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

IIC2343 – Arquitectura de Computadores

Guía 3 – Repaso Examen I

Profesor: Yadran Francisco Eterovic Solano

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. **(Examen - II/2014)** Una máquina de *stack* es un computador que utiliza un *stack* en vez de registros para almacenar los resultados de las operaciones. Esto significa que cada instrucción aritmética o lógica de dos parámetros, toma los dos valores en el tope del *stack* y luego los elimina, sustituyéndolos por el valor de la operación recién realizada. Para el caso de las operaciones de un parámetro, por ejemplo `NOT`, el computador solo sustituye el valor en el tope del *stack* por el nuevo valor. Además, una máquina de *stack* es capaz de cargar valores literales en el tope del *stack* y también descartarlos.

   i. Diagrame la microarquitectura de una máquina de *stack* de 8 bits. Este computador debe ser capaz de realizar las mismas operaciones aritméticas y lógicas que el computador básico. No es necesario tener soporte para saltos.
   Como no es necesario tener soporte para saltos, nos despreocupamos de la conexión entre el registro `PC` y la memoria de datos. Por otra parte, como estaremos siempre trabajando con el *stack*, basta con que la entrada *Address* de la memoria de datos tenga una conexión directa con el registro `SP`. Ahora, lo importante de esta microarquitectura es que como podemos acceder a una dirección de la `RAM` por ciclo, es necesaria la inclusión de un registro temporal (*Reg Temp*) que pueda almacenar el valor del tope del *stack*, para que así en el siguiente ciclo se pueda utilizar el valor que se encuentra en la siguiente posición (accediendo a este a través de un decremento de `SP`) e ingresando ambos resultados a la `ALU` con la operación correspondiente. Finalmente, este resultado se sobreescribe en la dirección a la que apunta el *stack pointer* actualmente, logrando el comportamiento solicitado.

Por otra parte, como se pueden añadir literales o descartar el tope, es necesario incluir las instrucciones `PUSH Lit` y `POP`, lo que motiva la inclusión de un multiplexor (*Mux Stack*) que determine si el ingreso a la memoria de datos es desde la `ALU` o de la memoria de instrucciones a (literal). El siguiente diagrama expone la idea antes planteada:

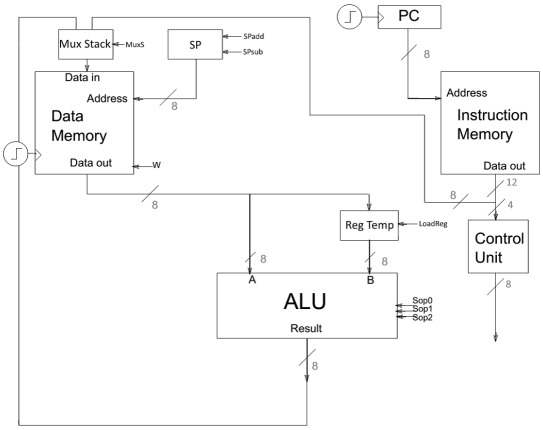

Figura 1: Diagrama de la máquina de *stack*.

II. Describa una `ISA` para la máquina del ítem anterior. Indique *opcodes*, señales de control y las instrucciones del *assembly* correspondientes.
La `ISA` descrita a continuación está adaptada al diagrama anterior:

Instrucción	***Opcode***	**Señales de control**
`PUSH Lit`	0000	`SPsub = 1`
	0001	`W = 1, MuxS = 1`
`POP`	0010	`SPadd = 1`
`ADD`	0011	`SPadd = 1, LoadReg = 1`
	0100	`Sop = ADD, W = 1, MuxS = 0`
`SUB`	0101	`SPadd = 1, LoadReg = 1`
	0110	`Sop = SUB, W = 1, MuxS = 0`
`AND`	0111	`SPadd = 1, LoadReg = 1`
	1000	`Sop = AND, W = 1, MuxS = 0`
`OR`	1001	`SPadd = 1, LoadReg = 1`
	1010	`Sop = OR, W = 1, MuxS = 0`
`XOR`	1011	`SPadd = 1, LoadReg = 1`
	1100	`Sop = XOR, W = 1, MuxS = 0`
`SHL`	1101	`Sop = SHL, W = 1, MuxS = 0`
`SHR`	1110	`Sop = SHR, W = 1, MuxS = 0`
`NOT`	1111	`Sop = NOT, W = 1, MuxS = 0`

Cuadro 1: `ISA` de la máquina de *stack*.

b. **(I1 - I/2016)** Una máquina `RAM` es un tipo de computador en el cual se utiliza solo la memoria `RAM` de datos para almacenar los resultados de las operaciones aritméticas y lógicas. Una máquina `RAM` puede tener más registros para uso interno, pero para un programador, solo se encuentran expuestos la memoria y los literales para realizar las operaciones. Por ejemplo, `ADD A,B` es sustituida por `ADD (Dir1),(Dir2)`. Construya una máquina `RAM` que posea las mismas funcionalidades que el computador básico. Especifique detalladamente el *hardware* y el formato de las instrucciones (señales de control, *opcodes*, *assembly*).

   Para construir esta máquina basta con utilizar el *hardware* del computador básico, manteniendo además todos los *opcodes*, pero modificando el *assembly* de tal manera que las operaciones que utilizan registros sean sustituidas por las que utilizan las direcciones de memoria. La implementación de cada una de estas instrucciones se puede realizar trivialmente utilizando una secuencia de *opcodes* adecuada que implemente con instrucciones del computador básico la misma funcionalidad. Por ejemplo, la secuencia para `ADD (Dir1),(Dir2)` serían los *opcodes* correspondientes a la siguiente secuencia de instrucciones del computador básico: `MOV A,(Dir1)`; `MOV B,(Dir2)`; `ADD (DIR)`.

c. **(I1 - II/2016)** En esta pregunta deberá diseñar un computador especializado en el manejo de matrices. El computador debe ser capaz de: i) copiar una matriz desde la memoria de datos a un registro y viceversa, ii) sumar 2 matrices almacenadas en registros distintos y almacenar el resultado en un registro.

   I. Haga el diagrama del computador, considerando que las matrices pueden tener como máximo $N \times N$ elementos, cada uno de 1 byte.
      En este caso, no es necesario modificar de forma significativa el diagrama del computador básico. Basta con aumentar el tamaño de los registros, buses de datos, `ALU` y las palabras de memoria a N^2 bytes para que puedan almacenar la matriz en su totalidad.

   II. Diseñe el *assembly* del computador. Cada instrucción debe estar asociada a un *opcode* y estos a sus respectivas señales de control.
      En base al diagrama anterior, basta utilizar el *assembly* del computador básico. La única diferencia se da en los literales, donde existen dos opciones: i) separar cada uno de los elementos de una matriz con coma (,) y ii) declarar la matriz completa como un único número de tamaño N^2 bytes (este último hace más sentido dada la representación anterior).

   III. Agregue tanto al *hardware* como al *assembly* soporte para una instrucción que permita modificar el valor de un elemento arbitrario de una matriz almacenada en la memoria de datos.
      Una posible solución es agregar unidades de *shifting* para aislar el elemento buscado y modificarlo y luego realizar operaciones lógicas para integrar este valor con la matriz completa. Notar que esto añade una gran complejidad en términos de *hardware*, pero logra cumplir el objetivo.

d. **(Examen - II/2012)** ¿Qué implicancia tiene en el tamaño de los programas el eliminar la conexión entre memoria de datos y `PC` (*program counter*) en el computador básico?

   Eliminar esta conexión repercute en la implementación de subrutinas, ya que no sería posible cargar en el registro `PC` las líneas del código que se almacenan en el *stack* para su posterior retorno. Esto implicaría un tamaño mucho mayor en los programas, al tener que repetir código constantemente (un ejemplo es el caso de la multiplicación).

e. **(I1 - II/2015)** Modifique el diagrama del computador básico (sin saltos y subrutinas) de manera que soporte la ejecución de la instrucción `GOTO dir`, que fuerza que la siguiente instrucción en ejecutarse sea la ubicada en la dirección `dir`.

Para lograr esto, basta con agregar una conexión directa entre el literal de la memoria de instrucciones y el registro `PC`. A continuación se muestra esta idea:

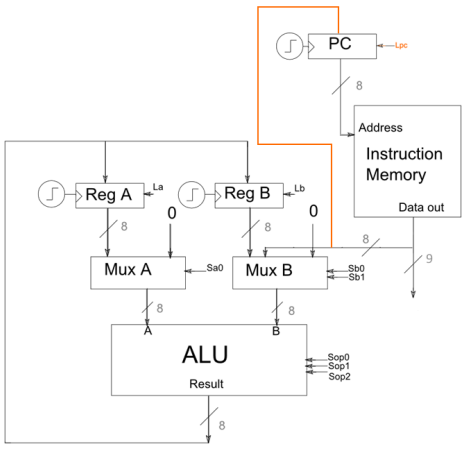

Figura 2: Computador básico con la implementación para la instrucción `GOTO dir`.

Basta con activar la señal L_{pc} una vez que esta instrucción sea llamada. Notar que esta instrucción es el equivalente al `JMP dir` que se encuentra en la versión final del computador básico, el que incluye saltos y subrutinas.

f. **(I2 - I/2015)** El soporte para subrutinas del computador básico tiene la limitación de que la instrucción `CALL` siempre tiene que llamarse con un *label* o un literal como parámetro. Esto significa que no es posible, por ejemplo, llamar a una subrutina mediante un número previamente calculado y almacenado en un registro, `CALL A` (una especie de direccionamiento indirecto de subrutinas). Describa cómo es posible saltarse esta limitación, *i.e.* permitir el llamado de subrutinas indicando su dirección mediante un número previamente calculado, usando el *assembly* del computador básico y sin modificar la arquitectura de ninguna manera.

Para esta pregunta, el análisis clave tiene que ver con qué instrucciones modifican el `PC`, sin contar lógicamente a `CALL`. Solo existen 2: `RET` y las instrucciones de salto. El problema con estas últimas es que tienen la misma limitante que `CALL`, por lo que están descartadas. Esto nos deja solo con `RET` que, a pesar de que pueda parecer poco intuitivo, puede realizar justamente lo que necesitamos. `RET` carga en el `PC` el valor almacenado en el tope del *stack*, por lo que basta con cargar en el *stack* la dirección a la que queremos saltar y luego llamar a `RET`. Para el retorno de una subrutina llamada de esta manera, basta con cargar en el *stack*, antes de introducir la dirección de la subrutina, la dirección donde se quiere retornar, de manera que el `RET` que es llamado dentro de la subrutina, extraiga desde el *stack* la dirección de retorno correcta. En resumen, si asumimos que la dirección a la que queremos saltar se encuentra almacenada en el registro `A`, la solución al problema es la siguiente:

```
    PUSH retorno
    PUSH A
    RET
retorno:
```

g. **(I1 - I/2018)** Modifique el *hardware* del computador básico para que las instrucciones `RET` y `POP` tomen un solo ciclo.

Una opción es hacer uso de un sumador de entradas `SP` y 1. Este puede tener su salida conectada a un registro llamado "`SP+1`" conectado al *clock* del computador básico, de forma que esté sincronizado con el resto de los registros. Luego, su salida puede estar conectada al `Mux Address` y, en caso de las operaciones `RET` y `POP`, se configura *Sadd* para su selección. Luego, se puede obtener `Mem[SP+1]` de forma inmediata para su posterior escritura en el registro que corresponda (ya sea `PC`, `A` o `B`). El siguiente diagrama muestra esta idea:

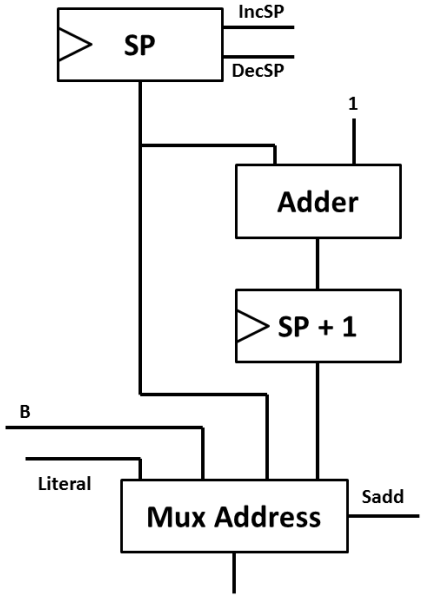

Figura 3: Diagrama del registro `SP+1`.

Esto, finalmente, permite reducir las operaciones `POP` y `RET` a un solo *opcode*, configurando correspondientemente al `Mux Address` para la selección del registro creado.

2. a. **(I2 - II/2014)** Describa una convención de llamada para x86, que sea más rápida que *stdcall* al momento de leer y escribir parámetros y valores de retorno. Contrapese las posibles ventajas y desventajas.

   Una posible convención podría utilizar los registros de la arquitectura x86 para almacenar los parámetros de entrada de las subrutinas. Si la cantidad de parámetros excede la cantidad de registros, se utiliza el *stack* para los restantes parámetros. Comparada con *stdcall*, esta convención es más rapida para leer los argumentos, pero también es más compleja, ya que no todos se ubican en el mismo lugar y habría que tener cuidado con el uso de los registros dentro de las subrutinas.

   b. **(I2 - II/2014)** ¿Es posible emular el funcionamiento del registro `BP` en el computador básico, sin modificar la arquitectura? Si su respuesta es positiva, esboce la solución. Si es negativa, explique el motivo.

   Sí, es posible emular el registro `BP`. Esto se puede hacer utilizando una dirección fija de memoria. De esta manera, en vez de hacer referencia a **BP**, se hará referecia a esta dirección (se podría configurar el *assembler* de forma que reemplace `BP` por su dirección). El único cuidado necesario es asegurarse que el valor almacenado en esa dirección no sea sobreescrito.

   c. **(Apuntes - Arquitectura x86)** El siguiente código en *Assembly* x86 obtiene una potencia mediante subrutinas.

```
MOV BL,exp
MOV CL,base
PUSH BX ; Paso de parametros (de derecha a izquierda)
PUSH CX
CALL potencia ; potencia (base,exp)
MOV pow,AL ; Retorno viene en AX
RET
potencia: ; Subrutina para el calculo de la potencia
    PUSH BP
    MOV BP,SP ; Actualizamos BP con valor del SP
    MOV CL,[BP + 4] ; Recuperamos los dos parametros
    MOV BL,[BP + 6]
    MOV AX,1 ;AX = 1
start:
    CMP BL,0 ; if exp <= 0 goto endpotencia
    JLE endpotencia
    MUL CL ; AX = AL * base
    DEC BL ; exp --
    JMP start
endpotencia:
    POP BP
    RET 4 ; Retornar , desplazando el SP en 4 bytes
base db 2
exp db 2
pow db 0
```

I. Describa cómo se ejecuta este programa.
El paso a paso se realiza de la siguiente forma:

1) Se almacena el exponente en el registro `BL` (bits menos significativos de `BX`) y la base en el registro `CL` (bits menos significativos de `CX`). Entonces, tenemos que `BX = 0x0002` y `CX = 0x0002`.
2) Si definimos la función como *potencia*(*base*, *exponente*), el paso de parámetros al *stack* es como sigue:
   - El tope del *stack* (`SP`) corresponde al final de la memoria, *i.e.* `0xFFFF`.
   - Se guarda el exponente (`PUSH BX`). Ahora, el tope es `0xFFFE`.
   - Se guarda la base (`PUSH CX`). Ahora, el tope es `0xFFFC`.
   - Tenemos entonces que `SP = 0xFFFC`.
3) Se llama a la subrutina "potencia", por lo que se almacena `PC+1` en el *stack*: `SP = 0xFFFA`.
4) Al ejecutar la subrutina, se guarda el registro `BP` en el *stack*: `SP = 0xFFF8`.
5) Ahora, con `MOV BP,SP` se tiene que `BP = 0xFFF8`.
6) Recuperamos los parámetros en los registros correspondientes: `CL = Mem[0xFFFC] = 0x02, BL = Mem[0xFFFE] = 0x02`. Notemos que esto funciona debido a que el orden en que se almacenan las palabras en memoria es *little endian* (si fuera *big endian*, lo correcto habría sido trabajar con los registros `BH,CH` desde el principio).
7) Usamos el registro `AX` (hasta ahora vacío) para almacenar el resultado final de la potencia. `AX = 0x0001`.
8) Comienza el código de "start". Al ver que `BL` es distinto de 0, se ejecuta `MUL CL`, *i.e.* `AX = AL * CL = 0x01 * 0x02 = 0x0002`. Entonces, tenemos ahora que `AL = 0x02`.
9) Decrece `BL`, `BL = 0x01` y por ende `BX = 0x0001`.
10) Volvemos al comienzo de "start". Al ver que `BL` es distinto de 0, se ejecuta `MUL CL`, *i.e.* `AX = AL*CL = 0x02 * 0x02 = 0x0004`. Entonces, tenemos ahora que `AL = 0x04`.
11) Decrece `BL`, `BL = 0x00` y por ende `BX = 0x0000`.
12) Volvemos al comienzo de "start". Al ver que `BL` es igual a cero, ejecutamos el código de "endpotencia", que realiza lo siguiente:
    - `BP = Mem[0xFFF8,0xFFF9] = 0x0000`. Ahora, `SP = 0xFFFA`.
    - `POP PC+1` (para el retorno). Entonces, `SP = 0xFFFC`.
    - Con `RET 4`, se tiene `ADD SP,4` $\rightarrow$ `SP = 0xFFFF`. El *stack pointer* recupera su posición inicial.
13) Terminada la subrutina, se almacena en "pow" lo almacenado en el registro `Al`. Finalmente, `pow = 0x04`.

Si bien aquí no pareciera haber mucha utilidad en almacenar `BP` en el *stack* al comienzo de la subrutina, se vuelve vital al llamar a otra subrutina dentro de la primera (que sucede, por ejemplo, en subrutinas recursivas).

II. ¿Cómo se modificaría este programa para hacer uso de variables locales? ¿Cómo cambia la dinámica desde el punto de vista de los registros `BP` y `SP`?
En los mismos apuntes se presenta una forma de utilizar variables locales:

```
MOV BL,exp
MOV CL,base
PUSH BX ; Paso de parmetros (de derecha a izquierda)
PUSH CX
CALL potencia ; potencia (base,exp)
MOV pow,AL ; Retorno viene en AX
RET
potencia: ; Subrutina para el calculo de la potencia
    PUSH BP
    MOV BP,SP ; Actualizamos BP con valor del SP
    SUB SP,2 ; Reservamos espacio para variable i
    MOV CL,[BP + 4] ; Recuperamos los dos parametros
    MOV BL,[BP + 6]
    MOV AX,1 ;AX = 1
    MOV [BP - 2],0 ; Variable i = 0
start:
    CMP [BP - 2],BL ; if i >= n goto endpotencia
    JLE endpotencia
    MUL CL ; AX = AL * base
    INC [BP - 2] ; i++
    JMP start
endpotencia:
    ADD SP,2 ; Eliminamos el espacio para la variable i
    POP BP
    RET 4 ; Retornar , desplazando el SP en 4 bytes
base db 2
exp db 2
pow db 0
```

Como ahora el *stack* crece con mis variables locales, puedo decrementar `BP` para acceder a ellas (que es equivalente a "subir" por el *stack*). Es importante notar que se modifica `SP` para que considere este nuevo tope (`SUB SP,2`), por lo que es válida la notación `BP-2`. Al final de la subrutina, antes de recuperar el valor de `BP`, es importante incrementar `SP` para que "olvide" las variables locales y que `POP BP` retorne efectivamente el valor del registro, no el valor de una variable local (además de posicionar correctamente al *stack pointer* finalizada la subrutina).

d. **(I2 - II/2014)** Implemente, usando el *assembly* x86 y la convención *stdcall*, un programa que calcule el máximo común divisor de dos enteros no negativos, donde ambos no pueden ser 0 simultaneamente, utilizando el algoritmo de Euclides, descrito a continuación:

$$\text{Para } a, b \geq 0,\ mcd(a,b) = \begin{cases} a & \text{, si } b = 0 \\ mcd(b, a \bmod b) & \text{, en cualquier otro caso.} \end{cases}$$

El siguiente programa cumple con el cálculo pedido:

```
        JMP    main
        a db    ?
        b db    ?
        res db 0
    main:
        MOV    AX, 0
        MOV    AL, b
        PUSH   AX
        MOV    AL, a
        PUSH   AX
        CALL   euclides
        MOV    res, AL
    euclides:
        PUSH   BP
        MOV    BP, SP
        MOV    BX, [BP-6]
        CMP    BL, 0
        JMP    setA
        MOV    AX, [BP-4]
        DIV    BL
        MOV    CX, 0
        MOV    CL, AH
        PUSH   CX
        PUSH   BX
        CALL   euclides
        JMP    return
    setA:
        MOV    AX, [BP-4]
    return:
        POP BP
        RET 4
```

3. 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 2: 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.
Nota: Recuerde que en la `ISA` x86 existe la instrucción `INT dir`, la que corresponde a una interrupción por *software*, siendo `dir` una dirección dentro del vector de interrupciones.

4. **(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 3: 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.

a. 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.
```

b. 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.