Arquitectura x86-32

1. El lenguaje de la arquitectura x86-32

La arquitectura x86-32 es una de las más exitosas desde el punto de vista comercial. Tal éxito ha provocado que existan muchos tipos diferentes de CPU que entienden su lenguaje. Al igual que ocurre con el Computador Teórico, el lenguaje de la arquitectura x86-32 consta de un conjunto de instrucciones que realizan tareas básicas, tales como transferencias u operaciones aritméticas. En el caso del lenguaje de la arquitectura x86-32, al ser un caso real, existen cientos de instrucciones que permiten trabajar con distintos tipos de datos y múltiples modos de direccionamiento. En la tabla 1 se muestra una comparativa entre las características del Computador Teórico y las de la arquitectura x86-32.

El lenguaje ensamblador de la arquitectura x86-32 tiene muchas similitudes con el del Computador Teórico. En la sección 1.6 se describe este lenguaje.

1.1. Operandos

Los operandos en este lenguaje hacen referencia, igualmente, al lugar donde están almacenados los datos sobre los que operan las instrucciones. Pueden ser datos almacenados en registros, en memoria o inmediatos. La diferencia fundamental con el Computador Teórico es que en la arquitectura x86-32 se puede trabajar con distintos tamaños de datos: el byte (8 bits), la palabra o *word* (16 bits) y la doble palabra o *double word* (32 bits). Dentro de una palabra se puede acceder a su byte más significativo, MSB (*Most Significant Byte*), o a su byte menos significativo, LSB (*Least Significant Byte*). Dentro de una doble palabra se puede acceder a su palabra más

	Computador Teórico	x86-32
Ancho de la arquitectura (tamaño de los operandos manejados)	16	32
Ancho de las posiciones de memoria	16	8
Ancho de las direcciones de memoria	16	32
Tamaño del espacio de direcciones	64 Kipalabras	4 Gipalabras

Tabla 1: Comparativa entre los parámetros arquitectónicos del Computador Teórico y de la arquitectura x86-32

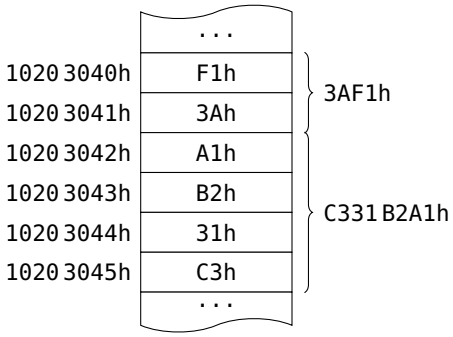

Figura 1: Almacenamiento de datos siguiendo el criterio *little-endian* en la arquitectura x86-32

significativa, MSW (*Most Significant Word*), o a su palabra menos significativa, LSW (*Least Significant Word*).

En la arquitectura x86-32 una posición de memoria almacena 8 bits. Por lo tanto, las palabras y dobles palabras no se pueden almacenar en una sola posición de memoria y se organizan ocupando un conjunto de posiciones de memoria contiguas siguiendo el criterio ***little-endian***. Este criterio se utiliza para organizar datos en memoria de modo que el byte menos significativo se sitúa en la posición de memoria más baja, y así ordenadamente hasta el byte más significativo que se sitúa en la posición más alta[1]. Por ejemplo, el dato de tipo palabra `3AF1h` es de 16 bits y, por lo tanto, ocupará en memoria dos posiciones consecutivas. Si se desea almacenar este dato a partir de la posición de memoria cuya dirección es `1020 3040h`, el byte menos significativo, es decir, `F1h`, se almacenaría en la posición de memoria más baja, la `1020 3040h`, y el más significativo, es decir, `3Ah`, en la posición de memoria consecutiva hacia direcciones crecientes, es decir, la `1020 3041h`. Para almacenar el dato de doble palabra `C331 B2A1h` a partir de la dirección `1020 3042h` se seguiría el mismo procedimiento, el byte menos significativo, `A1h`, se almacenaría en la dirección `1020 3042h`, y el resto, `B2h`, `31h` y `C3h`, de forma ordenada en las posiciones `1020 3043h`, `1020 3044h` y `1020 3045h`, respectivamente. El byte más significativo se almacenará siempre en la posición de memoria más alta. El resultado de almacenar estos datos en memoria se muestra en la figura 1.

En la arquitectura x86-32 existen una serie de registros de propósito general que se pueden utilizar para almacenar los operandos sobre los que actúan las instrucciones. Los registros reciben los siguientes nombres: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` y `ESP`. Todos estos registros son de 32 bits, pero se puede acceder a su palabra menos significativa eliminando la `E` (*Extended*) del nombre, de la forma: `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `BP` y `SP`. También se puede acceder al byte más significativo (*High*) de los registros `AX`, `BX`, `CX` y `DX` como `AH`, `BH`, `CH` y `DH`, o a su byte menos significativo (*Low*) como `AL`, `BL`, `CL` y `DL`, respectivamente. Se debe tener en cuenta el tamaño del dato con el que se opera para utilizar el registro adecuado en cada caso.

[1]El criterio análogo se denomina *big-endian*. Este criterio determina que el byte más significativo del dato se sitúa en la posición de memoria más baja, y así ordenadamente hasta el byte menos significativo que se sitúa en la posición de memoria más alta.

1.2. Sentencias de asignación

No existen diferencias significativas en la forma en la que se traducen las sentencias en lenguaje de alto nivel al lenguaje de la arquitectura x86-32 respecto a lo estudiado anteriormente en el Computador Teórico.

Por ejemplo, la traducción de la siguiente asignación, suponiendo que el compilador decida almacenar las variables `a` y `b` en los registros `EAX` y `EBX`, respectivamente, se muestra a continuación:

```
a = b;
```

```
1 mov eax, ebx
```

En caso de que el compilador decida almacenar las variables `a` y `b` a partir de las posiciones de memoria `1020 3040h` y `1020 3050h`, respectivamente, la traducción sería la siguiente.

```
1 mov eax, [10203050h]
2 mov [10203040h], eax
```

Se utiliza el registro `EAX` como almacén temporal y se accede a la memoria utilizando un modo de direccionamiento que no estaba disponible en el Computador Teórico. En este caso no es necesario utilizar un registro índice, se puede indicar directamente la dirección a la que se quiere acceder entre corchetes. En el ejemplo se supone que las variables son de 32 bits, por lo que se utiliza un registro temporal de 32 bits.

1.3. Sentencias aritméticas y lógicas

Una sentencia que incluya una operación aritmética podría ser la siguiente:

```
a = b + c;
```

Suponiendo que el compilador decida almacenar las variables `a`, `b` y `c` en las direcciones de memoria `1020 3040h`, `1020 3050h` y `1020 3060h`, respectivamente, la traducción sería la siguiente:

```
1 mov eax, [10203050h]
2 add eax, [10203060h]
3 mov [10203040h], eax
```

En la arquitectura x86-32 las instrucciones aritméticas y lógicas reciben dos operandos; ambos actúan de fuente, y el primero, además, también actúa de destino. Así pues, la instrucción de la línea 1 transfiere a `EAX` la variable `b`. La instrucción de la línea 2 suma `EAX`, que contiene la variable `b`, con la variable `c` y almacena el resultado en `EAX`. Por último, la instrucción de la línea 3 transfiere el resultado a la posición de memoria donde está almacenada la variable `a`. En este ejemplo también se supone que las variables son de 32 bits. En caso de que las variables fuesen de tipo byte o palabra habría que utilizar registros de ese tamaño, como por ejemplo `AL` y `AX`, respectivamente.

Letra	Significado	Observaciones
`J`	*Jump* (saltar)	–
`E`	*Equal* (igual)	–
`N`	*Not* (no)	–
`G`	*Greater than* (mayor que)	Magnitudes con signo
`L`	*Less than* (menor que)	Magnitudes con signo
`A`	*Above* (mayor que)	Magnitudes sin signo
`B`	*Below* (menor que)	Magnitudes sin signo

Tabla 2: Símbolos empleados en las instrucciones de salto en la arquitectura x86-32

Magnitudes sin signo	Relación	Magnitudes con signo
`JE`	$=$	`JE`
`JNE`	$\neq$	`JNE`
`JA` o `JNBE`	$>$	`JG` o `JNLE`
`JAE` o `JNB`	$\geq$	`JGE` o `JNL`
`JB` o `JNAE`	$<$	`JL` o `JNGE`
`JBE` o `JNA`	$\leq$	`JLE` o `JNG`

Tabla 3: Instrucciones de salto en la arquitectura x86-32

1.4. Sentencias condicionales

La forma de traducir una sentencia condicional en la arquitectura x86-32 es similar a la vista en el Computador Teórico: se implementa con una comparación y un salto condicional. La diferencia es que en la arquitectura x86-32 no es necesario fijarse en los bits de estado ya que, además de existir las instrucciones de salto en función de los bits de estado, existen otras instrucciones que saltan cuando se cumple cada una de las relaciones. Utilizando los símbolos mostrados en la tabla 2 se componen las instrucciones mostradas en la tabla 3.

Utilizando estas instrucciones, la traducción resulta más sencilla. Por ejemplo, una sentencia condicional podría ser la siguiente:

```
if (a <= b)
  c = d;
```

Esta sentencia se podría traducir de la siguiente forma:

```
1 cmp eax, ebx
2 ; Saltar si eax <= ebx
3 jle consecuente
4 jmp siguiente
5 consecuente:
6 mov ecx, edx
7 siguiente:
```

En la traducción se supone que las variables `a`, `b`, `c` y `d` se almacenan en los registros `EAX`, `EBX`, `ECX` y `EDX`, respectivamente. Primero se comparan las variables `a` y

b con la instrucción CMP y, a continuación, se salta cuando la condición es cierta. En este caso se supone que las variables son de tipo entero, es decir, magnitudes con signo.

Otra sentencia condicional en C++ podría ser la siguiente:

```
if (a > b)
  c = d;
```

Siguiendo con la misma asignación de variables a registros, pero suponiendo en este caso que las variables son de tipo natural, la traducción sería la siguiente:

```
cmp eax, ebx
; Saltar si eax > ebx
ja consecuente
jmp siguiente
consecuente:
mov ecx, edx
siguiente:
```

1.5. Bucles

La forma en la que se traducen los bucles en la arquitectura x86-32 es muy similar a la estudiada en el Computador Teórico. Las diferencias se deben a que el juego de instrucciones de la arquitectura x86-32 es mucho más rico, lo que permite implementar el mismo código con menos instrucciones.

A continuación, se muestra la traducción del siguiente ejemplo de acceso a los elementos de un vector:

```
for (i = 0; i < 100; i++)
  V[i] = V[i] - 25;
```

```
xor ecx, ecx

inicio_for:
cmp ecx, 100
jae fin_for ; Si i >= 100 entonces se termina el bucle for

; Cuerpo del for
mov eax, [10203040h + ecx*4] ; eax = V[i]
sub eax, 25 ; eax = V[i] - 25
mov [10203040h + ecx*4], eax ; V[i] = eax, es decir, V[i] = V[i] - 25

inc ecx ; i++
jmp inicio_for
fin_for:
```

La estructura del bucle for es idéntica a la estudiada anteriormente. Sin embargo, la forma en la que se accede a los elementos del vector resulta más sencilla al tener instrucciones que permiten modos de direccionamiento más complejos. En este caso se ha supuesto que la variable V se almacena a partir de la dirección de memoria 10203040h y que cada elemento del vector es de tipo doble palabra. La variable i se almacena en el registro ECX. Para acceder a cada elemento del

vector se accede a la posición de memoria cuya dirección es `10203040h + ecx*4`. En la primera iteración del bucle se accederá al primer elemento en la dirección `10203040h + 0*4 = 10203040h`, en la segunda iteración se accederá al segundo elemento en la dirección `10203040h + 1*4 = 10203044h`, y así consecutivamente. La variable índice se multiplica por cuatro, ya que cada elemento del vector ocupa cuatro posiciones de memoria.

1.6. Procedimientos

Un procedimiento en la arquitectura x86-32 sigue el mismo conjunto de pasos descrito en el Computador Teórico. Las diferencias nuevamente vienen determinadas por un juego de instrucciones más complejo. A continuación se muestra la traducción del procedimiento `EsPositivo`:

```
int EsPositivo(int Num)
{
  if (Num < 0)
    return 0;
  return 1;
}
```

```
1  EsPositivo:
2  push ebp
3  mov ebp, esp ; Prólogo
4
5  push edx ; Salvaguarda de registros
6
7  mov edx, [ebp + 8] ; Acceso al parámetro
8
9  ; Cuerpo del procedimiento
10 cmp edx, 0
11 jl negativo
12
13 mov eax, 1
14 jmp continua
15
16 negativo:
17 mov eax, 0
18
19 continua:
20
21 pop edx ; Restauración de registros
22
23 pop ebp
24 ret ; Epílogo
```

En la arquitectura x86-32, el registro `ESP` es el puntero de pila, equivalente al registro `R7` en el Computador Teórico. En este caso se utiliza el registro `EBP` como puntero auxiliar en la pila, resultando en un prólogo completamente similar al visto en el Computador Teórico. Una diferencia importante respecto al Computador Teórico es que cuando se apila un registro de 32 bits, como el registro `EBP`, se ocupan cuatro posiciones en la pila y, por lo tanto, el registro `ESP` se decrementará en cuatro unidades. Esto hace que para acceder al parámetro haya que sumar ocho posiciones

al lugar al que apunta EBP. Una vez que se tiene el parámetro en EDX, se implementa la sentencia condicional y se retorna el resultado en EAX.

Como se puede observar, el hecho de que en la arquitectura x86-32 existan modos de direccionamiento más complejos permite implementar el procedimiento en menos instrucciones. Por ejemplo, no es necesario modificar el registro EBP para apuntar a los parámetros en la pila, ya que se puede especificar mediante el modo de direccionamiento [EBP + Inmediato].

La traducción del procedimiento ContarPositivos se muestra a continuación:

```
int ContarPositivos(int Vector[], unsigned int NumElems)
{
  int NumPos;
  unsigned int i;

  NumPos = 0;
  for (i = 0; i < NumElems; i++)
  {
    if (EsPositivo(Vector[i]) == 1)
      NumPos++;
  }
  return NumPos;
}
```

```
1  ContarPositivos:
2  push ebp
3  mov ebp, esp ; Prólogo
4
5  sub esp, 4 ; Reserva de espacio para la variable local NumPos
6
7  push ecx
8  push edi
9  push edx ; Salvaguarda de registros
10
11 mov edi, [ebp + 8] ; Acceso al vector
12 mov edx, [ebp + 12] ; Acceso al número de elementos
13
14 ; Cuerpo del procedimiento
15 mov DWORD PTR [ebp - 4], 0 ; NumPos = 0
16
17 ; Comienzo del bucle
18 xor ecx, ecx ; i = 0
19
20 inicio_for:
21 cmp ecx, edx
22 jae fin_for ; Si i >= NumElems entonces se termina el bucle for
23
24 ; Cuerpo del for
25 push DWORD PTR [edi + ecx*4]
26 call EsPositivo
27 add esp, 4
28
29 cmp eax, 1
30 je consecuente ; Si Valor de retorno del proc. es 1 entonces saltar
31 jmp siguiente
32
33 consecuente:
34 inc DWORD PTR [ebp - 4] ; NumPos++
```

```
36 siguiente:

38 inc ecx ; i++
39 jmp inicio_for
40 fin_for:

42 mov eax, [ebp - 4] ; eax = NumPos

44 pop edx
45 pop edi
46 pop ecx ; Restauración de registros

48 mov esp, ebp ; Eliminar la variable local

50 pop ebp
51 ret ; Epílogo
```

Se repiten nuevamente los pasos del procedimiento: prólogo, salvaguarda de registros, acceso a parámetros, cuerpo, restauración de registros y epílogo. Tras la salvaguarda de registros, el estado de la pila se muestra en la figura 2.

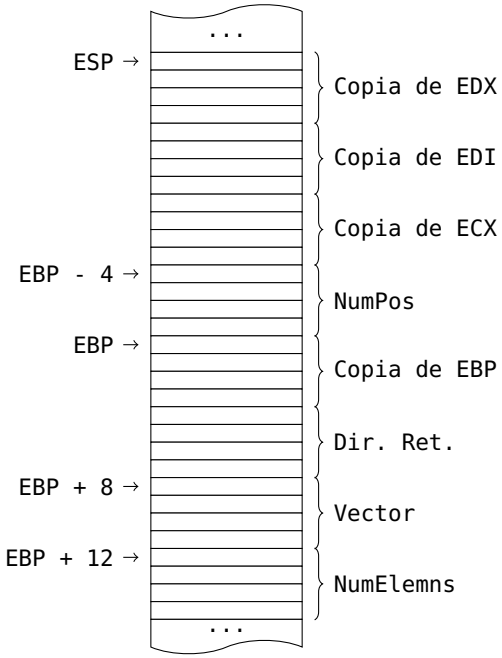

Figura 2: Estado de la pila durante la llamada a un procedimiento en la arquitectura x86-32

La reserva de espacio en la pila para la variable local se realiza con la instrucción de la línea 5, que decrementa `ESP` en cuatro unidades, haciendo hueco para la variable de tipo doble palabra `NumPos`. A continuación, se utiliza el registro `EBP` tanto para acceder a los parámetros (`EBP + 8` y `EBP + 12`) como para acceder a la variable local

(`EBP - 4`). En el registro `EDI` se almacena la dirección del vector y en `EDX` el número de elementos. Dentro del cuerpo del bucle se accede a un elemento del vector mediante la expresión `[edi + ecx*4]`. `EDI` apunta al primer elemento y `ECX` contiene el número de iteraciones que se usará como índice en el vector. El índice se multiplica por cuatro, ya que cada elemento es de tipo doble palabra. A medida que el índice se incremente en cada iteración se apilará cada uno de los elementos del vector para pasárselo como parámetro al procedimiento `EsPositivo`. Después, se eliminará el parámetro con la instrucción de la línea 27, y se incrementará `NumPos` si en `EAX` se ha retornado 1. Cuando termine el bucle, se almacenará en `EAX` la variable `NumPos`, se restaurará la pila y se retornará. Para eliminar la variable local se pude sumar a `ESP` el valor 4 o utilizar la instrucción de la línea 48, que restaura el valor de `ESP` al valor que tenía antes de la reserva de espacio para las variables locales. La eliminación de los parámetros se puede realizar de dos formas: desde el invocador del procedimiento o desde el propio procedimiento. Desde el invocador es similar a como se realiza en el Computador Teórico: hay que sumar al registro puntero de pila, en este caso el registro `ESP`, el número de posiciones de memoria que ocupan los parámetros, es decir, la instrucción sería `ADD ESP, 8`. Otra opción es eliminar los parámetros desde el propio procedimiento con la instrucción `RET 8`. Ambas opciones producen el mismo resultado.

2. Ensamblador de la arquitectura x86-32

2.1. Introducción

Como ensamblador de la arquitectura x86-32 se va a utilizar el MASM (*Microsoft Macro Assembler*), muy similar al del Computador Teórico.

Este lenguaje está formado por un conjunto de sentencias, una en cada línea. El formato de una sentencia es el siguiente:

```
[etiqueta] [operacion] [operandos] [;comentarios]
```

Los campos de una sentencia son opcionales.

Nuevamente aparecen dos tipos de sentencias: instrucciones y directivas. Las instrucciones son órdenes al procesador y las directivas dirigen el proceso de compilación pero no generan código.

El fichero creado con todas las sentencias que constituyen un programa se denomina fichero fuente. Este tipo de fichero tiene formato ASCII. Se recomienda que el fichero de código fuente tenga por extensión `.asm`.

El fichero fuente se compila utilizando el programa ensamblador, que generará un nuevo fichero, denominado fichero objeto, de igual nombre y de extensión `.obj`. A continuación este fichero debe ser enlazado con las librerías, generando el programa ejecutable con extensión `.exe`. El programa ejecutable contiene cabeceras con información para el sistema operativo, la codificación de las instrucciones del programa, las funciones de librería a las que llama el programa y, opcionalmente, información de depuración.

Computador Teórico	x86-32	Observaciones
`DIRECCION etiqueta`	`OFFSET etiqueta`	Calcula la dirección de `etiqueta`
`BYTEALTO`		
`BYTEBAJO`		
	`BYTE PTR`	Especifica tamaño byte
	`WORD PTR`	Especifica tamaño palabra
	`DWORD PTR`	Especifica tamaño doble palabra

Tabla 4: Equivalencias entre los operadores del lenguaje ensamblador del Computador Teórico y el de la arquitectura x86-32

2.2. Elementos del lenguaje

2.2.1. Comentarios

Al igual que en el ensamblador del Computador Teórico, todo lo que siga a un punto y coma (`;`) hasta el final de la línea se considera un comentario.

2.2.2. Palabras reservadas

El lenguaje ensamblador de la arquitectura x86-32 tiene cientos de palabras reservadas, incluyendo instrucciones y directivas. No se van a comentar de forma detallada, pero en las siguientes secciones se describirán las más importantes. Ni instrucciones ni directivas se podrán usar como identificadores de etiquetas

2.2.3. Identificadores

Un identificador es un nombre simbólico que hace referencia a una constante, a una zona de la sección de código o a una instrucción. La sintaxis es similar a la del Computador Teórico.

2.2.4. Constantes

La sintaxis para la definición y el uso de constantes es similar al ensamblador del Computador Teórico.

2.2.5. Operadores

En la tabla 4 se muestran las equivalencias entre algunos operadores del Computador Teórico y los de este lenguaje.

Tanto la directiva `DIRECCION`, en el ensamblador del Computador Teórico, como la directiva `OFFSET` en el de la arquitectura x86-32, sirven para calcular la dirección a la cual hace referencia una etiqueta. En este ensamblador, las directivas `BYTEALTO` y `BYTEBAJO` no son necesarias y, por tanto, no tienen equivalente. Por otro lado, aparecen tres nuevas directivas para indicar el tamaño de un operando.

Instrucción	Descripción
`MOV {R|M}, {R|M|I}`	Copia el contenido del operando fuente (el de la derecha) en el operando destino (el de la izquierda)
`MOVSX {R16|M16}, {R8|M8}`	Convierte el operando fuente interpretado con signo (entero) de 8 a 16 bits y lo transfiere al operando destino.
`MOVSX {R32|M32}, {R8|M8}`	Convierte el operando fuente interpretado con signo (entero) de 8 a 32 bits y lo transfiere al operando destino.
`MOVSX {R32|M32}, {R16|M16}`	Convierte el operando fuente interpretado con signo (entero) de 16 a 32 bits y lo transfiere al operando destino.
`MOVZX {R16|M16}, {R8|M8}`	Convierte el operando fuente interpretado sin signo (natural) de 8 a 16 bits y lo transfiere al operando destino.
`MOVZX {R32|M32}, {R8|M8}`	Convierte el operando fuente interpretado sin signo (natural) de 8 a 32 bits y lo transfiere al operando destino.
`MOVZX {R32|M32}, {R16|M16}`	Convierte el operando fuente interpretado sin signo (natural) de 16 a 32 bits y lo transfiere al operando destino.

Tabla 5: Instrucciones de transferencia y conversión de tipos en la arquitectura x86-32

Existen instrucciones con modos de direccionamiento a memoria que suponen una incertidumbre respecto al tamaño de los operandos, como por ejemplo la siguiente:

```
1 inc [ebx]
```

A partir de la instrucción anterior, el compilador no sabría si el dato al que apunta `EBX` es de tipo byte, palabra o doble palabra. Para eliminar esa indeterminación hay que utilizar un operador que indique el tamaño del dato. Por ejemplo, si el dato fuese de tipo byte la instrucción anterior se debería escribir de la siguiente forma:

```
1 inc byte ptr [ebx]
```

2.2.6. Instrucciones

El lenguaje ensamblador de la arquitectura x86-32 contiene cientos de instrucciones. Entre todas ellas, las que se utilizan con mayor frecuencia se muestran en las tablas 5, 6, 7, 8 y 9. El símbolo `R` se refiere a registro, `M` a memoria e `I` a inmediato. El número que aparece a continuación de estos símbolos indica el tamaño en bits.

En general las instrucciones sólo tienen dos operandos: el primero actúa de fuente y de destino y el segundo de fuente.

Instrucción	Descripción
`ADD {R|M}, {R|M|I}`	Suma el contenido del operando fuente y el destino. Almacena el resultado en el operando destino
`SUB {R|M}, {R|M|I}`	Resta del contenido del operando destino el operando fuente. Almacena el resultado en el operando destino
`AND {R|M}, {R|M|I}`	Realiza la operación lógica `AND` entre el operando fuente y el operando destino. Almacena el resultado en el operando destino
`OR {R|M}, {R|M|I}`	Realiza la operación lógica `OR` entre el operando fuente y el operando destino. Almacena el resultado en el operando destino
`XOR {R|M}, {R|M|I}`	Realiza la operación lógica `XOR` entre el operando fuente y el operando destino. Almacena el resultado en el operando destino
`INC {R|M}`	Incrementa el operando en una unidad
`DEC {R|M}`	Decrementa el operando en una unidad
`NOT {R|M}`	Realiza la operación lógica `NOT` con el operando. Almacena el resultado en el mismo operando
`CMP {R|M}, {R|M|I}`	Compara dos valores restándolos. No almacena el resultado pero modifica los bits de estado

Tabla 6: Instrucciones aritméticas y lógicas en la arquitectura x86-32

2.2.7. Directivas

En la tabla 10 se muestran las equivalencias entre algunas directivas del Computador Teórico y las de este lenguaje.

Sin embargo, entre las directivas de los lenguajes del Computador Teórico y de la arquitectura x86-32 existen varias diferencias. Por ejemplo, en el lenguaje ensamblador de la arquitectura x86-32 no existe directiva `.PILA`, ya que la reserva se realiza de forma predeterminada. De igual forma, no hay directivas de control de carga, realizando esa tarea el sistema operativo. Además, en el lenguaje ensamblador de la arquitectura x86-32 hay que indicar que se genera código para el procesador 80386 y compatibles, con la directiva `.386`, y que se utiliza un modelo de memoria plano con convención de llamadas a funciones estándar con la directiva `.MODEL flat, stdcall`. En el Computador Teórico sólo hay un tipo de dato, pero en x86-32 hay tres, de ahí que existan tres directivas para definir datos de esos tres tamaños: DB (*Data Byte*), DW (*Data Word*) y DD (*Data Double*, es decir, doble palabra). Por último, existe la directiva `PROTO` para indicar la existencia de procedimientos externos.

Instrucción	Descripción
`JMP Etiqueta`	Realiza un salto incondicional a la instrucción que hace referencia la etiqueta (salto relativo)
`JO Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Overflow` tiene valor 1
`JNO Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Overflow` tiene valor 0
`JC Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Carry` tiene valor 1
`JNC Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Carry` tiene valor 0
`JZ Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Zero` tiene valor 1
`JNZ Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Zero` tiene valor 0
`JS Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Signo` tiene valor 1
`JNS Etiqueta`	Realiza un salto a la instrucción referenciada por `Etiqueta` si el bit de `Signo` tiene valor 0
`LOOP etiqueta`	Realiza dos acciones: decrementa el registro `ECX` (`DEC ECX`) y salta a la etiqueta si el bit de `Zero` tiene valor 0

Tabla 7: Instrucciones de control de flujo en la arquitectura x86-32

Instrucción	Descripción
`PUSH {R|M|I}`	Apila datos en la pila. Realiza dos acciones: resta al registro `ESP` el tamaño del dato que se va a apilar y lo almacena en la posición de memoria apuntada por `ESP`
`POP {R|M}`	Desapila datos. Realiza dos acciones: copia el dato almacenado en la posición de memoria apuntada por el registro `ESP` al operando destino y suma al registro `ESP` el tamaño del dato desapilado

Tabla 8: Instrucciones para el manejo de la pila en la arquitectura x86-32

Instrucción	Descripción
`CALL {Etiqueta|R32|M32}`	Realiza la llamada a un procedimiento. Realiza dos acciones: apila la dirección de retorno y salta a la primera instrucción del procedimiento
`RET {I8}`	Retorna de un procedimiento desapilando la dirección de retorno. Opcionalmente elimina los parámetros de un procedimiento de la pila sumándole al registro `ESP` el valor inmediato. El valor inmediato debe coincidir con el número de bytes que ocupan los parámetros del procedimiento en la pila

Tabla 9: Instrucciones para el manejo de procedimientos en la arquitectura x86-32

Tipo de directiva	Computador Teórico	x86-32

Definición de la estructura en secciones	`.DATOS`
	`.CODIGO`
	`.PILA`
		`.DATA`
		`.CODE`

Control de carga	`ORIGEN` `INICIO`	 `END etiqueta`
Definición del modelo de programa		`.386` `.MODEL flat, stdcall`
Finalización de código	`FIN`	`END etiqueta`
Definición de procedimientos	`PROCEDIMIENTO` `FINP`	`PROC` `ENDP`

Definición de datos	
	`VALOR`
	
	`VECES`
		`DB`
		`DW`
		`DD`
		`DUP`

Procedimientos externos		`proc PROTO`

Tabla 10: Equivalencias entre las directivas del lenguaje ensamblador del Computador Teórico y el de la arquitectura x86-32

2.3. Estructura de un fichero en lenguaje ensamblador

A continuación se muestra un ejemplo de programa en lenguaje ensamblador. Como se puede observar, sigue una definición en secciones similar al del Computador Teórico.

```
1  .386                                ; Generación de código para el procesador 80386
2  .model flat, stdcall                ; Modelo de memoria plano y uso de convención
3                                      ; de llamadas estándar
4  ExitProcess proto, dwExitCode:dword ; Declaración del procedimiento externo ExitProcess

6  .data                               ; Comienza la definición de la sección de código
7    Lista     DB 1, -1, 4, 12        ; Definición de una lista de datos de tipo byte
8    ...

10 .code                               ; Comienza la definición de la sección de código

12 inicio:                             ; Inicio del programa
13   xor ebx, ebx
14   xor esi, esi
15   xor edi, edi
16   mov ecx, 4
17 bucle:
18   ...

20   loop bucle

22   push  0
23   call  ExitProcess
24 end Inicio                          ; Finalización del programa
```