x86: Documentación
Nociones básicas sobre la arquitectura x86 y el lenguaje ensamblador

Departamento de Ingeniería en Automática, Electrónica, Arquitectura y Redes de Computadores

Universidad de Cádiz

Autor: Jesús Relinque Madroñal
Supervisora: Mercedes Rodríguez García

Índice general

Capítulo 1

El entorno de ejecución

Se describirá a lo largo de este capítulo el entorno de ejecución tal y como es percibido por el programador, es decir, se describirá cómo ejecuta el procesador las instrucciones. En el presente documento se utilizará la nomenclatura recogida en la documentación de Intel.

1.1. Modos de operación

La arquitectura IA-32 permite tres modos de operación: el modo protegido, el modo real y el modo de gestión del sistema. La arquitectura Intel64 añade a estos tres el modo IA-32e y dos submodos de IA-32e: El submodo de compatibilidad y el submodo de 64 bits. En la práctica trabajaremos en el modo protegido de IA-32. Para más información sobre los modos de operación, lea el anexo de la práctica.

1.2. El ambiente básico de ejecución

La arquitectura IA-32 otorga una serie de recursos a los programas y tareas que se ejecutan. Dichos recursos se conocen como ambiente básico de ejecución. La arquitectura Intel64 soporta el ambiente básico de ejecución de IA-32 y un ambiente con ciertas diferencias en el modo IA-32e, en el que puede ejecutar programas de 64 bits (submodo 64 bits) y programas de 32 bits (submodo 32 bits). Los recursos serán descritos a continuación.

1.2.1. Espacio de direcciones

En las arquitecturas x86, se asigna una dirección física a cada byte direccionable. El conjunto de todas estas direcciones forma el espacio de direcciones físicas. En IA-32, el espacio de direcciones físicas comprende las direcciones entre 0 y $2^{36} - 1$. A la hora de acceder a memoria, los programas no utilizan direcciones físicas, sino que acceden utilizando uno de los tres modelos de memoria existentes: plano, segmentado o modo de direccionamiento real. Estos transforman la dirección dada en una dirección física.

- **Modelo de memoria plano:** Presenta la memoria a los programas como un espacio de direcciones continuo, llamado espacio de direcciones lineales, en el que cada byte tiene una dirección lineal. Las direcciones van desde 0 hasta $2^{32} - 1$ de forma contigua.
- **Modelo de memoria segmentado:** La memoria se presenta al programa como un grupo de espacios de direcciones independientes llamados segmentos. Para acceder a un byte concreto dentro de un segmento el programa proporciona una dirección lógica compuesta por un número de segmento

y un desplazamiento. El número de segmento permite seleccionar el segmento y el desplazamiento permite seleccionar un byte de dicho segmento. Un programa puede direccionar hasta 16383 segmentos de diferentes tamaños y tipos, siendo cada segmento como máximo de 2^{32} bytes.

- **Modelo de memoria del modo real:** Este es el modelo de memoria heredado del Intel 8086. Es soportado para ofrecer compatibilidad con los programas escritos para el Intel 8086. Consiste en una implementación particular de la memoria segmentada en el que el espacio de direcciones lineales consiste en un vector de segmentos de hasta 64 KB cada uno, siendo el tamaño máximo del espacio de direcciones lineales de 2^{20} bytes.

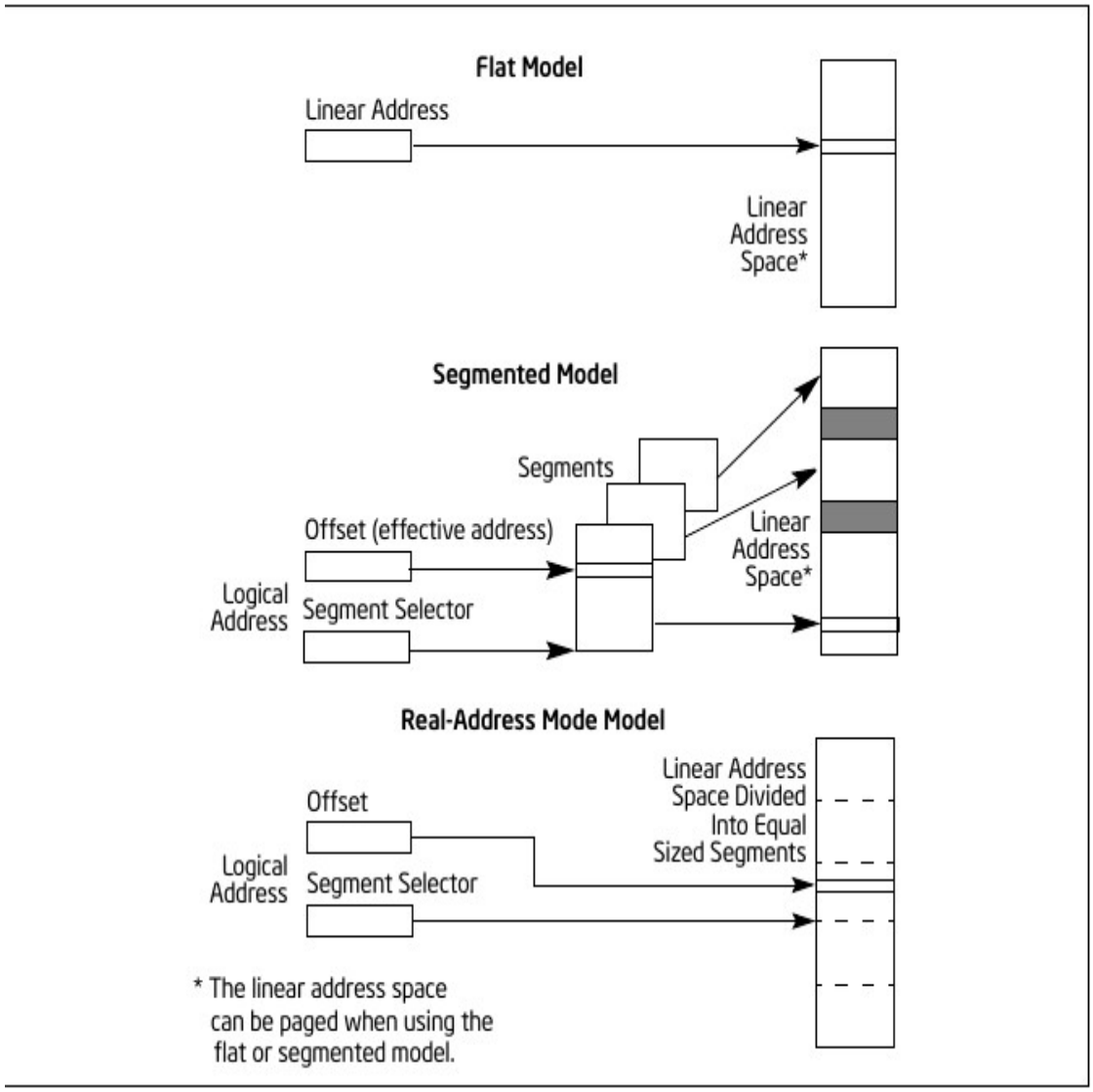

Figura 1.1: Modelos de memoria de IA-32. Imagen obtenida del manual de Intel.

La diferencia entre IA-32 y Intel64 en cuanto a los modelos de memoria está en que el espacio de direcciones físicas es mayor en esta última, por lo tanto, en el modo 64 bits el espacio de direcciones lineales puede ser de 64 bits, es decir, las direcciones estarán comprendidas entre 0 y $2^{64} - 1$.
En la práctica utilizaremos el modelo de memoria plano, con un solo segmento.

1.2.2. Registros de propósito general

IA-32 ofrece 8 registros de propósito general de 32 bits. Se utilizan para almacenar operandos de operaciones aritmeticológicas o de cálculos de direcciones, resultados de dichas operaciones, así como punteros a memoria. En principio, cualquiera de estos registros puede ser utilizado para guardar cualquiera de los datos antes mencionados, sin embargo ciertas instrucciones utilizan algunos registros de forma predeterminada. También destaca el caso del registro ESP, que contiene el puntero de pila y que por lo tanto no debería ser usado para otro propósito. A continuación se detalla un resumen de algunos de los usos predeterminados de registros de propósito general, aunque existen muchos más.

- **Registro EAX:** Acumulador de operandos y resultados.
- **Registro EBX:** Puntero a datos en el registro de segmento DS.
- **Registro ECX:** Contador para operaciones de cadena y bucles.
- **Registro EDX:** Puntero de entrada y salida.
- **Registro ESI:** Puntero a datos del segmento apuntado por el registro de segmento DS y puntero origen para operaciones de cadena.
- **Registro EDI:** Puntero a datos del segmento apuntado por el registro de segmento ES y puntero destino para operaciones de cadena.
- **Registro ESP:** Puntero de pila.
- **Registro EBP:** Puntero a datos de la pila.

Estos son los nombres que identifican a los 8 registros generales, de 32 bits. No obstante, existen nombres alternativos para hacer referencia solo a una parte de dichos registros.

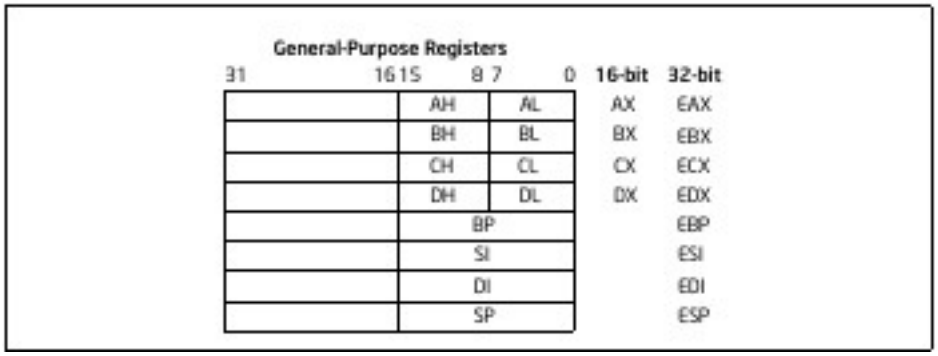

Figura 1.2: Nombres alternativos de los registros de propósito general. Imagen obtenida del manual de Intel.

En la figura 1.2 podemos observar los nombres alternativos para los registros de propósito general. La letra “E” que precede al nombre de los registros, hace referencia a la palabra “extended”, es decir,

dichos registros son en realidad una extensión de 32 bits de los originales, de 16 bits, así que podemos referirnos a los 16 bits menos significativos del registro utilizando el nombre del registro sin la letra “E”. También en el caso de los registros EAX, EBX, ECX y EDX, podemos acceder a la parte alta (8 bits más significativos) o a la parte baja (8 bits menos significativos) de los 16 bits menos significativos quitando la letra “E” del nombre del registro y sustituyendo la “X” por “H” (para acceder a la parte alta) o por “L” (para acceder a la parte baja).

1.2.3. Registros de segmento

Dado que utilizaremos el modelo de memoria plano no necesitaremos estos registros. Si aún así su curiosidad le pide saber más sobre ellos, acuda al anexo.

1.2.4. Registro de banderas EFLAGS

El registro EFLAGS de 32 bits contiene un grupo de banderas de estado y control, así como otras banderas del sistema. La figura 1.5 muestra el significado de las banderas del registro. Podemos observar que los bits del 22 al 31, así como el 3, el 5 y el 15 están reservados y deben contener siempre el valor 0, mientras que el bit 1 también está reservado pero debe tener el valor 1. Existen tres tipos de banderas:

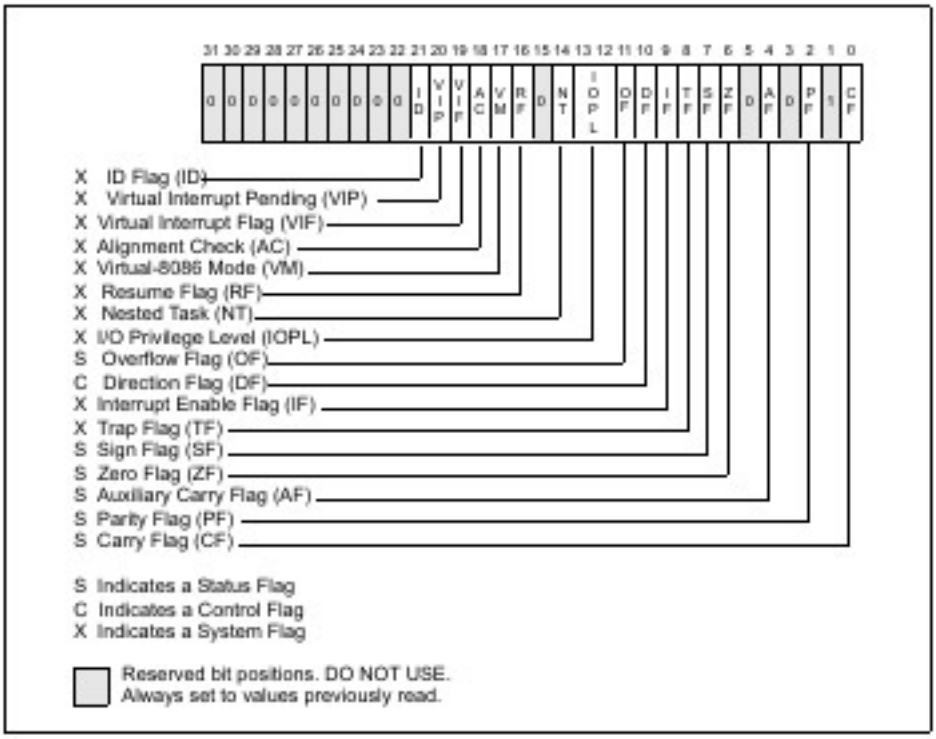

Figura 1.3: Registro de banderas. Imagen obtenida del manual de Intel.

- **Banderas de estado:** Se generan tras operaciones aritmeticológicas.
  - **CF (bit 0)** Bandera de acarreo (Carry flag) Indica si una operación genera un acarreo o un desbordamiento en una operación con enteros sin signo si su valor es 1. En caso de ser 0 indica lo contrario.

  - **PF (bit 2)** Bandera de paridad (Parity flag) Su valor es 1 si el byte menos significativo del resultado tiene un número par de unos y 0 en el caso contrario.
  - **AF (bit 4)** Bandera de ajuste (Adjust flag) Su valor es 1 si se genera acarreo en una operación aritmética entre decimales codificados en binario (BCD)
  - **ZF (bit 6)** Bandera de cero (Zero flag) Su valor es 1 si el resultado es 0 y 0 en caso contrario.
  - **SF (bit 7)** Bandera de signo (Sign flag) Su valor es el mismo que el bit más significativo del resultado, indicando 0 signo positivo y 1 signo negativo.
  - **OF (bit 11)** Bandera de desbordamiento (Overflow flag) Su valor es 1 si se produce desbordamiento (el resultado es imposible de almacenar en el registro destino) y 0 en caso contrario.

- **Bandera DF:** La bandera de dirección (Direction flag, en el bit 10) es utilizada por las instrucciones de cadena. Si su valor es 1, las instrucciones de cadena las recorrerán de los bytes más significativos a los menos significativos (desde el fin hasta el inicio); si el valor es 0, la cadena se recorrerá desde los bytes menos significativos hasta los más significativos (desde el inicio hasta el final). La instrucción STD se utiliza para poner la bandera a 1, y la instrucción CLD para ponerla a 0.

- **Banderas del sistema y campo IOPL:** Estas banderas son utilizadas por el sistema operativo, por lo que no las utilizaremos en esta práctica. En el anexo aparecen detalladas.

1.2.5. Puntero de instrucción

El puntero de instrucción (EIP) contiene el desplazamiento, con respecto al segmento de código, para obtener la siguiente instrucción. El valor de este registro puede ser modificado (tanto avanzando, como retrocediendo) por ciertas instrucciones como JMP, CALL y RET.

1.2.6. Registros de la unidad de coma flotante x87 (FPU):

La unidad de coma flotante incluye ocho registros de datos, de 80 bits cada uno, un registro de control, uno de estado y uno de etiqueta, cada uno de ellos de 16 bits; un registro para el código de operación, de 11 bits, un registro de puntero de instrucción de 48 bits y por último un registro de puntero de dato también de 48 bits. El funcionamiento de la arquitectura x87 se explicará más adelante.

1.2.7. Registros MMX:

Son ocho registros que permiten realizar instrucciones SIMD con paquetes de datos de 64 bits (este es el tamaño de cada uno de los registros). Su funcionamiento será explicado más adelante.

1.2.8. Registros XMM:

Son ocho registros que permiten realizar instrucciones SIMD con paquetes de datos de 128 bits (este es el tamaño de cada uno de los registros). Se incluye además otro registro llamado MXCSR de 32 bits. En el ambiente de Intel64 se incrementa el número de registros de datos a 16. Su funcionamiento será explicado más adelante.

1.2.9. Pila (Stack):

Para soportar llamadas a procedimientos o subrutinas así como el paso de parámetros de parámetros a los mismos, el ambiente de ejecución dispone de una pila y recursos para su manejo. La pila se sitúa en memoria.

1.3. Recursos adicionales

Además de los recursos del ambiente básico de ejecución, la arquitectura dispone de una serie de recursos utilizados por el sistema operativo y por el software de desarrollo de sistemas. No los utilizaremos en la práctica, pero están detallados brevemente en el anexo.

Capítulo 2

Instrucciones de propósito general

En este capítulo algunas de las instrucciones de propósito general. Estas han sido incluidas en todos los procesadores de las arquitecturas x86, tanto IA-32, como Intel64 (y por ende AMD64).

2.1. Instrucciones de transferencia de datos

Se utilizan para mover datos entre memoria y los registros, aunque también incluye operaciones para el acceso a la pila, la conversión de datos y los movimientos condicionales.

- MOV opdestino, oporigen Mueve datos entre registros, entre memoria y registros e inmediatos a registros
- CMOVE/CMOVZ opdestino, oporigen Movimiento condicional, si el origen es cero. Los operandos son sin signo.
- CMOVNE/CMOVNZ opdestino, oporigen Movimiento condicional, si el origen no es cero. Los operandos son sin signo.
- CMOVA/CMOVNBE opdestino, oporigen Movimiento condicional, si el origen es mayor que el destino. Los operandos son sin signo.
- CMOVAE/CMOVNB opdestino, oporigen Movimiento condicional, si el origen es mayor o igual que el destino. Los operandos son sin signo.
- CMOVB/CMOVNAE opdestino, oporigen Movimiento condicional, si el origen es menor que el destino. Los operandos son con signo.
- CMOVBE/CMOVNA opdestino, oporigen Movimiento condicional, si el origen es menor o igual que el destino. Los operandos son con signo.
- CMOVG/CMOVNLE opdestino, oporigen Movimiento condicional, si el origen es mayor que el destino. Los operandos son con signo.
- CMOVGE/CMOVNL opdestino, oporigen Movimiento condicional, si el origen es mayor o igual que el destino. Los operandos son con signo.
- CMOVL/CMOVNGE opdestino, oporigen Movimiento condicional, si el origen es menor que el destino. Los operandos son con signo.

- CMOVLE/CMOVNG opdestino, oporigen Movimiento condicional, si el origen es menor o igual que el destino. Los operandos son con signo.
- CMOVC opdestino, oporigen Movimiento condicional, si hay acarreo. Los operandos son con signo.
- CMOVNC opdestino, oporigen Movimiento condicional, si no hay acarreo. Los operandos son con signo.
- CMOVO opdestino, oporigen Movimiento condicional, si hay desbordamiento. Los operandos son con signo.
- CMOVNO opdestino, oporigen Movimiento condicional, si no hay desbordamiento. Los operandos son con signo.
- CMOVS opdestino, oporigen Movimiento condicional, si el signo es negativo. Los operandos son con signo.
- CMOVNS opdestino, oporigen Movimiento condicional, si el signo es positivo. Los operandos son con signo.
- CMOVP/CMOVPE opdestino, oporigen Movimiento condicional, si el bit de paridad es 1 (número par de unos).
- CMOVNP/CMOVPO opdestino, oporigen Movimiento condicional, si el bit de paridad es 0 (número impar de unos).
- XCHG operando1, operando2 Intercambia los dos operandos.
- BSWAP operando1 Invierte el orden de los bytes de un registro de 32 o 64 bits.
- XADD opdestino, oporigen Intercambia los dos operandos y almacena su suma en opdestino. Oporigen solo puede ser un regsitro.
- CMPXCHG opdestino, oporigen Comparar e intercambiar. Necesita dos operandos fuente (el segundo es EAX) y uno destino. Compara los valores de EAX y el destino y si son iguales intercambia el valor del destino por el origen y si no, almacena en EAX el valor del destino.
- PUSH oporigen Decrementa el valor del puntero de pila (en ESP) e introduce el operando fuente en la cima de la pila.
- POP opdestino Incrementa el valor del puntero de pila (en ESP) e introduce el valor de la cima en el destino, que puede ser cualquier registro o memoria.
- PUSHA/PUSHAD Igual que push, pero introduce en la pila los ocho registros de propósito general.
- POPA/POPAD Igual que pop, pero saca los ocho elementos de la cima y los coloca en los ocho registros de propósito general.
- CWD/CDQ CWD convierte palabra (16 bits) en doble palabra (32 bits) y CDQ convierte doble palabra en cuádruple palabra (64 bits). Utiliza como registro fuente EAX y su versión de 64 bits RAX.
- CBW/CWDE CBW convierte byte a palabra (16 bits) y CWDE hace lo mismo que CWD y utiliza como origen EAX.
- MOVSX opdestino, oporigen Mueve y hace extensión de signo.
- MOVZX opdestino, oporigen Mueve y rellena con ceros.

2.2. Instrucciones aritméticas en binario

Realizan cálculos aritméticos en binario sobre operandos almacenados en registros o en memoria.

- ADD opdestino, oporigen Suma oporigen y opdestino y almacena el resultado en opdestino.
- ADC opdestino, oporigen Suma oporigen, opdestino y el acarreo y lo almacena en opdestino.
- SUB opdestino, oporigen Resta opdestino-oporigen y lo almacena en opdestino.
- SBB opdestino, oporigen Suma el acarreo a oporigen, resta opdestino-oporigen y lo almacena en opdestino.
- IMUL oporigen Igual que MUL oporigen, pero con signo.
- MUL oporigen Multiplica sin signo. Tiene distintos formatos segun el tamaño de los operandos.l
- IDIV oporigen Realiza la división con signo. Tiene distintos formatos segun el tamaño de los operandos.
- DIV oporigen Igual que IDIV pero los operadores son sin signo.
- INC oporigen Incrementa en una unidad oporigen.
- DEC oporigen Decrementa oporigen en una unidad.
- NEG oporigen Almacena en oporigen el complemento a 2 de oporigen.
- CMP oporigen1, oporigen2 Compara oporigen1 y oporigen2 y modifica EFLAGS en consecuencia.

Capítulo 3

Ensamblador x86

3.1. El ensamblador y el IDE

Existen varios ensambladores para las arquitecturas x86. El MASM de Windows, el NASM, el FASM... Tras sopesar las ventajas y desventajas de cada uno de ellos, se decidió utilizar para estas prácticas el ensamblador GoASM junto con el enlazador GoLINK. GoASM nos permite realizar programas en modo protegido y que utilizan el modelo de memoria plano, además dispone de una documentación amplia en su web. Para facilitar además la labor de programación se decidió integrar dichas herramientas con el IDE EasyCode que nos permitirá ensamblar y enlazar nuestro programa más fácilmente y nos proporciona además el resaltado de la sintaxis.

3.2. Estructura del programa

El ensamblador nos permite declarar en nuestro programa tres secciones:

- Sección de datos (denotada por .DATA): En esta sección declararemos los datos ubicados en memoria. Para cada dato ubicado en memoria podemos asignar una etiqueta, un tamaño y un valor inicial, o dejarlo sin inicializar.
- Sección de código (denotada por .CODE): En esta sección escribiremos las instrucciones de nuestro programa.
- Sección de constantes (denotada por .CONST): En esta sección, al igual que en data, podemos declarar datos ubicados en memoria, pero estos no podrán cambiar a lo largo del programa.

3.3. Sintaxis básica

Existen dos tipos de sintaxis en los ensambladores de x86: la sintaxis Intel y la sintaxis AT&T. La mayoría de ensambladores utilizan la sintaxis de Intel, entre ellos el GoAsm. Un ejemplo de ensamblador que usa la sintaxis AT&T es el ensamblador GAS de GNU.

La sintaxis de GoAsm no varía mucho con respecto a otros ensambladores que utilizan la sintaxis Intel. A continuación se describen algunas de las características más importantes de la sintaxis del ensamblador GoAsm. Algunas de ellas son comunes con otros ensambladores que usan la sintaxis Intel y otras difieren ligeramente.

- Para colocar comentarios utilizaremos “;”. Cuando el ensamblador detecta “;” considera el resto de la línea como comentario y la descarta.

- Los números se considerarán por defecto como decimales. Para denotar números hexadecimales utilizaremos “0x” al principio del número o “h” al final del mismo y para los números reales utilizaremos “.”

- En cuanto a las mayúsculas, el ensamblador no distingue entre mayúsculas y minúsculas, excepto en las etiquetas. Estas deberán terminar en “:” si se utilizan en la sección de código, aunque no es necesario (y de hecho no es recomendable) para las etiquetas de la sección de datos. El programa debe comenzar con la etiqueta “Start:”.

3.4. Formato de las instrucciones

Dado que la arquitectura x86 es una arquitectura CISC, contamos con un gran número de instrucciones. Además, cada una de ellas dispone de varios formatos. Para conocer más sobre los formatos de cada instrucción, debemos consultar la documentación de Intel. Aquí detallaremos algunas características generales sobre el formato de instrucción.

- Todas las instrucciones admiten un solo operando de memoria a lo sumo.

- Todos los operandos de una misma instrucción deben ser del mismo tamaño. El tamaño de dichos operandos determinará el formato de la instrucción que se utilizará.

- Podemos determinar qué formato de una instrucción se utilizará mediante los nombres alternativos de los registros (por ejemplo, si utilizamos el registro EAX se ejecutará un formato de instrucción de 32 bits, si utilizamos AX, se usará un formato de 16 bits, etc.)

- En la documentación de Intel se puede encontrar todos los formatos disponibles de todas las instrucciones junto con su funcionamiento y si afecta o no al registro de banderas.

3.5. Directivas de la sección .data

En la sección .data incluiremos aquellos datos que queramos definir en memoria. Podemos definir etiquetas dentro de esta sección que nos permitirán más tarde acceder a dichos datos. Podemos seguir varios formatos:

- **NOM_ETIQUETA DX VALOR_INICIAL :** Pudiendo ser X: B (Byte), W (Palabra, 16 bits), D (Doble palabra, 32 bits), Q (Cuádruple palabra, 64 bits) o T (Párrafo, 10 bytes). De esta forma declararemos en memoria una posición del tamaño indicado por X e inicializada con VALOR_INICIAL.

- **NOM_ETIQUETA DX VALOR1, VALOR2,... :** En este caso se creará un vector con tantas posiciones como valores pongamos, cuya posición inicial vendrá representada por NOM_ETIQUETA y accederemos al resto de valores sumando el desplazamiento adecuado (1 por cada posición si X es B, 2 si es W, 4 si es D, 8 si es Q y 10 si es T).

- **NOM_ETIQUETA DX NÚMERO DUP VALOR:** En este caso se creará un vector con tantas posiciones como indiquemos en NÚMERO, todas ellas inicializadas a VALOR.

- **NOM_CADENA DB "Contenido de la cadena",0:** De esta forma podemos declarar una cadena de caractéres como las de C. El contenido de la cadena puede ir entre comillas simples o dobles, e incluso puede escribirse carácter a carácter ("h","o","l","a",0). Es recomendable terminarla en 0 (para utilizar las instrucciones de cadena) y poner de tamaño B.

Si queremos dejar una posición sin inicializar (no es recomendable) podemos sustituir VALOR por "?".

3.6. Modos de Direccionamiento

Existen varias formas de obtener los operandos que utilizamos en las distintas instrucciones. Para ello, nos valdremos de los distintos modos de direccionamiento existentes. Utilizaremos la instrucción MOV como ejemplo.

- **Direccionamiento de registro:** Nos permite recuperar un dato almacenado en un registro. Por ejemplo, la instrucción MOV ECX, EDX almacena el valor de EDX en ECX.
- **Direccionamiento implícito:** Ciertas instrucciones utilizan de forma predefinida un registro y no permiten utilizar otros. Para conocer qué instrucciones utilizan este direccionamiento y qué registro es el que usan de forma predefinida, se puede consultar la documentación de Intel.
- **Direccionamiento inmediato:** Nos permite especificar un dato en la misma instrucción. Por ejemplo, MOV AL, 12H almacenará el inmediato 12H en AL.
- **Direccionamiento a memoria:** Permite utilizar en operandos alojados en memoria principal. Para acceder al contenido de una dirección de memoria, tanto para lectura como para escritura, lo indicaremos encerrando la dirección entre corchetes. Por ejemplo, si en el registro EAX se encuetra una dirección de memoria que apunta a un dato que queremos usar, podremos acceder al dato apuntado indicando el operando así: Mov [EAX], ECX (guarda el contenido de ECX en la dirección a la que apunta EAX).

  Si lo que queremos es almacenar algo en dicha posición, y no está claro qué tamaño debe asignársele, como por ejemplo en MOV [EAX], 12h (12 puede almacenarse como byte, como palabra, etc.); deberemos indicar el tamaño antes del corchete usando las letras B (Byte), W(Word, palabra), D (Double Word, palabra doble) o Q (Quad Word, palabra cuádruple), quedando la instrucción así: MOV D[EAX], 12h.

  Si en lugar de acceder al contenido queremos obtener la dirección que representa una etiqueta, utilizaremos la palabra "ADDR". Por ejemplo, si tenemos en memoria un dato cuya etiqueta es operando1, para almacenar en EAX la dirección de operando1 utilizaremos la instrucción: MOV EAX, ADDR operando1.

3.6.1. Desplazamiento en el direccionamiento directo

En el direccionamiento directo utilizamos una dirección de memoria (bien de forma numérica o con una etiqueta), o una expresión que al evaluarla se convierta en una dirección de memoria válida. Como formato general, utilizaremos el siguiente formato:

Aunque generalmente solo utilizaremos la base, podemos utilizar también un índice multiplicado por una escala y sumar el desplazamiento. El ensamblador nos permitirá utilizar entre [ ] los operadores «+» y «*».

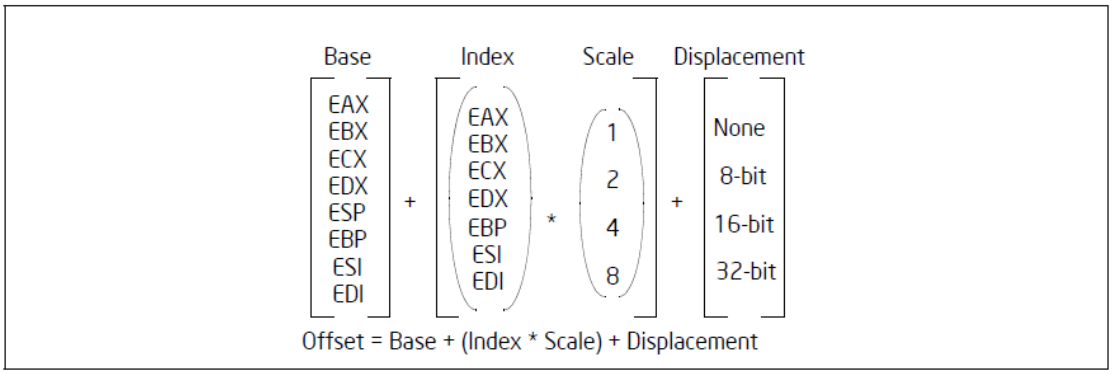

Figura 3.1: Esquema genérico para obtener el desplazamiento. Imagen obtenida del manual de Intel.

3.7. Llamadas al sistema

Al igual que las arquitecturas MIPS y ARM, x86 cuenta con una instrucción que permite realizar llamadas al sistema mediante el uso de interrupciones. La instrucción que nos permite provocar una interrupción es INT num, donde num es el número de interrupción. En los programas hechos para DOS se utilizaba INT 21h para realizar las llamadas al sistema y según el contenido del registro AH realizábamos distintas funciones como leer del teclado, escribir por pantalla, etc. Sin embargo, en los sistemas operativos Windows no se conocen las interrupciones de software necesarias para realizar las llamadas al sistema de este modo. Por lo tanto se sustituyen las interrupciones por llamadas a la API de Windows o a la API de C, con el objetivo adicional de conseguir un código más portable,. Para utilizar las funciones de ambas APIs, utilizaremos la pseudoinstrucción «Invoke» de la siguiente manera: «Invoke nom_funcion, param1, param2,...». Esta pseudoinstrucción empuja en la pila los parámetros y llama a la función.

3.7.1. Uso de la API de Win32

Podemos realizar llamadas al sistema a través de la API de Win32 (las funciones que de ella necesitamos las encontramos en la biblioteca Kernel32.dl, que deberemos enlazar a nuestro proyecto). De la API de Win32 utilizaremos las funciones ReadConsole y WriteConsole, que servirán para leer y escribir cadenas de caractéres únicamente. Dichas funciones reciben el mismo número de parámetros, en el siguiente orden:

- Manipulador de entrada estándar (ReadConsole) o salida estándar (WriteConsole). Para obtener los manipuladores de entrada estándar usaremos la función GetStdHandle, con el parámetro -11 para obtener el manipulador de la salida estándar y con el parámetro -10 para obtener el manipulador de la entrada estándar. El manipulador lo recibiremos en el registro EAX.
- La dirección donde se almacenará o del que se tomará la cadena (debe ser una dirección de memoria, **NO** un registro).
- El número de caractéres que serán leídos o escritos como máximo.
- Una dirección de memoria en la que almacenar el número de caractéres que se han leído o escrito.
- Un último parámetro al que siempre daremos el valor **NULL** o 0.

Para más información sobre las funciones del API de Win32 consultar Microsoft Developer Network.

3.7.2. Uso de la API de C

Podemos realizar también llamadas al sistema utilizando el API de C (cuyas funciones encontramos en la biblioteca msvcrt.dll, que deberá ser enlazada al proyecto). De la API de C podemos usar muchas de sus conocidas funciones, como puts, printf, scanf, gets,etc. Existen muchas web donde consultar la funcionalidad de las mismas.

3.8. Uso básico del depurador

El depurador es una herramienta de programación que nos permite encontrar los errores de ejecución (violación de segmento, cierres inesperados...) así como los errores lógicos (el programa funciona pero no se comporta según lo que queríamos) de un ejecutable. Esto es importante, ya que no nos ayudará a resolver los errores de ensamblado (para eso utilizaremos los mensajes de error del EasyCode). Una vez generado el ejecutable, podremos utilizar el depurador. Este nos permite ejecutar el código línea a línea y observar el cambio de los registros y de la memoria. Para ello debemos abrir el depurador (GoBug.exe) y abrir nuestro ejecutable.

Una vez abierto se nos presentan muchas opciones. La más interesante es “single step (jump over)”, que vemos en el menú action o pulsando F6. Esta opción nos permite ejecutar el programa línea por línea sin entrar en las llamadas a subprogramas. Si queremos que el depurador entre en las llamadas a subprogramas podemos utilizar la opción “single step(trace into)” o pulsar F5.

Para inspeccionar la memoria (no se muestra por defecto), seleccionamos en el menú «inspect» → «data by address» y seleccionamos OK en la pantalla que sale. Se mostrará entonces las posiciones de memoria. Además el depurador también nos permite observar el estado de la pila. La figura 3.2 muestra aquellas opciones del depurador que más útiles nos serán.

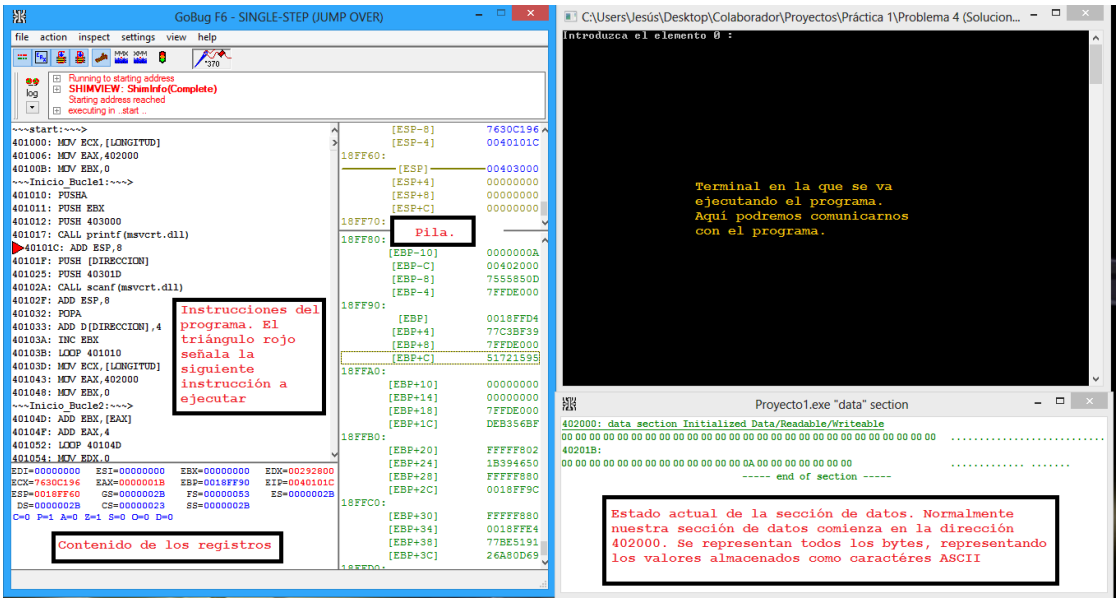

Figura 3.2: Ejemplo del depurador en funcionamiento.

Capítulo 4

Estructuras de control en ensamblador de x86

Si volvemos la vista atrás a los lenguajes de alto nivel, podemos encontrar en ellos estructuras de control (tales como la instrucción if, la instrucción while o la instrucción for) que nos permiten controlar el flujo de ejecución de nuestro programa. No obstante, dichas estructuras no existen en la mayoría de ensambladores de x86 (si bien es cierto que algunos ensambladores de x86 sí las admiten). La forma de controlar el flujo del programa será utilizando los saltos tratando de simular las instrucciones de las que disponíamos en los lenguajes de alto nivel. Veremos por lo tanto las estructuras más frecuentes y la forma de trasladarlas al ensamblador.

4.1. Comparaciones

Si bien no son estructuras de control por sí mismas, las comparaciones son utilizadas por las estructuras de control para orientar el flujo del programa en función de la evaluación de las mismas. En el ensamblador de x86 utilizaremos la instrucción CMP junto con el registro EFLAGS para llevar a cabo las comparaciones. La instrucción CMP operando1, operando2; realiza la operación operando1 - operando2 y coloca las banderas en función del resultado. El resultado de la operación no se guarda, puesto que el objetivo principal de la instrucción es el de colocar las banderas en función del resultado. Si necesitáramos dicho resultado, podríamos usar SUB en lugar de CMP.
Una vez colocadas las banderas, se pueden presentar dos casos:

- **Los operandos son sin signo:**
  - Si operando1 = operando2, entonces ZF $\leftarrow$ 1 (Zero Flag, bandera de cero) y CF $\leftarrow$ 0 (Carry Flag, bandera de acarreo)
  - Si operando1 $>$ opernado2, entonces ZF $\leftarrow$ 0 y CF $\leftarrow$ 0
  - Si operando1 $<$ operando2, entonces ZF $\leftarrow$ 0 y CF $\leftarrow$ 1
- **Los operandos son con signo:**
  - Si operando1 = operando2, entonces ZF $\leftarrow$ 1
  - Si operando1 $>$ operando2, entonces ZF $\leftarrow$ 0 y OF $\leftarrow$ SF (Overflow Flag y Sign Flag, bandera de desbordamiento y bandera de signo respectivamente)
  - Si operando1 $<$ operando2, entonces ZF $\leftarrow$ 0 y OF $\neq$ SF

4.2. Instrucciones de salto

Las instrucciones de salto no son tampoco estructuras de control por sí mismas, pero junto con las comparaciones nos permitirán simular las estructuras de control de los lenguajes de alto nivel. Existen dos tipos de instrucciones de salto: las condicionales y las incondicionales.

Las incondicionales, representadas por la instrucción JMP, realizarán siempre el salto. Para el ensamblador que venimos usando en estas prácticas, el uso de la instrucción será: JMP $>$ etiqueta, si el salto es hacia delante; o JMP $<$ etiqueta, si el salto es hacia atrás (dado que por defecto se considera que el salto es hacia atrás, no es necesario poner el signo $<$ si es hacia atrás, pero es recomendable)
Las instrucciones de salto condicionales realizarán el salto si y solo si se cumple la condición asociada a la instrucción. Algunas de estas instrucciones son:

- JZ salta si ZF = 1
- JNZ salta si ZF = 0
- JO salta si OF = 1
- JNO salta si OF = 0
- JS salta si SF = 1
- JNS salta si SF = 0
- JC salta si CF = 1
- JNC salta si CF = 0
- JP salta si PF = 1 (PF, Parity Flag, bandera de paridad, si es uno la paridad es par, si es cero, impar)
- JNP salta si PF = 0

Con estas instrucciones y las comparaciones que vimos antes, podríamos implementar una estructura de tipo if-else. Sin embargo, es necesario comprobar el estado de varias banderas para hacer la comparación, por lo tanto, sería necesario usar varias de estas instrucciones para comprobar las banderas que necesitásemos (por ejemplo, en el caso de números con signo teníamos que comprobar SF, OF y ZF). Es por ello que el ensamblador provee instrucciones más avanzadas que comprueban el estado de las banderas necesarias para implementar la condición. Estas son:

- **Si los operandos son sin signo:**
  - JE salta si operando1 = operando2
  - JNE salta si operando1 $\neq$ operando2
  - JL, JNGE salta si operando1 $<$ operando2 (Jump Less, Jump Not Greater Equal)
  - JLE, JNG salta si operando1 $\leq$ operando2
  - JG, JNLE salta si operando1 $>$ operando2
  - JGE, JNL salta si operando1 $\geq$ operando2
- **Si los operandos son con signo:**
  - JE salta si operando1 = operando2
  - JNE salta si operando1 $\neq$ operando2

- JB, JNAE salta si operando1 $<$ operando2 (Jump Below, Jump Not Above Equal)
- JBE, JNA salta si operando1 $\leq$ operando2
- JA, JNBE salta si operando1 $>$ operando2
- JAE, JNB salta si operando 1 $\geq$ operando2

Es necesario aclarar que antes de utilizar estas instrucciones debe usarse la instrucción CMP operando1, operando2 para colocar las banderas adecuadamente. Se puede observar también que existen algunas instrucciones con dos nombres. Esto se debe a que por ejemplo, en el caso de JL (Jump Less), la relación menor equivale lógicamente a la negación de la relación mayor o igual, esto es:

$$x < y \equiv no(x \geq y)$$

4.3. Estructuras if-else

Una estructura como la siguiente en C:

```
if (condicion) {
        //Hacer instrucciones del bloque if.
}
else {
        //Hacer instrucciones del bloque else.
}
```

Podría traducirse a ensamblador de la siguiente manera:

```
        ;Codigo para establecer las baderas, por ejemplo instruccion CMP
        jxx if          ;xx es la condicion para hacer if.
else:
        ;Instrucciones del bloque else
        jmp endif

if:
        ;Instrucciones del bloque if

endif:
```

Si no necesitamos el bloque else, en ensamblador quedaría de la siguiente manera:

```
        ;Codigo para establecer las baderas, por ejemplo instruccion CMP
        jxx endif               ;xx es la condicion del if
                                ;negada, puesto que saltara
                                ;fuera del if si se cumple.

        ;Instrucciones del bloque if
endif:
        ;Resto de instrucciones.
```

4.4. Bucle while

Un bucle while, como el que a continuación se presenta en C:

```
while (condicion) {
        //Cuerpo del bucle
}
```

Podríamos traducirla a ensamblador de la siguiente manera:

```
while:
        ;Codigo que establezca las banderas
        jxx endwhile    ;Condicion del bucle
        ;Cuerpo del bucle
        jmp while       ;Vuelve al principio. El anterior jxx
                        ;es el que termina el bucle.

endwhile:
```

4.5. Bucle do-while

Un bucle do-while, como el que a continuación se presenta en C:

```
do {
        //Cuerpo del bucle
} while (condicion);
```

Podríamos traducirla a ensamblador de la siguiente manera:

```
do:
        ;Cuerpo del bucle
        ;Codigo para establecer las banderas
        jxx do  ;Condcion para que se repita el bucle.

        ;Resto del codigo
```

4.6. Bucle for

Un bucle for, como el que a continuación se presenta en C:

```
for (i = 10; i<10; i++) {
        ;Cuerpo del bucle
}
```

Podríamos traducirla a ensamblador de la siguiente manera:

```
        mov ecx, 10     ;Metemos en ecx el numero
                        ;de repeticiones
for:
        ;Cuerpo del bucle
        loop for

        ;Resto del programa.
```

La instrucción loop decrementa en uno el contenido de ECX y si es cero entonces no realiza el salto y continúa el programa. Si ECX es distinto de cero, salta a la etiqueta que le indiquemos. Además de la instrucción loop, el ensamblador ofrece las instrucciones loope/loopz (son equivalentes) que realizan el salto si ecx $\neq 0$ y ZF = 1; y loopne/loopnz que salta si ecx $\neq 0$ y ZF = 0. Ninguna de estas instrucciones modifican EFLAGS.

Capítulo 5

Llamada a subprogramas

Consideramos un subprograma como un fragmento de código que puede ser accedido desde cualquier parte del programa y que, tras ser ejecutado, vuelve al sitio donde fue llamado. Si pensamos en los lenguajes de alto nivel, podríamos compararlos con los procedimientos o funciones.

5.1. Forma más básica de subprograma

La forma más básica de subprograma puede ser implementada en ensamblador mediante la instrucción JMP. Con la instrucción saltamos al comienzo del subprograma, habiendo guardado previamente en algún registro la dirección de retorno, así como aquellos parámetros que necesite el subprograma.

La desventaja de este tipo de subprogramas radica en que el programa llamador debe conocer dónde situar la dirección de retorno y los parámetros que quiere pasar a un subprograma, puesto que, si los sitúa mal, el subprograma no funcionará bien. Ésto conlleva a que el programador deba conocer cómo pasar los parámetros y la dirección de retorno a cada uno de los subprogramas que utilice. Un ejemplo de este tipo de subprograma sería:

```
1         ;Codigo del programa llamador
2         mov ecx, retorno        ;Direccion de retorno
3         mov eax, parametro1     ;Parametro 1 del subprograma
4         jmp nombresubprograma
5 retorno:
6         ;Resto del codigo del programa llamador
7
8 ;Inicio de subprograma
9 nombresubprograma:
10        ;Cuerpo del subprograma
11        ;El subprograma espera en eax el parametro1
12        jmp ecx ;Vuelve a la direccion de retorno
13                ;almacenada en ecx.
```

5.2. La pila y las instrucciones CALL y RET

Muchas CPU ofrecen soporte para crear una pila en memoria principal. Una pila es una estructura de datos secuencial. Su principal característica es que solo podemos insertar y eliminar datos por un extremo, al que llamamos cima. Al igual que si apilamos una serie de libros encima de una mesa, solo podemos poner y quitar libros en la parte superior de la pila. Es por esto que se conoce como estructura LIFO (del inglés Last In First Out, es decir, el último en entrar, el primero en salir)

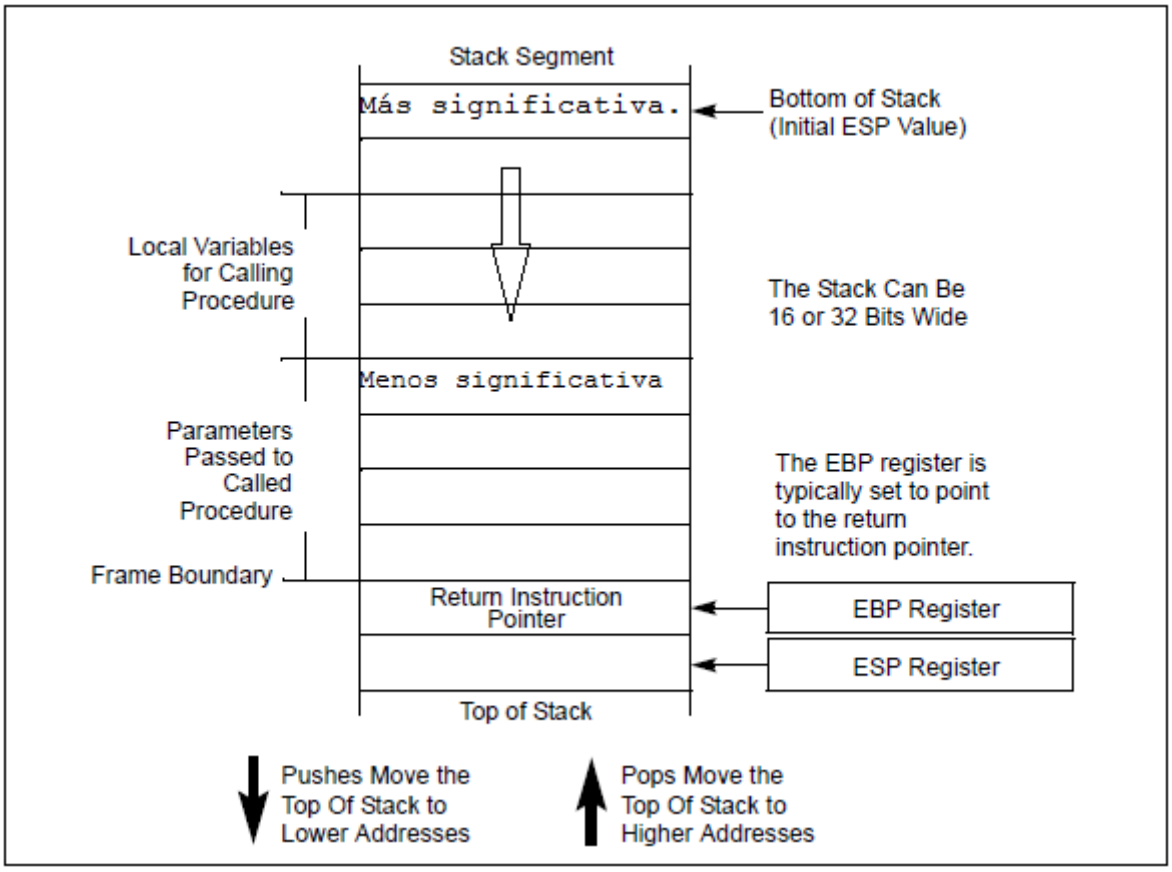

Figura 5.1: Funcionamiento de la pila. (Imagen del manual de Intel)

Sobre la pila que vamos a utilizar destacamos las siguientes características:

- Las posiciones de la pila son de 32 bits (4 bytes). Si insertamos un dato de tamaño menor, se rellenará con ceros, mientras que si el dato tiene un tamaño mayor, entonces ocupará varias posiciones.
- El registro ESP apunta a la posición que está en la cima de la pila.
- La inserción de un elemento en la pila se realiza mediante la instrucción PUSH. Si el dato que se va a insertar es de 4 bytes, la instrucción utilizará los 4 bytes anteriores a ESP para almacenar el dato y restará 4 al registro ESP, puesto que la pila comienza en los 4 bytes anteriores (la pila crece hacia las direcciones MENOS significativas).

- La extracción de una posición de la pila se realiza mediante la instrucción POP. Esta instrucción almacena el contenido de la primera posición de la pila en el lugar indicado y suma 4 al puntero ESP, puesto que la pila comienza en los 4 bytes siguientes (la pila decrece hacia las direcciones MÁS significativas). Si queremos sacar más de una posición de la pila a la vez sin almacenar lo que sacamos, podemos sumar directamente a ESP 4 por cada posición que queremos extraer de la pila.

La pila nos será muy útil como almacen de parámetros para los subprogramas así como para meter la dirección de retorno. El ensamblador nos ofrece además la instrucción CALL, cuyo funcionamiento consiste en meter en la pila la dirección de retorno y saltar a la primera dirección del subprograma. Como complemento tenemos la instrucción RET, que saca de la pila la dirección de retorno (que debe estar en la cima) y salta hacia ella. Mediante el uso de la pila y las instrucciones CALL y RET, las llamadas a subprogramas quedarían así:

```
1          ;Codigo del programa llamador
2          push parametro2
3          push parametro1
4          ;Los parametros se introducen en la pila
5          ;en orden inverso al que se pasan al subprograma
6          call nombrerutina
7
8          ;Resto del codigo del programa llamador
9
10 ;Inicio de subprograma
11 nombresubprograma:
12         ;Cuerpo del subprograma
13         ;El subprograma utiliza los parametros de la pila
14         ret     ;En la cima de la pila esta
15                 ;la direccion de retorno
```

En el ensamblador que utilizamos en las prácticas podemos cambiar las líneas 2,3 y 6 por:

```
1          invoke nombrerutina, parametro1, parametro2
```

Aunque no todos los ensambladores admiten esta pseudoinstrucción.

5.3. Convenciones de llamada

Una convención de llamada establece las condiciones que un programa llamador y un subprograma llamado deben cumplir para que la llamada se ejecute satisfactoriamente. Podemos definir nuestra propia convención de llamada, sin embargo, en los lenguajes de alto nivel existen convenciones de llamada normalizadas. Una de ellas es la convención de llamada de C. Algunas de las reglas definidas por ella son:

- Un subprograma es llamado mediante un CALL y finaliza en un RET.
- Al principio del subprograma se debe guardar el contenido de EBP en la pila y copiar el contenido de ESP en EBP. Esto se conoce como “prólogo del subprograma”
- Al final del subprograma se debe sacar de la pila el valor anterior de EBP (dejando en la cima la dirección de retorno) y almacenarlo de nuevo en EBP. A continuación se invocará a la instrucción RET. Esto se conoce como “epílogo del subprograma”.

- Para acceder a los parámetros que han sido pasados al subprograma no los sacamos de la pila, sino que utilizaremos EBP, que tras el prólogo del subprograma apunta a la posición de la pila donde está guardado el valor anterior de EBP. Dado que en [EBP] se encuentra el contenido anterior del registro EBP y en [EBP+4] la dirección de retorno, tendremos en [EBP+8] el primer parámetro, en [EBP+12] el segundo, en [EBP+16] el tercero, etc.
- El programa llamador es el encargado de eliminar los parámetros que ha introducido en la pila puesto que las funciones de C admiten que el número de parámetros sea indefinido (por ejemplo, printf puede tener tantos parámetros como sea necesario). Para esto, en lugar de pop, utilizaremos la instrucción add, sumando a ESP 4 por cada parámetro introducido.
- Se asegura que los registros EBX, ESI, EDI, EBP, CS, DS, SS y ES conservarán su contenido al finalizar el subprograma (podrían cambiar a lo largo de la ejecución del subprograma, pero en cualquier caso, el subprograma está obligado a restaurar el valor original antes de volver al programa llamador). No obstante el resto de registros sí podrían ser modificados. Es recomendable, por lo tanto, guardar los valores de aquellos registros que pueden ser modificados en la pila antes de introducir los parámetros del subprograma. En este punto, podría ser interesante las instrucciones pusha y popa, que introducen y extraen, respectivamente, los ocho registros de propósito general.
- En EAX se almacena el valor de retorno.

Adaptando nuestro código a la convención de llamada de C quedaría así:

```
        ;Codigo del programa llamador
        pusha   ;Guarda los registros de
                ;proposito general en la pila
        push parametro2
        push parametro1
        ;Los parametros se introducen en la pila
        ;en orden inverso al que se pasan al subprograma
        call nombrerutina
        add esp,8       ;Sumamos a esp 4 por cada parametro
        popa    ;Restaura el contenido de los
                ;registros de proposito general.

        ;Resto del codigo del programa llamador

;Inicio de subprograma
nombresubprograma:
        push ebp        ;Estas dos instrucciones son llamadas
        mov ebp, esp    ;el prologo del subprograma

        ;Cuerpo del subprograma
        ;El subprograma utiliza los parametros de la pila

        pop ebp ;Esto es llamado epilogo del subprograma
        ret     ;En la cima de la pila esta
                ;la direccion de retorno
```

Capítulo 6

Anexo

En este capitulo se incluye información relacionada con las prácticas pero que no es necesaria para la realización de las mismas. Queda por lo tanto a decisión del alumno la lectura de este capítulo.

6.1. Modos de operación

6.1.1. Modos de IA-32

La arquitectura IA-32 permite tres modos de operación. El modo determina qué instrucciones y características de la arquitectura pueden ser utilizados. Los modos son:

- **Modo Protegido:** Es el estado nativo del procesador. Una de las posibilidades de este modo es ejecutar directamente software del procesador Intel 8086 en modo de dirección real en un entorno multitarea (esta característica se denomina modo virtual 8086, aunque no es un modo propiamente dicho, solo una característica del modo protegido).
- **Modo real:** Este modo implementa el entorno de programación del Intel 8086 con extensiones (como por ejemplo la posibilidad de cambiar a los otros dos modos). Este modo de operación es el utilizado tras la puesta en marcha del procesador.
- **Modo de gestión del sistema:** Este modo ofrece la posibilidad de ejecutar tareas propias del sistema operativo, como son la gestión de energía o la seguridad del sistema. Este modo se activa al recibir una señal de interrupción en el pin SMI (System Management Interrupt) o al recibir una interrupción SMI desde el controlador de interrupciones programable avanzado (APIC).

6.1.2. Submodos de IA-32e

La arquitectura Intel64 añade a los tres anteriores el modo IA-32e, que a su vez se divide en dos submodos.

- **Submodo de compatibilidad:** Permite que la mayor parte de aplicaciones de 16 y 32 bits se ejecuten sin ser recompiladas en un sistema operativo de 64 bits. Si dichas aplicaciones funcionaban en modo virtual 8086 o si realizaban gestiones sobre el hardware, no funcionarán en este modo. Es similar al modo protegido.
- **Submodo de 64 bits:** Este submodo permite que un sistema operativo de 64 bits ejecute aplicaciones de 64 bits. En este submodo es posible acceder a más registros, así como a un espacio de direcciones lineales mayor (de 64 bits).

Relacionamos ahora los modos de operación con los modelos de memoria anteriormente referenciados. Veremos qué modelos de memoria podemos usar en cada modo.

- **Modo protegido:** El procesador puede utilizar cualquiera de los tres modelos.
- **Modo real:** En este modo el procesador soportará únicamente el modelo de memoria del modo real.
- **Modo de gestión del sistema:** El procesador utiliza en este modo un espacio de direcciones distinto, llamado RAM de gestión del sistema (SMRAM). El modelo usado en este espacio de direcciones el similar al modelo del modo real.
- **Submodo de compatibilidad:** Utiliza cualquiera de los tres modelos de memoria.
- **Submodo de 64 bits:** Únicamente se puede utilizar el modelo de memoria plano.

6.2. Registros de segmento

Los registros de segmento (CS, DS, ES, FS y GS) contienen selectores de segmento de 16 bits. Un selector de segmento es un puntero especial que indica la localización de un segmento en memoria. Generalmente, el programador crea los selectores con directivas del ensamblador o símbolos y el ensamblador se encarga de asignar el valor real del selector. El valor almacenado en estos registros dependerá del modelo de memoria utilizado. En el modelo plano, existen normalmente dos segmentos que se solapan y que empiezan en la dirección 0 del espacio de direcciones lineales: uno de ellos dedicado al código del programa y otro a los datos y las pilas. El registro CS apunta al segmento de código y el resto de registros al segmento de datos y pilas.
En el caso del modelo de memoria segmentada, cada registro apunta a la primera dirección lineal de cada segmento. En un momento dado, un programa puede acceder a un máximo de 6 segmentos, teniendo que cargar primero el selector de segmento en uno de los registros de segmento. Cada uno de los seis registros puede hacer referencia a un tipo de segmento. Existen tres tipos de segmento: segmento de código (apuntado por el registro CS), segmento de datos (apuntado por los registros DS,ES,FS y GS) y segmento de pila (apuntado por el registro SS). En las figuras 1.3 y 1.4 vemos los usos de los registros de segmento según el modelo de memoria usado.

6.3. Banderas del sistema y campo IOPL

Estas banderas del registro EFLAGS controlan las operaciones del sistema y no deben ser modificadas por aplicaciones de usuario.

- **TF (bit 8)** Bandera de trampa (Trap flag) Si su valor es 1 permite depurar un programa paso a paso, si es 0 no lo permite.
- **IF (bit 9)** Bandera de activación de interrupciones (Interrupt enable flag) Si su valor es 1 se responderá a las interrupciones enmascarables. Si es 0 estas serán ignoradas.
- **IOPL (bits 12 y 13)** Campo de nivel de privilegio de entrada y salida (I/O privilege level field) Estos bits indican el nivel de privilegios para acceder a la entrada y salida del programa actual.
- **NT (bit 14)** Bandera de tarea anidada (Nested task flag) Su valor es 1 si la tarea actual está relacionada a la tarea anteriormente ejecutada y 0 si la tarea no está vinculada a ninguna otra.

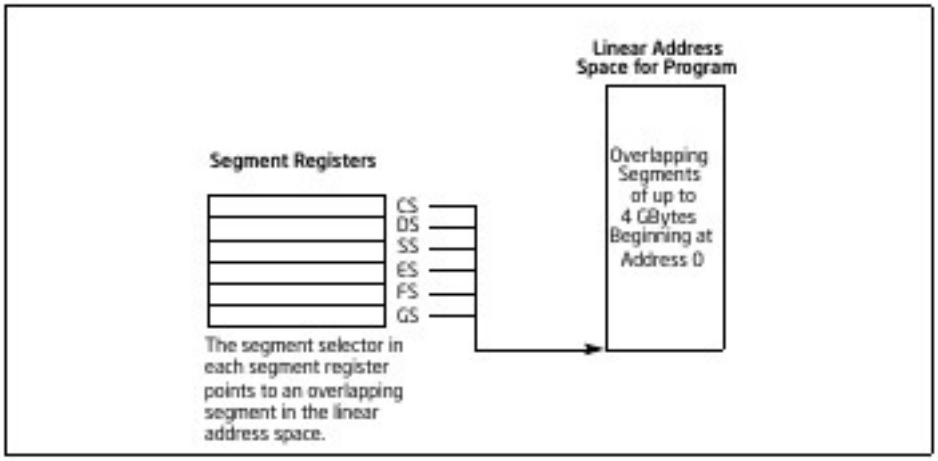

Figura 6.1: Uso de los registros de segmento con el modelo de memoria plano. Imagen obtenida del manual de Intel.

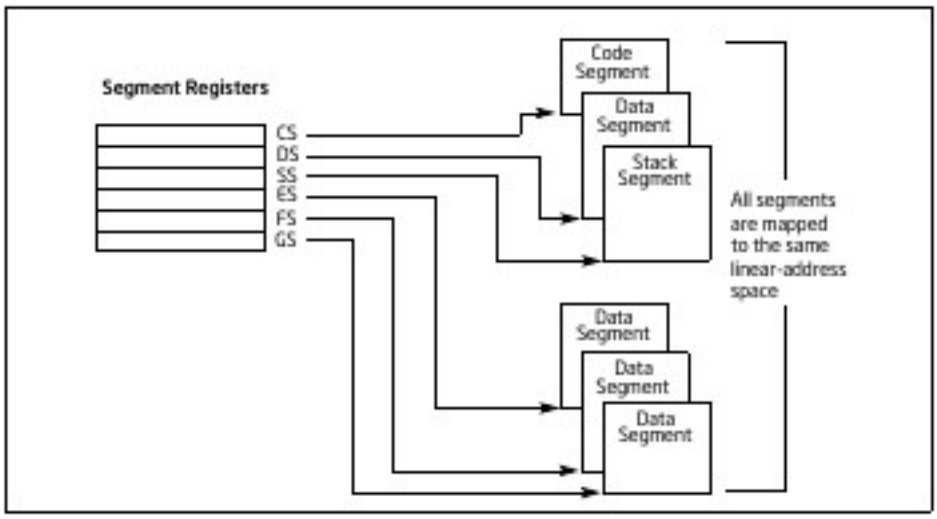

Figura 6.2: Uso de los registros de segmento con el modelo de memoria segmentado. Imagen obtenida del manual de Intel.

- **RF (bit 16)** Bandera de continuación (Resume flag) Se utiliza para controlar la respuesta del procesador ante las excepciones de depuración.
- **VM (bit 17)** Bandera de modo virtual 8086 (Virtual-8086 mode flag). Al poner su valor a 1, activa el modo virtual 8086, al ponerla a 0 el procesador vuelve al modo protegido sin la característica virtual 8086.
- **AC (bit 18)** Bandera de análisis de alineamiento (Alignment check flag) Si su valor y el del bit AM del registro CR0 son 1, entonces permite comprobar el alineamiento de las referencias a memoria. Si alguno de los dos tiene el valor 0, entonces deshabilita la comprobación.
- **VIF (bit 19)** Bandera de interrupción virtual (Virtual interrupt flag)
- **VIP (bit 20)** Bandera de interrupción virtual pendiente (Virtual pending flag)
- **ID (bit 21)** Bandera de identificación (Identifiation flag)

6.4. Recursos adicionales del entorno de ejecución

- **Puertos E/S (I/O ports):** Utilizados para la entrada y salida de datos. Se verán más detalladamente en capítulos posteriores.
- **Registros de control:** Son cinco (van desde CR0 hasta CR4). Determinan el modo de ejecución del sistema operativo, así como características del proceso que se está ejecutando en un instante determinado. En Intel64 los registros pasan de ser de 32 bits a ser de 64. Además se añade un nuevo registro (conocido como registro de prioridad de tarea, o TPR)
- **Registros de gestión de memoria:** Los registros GDTR, IDTR, TR y LDTR detallan la localización de las estructuras de datos usadas en el modo protegido. En Intel64 incrementan su tamaño.
- **Registros de depuración:** Los registros de depuración (que van desde DR0 hasta DR7) son utilizados para las operaciones de depuración. En Intel64 incrementan su tamaño hasta 64 bits.
- **Registros específicos de la maquina (MSRs):** Utilizados para controlar el funcionamiento del procesador. Algunos de registros destacados son el Time-stamp counter (TSC) o los registros de estado de la máquina.
- **Contadores para la monitorización:** Estos contadores permiten controlar el funcionamiento del procesador.

Capítulo 7

Bibliografía

En este capítulo se recopilan las distintas fuentes utilizadas para la realización de esta documentación.

- Manual de Intel (http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html).
- Paul A. Carter, Lenguaje Ensamblador para PC. (Puede descargarse de http://drpaulcarter.com/pcasm).
- Barry B. Brey, Los Microprocesadores Intel : arquitectura, programación e interfaz de los procesadores 8086/8088, 80186/80286, 80386 y 80486 Pentium, Pentium Pro y Pentium II. Pearson Educación.
- Sitio web para desarrolladores de Microsoft (msdn.microsoft.com).
- www.stackoverflow.com