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

IIC2343 – Arquitectura de Computadores

Ayudantía 5 – Solución propuesta

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. Indique la ISA del computador básico visto en el curso y la correspondiente a la arquitectura x86, explicando sus diferencias.

   La ISA del computador básico corresponde a un set de instrucciones RISC (*Reduced Instruction Set Computer*), la que posee funcionalidades básicas al tener una arquitectura más sencilla en el computador. Esto implica que para realizar operaciones más avanzadas sería necesario realizar un programa más complejo. En cambio, la ISA del computador con arquitectura x86 corresponde a un set de instrucciones CISC (*Complex Instruction Set Computer*), la que posee funcionalidades más avanzadas por una arquitectura más compleja y costosa. Un ejemplo de esto son los comandos de multiplicación y división que se pueden realizar de forma directa (en vez de tener que hacer un programa completo para realizarlo, como lo es en el caso de la RISC del computador básico).

   b. **(I2 - I/2017)** ¿Cuántas palabras de la memoria son modificadas al ejecutar la instrucción `ADD [BH], AX`?

   Se modifica un total de **cero** palabras, ya que la operación no es válida. En la arquitectura x86, las direcciones de memoria son de 16 bits, mientras que en este caso se está tratando de acceder a una con 8 bits (recordar que el registro `BH` es uno de los registros de 8 bits de `BX`).

c. **(I2 - I/2016)** ¿Por qué es necesaria la existencia de la instrucción `RET` Lit en un computador x86?

Su existencia es necesaria, ya que permite dejar el *stack pointer* (`SP`) en la posición que le corresponde según **la cantidad de parámetros ingresados a la llamada**. De no hacerlo, asumiendo que hubo un llamado anterior, al retornar a este no se podría acceder de forma correcta ni a los parámetros ni a las variables locales (en caso de haber), dado que el registro `SP` no estaría ubicado en el tope del *stack*, sino que más arriba.

d. **(I2 - I/2013)** Al iniciar el cuerpo de una subrutina, ¿por qué es necesario ejecutar las instrucciones `PUSH BP` y `MOV BP, SP`? ¿Qué pasa si no se ejecutan?

Veremos la importancia e implicancia de cada una por separado.

- `PUSH BP:` Respalda el valor previamente almacenado en el registro `BP`. Si no lo utilizamos, quizás no habría problema si estamos seguros de que usaremos una única subrutina con una llamada (ya que después no volveríamos a usar el registro). Sin embargo, imposibilitaría el llamado consecutivo de subrutinas (por ejemplo, con una recursión), ya que no podría volver a la dirección correspondiente, y los accesos a memoria para los parámetros y variables locales sería erróneo.
- `MOV BP, SP:` Le otorga al registro `BP` el valor almacenado en `SP`, lo que permite luego usarlo como referencia para tener la dirección de los parámetros y las variables locales. Si no lo utilizáramos, tendríamos solo la posibilidad de usar `SP` como referencia, lo que complejizaría de sobremanera el código ya que el programador debería estar siempre atento a la última posición adquirida del registro.

e. **(I2 - I/2017)** ¿Cuántas llamadas recursivas a una función es posible hacer como máximo en un computador x86 de 16 bits? Indique claramente sus supuestos.

Para obtener el número concreto, se utilizará el código más simple posible:

```
CALL f
f: CALL f
```

Asumiendo una microarquitectura Von Neumann, se tiene el almacenamiento inicial de los *opcodes* de las instrucciones y los literales asociados (en este caso, la dirección del label `f`).

Supuesto: Tanto los literales como los *opcodes* son de 16 bits (para simplificar el análisis).

Con el supuesto anterior, vemos que se hace uso de un total de 8 direcciones de memoria para el programa escrito. Esto es:

- Dos direcciones por cada *opcode* (registro de 16 bits almacenado en dos palabras de memoria de 8 bits). Se hace uso de un total de 4 direcciones de memoria en total.
- Dos direcciones por cada literal. Se hace uso de un total de 4 direcciones de memoria en total.

Esto conlleva a un total de $2^{16} - 8$ direcciones de almacenamiento disponibles. Ahora, por cada llamado recursivo, se almacenará en el *stack* la dirección de retorno. Al ser esta de 16 bits, hace uso de dos direcciones de memoria por salto. Finalmente, la cantidad de llamadas recursivas R será igual a:

$$R = \frac{2^{16} - 8}{2} = 2^{15} - 4$$

f. **(I2 - II/2016)** Describa un mecanismo para, en tiempo de ejecución, escribir el código de una subrutina y luego llamarla, utilizando el Assembly x86 de 16 bits. Asuma que tiene disponible la especificación completa de la ISA.

Separamos el mecanismo en distintas fases:

- **Fase 1:** Almacenamos una dirección de memoria específica donde comenzaremos a escribir el código de la subrutina.
- **Fase 2:** Desde dicha dirección, escribimos el opcode correspondiente a cada instrucción dentro de la subrutina (y de los literales, si corresponde).
- **Fase 3:** Una vez que termina la escritura, utilizamos el comando `CALL` para saltar directamente a la primera instrucción escrita.

g. **(I2 - I/2017)** Para el siguiente programa escrito en Assembly x86-16, indique los valores de los registros `SP` y `BP` y del stack completo, al momento de ingresar al label `set`.

```
        MOV BL,3
        PUSH BX
        CALL func
        RET
        func: PUSH BP
              MOV BP,SP
              MOV BL,[BP + 4]
              CMP BL,0
              JE set
              MOV CL,BL
              DEC CL
              PUSH CX
              CALL func
              MOV BL,[BP + 4]
              MUL BL
              JMP end

        set: MOV AX ,1

        end: POP BP
             RET 2
```

Para poder ver el estado final de nuestro *stack* y los registros, se describirá la ejecución del programa paso a paso hasta ingresar al label `set`. Por simplicidad, se asumirá como la primera línea de código un registro `PC = 0x0000`.

1) (`MOV BL,3`): Se almacena el número 3 en el registro `BL` (bits menos significativos de `BX`). Entonces, tenemos que `BX = 0x0003`.
2) (`PUSH BX`): Antes de ver en detalle la ejecución de esta instrucción, es importante notar que la función `func` tiene un solo parámetro, que en este caso será recibido a través del registro `BX`. El tope del *stack* (`SP`) inicialmente corresponde al final de la memoria, i.e. `0xFFFF`. Luego, al ejecutar esta instrucción se almacena en el *stack* el valor del registro. Al ocupar dos palabras de memoria, se tiene que ahora `SP = 0xFFFE`[1].
3) (`CALL func`): Se llama a la subrutina '`func`', por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). En este caso, `PC + 1 = 0x0003` y `SP = 0xFFFC`.
4) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFFA`. Cabe destacar que inicialmente no se conoce el valor de `BP` por lo que, sin pérdida de generalidad, se asumirá igual a `0x0000`.
5) (`MOV BP,SP`): Se tiene que `BP = 0xFFFA`.
6) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0003`
7) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
8) (`JE set`): Este salto **no será ejecutado**, puesto que la *flag* `Z` no estará activa (`BL` es distinto de cero).
9) (`MOV CL,BL`): Se almacena en el registro `CL` el valor del registro `BL`. Entonces, `CX = 0x0003`.
10) (`DEC CL`): El registro `CL` decrece en una unidad. Por lo tanto, `CX = 0x0002`.
11) (`PUSH CX`): Se almacena en el *stack* el valor del registro `CX`. Se tiene ahora `SP = 0xFFF8`. Este corresponderá al parámetro de la primera llamada recursiva.
12) (`CALL func`): Se llama a la subrutina '`func`' (primera llamada recursiva), por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). Entonces, `PC + 1 = 0x000D` y `SP = 0xFFF6`.
13) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFF4`. Cabe destacar que ahora **sí se conoce** el valor de `BP`: `BP = 0xFFFA`.
14) (`MOV BP,SP`): Se tiene ahora que `BP = 0xFFF4`.
15) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del stack, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0002` (Notar que ahora se almacenó el valor disminuido en una unidad anteriormente).
16) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
17) (`JE set`): Este salto **no será ejecutado**, puesto que la *flag* `Z` no estará activa (`BL` es distinto de cero).
18) (`MOV CL,BL`): Se almacena en el registro `CL` el valor del registro `BL`. Entonces, `CX = 0x0002`.
19) (`DEC CL`): El registro `CL` decrece en una unidad. Por lo tanto, `CX = 0x0001`.
20) (`PUSH CX`): Se almacena en el *stack* el valor del registro `CX`. Se tiene ahora `SP = 0xFFF2`. Este corresponderá al parámetro de la segunda llamada recursiva.

[1] Es importante recordar que en Assembly x86 el registro `SP` apunta a la última palabra ingresada (en el Assembly del computador básico el registro `SP` apunta a una posición más arriba).

21) (`CALL func`): Se llama a la subrutina '`func`' (segunda llamada recursiva), por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). Entonces, `PC + 1 = 0x000D` y `SP = 0xFFF0`.
22) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFEE`, mientras que el valor almacenado es `BP = 0xFFF4`.
23) (`MOV BP,SP`): Se tiene que `BP = 0xFFEE`.
24) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0001` (Notar que ahora se almacenó el valor disminuido en una unidad anteriormente).
25) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
26) (`JE set`): Este salto **no será ejecutado**, puesto que la flag `Z` no estará activa (`BL` es distinto de cero).
27) (`MOV CL,BL`): Se almacena en el registro `CL` el valor del registro `BL`. Entonces, `CX = 0x0001`.
28) (`DEC CL`): El registro `CL` decrece en una unidad. Por lo tanto, `CX = 0x0000`.
29) (`PUSH CX`): Se almacena en el *stack* el valor del registro `CX`. Se tiene ahora `SP = 0xFFEC`.
30) (`CALL func`): Se llama a la subrutina '`func`' (tercera llamada recursiva), por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). Entonces, `PC + 1 = 0x000D` y `SP = 0xFFEA`.
31) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFE8`, mientras que el valor almacenado es `BP = 0xFFEE`.
32) (`MOV BP,SP`): Se tiene que `BP = 0xFFE8`.
33) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0000` (Notar que ahora se almacenó el valor disminuido en una unidad anteriormente).
34) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
35) (`JE set`): Este salto **será ejecutado**, puesto que la `flag Z` estará activa (`BL` es igual a cero).

Finalmente, se tiene que:

- `SP = 0xFFE8`
- `BP = 0xFFE8`

- ***Stack*:**

Variable	Dirección	Palabra
BP	**0xFFE8**	**0xEE**
BP	**0xFFE9**	**0xFF**
PC + 1	**0xFFEA**	**0x0D**
PC + 1	**0xFFEB**	**0x00**
CX	**0xFFEC**	**0x00**
CX	**0xFFED**	**0x00**
BP	**0xFFEE**	**0xF4**
BP	**0xFFEF**	**0xFF**
PC + 1	**0xFFF0**	**0x0D**
PC + 1	**0xFFF1**	**0x00**
CX	**0xFFF2**	**0x01**
CX	**0xFFF3**	**0x00**
BP	**0xFFF4**	**0xFA**
BP	**0xFFF5**	**0xFF**
PC + 1	**0xFFF6**	**0x0D**
PC + 1	**0xFFF7**	**0x00**
CX	**0xFFF8**	**0x02**
CX	**0xFFF9**	**0x00**
BP	**0xFFFA**	**0x00**
BP	**0xFFFB**	**0x00**
PC + 1	**0xFFFC**	**0x03**
PC + 1	**0xFFFD**	**0x00**
BX	**0xFFFE**	**0x03**
BX	**0xFFFF**	**0x00**

Cuadro 1: *Stack* resultante del programa.

Es importante mencionar qué es lo que hace este programa finalmente. Al cumplirse `BL = 0`, se guarda en `AX` el valor igual a 1 (*i.e.* `AX = 0x0001`). Luego, al retornar a la llamada anterior, con las instrucciones `MOV BL,[BP +4]` y `MUL BL` se tendrá que `AX = AL * BL`. Dado el almacenamiento del *stack*, en primer lugar se tendrá `AX = 1 * 1 = 1`. No obstante, al volver a retornar se ejecutará `AX = 1 * 2 = 2` y, finalmente, `AX = 2 * 3 = 6`, retornando por último a la instrucción `RET` para finalizar el programa. De aquí se desprende que `func` almacena en `AX` el factorial de `BX`.

2. 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.

   a. 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.

   b. 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.

c. 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 examinar
MOV [BX], 0x04   ;y termina la subrutina.
JMP end_isr

check_retroceso:
MOV BX, 0x0003
CMP [BX], 0x02   ;Verificamos direccion de retroceso. Si es 2,
JE girar_izq     ;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 comando
MOV [BX], 0x02   ;de girar y termina la subrutina.

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

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