TRABAJO PRÁCTICO

TRADUCTOR ASSEMBLER - MÁQUINA VIRTUAL

PARTE II

Introducción **2**

Programa y Proceso **3**
- Traductor 4
- Ejecutor 4

Símbolos **6**
- Constantes implícitas 6
- Constantes string 6
- Notas adicionales 7
- Detección de errores en símbolos 7

Operando indirecto **8**
- Interpretación 8
- Offset 8
- Sintaxis y formato 9
- Detección de errores de acceso a memoria 10

Memoria dinámica **10**

Cadenas de caracteres **15**
- En memoria 15
- Instrucciones 15
- Interrupciones 16

Gestión de la pila **17**
- REGISTROS 17
- Detección de error de pila 18
- INSTRUCCIONES 18

Adicionales **19**

Resumen de errores a detectar **19**

FORMATO DE ENTREGA Y EVALUACIÓN **20**

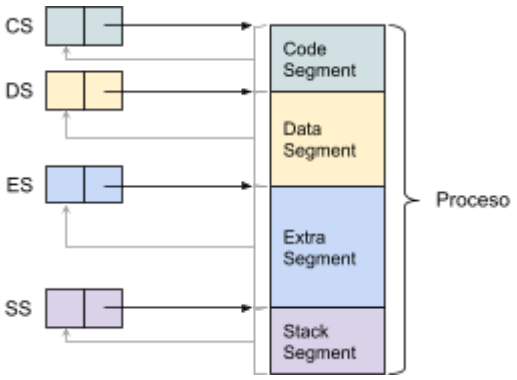
Introducción

En esta segunda parte del trabajo práctico se deberá ampliar la máquina virtual para que soporte trabajar con memoria secundaria (virtual disk drives), 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 al sistema, nuevas instrucciones y directivas al compilador, se incorporan 2 nuevos segmentos de memoria: “Extra Segment” y “Stack Segment”, y 6 nuevos registros (CS, ES, SS, SP, BP y HP).

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

- **Code Segment**: incluye el código fuente y constantes string, definido por el registro CS (ya no será necesariamente 0 (cero) la posición inicial).
- **Data Segment**: utilizado para los datos del proceso a disposición del programador, y definido por el registro DS.
- **Extra Segment**: reservado para el uso de memoria dinámica, definido por el registro ES.
- **Stack Segment**: segmento dedicado exclusivamente para la pila del proceso, definido por el registro SS.



Los registros quedarán dispuestos de la siguiente manera:

Posición	Nombre				Descripción
	32bits	16bits	byte alto	byte bajo	
0	DS	-	-	-	Segmentos
1	**SS**	-	-	-	
2	**ES**	-	-	-	
3	**CS**	-	-	-	
4	**HP**	-	-	-	HEAP
5	-	IP	-	-	Instruction Pointer
6	-	**SP**	-	-	Stack
7	-	**BP**	-	-	
8	CC	-	-	-	Condition Code
9	AC	-	-	-	Accumulator
10	EAX	AX	AH	AL	General Purpose Registers
11	EBX	BX	BH	BL	
12	ECX	CX	CH	CL	
13	EDX	DX	DH	DL	
14	EEX	EX	EH	EL	
15	EFX	FX	FH	FL	

Los registros **DS**, **ES**, **SS** y **CS** se dividen en una parte alta de 16 bits y una baja de 16 bits. La parte Alta de cada registro contiene el tamaño en celdas del segmento que definen, la parte baja la dirección absoluta de la memoria donde comienza el segmento. Nótese que con este diseño los segmentos no necesariamente deben estar contiguos ni ordenados en la memoria, y tampoco el proceso necesariamente ocupará toda la memoria disponible. Si bien la máquina virtual NO tendrá la capacidad para ejecutar concurrentemente múltiples procesos, este diseño lo permitiría sin problemas estando cada uno definido por sus registros.

El registro **HP** será utilizado para administrar la memoria dinámica, sobre el *Extra Segment*.

Los registros **SP** y **BP** serán registros que utilizará el programador exclusivamente para manejar la pila.

Los registros **IP**, **CC**, **AC** y **EAX** a **EFX** no cambian en su propósito respecto a la primera parte de la MV.

NOTA: En esta segunda parte tiene relevancia referirse la parte alta (primeros 2 bytes) o baja (últimos 2 bytes) de algún registro, para ello se utilizará el sufijo H (de High) o L (de Low) respectivamente. Por ejemplo si escribimos EAX(H) nos estaremos refiriendo a los primeros 2 bytes del registro EAX, o si por ejemplo decimos DS(L) nos estaremos refiriendo a los últimos 2 bytes del registro DS. Esto **no afecta al lenguaje ASM**, solo es una abreviatura coloquial para este documento.

Programa y Proceso

Como se vio en la primera parte **el programa** es el resultado de la traducción y punto de entrada del ejecutor (máquina virtual propiamente dicha). **Un proceso** es un programa en ejecución, por lo tanto cuando MV carga el programa en la memoria y configura los registros el mismo pasará a ser un proceso.

En esta segunda parte, el programa binario tendrá extensión mv2 (*.mv2) y se modifica el header para que pueda ser identificado como tal, además de proporcionar información sobre cómo deben estructurarse los segmentos.

Header (MV-2)	
Nº bloque	**Contenido**
0	“MV-2” 4 chars FIJO
1	Tamaño del DS
2	Tamaño del SS
3	Tamaño del ES
4	Tamaño del CS
5	“V.22” 4 chars FIJO

Traductor

En el tamaño de cada segmento se indica la cantidad de celdas que se deben reservar para el mismo, por defecto será 1024 a excepción del CS que está condicionado a la cantidad de líneas de código y constantes string del programa. Sin embargo, el traductor debe ser capaz de implementar directivas donde el programador puede definir el tamaño de los segmentos, las mismas se pueden realizar en cualquier línea del código (normalmente suelen ser las primeras), indicando con \\ al comienzo de la misma.

Sintaxis:

```
\\<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, y debe ser un números entero positivo entre 0 y 65535 (0xFFFF).

Ejemplo 1:

```
\\DATA 10
\\EXTRA 3000
\\STACK 5000
```

No necesariamente deben estar definidos los 3 segmentos, y tampoco en el mismo orden. La línea de directivas debe estar solo una vez en el código, no pudiendo haber ninguna otra instrucción o comentario en la misma línea. Cada directiva se separa por espacios en blanco o tabulaciones y por simplicidad no puede haber espacios entre el nombre del segmento, el signo igual y el tamaño. El tamaño del segmento, como se dijo, debe estar entre 0 y 65535 (0xFFFF) aunque el programador no debería designar un tamaño mayor a la memoria disponible, 65535 es el máximo teórico que la arquitectura permitiría como tamaño de segmento, sin embargo, la MV solo dispondrá de 8192 celdas y por lo tanto en la práctica la sumatoria del tamaño de los segmentos no podrá superar 8192 celdas menos el tamaño del Code Segment.

```
(DATA + EXTRA + STACK) <= (8192 - <TAMAÑO CODE SEGMENT>)
```

Ejecutor

El ejecutor debe leer el encabezado del archivo y:

- Determinar si la primera celda contiene “MV-2” y la última “V.22” para saber si puede ejecutarlo. Si no tiene este encabezado directamente interrumpe la ejecución con un mensaje diciendo que el formato de archivo x.mv2 no es correcto.
- Obtener el tamaño de cada segmento y validar si cabe en la memoria. Si no hay suficiente memoria interrumpirá la ejecución mostrará un mensaje diciendo que el proceso no puede ser cargado por memoria insuficiente.
- Una vez obtenidos y validados los datos utiliza los tamaños de los segmentos para cargar los registros DS, ES, SS y CS. En la parte alta del registro carga el tamaño y en la parte baja asigna una

posición de memoria con el siguiente criterio: Ubica el *Code Segment* en la posición 0, el *Data Segment* a continuación, luego el *Extra Segment* y finalmente el *Stack Segment*.

- Una vez finalizada la iniciación de los registros DS, ES, SS, CS, los demás registros se inicializan del siguiente modo:
  HP(H) = 2; HP(L) = 0
  IP(H) = 3; IP(L) = 0
  SP(H) = 1; SP(L) = SS(H)
  BP(H) = 1; BP(L) = No especificado.
  CC a EFX: No especificado.
- Finalmente cargará el *Code Segment* con el resto del archivo binario, ubicándolo en la posición de memoria designada.

Ejemplo 2:

Supongamos que el Header contiene la siguiente información:

Header		
Nº bloque	**Contenido (HEXA)**	**Significado**
0	4D562D32	"MV-2"
1	0000000A	10
2	00001388	5000
3	00000BB8	3000
4	00000019	25
5	562E3232	"V.22"

Se calcula: 10 + 3000 + 5000 + 25 = 8035
Como 8035 < 8192 (tamaño de la memoria), la MV puede cargar el proceso.

Los registros quedarían definidos así:

Registros		
Registro	**Contenido (HEXA)**	**Significado**
DS	000A0019	10 celdas, iniciando en 25
ES	0BB80023	3000 celdas, iniciando en 35
SS	13880BDB	5000 celdas, iniciando en 3035
CS	00190000	25 tamaño, inicia en 0

IMPORTANTE: Ahora para calcular una dirección de memoria relativa a algún segmento, deberá utilizarse la parte baja de cada uno. Por lo tanto cambia el cálculo de cada acceso a memoria, los operandos directos deberán utilizar DS(L) como base (en lugar de todo el DS) para calcular la dirección absoluta de memoria donde estará el dato.

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 implícitas

Una constante implícita se define por la directiva EQU.

Ejemplo 3:

```
    BASE   EQU  #16
```

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

Ejemplo 4:

```
    ADD EAX,BASE
```

Suma 16 al registro EAX, tomando en cuenta que los ejemplos 3 y 4 pertenecen al mismo código.

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 relativa al *Code Segment*.

Constantes string

Una constante string 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.

Ejemplo 5:

```
    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).

Ejemplo 6:

Si el código assembler tiene 24 líneas, ocupará de la celda 0 a la 23, la “H” (de Hola) se almacenará en la celda 24, y el “\0” en la celda 28. El carácter “C” de TEXTO2 se almacenará en la celda 29, y así sucesivamente con los demás caracteres y constantes.

Por lo tanto en la tabla de símbolos TEXTO1 tendrá el valor 24 y TEXTO2 el valor 29.

Es importante destacar que tanto las **constantes implícitas** como las **constantes string** tendrán diferencias para calcular su valor para la tabla de símbolos, pero la mecánica de reemplazo será la misma: cuando se exprese una constante como operando se reemplazará por el valor de la misma. Esto deja como responsable al programador de ASM utilizar las constantes de un modo adecuado.

Nótese que para acceder a constantes string **no podrá utilizar un operando directo**, debido a que los operadores directos solo pueden acceder a datos en el *Data Segment* (es decir que toma como base el DS(L)), por lo tanto el acceso a las constantes string deberá realizarse en forma indirecta utilizando un operando indirecto explicado en la siguiente sección (**Operando indirecto**).

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, a modo de no confundirse con registros.

- Las constantes al igual que los rótulos, se resuelven en la traducción y comparten la misma tabla. Es decir si existe un rótulo llamado “otro” que se le asigna la línea 10, no puede existir una constante llamada “otro”, porque se tomará como símbolo duplicado.

- 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 string, 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.

Operando indirecto

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

Ejemplo 7:

```
    MOV EAX , 5
    MOV EBX , [EAX]
```

Almacena en EBX el contenido de la dirección de memoria 5 del Data segment, o dicho de otro modo, la dirección de memoria apuntada por el registro AX.

Interpretación

Para interpretar una indirección por registro se debe utilizar la parte alta del registro (16 bits) para **identificar el código del segmento** al que quiere referenciar, en el ejemplo EAX = 0x00000005 resulta ser EAX(H) = 0x0000, (es decir 0, que el código del reg DS) entonces la base será DSL (DS Low). AX = 0x0005, por lo tanto si DS = 0x000A0019 (10|25), la dirección absoluta de la memoria será 25 + 5 = 30.

Esta mecánica, es lo que permite acceder a las celdas de memoria del *Extra Segment* o del *Stack Segment*. Si bien más adelante se va a especificar el uso, la forma de interpretar una indirección será la misma. Por ejemplo [EDX] siendo EDX = 0x0002000A, es decir que EDX(H)=2 (código del reg ES) y DX=10 (desplazamiento dentro del segmento), si el registro ES= 0x0BB80023 (3000|35) entonces la celda memoria absoluta apuntada por [DX] será 35 + 10 = 45.

Para el caso de las constantes string, necesariamente se deberán acceder utilizando este operando, como se muestra en el ejemplo 5.

Ejemplo 8:

```
TEXTO1  EQU  "Hola"

        LDH   3
        LDL   TEXTO1
```

En este fragmento de código se configuró el registro AC = 0x00030018 (3|24), por lo tanto [AC] apuntará a la celda 24 ya que si se toma AC(H) = 3 (3 es el código del CS) y CS(L) = 0, CS(L) + AC(L) = 24. [**NOTA**: hay que tomar en cuenta que el diseño debe realizarse considerando que el CS podría estar en cualquier parte de la memoria].

Offset

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

Ejemplo 9:

```
        MOV ECX , [EAX+2]
```

Suponiendo que EAX=5, esta instrucción almacena en ECX el contenido de la celda de memoria 7 del *data segment*.

Ejemplo 10:

```
Vector   EQU #100
         MOV EDX , [EBX+Vector]
```

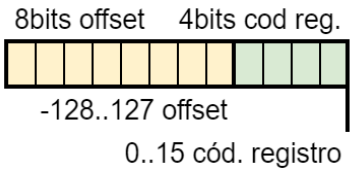
Suponiendo que EBX(H)=2, BX=10, ES(H)=3000, ES(L)=35, esta instrucción almacena en EDX el contenido de la celda de memoria 145 del Extra Segment (35+10+100).

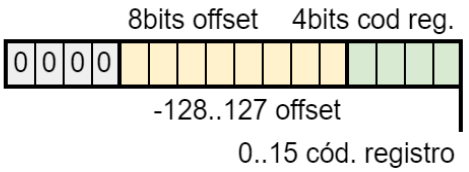
Sintaxis y formato

El formato de un operando indirecto es:

[<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 indirección y la constante. Para ello se usará el siguiente formato:





Tanto para operandos de 12 bits como de 16 bits el operando indirecto **siempre** utiliza 12 bits. Los primeros 8 bits serán para el offset y los últimos 4 para el registro.

Además solo se utilizan 4 bits para el código de registro, por lo tanto solo se puede indireccionar con registros completos (AC, EAX, EBX, etc…) y no con sub-registros (como AH, o BL). No obstante por compatibilidad, se aceptará escribir los sub registros AX, BX, CX, DX, EX y FX; como equivalentes a los extendidos: EAX, EBX, ECX, EDX, EEX y EFX. Es decir por ejemplo si se escribe [AX+3] equivale a [EAX+3] teniendo ambos la misma codificación: 0x03A.

Detección de errores de acceso a memoria

En esta segunda parte el ejecutor deberá detectar un error de acceso a memoria (Segmentation Fault). El mismo se produce cuando un acceso a memoria, directo o indirecto, calcula la dirección de una celda que no se encuentra en el rango del segmento referenciado.

Ejemplo 11: suponiendo que DS(H)=10, DS(L)=25, ES(H)=3000, ES(L)=35 y EAX=25

```
MOV EBX,[EAX-100]   << Error! porque se quiere acceder a una dirección inferior al DS(L)

LDH 2
LDL 3001
MOV EBX,[AC]        << Error! AC tiene cargado el segmento 2 e intenta acceder a la celda 3001
                       cuando el Extra Segment tiene tamaño para 3000 celdas.
```

También devolverá este error cuando se intente utilizar de base un registro inexistente. De este modo cuando se asigna -1 a un registro (para indicar NULL) y el mismo se utiliza como indirección, se interrumpirá la ejecución por ***Segmetation Fault***.

Memoria dinámica

Para facilitar el uso de memoria dinámica, la MV proporcionará el registro HP, el cual se iniciará al comienzo de la ejecución, junto con el IP, SP, y los demás registros de segmentos.

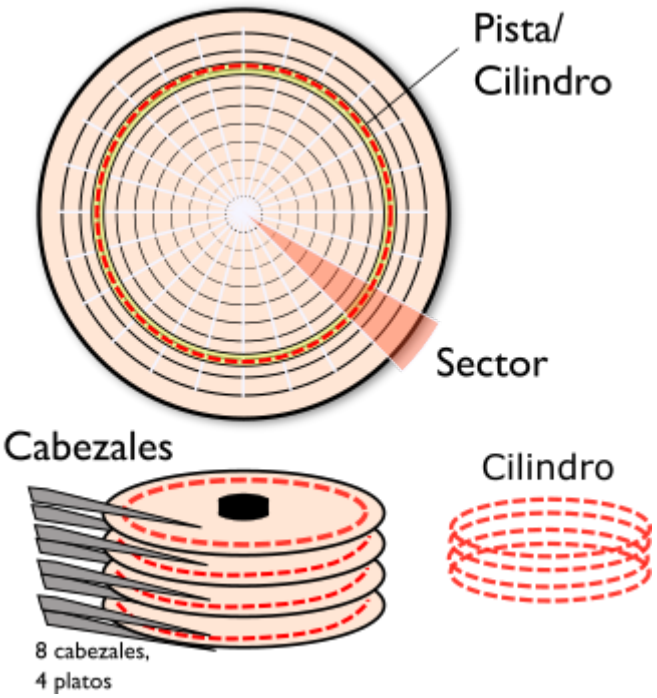
Inicialmente HP tendrá en la parte alta el valor 2 correspondiente al Extra Segment, y en la parte baja el valor 0 (cero). Por lo tanto HP = %00020000.

Luego de la inicialización, será responsabilidad del programador el uso adecuado del registro.

Memoria secundaria (trabajar con dispositivos)

La MV recibe por la línea de comandos una lista de archivos *.vdd (virtual disk drive) que representarán unidades de discos. Cada unidad de disco se numera desde **0** hasta **N** según el orden en el listado de archivos que se conforman en la línea de comandos (Máximo 255 archivos, límite teórico).

Cada disco se compone de Cilindros, Cabezas y Sectores:



Formato de los archivos VDD

Cada archivo *.vdd contendrá una cabecera (*header*) de 512 bytes con la información necesaria para su interpretación:

Bytes	Contenido
4	**“VDD0” 4 caracteres identifica el tipo de archivo. [FIJO: 0x56444430]**
4	**Número de versión, llevará un 1 entero sin signo. [FIJO: 1]**
16	GUID - Identificador del disco. [ej: 0x7ee914b137774e84b31387ee9bed04a2]
4	Fecha de creación: año(2B), mes(1B), día(1B)
4	Hora creacion: hora(1B), Minutos(1B), Segundos(1B), Décimas(1B)
1	**Tipo, 0: estático, 1: dinámico [FIJO: 1]**
1	Cantidad de cilindros, entero sin signo [default: 128]
1	Cantidad de cabezas, entero sin signo [default: 128]
1	Cantidad de sectores, entero sin signo [default: 128]
4	**Tamaño del sector en bytes, entero sin signo. [FIJO: 512]**
211	Relleno, reservado para uso futuro [default: 0]

Se considerará que el tipo de disco siempre será 1(dinámico) y cada sector contendrá siempre 512 Bytes de información, que constituyen la mínima cantidad de información que se puede obtener de un disco.

Si el *.vdd no existe, la MV debe crearlo y lo hará con los valores por defecto. Por lo tanto un disco virtual creado por defecto tendrá 128 * 128 * 128 * 512 bytes = $2^7 * 2^7 * 2^7 * 2^9$ bytes = 2^{30} bytes = 1 GiB.

El identificador de disco GUID se genera de forma aleatoria.

La máquina virtual deberá guardar la información de cada disco virtual: número unidad, archivo físico real, cantidad de cilindros, cantidad de cabezas, cantidad de sectores y tamaño del sector.

Se toma la siguiente convención: los cilindros crecen de la periferia hacia el centro, las cabezas se numeran de arriba hacia abajo, y los sectores en sentido horario.

En el archivo la información se guarda en forma binaria y contigua, a continuación del header en bloques de 512 bytes ordenados por cilindro, cabeza y sector (CHS).

Es decir que si el disco tiene 128 cilindros, 8 cabezas, 128 sectores y se quiere acceder a la cilindro 2, cabeza 8, sector 10; se deberá hacer un seek de: **17307136 Bytes** (fseek(disco, 17307136 , SEEK_SET);)

Calculado del siguiente modo:

HD + c * C * S * TS + h * S * TS + s * TS

Donde:

HD = Tamaño del header
c = número de cilindro (inicia en 0)
h = número de cabeza (inicia en 0)
s = número de sector (inicia en 0)
C = Cantidad de cilindros
S = cantidad de sectores
TS = Tamaño del sector en Bytes

512 Bytes + 2 * 128 * 128 * **512 Bytes** + 8 * 128 * **512 Bytes** + 10 * **512 Bytes** =
512 Bytes + 16777216 **Bytes** + 524288 **Bytes** + 5120 **Bytes** =
17307136 Bytes

Si cuando se accede a leer o escribir en el disco, el archivo es de menor tamaño se lo extiende, sin devolver error a la MV, Es decir que el tamaño del disco virtual es fijo para la máquina virtual pero dinámico para máquina real.

Cada acceso de escritura en un disco virtual, deberá corresponderse con la escritura en el archivo real al que está vinculado.

Acceso al disco desde al MV

El acceso a la información del disco se realizará mediante la **llamada al sistema 13 (%D)**. Por medio de esta interrupción se podrá consultar el estado del disco, cargar ***n*** sectores a un buffer de memoria o descargar ***n*** sectores desde un buffer de memoria.

El buffer de memoria no es más que un rango de celdas de memoria (en el *Data Segment* o *Extra Segment*) en donde se cargará, o descargará, la información del disco.

La interrupción requiere el seteo de los siguientes registros/subregistros:

reg	*Parámetros de entrada*	*Resultados de salida*

AH	Se indica el número de operación:
	`%00 Consultar último estado`
	`%02 Leer del disco`
	`%03 Escribir en el disco`
	`%08 Obtener los parámetros del disco`
		Código de estado
		`%00 Operación exitosa`
		`%01 Función inválida (parám AH)`
		`%04 Error de lectura`
		`%0B Número inválido de cilindro`
		`%0C Número inválido de cabeza`
		`%0D Numero invalido de sector`
		`%31 No existe el disco`
		`%CC Falla de escritura`
		`%FF Falla en la operación`

AL	Se indica la cantidad de sectores a leer/escribir	Cantidad de sectores transferidos
CH	Se indica el número de cilindro	Cantidad de cilindros (caso de %08)
CL	Se indica el número cabeza	Cantidad de cabezas(caso de %08)
DH	Se indica la sector	Cantidad de sectores(caso de %08)
DL	Se indica el número de disco (*.vdd)	-
EBX	Se indica la primer celda del buffer de lectura/escritura	-

Ejemplo 12: se solicitan leer 3 sectores de la unidad de disco 0 a partir del cilindro 8, cabeza 2, sector 10 y se debe almacenar en el *Extra Segment* a partir de la celda 64.

```
ldh       2        ; Código del ES
ldl      64        ; offset del ES
mov     EBX, AC    ; Dirección del buffer de lectura
mov      AH, %02   ; Leer del disco
mov      AL, 3     ; 3 sectores a leer
mov      CH, 2     ; Cilindro 2
mov      CL, 8     ; Cabeza 8
mov      DH, 10    ; Sector 10
mov      DL, 0     ; Unidad de disco 0
sys      %D        ; Indica que se realiza la lectura
```

Una vez finalizada la lectura, se rellenaran 384 celdas, a partir de la celda 64 del Extra Segment:

512 bytes/sector * 3 sectores / 4 bytes/celda = 384 celdas

Lo mínimo que se puede leer o escribir del disco es 1 sector, por lo tanto son 512 bytes que se corresponde con 128 celdas de 4 bytes de la memoria principal. Tanto para la lectura como la escritura las celdas deben estar contiguas en la memoria y debe hacer espacio en el segmento en el que se escriben, en caso contrario se devolverá error **%04 "Error de lectura"** si estaba leyendo, o **%CC "Falla de escritura"** si estaba escribiendo.

Siempre que se ejecute una de las funciones de la interrupción 13 (%D), a excepción de función "Obtener los parámetros del disco" (%08), deberá chequear que:

- la unidad de disco exista (DL),
- el número de cilindro no supere el máximo (CH),
- la cantidad de cabezas no supere el máximo (CL),
- el número de sector no supere el máximo (DH)

En caso de error debe devolver el código correspondiente en el registro AH: **%31, %0B, %0C, %0D**. Si se invoca una función fuera de las 4 permitidas (**%00, %02, %03, %08**) devolverá error **%01 "Función inválida"**.

En el caso de la función "Obtener los parámetros del disco" (%08), solo devolverá el error %31 **"No existe el disco"** en AH, cuando el disco en cuestión no exista.

Ejemplo 13: se solicita la información del disco 1.

```
mov     AH, %08   ; Obtener los parámetros del disco
mov     DL, 1     ; Unidad de disco 0
sys     %D        ; Indica que se realiza la lectura
```

Finalizada la interrupción completará los registros CH, CL y DH, con la cantidad de cilindros, cabezas, y sectores del disco (respectivamente) obteniendo esa información de la MV y en el registro AH dejará %00, indicando operación exitosa; en caso contrario los registros CH. CL, DH no se modifican y en AH dejará %31 indicando que no existe el disco.

Cuando se solicita escribir o leer ***N*** sectores y la cantidad de sectores supera a las del cilindro/pista, continua por el sector 0 de la siguiente cabeza (se evita mover los cabezales a otro cilindro); así mismo si se supera la cantidad de cabezas, continuará leyendo desde el siguiente cilindro con la cabeza número 0 y sector 0.

Si una lectura supera el tamaño del disco, se especificará la cantidad de sectores transferidos en el **registro AL** y se considerará que la operación fue exitosa cargando en el registro AH el valor **%00 Operación exitosa.** Si se intenta leer un sector inexistente en el archivo *.vdd, pero dentro del rango de los parámetros del disco, se devuelve 0 y se extiende el archivo.

Si una escritura supera el tamaño del disco, se cargará el registro AH con el error **%FF "Falla en la operación".**

Este último error (%FF) también se utilizará para cualquier error arrojado por el sistema real al intentar leer o escribir.

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 los System Calls: SYS %3 y SYS %4.

En memoria

Los strings, tal como se explicó previamente, se almacenan como secuencia consecutiva de caracteres en código ASCII, ocupando cada carácter el byte menos significativos de una celda de memoria (de 32 bits) y se finaliza 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.

Ejemplo 14:

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 byte)	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	C
SMOV	D
SCMP	E

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 15:

```
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 = %00000000	CC = %00000001	CC = %80000000

Interrupciones

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

SYS %3 (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 EDX. tomando como base el Segmento indicado por el mismo registro en la parte alta (al igual que en toda indirección). En AX se puede especificar si la lectura incluye promt o no, si en el bit 11 (%800) hay 0 escribe el prompt, si hay 1 omite el promt. En CX se de especificar la cantidad máxima de caracteres a leer.

Ejemplo 16:

```
MOV EDX, 123
MOV CX, 50
MOV AX, %000
SYS %3
```

Por pantalla (Suponiendo que el *Code Segment* ocupa 100 celdas):
[0223]: Hola***<Enter>***

Leerá “Hola” y lo almacenará comenzando por la celda 123 del *Data Segment* donde colocará la H y en la celda 127 colocará el carácter de fin de string “\0”. NOTA: si en CX hubiera puesto 4 (en lugar de 50) habría almacenado “Hol” poniendo en la celda 126 “\0”.

Ejemplo 17:

```
MOV CX, 50
MOV EDX, HP
ADD HP, CX
MOV AX, %000
SYS %3
```

En este ejemplo (17) combina el uso de la interrupción %3 con la memoria dinámica. Si el registro HP apunta a la última celda libre del *Extra Segment*, un modo simple para obtener memoria dinámica es

obtener su valor e incrementarlo por el espacio requerido. Previamente la dirección del HP se guarda en EDX el cual, junto con CX, será utilizado por la interrupción %3 para escribir los caracteres ingresados por el usuario.

SYS %4 (STRING WRITE): Permite imprimir por pantalla un rango de celdas donde se encuentra un String. Inicia en la posición de memoria apuntada por EDX. En AX se puede especificar si la escritura incluye promt o no, si en el bit 11 (%800) hay 0 escribe el prompt, si hay 1 omite el promt. También se puede especificar en el bit 8 de AX (%100) un 0 para que agregue un ***endline*** al final del string o un 1 para omitir el ***endline***.

Gestión 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 contiene la cantidad de celdas y comienzo del segmento (en su parte alta y baja respectivamente), se utilizan los registros SP (*Stack pointer*) de 16 bits y BP (*Base Pointer*) también de 16 bits. Siempre que estos registros se utilicen en indirecciones se considerarán relativos al SS.

NOTA: Para garantizarlo, esta máquina virtual, guardará en la parte alta de los registros SP y BP (SP(H) y BP(H)) el código 1 correspondiente al SS, de modo tal que siguiendo las mismas reglas de direccionamiento se pueda acceder a la pila.

El SP se utiliza para apuntar al tope de la pila. Se inicializa con el tamaño de la pila, o sea SS(H). 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 error de 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 SPL=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 SP(L) >=SS(H).

En ambos casos se debe mostrar un mensaje por pantalla detallando el tipo de error y abortar la ejecución del proceso.

INSTRUCCIONES

Las nuevas instrucciones a implementar son:

Mnemónico	Código Hexa
PUSH	FC
POP	FD
CALL	FE
RET	FF0

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 un salto a dicha dirección.

Adicionales

Adicionalmente se agrega un SystemCall **SYS %7** que ejecuta un clear screen, no requiere ningún registro configurado, y tampoco modifica ninguno.

También se completa la especificación de la instrucción **RND** que consiste en cargar en AC un valor aleatorio entre 0 y valor del operando.

Resumen de errores a detectar

En resumen se deberán detectar los siguientes errores:

Traducción

- **Valores apropiados en directivas:** debe verificar que los valores puestos por el usuario sean coherentes a la arquitectura (NO valida el tamaño de la memoria)
- **Mnemónico desconocido:** Al igual que en la primera parte, si no puede interpretar un mnemónico lo informa y no crea el archivo .bin, pero continúa la traducción.
- **Símbolo duplicado:** cuando un rótulo o constante definido ya se encuentra en la lista de símbolos.
- **Símbolo inexistente:** cuando un rótulo o constante utilizado no se encuentra en la lista de símbolos.
- **(Warning) inmediato truncado:** cuando se pone un valor inmediato como operando de una instrucción y el mismo debe ser truncado para poder codificar la instrucción. Considerar que ahora puede suceder un truncamiento por el uso de símbolos.

Ejecución

- **Validar programa binario:** Cuando comienza la lectura del binario debe verificar que el formato del *header* sea válido, si no lo es, interrumpe la ejecución informando.
- **Memoria insuficiente:** el formato puede ser válido, y aún así cuando se arma el proceso puede no entrar en la memoria. En este caso se informa al usuario y se corta la ejecución.
- **Segmentation fault:** Cuando se quiere acceder a una celda fuera del segmento o cuando el segmento para el cálculo de dirección es incorrecto.
- **Stack overflow:** Es similar al *Segmentation fault*, pero en el caso en que se quiera hacer un PUSH o CALL y la pila esté llena.
- **Stack underflow:** Es similar al *Segmentation fault*, pero en el caso en que se quiera hacer un POP o RET y la pila esté vacía.

FORMATO DE ENTREGA Y EVALUACIÓN

El día de la evaluación 30/MAY/2022 o 01/JUN/2022

- Cada grupo deberá tener en su PC **los ejecutables MVC.EXE y MVX.EXE** compilados y funcionando por consola, y el código fuente de ambos abierto para consultar.
- Cada grupo será sorteado para definir en qué orden será evaluado.
- Se entregará a cada grupo un conjunto ejemplos (archivos *.asm) que deben ser ejecutados **por consola**.
- Los docentes podrán hacer consultas a cualquier miembro del grupo sobre algún aspecto del código fuente sobre cualquiera de los dos programas.
- Si alguno de los códigos evaluados no presenta los resultados correctos, el grupo podrá hacer ajustes a sus programas en lo que quede de tiempo hasta evaluar al resto de los grupos, en una posible segunda vuelta.
- Los grupos que deban realizar ejercicios funcionando de la primera parte podrán recuperarse en esta instancia.
- El resultado de los ejercicios de la segunda parte será evaluado por los docentes a fin de decidir si el grupo tiene aprobado el práctico, de no ser así el grupo se deberá volver a presentar en la instancia de recuperatorio.
- Se deberá entregar via Moodle, un archivo Zip, Rar o 7zip con los fuentes y programas ejecutables compilados (para WIN32 o Linux), el mismo día de la evaluación antes de comenzar y al finalizar la misma se entregará sólo la última versión.