TRABAJO PRÁCTICO

MÁQUINA VIRTUAL - PARTE II

Introducción

En esta segunda parte se deberá ampliar la máquina virtual para que, además de ejecutar un programa desde el comienzo, pueda cargar la imagen de un proceso y continuar su ejecución. También deberá brindar soporte a un proceso con hasta seis segmentos, manejo de pila con nuevas instrucciones y sus respectivos registros, modificadores sobre los operandos, nuevas llamadas al sistema para lectura y escritura de *strings*, limpiar consola y definir breakpoints.

Máquina virtual

Se debe entregar el código fuente y el ejecutable compilado de la máquina virtual, la cual debe poder utilizarse desde una consola del siguiente modo:

```
vmx [filename.vmx] [filename.vmi] [m=M] [-d] [-p param1 param2 ... paramN]
```

Donde:

- ***vmx*** es el programa ejecutable del proceso Ejecutor o Máquina Virtual.
- ***filename.vmx*** (opcional*) es la ruta y nombre del archivo con el programa en lenguaje máquina (puede ser cualquier nombre con extensión ***.vmx***).
- ***filename.vmi*** (opcional*) es la ruta y nombre del archivo donde se almacenará la imagen de la máquina virtual (puede ser cualquier nombre con extensión ***.vmi***).
- ***m=M*** (opcional) permite indicar el tamaño de la memoria principal, donde ***M*** es un valor expresado en KiB. Si se omite, el valor por defecto sigue siendo 16 KiB.
- ***-d*** (opcional) es un flag que fuerza a la máquina virtual a mostrar el código *Assembler* correspondiente al código máquina cargado en la memoria principal.
- ***-p*** (opcional) es un flag que sirve para indicar los parámetros (***param1 param2 ... paramN***) que se le deben pasar a la subrutina principal del proceso. Siempre se debe escribir al final del comando.

Nota: para la ejecución es obligatorio al menos uno de los dos archivos: .vmx y/o .vmi. En caso de no especificarse un archivo .vmx, se ignoran los parámetros -p.

Descripción de la máquina virtual

La máquina virtual a implementar en esta segunda parte, debe tener los siguientes componentes:

- Memoria principal (RAM) de tamaño variable
- Tabla de descriptores de segmentos
- 32 registros de 4 bytes
- Procesador con capacidad para:
    - decodificar instrucciones en lenguaje máquina
    - direccionar a cada byte de la memoria principal
    - realizar operaciones aritméticas y lógicas en 32 bits

Archivos de entrada

Programa (*vmx*)

El programa binario de esta segunda parte contará con una nueva cabecera (header) que incorporará nueva información respecto de la primera parte. La máquina virtual debe ser capaz de interpretar **ambas cabeceras**, utilizando como referencia el número de versión. Para esto, el traductor (vmt) posee un parámetro v=V que permite indicar el número de versión V del archivo binario.

A continuación de la cabecera, se ubicará el código máquina del programa y, seguidamente, el contenido del *Const Segment* (cadenas de caracteres constantes).

Header		
Nº byte	**Campo**	**Valor**
0 - 4	Identificador	"VMX25"
5	Versión	2
6 - 7	Tamaño del *Code Segment*	–
8 - 9	Tamaño del *Data Segment*	–
10 - 11	Tamaño del *Extra Segment*	–
12 - 13	Tamaño del *Stack Segment*	–
14 - 15	Tamaño del *Const Segment*	–
16 - 17	*Offset* del *entry point*	–

Imagen (*vmi*)

Los archivos de imagen (**.vmi*) almacenan el estado de la máquina virtual en un instante de tiempo. Al comienzo de este archivo se encuentra una cabecera (*header*) con la siguiente estructura:

Header		
Nº byte	**Campo**	**Valor**
0 - 4	Identificador	"VMI25"
5	Versión	1
6 - 7	Tamaño de la memoria principal (KiB)	–

Seguidamente se almacenan los registros, luego la tabla de descriptores de segmentos y finalmente la totalidad de la memoria. En resumen, el archivo debe estructurarse de la siguiente manera:

Archivo de imagen (.vmi*)**	
Sección	**Tamaño (bytes)**
Header	8
Registros	32 x 4 = 128
Tabla de descriptores de segmentos	8 x 4 = 32
Memoria principal	(variable)

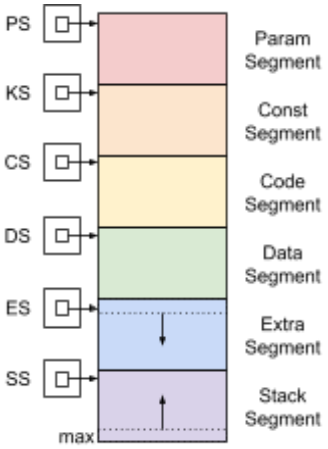
Ejecución

En esta segunda parte, la máquina virtual deberá ser capaz de ejecutar un programa o continuar la ejecución de una imagen:

- Si recibe como entrada un archivo *.vmx*, crea la memoria según el tamaño definido, crea y ubica los segmentos según el *header* del archivo y parámetros, luego configura la tabla de segmentos. Además, debe inicializar el registro IP con el *entry point* y actualizar el *Stack Segment* y el *Param Segment* (más detalles en la sección "Subrutina principal").
- Si no recibe un *.vmx* y recibe solo un archivo de imagen *.vmi*, debe cargar la memoria principal, la tabla de segmentos y los registros tal como están en el archivo y continuar la ejecución.
- En caso de recibir ambos archivos, se ejecuta el archivo *.vmx* y se utiliza el *.vmi* para generar la imagen en cada *breakpoint* (más detalles en la sección "Herramientas de depuración").

Memoria principal

En esta segunda parte, la memoria principal tendrá un tamaño variable que será indicado por un parámetro en la línea de comandos al momento de ejecutarla. Al iniciar la máquina virtual, se debe crear el proceso de acuerdo a los tamaños de los segmentos y cargar el código byte a byte en el *Code Segment*, así como las constantes en el *Const Segment* y los parámetros en el *Param Segment*.



Todos los segmentos deben cargarse de forma contigua en la memoria, respetando el siguiente orden: *Param Segment*, *Const Segment*, *Code Segment*, *Data Segment*, *Extra Segment*, *Stack Segment*. Sin embargo, a excepción del *Code Segment* y del *Stack Segment*, puede suceder que no todos los segmentos estén presentes.

El proceso no necesariamente ocupará toda la memoria principal. En caso de que no cuente con espacio suficiente para alojar al proceso en su totalidad, la máquina virtual deberá detener su ejecución e indicar el error.

Tabla de descriptores de segmentos

La tabla de descriptores de segmentos se mantiene igual que en la primera parte. Sin embargo, ahora solo deberán crearse entradas para los segmentos que tengan un tamaño mayor a cero. Esto quiere decir que la posición de cada segmento en la tabla no será siempre la misma. El orden de los segmentos en la tabla debe ser el mismo que su ubicación física en la memoria principal.

La construcción de la tabla debe ser posterior a la creación del *Param Segment*, si es que se definieron parámetros en la ejecución. Nótese que la ubicación de cada segmento en la memoria pueden variar de una ejecución a otra con el mismo *.vmx*, variando la cantidad de parámetros ingresados.

Registros

Los registros quedarán dispuestos de la siguiente manera:

Código	Nombre	Descripción	Código	Nombre	Descripción
0	LAR	Acceso a memoria	16	AC	Acumulador
1	MAR		17	CC	Código de condición
2	MBR		18	-	Reservado
3	IP	Instrucción	19	-	
4	OPC		20	-	
5	OP1		21	-	
6	OP2		22	-	
7	SP	Pila	23	-	
8	BP		24	-	
9	-	Reservado	25	-	
10	EAX	Registros de propósito general	26	CS	Segmentos
11	EBX		27	DS	
12	ECX		28	ES	
13	EDX		29	SS	
14	EEX		30	KS	
15	EFX		31	PS	

Antes de comenzar la ejecución, la máquina virtual debe inicializar los registros CS, DS, ES, SS, KS y PS con los punteros al comienzo de sus respectivos segmentos. Si alguno de los segmentos no existe (su tamaño es igual a cero), su respectivo registro deberá quedar cargado con un -1 (0xFFFFFFFF).

El registro SP es un puntero al tope de la pila. Inicialmente, la pila estará vacía, porque debe inicializarse con el valor del SS + el tamaño de la pila. En su estado inicial, el registro SP apunta a una posición de memoria fuera del *Stack Segment*.

El registro BP también se utiliza para referenciar posiciones de la pila. Sin embargo, en principio no requiere ninguna inicialización, sino que será cuestión del programador utilizarlo correctamente.

Instrucciones en lenguaje máquina

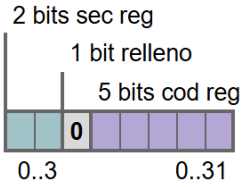
Códigos de operación

A continuación se listan las nuevas instrucciones a implementar, junto con sus códigos de operación asociados en hexadecimal, clasificadas según la cantidad de operandos.

1 Operando		0 Operandos	
Mnemónico	**Código**	**Mnemónico**	**Código**
PUSH	0B	RET	0E
POP	0C		
CALL	0D		

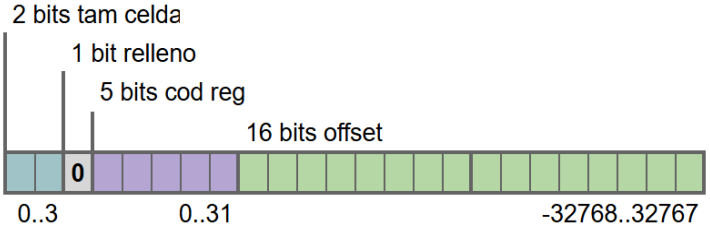
Operandos

Operando de registro: los dos bits más significativos indican el sector de registro, de acuerdo a la siguiente codificación:



Código binario	Descripción	Ejemplo
00	registro de 4 bytes	EAX
01	4to byte del registro	AL
10	3er byte del registro	AH
11	registro de 2 bytes	AX

Operando de memoria: los dos bits más significativos indican el tamaño de la celda a leer o escribir, de acuerdo a la siguiente codificación:



Código binario	Tamaño de la celda
00	long (l)
10	word (w)
11	byte (b)

Manejo de pila

Si bien el correcto uso de la pila queda bajo la responsabilidad del programador, se debe tener en cuenta la estricta implementación de las operaciones que se realizan sobre la misma.

Las instrucciones **PUSH**, **POP**, **CALL** y **RET**, por las cuales el programador utiliza la pila, siempre almacenan o extraen 4 bytes, aunque los operandos no sean de 4 bytes (en el caso de **PUSH** y **POP**). Tanto al almacenar como al extraer un valor en la pila, el orden de los bytes es *big endian*, al igual que las operaciones de memoria sobre otros segmentos.

La pila crece hacia las direcciones inferiores de la memoria, por lo tanto cuando se está **agregando** a la pila se **decrementa** el valor del SP, y cuando se está **quitando** de la pila se **incrementa** el valor del SP.

La instrucción **PUSH** debe seguir estrictamente los pasos:

1. Decrementar el valor del SP en 4.
2. Si el valor de **SP** es menor que el valor del registro **SS**, entonces será un **STACK OVERFLOW** y se aborta la ejecución de la máquina virtual.
3. Obtener el valor del operando (sea inmediato, de registro o de memoria).
4. Transformar el valor obtenido a 4 bytes (al igual que con cualquier operación).
5. Almacenar en la pila desde los bytes menos significativos del valor, dejando en el tope el byte más significativo.

Por ejemplo: PUSH AX, decrementar el valor de **SP** en 4, si EAX = 0x1234ABCD se toman los 2 bytes menos significativos (AX), 0xABCD, se lo transforma en 4 bytes, 0xFFFFABCD (propagando el signo), y se almacena en la pila, quedando el SP apuntando al byte más significativo del valor.

La instrucción **POP** debe realizar la operatoria inversa:

1. Extraer 4 bytes desde el tope de la pila.
2. Si al realizar esta acción, no se pudo completar porque no había bytes suficientes o la pila estaba vacía, entonces será un **STACK UNDERFLOW** y se aborta la ejecución.
3. Convertir los 4 bytes extraídos en un valor, el cual tendrá en el byte más significativo lo que estaba en el tope de la pila, y continúa en orden hasta el menos significativo.
4. Asignar el valor extraído al primer operando, si el operando es menor a 4 bytes se truncan los bytes más significativos.
5. Incrementar el valor del SP en 4.

Nótese que la instrucción **POP no modifica los valores de la pila**, solo modifica el registro **SP**.

Continuando con el ejemplo: si luego se hace un POP w[ES+4], se leen los 4 bytes apuntados por SP y, por lo tanto, en el byte 4 del *Extra Segment* quedará 0xAB y en el byte 5 0xCD. Finalmente, se debe incrementar el valor del SP en 4. Aunque el valor leído sea 0xFFFFABCD, se debe asignar a 2 bytes (w[...]) y, al igual que en toda la máquina virtual, se truncan los bytes más significativos.

La instrucción **CALL** siempre almacenará en la pila los 4 bytes del valor del **IP** (que ya se encuentra apuntando a la siguiente instrucción) y luego modificará los 2 bytes menos significativos del IP con el valor del operando. Es decir, técnicamente la instrucción **`CALL subrut`** equivale a hacer **`PUSH IP`** y seguido un **`JMP subrut`**.

En el caso de la instrucción **RET**, modificará el **IP** obteniendolo del tope de la pila. Por lo tanto, una instrucción **RET**, equivale a **`POP IP`**.

Segmento de parámetros

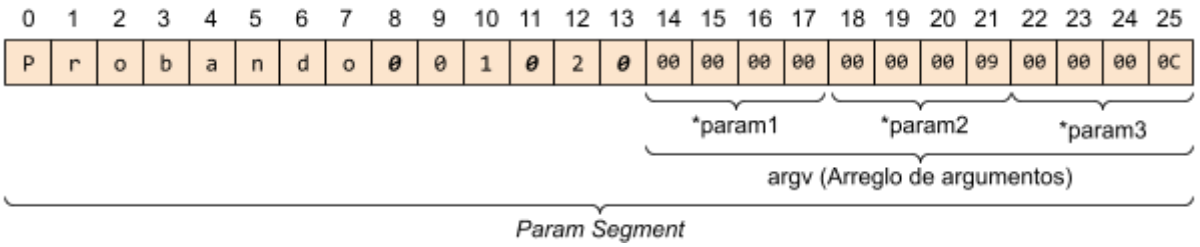
El *Param Segment* es un segmento que se crea según los parámetros de ejecución, si es que estos se encuentran indicados en la línea de comandos. Debe construirse previo a la ejecución y configuración de la tabla de segmentos. De existir, siempre se ubicará a partir de la dirección física `0x00000000` de la memoria, dado que será el primer segmento del proceso y la máquina virtual solo ejecuta un único proceso; por lo tanto, estará definido en la posición `0` de la tabla de segmentos.

El contenido del segmento será: primero, todas las cadenas de caracteres (*strings*) de los parámetros; al final, un arreglo ***argv*** de tamaño ***argc*** de punteros de 4 bytes a cada posición de inicio de los *strings* de cada parámetro, siendo ***argc*** la cantidad de parámetros. Los punteros contendrán en los 16 bits menos significativos el *offset* de cada *string* dentro del segmento y en los 16 bits más significativos `0x0000`, la entrada a la tabla de segmentos.

Ejemplo:

```
vmx program.vmx -p probando 01 2
```

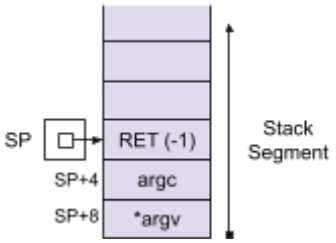
Aquí se definieron tres parámetros: “probando”, “01” y “2” y se dispondrán en el *Param Segment* de la siguiente manera:



De este modo el queda construido el *Param Segment* con un tamaño de 26 bytes, 14 de los cuales están ocupados con los *strings* y 12 por el arreglo de punteros a los *strings*.

Subrutina principal

La ejecución de un nuevo programa debe comenzar en la subrutina principal, por lo que la máquina virtual debe inicializar el registro IP con la posición del Code Segment en los 16 bits más significativos y con el offset del entry point en los 16 bits menos significativos. La pila debe quedar cargada de la siguiente manera:



El valor de la dirección de retorno de la subrutina principal debe ser -1 (0xFFFFFFFF), de manera tal que el proceso finalice al ejecutar la instrucción RET, ya que el registro IP quedará cargado con un puntero fuera de Code Segment.

Los parámetros de la subrutina principal (cadenas de caracteres), de encontrarse definidos, se encuentran en el Param Segment. En la pila debe quedar a continuación: la cantidad de argumentos (argc) y el puntero a la posición inicial del arreglo de argumentos (argv).

En el caso de que no se especifique ningún parámetro, igualmente debe cargarse en la pila la dirección de retorno de la subrutina principal, 0 (0x00000000) en la cantidad de argumentos (argc) y -1 (0xFFFFFFFF) como puntero al arreglo de argumentos (argv).

Llamadas al sistema

3 (STRING READ): almacena 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. En CX (16 bits) se especifica la cantidad máxima de caracteres a leer. Si CX tiene -1 no se limita la cantidad de caracteres a leer.

4 (STRING WRITE): imprime por pantalla un rango de celdas donde se encuentra un *string*. Inicia en la posición de memoria apuntada por EDX, e imprime hasta encontrar un '\0' (0x00).

7 (CLEAR SCREEN): ejecuta una limpieza de pantalla. No requiere ningún registro configurado y tampoco modifica ninguno.

F (BREAKPOINT): si existe un archivo *.vmi* en los parámetros de ejecución, pausa la ejecución, genera una imagen con el estado actual de la máquina virtual en ese archivo y queda en espera de una acción del usuario para *debug* (se explica a continuación). No requiere ningún registro configurado y tampoco modifica ninguno.

Herramientas de depuración

Breakpoint

Los *breakpoints* son un tipo de llamada al sistema especial que permiten pausar o detener la ejecución para observar el estado actual de la máquina virtual. Estas llamadas al sistema deben ser ignoradas si al momento de ejecutar la máquina virtual no se ha incluido el parámetro que indica el archivo de imagen.

Cuando se ejecuta un *breakpoint*, la máquina virtual debe pausar su ejecución y generar un archivo de imagen. Luego, debe quedar a la espera de que el usuario realice una de las siguientes acciones:

- Si se ingresa el carácter 'g' (*go*), la máquina virtual continúa su ejecución hasta el próximo *breakpoint* o hasta finalizar la ejecución.
- Si se ingresa el carácter 'q' (*quit*), la máquina virtual aborta la ejecución, dejando intacto el archivo *.vmi* de modo que se pueda retomar la ejecución del mismo.
- Si únicamente se presiona la tecla *Enter*, la máquina virtual debe ejecutar la siguiente instrucción y luego volver a realizar un nuevo *breakpoint*, sin importar qué haga dicha instrucción. Esto posibilita que el código pueda ser ejecutado paso a paso.

Debugger

El *debugger*, provisto por la cátedra, permite visualizar en tiempo real el estado de la máquina. Una vez que se encuentra en ejecución, monitorea constantemente un archivo de imagen y actualiza su salida por pantalla ante cualquier cambio en el mismo. Permite visualizar el código assembler desensamblado, la ubicación del registro IP, los valores de todos los registros y la tabla de descriptores de segmentos. Así mismo, permite realizar las siguientes acciones:

- Si se ingresa un valor inmediato o un intervalo (dos valores), se muestra el contenido de las celdas de memoria en esas direcciones físicas.
- Si se ingresa el carácter 'q' (*quit*), el *debugger* finaliza su ejecución.

Se utiliza por consola del siguiente modo:

```
vmg filename.vmi [-r] [-s] [w=W]
```

Donde:

- ***vmg*** es el programa ejecutable del *debugger*.
- ***filename.vmi*** (obligatorio) es la ruta y nombre del archivo donde se almacenará la imagen de la máquina virtual (puede ser cualquier nombre con extensión ***.vmi***).
- ***-r*** (opcional) es un flag que le indica al *debugger* que muestre los valores de los registros.
- ***-s*** (opcional) es un flag que le indica al *debugger* que muestre los valores de los segmentos.
- ***w=W*** (opcional) permite indicar el tamaño de la ventana de código, donde ***W*** es un valor que indica la cantidad de líneas de código a mostrar (alrededor de la apuntada por el registro IP).

Errores

Además de los errores descritos en la primera parte, la máquina virtual debe ser capaz de detectar:

- **Memoria insuficiente:** cuando la memoria principal no cuenta con espacio suficiente para alojar todos los segmentos del proceso.
- **Stack overflow:** cuando se ejecuta una instrucción PUSH o CALL y no haya espacio suficiente en la pila.
- **Stack underflow:** cuando se ejecuta una instrucción POP o RET y la pila ya está vacía o no puede extraer los 4 bytes correspondientes.

Ante la ocurrencia de cualquiera de estos errores, la máquina virtual debe informarlo e inmediatamente abortar la ejecución del proceso.

Disassembler

En esta segunda parte, si a la máquina virtual se le indica que muestre el código *Assembler* (***-d***), también deberá mostrar las cadenas de caracteres constantes con el siguiente formato:

```
[0000] XX XX XX XX          | "abc..."
```

- **[0000]** es la dirección de memoria donde está alojada la cadena de caracteres, expresada con 4 dígitos hexadecimales.
- **XX XX XX XX** es la cadena de caracteres en hexadecimal (incluyendo el '\0'), agrupada por bytes. Si se superan los 7 bytes, se deben mostrar los primeros 6, seguido de dos puntos (**..**) para indicar que la cadena continúa.
- **"abc..."** es la cadena de caracteres completa entre comillas (**"**). Cuando el caracter ASCII no es imprimible, debe escribir un punto (**.**) en su lugar.

Además, se deberá indicar con el símbolo mayor (**>**) la ubicación del *entry point*. También deberán visualizarse los “pseudónimos” de los sectores de registros y los modificadores (prefijos) de los operandos de memoria.

Por ejemplo:

```
 [0000] 56 4D 58 32 35 00    | "VMX25"
 [0006] 61 62 63 08 0A 0D 00 | "abc..."
 [000D] 41 72 71 75 69 74 .. | "Arquitectura de Computadoras"
>[002A] B1 00 0A 9B 00 05    |  ADD    w[DS+5],         10
 [0030] 96 00 08 CA          |  SHL         AX,          8
 [0034] 48 8E                |  NOT         EH
```