Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Ayudantía 6 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice "solución propuesta" por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

Preguntas

1. a. **(I2 - II/2015)** Explique cómo funciona la transferencia de direcciones y datos desde/hacia los dispositivos *mapeados* a memoria.

   La pieza clave para el mecanismo de *mapeo* a memoria corresponde al ***address decoder***. La idea es que tanto la memoria como los dispositivos I/O ocupan el mismo bus de direcciones y el *address decoder* determina si la dirección corresponde a la memoria o a alguno de los dispositivos I/O. Si el caso es el último, entonces se accede al dispositivo correspondiente, desde el que se recibe un dato (o es enviado) desde el bus de direcciones.

   b. Describa las instrucciones de la `ISA` de un computador `x86` que permiten acceder a dispositivos mediante *port* I/O.

   Las instrucciones principales son dos:

   - `IN Reg,Port:` Se copia en el registro `Reg` el dato enviado por el dispositivo I/O asociado a la dirección `Port`.
   - `OUT Port,Reg:` Se escribe en el dispositivo I/O asociado a la dirección `Port` el contenido del registro `Reg`.

   En este caso, las direcciones `Port` son independientes de las direcciones de la memoria y su valor se encuentra entre 0 y 65535 (`0xFFFF`) para direcciones de 16 bits.

c. **(I2 - I/2017)** ¿Con qué tipo de dispositivo de I/O es preferible utilizar *mapeo* de memoria por sobre puertos?

Es preferible utilizarlo con dispositivos que utilicen pocas direcciones de memoria. Si utilizan menos, el número de direcciones a ocupar será menor dentro del espacio disponible en el bus de direccionamiento (evitando llegar a una eventual *memory barrier*). Por ejemplo, no convendría en absoluto mapear a memoria una tarjeta de video (¡imagina la cantidad de direcciones que habrían tomado los pixeles!).

d. ¿Por qué es mejor hacer uso de interrupciones en vez de *polling*? Mencione un ejemplo.

Las interrupciones permiten que la interacción entre la `CPU` y los dispositivos I/O se realice solo cuando uno de estos dos lo requiera. En *polling*, se revisa constantemente si algún dispositivo I/O requiere enviar datos (o que le envíen), hasta que alguno lo solicita y se realiza la solicitud. La revisión constante limita la capacidad de la `CPU` de realizar otras tareas, haciendo que el funcionamiento general sea ineficiente.

Un ejemplo sería el uso del teclado y el *mouse*. Si el computador estuviera revisando constantemente si el usuario escribió algo o no, o si movió el cursor, la eficiencia del computador sería deplorable (más aún, si revisara más dispositivos, como el lector de discos, ¡sería prácticamente imposible que corriera!). En cambio, si espera la interrupción por teclado o por cursor, no deja de realizar su conjunto de tareas mientras no se escriba ni se mueva el *mouse*.

e. **(I2 - I/2016)** ¿Cuál es la función del vector de interrupciones? ¿Cuál es su contenido?

El vector de interrupciones alberga las direcciones de las `ISR` de los dispositivos I/O que se encuentran registrados en el sistema. Entonces, al procesar una `IRQ`, el vector otorga la dirección de la `ISR` del dispositivo que hizo la interrupción, lo que permite finalmente ejecutarla.

f. **(I2 - II/2016)** Luego de recibir la señal `INTA`, ¿qué tarea(s) debe realizar un controlador de interrupciones?

El controlador de interrupciones, al recibir la `INTA`, busca en sus registros internos el dispositivo que produjo la `IRQ`. Luego, el ID de esta es enviada a la `CPU` a través del bus de datos. Finalmente, la `CPU` utiliza este ID para buscar la dirección de la `ISR` a ejecutar en el vector de interrupciones.

g. Detalle, paso a paso, cómo se manejaría una interrupción realizada por un dispositivo (llamémoslo IO_i) que se encuentra conectado a otro dispositivo (llamémoslo IO_j), donde la `ISR` de este último se encuentra almacenada en el vector de interrupciones del computador.

Enumeramos el paso a paso lo más detallado posible.

1) Asumimos que IO_j genera una `IRQ` a partir de la que realiza IO_i.
2) La `PIC` revisa su registro `IMR` para ver si la interrupción no está enmascarada. En este caso, marca un 1 en el bit correspondiente del *Interrupt Request Register*.
3) `PIC` escoge la interrupción de mayor prioridad (menor `IRQ`). En este caso, asumimos que es la enviada por IO_i. Se marca un 1 en el bit correspondiente del *In-Service Register*.
4) `PIC` envía interrupción (`INT`) a la `CPU`.
5) `CPU` termina de ejecutar la instrucción actual y guarda en el *stack* los *condition codes* (*flags*).

6) `CPU` revisa si el *flag* de las interrupciones está activo (`IF = 1`), en cuyo caso la atiende (asumimos que lo hace).
7) `CPU` deshabilita la atención de más interrupciones (`IF = 0`).
8) `CPU` envía `INTA` para saber quién interrumpió.
9) `PIC` revisa *In-Service Register* para saber el ID del `IRQ` que está siendo atendido (en este caso, el de IO_j) y lo envía mediante el bus de datos.
10) `CPU` usa el ID para buscar la dirección de la `ISR` en el vector de interrupciones.
11) `CPU` llama a la `ISR` asociada al dispositivo (`CALL Mem[id]`).
12) `ISR` respalda el estado actual de la `CPU`.
13) `ISR` ejecuta su código. Esta es la parte clave: Definimos la `ISR` como la búsqueda en el registro de estado del controlador de IO_j de la dirección de la `ISR` asociada a IO_i (que puede ser una subrutina, por ejemplo) y la llama.
14) Terminada la subrutina, el `ISR` original envía un comando `EOI` a la `PIC`, con la que se cambia a 0 el bit asociado en el *In-Service Register*, lo que indica que se terminó de atender la interrupción.
15) `ISR` devuelve el estado previo a la `CPU`.
16) `ISR` retorna.
17) `CPU` rehabilita la atención de interrupciones (`IF = 1`).
18) `CPU` recupera los *conditions codes* (*flags*) desde el *stack*.

h. Explique la diferencia entre las interrupciones realizadas por *hardware* y *software*, dando un ejemplo de cada una.

Las interrupciones por *hardware* pueden ser de dos tipos:

- Gatilladas por dispositivos I/O: Son las que realizan estos dispositivos para actualizar su estado o el de la `CPU`, generando una interrupción que ejecuta una `ISR` específica. Un ejemplo sería la actualización del cursor en el monitor de un computador a partir de la posición del *mouse*.
- Excepciones: Son las que se gatillan al encontrarse errores en la programación, generando una interrupción que ejecuta una `ISR` especializada (*exception handlers*). Un ejemplo sería dividir por 0 en una instrucción.

A diferencia de estas, las interrupciones por *software* son las que generan los mismos programas ejecutados, generalmente para ejecutar una `ISR` específica. La principal diferencia es que este no deshabilita la atención a otras interrupciones en *hardware*, por lo que hay que modificar la `IF` directamente mediante instrucciones: `CLI` (deshabilitar las interrupciones) y `STI` (habilitar las interrupciones). Un ejemplo concreto de esto es, por ejemplo, el manejo gráfico en una consola (tomaremos la *Super Nintendo*). El programa del juego *Super Mario World* genera una interrupción por *software* para modificar la tarjeta de video y desplegar una nueva imagen (por ejemplo, el movimiento de *Mario* y de un par de *Goombas*). Entonces, se ejecuta `INT dir` (siendo `dir` la dirección de memoria correspondiente a la tarjeta en el vector de interrupciones) y la `ISR` ejecutada se encarga de actualizar el contenido gráfico. Notar que aquí se trabaja directamente con la memoria de la `CPU`, por lo que convendría, en este caso, utilizar un controlador `DMA` para actualizar el estado (y así la `CPU` se encarga de ejecutar otras tareas).

i. **(I2 - II/2016)** Para los siguientes ejercicios, considere la siguiente tabla, que presenta el vector de interrupciones completo de un computador con `ISA x86` de 16 bits. El vector de interrupciones se encuentra almacenado a partir de la dirección de memoria `0x0000`:

IRQ	Dispositivo	Pos. en vector
IRQ0	*Timer* del sistema	00
IRQ1	Disco Duro	01
IRQ2	Interfaz `USB`	02
IRQ3	Interrupción *software*	03

Cuadro 1: Vector de interrupciones del computador.

I. ¿Cuantos dispositivos que generen solicitudes de interrupción pueden conectarse?
Es importante notar que uno de los dispositivos disponibles es una interfaz `USB`. Como se puede conectar un número arbitrario de dispositivos en ella, la respuesta es una cantidad infinita.

II. Dos dispositivos, teclado y *mouse*, están conectados a la interfaz USB. Describa un mecanismo para ejecutar la `ISR` correspondiente al *mouse*, cuando este genera una interrupción.
   - Se llama a la `ISR` de la controladora `USB`.
   - Esta `ISR` en particular puede tener como función leer un registro de estado específico de la interfaz para determinar la `ISR` real que se busca ejecutar (en este caso, la del *mouse*).
   - Se invoca dicha `ISR`. Un método puede ser, por ejemplo, implementarla como subrutina. También podría generar otra interrupción por *software* (*trap*).

III. ¿Que ocurriría en este computador si se ejecuta la instrucción `MOV [0],AX`?
Esto generaría un cambio en el puntero que apunta a la `ISR` del *timer* del sistema. Esto es **fatal**, ya que es utilizado para generar interrupciones que controlan la ejecución de los procesos dentro del computador y permiten realizar cambios de contexto.

IV. Proponga un esquema para permitir el acceso (lectura y escritura) controlado y centralizado al vector de interrupciones por parte de los programas, *i.e.*, el acceso solo puede realizarse a través de una interfaz entregada por el sistema operativo (o la `BIOS`).
Hint: El esquema puede incluir cambios a la arquitectura del computador.
Una opción, sería traspasar completamente el vector de interrupciones a un registro especializado (llamémoslo `I`), que solo puede ser escrito en modo supervisor/*kernel* (*i.e.* por el sistema operativo), y que es accedido/modificado por una instrucción especializada: `SINT id,ISR`, siendo `id` el ID del vector e `ISR` la dirección nueva de un `ISR` a almacenar. Se seguiría usando `INT` en la `ISA`, pero su ejecución debe ser modificada para ir acorde a lo especificado recientemente.

2. a. **(I3 - I/2013)** Un robot simple, conectado a un computador, es accesible mediante *mapeo* a memoria. Este robot se mueve en un espacio cuadrado infinito, en el cual cada grilla puede estar vacía o contener una muralla. El robot tiene comandos para ser prendido, apagado, avanzar 1 espacio hacia adelante, girar a la izquierda en 90^{o} y examinar lo que hay adelante. Cada vez que el robot se encuentra desocupado, *i.e.* ha sido recién iniciado o ha terminado una acción, genera una interrupción para informar que es posible darle un nuevo comando.

   I. Describa el mapa de memoria necesario para manejar el robot.

Dirección	Contenido/Función asociada
0	Dirección `ISR` de manejo del robot.
1	Registro de comandos del robot.
2	Registro de estado del robot.
3-...	Memoria de uso libre

Cuadro 2: Mapa de memoria definido para el robot.

Aquí es importante notar que las direcciones no siguen un orden preestablecido, se pudieron haber usado de otra forma siempre que estén las funciones asociadas correspondientes.

   II. Defina el formato de los datos que recibirá el robot como comandos y que entregará este para informar su estado.

Ubicación	Comando/Estado	Valor
Reg. Comandos	Encender	255
Reg. Comandos	Apagar	0
Reg. Comandos	Avanzar	1
Reg. Comandos	Girar Izq.	2
Reg. Comandos	Examinar	4
Reg. Estado	Recién encendido	0
Reg. Estado	Nada que informar	255
Reg. Estado	Espacio libre adelante	1
Reg. Estado	Muralla adelante	2

Cuadro 3: Valores para definir los comandos y estados del robot.

Nuevamente, aquí lo más importante es que se contengan los comandos y estados necesarios para poder cumplir con las funcionalidades del robot. Los valores utilizados no necesariamente deben coincidir con los de su respuesta.

III. Escriba en *Assembly* `x86` la `ISR` asociada al control del robot, siguiendo el siguiente comportamiento: el robot avanza hasta encontrar una muralla, en cuyo caso girará a la izquierda hasta encontrar un espacio vacío para avanzar, teniendo la precaución de que el robot no retroceda. Asuma que el espacio ha sido diseñado para que el robot no se quede pegado girando eternamente.

```
ISR_robot:
MOV BX, 0x0002       ;Revisamos el estado del robot.
CMP [BX], 0x00       ;Si es 0, inicializamos.
JE inicializar
CMP [BX], 0xFF       ;Si es 255, el robot examina.
JE examinar
CMP [BX], 0x01       ;Si es 1, antes de avanzar verificamos
JE check_retroceso   ;que no retroceda el robot.
JMP girar_izq        ;E.O.C., hay muralla. El robot gira.

inicializar:
MOV BX, 0x0003   ;Inicializamos variable en direccion de
MOV [BX], 0x00   ;memoria 3 para que el robot no retroceda.

examinar:
MOV BX, 0x0001   ;Se le da el comando al robot para
MOV [BX], 0x04   ;examinar y termina la subrutina.
JMP end_isr

check_retroceso:
MOV BX, 0x0003
CMP [BX], 0x02   ;Verificamos direccion de retroceso. Si es
JE girar_izq     ;2, el robot retrocede. Debe girar.

avanzar:
MOV [BX], 0x00   ;El robot puede avanzar. Se resetea la
MOV BX, 0x0001   ;variable de retroceso, se da el comando
MOV [BX], 0x01   ;de avanzar y termina la subrutina.
JMP end_isr

girar_izq:
MOV BX, 0x0003   ;Se actualiza la variable auxiliar de
ADD [BX], 0x01   ;retroceso para verificar en la siguiente
MOV BX, 0x0001   ;llamada que no retroceda, se da el
MOV [BX], 0x02   ;comando de girar y termina la subrutina.

end_isr:         ;Instruccion que retorna de la
IRET             ;interrupcion.
```

Lo importante de este código, además de su funcionamiento, es que se condiga con las tablas antes definidas.

b. **(I3 - II/2012)** Suponga que se tiene un dispositivo de adquisición de imágenes térmicas conectado a un computador que tiene una microarquitectura especializada para la adquisición de imágenes, pero con `ISA` compatible con `x86` de 16 bits. El computador tiene una memoria principal de 64 kilobytes, con el siguiente mapa de memoria para los primeros 4096 bytes:

Dirección	**Función asociada**
0-5	*Exception handlers.*
6	Registro de comandos de la cámara.
7	Registro de estado de la cámara.
8-14	Vectores de interrupciones de *hardware.*
15	Vector de interrupción de escritura en disco.
16	Vector de interrupción de adquisición de imagen.
17-31	Vectores de interrupciones de *software* de uso libre.
32-123	Memoria de uso libre.
124-1023	*Buffer* de adquisición de la cámara.
1024-4096	Espacio de memoria del disco.

Cuadro 4: Tabla que muestra el mapa de memoria del dispositivo.

Se desea escribir un programa que permita adquirir imágenes mediante la cámara y luego almacenarlas en disco. Las imágenes generadas por la cámara se encuentran en escala de grises de 8 bits, ordenadas por filas en una matriz cuadrada de 30x30.

i. Escriba una `ISR` para alguna interrupción de *software* disponible, que permita adquirir una imagen y luego escribirla en disco.

   La `ISR` de la cámara no recibe parámetros y retorna en su registro de estado información sobre la adquisición. Si la adquisición fue exitosa, el registro contendrá `0xFF` y la imagen se encontrará en el `buffer` de la cámara. En caso contrario, si la adquisición falló, el registro contendrá `0x00`. Durante la adquisición, el registro contendrá el valor `0xF0`.
   La `ISR` del disco utiliza internamente el controlador de `DMA`, por lo que necesita los siguientes parámetros en los siguientes registros:

   - La dirección de inicio del origen en el registro `AX`.
   - La dirección de inicio del destino en disco en el registro `BX`.
   - La cantidad de palabras a copiar en el registro `CX`.

   Puede utilizar la cantidad de parámetros que estime conveniente para su `ISR`, pero debe dejar explícitamente escrito qué significan y dónde se almacenan.

La solución utiliza la `IRQ 17` y asume que recibe como parámetro la dirección de inicio de escritura en disco en el registro `BX`.

```
ISR17:
    INT 16       ; Se hace la obtencion de la imagen.
while:
    MOV AX, [7] ; Se revisa el estado de adquisicion.
    CMP AX, F0h ; Si es 0xF0, se sigue revisando.
    JE while    ;
    ; Se llega a este punto completado el proceso.
    MOV AX, 124 ; Se guarda la direccion de inicio del buffer.
    MOV CX, 900 ; Se guarda la cantidad de palabras del
                ;buffer.
    INT 15       ; Se inicia la escritura en disco.
```

II. Escriba un programa que llame a la subrutina del ítem anterior para adquirir tres imágenes y almacenarlas de manera consecutiva en disco. Considere que la adquisición puede fallar y que se intentará esta un máximo de tres veces por imagen.
La solución utiliza la `IRQ 18` y asume que recibe como parámetro la dirección de inicio de escritura en disco en el registro `BX`.

```
ISR18:
    MOV DX, 0    ; Numero de intentos de adquisicion.
    MOV BX, 1024 ; Se guarda la direccion de inicio del
                 ; destino.
    MOV SI, 0    ; Numero de imagenes adquiridas.
while:
    CMP DX, 3    ; Tres intentos fallidos = se deja de
                 ; intentar.
    JE end
    ADD DX, 1    ; Aumenta el numero de intentos.
    INT 17       ; Se llama la subrutina para la
                 ; transferencia.
    MOV AX, [7]  ; Se revisa el estado de adquisicion.
    CMP AX, 00h  ; Si es 0x00, fallo y se debe intentar de
                 ; nuevo.
    JE while
end:             ; Se llega aqui por exito o 3 fallos
                 ; seguidos.
    MOV DX, 0    ; Reiniciamos el numero de intentos.
    ADD BX, 900  ; Cambia direccion de destino para otra
                 ; imagen.
    ADD SI, 1    ; Aumenta el numero de imagenes adquiridas.
    CMP SI, 3    ; Si obtenemos tres imagenes, terminamos.
    JLT while
```

Importante: Notar que se pudo haber inicializado el valor del registro `BX` en la primera subrutina. No obstante, hacerlo en la segunda permite entender de mejor forma cómo se va cambiando su valor para almacenar las tres imágenes en el disco.