TRABAJO PRÁCTICO

TRADUCTOR ASSEMBLER - MÁQUINA VIRTUAL

PARTE I

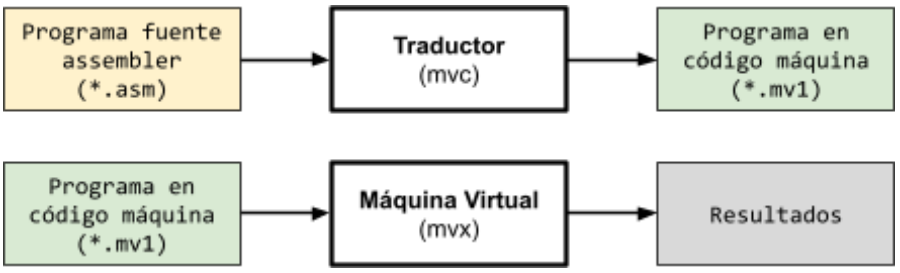
Introducción

El trabajo práctico consiste en realizar dos programas en un lenguaje a elección. Uno que lea un código fuente escrito en un lenguaje assembler particular de la MV, lo traduzca al código máquina de una 'arquitectura virtual' (proceso de traducción) y otro que emule su ejecución (proceso de ejecución).

PROCESOS

TRADUCCIÓN (Traductor): donde se debe leer el código fuente assembler de un archivo de texto (*.asm), traducirlo a código máquina, y generar otro archivo binario codificado (*.mv1) (Programa ejecutable de la MV)

EJECUCIÓN (Máquina Virtual): este proceso debe obtener las instrucciones a ejecutar desde el archivo generado por el proceso Traductor (*.mv1 junto con sus parámetros), configurar la memoria RAM virtual, configurar los registros Virtuales, interpretar las instrucciones, emular su funcionamiento y producir los resultados generados por dicha ejecución. Este proceso representa la máquina virtual donde corren los programas compilados.



PROGRAMAS

Se deben entregar los fuentes y los ejecutables compilados para Win32 o Linux del Traductor y el Ejecutor. Ambos se deben poder utilizar pasándole los parámetros del siguiente modo:

```
mvc.exe AsmFilename BinFilename [-o]
mvx.exe BinFilename [-b] [-c] [-d]
```

Donde:

- ***mvc.exe*** es el programa ejecutable del proceso Traductor (en caso de hacerlo en java será mvc.java).
- ***AsmFilename*** es la ruta del archivo de texto donde está escrito el código fuente que será traducido por la máquina virtual (puede ser cualquier nombre con extensión .asm).
- ***BinFilename*** es la ruta del archivo generado por el Traductor conteniendo el Programa Código Máquina o Imagen Memoria (puede ser cualquier nombre con extensión .mv1).
- ***-o*** es un flag o bandera opcional para indicar que se omita la salida por pantalla de la traducción. Este flag no omite los mensajes de error producidos durante la compilación.
- ***mvx.exe*** es el programa ejecutable del proceso Ejecutor o Máquina Virtual (en caso de hacerlo en java será mvx.java).
- ***-b*** es un flag que fuerza a la máquina virtual (***mvx.exe***) a detener su ejecución con un *breakpoint* y solicitar al usuario un comando para mostrar un fragmento de la memoria.
- ***-c*** hace clear screen cada vez que se ejecuta la máquina virtual y encuentra un *breakpoint*.
- ***-d*** es un flag que fuerza a la máquina virtual (***mvx.exe***) a mostrar el código assembler cargado en la RAM desde el archivo Imagen Memoria. Ni los rótulos ni los comentarios serán mostrados, porque no existen en el archivo binario. Además hace el disassembler cada vez que se ejecuta un *breakpoint* indicando la posición del registro IP.

Programa MV

Como se dijo previamente, **el programa (*.mv1)** es el resultado de la traducción y punto de entrada del ejecutor (máquina virtual propiamente dicha). 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 al mismo.

Esta configuración es equivalente a la de un programa ejecutable en cualquier sistema operativo.

El *header* se compone de 6 bloques de 4 bytes, para respetar el formato equivalente a 6 instrucciones (cada instrucción es de 32bits). A partir del 7mo bloque se encuentra el código que se deberá cargar íntegramente en la memoria.

Header	
Nº bloque	**Contenido**
0	“MV-1” 4 chars FIJO
1	Tamaño del Código
2	4 byte reservados
3	4 byte reservados
4	4 byte reservados
5	“V.22” chars FIJO

Descripción de la Máquina Virtual

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

- Memoria RAM de 4096 celdas de 4 bytes cada una.
- 16 registros de 4 bytes (se utilizan 10 en esta primera parte)
- 6 sub-registros de 16 bits
- 12 sub-registros de 8 bits

Posición	Nombre				Descripción
	32bits	16bits	3er byte	4to byte	
0	DS	-	-	-	Segmento de datos
1					Reservado
2					
3					
4					
5	IP	-	-	-	Instruction Pointer
6					Reservado
7					
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	

La memoria RAM se divide en 2 segmentos: Segmento de código (*Code Segment*), Segmento de datos (*Data Segment*) Cada segmento se define en la traducción.

- El **segmento de código** siempre inicia en la posición 0 de la memoria, y almacena el código del programa traducido a lenguaje máquina, su longitud se calcula en la traducción.
- El **segmento de datos** iniciará a continuación del segmento de código y su posición quedará indicada en el registro DS, y se utiliza para almacenar datos durante la ejecución.

Las direcciones de memoria para uso de datos se calculan tomando como base el **registro DS**.

```
Dirección física del dato = Valor del registro DS + Dirección Relativa (desplazamiento)
```

El **registro IP (Instruction Pointer)** se usará para apuntar a la próxima instrucción a ejecutar dentro del Code Segment, es decir que se utilizará 0 como base del cálculo de direccionamiento.

El **registro AC (Accumulator)** se utiliza para algunas operaciones especiales y también podrá ser utilizado para almacenar datos auxiliares.

El **registro CC (Condition Code)** informará sobre el resultado de la última operación matemática o lógica ejecutada. De los 32 bits que tiene, solamente se usarán dos:

- El bit menos significativo es el '**bit indicador de cero'** (bit Z). Valdrá 1 cuando la última operación matemática o lógica haya dado por resultado cero, y 0 en cualquier otro caso.
- El bit más significativo es el '**bit indicador de signo**' (bit N). Valdrá 1 cuando la última operación matemática o lógica haya dado por resultado un valor negativo, y 0 en otro caso.

Los **registros de uso general (General Purpose Registers)**, desde EAX hasta EFX, servirán para almacenar datos y realizar cálculos durante la ejecución.

Todos los registros son de 32 bits, sin embargo según la forma cómo se lo identifique en el código se accede al registro completo, a los últimos 16 bits (AX a FX), al último byte (AL a FL) o al anteúltimo byte (AH a FH).

La máquina virtual debe ser capaz de interpretar y procesar una serie de instrucciones que se explican más adelante en la sección **Ejecución**.

Traducción

Cada línea del programa fuente assembler puede contener una sola instrucción. Cada instrucción se compone como máximo de **un rótulo**, **un mnemónico** (palabra que representa un código de operación), **dos, uno o ningún operando** separados por coma (dependiendo del tipo de instrucción) y **un comentario**. Con la siguiente sintaxis:

```
RÓTULO:    MNEMÓNICO    OPN_A, OPN_B ;COMENTARIO
```

Lo único obligatorio para ser considerado instrucción es el **mnemónico**, todo lo demás (dependiendo de la instrucción) puede no estar o ser opcional.

La traducción **consiste en codificar cada instrucción del programa fuente escrito en assembler** y generar un archivo binario con el código correspondiente en lenguaje máquina. Una vez finalizado, y si no hay errores, se puede dar comienzo a la ejecución.

La traducción debe mostrar por pantalla el resultado del proceso, el código de máquina mismo y la instrucción a la que pertenece (línea de texto original). Así como los errores, si los hubo.

Puede haber líneas de código que estén en blanco o que solo tengan comentarios, en cuyo caso deben ser ignoradas en la traducción, pero se muestran por pantalla durante el proceso de traducción.

La disposición de las instrucciones en el programa fuente no están sujetas a una tabulación predeterminada, es decir que puede haber una cantidad variable de caracteres separadores delante de una instrucción y entre las partes que la forman. Se consideran separadores a los espacios en blanco y al carácter de tabulación.

Instrucciones del lenguaje Assembler

Las instrucciones de la máquina se componen de un código de operación (Operation Code) y operandos.

El lenguaje assembler es una representación del lenguaje de máquina donde las operaciones se describen con un **mnemónico**, que pueden estar escritos en minúsculas o mayúsculas.

En total serán 32 instrucciones las que reconoce la MV, en esta primera parte solo se implementarán 25. En el apartado SEMÁNTICA DE INSTRUCCIONES se especifica cada una de las instrucciones.

A continuación se lista el conjunto de instrucciones a implementar y el código de máquina asociado (en base hexadecimal), clasificadas según la cantidad de operandos.

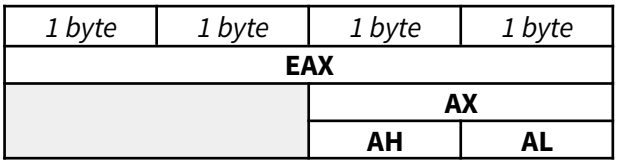
2 Operandos		1 Operando		0 Operandos	
mnem	cod	mnem	cod	mnem	cod
MOV	0	SYS	F0		
ADD	1	JMP	F1	STOP	FF1
SUB	2	JZ	F2		
SWAP	3	JP	F3		
MUL	4	JN	F4		
DIV	5	JNZ	F5		
CMP	6	JNP	F6		
SHL	7	JNN	F7		
SHR	8	LDL	F8		
AND	9	LDH	F9		
OR	A	RND	FA		
XOR	B	NOT	FB		

Se admiten tres tipos de operandos:

Operando inmediato: El dato es directamente el valor del operando.
Se pueden tener valores numéricos en base 10, 8 y 16, que se representan con los símbolos **#**, **@** y **%** respectivamente delante del dato. El símbolo # es opcional.
Además se puede usar el apóstrofe (‘) para indicar valores ASCII.
Ejemplos:

#65 o 65, @101, %41
‘A o ‘A’ (valor decimal 65)
rotulo (ver apartado de rótulo)

Operando de registro: El dato es el contenido, o parte, de alguno de los registros de la máquina virtual. Se especifican por el identificador del registro.
Para los registros EAX a EFX, se puede especificar un "pseudónimo" para identificar el sector y la cantidad de bytes del registro a la que se accede. El prefijo "E" indica "extendido" quiere decir que se accede a los 4 bytes, sin prefijo se accede sólo a los 2 bytes menos significativos y con el sufijo L o H (en lugar de la X) indica el byte bajo (L = low) o el byte alto (H = high) como si indica en la figura:



IMPORTANTE: el registro solo tiene 4 bytes (32 bits). Si se asigna un valor al EAX, y luego otro valor al AX, esta última operación "pisó" los últimos 2 bytes del registro EAX.

Ejemplos:

EAX o **eax** (case insensitive) accede al registro extendido de 32bits eax
FH : accede al el 3er byte del registro EFX

Operando directo: el dato es el contenido de una posición de memoria RAM.
El operando indica el desplazamiento dentro de la zona de datos respecto del valor de DS. Se expresa entre corchetes la dirección relativa con un valor, al igual que en el operando inmediato puede estar dada en decimal, octal, hexadecimal o con un caracter.
Ejemplos: si el segmento de datos comienza en la celda 40 **(DS = 40)**.

[11] sería el valor de la celda 51,
[@10] el valor de la celda 48,
[%101] el valor de la celda 297 y
['A'] el valor de la celda 105.

Llamadas al sistema (System Calls o Interrupciones por software)

Un System Call es un mecanismo por el cual el programador solicita al sistema una acción, y por lo tanto se detiene el programa en ejecución resguardando los registros, para ejecutar un rutina de atención a un servicio generalmente asociado a los recursos del computador (teclado, monitor, archivo, etc). Cuando finaliza la rutina de atención, se restauran los registros y el programa retorna a su ejecución normal.

En esta máquina virtual los System Calls se implementan utilizando la instrucción **SYS e indicando el código de la operación**. El código de operación es un número entero positivo (constante decimal) y se ubica como primer (y único) operador de la instrucción. y pueden ser:

Código	Significado
1	READ
2	WRITE
F	BREAKPOINT

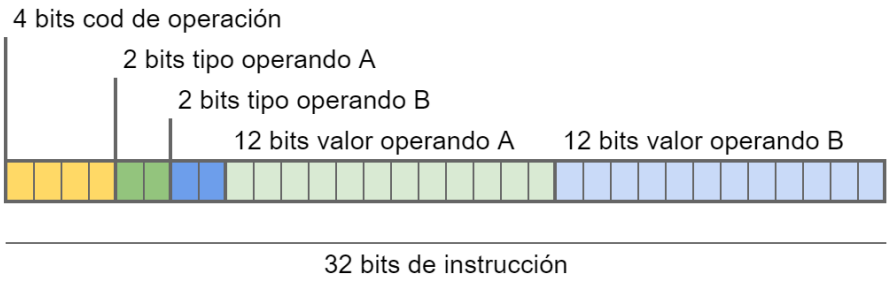
En el apartado de **Semántica de las system calls** se especifica el comportamiento de cada uno.

Instrucciones del lenguaje máquina

Cada instrucción en lenguaje máquina ocupa 1 celda (de 4 bytes) de memoria. La celda contiene: el código de operación, el tipo de cada operando y el valor del o los operandos. Tanto el tamaño (cantidad de bits) del código de la operación como el espacio para los operandos varía según la clasificación de la instrucción. Las instrucciones están codificadas siguiendo el criterio de Código de Operación con Extensión.

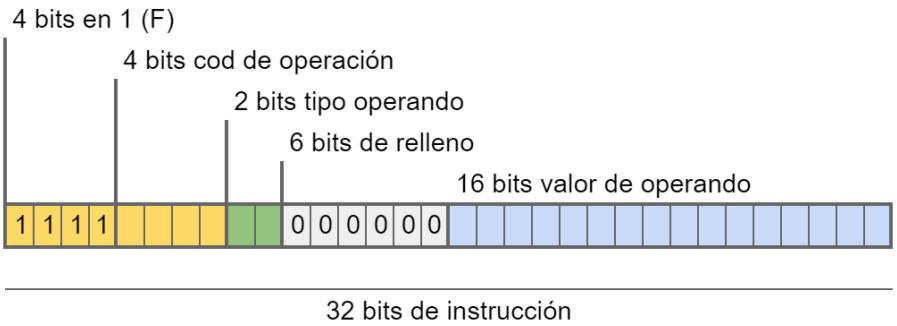
Instrucciones con 2 operandos

Estas instrucciones, de la celda de 32 bits, utilizan los 4 bits más significativos (más a izquierda) para el código de instrucción. Los siguientes 4 bits se utilizan para determinar el tipo de los operandos, 2 bits para el tipo del operando A y 2 bits para el tipo del operando B. Los últimos 24 bits se utilizan para los valores de los operandos, 12 bits para cada uno en secuencia: primero el operando A luego el B.



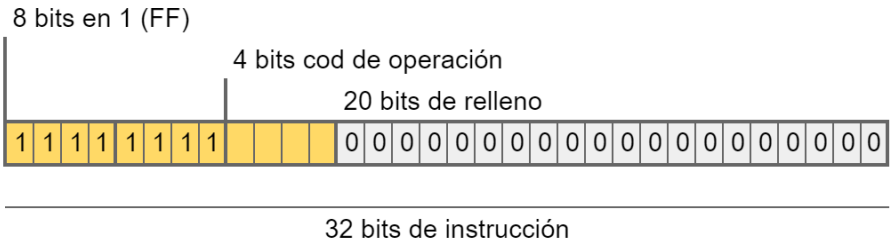
Instrucciones con 1 operando

Estas instrucciones utilizan 8 bits para el código de instrucción, estando los 4 primeros en 1 para indicar esta clase de instrucciones y los 4 siguientes para distinguir cada una de estas instrucciones. Los 2 bits siguientes se utilizan para codificar el tipo de operando. Luego van 6 bits en 0 de relleno, sin uso. Finalmente los últimos 16 bits se utilizan para codificar el valor del operando.



Instrucciones sin operandos

Estas instrucciones utilizan 12 bits para el código de instrucción, siendo los primeros 8 bits 1, y los siguientes 4 bits el código de instrucción de esta clase. Los 20 bits restantes son de relleno en 0.



En esta MV solo habrá 2 instrucciones sin argumentos (1 en esta primera parte), aunque tenga potencial para 16.

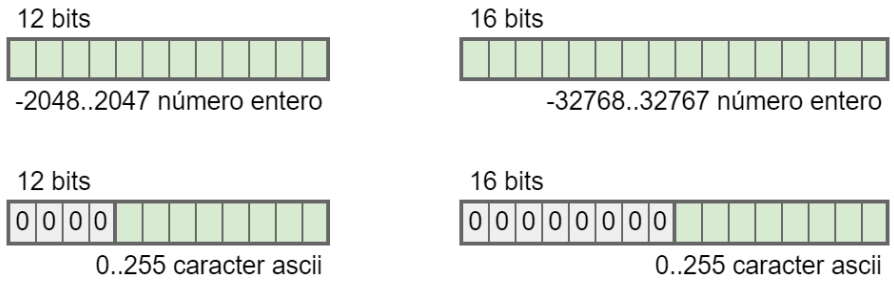
Codificación de operandos

Como se mencionó previamente, en esta primera parte, existen 3 tipos de operandos: Inmediatos (0), de registro (1) y directos (2), codificados en 2 bits dentro de la celda de instrucción.

Tipo Operando	Código binario
INMEDIATO	00
DE REGISTRO	01
DIRECTO	10

Inmediato

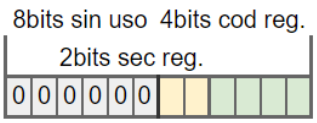
El valor del operando inmediato puede codificarse en 16 o 12 bits dependiendo de la clase de instrucción (1 o 2 operandos). Se guarda el valor binario directamente (independientemente de la forma en la que esté representado: decimal, octal, hexadecimal o caracter). Obviamente la cantidad de bits disponibles para el operando condiciona el rango de valores que puede soportar.

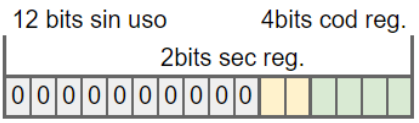


De Registro

En valor del operando de registro se almacena el código de registro utilizando solo los últimos 4 bits del espacio reservado para el operando. En el caso de los registros EAX a EFX, se utilizan además los 2 bits anteriores para identificar a qué parte del registro se accede.

De Registro



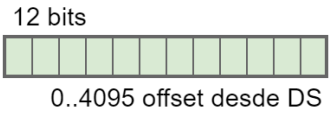


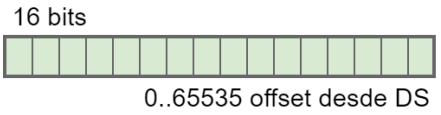
Los 2 bits para identificar el sector de registro se codifican de la siguiente manera:

00	registro de 4 bytes
01	4to byte del registro
10	3er byte del registro
11	registro de 2 bytes

Directo

Se almacena el valor de la dirección en binario, utilizando los 16 o 12 bits disponibles según el tipo de instrucción.



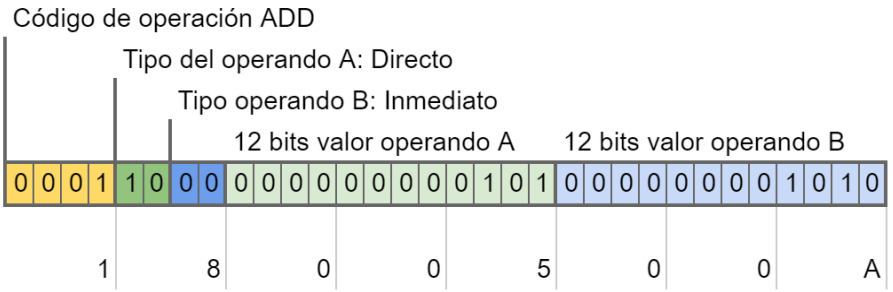


Ejemplos de codificación de instrucciones

ADD [5],10

1. El primer paso sería reconocer el mnemónico: "ADD".
2. Luego buscar el código correspondiente: $1_{16} = 0001_2$
3. De aquí se deduce que es una instrucción con 2 operandos. por lo tanto hay que reconocer ambos.
4. El primer operando es "[5]", por los corchetes sabemos que es un operando indirecto y el contenido se interpreta como un inmediato, en este caso es 5 en decimal.
    a. Tipo = 10_2
    b. Valor = $5_{10} = 000000000101_2$ (12 bits por ser instrucción de 2 operandos)
5. El segundo operando es "10", se interpreta con un inmediato en decimal.
    a. Tipo = 00_2
    b. Valor = $10_{10} = 000000001010_2$ (12 bits por ser instrucción de 2 operandos)

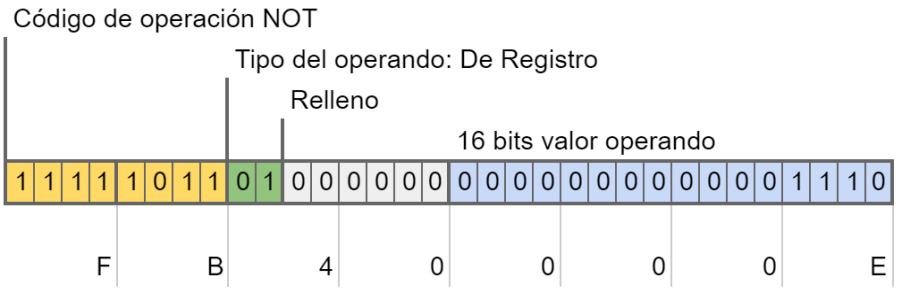
Por lo tanto la instrucción codificada será:



Para escribir la instrucción se utiliza su representación en hexadecimal: 18 00 50 0A

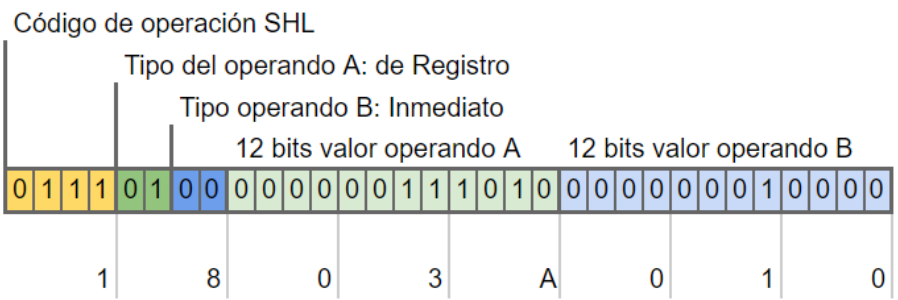
NOT EEX

1. Se reconoce el mnemónico “NOT”,
2. con código correspondiente FB_{16},
3. cómo inicia con F se deduce que es de un operando,
4. el operando es “EEX”, por lo que se sabe que es de registro con codificación: 14_{10} (E_{16})
5. y como es un registro extendido (4 bytes), los 2 bits previos al código de registro van en 00_2



SHL AX, %10

1. Se reconoce el mnemónico “SHL”,
2. con código correspondiente 07_{16},
3. se reconoce como de 2 operandos.
4. El primer operando es tipo registro (01_2), puntualmente AX, registro de 2 bytes, por lo tanto debe agregarse 11_2 antes del código de registro, y el código del registro es 10_{10} por lo tanto 1010_2.
5. El segundo operando es un inmediato (00_2), expresado en Hexadecimal %10 el es número 16_{10} y en binario 10000_2



Rótulos (labels)

Un rótulo es un “nombre” que puede asignarse a una instrucción, para luego poder hacer referencia a ella en una instrucción de salto.

Se define como una cadena de caracteres seguida del carácter **dos puntos**. Por ejemplo:

```
OTRO: ADD [5],10
```

Hace que **OTRO** tenga como valor el número de **la posición de memoria** de la instrucción ADD.
Las instrucciones de salto pueden hacer uso de los rótulos en sus operandos. Por ejemplo:

```
JMP OTRO
```

Indica realizar un salto incondicional a la instrucción ADD del ejemplo.

Los rótulos son necesarios para independizar al programador de lenguaje assembler de tener que calcular la posición de memoria donde se ubica la línea a la que quiere saltar.

El rótulo utilizado como argumento se debe **reemplazar en la traducción como un operando inmediato en el código máquina**, haciendo a la ejecución transparente a los rótulos (es decir que los rótulos no se encuentran en el código máquina por sus nombres).

Errores

En la traducción se deben detectar, como mínimo, los siguientes errores:

- **Error de sintaxis** es el de una instrucción inexistente, en cuyo caso mostrará un mensaje indicándolo. La línea no será traducida, en la celda de la instrucción quedará en FF FF FF FF.
- **No se encuentra el rótulo** (o label) cuando se hace referencia a un rótulo y el mismo no se encuentra en ninguna línea. Deberá mostrar FFF en lugar del argumento del rótulo.

Ante alguno de estos errores la traducción deberá continuar, pero **no** se generará el archivo Programa Código Máquina.

Advertencias (Warning)

En la traducción se deben detectar al menos un tipo advertencia:

- **Truncado de operando** cuando se asigna un valor a un operando inmediato, sea de 16 o 12 bits y el valor escrito en la constante no puede ser almacenado en el espacio de bits, se debe truncar la parte alta del valor y escribir esta advertencia.

Ante la presencia de una o más advertencias la traducción deberá mostrar el mensaje y continuar generando el archivo Programa Código Máquina.

Salida por pantalla

A medida que el traductor va generando las instrucciones, si está activa la salida por pantalla (***-o***), deberá mostrar una línea por cada instrucción con el siguiente formato:

```
[0000] XX XX XX XX          lin: mnem    opnA, opnB      ;com
```

- **[0000]** La dirección de memoria donde está alojada la instrucción, con 0 adelante para mantener la tabulación
- **XX XX XX XX** sería la instrucción completa en hexadecimal agrupadas por bytes (cada dígito hexa es un nibble o cuatro bits)
- **lin:** si la instrucción tiene rótulo se escribe el rótulo, sino lo tiene se escribe el número de línea, alineado a la derecha de los dos puntos (pos 32) .
- **mnem** de la instrucción
- **argA** sería el argumento A tal como está escrito en el código y alineado a la coma (pos 46)
- **argB** a un espacio de la como el argumento B tal como está escrito
- **;com** finalmente el comentario luego del punto y coma (pos 58)

Las posiciones son una referencia para que quede alineado, si no es posible respetar se desplaza.

Por ejemplo:

```
[0003] 18 00 50 0A         OTRO: ADD      [5], 10        ;suma 10 a en pos 5
[0004] FB 40 00 0E            5: NOT       EX           ;niega Ex
```

Ejemplo de un programa completo

Teniendo un archivo fibo.asm con el siguiente contenido en caracteres ascii:

```
        mov     [10], 0         ;inicializo variables
        mov     [20], 1
otro:   cmp     [20], 100       ;compara
        jp      fin             ;salta si llegó a 100 o más
        swap    [10], [20]
        add     [20], [10]
        mov     eax, %001
        mov     ecx, 1
        mov     edx, 10
        sys     2               ;print [10] en decimal
        jmp     otro
fin:    stop
```

La salida por pantalla (de estar habilitada) sería:

```
[0000]: 08 00 A0 00           1: mov     [10], 0         ;inicializo variables
[0001]: 08 01 40 01           2: mov     [20], 1
[0002]: 68 01 40 64        otro: cmp     [20], 100       ;compara
[0003]: F3 00 00 0B           4: jp       fin           ;salta si llegó a 100 o más
[0004]: 3A 00 A0 14           5: swap    [10], [20]
[0005]: 1A 01 40 0A           6: add     [20], [10]
[0006]: 04 00 A0 01           7: mov      eax, %001
[0007]: 04 00 C0 01           8: mov      ecx, 1
[0008]: 04 00 D0 0A           9: mov      edx, 10
[0009]: F0 00 00 02          10: sys        2            ;print [10] en decimal
[0010]: F1 00 00 02          11: jmp     otro
[0011]: FF 10 00 00         fin: stop
```

Archivo binario

El archivo contendrá el encabezado o header, como se explicó en la sección “Programa MV”, y seguido byte a byte el código máquina. Por convención deberá tener extensión “*mv1*” para ser identificado fácilmente como un archivo ejecutable por la máquina virtual.

Ejecución

Una vez traducido el programa fuente assembler, el código máquina correspondiente estará alojado en un archivo binario con el Programa en Código Máquina (*binFilename*). Lo primero que debe hacer el ejecutor es cargar en la memoria RAM de la Máquina Virtual byte a byte el contenido del archivo, obviamente agrupando de a 4 bytes por celda de memoria.

Proceso central

Para procesar dicho código, se irá obteniendo cada instrucción máquina (y su operandos), desde la memoria haciendo uso del registro IP (puntero a instrucción), se interpretará y se realizará la acción que corresponda.

La estructura general del módulo de ejecución podría ser similar a:

```
do {
    step(mv);
} while ((0 <= mv.reg[IP]) && (mv.reg[IP] < mv.reg[DS]));
```

```
step(mv){
    <Obtener próxima instrucción> memoria[mv.reg[IP]]
    mv.reg[IP]++
    <Decodificar Instrucción>
    <Decodificar Operandos>
    <Ejecutar Instrucción>
}
```

```
IP = 0
WHILE (0<=IP AND IP<DS) DO
    BEGIN
        <OBTENER PRÓXIMA INSTRUCCIÓN>
        IP = IP + 1;  // incrementa el IP a la próxima instrucción.
        <DECODIFICAR INSTRUCCIÓN>
        <DECODIFICAR OPERANDOS>
        <EJECUTAR INSTRUCCIÓN>
    END
```

Semántica de las instrucciones

A continuación se detalla el significado de las instrucciones del lenguaje assembler. Cuando se hace referencia a una dirección de memoria, debe interpretarse como una dirección relativa, cuyo valor deberá sumarse al contenido del registro de segmento de datos (DS) para calcular la dirección efectiva.

Instrucciones con 2 operandos

MOV: Permite asignar a un registro o posición de memoria un valor, que puede ser el contenido de otro registro, posición de memoria o un valor inmediato.

```
MOV AX,10          ;Carga en AX el valor 10 decimal
MOV BX,CX          ;Carga en BX el valor del registro CX
MOV DX,[10]        ;Carga en DX el valor almacenado en la celda de memoria 10.
```

Si el segundo operando es inmediato al ser asignados a un espacio con más bits (sea un registro de 32 bits o una celda de memoria de 32 bits) hay que tener en cuenta que se debe propagar el bit de signo.

ADD, SUB, MUL, DIV: Efectúan las cuatro operaciones matemáticas básicas. El primer operando debe ser de registro o memoria, ya que es donde se guarda el resultado. El resultado de estas instrucciones afecta el valor del registro CC. El DIV tiene la particularidad de dejar el **resto de la división entera (módulo) en AC**, además de obviamente dejar el resultado de la división en el primer operando.

```
ADD AX,2           ;Incrementa AX en 2
MUL EAX,BX         ;Multiplica EAX por BX dejando el resultado en EAX
SUB [10],DL        ;Resta DL (el byte bajo de DX) del valor de la celda 10,
                   ; y el resultado queda en la celda 10
DIV [10],7         ;Divide el valor de la celda 10 por 3,
                   ;el resultado queda en la celda 10 y en AC el resto
```

SWAP: Permite intercambiar los valores de los dos operandos. (ambos deben ser registros y/o celdas de memoria)

CMP: Similar a la instrucción SUB, el segundo operando se resta del primero, pero éste no almacena el resultado, solamente se modifican los bits del registro CC (Código de Condición). Es útil para comparar dos valores y generalmente se utiliza antes de una instrucción de salto condicional.

```
CMP EAX,[1000]     ;Compara los contenidos de EAX y la celda 1000
```

AND, OR, XOR: Efectúan las operaciones lógicas básicas bit a bit entre los operandos y afectan al registro CC. El resultado se almacena en el primer operando.

```
AND AX,BX          ;efectúa el AND entre AX y BX, el resultado queda en AX
```

SHL, SHR: shift left y shift right permite efectuar el desplazamiento (a izquierda y derecha) de los bits almacenados en un registro o una posición de memoria. También afectan al registro CC.

En SHL los bits derechos que quedan libres se completan con ceros.

En SHR los bits de la derecha propagan el bit anterior, es decir si el contenido es un número negativo el resultado seguirá siendo negativo, porque agregará 1, si era un número positivo agregará 0.

```
SHL AX,1           ;corre los 16 bits de AX una posición a la izquierda
                   ;No invade los 16 bits altos de EAX (equivale a multiplicar AX por 2).

SHR [200],BX       ;corre a la derecha los bits de la celda 200,
                   ;la cantidad de veces indicada en BX.
```

Instrucciones con 1 operando

SYS: ejecuta una llamada al sistema, solo puede tener un operando inmediato. (ver SEMÁNTICA DE

LOS SYSTEM CALLS)

JMP: efectúa un salto incondicional a la celda del segmento de código indicada en el operando.

```
JMP 0 ;asigna al registro IP la dirección de memoria 0 donde se almacenó la instrucción 1.
```

JZ, JP, JN, JNZ, JNP, JNN: permiten efectuar saltos condicionales en función de los bits del registro CC. Requieren de un solo operando que indica el número de línea donde se debe bifurcar.

```
JP 2        ;se salta a la celda 2 del CS si el bit de signo de CC es cero.
JN BX       ;se salta a la celda indicada en BX si el bit de signo de CC está en 1.
JZ [8]      ;se salta a la celda indicada en la celda 8 si el bit de cero de CC es 1.
JNZ 10      ;se salta a la celda 10 si el bit de cero de CC está en cero.
```

LDH: Carga en los 2 bytes más significativos del registro AC, con los 2 bytes menos significativos del operando. Esta instrucción está especialmente pensada para poder cargar un inmediato de 16 bits, aunque también se puede utilizar con otro tipo de operando.

LDL: Carga en los 2 bytes menos significativos del registro AC, con los 2 bytes menos significativos del operando. Esta instrucción está especialmente pensada para poder cargar un inmediato de 16 bits, aunque también se puede utilizar con otro tipo de operando.

RND: Carga en el primer operando un número aleatorio entre 0 y el valor del segundo operando.

NOT: Efectúa la negación bit a bit del operando y afectan al registro CC. El resultado se almacena en el primer operando.

```
NOT [15]          ;invierte cada bit del contenido de la posición de memoria 15.
```

Instrucciones sin operandos

STOP: Detiene la ejecución del programa. No requiere parámetros.

SEMÁNTICA DE LOS SYSTEM CALLS

A continuación se detalla el significado de cada interrupción, la codificación necesaria en los registros y cómo debe responder.

SYS %1 (READ): Permite almacenar en un rango de celdas de memoria los datos leídos desde el teclado. Almacenando lo que se lee por teclado en la posición de memoria apuntada por EDX hasta CX celdas de memoria como máximo (si se ingresan más caracteres que los especificados en CX, se finaliza la lectura).

El modo de lectura depende de la configuración de cada bit del registro AX

Valor	Bit	Significado
%800	11	0: Escribe un prompt [0000]: con la dirección de memoria (en decimal). 1: No escribe el prompt.
%400	10	
%200	9	
%100	8	0: Interpreta el contenido luego del endline <Enter> según la especificación de los 4 bits menos significativos de este formato (Hexa, Octal, Decimal). 1: lee caracter a caracter, y guarda uno por celda. Y se ignoran los formatos de interpretación hexadecimal, octal y decimal.
%080	7	
%040	6	
%020	5	
%010	4	
%008	3	1: interpreta hexadecimal
%004	2	1: interpreta octal
%002	1	
%001	0	1: interpreta decimal

Ejemplo1:

Código	Pantalla

`MOV AX, %001`
`MOV EDX, 11`
`MOV CX, 2`
`SYS %1`
	[0011]: 23 <Enter>
	[0012]: 25 <Enter>

Al finalizar la lectura, las celdas 11 y 12 quedarán con los valores 23 y 25 respectivamente.

Ejemplo2:

Código	Pantalla

`MOV AX, %100`
`MOV EDX, 11`
`MOV CX, 50`
`SYS %1`
	[0011]: Hola <Enter>

Las celdas quedarán de la siguiente manera: 11 = ‘H’, 12 = ‘o’, 13 = ‘l’, 14 = ‘a’, 15 = %00

SYS %2 (WRITE): Muestra en pantalla el contenido del rango de celdas especificado. Comenzando por la celda apuntada por EDX y muestra una cantidad de celdas CX

Al igual que la lectura, la forma de escritura se configura en AX

Valor	Bit	Significado
%800	11	0: Escribe un prompt [0000]: con la dirección de memoria (en decimal). 1: No escribe el prompt.
%400	10	
%200	9	
%100	8	0: Agrega endline después de imprimir cada celda 1: Omitir endline al final
%080	7	
%040	6	
%020	5	
%010	4	1: imprime solo el byte menos significativo como **caracter**.
%008	3	1: imprime la celda de memoria en hexadecimal
%004	2	1: imprime la celda de memoria en octal
%002	1	
%001	0	1: imprime la celda de memoria en decimal

Cuando el caracter ASCII **no es imprimible** escribe un punto '.' en su lugar.

Ejemplo 1:

Código	Pantalla

`MOV [1],´H´`
`MOV [2],´o´`
`MOV [3],´l´`
`MOV [4],´a´`
`MOV [5],%00`
`MOV DX,1`
`MOV CX,4`
`MOV AX, %010`
`SYS %2`
	[0001]: H
	[0002]: o
	[0003]: l
	[0004]: a
	_

Ejemplo 2:

Código	Pantalla

`< mismo código de ejemplo 1 >`
`MOV AX, %800 //omite prompt`
`OR AX, %100 //omite endline`
`OR AX, %010 //imp caracter`
`SYS %2`
	Hola_

Ejemplo 3:

Código	Pantalla

`MOV [10], %41`
`SHL [10], 8`
`OR [10], %61`
`MOV EDX, 10`
`MOV CX, 1`
`MOV AX, %01F`
`SYS %2`
	[0010]: a %4161 @40541 16737

SYS %F (BREAKPOINT): Permite interrumpir la ejecución y mostrar resultados por pantalla.

Los breakpoints son una herramienta esencial para el programador. Resultará muy útil poder agregar *breakpoints* al código fuente para que detengan su ejecución y se pueda visualizar el estado de la Máquina Virtual, especialmente cuando los programas se vuelvan más complejos.

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

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

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

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

```
[0005] cmd:
```

  Donde [0005] es la dirección de la celda de la instrucción ejecutada, y:
  - Si el usuario escribe **r** y enter, la MV continúa la ejecución.
  - Si el usuario escribe **p** y enter, la máquina ejecuta la próxima instrucción y repite la ejecución del breakpoint. (ejecución paso a paso)
  - Si el usuario escribe un número decimal entero positivo, la MV muestra el valor de la celda tomando el número ingresado por el usuario como la dirección absoluta de la memoria. y se muestra con el siguiente formato:

```
[0005] cmd:302
[0302]: 0000 007B 123
```

  Entre [ ] va la dirección de memoria en base 10, a continuación el valor de la celda en hexadecimal y decimal.

  - Si el usuario escribe 2 números decimales positivos la máquina mostrará los valores en hexadecimal y decimal del rango de celdas tomando las direcciones absolutas.

```
[0005] cmd:302 305
[0302]: 0000 007B 123
[0303]: 0000 01C8 456
[0304]: 0000 0009 9
[0305]: FFFF FF85 -123
```

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

- ***-d*** hace el disassembler al inicio de la ejecución y cada vez que se ejecuta un breakpoint indicando la posición del IP. Es decir que muestre una porción del código desensamblado (10 líneas donde se encuentra el IP), junto al valor de los registros y **señalando con ">" en la línea donde detuvo la ejecución**.
  Por ejemplo:

```
Código:
 [0000]: 04 00 E3 E9  1: MOV       EEX, 1001
 [0001]: 04 00 B0 64  2: MOV       EBX, 100
 [0002]: 14 00 E0 0B  3: ADD       EEX, EBX
 [0003]: 04 00 10 0E  4: MOV       [1], EEX
 [0004]: F0 00 00 0F  5: SYS        15
>[0005]: 04 00 A9 01  6: MOV       EAX, 2305
 [0006]: 04 00 D0 01  7: MOV       EDX, 1
 [0007]: 04 00 C0 01  8: MOV       ECX, 1
 [0008]: F0 00 00 02  9: SYS         2
Registros:
 DS =           9 |                  |                  |                  |
                  | IP =           5 |                  |                  |
 CC =           0 | AC =           0 |EAX =        2305 |EBX =         100 |
ECX =           0 |EDX =           0 |EEX =        1101 |EFX =           0 |
[0005] cmd: 10
[0010]: 1100
[0005] cmd:_
```

FORMATO DE ENTREGA Y EVALUACIÓN

El día de la evaluación 18/04/2022

- Cada grupo deberá tener en su PC (o una del laboratorio) **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.
- No aprobar la primera parte del trabajo, no implica desaprobar. Cuando se realice el coloquio de la segunda parte se pueden volver a evaluar los códigos de la primera parte (o similares), como 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**.