Organización de Computadoras 2004

Apunte 4: Lenguaje Assembly

Introducción

Las computadoras son máquinas diseñadas para ejecutar las instrucciones que se le indican, de manera de resolver problemas o hacer algún otro tipo de cosas. Dichas instrucciones básicas no suelen ser más complicadas que realizar una suma o resta, una comparación, leer un dato o escribir un resultado en la memoria. Además, lo normal es que cada instrucción esté almacenada mediante un código en la memoria, de forma que la computadora "ve" esos números y los entiende como instrucciones. También, estos códigos son específicos de cada tipo de CPU. A este conjunto de instrucciones codificadas se le llama lenguaje de máquina y es el único lenguaje que una computadora entiende.

Ahora bien, como programador, no resulta reconfortante pensar y escribir programas como una secuencia de números, de la manera en que la computadora los quiere ver, sino que es natural verlos como una secuencia de instrucciones. Para ello, se define un lenguaje, en principio, que es un mapeo directo de esos códigos a instrucciones comprensibles por un humano. A este lenguaje se lo denomina ***assembly*** o ***lenguaje de ensamble*** y al programa encargado de tomar programas escritos en assembly y generar los códigos que la computadora entenderá se lo denomina ***assembler*** o ***ensamblador***. Como veremos, no todas las instrucciones de assembly se corresponden directamente con instrucciones de lenguaje de máquina, sino que varias son instrucciones para el ensamblador mismo.

El Simulador

Antes de comenzar con algunos ejemplos de programas escritos en assembly, veremos una breve descripción de un programa cuyo propósito es simular una CPU y mostrar los pormenores de su funcionamiento. Este programa se llama **MSX88** y simula una computadora basada en una versión simplificada del célebre procesador 8086 de Intel, denominada **SX88**. Gráficamente se muestran los flujos de información existentes entre los diversos elementos que lo componen. Utiliza como plataforma de sistema operativo DOS, pudiéndose utilizar desde una ventana DOS bajo Windows. En la Figura 1 se muestra la pantalla inicial.

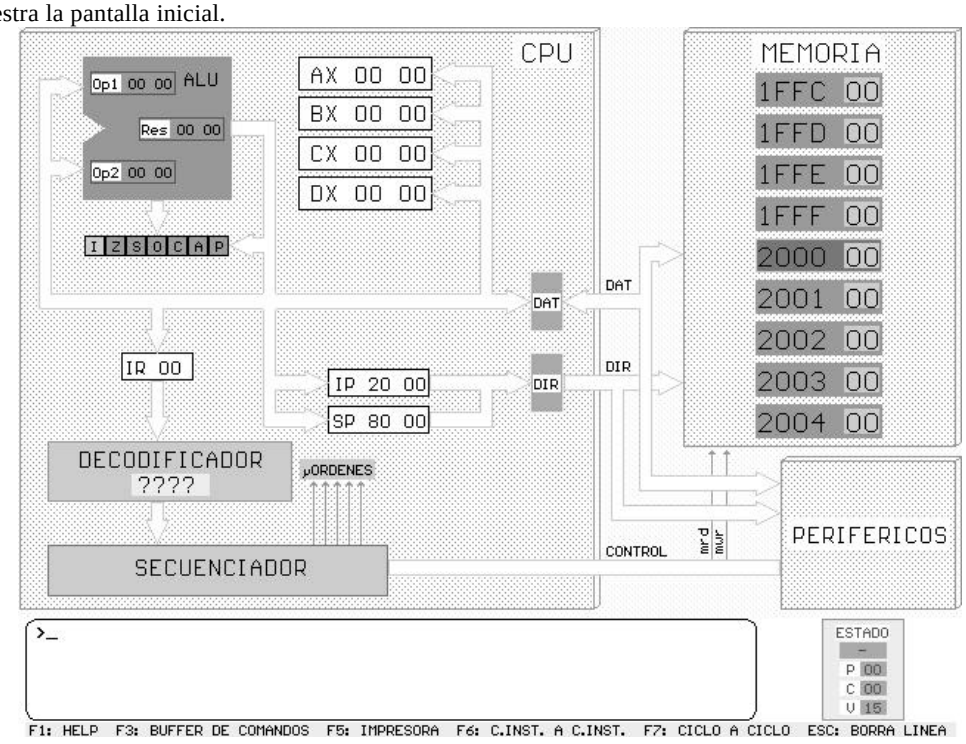

Figura 1

En la Figura 1 podemos distinguir los siguientes bloques:

- CPU
  - ALU
  - Registros AX, BX, CX, DX, SP, IP, IR.
  - Registro de Flags
  - Decodificador
  - Secuenciador
- Memoria principal
- Periféricos
- Bus de datos y de direcciones
- Toma de entrada de comandos

La CPU es la encargada de ejecutar un programa contenido en la memoria instrucción por instrucción.
La ALU es la encargada de ejecutar las operaciones aritméticas y lógicas entre los registros temporarios Op1 y Op2, dejando el resultado en Res. Dichas operaciones serán las dispuestas por la instrucción en ejecución
Los registros AX, BX, CX y DX son de uso general, 16 bits de longitud, se pueden dividir en 2 partes de 8 bits cada uno. Ejemplo: AX en AH y AL.
El registro IP (*instrucción pointer*) contiene la dirección de memoria de la próxima instrucción a ser ejecutada.
El registro SP (*stack pointer*) contiene la dirección de memoria del tope de la pila.
El registro de flags nos mostrará el estado de las banderas o flags luego de cada operación. Son 8 bits que indican el estado de las correspondientes 8 banderas. De estas 8 se utilizaran en la práctica las correspondientes a:

- Bandera de cero: identificada por la letra Z.
- Bandera de overflow: identificada por la letra O.
- Bandera de carry/borrow: identificada por la letra C.
- Bandera de signo del número: identificada por la letra S.

La entrada de comandos es el lugar donde se introducirán los comandos del simulador una vez iniciada la aplicación.
El bloque de periféricos engloba la entrada-salida del simulador.

Modo de uso

Par tener una primera toma de contacto con el programa se recomienda arrancar la demostración incluida en el archivo de distribución de MSX88. Asegúrese que los archivos DEMOMEMO.EJE, DEMOPIO.EJE y DEMODMA.EJE estén en el mismo directorio que DEMO.EXE. Se supone que todos los archivos del MSX88 se encuentran en el directorio SIMULADOR dentro del directorio raíz del disco C. Para arrancar la demo se debe teclear:

```
C:\SIMULADOR> DEMO <Enter>
```

Este programa ofrece una introducción muy detallada a las características principales de MSX88.
Ejecutando el programa DEMO, puede finalizar la ejecución pulsando varias veces la tecla 2.
Para arrancar el programa MSX88 ejecute el comando DOS:

```
C:\SIMULADOR> MSX88 <Enter>
```

Aparecerá la pantalla de la Figura 1. En la parte inferior, solo aparecerá el prompt (>), en donde se introducirán los comandos. MSX88 tiene un programa monitor que controla todas las partes de la máquina virtual. El primer comando que se debe conocer es, como en la mayoría de los sistemas operativos, el de help. Para invocar al help del monitor puede pulsar la tecla F1, teclear "help", o simplemente "h" que aparecerán escritos en la ventana de comandos del monitor.

Elaboración y ejecución de un programa de prueba

Primero se deberá editar el programa, escrito en el lenguaje de ensamble de MSX88 en un editor de texto como el EDIT o el NOTEPAD. En caso de utilizar el NOTEPAD, luego de la última línea del programa (END) se debe teclear un Enter, si no dará error al ensamblarlo. El programa fuente así generado se guardara con extensión .asm (prueba.asm por ejemplo). Se deberá usar una sintaxis determinada.
Luego habrá que ensamblarlo con el ensamblador del entorno: el ASM88. Esto se realiza con el comando

```
C:\SIMULADOR> ASM88 prueba.asm <Enter>
```

ASM88 producirá dos archivos: objeto con extensión ".o" (prueba.o) y, opcionalmente un archivo listado con extensión ".LST" (prueba.lst). Este proceso podrá dar errores en caso de que el programa fuente los tenga, por lo que habrá que editarlo nuevamente, corregirlos, volver a ensamblar, y así hasta que se eliminen los mismos.
Cuando haya conseguido ensamblar sin errores deberá utilizar el programa enlazador para generar el programa ejecutable de extensión ".EJE". LINK88 no es realmente un enlazador, dado que no presenta funcionalidades de tal, ya que no enlaza otros programas objeto al que tenemos, pero se ha considerado interesante que el alumno vaya adquiriendo unos hábitos que le ayuden en su trabajo con las herramientas profesionales que si cuentan con él. Esto se realiza con el comando

```
C:\SIMULADOR> LINK88 prueba.o <Enter>
```

Si no se cometieron errores, debemos tener en el mismo directorio el archivo prueba.eje. Para cargarlo en la memoria de MSX88 se ha de ejecutar el siguiente comando dentro del área de comandos del MSX88 (ojo, no es un comando de DOS):

```
> L prueba <Enter>
```

Por último, para ejecutar el programa, se utiliza el comando

```
> G <Enter>
```

Podremos observar en la pantalla como se obtienen instrucciones, se las decodifican, como se producen movimientos de datos, operaciones aritméticas, etc. Si se desea volver a ejecutar el programa, se deberá modificar el registro IP para que apunte a la dirección de comienzo del programa con el comando

```
> R IP 2000 <Enter>
```

que coloca en el registro IP (dirección de la próxima instrucción) la dirección de la celda donde está la primer instrucción del programa (por defecto, 2000h en el simulador).
Éstos y otros comandos más se encuentran detallados en el manual del simulador.

Directivas para el ensamblador

Como se había mencionado en un principio, un programa en assembly no está comprendido únicamente por instrucciones del lenguaje de máquina sino que existen otro tipo de instrucciones que le indican al ensamblador como realizar algunas tareas.
Un hecho notable cuando se trata del uso de variables es que en las instrucciones del lenguaje de máquina no se hace referencia a ellas por un nombre sino por su dirección en la memoria. Para evitarnos la muy poco grata tarea de recordar que tal variable está almacenada en una dirección particular (que, de hecho, podría cambiar a lo largo del desarrollo de un programa, forzándonos a revisar el programa entero para realizar los ajustes necesarios), en assembly existen instrucciones para definir variables. La sintaxis para definir una variable es la siguiente:

```
nombre_variable  especificador_tipo  valor_inicial
```

El nombre de la variable debe comenzar con una letra o un underscore (_) seguido por números, letras o underscores. El tipo indica el tamaño que ocupa en memoria dicha variable y puede ser alguno de los enumerados en la Tabla 1. El valor inicial puede no especificarse, usando el carácter '?', lo que le informa al ensamblador que dicho valor será el que se encuentre en la memoria del simulador en el momento de cargar el programa.

Especificador	Tipo	Tamaño
DB	Byte	8 bits
DW	Word	16 bits

Tabla 1: Especificadores de los tipos de variables.

Ejemplos de definición de variables:

```
Var1 DB 10
Var2 DW 0A000h
```

Podemos observar que los valores numéricos se interpretan en decimal, a menos que terminen con una letra 'h', que en cuyo caso se interpretarán como valores en hexadecimal. Además, como los números deben comenzar con un dígito decimal, en el caso del A000h, se antepone un cero para evitar que se la confunda con una variable llamada A000h.

A veces resulta bueno poder definir valores constantes. Esto se hace del siguiente modo:

```
nombre  EQU  valor
```

Debe notarse que en este caso el ensamblador reemplazará cualquier ocurrencia de 'nombre' por el valor indicado, pero que dicho valor no va a ocupar ninguna dirección de memoria, como lo hacen las variables.

Es posible extender la definición de variables para incluir la idea de tablas, de la siguiente manera:

```
nombre_variable  especificador_tipo  valores
```

En la definición anterior, *valores* es una lista de datos del mismo tipo de la variable separados por coma:

```
tabla DB  1,  2,  4,  8, 16, 32, 64, 128
```

Esto genera una tabla con los ocho valores especificados, uno a continuación del otro. Esto se puede ver como un arreglo de ocho bytes pero en el que se inicializaron sus celdas con dichos valores. Por otro lado, si quisiéramos definir algo equivalente a un string, podemos aplicar la misma idea de la tabla anterior, en donde en cada celda se almacenaría cada carácter del string. Sin embargo, escribir los códigos ASCII de cada carácter no simplifica mucho las cosas, así que existe una sintaxis alternativa:

```
string DB "Esto es un String."
```

Si quisiéramos definir una tabla en la que los datos que contienen son iguales o cumplen algún patrón repetitivo, es posible utilizar el modificador DUP en la lista de valores, de la siguiente manera:

```
cantidad DUP (valores)
```

En este caso, *cantidad* indica la cantidad de veces que se repiten el o los valores indicados entre paréntesis. Por ejemplo, para definir un arreglo de 20 palabras, inicialmente conteniendo 1234h y 4321h alternadamente, se le indica al ensamblador lo siguiente:

```
cantidad EQU 10
arreglo  DW cantidad DUP (1234h, 4321h)
```

Otra cuestión que se obvió hasta el momento es que dirección se le asigna a cada variable. El ensamblador lleva cuenta de las variables e instrucciones que va procesador y de la dirección que le corresponde a cada una, dependiendo de una dirección inicial y del tamaño correspondiente. Existe una directiva del ensamblador, llamada ORG, que permite cambiar sobre la marcha la dirección a partir de la cual se colocarán las cosas que estén a continuación de la misma. La sintaxis de la misma es:

```
ORG dirección
```

El uso de ORG se ejemplifica a continuación:

```
ORG 1000h

      contador DW 1234h
      cantidad DB 0

ORG 2000h

      arreglo DB 0A0h, 3 DUP (15)

      cadena  DB
"Un string es un
arreglo de
bytes."

      END
```

1000h	contador	34h
1001h	contador	12h

1002h	cantidad	00h
2000h	arreglo	A0h
2001h	arreglo	0Fh
2002h	arreglo	0Fh
2003h	arreglo	0Fh
2004h	cadena	U
2005h	cadena	n

2006h	cadena	
2007h	cadena	s
2008h	cadena	t

En el ejemplo anterior, el primer ORG le indicará al ensamblador que todo lo que esté a continuación deberá ubicarse a partir de la dirección 1000h. Por eso, el contenido de la variable "contador" estará almacenado en las direcciones 1000h y 1001h, pues es un valor de 16 bits. A continuación de "contador", se almacenará "cantidad", a la que le toca la dirección 1002h.
El siguiente ORG indica que lo que se encuentre a continuación de él, será ubicado a partir de la dirección 2000h. Por ello, "arreglo" se almacena desde 2000h hasta 2003h y "cadena" comienza en la dirección 2004h. En la tabla a la derecha del ejemplo podemos ver como se ubican las variables definidas en la memoria: "arreglo" ocupa 4 bytes, uno por el primer valor de la lista y tres más por duplicar tres veces al valor entre paréntesis; "cadena" ocupa tantos bytes como caracteres contenga el string.

Hasta aquí vimos como definir constantes y variables de distintos tipos y como indicarle al ensamblador en que dirección ubicarlas. Ahora veremos como escribir las instrucciones que darán forma al programa.

Instrucciones

Recordemos que el procesador que utiliza el simulador MSX88, llamado SX88, cuenta con varios registros de propósito general (AX, BX, CX y DX), a los que podemos tratar como registros de 16 bits o como un par de registros de 8 bits, tomando la parte baja separada de la parte alta. En el caso de AX, tendríamos AH (parte alta o *high* de AX) y AL (parte baja o *low* de AX), a BX como BH y BL, etc. Además existe un registro llamado IP, que contiene la dirección de la próxima instrucción a ejecutar, y otro llamado SP, que contiene la dirección del tope de la pila.

Instrucciones de transferencia de datos

La primera instrucción que veremos es la que permite realizar movimientos de datos, ya sea entre registros o desde un registro a la memoria y viceversa. Debe notarse que no es posible realizar movimientos de datos desde una parte a otra de la memoria mediante una única instrucción. La instrucción en cuestión se llama MOV (abreviatura de MOVE, *mover* en inglés) y tiene dos operandos:

```
MOV destino, origen
```

El valor contenido en *origen* será asignado a *destino*. La única restricción es que tanto *origen* como *destino* sean del mismo tipo de datos: bytes, words, etc. Ejemplo:

```
ORG 1000h

      var_byte DB 20h
      var_word DW ?

ORG 2000h

      MOV AX, 1000h
      MOV BX, AX
      MOV BL, var_byte
      MOV var_word, BX

END
```

Instante	AX		BX	
	AH	**AL**	**BH**	**BL**
0	00	00	00	00
1	**10**	**00**	00	00
2	10	00	**10**	**00**
3	10	00	10	**20**
4	10	00	10	20

Aquí vemos el uso más común de la directiva ORGL: La idea es separar las variables del programa. En el ejemplo, las variables serán almacenadas a partir de la dirección 1000h mientras que las instrucciones del programa estarán a partir de la dirección 2000h. Además, como el simulador, por defecto, comienza ejecutando lo que está contenido en la dirección 2000h, eso hará que nuestro programa comience en la primera instrucción MOV.

Dicha instrucción asigna el valor inmediato 1000h al registro AX. Esta instrucción emplea el modo de direccionamiento conocido como "inmediato". Como ambos son valores de 16 bits, no hay inconveniente en esa asignación. En la tabla a la derecha del ejemplo se muestran el contenido de los registros AX y BX. Inicialmente, ambos estaban en cero (instante 0) y luego de la primer instrucción MOV, el registro AX cambia (instante 1, resaltado el cambio en negrita).

El siguiente MOV asigna el contenido del registro AX al registro BX (instante 2). De nuevo, como ambos son de 16 bits, es una asignación válida. El modo de direccionamiento que usa es el denominado "registro".
El tercer MOV asigna el contenido de la variable "var_byte" (que es 20h) al registro BL. Como BL es la parte baja de BX, el cambio que se produce en BX es el indicado en la tabla: pasa de 1000h a 1020h (instante 3). Como BL y "var_byte" son ambos de 8 bits, es una asignación permitida. Además, se emplea el modo de direccionamiento "directo".

El último MOV asigna el valor contenido en el registro BX a la dirección de memoria a la que "var_word" hace referencia. Ahora, "var_word" contiene el valor 1020h.

Mover datos de un lugar a otro de la memoria resulta poco útil si no podemos trabajar sobre ellos. Si recordamos que la CPU cuenta con una ALU capaz de realizar operaciones aritméticas y lógicas, deben existir instrucciones para utilizarlas, las cuales se detallarán a continuación.

Instrucciones aritméticas

Existen dos operaciones aritméticas básicas que puede realizar la CPU: sumar y restar. Para cada una de ellas existen instrucciones correspondientes: ADD (*sumar* en inglés) y SUB (*restar* en inglés). El formato de estas instrucciones es el siguiente:

```
ADD operando1, operando2
SUB operando1, operando2
```

En principio, parecería que falta algo, ya que se indican con que valores se va a operar pero no en donde va a quedar almacenado el resultado de la operación. En realidad, esto está implícito en la instrucción, puesto que el resultado se almacenará en *operando1* (recordar la idea de máquina de dos direcciones). En otras palabras, estas instrucciones hacen lo siguiente (los paréntesis indican *"el contenido de"*):

```
ADD: (operando1) ← (operando1) + (operando2)
SUB: (operando1) ← (operando1) - (operando2)
```

Esto implica que el valor contenido en operando1 es reemplazado por el resultado de la operación, por lo que si dicho valor se utilizará más tarde, es necesario operar sobre una copia del mismo. Por ejemplo:

```
ORG 1000h
      dato1     DW 10
      dato2     DW 20
      resultado DW ?

ORG 2000h
      MOV AX, dato1
      ADD AX, dato2
      MOV resultado, AX
END
```

En el ejemplo anterior, se utiliza el registro AX como variable auxiliar, de manera de no afectar a *dato1*.

Existen dos instrucciones más similares a las anteriores pero que tienen en cuenta al flag de *carry/borrow*:

```
ADC operando1, operando2
SBB operando1, operando2
```

La semántica de dichas instrucciones es la siguiente:

```
ADC: (operando1) ← (operando1) + (operando2) + (C)
SBB: (operando1) ← (operando1) - (operando2) - (C)
```

C es el valor del flag de *carry/borrow*, que puede ser 0 o 1. En otras palabras, estas instrucciones suman o restan incluyendo a dicho flag, por lo que resultan útiles para encadenar varias operaciones de suma. Supongamos que queremos sumar valores de 32 bits. Dado que nuestra CPU opera con valores de 8 o 16 bits, no sería posible hacerlo en un solo paso. Sin embargo, podríamos sumar la parte baja (los 16 bits menos significativos) por un lado y la parte alta (los 16 bits más significativos) por otro usando dos instrucciones ADD. El problema se presenta cuando se produce un acarreo al realizar la suma en la parte baja, ya que no podemos simplemente ignorarlo pues el resultado no sería el correcto, como se muestra en este ejemplo:

```
  Correcto:          Incorrecto:
  0015 FFFF          0015    FFFF
+ 0002 0011        + 0002  + 0011
  0018 0010          0017    1 0010   ———— 0017 0010
```

Para resolver ese problema, utilizamos estas instrucciones que acabamos de ver de la siguiente manera:

```
ORG 1000h
      dato1_l   DW 0FFFFh
      dato1_h   DW  0015h
      dato2_l   DW  0011h
      dato2_h   DW  0002h

ORG 2000h
      MOV AX, dato1_l
      ADD AX, dato2_l
      MOV BX, dato1_h
      ADC BX, dato2_h
END
```

Instante	AX	BX	Flag C
0	0000	0000	0
1	**FFFF**	0000	0
2	**0010**	0000	**1**
3	0010	**0015**	1
4	0010	**0018**	**0**

Nótese que la segunda suma es realizada usando un ADC y no un ADD. De esta manera, si en el ADD se produce un acarreo, éste es sumado junto a *dato1_h* y *dato2_h* durante el ADC, produciendo el resultado correcto. Esta misma idea puede aplicarse a la resta usando SUB y SBB.

Existen un par de instrucciones que son casos particulares del ADD y del SUB, llamadas INC y DEC, que poseen un solo operando y simplemente le suman o restan 1 a dicho operando.

```
INC operando
DEC operando

INC: (operando) ← (operando) + 1
DEC: (operando) ← (operando) - 1
```

Estas instrucciones existen porque ocupan menos bits en memoria que su contrapartida empleando ADD o SUB, por lo que suelen ejecutarse más velozmente.

Finalmente, las mismas restricciones que se imponen al MOV se aplican a estas instrucciones: Los operandos deber ser del mismo tipo y no es posible hacer referencias a memoria en los dos operandos al mismo tiempo; siempre deben moverse datos entre registros o entre un registro y la memoria.

Instrucciones lógicas

Las operaciones lógicas que posee la CPU del simulador son las siguientes:

```
AND operando1, operando2
OR  operando1, operando2
XOR operando1, operando2
NOT operando
NEG operando
```

Al igual que el ADD y el SUB, el resultado de las instrucciones AND, OR y XOR queda almacenado en el primer operando. Las primeras tres instrucciones realizan la operación lógica correspondiente aplicada bit a bit sobre todos los bits de ambos operandos. NOT y NEG calculan sobre el operando el complemento a 1 y a 2 respectivamente.

Instrucciones de comparación

Esta CPU cuenta con una instrucción de comparación llamada CMP (abreviatura de COMPARE, *comparar* en inglés). CMP es esencialmente equivalente en funcionamiento a SUB, con excepción de que el resultado de la resta no se almacena en ninguna parte, por lo que ninguno de los operandos se ve alterado. Esta instrucción da la impresión de que no hace nada, pero en realidad, al hacer la resta, causa que se modifiquen los flags. Ver como quedaron los flags luego de una resta nos dice muchas cosas sobre los números que se restaron. Por ejemplo, si el flag Z queda en 1, el resultado de la resta fue cero, con lo que podemos concluir que los dos números que se restaron eran iguales. Si, en cambio, el flag S (signo) quedó en 1 y vemos a los operandos como números en CA2, podemos inferir que el segundo operando era mayor que el primero, puesto que al restarlos nos da negativo.
Existen combinaciones de los flags que nos indican si un número es mayor, mayor o igual, menor, menor o igual, igual o distinto a otro. Estas combinaciones pueden depender del sistema en el que interpretamos a los números, normalmente, en BSS o CA2.

Una instrucción CMP por si sola es bastante inútil pero adquiere su poder al combinarla con instrucciones de salto condicionales, que se verán a continuación.

Instrucciones de salto

Las instrucciones de salto, como indica su nombre, permiten realizar saltos alterando el flujo de control a lo largo de la ejecución de un programa. Estas instrucciones tienen un operando que indica la dirección que se le asignará al registro IP. Al alterar este registro, se modifica cual va a ser la próxima instrucción que va a ejecutar la CPU. Por otro lado, tenemos dos tipos de instrucciones de salto: los saltos incondicionales y los saltos condicionales. Los primeros se producen siempre que se ejecuta la instrucción mientras que los segundos dependen de alguna condición para que se produzca o no dicho salto.
A continuación se detallan las instrucciones de salto que el SX88 posee, junto con la condición por la cual se realiza el salto:

```
      JMP dirección     ; Salta siempre

      JZ  dirección     ; Salta si el flag Z=1
      JNZ dirección     ; Salta si el flag Z=0
      JS  dirección     ; Salta si el flag S=1
      JNS dirección     ; Salta si el flag S=0
      JC  dirección     ; Salta si el flag C=1
      JNC dirección     ; Salta si el flag C=0
      JO  dirección     ; Salta si el flag O=1
      JNO dirección     ; Salta si el flag O=0
```

Al igual que lo que ocurría con las variables, el ensamblador nos facilita indicar la dirección a la que se quiere saltar mediante el uso de una etiqueta, la cual se define como un nombre de identificador válido (con las mismas características que los nombres de las variables) seguido de dos puntos, como se muestra en este ejemplo:

```
ORG 2000h
      MOV AX, 10

Lazo: ... ;
      ... ; <Instrucciones a repetir>
      ... ;

      DEC AX
      JNZ Lazo
```

```
Fin:  JMP Fin

END
```

En el ejemplo podemos ver que se definen dos etiquetas: *Lazo* y *Fin*. También se ve en acción a un par de instrucciones de salto: JMP y JNZ. El programa inicializa AX en 10, hace lo que tenga que hacer dentro del bucle, decrementa AX en 1 y salta a la instrucción apuntada por la etiqueta *Lazo* (el DEC) siempre y cuando el flag Z quede en 0, o sea, que el resultado de la operación anterior no haya dado como resultado 0. Como AX se decrementa en cada iteración, llegará un momento en que AX será 0, por lo que el salto condicional no se producirá y continuará la ejecución del programa en la siguiente instrucción luego de dicho salto. La siguiente instrucción es un salto incondicional a la etiqueta *Fin*, que casualmente apunta a esa misma instrucción, por lo que la CPU entrará en un ciclo infinito, evitando que continúe la ejecución del programa. Nótese que sin esta precaución la CPU continuará ejecutando lo que hubiera en la memoria a continuación del fin del programa. La palabra END al final del programa le indica al ensamblador que ahí finaliza el código que tiene que compilar, pero eso no le dice nada a la CPU. Más adelante, veremos como subsanar este inconveniente.
El ejemplo anterior plantea un esquema básico de iteración usado comúnmente como algo equivalente a un "`for i := N downto 1`" del pascal.

Otras instrucciones

Algo poco elegante en el ejemplo del FOR es hacer que la CPU entre en un bucle infinito para detener la ejecución del programa. Existe una manera más elegante de hacer esto y es pedirle gentilmente a la CPU que detenga la ejecución de instrucciones mediante la instrucción HLT (HALT, *detener* en inglés).
Existe otra instrucción que no hace nada. Si, es correcto, no hace nada, simplemente ocupa memoria y es decodificada de la misma manera que cualquier otra instrucción, pero el efecto al momento de ejecutarla es no hacer nada. Dicha instrucción es NOP.

Cabe mencionar que el SX88 cuenta con otras instrucciones que no se detallan en este apunte, ya que escapan a los contenidos propuestos para este curso. Sin embargo, el lector interesado puede recurrir al manual del simulador MSX88, en donde se enumeran todas las instrucciones que dicho simulador soporta.

Modo de direccionamiento indirecto

Existe un modo de direccionamiento que aún no se mencionó y que facilita el recorrido de tablas. Estamos hablando del modo de direccionamiento indirecto vía el registro BX y se ilustra el uso de éste en el siguiente ejemplo:

```
ORG 1000h
      tabla     DB 1, 2, 3, 4, 5                    ; (1)
      fin_tabla DB ?                                ; (2)
      resultado DB 0                                ; (3)

ORG 2000h
      MOV BX, OFFSET tabla                          ; (4)
      MOV CL, OFFSET fin_tabla – OFFSET tabla       ; (5)

Loop: MOV AL, [BX]                                  ; (6)
      INC BX                                        ; (7)
      XOR resultado, AL                             ; (8)
      DEC CL                                        ; (9)
      JNZ Loop                                      ; (10)

      HLT                                           ; (11)
END
```

En este ejemplo se introducen varias cosas además del modo de direccionamiento indirecto.
El ejemplo comienza definiendo una tabla (1) y dos variables más (2) y (3).
Luego, en (4), comienza inicializando BX con "*OFFSET tabla*". Esto indica que se debe cargar en BX la dirección de *tabla*, no el contenido de dicha variable.

En (5) se asigna a CL la diferencia entre la dirección de *tabla* y la dirección de *fin_tabla*. Si meditamos un segundo sobre esto, veremos que lo que se logra es calcular cuantos bytes hay entre el comienzo y el final de la tabla. De esta manera, obtenemos la cantidad de datos que contiene la tabla.
En (6) vemos que en el MOV aparece BX entre corchetes. Esto significa que se debe asignar en AL el contenido de la celda de memoria cuya dirección es el valor contenido en BX. Así, como BX se había inicializado con la dirección del comienzo de la tabla, esto causa que se cargue en AL el contenido de la primer entrada de la tabla.
En (7) se incrementa BX, con lo que ahora apunta al siguiente byte de la tabla.
En (8) se calcula una operación XOR con el contenido de la variable *resultado* y el byte que se acaba de obtener en AL, dejando el resultado de esa operación en la misma variable.
(9) y (10) se encargan de iterar saltando a la etiqueta Loop mientras CL sea distinto de 0, cosa que ocurrirá mientras no se llegue al final de la tabla. Cuando esto ocurra, no se producirá el salto condicional y la ejecución seguirá en la próxima instrucción a continuación de esta. Dicha instrucción (11) es un HLT que detiene la CPU.

Ejemplos

A continuación veremos una serie de ejemplos que ilustran el uso de las instrucciones que se vieron a lo largo del apunte y también como codificar las estructuras de control que son comunes en los lenguajes de alto nivel pero que no existen en assembly.

Selección: IF THEN ELSE

No existe una instrucción en assembly que sea capaz de hacer lo que hace la estructura IF-THEN-ELSE de pascal. Sin embargo, es posible emularla mediante la combinación de instrucciones CMP y saltos condicionales e incondicionales. Por ejemplo, intentemos simular el siguiente código de pascal:

```
IF AL = 4 THEN
BEGIN
  BL = 1;
  CL = CL + 1;
END;
```

La idea es comenzar calculando la condición del IF, en este caso, comparar AL con 4. Eso se logra con una instrucción CMP:

```
CMP AL, 4
```

Esta instrucción alterará los flags y en particular, nos interesa ver al flag Z, ya que si dicho flag está en 1, implica que al resta AL con 4, el resultado dio 0, por lo que AL tiene que valer 4. Entonces, si esa condición es verdadera, deberíamos ejecutar las instrucciones que están dentro del THEN. Si no, deberíamos evitar ejecutarlas. Una solución simplista es usar saltos, del siguiente modo:

```
        CMP AL, 4     ; (1)
        JZ Then       ; (2)
        JMP Fin_IF    ; (3)

Then:   MOV BL, 1     ; (4)
        INC CL        ; (5)

Fin_IF: HLT           ; (6)
```

Analizando el código, vemos lo siguiente:
Si la comparación en (1) establece el flag Z en 1, el salto en (2) se produce, haciendo que la ejecución continúe en la etiqueta *Then*. Ahí, se ejecutan las instrucciones (4) y (5) que hacen lo que se encuentra en el THEN del IF y continúa la ejecución en la instrucción apuntada por la etiqueta *Fin_IF*.
Si el flag Z quedó en 0 en (1), el salto en (2) no se produce, por lo que la ejecución continúa en la próxima instrucción, el JMP en (3), que saltea las instrucciones y continúa la ejecución en la instrucción apuntada por la etiqueta *Fin_IF*, que señala el final del IF.
En el final del IF, se ejecuta un HLT para terminar la ejecución del programa.

El ejemplo anterior se puede mejorar un poco más. El par JZ/JMP en (2) y (3) pueden reemplazarse por un "`JNZ Fin_IF`", ya que si no se cumple la condición, se omite la parte del THEN y continúa la ejecución al final del IF. Estas "optimizaciones" contribuyen a un código más compacto y, en consecuencia, más eficiente. Sin embargo, como toda optimización, atenta contra la claridad del código.

```
        CMP AL, 4     ; (1)
        JNZ Fin_IF    ; (2)

Then:   MOV BL, 1     ; (3)
        INC CL        ; (4)

Fin_IF: HLT           ; (5)
```

El ejemplo fue planteado de esa manera, con instrucciones aparentemente redundantes, porque así como está, es más sencillo extenderlo para codificar una estructura IF-THEN-ELSE completa. Por ejemplo, si queremos simular el siguiente código de pascal:

```
      IF AL = 4 THEN
        BEGIN
          BL = 1;
          CL = CL + 1;
        END
      ELSE
        BEGIN
          BL = 2;
          CL = CL – 1;
        END;
```

Como ahora tenemos las dos alternativas, el código equivalente en assembly podría ser así:

```
        CMP AL, 4     ; (1)
        JZ Then       ; (2)
        JMP Else      ; (3)

Then:   MOV BL, 1     ; (4)
        INC CL        ; (5)
        JMP Fin_IF    ; (6)

Else:   MOV BL, 2     ; (7)
        DEC CL        ; (8)

Fin_IF: HLT           ; (9)
```

La cuestión ahora es que en (2) y (3) se decide que parte del IF se ejecutará, el THEN o el ELSE. Además, es necesario un salto en (6) para que una vez finalizada la ejecución del THEN, se sigua con la próxima instrucción luego del fin del IF. El resto es equivalente al ejemplo anterior.

Es posible eliminar el salto JMP (3) intercambiando las partes del THEN y del ELSE, como se ilustra a continuación:

```
        CMP AL, 4     ; (1)
        JZ Then       ; (2)

Else:   MOV BL, 2     ; (3)
        DEC CL        ; (4)
        JMP Fin_IF    ; (5)

Then:   MOV BL, 1     ; (6)
        INC CL        ; (7)

Fin_IF: HLT           ; (8)
```

Este último ejemplo muestra la forma más compacta de generar la estructura IF-THEN-ELSE mediante instrucciones de assembly.

Iteración: FOR, WHILE, REPEAT-UNTIL

Ya vimos como se codifica en assembly una estructura IF-THEN-ELSE. Ahora veremos como implementar un FOR, un WHILE o un REPEAT-UNTIL. En el caso del primero, ya algo se comentó cuando se des cribieron los saltos condicionales. La idea simplemente es realizar un salto condicional al inicio del código a repetir mientras no se haya alcanzado el límite de iteraciones. Por ejemplo:

```
AL := 0;
FOR CL := 1 TO 10 DO
  AL := AL + AL;
```

Se puede implementar mediante el siguiente esquema:

```
        MOV AL, 0
        MOV CL, 1
Iterar: CMP CL, 10
        JZ Fin
        ADD AL, AL
        INC CL
        JMP Iterar
Fin:    HLT
```

Si quisiéramos hacer un *downto* en lugar de un *to*, simplemente se realizan los siguientes cambios en el código:

```
        MOV AL, 0
        MOV CL, 10
Iterar: CMP CL, 1
        JZ Fin
        ADD AL, AL
        DEC CL
        JMP Iterar
Fin:    HLT
```

Implementar un WHILE es exactamente igual al esquema anterior, solo que en lugar de evaluar si el contador llegó al valor límite, lo que se hace es evaluar la condición del WHILE. Si quisiéramos ver un REPEAT-UNTIL, la diferencia es que en éste, la condición se evalúa luego de ejecutar el código a repetir al menos una vez. Esto lo logramos simplemente cambiando de lugar algunas cosas del ejemplo anterior:

```
        MOV AL, 0
        MOV CL, 10
Iterar: ADD AL, AL
        DEC CL
        CMP CL, 1
        JNZ Iterar
Fin:    HLT
```

Resumiendo, todas las estructuras de iteración se resuelven de la misma manera. Lo que varía es donde y que condición se evalúa.

Arreglos y tablas

Anteriormente se mostró como definir arreglos y tablas y de que manera se inicializan. También se mencionó el modo de direccionamiento indirecto mediante el registro BX, con lo que se facilitaba recorrer la tabla definida. Veremos a continuación unos ejemplos concretos.

Supongamos que queremos encontrar el máximo número almacenado en una tabla de words. No se sabe si los números están o no en orden, pero si que son todos números positivos (BSS). Si planteamos la solución en PASCAL, obtendríamos algo como esto:

```
const
  tabla: array[1..10] of Word = {5, 2, 10, 4, 5, 0, 4, 8, 1, 9};

var
  max: Word;

begin
  max := 0;
  for i := 1 to 10 do
    if tabla[i] > max then
      max := tabla[i];
end.
```

El programa inicializa la variable *max* con el mínimo valor posible y recorre la tabla comparando dicho valor con cada elemento de la tabla. Si alguno de esos elementos resulta ser mayor que el máximo actualmente almacenado en la variable *max*, se lo asigna a la variable y sigue recorriendo. Al finalizar la iteración, en *max* queda almacenado el máximo valor de la tabla.
Una posible implementación del programa anterior en assembly sería:

```
ORG 1000h

       tabla dw 5, 2, 10, 4, 5, 0, 4, 8, 1, 9;
       max   dw 0

ORG 2000h

       MOV BX, OFFSET tabla        ; (1)
       MOV CL, 0                   ; (2)
       MOV AX, max                 ; (3)
Loop:  CMP [BX], AX                ; (4)
       JC Menor                    ; (5)
       MOV AX, [BX]                ; (6)
Menor: ADD BX, 2                   ; (7)
       INC CL                      ; (8)
       CMP CL, 10                  ; (9)
       JNZ Loop                    ; (10)
       MOV max, AX                 ; (11)
       HLT                         ; (12)
END
```

La instrucción (1) se encarga de cargar en BX la dirección del comienzo de la tabla. En (2) se inicializa en 0 el contador que va a usarse para saber cuantas iteraciones van haciendo. En (3) se carga en AX el valor almacenado en la variable *max*, que inicialmente es el mínimo posible. En (4) se compara el máximo actual, que se encuentra en AX, con el número de la tabla apuntado actualmente por BX. Si el número es mayor o igual que AX, al restarlos no habrá *borrow*. Si el número en la tabla es menor que AX, al restarlos se producirá un *borrow*, que se indicará en el flag C. Por eso, si el flag C queda en 1, en (5) salta a la etiqueta *Menor*, por lo que la ejecución continua en la instrucción (7). Si el flag C queda en 0, continua ejecutando (6), que lo que hace es reemplazar el máximo actual guardado en AX por el nuevo máximo encontrado. En ambos casos, la ejecución continúa en la instrucción (7), que se encarga de incrementar BX para que apunte al próximo número de la tabla. Como cada entrada de la tabla es de 16 bits (o dos bytes), es necesario incrementar BX en 2 para que apunte a la siguiente palabra. En (8) se incrementa el contador y en (9) se verifica que no se haya llegado al final de la iteración. Si el contador no llegó a 10, en (10) se produce el salto a la etiqueta *Loop* de manera de continuar con la iteración. Si, en cambio, el contador CL llegó a 10, el salto no se produce y continúa la ejecución en (11), instrucción que se encarga de asignar el máximo almacenado en AX a la variable *max*. Por último, el programa termina su ejecución con un HLT en (12).