porqué no utilizar esas “instrucciones muertas” para atender los requerimientos de otros usuarios?. Podríamos, por ejemplo, conectar múltiples terminales “tontas” con un programa de edición de texto que fuera consultando los distintos puertos de comunicación serie de forma de detectar los caracteres que vayan llegando y procesando los mismos.

Si bien esto es posible surge el inconveniente que el programa de edición de texto debería ser adaptado (modificado) cada vez que se agrega o se quita una conexión de terminal y aún en el caso que pensáramos en un mecanismo mas dinámico que la re-compilación (ej: un array dinámico de direcciones de puertos serie "activos" parametrizable en tiempo de ejecución) tendríamos la restricción que el programa cumpliría la misma función para todos los usuarios conectados.

Obviamente que esta restricción también podríamos levantarla haciendo programas "combinados" que atendieran múltiples tipos de requerimientos (editores de texto, compiladores, planillas de cálculo, procesadores de imágenes, etc, etc) pero es evidente que las combinaciones serían muchas y el código sería cada vez mas complicado para poder "polear" distintos puertos de comunicación serie e invocar los módulos apropiados en cada caso.

La solución que se encontró a este problema es, en definitiva, la existencia de un programa (denominado Sistema Operativo) que se encarga del manejo de los controladores de comunicación y le pasa los caracteres a los programas apropiados. Y para no desperdiciar instrucciones en bucles ociosos se recurre a la técnica que el controlador de E/S que maneja la comunicación tenga la capacidad de "avisar" cuando dispone de un carácter para ser procesado. El mecanismo de aviso es justamente a través de las **interrupciones**.

16.2.2 Sincronización con la E/S

Otra razón por la cual se utiliza este mecanismo tiene que ver con la eficiencia del acceso a los dispositivos de entrada/salida y, en particular, a los dispositivos de almacenamiento masivo.

Una gran proporción de los programas utiliza las unidades de disco para almacenar la información que requieren procesar y para guardar el resultado de ese proceso en los denominados **sistemas de archivos**. Estos dispositivos involucran habitualmente elementos mecánicos en su construcción (ej: discos que giran y que son leídos por una cabeza que se desplaza movida por un "actuador"). Los "tiempos de respuesta" de los dispositivos mecánicos son siempre varios órdenes de magnitud más lentos que los electrónicos. Por ejemplo un disco moderno tiene un muy buen tiempo de acceso (se denomina así al tiempo promedio para acceder al bloque de información grabado en alguna de sus pistas) respecto a los diseños originales, estando actualmente por debajo de los 10 ms. Sin embargo ese tiempo es "una eternidad" si lo comparamos con el tiempo de acceso de una memoria electrónica (entre 10 ns y 100 ns).

Esta situación lleva al hecho que si un programa requiere un dato de un disco debe esperar del orden de 10 ms para poder disponer de él. Aún si tomamos el ejemplo del procesador de 1 MIPS (en la práctica hoy se utilizan procesadores capaces de ejecutar cientos de MIPS) vemos que durante ese tiempo de espera el procesador podría haber ejecutado del orden de 10.000 instrucciones. Por esto realizar la sincronización de la entrada/salida mediante la técnica de enviar el comando correspondiente y quedarse a esperar la respuesta resulta en algo sumamente ineficiente.

Al igual que en el caso de las terminales tontas, aquí conviene utilizar el procesador para ejecutar otros programas mientras los controladores de E/S, con su inteligencia propia, realizan la operación requerida y cuando terminan "avisan" que los datos ya están disponibles en algún lugar accesible a tiempos "electrónicos" (ej: un "buffer" en el controlador de E/S ó incluso en la propia memoria del sistema).

Al igual que en el caso anterior la solución pasa por un programa especial (el Sistema Operativo) que se encargue de la E/S y por el mecanismo de interrupciones que habilite la posibilidad de avisar de la culminación de la operación de E/S solicitada.

16.3 Definición

Una definición de interrupción puede ser:

> *Una interrupción consiste en un mecanismo que provoca la alteración del orden lógico de ejecución de instrucciones como respuesta a un evento externo, generado por el hardware de entrada/salida en forma asincrónica al programa que está siendo ejecutado y fuera de su control.*

En forma alternativa se puede decir que:

> *Una interrupción consiste en un mecanismo que le permite al hardware la invocación de una rutina fuera del control del programa que está siendo ejecutado.*

Analicemos el contenido de la primer definición que hemos dado. Por un lado decimos que se trata de un mecanismo, cuyos detalles veremos mas adelante, que provoca la *alteración del orden lógico de ejecución* de instrucciones. Esto significa que cuando ocurre una interrupción la próxima instrucción a ser ejecutada no es la que corresponde a la secuencia lógica de instrucciones del programa que se está ejecutando, que vendría a ser la apuntada por el IP (Instruction Pointer), sino que dicha secuencia se modifica y se pasa a ejecutar otra instrucción (concretamente la primera de la **rutina de servicio de la interrupción** correspondiente al pedido atendido).

Por otro lado nos referimos a que es en respuesta a un *evento externo*, que está *generado por el hardware de entrada/salida*, es decir que no es el programa que lo genera (al menos no en forma directa), siendo provocado por el hardware, en particular el asociado a los dispositivos de entrada/salida, más concretamente: los controladores de E/S.

Y finalmente se menciona que es *asincrónico y fuera de su control*, lo que significa que el programa que está siendo ejecutado no tiene control sobre el momento en que este mecanismo se dispara, ni sobre la propia ocurrencia del fenómeno. Es decir que, en particular, no hay sincronismo entre la ejecución del programa y la, eventual, invocación a esa rutina "especial" disparada por la interrupción.

16.4 Pedido de Interrupción

El mecanismo de interrupción comienza con el pedido de interrupción ("interupt request") generado por un controlador de entrada/salida.

Este pedido se genera a raíz de alguna condición detectada por el controlador (ej: dispone de un dato para ser leído por la CPU, terminó de ejecutar la lectura del sector de disco solicitada, hay una condición de error en el byte recibido por la línea de comunicaciones, etc, etc).

Las condiciones que generan el pedido varían de acuerdo al tipo de controlador que se trate y de acuerdo a como esté configurado el mismo. Hay controladores capaces de solicitar interrupciones ante determinadas y diferentes condiciones de error, pero que pueden ser programados para no hacerlo a través de bits que habilitan/deshablitan la posibilidad que una situación particular pueda generar, o no, una interrupción.

Los controladores de E/S disponen de una señal de **salida** (habitualmente denominada INT ó INTR ó IRQ) que toma el valor "1" cuando el controlador interrumpe. Normalmente esa salida es el reflejo hardware de un bit del registro de *ESTADO*. Es decir que el nivel actual (0 ó 1) de la señal INT del controlador de E/S puede leerse en un bit de dicho registro. Este bit de "pedido de interrupción pendiente" y su correspondiente reflejo en el hardware permanece en nivel alto hasta que la rutina de atención de la interrupción realice alguna operación sobre los registros del controlador de E/S que satisfaga el pedido, con lo cual el controlador procederá a pasar ese bit (y su salida INT) a 0. Cuál es esa acción dependerá del controlador y, posiblemente, de la condición que provocó el pedido. Algunos pedidos de interrupción se satisfacen con la lectura del registro de estado, otros requieren de la lectura de algún registro adicional, otras requieren de la escritura de un registro, etc.

En las CPUs también existe esta señal, que recibe denominaciones similares (INT, IRQ, etc), pero que es de **entrada**. Esta entrada es la que el hardware de la CPU consulta para determinar si hay algún pedido de interrupción por parte de algún controlador de E/S.

Luego lo veremos en mayor detalle pero adelantemos que la CPU chequea la existencia de un pedido de interrupción al finalizar la ejecución de cada instrucción. Si detecta una solicitud, procede a desencadenar el mecanismo que desemboca en la ejecución de la rutina de servicio de la interrupción, luego de la cual se pasa a ejecutar la próxima instrucción del programa interrumpido (la que se hubiera ejecutado si no hubiera existido la interrupción), siempre que no existe un nuevo pedido de interrupción, en cuyo caso se desencadenará un nuevo proceso de invocación a la rutina de servicio que corresponda.

De acuerdo a la forma que la CPU detecta que hay un pedido de interrupción se distinguen dos casos: detección por nivel y detección por flanco.

16.4.1 Detección por Nivel

En este caso la CPU reconocerá que hay un pedido de interrupción pendiente mientras su entrada INT esté en el **nivel** lógico alto ("1").

Esto significa que toda vez que la CPU detecte esa entrada en alto, desencadenará la invocación a la rutina de atención, incluso cuando haya terminado recién de ejecutar una rutina de interrupción.

16.4.2 Detección por Flanco

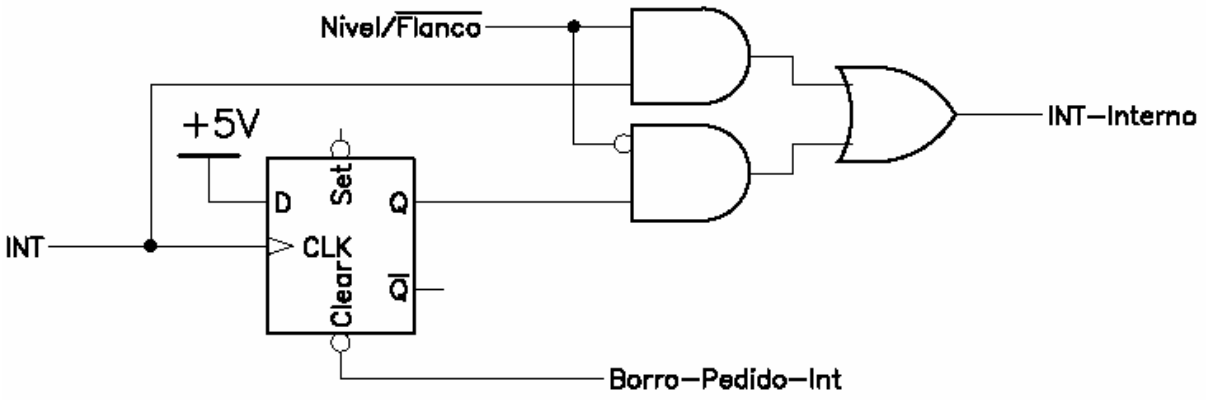
En este caso lo que interesa a los efectos de determinar que existe un requerimiento de interrupción es la existencia de un cambio de nivel bajo ("0") a alto ("1"), es decir un **flanco**, en la entrada INT.

Es decir que no importa el valor absoluto actual de la entrada INT, lo que se tiene en cuenta es si existió un cambio de 0 a 1. Si hubo un flanco y aún no ha sido invocada la rutina de atención, la CPU lo hace y da por cumplido el pedido.

Luego veremos que esto tiene sus implicancias en el diseño de las rutinas de servicio a las interrupciones.

16.4.3 Circuito Equivalente de la entrada INT

La entrada INT tiene el siguiente circuito lógico equivalente:



La entrada INT a la CPU alimenta la entrada de reloj de un flip-flop D cuya entrada D está en nivel lógico alto ("1"). Esto hace que en cada flanco ascendente de INT el flip-flop adopta el valor "1" en su salida Q. La salida del FF y la propia entrada INT se presentan a un multiplexor cuya entrada de control selecciona si la salida INT-Interno corresponde directamente a la señal INT (cuando es por nivel) ó a la señal Q (cuando es por flanco). Esta entrada de control del multiplexor permite configurar si la CPU va a trabajar por nivel o por flanco en su entrada de pedido de interrupción. Finalmente la CPU activa la señal Borro-Pedido-Int cada vez que invoca a la rutina de servicio de la interrupción de forma de pasar el FF a 0. La señal INT-Interno es la que la CPU consulta para determinar si hay un pedido de interrupción pendiente.

16.5 Atención del Pedido de Interrupción

Cuando la CPU reconoce que hay un pedido de interrupción desencadena el mecanismo de atención a dicha solicitud. Por ahora vamos a considerar que la CPU acepta la interrupción, dejando para más adelante la consideración de en qué casos esto ocurre.

Para atender una solicitud de interrupción, la CPU realiza los siguientes pasos:

- "termina" de ejecutar la instrucción actual. Notemos que utilizamos las comillas para señalar que, en realidad, lo que ocurre es que la CPU consulta si hay un pedido de interrupción al final del ciclo de instrucción, luego de la etapa de "write" y antes del siguiente "fetch". Esto se traduce en que el pedido de interrupción debe esperar hasta el final de la ejecución de la instrucción para que sea efectivamente detectado como tal.

- salva el valor actual del puntero de instrucción (IP), como forma de poder regresar a ejecutar la siguiente instrucción del programa que fue interrumpido, luego de la ejecución de la rutina de servicio a la interrupción.
  Es de destacar que las distintas arquitecturas realizan esta actividad de diferentes formas. Podemos distinguir básicamente dos:
  - utilizando un stack. Esta vía es la habitual en las arquitecturas que implementan un stack por hardware, como es el caso de los procesadores Intel.
  - utilizando un registro. Esta vía es la habitual de las arquitecturas RISC, en particular la SPARC.

También es de destacar que hay arquitecturas que salvan adicionalmente otros registros. Por ejemplo los procesadores Intel salvan el registro que contiene el estado del CPU (el registro de FLAGS) incluyendo los bits de condición (**Z**ero, **N**egative, **C**arry, o**V**erflow) que generó la ALU en la última instrucción ejecutada. En el mismo caso de Intel x86 además, como consecuencia de utilizar un modelo de direccionamiento de memoria segmentado, salvan el registro que contiene el identificador del segmento de código del programa que fue interrumpido.

- identifica el controlador de E/S que realizó el pedido de interrupción. Veremos este tema en el siguiente punto.

- obtiene la dirección de la rutina de servicio de la interrupción correspondiente al controlador identificado. También hay distintas formas de implementar esto, aunque la mayoría son variaciones sobre el concepto de **vector de interrupciones**.
  - la arquitectura Intel x86 utiliza un identificador de 8 bits del pedido de interrupción como índice a una tabla (el vector de interrupciones) que en cada entrada tiene la dirección absoluta de la rutina de interrupción asociada, formada por el segmento (valor a cargar en el registro de segmento de código) y el desplazamiento dentro del segmento (valor a cargar en el puntero de instrucción) de la dirección de memoria de la primer instrucción de la rutina de interrupción.
  - la arquitectura SPARC utiliza también un identificador de 8 bits para el pedido de interrupción, el cuál multiplica por 16 antes de sumarle el valor contenido en un registro especial (el Trap Base Register, ó registro base de "traps", nombre que le da SPARC a las interrupciones) para formar la dirección a la cuál se va a pasar el control. De esta forma tenemos una especie de tabla que almacena las primeras 4 instrucciones (SPARC tiene instrucciones de largo fijo de 4 bytes) de cada rutina de interrupción.

- enmascara las interrupciones. Esto significa que inhibe la aceptación de nuevos pedidos de interrupción hasta tanto ocurran uno de los siguientes eventos: se retorne de la rutina de servicio de la interrupción (todas las arquitecturas tienen una instrucción específica para hacer esto) ó el código de la rutina de servicio habilite en forma explícita (a través de una instrucción) la posibilidad de aceptar nuevas interrupciones.
  Esto lo hace con el fin de que múltiples interrupciones simultáneas (desde el punto de vista del mecanismo de detección) no sobre-escriban los lugares que se están utilizando para salvar el puntero de instrucción ú otros registros vitales para poder retornar sin inconvenientes al programa originalmente interrumpido. De esta forma le permiten a la rutina de interrupción tomar los recaudos apropiados para evitar esto y recién luego, si corresponde y tiene sentido para el sistema en su globalidad, proceder a habilitar nuevamente las interrupciones.

- pasa a ejecutar la rutina de servicio a la interrupción (ó rutina de interrupción a secas), cargando en el puntero de instrucción la dirección de la primer instrucción de dicha rutina.

16.6 Identificación del Controlador que solicita la Interrupción

Es normal que un sistema tenga múltiples controladores de E/S. Por tanto hay que tener un mecanismo que permita saber cuál controlador de E/S generó un pedido concreto

de atención mediante el mecanismo de interrupción.

Las soluciones a este problema son variadas y pueden estar basadas en el hardware o en el software.

16.6.1 Identificación por Hardware

16.6.1.1 Líneas INT/IRQ independientes en la CPU

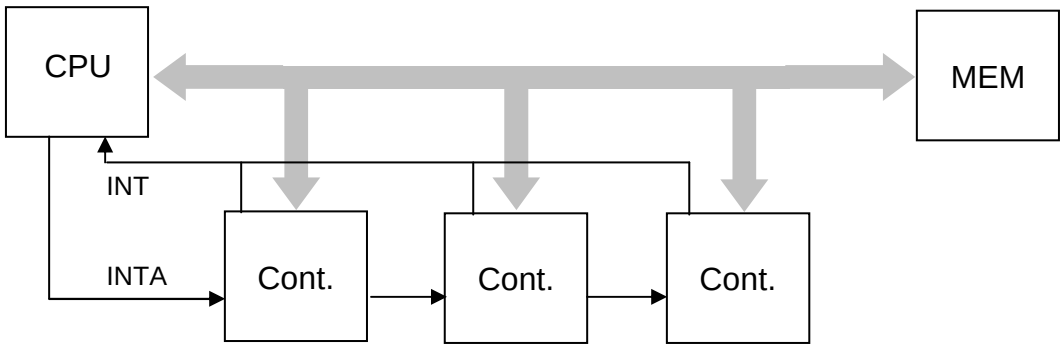
En este caso el propio CPU tiene múltiples entradas INT (numeradas, por ej: INT0, INT1, INT2, etc) y la idea es conectar un controlador de E/S a cada una de ellas. La identificación es entonces por hardware y directa: el controlador que está pidiendo la interrupción es el que está conectado a la entrada INTn donde se detecta la solicitud. En este caso hay una dirección de la rutina de servicio a la interrupción por cada línea de pedido disponible.

En algunos casos, como en el ejemplo de la arquitectura SPARC, las entradas de solicitud de interrupción están codificadas. Esto quiere decir que el SPARC en vez de tener 16 entradas de solicitud independientes, tiene 4 entradas en las cuáles se presenta el código binario del pedido de interrupción. Esto exige la inclusión de un hardware de codificación externo a la CPU de forma de convertir las entradas de solicitud individual a las entradas codificadas de la CPU.

16.6.1.2 Mecanismo INT/INTA

En mecanismo fue diseñado por Intel para sus primeros microprocesadores de 8 bits (8080 y 8085) y luego mantenido en las sucesivas familias arquitectónicas que los sucedieron. La idea es que el CPU dispone de una única entrada de pedido de interrupción INT, a la cuál se conectan en modalidad OR-cableado todos los pedidos de interrupción de los distintos controladores de E/S. También dispone de una salida denominada INTA (Interrupt Acknowledge) que le avisa al controlador de E/S que ha sido aceptado su solicitud de interrupción y le indica con esa señal que coloque en el bus de datos su identificación. La CPU entonces realiza una lectura del bus de datos y obtiene el identificador.

Este procedimiento se extiende para el caso de múltiples controladores mediante el encadenamiento de las señales INTA (los controladores de E/S que soportan este mecanismo tienen una entrada y una salida INTA) según el siguiente esquema:



De esta manera cuando la CPU inicia el ciclo de atención a un pedido de interrupción y levanta su señal de salida INTA, la misma se va propagando por los distintos

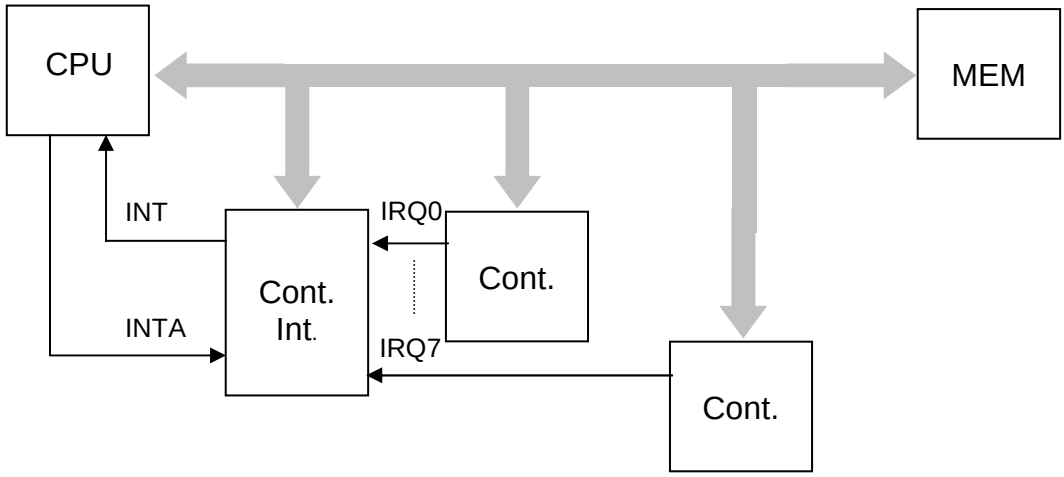
controladores en forma encadenada. De esta forma el primer controlador de la cadena que tenga un pedido de interrupción pendiente procederá a no continuar con la propagación de la señal INTA hacia los restantes controladores y será él quien coloque en el bus de datos su identificador.

Esto establece una prioridad implícita en la forma de interconectar los controladores, ya que los que estén mas cerca de la CPU tendrán la posibilidad de ser atendidos antes en caso de coincidir en el tiempo con un pedido de otro controlador que esté más distante (en "saltos de INTA") de la CPU.

Este mecanismo, si bien técnicamente es viable, parte del supuesto que cada controlador de E/S existente tendrá su propio identificador (fijado en el hardware). Esto sumado al hecho que inicialmente se reservaron solamente 8 bits para dicho identificador llevó a que no fuera posible acomodar las distintas variantes de controladores que los distintos fabricantes diseñaban. Esto sumado al hecho que solamente los microprocesadores de Intel tenían este mecanismo, llevó esta propuesta al fracaso comercial y en poco tiempo la propia Intel debió reconocer que no podía imponer a la industria su idea y generó el concepto de **controlador de interrupciones** como forma de permitir que un controlador de E/S que no estuviera diseñado para usar este mecanismo pudiera de todos modos conectarse a un sistema con INTA.

16.6.1.3 Controlador de Interrupciones

En particular se utiliza en la arquitectura x86 como forma de compatibilizar el mecanismo INT/INTA con controladores de E/S que no lo utilicen. La idea es que un controlador de interrupciones posee múltiples líneas de entrada de solicitud INT (en el caso del 8259 de Intel se denominan IRQ) para la conexión de los pedidos de interrupción de los diferentes controladores de E/S. Tiene, a su vez, una única salida de pedido de interrupción INT con su correspondiente INTA para implementar el mecanismo de identificación por hardware.



Cuando un controlador de E/S solicita una interrupción, el controlador de interrupciones genera el pedido a la CPU a través de la señal INT. La CPU cuando acepta la interrupción activa la señal INTA y en ese momento es el controlador de interrupciones quién coloca en el bus de datos la identificación que está asociada a la entrada IRQ por la que llegó el pedido. El controlador de interrupciones es configurable y tiene registros en los cuales se puede almacenar cuál es la identificación que presentará para cada una de las líneas IRQ. De esta forma al inicializar los contenidos de estos registros se puede

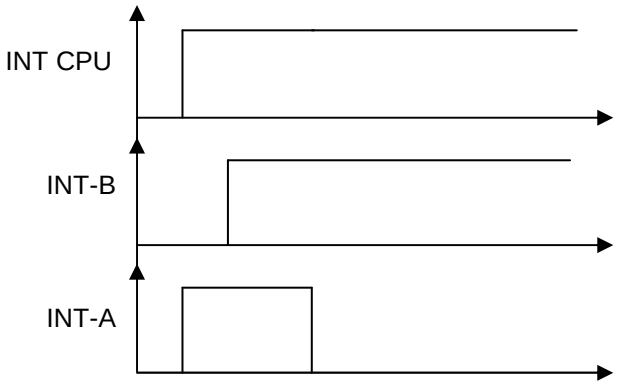
configurar dinámicamente cuáles identificadores se utilizarán para los distintos controladores de E/S que se conecten en las distintas entradas de requerimiento de interrupción.

16.6.2 Identificación por Software

Esta categoría aplica cuando todos los controladores de E/S conecten su salida de pedido de interrupción en OR-Cableado a la única entrada INT de la CPU. Más genéricamente aplica cuando hay múltiples controladores conectados en OR a la misma entrada de INT, sea ésta única o no. En esta situación la rutina de servicio de la interrupción debe tener una parte inicial que consista en el recorrido de los distintos controladores de E/S, leyendo los registros de estado hasta encontrar aquél que tenga su bit de "pedido de interrupción" en "1". En ese caso se invocará a la sub-rutina asociada a ese controlador específico.

Como la entrada de INT es el OR de todos los pedidos de los controladores, para el caso de que la CPU detecte por nivel no corro el riesgo de dejar de atender a algún controlador, aún en el caso que la estrategia de la rutina de atención sea del estilo "encuentro el primer controlador con solicitud pendiente, lo atiendo y salgo de la rutina de servicio de interrupción". Pero en el caso de la detección por flanco existe ese riesgo como vamos a analizar a continuación.

Para entender el problema analicemos el siguiente diagrama de tiempos, donde se muestra por un lado las salidas de pedidos de interrupción de dos controladores A y B y por otro el OR de ambas (que es lo que la CPU ve a su entrada de pedido de interrupción).



En el diagrama se muestra que primero ocurre el pedido de interrupción del controlador A, el cual provoca que la entrada a la CPU pase también a 1 (esto es porque es el OR cableado). Antes que la rutina de servicio a la interrupción llegue a ejecutar la acción que satisface el requerimiento del controlador A, el controlador B pasa su salida a 1, indicando que también tiene un pedido de interrupción para ser procesado. En la entrada a la CPU ese cambio no se ve reflejado porque al ser un OR, resulta que 1 OR 0, ó 1 OR 1 da el mismo resultado. Luego en el diagrama de tiempo se muestra que la salida del controlador A pasa a 0, en el momento que la CPU, luego de haber identificado que el controlador A está solicitando interrupción, ejecuta la instrucción que corresponde a lo que el controlador de E/S entiende como condición para dar por satisfecho su requerimiento. Notemos que si bien INT-A pasa a 0, la entrada de pedido de interrupción a la CPU sigue en 1 (porque INT-B sigue en ese valor).

Analicemos que pasaría si en este momento y en esta situación, enseguida después de haber detectado y procesado el pedido del controlador A, la rutina de servicio de la

interrupción termina su ejecución y el control retorna al programa que fue interrumpido.

Si la detección es por nivel esto no representa un problema, porque de darse esta situación el pedido de interrupción del controlador B sigue vigente y el mecanismo de atención al pedido de interrupción se disparará inmediatamente de nuevo.

Pero en cambio si la detección es por flanco y sucede la situación descripta, la rutina de servicio de la interrupción no será invocada nuevamente porque a los efectos de la CPU la solicitud del controlador B no existió, ya que el flanco que hubiera generado en la entrada INT quedó "enmascarado" por el nivel alto que ya existía en ese momento como consecuencia del pedido, aún no satisfecho, del controlador A. En otras palabras: el controlador B queda sin ser atendido. Y lo que es peor, si el controlador A (o cualquier otro controlador que estuviera conectado a esa línea de entrada) pide una nueva interrupción, ésta tampoco sería detectada. Es más: todo el mecanismo de interrupciones queda bloqueado.

Enfrentado a este problema una primer solución que viene a la mente es que la rutina de servicio de la interrupción consulte el estado del controlador B antes de terminar su ejecución. Si el controlador B tiene un pedido pendiente se lo atiende y entonces su salida baja a 0 y se resuelve el tema. ¿Se resuelve?. Falso. Porque si bien resolvimos la situación concreta analizada, es fácil ver que si mientras estoy atendiendo al controlador B, el controlador A vuelve a pedir interrupción y al terminar de procesar el B la rutina de servicio termina, estaremos en un problema similar (sólo que esta vez será el A el que bloquea el mecanismo). Esta situación podría repetirse múltiples veces, con lo que parecería no haber solución al problema. Sin embargo existe una estrategia de diseño de la rutina de servicio que permite que el sistema de pedidos de interrupción no se tranque al utilizar detección por flanco con múltiples controladores en la misma línea de solicitud. Esta estrategia será motivo de un ejercicio práctico, aunque se puede adelantar que la misma tiene que ver con asegurarse de alguna forma que la entrada INT estuvo en algún momento en 0, con lo que no interesa el estado de dicha entrada al momento de salir de la rutina de servicio de la interrupción. Si esto ocurrió (INT estuvo en 0 en algún momento), entonces se habrá detectado un nuevo flanco y por tanto si el flanco existió la rutina de atención será invocada nuevamente.

16.7 Habilitación de Interrupciones

La CPU tiene la capacidad de aceptar o no los pedidos de interrupción de los controladores de E/S. Esta capacidad está implementada de dos maneras, una general y otra selectiva, a las que denominaremos enmascaramiento y deshabilitación, sólo a los efectos de distinguirlas en la denominación.

- **enmascaramiento**: en este caso la CPU posee la propiedad de poder inhibir la aceptación de todas las solicitudes de interrupción, sin importar de cuál controlador provengan. Esto se realiza a través del valor de un bit de máscara de interrupción (que recibe el nombre de IM = Interrupt Mask, EI = Enable Interrupt, ET = Enable Traps, etc, dependiendo de la arquitectura), normalmente perteneciente al registro de estado de la CPU (denominado registro de FLAGS, o regitro PS por Processor Status). La CPU para determinar si hay un pedido efectivo de interrupción hace el AND de la señal de pedido interno de INT con el valor de este bit y utiliza este resultado para tomar la decisión. Si el bit está en 0 no existirá un pedido efectivo (por más que haya controladores de E/S solicitando interrupción). Si está en 1 los requerimientos de interrupción que hubiera serán procesados.

El valor de este bit se cambia mediante instrucciones específicas del set de instrucciones del procesador. A estas instrucciones las denominaremos **enable** y **disable**. En un lenguaje de alto nivel las representaremos por los procedimientos **enable()** y **disable()**.
La CPU pone automáticamente a 0 este bit al invocar a la rutina de servicio de la interrupción. Por ello una **rutina de interrupción comienza su ejecución con las interrupciones deshabilitadas** y así continuarán a menos que explícitamente sean habilitadas por el código de la propia rutina. La instrucción que provoca la finalización de la rutina de interrupción y el retorno a la ejecución del programa interrumpido vuelve a recuperar el valor anterior del bit (el cuál obviamente era 1 porque sino no se hubiera invocado).
También es de señalar que cuando una CPU comienza a funcionar (se le aplica energía por primera vez) también arranca con las interrupciones enmascaradas (deshabilitadas).

En todas las CPUs existe una entrada, independiente de la INT, denominada **NMI** (por Non Maskable Interrupt) que trabaja en forma independiente de la máscara de inhibición y un pedido de interrupción que por ella sea presentado será siempre procesado. La idea de esta entrada es conectar a ella pedidos de interrupción críticos (tales como la falla de la fuente de alimentación ó la detección de un error de paridad en la memoria). En la práctica la arquitectura de los computadores tipo PC no la utilizan.

- **deshabilitación**: en este caso se actúa sobre los controladores de E/S en forma individual, de forma de inhibir su eventual pedido de interrupción. Los controladores de E/S tienen habitualmente un bit en su registro de *CONTROL* que actúa como máscara para su salida de pedido de interrupción. De hecho la salida tendrá el valor resultante de la ecuación:

  salida INT = bit INT registro de *ESTADO* AND bit MASK_INT registro de *CONTROL*

  Notemos que el bit que refleja la condición de "pedido de interrupción pendiente" del registro ESTADO sigue reflejando si el controlador requiere de la intervención de la CPU. Lo que sucede es que la salida de hardware hacia la CPU queda condicionada al valor del bit de habilitación del registro *CONTROL*.
  Este mecanismo permite al programador tener poder de decisión selectivo sobre qué controladores de E/S podrán generar pedidos de interrupción y cuáles no.

  El bit MASK_INT descripto inhibe totalmente la generación de interrupciones por parte del controlador de E/S. Algunos controladores de E/S disponen también de otros bits de máscara que actúan a nivel diferenciado sobre las distintas posibles condiciones que generan interrupciones. Por ejemplo en un controlador de comunicaciones se pueden generar interrupciones por distintos motivos: recepción de un carácter, error en la línea de comunicaciones, carácter transmitido, cambio en las líneas de control de la comunicación, etc. y el controlador puede tener bits que inhiban en forma individual la generación de solicitudes de interrupción asociadas a cada condición.

16.8 Prioridades

Existen dos situaciones que pueden llevar al mecanismo de interrupciones a tener que tomar una decisión acerca de cómo proceder:

- hay dos (o más) solicitudes de interrupción simultáneas
- hay una (o más) solicitud de interrupción mientras se está ejecutando una rutina de servicio de una interrupción previa y las interrupciones han

sido habilitadas.

En cualquiera de ellas la decisión dependerá de la existencia de **prioridades** entre los distintos requerimientos.

16.8.1 Interrupciones Simultáneas

Cuando una CPU o un controlador de interrupciones implementa un mecanismo de prioridades, éste determinará cuál de múltiples solicitudes que ocurran a la misma vez será atendida.

Es de notar que el concepto de simultaneidad no implica que los pedidos de interrupción lleguen exactamente en el mismo instante del tiempo, sino que lleguen durante el período entre una verificación y otra por parte de la CPU, la cuál puede estar postergada en el tiempo si las interrupciones están deshabilitadas.

El mecanismo de prioridades puede adoptar distintas estrategias: prioridad fija, prioridad configurable o sin prioridad.

Cuando la prioridad es **fija** siempre será atendida primero la solicitud que provenga de una línea de pedido que tenga mayor jerarquía. Normalmente se utiliza la numeración de las entradas para fijar su prioridad.

Cuando la prioridad es **configurable**, la misma se puede cambiar en función de las condiciones del sistema en general.

Cuando el sistema es **sin prioridad** debe implementar un sistema de selección de la solicitud a atender que asegure la equitatividad en selección de las distintas entradas.

El mecanismo de prioridades puede estar implementado por **hardware** o por **software**. Cuando es por hardware se puede realizar a nivel de la CPU (si tiene múltiples entradas de interrupción, ya sean directas o codificadas) ó a nivel del controlador de interrupciones. Cuando es por software el mecanismo se implementa como un algoritmo en la rutina de atención a la interrupción (notemos que esta vía es la única para el caso de CPUs con una única entrada INT y sin controlador de interrupciones). El algoritmo puede implementar cualquiera de las estrategias vistas, mediante un apropiado recorrido de los controladores de E/S para averiguar cuál fue el que interrumpió, como ser:

- recorido lineal: siempre comenzando por el mismo y con un orden pre-establecido (prioridad fija)
- recorrido programable: utilizando una tabla que establece el orden a seguir (prioridad configurable)
- recorrido "round-robin": comenzando cada vez en el siguiente al último atendido, en forma circular.

16.8.2 Interrupción de Interrupción

Suponiendo que las interrupciones no están enmascaradas durante la ejecución de la rutina de interrupción (recordar que para que esto ocurra la rutina debe explícitamente ejecutar la instrucción que desenmascara las interrupciones) y ocurre un pedido de interrupción, la atención o no de dicha solicitud dependerá del esquema de jerarquías/prioridades que tenga implementada la CPU y/o el controlador de interrupciones.

Como regla general sólo se aceptará una interrupción de un nivel igual o superior (en algunas arquitecturas puede ser solamente si es superior en sentido estricto) al de la interrupción que está siendo actualmente atendida.

Para esto el mecanismo de interrupciones mantiene en algún registro el "nivel actual de interrupción" que refleja la prioridad de la interrupción que está siendo servida. Al ocurrir

una nueva solicitud, se compara el nivel de la interrupción asociado al nuevo pedido con el actual y sólo se procede si es mayor ó igual (ó mayor estricto) al actual. Ese valor es actualizado al momento de invocar la rutina de interrupción.

Las distintas propuestas arquitectónicas resuelven esto de distintas formas. La propuesta Intel lo hace en el controlador de interrupciones a través del registro ISR (In Service Register), mientras que SPARC lo hace en el procesador a través de los bits de IPL (Interrupt Processor Level) del registro PS (Processor Status).

16.9 Rutinas de Interrupción

16.9.1 Preservación del Contexto

Las rutinas de interrupción (también denominadas de servicio a la interrupción ó de atención a la interrupción) son la pieza de código que se ejecuta como resultado del mecanismo que desencadena la CPU al procesar un pedido de interrupción.

Como esta pieza de código se ejecuta en forma asincrónica a la ejecución del programa que es interrumpido y sin conocimiento de parte de éste, hay que tener ciertas precauciones al escribir este tipo de rutinas.

La consideración fundamental es que las rutinas de interrupción deben **preservar el contexto** del programa interrumpido.

Se entiende por contexto de un programa a su estado completo, el cuál incluye todas sus variables y estructuras de memoria, incluyendo las que están almacenadas en forma transitoria en los registros de la CPU y alcanzando al valor de los registros especiales de la misma, en particular el registro de banderas o bits de condición (los bits Z, N, C y V).

Parte de este contexto es salvado por el hardware quién, además del puntero de instrucción, en algunas arquitecturas salva el registro de estado del procesador (o registro de banderas o flags) como es el caso de Intel x86. Pero el resto debe ser salvado (o no alterado) por la propia rutina de interrupción, con las consideraciones correspondientes a dos situaciones distintas: no es lo mismo si la máquina es **no dedicada** que si es **dedicada**.

16.9.1.1 Máquina No Dedicada

En este caso en la máquina donde se ejecuta la rutina de interrupción existen múltiples programas en ejecución, con distintos propósitos, los cuáles han sido programados por distintos equipos de programadores, sin coordinación alguna.

En este caso hay que respetar a rajatabla la regla que la rutina de interrupción no puede modificar ninguna parte del contexto del programa interrumpido y si requiere del uso de algún registro de la CPU (cosa harto probable) deberá previamente salvarlo. Los lugares y procedimientos paa hacerlo dependerá del tipo de arquitectura (stack, array de memoria local, etc).

16.9.1.2 Máquina Dedicada

En este caso la máquina está completamente dedicada a una función específica y todos los programas y rutinas que en ella se ejecutan están desarrollados por el mismo equipo de programadores o por equipos fuertemente coordinados.

En este caso los equipos de programadores pueden establecer reglas de uso de los

recursos compartidos del sistema (típicamente los registros de la CPU) de forma que se evite la necesidad de tener que salvarlos en la rutina de interrupción e incluso puede existir la necesidad de que la rutina de interrupción modifique alguna variable del programa interrumpido (típicamente el programa principal) de forma de alertarlo de alguna condición ocurrida en la E/S y comunicada por este mecanismo.

16.9.2 Estrategias de Programación

Para programar sistemas que contengan rutinas de interrupción existen multiplicidad de estrategias y estilos. Normalmente constarán de un programa principal y una o más rutinas de interrupción asociadas a los dispositivos de E/S que deben ser considerados por el sistema.

Se pueden distinguir los siguientes casos en cuanto a la decisión de dónde colocar la lógica del sistema (la parte del código que implementa el núcleo de su comportamiento):

- toda la lógica del sistema se implementa en el programa principal y las rutinas de interrupción solamente modifican banderas indicando que se ejecutaron.

- toda la lógica se implementa en las rutinas de interrupción y el programa principal o bien está en loop infinito (máquina dedicada) o bien instala e inicializa las rutinas de interrupción y termina (máquina no dedicada).

- la lógica se implementa en parte en el programa principal y en parte en las rutinas de interrupción.

No hay una regla general que indique cuál de las estrategias es la más apropiada. Quizás lo que sí se pueda decir es que conviene alguno de los casos extremos, porque la distribución de la lógica entre el programa principal y las rutinas vuelve bastante engorroso el análisis y la depuración (debug) del código resultante.

En particular la estrategia de poner toda la lógica en el programa principal es la que permite mayor facilidad a la hora de las tareas de depuración (aunque no implica necesariamente que el código resultante sea el más sencillo siempre).

También se debe tener en cuenta que en las máquinas no dedicadas **no se debe** dejar en loop infinito un programa (porque lleva el consumo de CPU al 100% en forma permanente, lo que no es adecuado en un ambiente multitarea / multiusuario). Por lo tanto en este caso la estrategia apropiada es la de implementar la lógica totalmente en las rutinas de interrupción.

La otra consideración a tener en cuenta es como implementar técnicas de **polling** (para controladores sin capacidad de interrumpir) en sistemas con interrupciones. En este sentido hay dos formas posibles:

- se dispone de un timer (reloj) que genera una interrupción periódica y en la rutina de atención asociada se realiza la consulta al controlador de E/S. Esta variante tiene la característica de poseer cierto retardo de reacción frente a un cambio en la E/S, ya que depende del período del timer, porque éste determinará cada cuanto tiempo, como máximo, se producirá una interrupción.
- se realiza el "polling" en el programa principal. Es de notar que esta variante es la que asegura que la reacción frente a un cambio en la E/S sea instantánea.