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

IIC2343 – Arquitectura de Computadores

Ayudantía 5 – 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. 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/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.

c. **(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.

d. **(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.

e. **(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`).

f. **(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:

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

g. **(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.

h. **(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.

2. *a*) **(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.* `0xFFFE`[1]. Luego, al ejecutar esta instrucción se almacena en el *stack* el valor del registro. Al ocupar dos palabras de memoria, se tiene que `SP = 0xFFFE`[2].
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`.

[1] Al trabajar con elementos de 2 bytes en el *stack*, se parte de este valor con valores "basura" almacenados en un comienzo para ser reemplazados posteriormente.

[2] No cambia, solo se reemplaza el valor en memoria. Por otra parte, 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).

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.
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` y `BP = 0xFFE8`. A continuación, el *stack* final:

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`.

b) **(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).

c) **(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 simultáneamente, 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
```