TRABAJO PRÁCTICO

TRADUCTOR ASSEMBLER - MÁQUINA VIRTUAL

PARTE II

0 - FORMATO DE ENTREGA

Se deberá entregar, por el moodle de la asignatura, un archivo Zip, Rar o 7zip con el siguiente contenido:

- Los fuentes completos de la máquina virtual.
- Los programas ejecutables de mvc y mvx (.EXE compilado para Windows o Linux).

PROGRAMAS: A los programas se les agregan nuevos argumentos con archivos para utilizar.

```
mvc.exe AsmFilename ImgFilename [-o]
mvx.exe ImgFilename1 ImgFilename2 … ImgFilenameN [-a] [-b] [-c] [-d]
```

Donde (se agrega):

- **mvx** debe soportar una lista de imágenes (archivos.img) para incorporar a la máquina virtual. (más adelante se especifica como)
- ***-a*** es un flag que permite visualizar el estado final de la memoria de administración al finalizar la ejecución de las imágenes. (detalles en el apartado 2)
- ***-b*** es un flag que fuerza a la máquina virtual (***mvx.exe***) a detener su ejecución con un *breakpoint* y solicitar al usuario un comando para mostrar un fragmento de la memoria.
- ***-c*** hace clear screen cada vez que se ejecuta la máquina virtual y encuentra un *breakpoint*.
- ***-d*** hace el disassembler al inicio de la ejecución y cada vez que se ejecuta un *breakpoint* indicando la posición del IP.

El orden de los argumentos siempre es el mismo, sin embargo hay que tener en cuenta que algunos argumentos pueden estar ausentes.

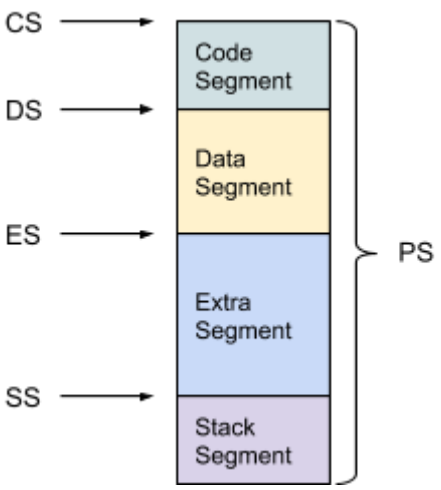
1 - INTRODUCCIÓN

En esta segunda parte del trabajo práctico se deberá ampliar la máquina virtual para que soporte, ejecutar varios procesos (no concurrentes) en la misma memoria, además instrucciones para el manejo de Strings, pila, llamado y retorno de subrutinas; además de un nuevo tipo de operando: el operando indirecto y un nuevo tipo símbolo: constantes.

También se incrementa el tamaño de la memoria a 32KiB, es decir 8192 celdas de 4 bytes, se agregan nuevas llamadas la sistema, nuevas instrucciones y directivas al compilador, se incorpora un nuevo segmento de memoria: “Stack Segment” y 5 nuevos registros (CS, PS, SS, SP y BP).

La estructura de cada proceso en la memoria quedará de 4 segmentos:

- **Code Segment**: incluye el código fuente y constantes string, ahora es apuntado por el registro CS (ya no será 0 la posición inicial).
- **Data Segment**: no se modifica, queda a continuación del Code Segment, y apuntado por el registro DS.
- **Extra Segment**: no se modifica, apuntado por ES, pero ahora puede ser compartido por varios procesos.
- **Stack Segment**: nuevo segmento dedicado para la pila del proceso, apuntado por el registro SS.



Los registros quedarán dispuestos de la siguiente manera:

Posición	Nombre	Descripción
0	**PS**	Partition Size
1	**CS**	Segments
2	DS	
3	ES	
4	IP	Instruction Pointer
5	**SS**	Stack
6	**SP**	
7	**BP**	
8	AC	Accumulator
9	CC	Condition Code
10	AX	General Purpose Registers
11	BX	
12	CX	
13	DX	
14	EX	
15	FX	

A su vez la memoria se va a dividir en una partición inicial para datos de los procesos, y una partición por cada proceso. Por ello cada proceso tendrá un registro PS (Partition Size) que contiene su tamaño de partición.

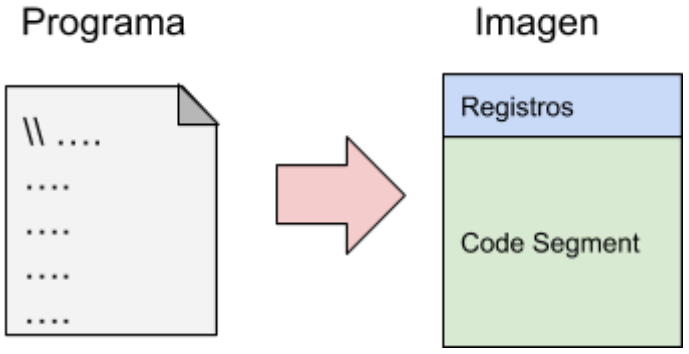
2 - ADMINISTRACIÓN DE PROCESOS

El traductor (mvc)

Al igual que en la primera parte el traductor (mvc) "compilará" a lenguaje máquina un archivo de texto con el código assembler (*.asm) y generará un archivo imagen (*.img). Siempre un archivo por vez.

A diferencia de la primera parte, la imagen generada por el traductor ya no será la totalidad de la memoria de la máquina. La imagen solo contendrá los registros y el code segment. De hecho el traductor ya no necesita conocer el tamaño de la memoria de la máquina virtual.

Además el traductor deberá completar los valores de los registros PS, CS, DS, ES, SS y SP en forma relativa a sus especificaciones, para ello se incorporan directivas al traductor, que permite al programador definir el tamaño cada segmento del proceso (a excepción del code segment) y se incorporan dentro del archivo ASM, generalmente en la primera línea, precedidas de dos barras invertidas (\\).



Sintaxis para las directivas al traductor:

```
\\ASM <SEGMENTO>=<TAMAÑO> <SEGMENTO>=<TAMAÑO> ...
```

Donde:

- SEGMENTO puede ser: DATA (para definir el *data segment*), EXTRA (para definir el *extra segment*) o STACK (para definir el *stack segment*).
- TAMAÑO es la cantidad (en decimal) de celdas de memoria destinadas a cada segmento, o -1 que indica que utilizará el segmento del proceso anterior (se dará más detalles en la especificación de la ejecución).

Por defecto (u omisión) cada segmento tendrá un tamaño de 500 celdas. **Solo** al ***Extra Segment*** se le permite asignar -1 como valor de tamaño, que sería un flag para indicar que se utilizará el mismo segmento de memoria que el programa anterior.

Ejemplos:

PRGM1.ASM

```
\\ASM DATA=300 EXTRA=200
...
<30 líneas de código del programa>
<10 celdas de constantes>
...
```

PRGM2.ASM

```
\\ASM DATA=100 EXTRA=-1 STACK=600
...
<50 líneas de código del programa>
<20 celdas con constantes>
...
```

Se ejecuta:

```
>mvc PRGM1.ASM PROG1.IMG
```

PROG1.IMG tendrá:
Partiendo con **CS=0**
El **DS** = <Tamaño del *Code Segment*> = <30 líneas * 3 Celdas + 10 Celdas> = **100**
El **ES** como está definido con un número positivo, debe ubicarse a continuación del *Data Segment*, por lo tanto será igual a DS + DATA (Tamaño del *Data segment*) = 100 + 300 = **400**
El Stack, (como hay un especificación de tamaño del ES) se coloca a continuación, por lo tanto el **SS** tendrá el valor del ES + EXTRA (Tamaño del *Extra segment*) = 400 + 200 = **600**
El **SP** se carga con el tamaño del Stack, en este caso **500** (por defecto ya que no está definido por directivas).
Finalmente se debe calcular el tamaño de la partición y cargarlo en el PS.
El **PS** es la suma del tamaño de cada segmento definido. En este caso están definidos todos los segmentos por lo tanto será:
PS = Tamaño del *Code Segment* + DATA + EXTRA + STACK
PS = 100 + 300 + 200 + 500 = **1100**

En cuanto al segundo programa cuando se ejecuta:

```
>mvc PRGM2.ASM PROG2.IMG
```

Se parte de **CS=0**
El **DS** = <Tamaño del *Code Segment*> = (50 * 3 + 20) = **170**
El Extra Segment está definido con -1, esto quiere decir que compartirá el segmento con el proceso anterior, por lo tanto en **ES** se deja **-1**
El **SS** tendrá el valor del DS + <Tamaño del *Data Segment*> = 170 + 100 = **270**
El **SP** tendrá el tamaño del Stack = **600**
El **PS** se calcula como: 170 + 100 + 600 = **870**

Finalmente quedarían las imágenes configuradas con:

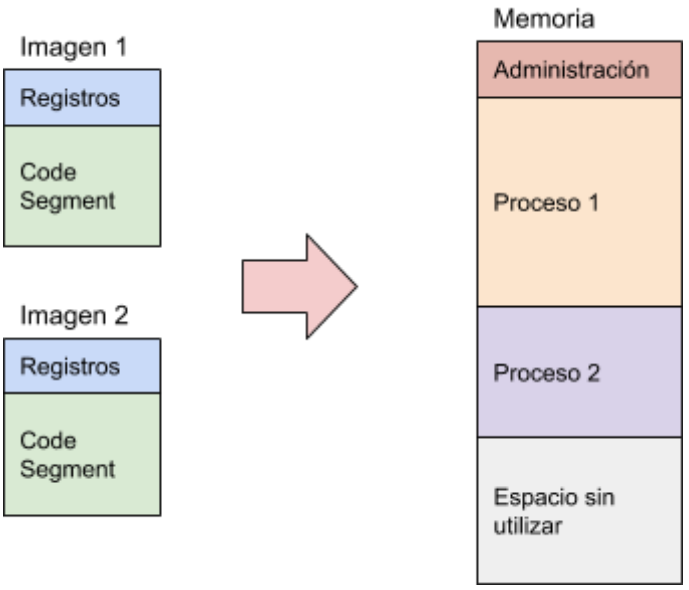
	PS	CS	DS	ES	IP	SS	SP	...
PROG1.IMG	1100	0	100	400	0	600	500	...
PROG2.IMG	870	0	170	-1	0	270	600	...

Nota: Los demás registros, en este momento, tendrán valor cero (0).

El ejecutor (mvx)

A diferencia de la primera parte, el ejecutor permite cargar en la memoria varias imágenes de procesos y ejecutarlas una por vez. Para ello debe tomar los registros CS, DS, ES y SS de cada imagen y modificar sus valores para ubicarlos en la posición correcta de la memoria.

Como la máquina virtual ahora debe administrar más de 1 proceso, se reserva un espacio en la zona más baja de la memoria para tal fin (es decir desde la 0 a la N). Este espacio contendrá 1 celda para la cantidad de procesos a ejecutar (celda 0), 1 celda para el número de procesos ejecutados (celda 1), y 16 celdas con los valores de los registros para cada uno de los procesos a ejecutar.



Los programas (imágenes) al pasar a la memoria se convierten en procesos. Cada proceso puede contener sus propios segmentos, o compartir algún segmento con otro proceso. Es por eso que para cada proceso se vuelven a configurar los registros de segmento con valores absolutos de memoria a los que deben apuntar en el momento de la ejecución de la máquina, esto sería la preparación de la mv, luego le sigue la ejecución de cada proceso.

Preparación

Siguiendo con el ejemplo anterior, ahora se ejecuta:

```
>mvx PROG1.IMG PROG2.IMG
```

En primer lugar se debe reservar espacio en memoria para la administración de los procesos. Esto sería 2 celdas más 16 celdas por la cantidad de procesos.
En el ejemplo la primer celda (celda 0) de memoria se carga 2, porque se ejecutarán 2 procesos.
En la segunda celda (celda 1), en principio carga 0, puesto que aún no se ha ejecutando ningún proceso. Luego se deben reservar (para este ejemplo) 32 celdas más, 16 para los registros de cada proceso.
Para re-calcular el valor de cada registro se debe primero calcular el valor de CS de cada proceso. El primer proceso podrá ubicarse en la primer celda libre contigua a la zona de administración, por lo tanto en este ejemplo el CS del proceso 1 (CS_1 de ahora en más) será 34. Y CS_2 será igual a CS_1 + PS_1.

En resumen:

CS_1 = 2 Celdas de control + 16 registros del proceso 1 + 16 registros del proceso 2 = 34
$CS_2 = CS_1 + PS_1$ (Partition Size del proceso 1) = 34 + 1100 = 1134

Nota: si hubiera un 3er proceso $CS_3 = CS_1 + PS_1 + PS_2$

Una vez obtenido el CS de cada proceso, para calcular a donde apuntan los demás registros de segmento (DS, ES y SS) se realiza:
Si el registro de segmento tiene -1, se utiliza el valor del registro de segmento del proceso anterior,
sino al valor del registro de segmento se le suma el CS recalculado de su proceso.

Por ejemplo:
$DS_1 = CS_1$ + DS de la imagen 1 = 34 + 100 = 134
$ES_1 = CS_1$ + ES de la imagen 1 = 34 + 400 = 434
Y el ES_2 como en la imagen 2 venía un -1, tendrá el mismo valor que ES_1.

Resumiendo los valores de cada registro de los proceso quedarían así:

	PS	CS	DS	ES	IP	SS	SP	...
PROG1.IMG	1100	0	100	400	0	600	500	...
Proceso 1	1100	34	134	434	0	634	500	
PROG2.IMG	870	0	170	-1	0	270	600	...
Proceso 2	870	1134	1304	434	0	1404	600	

De este modo cada proceso queda ubicado en la memoria y los registros de segmento contienen las direcciones absolutas de la memoria principal.

Si al hacer los cálculos el ejecutor detecta que los procesos no entran en la memoria, se muestra por pantalla el mensaje “Memoria insuficiente” y no se ejecuta ningún proceso.

¿Cómo se sabe, en el ejemplo, que la memoria es suficiente?
Porque:
CS2 + PS2 = 1134 + 870 = 2004 <= 8192 (tamaño de la memoria)

Es decir que el CS del último proceso más el tamaño de la partición del mismo debe ser menor o igual al tamaño de la memoria.

Finalmente, el último paso de la preparación sería copiar el código del programa *(Code Segment)* de la imágen de cada proceso a la posición de memoria apuntada por el CS recalculado de cada proceso.

Nota: Este proceso puede hacerse al final de todos los cálculos o con la carga de cada imágen reservando el espacio de memoria correspondiente en cada paso.

Ejecución

Una vez concluida la preparación, el ejecutor comenzará la ejecución del primer proceso y al finalizar ejecuta el siguiente, y así sucesivamente hasta que la cantidad de procesos ejecutados (celda 1 de la memoria) sea igual a la cantidad de procesos cargados (celda 0 de la memoria).

Al iniciar la ejecución de un proceso, lo primero que debe hacer la MV es copiar los valores de los registros del proceso que están en la memoria a los registros "reales" de la MV. Recuerde que si bien los valores de los registros están en la memoria, a la hora de ejecutar un proceso se debe copiar esos valores a los registros de la máquina virtual.

También hay que tomar en cuenta que ahora el IP, que se inicia en 0, ahora es relativo al CS. Por lo tanto cada vez que se trae una instrucción de memoria, ahora se tiene que extraer de la celda CS+IP.

En resumen, una posible ejecución de la máquina virtual ahora sería:

```
while (mem[1] < mem[0])
    copiar registros desde la celda (16 * mem[1] +2);
    ejecutar proceso;
    retornar los valores de los registros a la celda (16 * men[1] +2);
    mem[1] = mem[1] + 1;
endwhile
```

Si alguno de los procesos se da un "error fatal" (como stack overflow o división por cero) y no puede continuar la ejecución, se aborta la ejecución de todos los procesos siguientes.

Si se había invocado el ejecutor con el flag ***-a***, al finalizar la ejecución, sea con normalidad de todos los procesos o por un error en alguno, la MV deberá mostrar el estado final de la memoria de administración con el siguiente formato:

```
Cantidad total de procesos = 2
Cantidad de procesos finalizados correctamente = 2

Proceso 1:
PS = 0123456789 | CS = 0123456789 | DS = 0123456789 | ES = 0123456789 |
IP = 0123456789 | SS = 0123456789 | SP = 0123456789 | BP = 0123456789 |
AC = 0123456789 | CC = 0123456789 | AX = 0123456789 | BX = 0123456789 |
CX = 0123456789 | DX = 0123456789 | EX = 0123456789 | FX = 0123456789 |

Proceso 2:
PS = 0123456789 | CS = 0123456789 | DS = 0123456789 | ES = 0123456789 |
IP = 0123456789 | SS = 0123456789 | SP = 0123456789 | BP = 0123456789 |
AC = 0123456789 | CC = 0123456789 | AX = 0123456789 | BX = 0123456789 |
CX = 0123456789 | DX = 0123456789 | EX = 0123456789 | FX = 0123456789 |
...
```

Donde "0123456789" es el valor final decimal del registro alineado a derecha.

3 - SÍMBOLOS

El lenguaje assembler de MV, ya disponía de un tipo de símbolo: los rótulos (o labels), a estos se le agregan las constantes.

CONSTANTES INMEDIATAS

Una constante se define por la directiva EQU. Por ejemplo:

```
BASE  EQU  #16
```

Hace que el símbolo BASE tenga el valor 16 decimal. Luego, dicho símbolo podrá utilizarse en una instrucción. Por ejemplo:

```
ADD AX,BASE
```

Suma 16 al registro AX.

Las directivas EQU no son instrucciones ejecutables y no generan código máquina (son pseudo-instrucciones), por lo tanto son ignoradas al contar las líneas de código del mismo modo que los comentarios. Suelen colocarse al principio del programa, pero podrían estar ubicadas en cualquier parte sin que afecte a la ejecución.

Estas directivas deben soportar las mismas bases que maneja el lenguaje: octal, decimal, hexadecimal, carácter y además una cadena de caracteres (string). Sin embargo en este último caso requiere un tratamiento especial. Mientras el símbolo de un EQU octal, decimal, hexadecimal y carácter se reemplazan por un valor inmediato, un símbolo con un EQU string se reemplaza por una dirección de memoria (Constante Directa).

CONSTANTES DIRECTAS

Una constante directa es similar a una inmediata pero permite alojar un String y el valor de la constante se reemplaza por la dirección de memoria donde comienza. Por ejemplo:

```
TEXTO1  EQU  "Hola"
TEXTO2  EQU  "Chau"
```

Las cadenas de caracteres (Strings) constantes se almacenan contiguos dentro del *code segment* a continuación de la última instrucción. como secuencia consecutiva de caracteres en código ASCII, ocupando cada carácter una celda de memoria (de 32 bits) y se finaliza con con el carácter ´\0´ (es decir %00000000).

En el ejemplo: si el código *assembler* tiene 30 líneas, la "H" (de Hola) se almacenará en la celda 90 (en la traducción con CS=0), y el \0 en la celda 94. El carácter "C" de TEXTO2 se almacenará en la celda 95, y así sucesivamente con los demás caracteres y constantes.

Por lo tanto en la tabla de símbolos TEXTO1 tendrá el valor 90 y TEXTO2 el valor 95.

NOTAS ADICIONALES

- En el manejo de símbolos no se deben diferenciar mayúsculas de minúsculas.
  (Ej:'TOPE', 'Tope' y 'tope' son el mismo símbolo).

- La longitud máxima de un símbolo es de 10 caracteres.

- Los símbolos deben comenzar siempre por una Letra y tener al menos 3 caracteres alfanuméricos.

- Las constantes al igual que los rótulos, se resuelven en la traducción y comparten la misma tabla. Es decir si existe un rótulos llamado "otro" que se le asigna la línea 10, no puede existir una constante llamada "otro", porque se tomará como símbolo duplicado. Nótese que los símbolos son un problema meramente del traductor, ya que no afectan a la ejecución.

- Generalizando, tanto los rótulos como las constantes se reemplazan como **valores inmediatos** dentro de las instrucciones pudiéndose utilizar indistintamente en cualquier instrucción que admita un argumento inmediato. Las constantes directas, si bien su valor es una dirección de memoria también se reemplazan en el código como un operando inmediato.

DETECCIÓN DE ERRORES EN SÍMBOLOS

El traductor debe ser capaz de detectar errores de **símbolos duplicados e indefinidos**. En ambos casos deberá informar el error, continuar con la traducción pero no generar la imagen del programa.

4 - OPERANDO INDIRECTO

La máquina virtual debe ser capaz de manejar un nuevo tipo de operando: el indirecto, que se codifica como 11 (binario).

Por ejemplo:

```
MOV AX , #1000
MOV BX , [DS:AX]
```

Almacena en BX el contenido de la dirección de memoria apuntada por AX (en este caso el valor de la celda 1000 del *data segment*).

Dependiendo del registro utilizando varia el registro base que se toma por omisión:

- Si se utilizan los registros AX a FX la dirección es relativa al DS por defecto, o al ES si se lo indica.
- Si se utilizan los registros BP o SP la dirección es relativa al SS.
- Los registros de de base son exclusivamente CS, DS, ES o SS, ningún otro registros puede ser utilizado como base. Es decir a la izquierda de los dos puntos
- No se permite el uso de los registros IP, CC, CS, DS, ES o SS como indirección. Es decir a la derecha de los dos puntos.
- El uso de indirecciones tomando como base CS es de solo lectura. Es decir que solo podrá estar en el segundo operando de las instrucciones.

Es decir que si se encuentra [AX] equivale a [DS:AX], y si se encuentra [BP] equivale a [SS:BP]. Si está permitido [CS:saludo] o [CS:AX] siempre y cuando solo se utilice como lectura.

Además, a este tipo de operando se le puede añadir una constante positiva o negativa (en base 10) o un símbolo.

Ejemplo 1:

```
        MOV CX , [AX+2]
```

Suponiendo que AX vale 1000, esta instrucción almacena en CX el contenido de la celda de memoria 1002 del *data segment*.

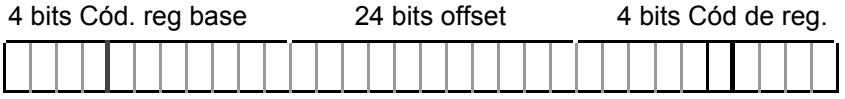
Ejemplo 2:

```
Vector EQU #100
        MOV DX , [ES:BX+Vector]
```

Suponiendo que BX vale 3, y ES=400, esta instrucción almacena en DX el contenido de la celda de memoria 503. Resumiendo, el formato de un operando indirecto es:

[{<segmento base>:}<registro> {+ / **-** <valor decimal>/<símbolo>}**]**
(las llaves indican partes opcionales)

En la celda de memoria donde se almacena el operando indirecto, se debe registrar información sobre el registro de segmento base, el registro de indirección y la constante. Para ello se usará el siguiente formato.



Por ejemplo, el operando [DS:CX+23], se debe codificar de la siguiente forma:

```
Segmento  DS:   0001                                  b
Constante 23:       0000 00000000 00000001 0111      b
Registro  CX:   ____________________________ 1100 b
                00010000 00000000 00000001 01111100 b (%1000017C)
```

5 - CADENAS DE CARACTERES

Se incorpora a la MV la posibilidad de trabajar con cadenas de caracteres (Strings) para ello se agregan nuevas instrucciones (SMOV, SCMP, SLEN) y nuevas interrupciones (SYS 10, SYS 20).

EN MEMORIA

Los strings, tal como se explicó previamente, se almacenan como secuencia consecutiva de caracteres en código ASCII, ocupando cada carácter una celda de memoria (de 32 bits) y se finaliza con con el carácter ´\0´ (es decir %00000000).

Además de las constantes strings, se pueden leer y almacenar variables strings en el *data segment* o *extra segment*. Solo las constantes string, definidas por EQU se ubican dentro del *code segment*, y como tal son de solo lectura.

Por ejemplo:

Posición de memoria relativa al origen del String	(+0)	(+1)	(+2)	(+3)	(+4)	(+5)	(+6)	(+7)	(+8)	(+9)	(+10)	(+11)
Valor Hexa en memoria (último bytes)	48	6F	6C	61	20	6D	75	6E	64	6F	21	00
Carácter	H	o	l	a	*espacio*	m	u	n	d	o	!	*fin*

INSTRUCCIONES

Las nuevas instrucciones a implementar son: SMOV, SCMP y SLEN.

Mnemónico	**Código Hexa**
SLEN	50
SMOV	51
SCMP	53

SLEN: Permite obtener la longitud de un String. Tiene 2 operandos, el primero es donde se guarda la longitud, el segundo es la dirección de memoria donde se encuentra el String. El primer operando puede ser de registro, directo o indirecto; el segundo operando solo puede ser **directo** o **indirecto**.

SMOV: Permite copiar un String de una posición de memoria a otra con la misma mecánica del MOV. Tiene 2 operandos, el primero indica la dirección de memoria destino y el segundo la dirección de memoria origen. Los operandos pueden ser **directos** o **indirectos**.

SCMP: Permite comparar 2 Strings, consiste en restar el carácter apuntado por el primer operando menos el carácter apuntado por el segundo operando, y modificar el CC acorde al resultado obtenido. Si el resultado es 0 continua con el segundo carácter. Finaliza cuando la resta resulta distinto de 0 o cuando haya llegado al final de una de las cadenas. Los operandos pueden ser **directos** o **indirectos**.

Ejemplo: `SCMP [AX],[BX]`

`[AX] ->"Hola"` `[BX] ->"HOLA"`	`[AX] ->"Oh"` `[BX] ->"Oh"`	`[AX] ->"Oh"` `[BX] ->"Oh..."`
"H" - "H" = %48 - %48 = 0 "o"-"O"= %6F- %4F= 32 (32=%20=´ ´)	"O" - "O" = %4F - %4F = 0 "h" - "h" = %68 - %68 = 0 "\0" - "\0" = %00 - %00 = 0	"O" - "O" = %4F - %4F = 0 "h" - "h" = %68 - %68 = 0 "\0" - "." = 0 - %2E = -46
CC = %0000	CC = %0001	CC = %8000

INTERRUPCIONES

A continuación se detallan 2 nuevas system calls para leer y escribir strings.

SYS 10 (STRING READ): Permite almacenar en un rango de celdas de memoria los datos leídos desde el teclado. Almacena lo que se lee en la posición de memoria apuntada por DX. tomando como base el Segmento indicado por BX (2=DS o 3=ES). En AX se puede especificar el si la lectura incluye promt o no, si en el bit 12 (%1000) hay 0 escribe el prompt, si hay 1 omite el promt.

Ej:

```
MOV DX, 123
MOV BX, 3
MOV AX, %0000
SYS 10
```

Por pantalla (Suponiendo que el ES = 434):
[0557]: Hola***<Enter>***

Leerá "Hola" y lo almacenará comenzando por la celda 557 (es decir la celda 123 del *extra segment*) donde colocará la H y en la celda 561 colocará el carácter de fin de string "\0".

SYS 20 (STRING WRITE): Permite imprimir por pantalla un rango de celdas donde se encuentra un String. Inicia en la posición de memoria apuntada por DX. tomando como base el Segmento indicado por BX (1=CS, 2=DS o 3=ES). En AX se puede especificar el si la escritura incluye promt o no, si en el bit 12 (%1000) hay 0 escribe el prompt, si hay 1 omite el promt. También se puede especificar en el bit 8 de AX (%0100) un 0 para que agregue un ***endline*** al final del string o un 1 para omitir el ***endline***.

6 - MANEJO DE LA PILA

El *Stack Segment* previamente mencionado será para uso exclusivo de la pila (stack) del proceso. La pila permite implementar de forma eficiente el trabajo con subrutinas: llamadas, retorno, pasaje de parámetros y recursividad.

REGISTROS

Para trabajar con la pila, además del registro SS que marca el comienzo del segmento, se utilizan los registros SP (Stack pointer) y BP (Base Pointer).

El SP se utiliza para apuntar al tope de la pila. Se inicializa con el tamaño de la pila. La pila va creciendo hacia las posiciones inferiores de memoria, por lo tanto el valor de SP se irá decrementando cuando se guarden datos en la pila y se aumenta cuando se sacan datos.

El registro BP, servirá para acceder a celdas dentro de la pila, haciendo uso del operando indirecto. Se puede utilizar para implementar pasaje de parámetros a través de la pila.

Cuando se utilicen los registros BP o SP en indirecciones, se deberá considerar el valor de SS como la dirección base para el cálculo de las direcciones efectivas, ya que ambos trabajan dentro del *stack segment*. Así como el IP siempre es relativo al CS.

Es decir, que para el modo de direccionamiento indirecto, los registros BP y SP deben trabajar con el registro SS, y no debe utilizarse otro registro de segmento como base de la indirección.

DETECCIÓN DE ERRORES PILA

El ejecutor (mcx) debe detectar en tiempo de ejecución los siguientes errores:

1. Stack overflow (desbordamiento de pila): se produce cuando se intenta insertar un dato en la pila y ésta está llena. Es decir cuando SP=0 y se hace el intento de agregar .
2. Stack underflow (pila vacía): se produce cuando se intenta sacar un dato de la pila y ésta está vacía. Es decir que SS+SP >= CS+PS.

En ambos casos se debe mostrar un mensaje por pantalla detallando el tipo de error y abortar la ejecución del proceso y todos los procesos siguientes (si los hubiera).

INSTRUCCIONES

Las nuevas instrucciones a implementar son:

Mnemónico	Código Hexa
PUSH	44
POP	45
CALL	40
RET	48

PUSH: Permite almacenar un dato en el tope de la pila. Requiere un solo operando que es el dato a almacenar. El mismo puede ser de cualquier tipo.

> PUSH AX // decrementa en uno el valor de SP y almacena en la posición apuntada por este registro el valor de AX.

POP: Extrae el dato del tope de pila y lo almacena en el operando (que puede ser registro, memoria o indirecto).

> POP [1000] // el contenido de la celda apuntada por SP se almacena en la celda 1000 y luego el valor de SP se incrementa en 1.

CALL: Permite efectuar un llamado a una subrutina. Requiere un solo parámetro que es la posición de memoria a donde se desea saltar (por supuesto, puede ser un rótulo).

CALL PROC1 // se salta a la instrucción rotulada como PROC1. Previamente se guarda en la pila la dirección memoria siguiente a esta instrucción (dirección de retorno).

RET: Permite efectuar una retorno desde una subrutina. No requiere parámetros. Extrae el valor del tope de la pila y efectúa una salto a dicha dirección.

7 - BREAKPOINTS

Los *breakpoints* son una herramienta esencial para el programador. Dado que los programas que se procesarán en esta segunda parte de la MV serán de mayor complejidad, resultará muy útil poder agregar *breakpoints* al código fuente para que detengan su ejecución y se pueda visualizar el estado de la Máquina Virtual.

Se implementan con la System Call "SYS 0" y no requiere configurar ningún registro.

Cuando se encuentra un *breakpoint* el comportamiento depende de los flags en los parámetros de invocación de la **mvx**.

- ***-b*** fuerza a la MV a detener su ejecución, muestra el prompt "cmd:" y solicita al usuario que ingrese un comando. Si se omite este flag, no se detiene la ejecución y el programa continúa ignorando el SYS 0.

  Si se encuentra el flag **-b** muestra:

```
[005] cmd:
```

  Donde [005] es el número de línea, y:
  - Si el usuario presiona Enter sin escribir nada, la MV continúa la ejecución.
  - Si el usuario escribe un número decimal entero positivo, la MV muestra el valor de la celda tomando el número ingresado por el usuario como la dirección absoluta de la memoria. y se muestra con el siguiente formato:

```
[005] cmd:302
[0302]: 0000 0000 A 0123456789
```

  Entre [ ] va la dirección de memoria en base 10, a continuación el valor de la celda en Hexadecimal, luego si el valor se corresponde con un caracter imprimible el carácter, sinó "." y finalmente el valor decimal de la celda.

  - Si el usuario escribe 2 numeros decimales positivos la máquina mostrará los valores del rango de celdas tomando las direcciones absolutas.

```
[005] cmd:302 305
[0302]: 0000 0000 A 0123456789
[0303]: 0000 0000 A 0123456789
[0304]: 0000 0000 A 0123456789
[0305]: 0000 0000 A 0123456789
```

- ***-c*** hace clear screen al iniciar la ejecución de la máquina virtual y cuando encuentra un *breakpoint*.

- ***-d*** hace el disassembler al inicio de la ejecución y cada vez que se ejecuta un breakpoint indicando la posición del IP. Es decir que vuelva a mostrar el código desensamblado, junto al valor de los registros y **señalando con ">" en la línea donde detuvo la ejecución**. Por ejemplo:

```
Código:
 [0019]: 0001 0100 0000 000E 0000 03E9  1: MOV        EX, 1001
 [0022]: 0001 0100 0000 000B 0000 0064  2: MOV        BX, 100
 [0025]: 0002 0101 0000 000E 0000 000B  3: ADD        EX, BX
 [0028]: 0001 0201 2000 0001 0000 000E  4: MOV    [DS:1], EX
>[0031]: 0081 0000 0000 0000 0000 0000  5: SYS         0
 [0034]: 0001 0100 0000 000A 0000 1101  6: MOV        AX, %1101
 [0037]: 0001 0100 0000 000D 0000 0001  7: MOV        DX, 1
 [0040]: 0001 0100 0000 000C 0000 0001  8: MOV        CX, 1
 [0043]: 0081 0000 0000 0002 0000 0000  9: SYS         2
Registros:
PS =       1524 | CS =         19 | DS =         46 | ES =        546 |
IP =         34 | SS =       1546 | SP =        500 | BP =          0 |
AC =          0 | CC =          0 | AX =          0 | BX =        100 |
CX =          0 | DX =          0 | EX =        100 | FX =          0 |
[005] cmd: 47
[0047]: 0000 0064 d 100
[005] cmd:_
```