TRABAJO PRÁCTICO

TRADUCTOR ASSEMBLER - MÁQUINA VIRTUAL

PARTE II

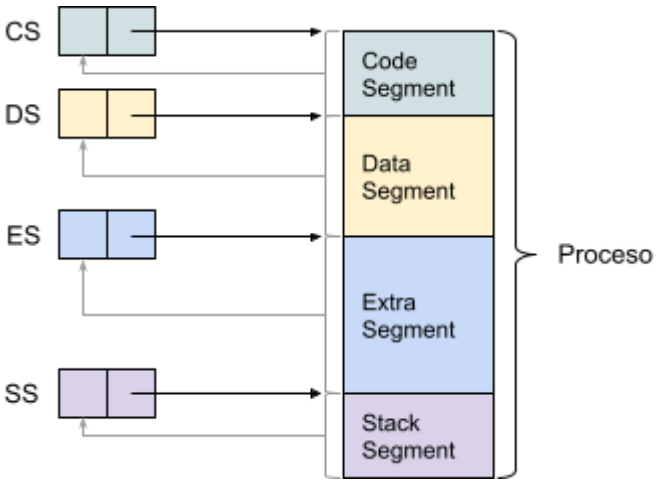
Introducción

En esta segunda parte del trabajo práctico se deberá ampliar la máquina virtual para que soporte instrucciones para el manejo memoria dinámica, 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 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:

Código	Nombre	Descripción
0	DS	Segments
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	AX	General Purpose Registers
11	BX	
12	CX	
13	DX	
14	EX	
15	FX	

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.

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

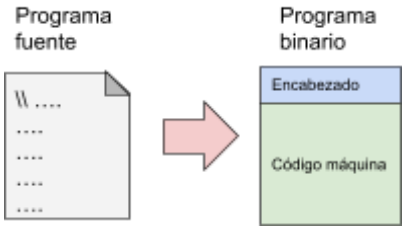
Los registros **IP**, **CC**, **AC** y **AX** a **FX** no cambian en su propósito respecto a la primera parte de la MV.

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 AXH nos estaremos refiriendo a los primeros 2 bytes del registro AX, o si por ejemplo decimos DSL 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, producto de la traducción, tendrá, además del código máquina, información en una cabecera (*header*) para poder identificarlo como tal. De modo que el ejecutor examinará el *header* para determinar si es capaz de ejecutar ese programa y configurar el espacio de memoria asignado a cada segmento. Esta configuración es equivalente a la de un programa ejecutable en cualquier sistema operativo.



El *header* se compone de 5 bloques de 4 bytes, para respetar el formato equivalente a 5 instrucciones. A partir del 6to bloque se encuentra el código (incluyendo las constantes de String) que se deberá cargar íntegramente en el Code Segment.

Header	
Nº bloque	**Contenido**
0	“MV21” 4 chars FIJO
1	Tamaño del DS
2	Tamaño del SS
3	Tamaño del ES
4	Tamaño del CS

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 suele ser la primera), indicando \\ASM al comienzo de la misma.

Sintaxis:

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

Ejemplo 1:

```
\\ASM 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 “MV21” para saber si puede ejecutarlo. Si no tiene este encabezado directamente interrumpe la ejecución con un mensaje diciendo que el formato de archivo x.bin 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*.
- Finalmente cargará el *Code Segment* con el resto del archivo binario, ubicándolo en la posición de memoria designada.

Ejemplo 2:

Header		
Nº bloque	**Contenido (HEXA)**	**Significado**
0	4D563231	“MV21”
1	0000000A	10
2	00001388	5000
3	00000BB8	3000
4	00000019	25

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

Una vez finalizada la iniciación de los registros, los demás registros se inicializan en 0, a excepción del HP que se iniciará con -1 (0xFFFFFFFF). El registro IP técnicamente debería inicializarse con la parte baja del registro CS (que indica el comienzo del *Code Segment*), sin embargo por definición en esta MV el valor será igualmente cero.

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 deberá utilizar DSL 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 AX,BASE
```

Suma 16 al registro AX, 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 DSL), 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 rótulos 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 AX , 5
MOV BX , [AX]
```

Almacena en BX 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 AX = 0x00000005 resulta ser AXH = 0x0000, (es decir 0, que el código del reg DS) entonces la base será DSL (DS Low). AXL = 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 [DX] siendo DX = 0x0002000A, es decir que DXH=2 (código del reg ES) y DXL=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 ACH = 3 (3 es el código del CS) y CSL = 0, CSL + ACL = 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 CX , [AX+2]
```

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

Ejemplo 10:

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

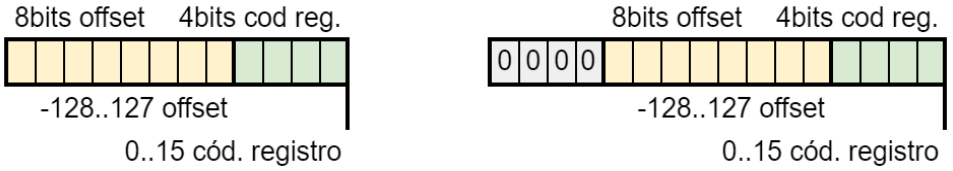
Suponiendo que BXH=2, BXL=10, ESH=3000, ESL=35, esta instrucción almacena en DX el contenido de la celda de memoria 145 (35+10+100).

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.

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 DSL=25, DSH=10, ESL=35, ESH=3000 y AX=25

MOV BX,[AX-100]	**<< Error!** porque se quiere acceder a una dirección inferior al DSL
LDH 2 LDL 3001 MOV BX,[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

La máquina virtual deberá ser capaz de administrar memoria dinámica. Sin bien en lenguajes convencionales la memoria dinámica es administrada por el mismo proceso (generalmente utilizando funciones de librerías o definidas en el propio lenguaje y creadas por el compilador) en la máquina virtual, por simplicidad, haremos que sea la misma MV que mediante Llamadas al Sistema sea capaz de asignar un espacio de memoria, solicitando una cantidad de celdas (NEW) o liberar un un espacio previamente solicitado (FREE).

Implementación

Como se mencionó previamente, se solicita memoria y se libera por medio de llamadas al sistema:

SYS	Significado	Descripción
%5	NEW	Requiere en CX la cantidad de celdas que se solicitan y devuelve en DX un puntero a la primera celda para su uso dentro del **ES**. Si no hay memoria disponible devuelve DX = -1 (NULL).
%6	FREE	Libera la memoria indicada en DX.

La memoria dinámica, si el programador decide utilizarla, se aloja en el *Extra Segment*. Y se implementa utilizando 2 listas circulares ordenatas: una lista con las celdas utilizadas y otra con las celdas libres. Los nodos de cada lista son celdas de memoria en el *Extra Segment* y son apuntados utilizando el registro HP: El HPH (HP High) apunta a la lista libres y el HPL (HP Low) apunta a la lista de espacios ocupados.

Cada nodo de las listas tiene un header de una celda y el resto de las celdas se utilizan para el contenido. El header se divide en dos: en la parte alta se guarda la cantidad de celdas de memoria que conforman el bloque alojado y en la parte baja el puntero al siguiente nodo.

Al iniciar el registro HP será igual 0xFFFFFFFF (en el contexto de la máquina virtual -1 se considera equivalente NULL). Con la primera invocación a una SYS %5 asumirá que tiene a su disposición el *Extra Segment* entonces inicializará la lista de libres en la posición de memoria 0 relativa al *Extra Segment* (abreviando ES[0]), poniendo en la parte Alta de ES[0] el tamaño del *Extra Segment* -1 y en la parte baja un puntero a sí misma (0) y setea el HPH = 0 y el HPL = -1. Acto seguido ejecuta el algoritmo de asignación.

Algoritmo de Asignación de memoria (NEW)

Cuando se invoca un SYS %5 se debe recorrer la lista de espacios libres apuntada por HPH comparando el tamaño de cada nodo y se elige el primer nodo que cumple con el tamaño solicitado indicado en CX, o tiene un tamaño mayor.

Si el tamaño es el mismo, el nodo libre pasará a la lista de utilizados. Si el nodo tiene mayor capacidad de espacio, se divide en 2, uno con la cantidad de memoria solicitada + 1 (para el header) que pasará a la lista de utilizados y con el resto del espacio libre se crea un nuevo nodo en la lista de libres.

El nodo nuevo se debe enlazar, en forma ordenada por posición de memoria, en la lista de espacios utilizados apuntados por HPL.

El nuevo nodo libre se debe enlazar, en forma ordenada por posición de memoria, a la lista apuntada por HPH.

Al finalizar HPH quedará apuntando al primer nodo de la lista de libres (es decir el que tiene la dirección más baja en la memoria) y el HPL quedará enlazado al nodo recientemente asignado.

Finalmente asigna en DXH = 2 (código del ES) y en DXL = HPL + 1 (es decir la dirección siguiente al header del nodo).

Ejemplo 12:

En la figura 1.a se muestra un fragmento de un Extra Segment. La lista de libres (celeste) está compuesta por 3 nodos: El primero en la posición ES[0] el segundo en la posición ES[2] y el tercero en la posición ES[18]. La lista de utilizados (en rojo) está compuesta por 2 nodos: el primero comienza en ES[3] y el segundo en ES[13]. Se puede ver como la cabecera de cada nodo tiene la cantidad de celdas del nodo en la parte alta de la celda y en la baja la dirección relativa el ES del siguiente nodo, por ejemplo en el primer nodo de la lista libres (ES[0]) tiene capacidad = 2 y el siguiente = 9. En esta situación se solicitan 3 celdas de memoria. El algoritmo arranca con ese primer nodo, ve que tiene capacidad para 2 entonces continua por el siguiente. El segundo nodo tiene capacidad 3, como coincide con la cantidad de memoria solicitada es el candidato. Entonces al nodo anterior (ES[0]) le asigna el puntero siguiente del candidato, por lo tanto 18 pasa a ser el siguiente de ES[0] (como se ve en la figura 1.b). Luego debe recorrer la lista de utilizados para encontrar el nodo anterior: ES[3] y asignar al nodo candidato ES[9] el siguiente del anterior, es decir 13 y al anterior le debe asignar como siguiente el candidato, es decir 9, como se muestra en la figura 1.c. Finalmente devuelve en DX la celda siguiente al header del candidato asignando en DXH = 2 y en DXL = 10.

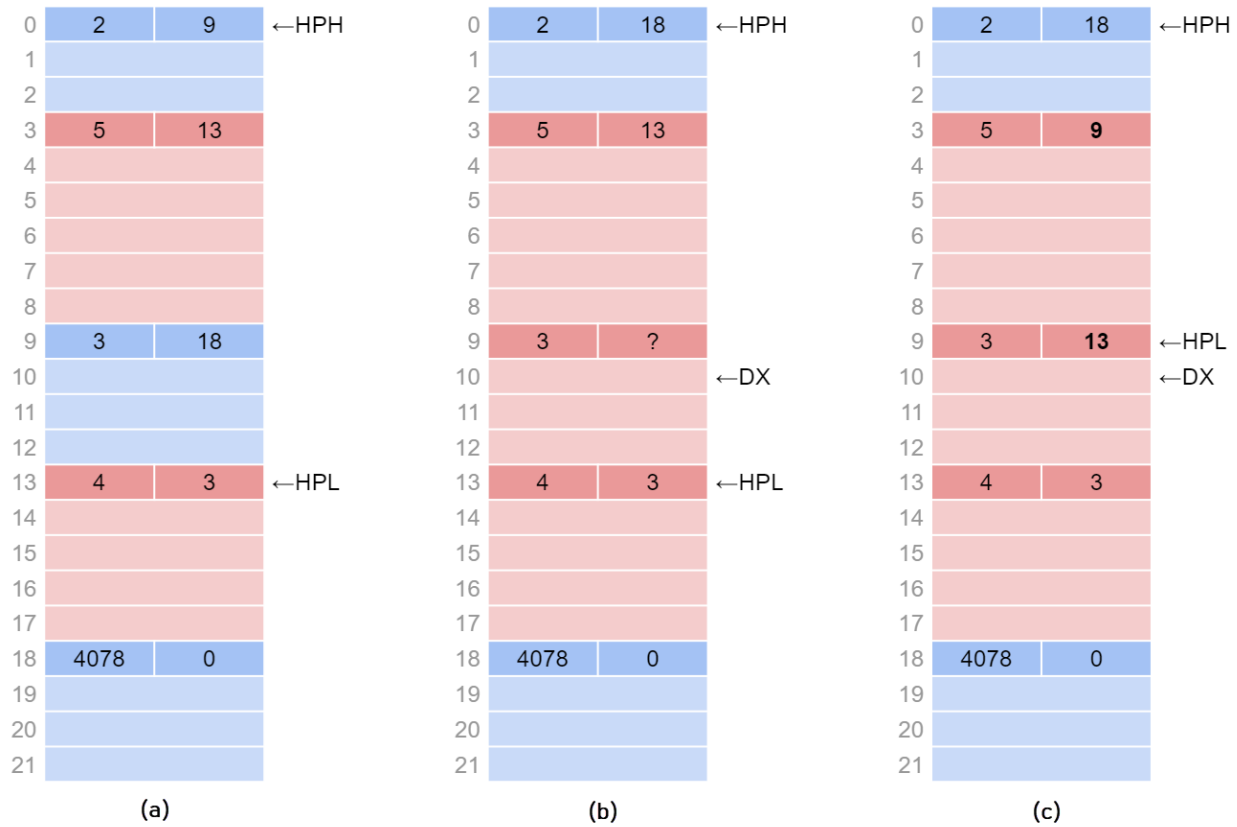


Figura 1

Algoritmo de Liberación de memoria (Free)

Cuando se invoca un SYS %6 se debe recorrer la lista de espacios libres ocupados por HPL buscando el nodo que contenga la dirección de memoria apuntada por DX, quitarlo de la lista de utilizados y enlazarlo en la lista de nodos libres.

Puede suceder que al enlazar el nodo en la lista de espacios libres el mismo quede contiguo en la memoria a otro nodo libre (recordar que la lista se debe mantener ordenada). Si sucede este caso se deben unificar ambos nodos creando uno con la capacidad de ambos +1 (por el header que desaparece). Este proceso, conocido como compactación, se debe repetir hasta que ningún nodo de la lista de libres esté contiguo a otro.

Ejemplo 13:

En la figura 2.a se muestra una situación inicial. Se solicita liberar la celda ES[4], el algoritmo debe comenzar a buscar por el nodo apuntado por HPL, que en la figura se encuentra en ES[13]. Como 13 es mayor que 4, ese nodo no contiene la dirección de memoria a liberar, entonces debe continuar por el siguiente ES[3]. El nodo ES[3] tiene cantidad 5 y como 3 es menor que 4 y 4 es menor o igual a (3+5) sabrá que ese es el nodo a liberar. A continuación debe recorrer la lista de libres (desde HPH) hasta encontrar el nodo libre anterior al nodo a liberar, en este caso ES[0] y debe enlazarlo a la listas poniendo como siguiente del actual el puntero del siguiente del anterior y como siguiente del anterior la posición del actual, es decir el 9 de la parte alta de ES[0] pasará a la parte alta de ES[3] y la parte alta de ES[0] pasará a tener 3; como se muestra en la figura 2.b. Finalmente como se puede ver quedan 3 nodos de memoria libre continuos entonces se procederá a la compactación: Primero funcionando el primer nodo con el segundo y luego repitiendo la operación otra vez hasta que no haya ningún nodo de la lista de libres continuos, dejando las listas como se muestran en la figura 2.c.

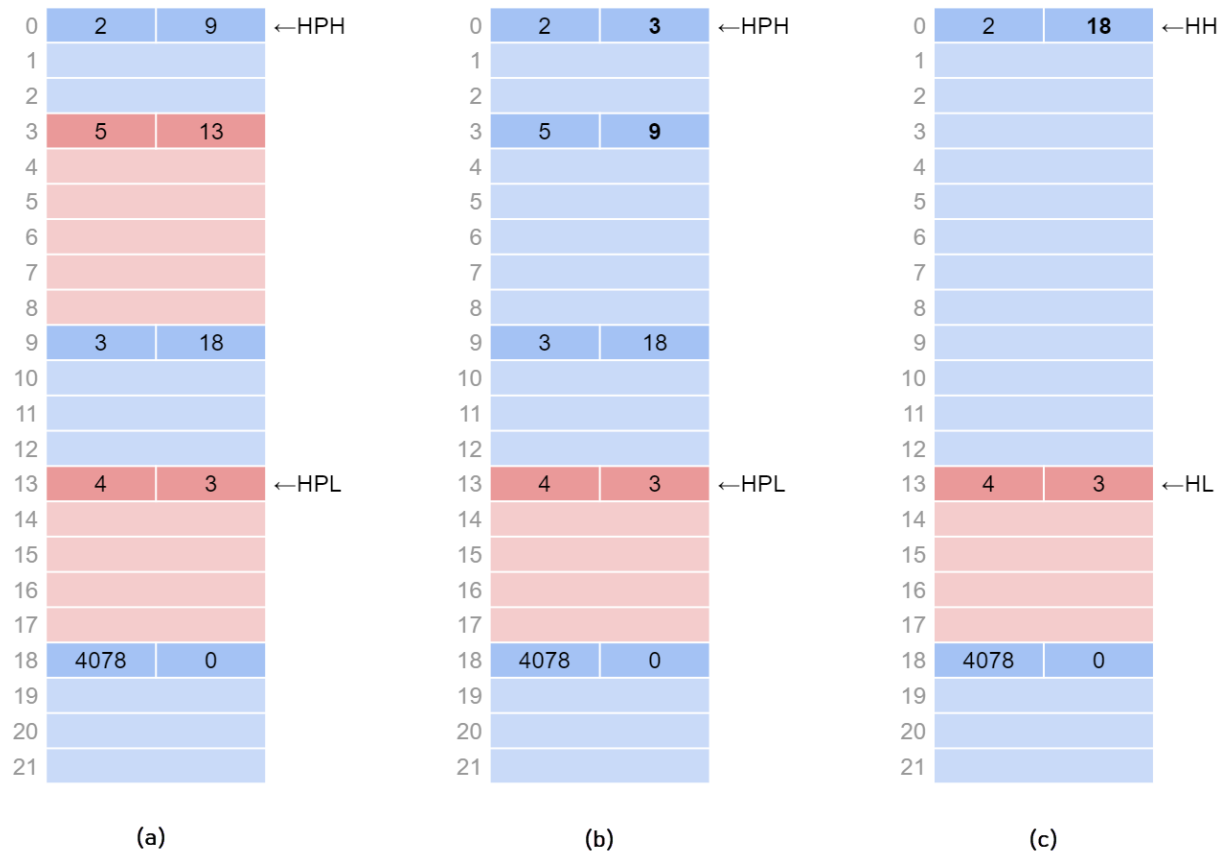


Figura 2

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 más SystemCalls (SYS %3, SYS %4).

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 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 DX. 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 DX, 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
SYS %5
MOV AX, %000
SYS %3
```

Este ejemplo combina 2 interrupciones, el SYS %5 solicita memoria dinámica con espacio para 50 celdas y deja configurada en DX la dirección de memoria a utilizar, luego el SYS %3 solicitará al usuario la entrada de caracteres.

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 DX. 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*) 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.

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

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 SPL >=SSH.

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 26/MAY/2021

- 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.