UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

ENS2001. ENSAMBLADOR Y SIMULADOR DEL LENGUAJE ENSAMBLADOR IEEE 694

AUTOR: Federico J. Álvarez Valero
TUTOR: José Luis Fuertes Castro

MARZO 2003

Índice

1. Introducción

En la Facultad de Informática de la Universidad Politécnica de Madrid se imparte la asignatura de Compiladores. Como apoyo a la formación teórica, existe una práctica que abarca casi la totalidad del curso y que recorre de principio a fin el proceso de creación de un compilador, desde el analizador léxico hasta la generación de código final. Sin embargo, no se exige a los alumnos, por ser quizás de índole de otras asignaturas, la generación de ejecutables para un sistema operativo concreto, sino que se genera un código máquina "ficticio". ¿Cómo probar entonces los programas generados?

Hasta el momento, el departamento ha puesto a disposición de los alumnos dos herramientas, *ASS* y *ENS96*. La principal motivación de este Trabajo Fin de Carrera (que en adelante será aludido por el nombre de la aplicación que se va a construir, *ENS2001*) consiste en mejorar las prestaciones que ofrecen estas herramientas, principalmente incorporando una interfaz gráfica más acorde con los tiempos actuales, ya que ambas funcionan en modo texto, y corrigiendo o ampliando algunas de sus características. No obstante, no es ésta la única finalidad que se le quiere dar. A grandes rasgos, *ENS2001* es un simulador, pero pretende ser algo más.

En primera instancia, consultando el Diccionario de la Real Academia Española [RAE 2002] se obtienen las siguientes entradas:

> *simulador, ra. (Del lat. simulator, -oris).*
>
> 1. *adj. Que simula. U. t. c. s.*
> 2. *m. Tecnol. Aparato que reproduce el comportamiento de un sistema en determinadas condiciones, aplicado generalmente para el entrenamiento de quienes deben manejar dicho sistema.*
>
> *simular. (Del lat. simulare).*
>
> 1. *tr. Representar algo, fingiendo o imitando lo que no es.*

En el caso de *ENS2001*, el sistema real no existe, sólo se trata de una definición teórica, pero sin embargo, de una gran utilidad didáctica. Seguramente no se construya nunca un ordenador con una arquitectura y un juego de instrucciones como los que *ENS2001* pretende modelizar. *ENS2001*, aparte de permitir a los alumnos de Compiladores probar y depurar sus prácticas con una mayor comodidad, se ha diseñado pensando en personas interesadas en aprender los fundamentos de la programación en ensamblador y temas relacionados con la Arquitectura de Computadores. Se ha intentado construir un sistema con una arquitectura sencilla, unos recursos generosos, y sobre todo un lenguaje máquina lo más homogéneo posible.

En cuanto a los aspectos más técnicos, *ENS2001* se basa en dos ideas fundamentales: una interfaz de usuario cómoda y una implementación multiplataforma. En un principio se pensó en el entorno *Java* como el soporte a estas dos ideas. Sin embargo, al final se ha optado por una solución mixta *C*/*C++* más clásica, con un motor de simulación totalmente independiente de la interfaz, de forma que se presenta con dos "sabores": interfaz de línea de comandos o consola (semejante a la de *ENS96*), e interfaz gráfica.

ENS2001 se compone de dos partes bien diferenciadas, que se presentan integradas pero que podrían funcionar independientemente. Además, en cada una de ellas se ha aplicado una metodología de desarrollo distinta, intentando que el trabajo realizado en cada una de ellas sea lo más completo posible.

1. Por una parte, el simulador, encargado de recrear la arquitectura teórica, está construido siguiendo técnicas de orientación a objetos, con *C++* estándar como lenguaje fuente. Como se ha comentado anteriormente, el motor de simulación es el corazón del sistema, y la interfaz de usuario se construye a su alrededor, de forma que podría sustituirse por otra distinta sin afectar al funcionamiento interno (y de la misma manera, podría sustituirse el motor por otro más potente u optimizado sin modificar la interfaz de usuario).
2. Por otra parte, el ensamblador, que se encargará de traducir los programas escritos en lenguaje máquina (comprensible por el simulador), se trata de un clásico ejercicio de diseño e implementación de compiladores, construido con ayuda de las herramientas clásicas *Lex* y *Yacc* y en lenguaje *C* estándar.

Sobre estos dos módulos se superponen dos interfaces de usuario diferentes, una de línea de comandos y otra gráfica. Ambas interfaces, consola y gráfica, están construidas según el paradigma orientado a objetos y en *C++* también. Sin embargo, para la gráfica se ha contado con la inestimable ayuda de una herramienta de desarrollo visual de aplicaciones, como es *Borland C++ Builder* en su versión 5, que ha simplificado enormemente el diseño y tratamiento de los elementos gráficos que componen una interfaz de usuario (ventanas, botones, cuadros de texto, etc.).

En resumen, el ensamblador, partiendo de listados de código fuente (escritos en lenguaje ensamblador *IEEE 694*), convierte dichos programas en lo que será el código máquina del simulador (secuencias de unos y ceros). Gracias a la interfaz, se accede fácilmente a todos los componentes de la arquitectura simulada y se permite conocer su estado e interactuar con ellos.

Estructura del Trabajo Fin de Carrera

En primer lugar, se efectuará un pequeño repaso de la actualidad en cuanto a la temática objeto de este trabajo, las soluciones existentes, sus puntos débiles, y qué mejoras se van a introducir que justifiquen el desarrollo de este Trabajo Fin de Carrera.

La solución del problema se va a abordar siguiendo el proceso habitual de desarrollo software, que constituye el cuerpo principal de la documentación.

En principio se presenta la Especificación de Requisitos Software, que es el documento en el que se describe la funcionalidad que debe tener el sistema que se construirá posteriormente, los problemas que debe resolver y las restricciones que debe cumplir. Es en definitiva donde se describen las características del software.

A continuación se encuentra la fase de Análisis, en la que se va a construir un modelo que recoja la problemática planteada durante la fase de Especificación de Requisitos, sin pensar todavía en ningún tipo de solución concreta.

En la fase de Diseño se abordará la solución de los problemas planteados, en un nivel de abstracción teniendo en mente la construcción de un sistema software, pero en el que no se va a entrar en detalles todavía acerca de su implementación.

Además, complementariamente, ya que se trata de un sistema con el que van a interaccionar los usuarios, se diseñarán las interfaces apropiadas para la comunicación hombre-máquina (en este caso serán dos, línea de comandos y gráfica).

En la fase de Implementación se describen los detalles más relevantes que surgen al poner en práctica las soluciones adoptadas en la fase de Diseño. No se va a hacer un recorrido exhaustivo del código fuente. Con ayuda de la documentación y los comentarios intercalados en el propio código fuente, sería suficiente para comprenderlo en su totalidad.

Como última parte de la solución, para comprobar la corrección del sistema, se presenta un Plan de Pruebas, desde el nivel unitario (sin detallar, comentando únicamente los casos más interesantes) hasta las pruebas de sistema.

A modo de colofón, tras finalizar el proceso de desarrollo, pruebas e implantación del sistema, se presentan una serie de conclusiones y se esbozan algunos caminos por los que deberían seguir futuros desarrollos en este campo.

2. Estado de la Cuestión

Aunque la programación en ensamblador está prácticamente en desuso en el contexto de los ordenadores personales (salvo en el ámbito de programación de sistemas operativos), todavía es de gran utilidad en el campo de sistemas empotrados, de tiempo real, o donde la velocidad de ejecución sea crítica. Hay que recordar que el lenguaje ensamblador constituye el paso más cercano al lenguaje de la máquina "legible" por las personas.

El hardware de los ordenadores personales ha experimentado una evolución meteórica en los últimos años, tanto en velocidad de cómputo como en capacidad de almacenamiento. Esto libera al programador de la ardua tarea de optimizar sus programas tanto en espacio ocupado como en velocidad, y le permite centrarse en la tarea creativa: el diseño. Por ello, cada vez se utilizan lenguajes con una capacidad de abstracción más alta, más cercanos al lenguaje natural, que prácticamente se limitan a "describir" la solución de los problemas, siendo tarea de los compiladores e intérpretes traducir dicha descripción de alto nivel al lenguaje máquina.

Sin embargo, aún puede existir la posibilidad de necesitar bajar de nivel, para obtener más velocidad de ejecución (tiempo real, grandes sistemas multiusuario), acceder directamente al hardware del ordenador (programación de controladores o *drivers*), creación de dispositivos electrónicos actualizables (mediante *firmware* implantado en dispositivos de tipo *flash BIOS*), etc.

En el mercado de ordenadores personales compatibles han aparecido a lo largo de la historia multitud de ensambladores. En la actualidad, siempre haciendo referencia al sistema operativo *Microsoft Windows*, prácticamente copan el panorama dos herramientas: *Borland Turbo Assembler* [Borland TASM 2002] y *Microsoft Macro Assembler* [Microsoft MASM 2002]. Además de éstas, se pueden encontrar multitud de herramientas *shareware* y de libre distribución, pero sobre todo para entornos *Linux*. Tampoco es casualidad que los compiladores más extendidos dentro del ámbito comercial sean de estas mismas compañías, y de hecho las herramientas de programación y depuración en ensamblador se suelen incluir en los paquetes de desarrollo, como por ejemplo *Borland Builder* o *Microsoft Developer Studio*. Realmente, la potencia de estas herramientas no está en el propio ensamblador, sino en la herramienta de depuración, que permite al programador ejecutar código paso a paso y tener un control muy exhaustivo sobre los recursos de la máquina.

Aunque el lenguaje ensamblador de la familia de procesadores *Intel x86* y compatibles (*AMD*, *Cyrix*, etc.) es independiente del sistema operativo que esté instalado en el ordenador, no se suele programar en ensamblador sin apoyarse en ciertos servicios del sistema, por lo que se hace necesario emplear ensambladores adecuados a cada sistema operativo (*Windows*, *MS-DOS*, *Linux*, *UNIX*, etc.). Las dos herramientas citadas se ejecutan en entornos *MS-DOS* o *Windows*. Por tanto, no serían válidos para escribir sofware en ensamblador que se ejecute en entorno *Linux*, por ejemplo, aunque el procesador sobre el que corra sea el mismo.

Debido al que el uso de estas herramientas se aparta del objetivo de este trabajo, no se va a entrar en detalles acerca de su uso, si bien el funcionamiento básico de sus depuradores es el modelo al que *ENS2001* trata de acercarse.

Anteriormente se ha hablado acerca de sistemas empotrados, pequeños (o no tanto) montajes electrónicos gobernados por un microprocesador o un microcontrolador que suelen estar especializados en una tarea determinada (los ordenadores personales tienen un propósito más general). La solución más común consiste en construir un programa en ensamblador que rija el funcionamiento del sistema, y almacenarlo en algún tipo de memoria no volátil.

Para programar estos dispositivos, se podría emplear un ensamblador cruzado. El programa ensamblador corre en un ordenador personal y el código máquina generado se transfiere a la memoria del sistema empotrado. Podría ser, incluso, que el sistema no esté todavía construido, por lo que sería de gran utilidad un simulador en el que observar el comportamiento del diseño incluso antes de construirlo (con el coste económico y en tiempo que ello implica).

ENS2001 bebe de estas dos fuentes, ya que se trata de un ensamblador para un sistema ficticio, con fines didácticos, y un simulador para comprobar y depurar la ejecución de los programas que el usuario construya. Tanto el ensamblador como el simulador podrían extenderse o adaptarse al formato de algún microprocesador en concreto sin demasiado esfuerzo.

2.1. Breve introducción a la teoría de compiladores

A grandes rasgos, un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto [Aho 1990].

En la compilación hay dos partes, análisis y síntesis. La parte correspondiente al análisis divide el programa fuente en sus elementos básicos y crea una representación intermedia del mismo. La parte correspondiente a la síntesis construye el programa objeto deseado a partir de la representación intermedia.

El análisis consta de tres fases:

1. Análisis léxico, en el que la cadena de caracteres que constituye el programa fuente se lee correlativamente de izquierda a derecha y se agrupa en componentes léxicos o *tokens* (secuencias de caracteres con un significado colectivo).
2. Análisis jerárquico o sintáctico, en el que los componentes léxicos se agrupan jerárquicamente en colecciones anidadas con un significado colectivo. Es decir, se agrupan en frases gramaticales que el compilador utiliza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico.
3. Análisis semántico, en el que se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan al significado adecuado. Se trata de encontrar posibles errores semánticos y reunir la información sobre tipos de datos para la fase posterior de generación de código, verificando si cada operador tiene operandos permitidos por la especificación del lenguaje fuente. En esta fase se suele usar la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar este

representación intermedia como un programa para una máquina abstracta, y debe cumplir dos requisitos importantes: ser fácil de producir y fácil de traducir al programa objeto.

Antes de producir el código objeto, se puede introducir una fase de optimización de código, que trata de mejorar el código intermedio, de modo que resulte de él un código máquina más rápido de ejecutar.

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones máquina que se corresponden con la misma tarea.

2.2. Breve introducción a la arquitectura *Von Neumann*

La arquitectura del ordenador define su funcionamiento. El modelo básico de arquitectura empleado en los computadores digitales fue establecido por John Von Neumann. Esta arquitectura es prácticamente idéntica a la empleada en la actualidad por la mayoría de fabricantes.

La Figura 2.2.1 muestra la estructura general de un computador tipo *Von Neumann* [De Miguel 1996]. Esta máquina es capaz de ejecutar una serie de instrucciones elementales, denominadas "instrucciones máquina", que deben estar almacenadas en la memoria principal para poder ser leídas y posteriormente ejecutadas. Puede observarse que la máquina está compuesta por los siguientes bloques:

- Memoria principal.
- Unidad aritmética.
- Unidad de control.
- Unidad de entrada/salida.

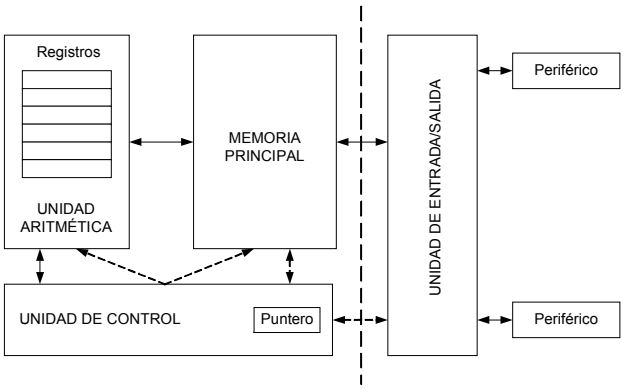


Figura 2.2.1. Arquitectura *Von Neumann*

La memoria principal se compone de un conjunto de celdas del mismo ancho de bits. En un instante determinado se puede seleccionar una de esas celdas, para lo que se emplea una dirección que la identifica. Sobre la celda seleccionada se puede realizar una operación de lectura, que permite recuperar el valor almacenado previamente en la celda, o de escritura, que permite almacenar un nuevo valor. Estas celdas se emplean tanto para almacenar datos como las instrucciones máquina, pudiendo entremezclarse [De Miguel 1996].

La unidad aritmética permite realizar una serie de operaciones básicas tales como suma, resta, y lógico (*and*), o lógico (*or*), etc. Los datos sobre los que opera esta unidad provienen de la memoria principal, y pueden estar almacenados de forma temporal en algunos registros de la propia unidad aritmética [De Miguel 1996].

La unidad de control se encarga de leer, una tras otra, las instrucciones máquina almacenadas en la memoria principal, y de generar las señales de control necesarias para que todo el computador funcione y ejecute las instrucciones leídas. Para conocer en todo momento la posición de memoria en la que se encuentra la instrucción que corresponde ejecutar, existe un registro apuntador denominado contador de programa (PC, *Program Counter*) que contiene dicha información [De Miguel 1996].

La unidad de entrada/salida realiza la transferencia de información con unas unidades exteriores llamadas periféricos, lo que permite, entre otras muchas cosas, cargar datos y programas en la memoria principal y obtener resultados impresos [De Miguel 1996].

Finalmente, es conveniente resaltar que existen unos caminos (*buses*) cuyo objetivo es hacer que las instrucciones y los datos circulen entre las distintas unidades del computador. Estos caminos están representados por flechas de trazo continuo en la Figura 2.2.1 [De Miguel 1996].

Se denomina Unidad Central de Proceso (UCP o bien CPU, de *Central Processing Unit*) al conjunto de la unidad de control, los registros y la unidad aritmética de un computador. En definitiva, el bloque que ejecuta las instrucciones. Para formar un computador hay que añadir la memoria principal, la unidad o unidades de entrada/salida y los periféricos. Con frecuencia se denomina Procesador a una UCP, aunque a veces se extrapola este concepto al conjunto formado por una UCP y una pequeña memoria, como, por ejemplo, cuando se habla de procesadores de entrada/salida [De Miguel 1996].

Las fases de ejecución de las instrucciones máquina son las siguientes:

1. La unidad de control envía a la memoria principal la dirección de la instrucción que se va a ejecutar, que está almacenada en el contador de programa, y activa las señales de control necesarias para que ésta le entregue la mencionada instrucción (*fetch*)
2. La unidad de control recibe la instrucción, la analiza y, en caso necesario, lee los operandos de la memoria principal, enviando su dirección y activando las correspondientes señales de control (decodificación).
3. Seguidamente, siempre bajo la supervisión de la unidad de control, se realiza la operación sobre los operandos y, si es necesario, se salvaguarda el resultado en memoria principal o en un registro (ejecución).
4. Una vez ejecutada la instrucción, se incrementa el contador de programa, con lo que se puede pasar a ejecutar la instrucción siguiente.

Si bien en [De Miguel 1996] se indica así, los pasos 3 y 4 podrían intercambiarse. Así, en el momento de ejecutarse la instrucción, el contador de programa ya poseería la dirección de la siguiente. Si se siguen los pasos 3 y 4 ordenadamente, el paso 4 sería superfluo tras ejecutar una instrucción que altere el contenido del contador de programa, rompiéndose la uniformidad de este esquema.

Este funcionamiento muestra que el ordenador únicamente es capaz de ir ejecutando una secuencia consecutiva de instrucciones máquina. Dicha secuencia de instrucciones recibe

el nombre de programa. Cabe destacar que para romper la secuencia lineal (o consecutiva) de ejecución de instrucciones, es necesario que existan instrucciones máquina que permitan alterar el contenido del contador de programa, es decir, que permitan hacer lo que se denominan bifurcaciones.

Para que el ordenador pueda realizar una función determinada, es necesario que previamente se realice la descomposición de esa función en su correspondiente conjunto de instrucciones máquina. Este proceso recibe el nombre de programación. Además, para que el ordenador sea capaz de ejecutar un programa se deben verificar las siguientes condiciones:

- Debe existir el correspondiente programa en lenguaje máquina.
- Tanto el programa como sus datos deben residir en memoria principal. Esto exige generalmente una operación de transferencia, desde un periférico de almacenamiento secundario (por ejemplo, un disco), hasta la memoria principal, operación que deberá efectuarse mediante un programa denominado cargador.
- El contador de programa de la unidad de control debe ser actualizado con la dirección correspondiente a la primera instrucción del mencionado programa.

Cuando estas tres condiciones se cumplan, el ordenador irá leyendo y ejecutando las instrucciones que conforman el programa. Para ello, otro programa deberá haber realizado ciertas tareas para que dichas condiciones se satisfagan. Ese otro programa detendrá su ejecución una vez cumplidas, y continuará la misma una vez que el nuevo haya finalizado su ejecución.

2.3. Breve introducción al lenguaje ensamblador *IEEE 694*

El estándar *IEEE 694* define tanto un posible juego de instrucciones, como los modos de direccionamiento que deben admitir sus operandos, así como la notación que se debe emplear a la hora de escribir los programas en lenguaje ensamblador. Como el estándar no es de libre acceso (hay que pagar por una copia), la descripción está basada en la información que se puede encontrar en [De Miguel 1996].

El estándar *IEEE 694* define los siguientes modos de direccionamiento:

Modo	**Prefijo**	**Ejemplo**
Absoluto	prefijo `/`	`/dir`
Página base	prefijo `!`	`!dir`
Indirecto	Corchetes	`[dir]`
Relativo a PC	prefijo `$`	`$dir`
Inmediato	prefijo `#`	`#valor`
Relativo a Registro Base	Corchetes	`desp[.reg]` o `[.reg,desp]`
Registro	prefijo `.`	`.dir`
Autopreincremento	prefijo `++`	`++dir`
Autopostincremento	sufijo `++`	`dir++`
Autopredecremento	prefijo `--`	`--dir`
Autopostdecremento	sufijo `--`	`dir--`

Para describir el juego de instrucciones propuesto por el estándar, se dividen en grupos según la clasificación de [Fairclough 1982]. Aunque las instrucciones máquina son cadenas binarias, para identificarlas con mayor facilidad se emplearán sus nombres más corrientes

(en inglés). Los códigos mnemónicos empleados por los ensambladores de los distintos microprocesadores se deriban de estos nombres:

Instrucciones de transferencia de datos. Permiten copiar al operando destino la información almacenada en el operando origen.

`MOVE`	Transfiere el contenido de un registro a otro, o de una posición de memoria a otra.
`STORE`	Transfiere el contenido de un registro a la memoria.
`LOAD`	Transfiere el contenido de una posición de memoria a un registro.
`MOVE BLOCK`	Transfiere un bloque de datos.
`MOVE MULTIPLE`	Copia el contenido del origen en múltiples posiciones de memoria.
`EXCHANGE`	Intercambia el contenido de los operandos.
`CLEAR`	Pone a "ceros" el destino.
`SET`	Pone a "unos" el destino.
`PUSH`	Transfiere el origen a la cabecera de la pila.
`POP`	Transfiere la cabecera de la pila al destino.

Instrucciones de bifurcación. Permiten alterar la secuencia normal de ejecución del programa. Aunque no dejan de ser instrucciones de transferencia en las que el destino es el contador de programa, se consideran de forma independiente.

`BRANCH`	Bifurcación condicional o incondicional, en ocasiones combinada con el incremento o decremento automático de un registro.
`CALL`	Bifurcación con retorno, empleada para llamar a una subrutina.
`RETURN`	Instrucción complementaria de `CALL`, que restituye la dirección del programa llamante.
`SKIP`	Bifurcación condicional que salta una instrucción si se cumple la condición.
`RETURN WITH SKIP`	Incrementa la dirección de retorno antes de realizar el mismo.

Las bifurcaciones condicionales se ejecutan o no basándose en ciertos criterios (condiciones). El estándar considera los que se muestran en la Tabla 2.3.1.

Condición	Significado	Condición	Significado
`ZERO [Z]`	Cero	`NOT ZERO [NZ]`	Distinto de cero
`EQUAL [E]`	Igual	`NOT EQUAL [NE]`	Distinto
`CARRY [C]`	Acarreo	`NOT CARRY [NC]`	Sin acarreo
`POSITIVE [P]`	Positivo	`NOT POSITIVE [NP]`	No positivo
`NEGATIVE [N]`	Negativo	`NOT NEGATIVE [NN]`	No negativo
`OVERFLOW [V]`	Desbordamiento	`NO OVERFLOW [NV]`	Sin desbordamiento
`GREATER THAN [GT]`	Mayor que	`GREATER THAN OR EQUAL [GE]`	Mayor o igual

LESS THAN [LT]	Menor que	LESS THAN OR EQUAL [LE]	Menor o igual
HIGHER* [H]	Superior a	NOT HIGHER* [NH]	No superior
LOWER* [L]	Inferior a	NOT LOWER* [NL]	No inferior
TRUE [T]	Verdadero	FALSE [F]	Falso
PARITY EVEN [PE]	Paridad par	PARITY ODD [PO]	Paridad impar

Tabla 2.3.1. Codificación de Modos de Direccionamiento

* NOTA: Estas condiciones se refieren a operandos sin signo, por lo que son distintas de las de mayor y menor.

Instrucciones aritméticas. Efectúan operaciones aritméticas sobre los operandos.

ADD	Suma.
SUBSTRACT	Resta.
INCREMENT	Incrementa.
DECREMENT	Decrementa.
MULTIPLY	Multiplica.
DIVIDE	Divide.
NEGATE	Cambia de signo.
ABSOLUTE	Valor absoluto.
ADD WITH CARRY	Suma, añadiendo el acarreo del resultado anterior. Se emplea en operaciones de precisión múltiple.
SUBSTRACT REVERSE	Resta en orden inverso.
SUBSTRACT WITH BORROW	Resta, teniendo en cuenta el acarreo de la resta de la operación anterior. Se usa en precisión múltiple.

Instrucciones de comparación. Comparan dos o más operandos entre sí, almacenando el resultado de la comparación en los biestables de estado.

COMPARE	Resta o hace la operación XOR de cada bit de dos o más operandos. No se almacena el resultado, pero sí se modifican los biestables de estado
TEST	Comparación con cero.

Instrucciones lógicas. Se efectúan en cada uno de los bits de los operandos de forma independiente.

AND	Y lógico.
OR	O lógico.
NOT	Negación lógica.
XOR	O exclusivo lógico.

Instrucciones de desplazamiento. Desplazan los bits de un operando hacia la izquierda o hacia la derecha.

SHIFT	Desplazamiento de bits, a izquierdas o derechas, aritmético o lógico.
ROTATE	Rotación de bits, a izquierdas o derechas, aritmética o lógica.

Instrucciones de bit. Efectúan operaciones sobre un bit en concreto del operando. Por tanto, además del operando se deberá indicar sobre qué bit se va a operar.

TEST	Devuelve el valor del bit indicado (cero o uno).
SET	Pone el valor del bit indicado a uno.
CLEAR	Pone el valor del bit indicado a cero.

Instrucciones de entrada/salida y misceláneas. Las instrucciones de entrada/salida son instrucciones de transferencia en las que el origen o el destino son un registro de un periférico.

INPUT	Transfiere la información de un puerto de entrada a un registro o a memoria.
OUTPUT	Operación inversa a INPUT.
WAIT	Detiene la ejecución hasta que se recibe una interrupción externa.
HALT	Detiene el procesador.
CONVERT	Cambia los formatos de las instrucciones, generalmente para realizar operaciones de entrada/salida, o bien extensiones de signo.
NO OPERATION	No realiza ninguna operación. Se usa para rellenar huecos en los programas o para temporizar esperas.

Como puede apreciarse, el estándar propone un conjunto rico y amplio de instrucciones. A la hora de aplicarlo a una arquitectura real, puede ser interesante prescindir de algunas de ellas o introducir otras nuevas. De hecho hay investigadores que teorizan la existencia de juegos de instrucciones completos, es decir, que permiten calcular en un tiempo finito cualquier tarea computable, compuestos de tan solo dos (decrementar y bifurcar si cero e incrementar [Minsky 1967]) o incluso una instrucción, como propone Van Der Poel.

2.4. Otros ensambladores simbólicos

Como ejemplo de herramienta con la que *ENS2001* querría compararse, pero enfocada al ámbito de las estaciones de trabajo, existe una herramienta llamada *SPIM S20* [Larus 2002]. *SPIM S20* es un simulador que ejecuta programas para los ordenadores *MIPS R2000/R3000*. Lee y ejecuta ficheros de código ensamblador. Es un sistema que aúna ensamblador, simulador y depurador, así como interfaz para algunos servicios del sistema operativo (Figura 2.4.1).

Citando la propia documentación de *SPIM*, "la arquitectura de los ordenadores *MIPS* es simple y regular, lo que la hace sencilla de aprender y comprender", al igual que la arquitectura del sistema que *ENS2001* pretende simular. Por otra parte, la siguiente pregunta es: ¿por qué usar un simulador, especialmente si el usuario puede acceder al hardware simulado? Desde un punto de vista académico, se puede enfocar la simulación de manera que se centre en los aspectos más interesantes de la arquitectura, dejando a un lado detalles que añaden dificultad para comprender el modelo, como podrían ser temas de optimización, paralelismo, etc. De acuerdo de nuevo a la documentación de *SPIM*, los simuladores pueden ofrecer un entorno mejor para programar a bajo nivel que las máquinas reales, porque pueden detectar más errores y ofrecer más características que los ordenadores reales. Por ejemplo, "*SPIM* tiene una interfaz basada en *X-Windows* que es mejor que la mayoría de los depuradores de las máquinas reales".

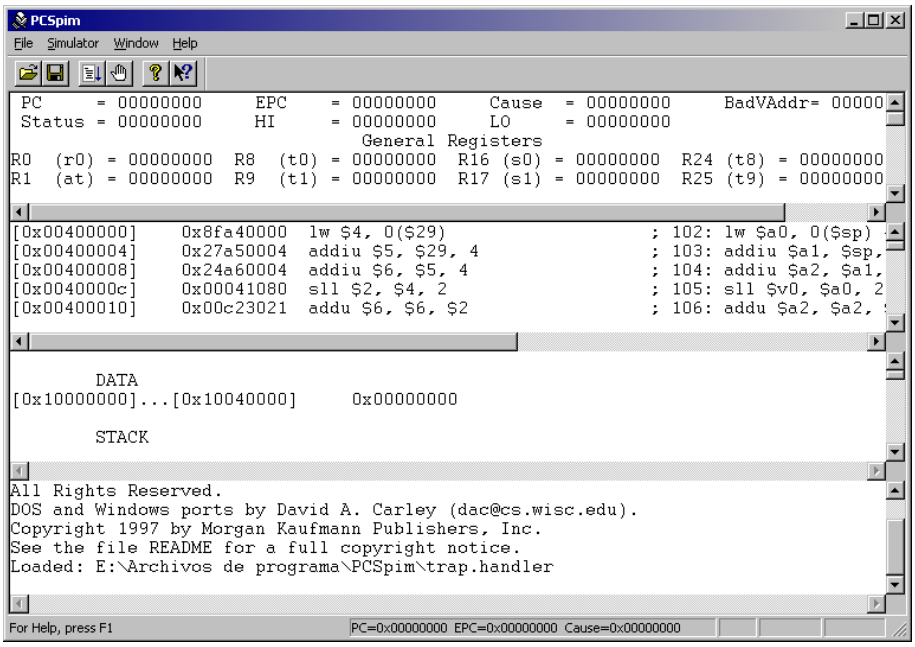


Figura 2.4.1. *PCSPIM* para *Windows*

Otro buen ejemplo de simulador orientado al aprendizaje del lenguaje ensamblador es *Simuproc* [Yepes 2002]. *Simuproc* es un simulador de un procesador hipotético, con el cual es posible aprender las nociones básicas para empezar a programar en lenguaje ensamblador. En él se puede observar todo el proceso interno de ejecución del programa a través de cada ciclo del procesador. La ventana principal se observa en la Figura 2.4.2.

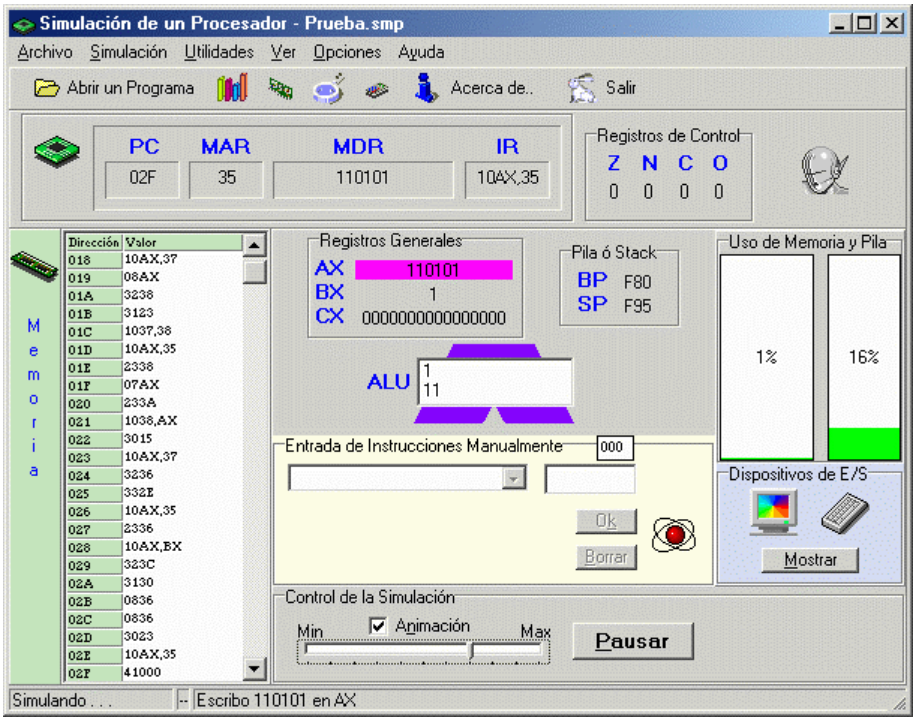


Figura 2.4.2. *Simuproc* para *Windows*

El juego de instrucciones soportado se basa en la arquitectura *Intel x86*, conteniendo instrucciones de transferencia, manejo de pila, aritméticas enteras, lógicas, desplazamiento de bits, comparación, bifurcación, entrada/salida, no operación y fin de ejecución.

El procesador simulado consta de una memoria de 4096 posiciones de 16 bits, tres registros generales de 16 bits (AX, BX y CX), cuatro registros apuntadores (contador de programa, registro de direcciones, registro de datos y registro de instrucción), registro de base de pila, registro puntero de pila y registro de control (*flags*).

La aplicación cuenta además con algunas utilidades, como son un editor de memoria, un conversor de bases, estadísticas del programa en ejecución y un vigilante de memoria, en el que se pueden observar las variables en sus posiciones de memoria.

Con intenciones más abstractas, orientado a la simulación de ordenadores virtuales, se encuentra una herramienta llamada *BSVC* [Mott 1998]. *BSVC* ofrece un marco de trabajo que permite construir y simular ordenadores virtuales basados en el microprocesador *Motorola 68000*. Está escrito en *C++* y *Tcl/Tk*, originalmente para sistemas *UNIX* y siendo portado posteriormente a sistemas *Windows*. El entorno de trabajo de la herramienta se puede observar en la Figura 2.4.3.

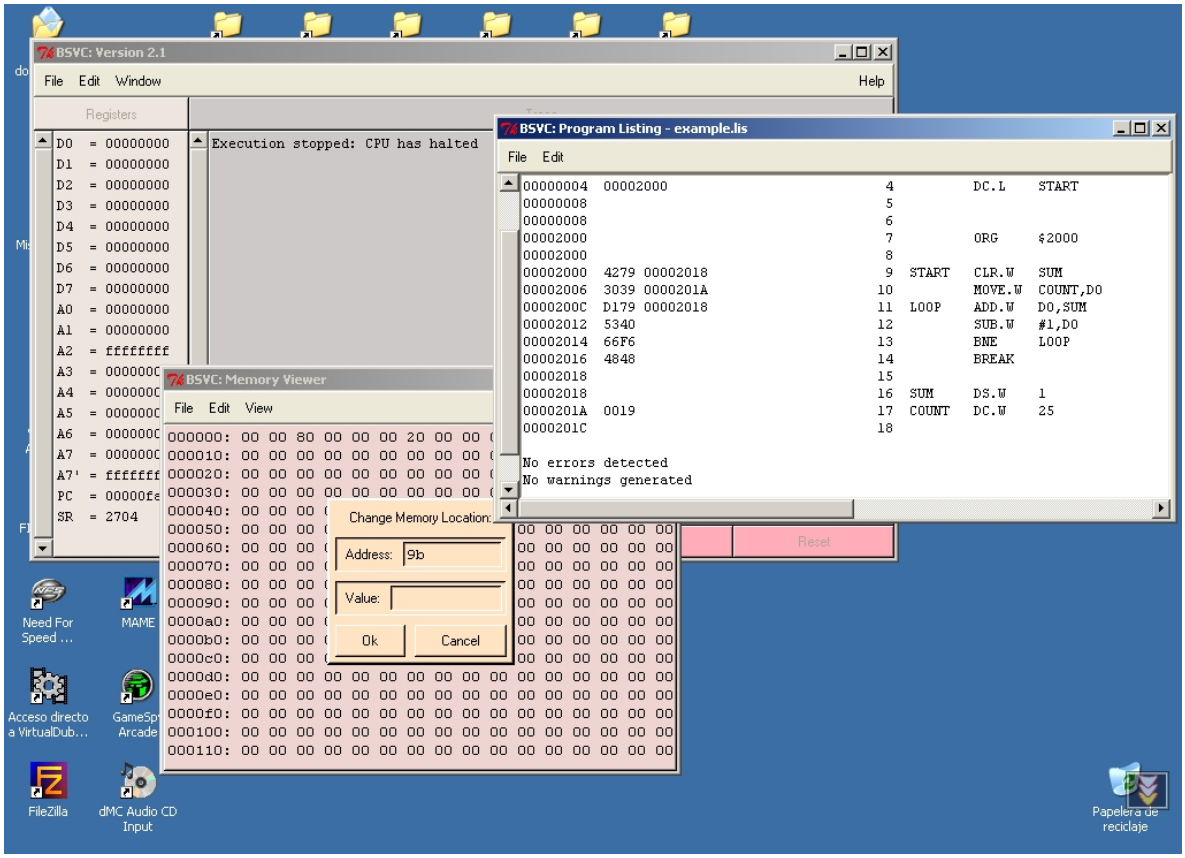


Figura 2.4.3. Entorno de trabajo de *BSVC*

BSVC comenzó como un simulador del procesador *Motorola 68000* que soportaba el uso del *chip UART 6850*. Después de descomponer en componentes el simulador (procesador y dispositivos), saltaba a la vista que podía convertirse en un simulador de ordenadores virtuales que permitiera al usuario "construir" su propio ordenador. En vez de un único objeto representando al *68000*, consistiría en varios procesadores y varios dispositivos que se podrían interconectar para construir un ordenador. Esto permite al usuario construir un ordenador virtual que simule los ordenadores basados en placas de *68000* en el laboratorio

ECE 218 de la Universidad del Estado de Carolina del Norte, o cualquier otro ordenador mientras que el procesador y los dispositivos estén soportados.

La distribución de *BSVC* consta de:

- Simulador y ensamblador *Motorola 68000* (soportando el *chip M68681 UART dual* y un temporizador).
- Simulador de *Motorola 68360* (del tipo *CPU32*).
- Interfaz gráfico de usuario (escrito en *Tcl/Tk*).
- Clases *C++* para el desarrollo de simuladores.

Por último, escapando al ámbito que quiere abarcar este Trabajo Fin de Carrera, hay que mencionar el proyecto *VisualOS* [Estrada 2002]. *VisualOS* es un simulador visual con fines educativos que corre sobre *GNOME/GTK+*. En la Figura 2.4.4 se puede observar una pantalla en funcionamiento.

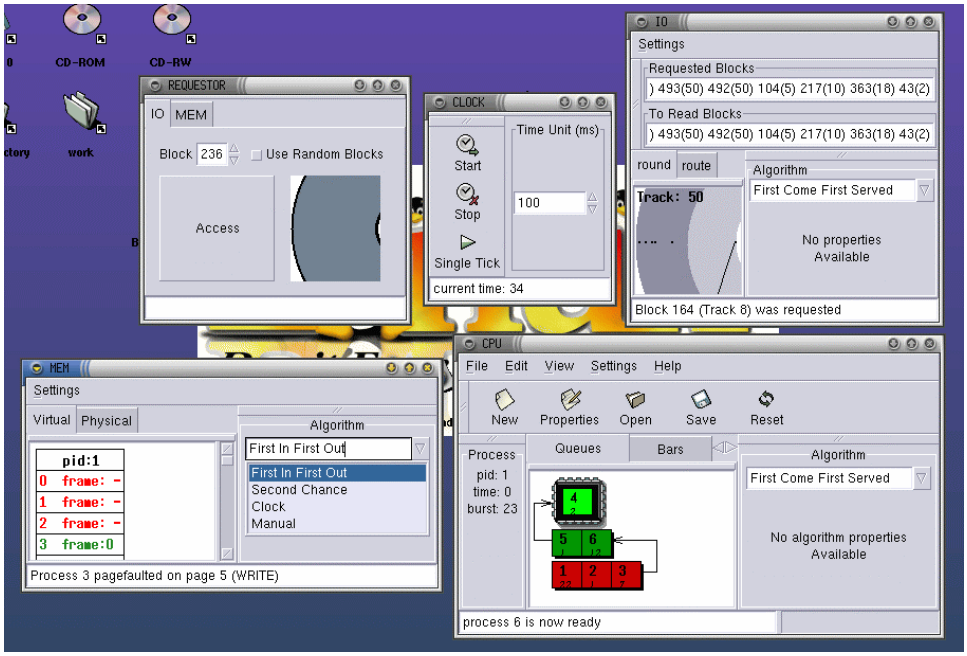


Figura 2.4.4. *VisualOS* para *X-Windows*

Representa visualmente el funcionamiento de un sistema operativo, permitiendo al usuario seleccionar entre los diferentes algoritmos existentes para cada subsistema simulado (*CPU*, memoria y entrada/salida a disco).

2.5. Los predecesores de *ENS2001*

En el apartado introductorio se citaron las dos herramientas que sirven de modelo a *ENS2001*. Se estudiarán a continuación con un poco más de detalle, prestando especial atención a sus características más positivas e intentando identificar las carencias que presenten, para intentar superarlas. Estas aplicaciones son *ASS* y *ENS96*. El pequeño estudio se centra en la arquitectura que simulan y la funcionalidad que ofrecen.

2.5.1. ASS

La información a la que se hace referencia ha sido extraída del manual de instrucciones que acompaña a la versión 1.0 [ASS 1992]. Existen ejecutables compilados para *MS-DOS* y para *Linux*. Este estudio se ha efectuado sobre la versión *MS-DOS* (aunque en teoría ambas son idénticas). Las dos versiones van acompañadas de algunos ficheros con código de ejemplo. Además, la distribución para *Linux*, cuya versión es la 1.2, incluye el código fuente en *C*. El ejecutable tiene como marca de fecha y hora: 03-02-1992-15:26.

Máquina Virtual.

- Consta de un procesador y una memoria de 16 KPalabras de 16 bits (32 Kbytes).
- Maneja enteros con signo de 16 bits, en el rango -32768..32767.
- El código ejecutable se almacena a partir de la dirección 0.
- La pila comienza en la última posición de memoria y crece hacia posiciones inferiores (no es configurable).
- El procesador consta de 5 registros:
    - **Acumulador** (Acum).
    - **Contador de Operación** (CO), es el contador de programa.
    - **Puntero de Pila** (SP).
    - **Registro Especial** (R) para controlar resultados de operaciones aritméticas. Tras una división, R almacena el resto de la misma. En los demás casos, contiene los 16 bits altos de la diferencia entre el resultado correcto y el obtenido, interpretados como enteros con signo de 32 bits (R=0 significaría que no se ha producido desbordamiento).
    - **Registro Índice** (IX).

- Se lee el código fuente de un archivo de texto y lo traduce al código máquina que entiende la máquina virtual de *ASS*.
- Los datos se interpretan como enteros con signo de 16 bits y las direcciones como enteros sin signo (se desprecia el bit más significativo).

Modos de direccionamiento.

- **Inmediato** → ,iOP
- **Directo** → OP
- **Indirecto** → (OP)
- **Relativo** → [OP]

No se permite usar direccionamiento inmediato en las instrucciones `STA`, `STSP`, `STR`, `STIX`, `INC`, `DEC`, `NEG`, `NOT`, `POP`, `INPUT`, `ININT`, `WRSTR`, los saltos y las llamadas.

Si se omite el operando, se supone que se está haciendo referencia al acumulador. Esto no se permite en los modos de direccionamiento inmediato y relativo. Tampoco en el modo de direccionamiento directo para los saltos y llamadas y en la instrucción `WRSTR`.

Juego de Instrucciones.

El juego de instrucciones es el siguiente (en todos los casos, `Op` puede ser un entero o una etiqueta):

- Grupo de NOP:

`NOP`	No operación.

- Grupo de movimientos:

`LDA Op`	Carga en el acumulador el contenido de `Op`.
`STA Op`	Almacena en Op el contenido del acumulador.
`LDSP Op`	Carga en el puntero de pila el contenido de `Op`.
`STSP Op`	Almacena en `Op` el contenido del puntero de pila.
`LDR Op`	Carga en el registro `R` el contenido de `Op`.
`STR Op`	Almacena en `Op` el contenido del registro `R`.
`LDIX Op`	Carga en el registro `IX` el contenido de `Op`.
`STIX Op`	Almacena en `Op` el contenido del registro `IX`.

- Grupo de operaciones aritméticas:

`ADD Op`	Suma al acumulador el contenido de `Op`, y lo almacena en el acumulador.
`SUB Op`	Resta al acumulador el contenido de `Op`, y lo almacena en el acumulador.
`MUL Op`	Multiplica el acumulador por el contenido de `Op`, y lo almacena en el acumulador.
`DIV Op`	Divide el acumulador entre el contenido de `Op`, y lo almacena en el acumulador.
`INC Op`	Incrementa el contenido de `Op` en una unidad.
`DEC Op`	Decrementa el contenido de `Op` en una unidad.
`NEG Op`	Cambia el signo de `Op`.

  - Las operaciones aritméticas alteran el contenido del registro `R`, de la manera que se indicó anteriormente.
  - Si se encuentra una división por cero, se produce un error de ejecución.
  - Si el cociente de una operación de división excede el rango de enteros de 16 bits con signo, se produce un error de ejecución.

- Grupo de operaciones lógicas:

`AND Op`	Efectúa un y lógico entre el contenido del acumulador y `Op`, y almacena el resultado en el acumulador.
`OR Op`	Efectúa un o lógico entre el contenido del acumulador y `Op`, y almacena el resultado en el acumulador.
`NOT Op`	Efectúa la negación lógica de `Op`.

Se considera falso un operando igual a cero y verdadero uno distinto de cero.

- Grupo de manejo de pila:

`PUSH Op`	Apila `Op`, es decir, decrementa el contenido del puntero de pila y almacena `Op` en la dirección a la que apunta `SP`.

`POP Op`	Desapila `Op`, es decir, almacena en `Op` el contenido de la posición a la que apunta el puntero de pila y a continuación incrementa `SP`.

- Grupo de saltos:

`J Op`	Salto a la dirección `Op`, es decir, carga en el contador de programa el contenido de `Op`.
`JZ Op`	Salto a `Op` si el acumulador es igual a 0.
`JNZ Op`	Salto a `Op` si el acumulador es distinto de 0.
`JP Op`	Salto a `Op` si el acumulador es mayor que 0.
`JNP Op`	Salto a `Op` si el acumulador es menor o igual que 0.
`JM Op`	Salto a `Op` si el acumulador es menor que 0.
`JNM Op`	Salto a `Op` si el acumulador es mayor que 0.

- Grupo de llamadas y retornos:

`CALL Op`	Llamada a la dirección `Op`, es decir, apila el valor del contador de programa y carga `CO` con el valor de `Op`.
`RET`	Retorno desde la última llamada, esto es, desapila el valor de `CO`.
`STOP`	Retorno al sistema operativo.

- Grupo de entrada-salida:

`INPUT Op`	Carga en `Op` el código *ASCII* del carácter introducido por teclado.
`WRITE Op`	Escribe en pantalla el carácter cuyo código *ASCII* viene dado por los 8 bits bajos del valor de `Op`.
`ININT Op`	Carga en `Op` un entero introducido por teclado.
`WRINT Op`	Escribe en pantalla el entero cuyo valor contiene `Op`.
`WRSTR Op`	Escribe en pantalla la cadena que empieza en la dirección `Op` y termina con un código nulo (se visualizan caracteres cuyos códigos *ASCII* viene dados por los 8 bits bajos de cada posición).

Pseudoinstrucciones del ensamblador.

- `L: EQU Expresión`
  Asigna a la etiqueta `L` el valor de `Expresión`
- `DC N`
  Define la constante `N`
- `DS N`
  Reserva `N` posiciones de memoria y las llena con códigos nulos
- `DFSTR Cadena`
  Pone `Cadena` en memoria, finalizándola con un código nulo
- `END`
  Señala el final del programa

Representación de operandos.

- Rango de representación [-32768..32767] = [$0-$FFFF]
- Los caracteres se representan entre comillas simples o por su código *ASCII*. Se permiten caracteres especiales según las convenciones del lenguaje *C*.
- Los caracteres especiales no se pueden usar en cadenas entrecomilladas. Por caracteres especiales se entiene aquéllos como `\t` (tabulador), `\0` (nulo), `\r` (retorno de carro) y `\n` (salto de línea), que no son directamente "imprimibles".

Interfaz de Usuario.

El programa se invoca desde la línea de comandos, con la siguiente sintaxis:

```
> ass [opciones] archivo [opciones]
```

Donde *archivo* es el nombre del archivo que contiene el fuente en ensamblador.

Se permiten las siguientes opciones:

/c Emitir el código máquina en un archivo de nombre *codmaq* y la tabla de etiquetas en el archivo *tabetq*.
/d Depuración. El código se ejecuta paso a paso, mostrando el contenido de los registros.

El ensamblador distingue entre mayúsculas y minúsculas, reconociendo las instrucciones en ambos casos.

Realiza el ensamblado de una sola pasada si al final de la misma no quedan etiquetas o expresiones indefinidas (se usa la técnica del "relleno por retroceso", lo que en condiciones normales permite definir todas las etiquetas en una sola pasada). En caso contrario, se realizan nuevas pasadas, hasta que no queden etiquetas o expresiones por definir.

Limitaciones.

- Longitud de un identificador: 24 caracteres.
- Longitud de una línea de programa: 80 caracteres. Esto tiene una implicación muy clara a la hora de definir expresiones y cadenas entrecomilladas.
- Número máximo de etiquetas: 256.

2.5.2. *ENS96*

Como en el caso de *ASS*, la información que se muestra a continuación está extraída del manual de la herramienta [Rodríguez 1997]. En este caso, se trata de una aplicación creada por Raúl Rodríguez en 1996, y la versión sobre la que se hace el estudio es la 2.0.1. El ejecutable, `ens96.exe`, tiene como marca de fecha y hora: 18-07-1997-12:41. Viene acompañado de algunos ficheros de texto con el manual y detalles acerca de las últimas correcciones. También se adjuntan dos programas de prueba, `fact.ens` y `matriz.ens`.

ENS96 nació como una mejora de *ASS*, por lo que no resulta extraño reconocer características similares en ambos programas. Sin embargo, aporta notables logros, no sólo por la máquina virtual que se simula, sino sobre todo en cuanto a la interacción con el usuario, en el que se muestra mucho más rico en funcionalidad a la hora de ejecutar y depurar programas.

Máquina Virtual.

- Consta de un procesador y una memoria de 32 KPalabras de 16 bits (64 Kbytes).
- Maneja enteros con signo de 16 bits, en el rango -32768..32767.
- El procesador consta de 14 registros:
  - **Acumulador** (A).
  - **Contador de Programa** (PC).
  - **Puntero de Pila** (SP).
  - **Registros Índices** (IX, IY).
  - **Registro de Estado** (SR). Almacena los biestables Positivo, Negativo, Cero y Desbordamiento.
  - **Registros de Apoyo** (R1..R8).
- El código fuente se toma de un archivo de texto, y se puede ejecutar y depurar mediante una sencilla interfaz en línea de comandos.

Modos de direccionamiento.

- **Inmediato** → OP (dec), o bien
#OP (dec), o bien
$OP (hex)
- **Registro** → .OP
- **Directo** → /OP
- **Indirecto** → [.OP]
- **Relativo** → #desp[.INDICE] (dec), o bien
$desp[.INDICE] (hex)

Juego de Instrucciones.

El juego de instrucciones es el siguiente:

- Grupo de NOP:

`NOP`	No operación.

- Grupo de movimientos:

`LD Reg, Op2`	Carga en el registro `Reg` el contenido de `Op2`.
`ST Op1, Op2`	Carga en `Op1` el contenido de `Op2`.
`MOVE Reg1, Reg2`	Mueve el registro `Reg1` al registro `Reg2`.

- Grupo de operaciones aritméticas:

`ADD Op1, Op2`	Suma los dos operandos.
`SUB Op1, Op2`	Resta los dos operandos (`Op1` - `Op2`).
`MUL Op1, Op2`	Multiplica los dos operandos.
`DIV Op1, Op2`	Divide los dos operandos (`Op1` / `Op2`).
`MOD Op1, Op2`	Módulo de la división de los dos operandos (`Op1` % `Op2`).
`INC Op`	Incrementa `Op` en una unidad.
`DEC Op`	Decrementa `Op` en una unidad.

`NEG Op`	Cambia el signo de `Op`.

- El resultado de las operaciones binarias se almacena en el registro Acumulador.
- Las operaciones aritméticas modifican los Biestables de Estado, Positivo, Negativo, Cero y Desbordamiento en función del resultado de la operación.

- Grupo de operaciones lógicas:

`AND Op1, Op2`	Y lógico de los dos operandos.
`OR Op1, Op2`	O lógico de los dos operandos.
`XOR Op1, Op2`	O exclusivo lógico de los dos operandos.
`NOT Op`	Negación lógica de `Op`.

  - Operan directamente con cada uno de los bits de los operandos. El resultado de las operaciones binarias se almacena en el registro Acumulador.
  - Las operaciones lógicas modifican los Biestables de Estado, Positivo, Negativo, Cero y Desbordamiento en función del resultado de la operación.

- Grupo de comparaciones:

`CMP Op1, Op2`	Compara ambos operandos, es decir, efectúa la resta `Op1` – `Op2` pero sin almacenar el resultado, sólo modifica los biestables de estado, Positivo, Negativo, Cero y Desbordamiento, en función del resultado de la resta.

- Grupo de manejo de pila:

`PUSH Op`	Introduce un dato en la pila (por defecto con post-decremento).
`POP Op`	Extrae un dato de la pila (por defecto con pre-incremento).

- Grupo de saltos:

`J Op`	Salto incondicional a `Op`.
`JZ Op`	Salto si cero a `Op`.
`JNZ Op`	Salto si distinto de cero a `Op`.
`JP Op`	Salto si positivo a `Op`.
`JM Op`	Salto si negativo a `Op`.
`JNM Op`	Salto si no negativo a `Op`.
`JNP Op`	Salto si no positivo a `Op`.
`JOV Op`	Salto si desbordamiento a `Op`.
`JNOV Op`	Salto si no desbordamiento a `Op`.

  Los saltos condicionales se basan en los Biestables de Estado.

- Grupo de llamadas y retornos:

`CALL Op`	Salva la posición actual en la pila y salta a la dirección indicada por el operando `Op`.

`RET`	Recupera la posición de retorno de la pila y salta a dicha dirección.
`STOP`	Detiene la ejecución.

- Grupo de entrada-salida:

`INPUT Op`	Lee un carácter.
`ININT Op`	Lee un entero.
`WRITE Op`	Escribe un carácter.
`WRINT Op`	Escribe un entero.
`WRSTR Op`	Escribe una cadena de caracteres.

Pseudoinstrucciones del ensamblador.

- `L: EQU Expresión`
  Asigna a la etiqueta `L` el valor de `Expresión`.
- `DC N`
  Define la constante `N`.
- `DS N`
  Reserva `N` posiciones de memoria y las llena con códigos nulos.
- `DFSTR Cadena`
  Pone `Cadena` en memoria, finalizándola con un código nulo.
- `END`
  Señala el final del programa.

Representación de operandos.

- Rango de representación [-32768..32767] = [$0-$FFFF].
- Los enteros podrán representarse en decimal, o bien en hexadecimal precedidos del carácter '$'.
- Los caracteres se representan entre comillas simples o por su código *ASCII*. Se permiten caracteres especiales según las convenciones del lenguaje *C*.

Interfaz de Usuario.

El programa se invoca desde la línea de comandos, con la siguiente sintaxis:

```
> ens96
```

El ensamblador distingue entre mayúsculas y minúsculas, reconociendo las instrucciones en ambos casos.

Realiza el ensamblado de dos pasadas. En la segunda de ellas resuelve el valor de las etiquetas que constituyeran referencias adelantadas en el código.

La interfaz entre el usuario y la aplicación se realiza mediante línea de comandos. Como cualquier aplicación de consola, se indica que está preparada para recibir las órdenes del usuario mostrando un *prompt* (indicador) en pantalla.

Además, dicho indicador mantendrá un puntero a una posición de memoria que podrá utilizarse para ir recorriendo el código. Si bien su uso no es imprescindible, permite al

usuario ahorrarse información a la hora de teclear los comandos, ya que la ausencia de parámetros hace referencia a este puntero. Por ejemplo, si el puntero tiene el valor 100h y el usuario introduce el comando 'D', la herramienta desensamblará 15 instrucciones a partir del puntero (dirección 100h), y el puntero tomará el nuevo valor 10fh. Si vuelve a teclear 'D', la herramienta volverá a desensamblar 15 instrucciones, esta vez a partir de la dirección 10fh, y el puntero tomará el nuevo valor 11eh.

Cada comando introducido se interpretará de la siguiente manera:

- Si el comando tiene una sola letra se considerará como un comando interno del compilador, y que podrá ser cualquiera de los indicados a continuación.
- Si tiene más de una letra se considerará como comando *MS-DOS* y se ejecutará como tal.

Los argumentos de los comandos de *ENS96* que indican posiciones o valores podrán ir en hexadecimal (precedido del carácter '$') o en decimal.

Los comandos de *ENS96* son los siguientes:

- **'P'** Función: Posicionar puntero.
  Argumentos: dirección de memoria.
  Se asigna al *prompt* (indicador) de la interfaz el valor de la dirección de memoria indicada.
- **'D'** Función: Desensamblar.
  Saca por pantalla el código máquina almacenado en la memoria en modo ensamblador.
  Argumentos: Dos posibles argumentos:
  - Primero: posición inicial a partir de la que se va a desensamblar.
  - Segundo: hasta donde se va a desensamblar.
- **'V'** Función: Volcado de Memoria.
  Saca por pantalla un volcado hexadecimal y *ASCII* de la memoria.
  Argumentos: Dos posibles argumentos de inicio y fin.
- **'R'** Función: Ver y modificar registros.
  Argumentos: Registro y valor de registro.
  Comentario: Si sólo se indica el registro, se muestra el valor de dicho registro. Si se indica el valor, se modifica el valor del registro.
- **'K'** Función: Modificar el tipo de pila.
  Por defecto el SP apuntará a la cima libre de la pila, o sea, introduce datos en la pila con post-decremento y los saca con pre-incremento.
  Comentario: El otro modo es el inverso, SP apuntará al ultimo elemento introducido en la pila. Meterá datos con pre-decremento y los sacará con post-incremento.
- **'B'** Función: Coloca una interrupción de ejecución. Solo se permitirá una.
  Argumentos: Posición de la interrupción. Sin argumento, conecta y desconecta la interrupción.
- **'E'** Función: Ejecutar.
  Ejecuta a partir del valor del PC con el estado de la máquina actual.
  Comentario: Se debe asignar valor al contador de programa antes de ejecutar. Se puede parar la ejecución con la tecla ESCAPE.
- **'A'** Función: Ejecutar paso a paso (activo/inactivo).

Comentario: Por defecto no para.

- **'Z'** Función: Salida de ejecución (activo/inactivo).
  Comentario: Muestra qué instrucciones se van ejecutando. Por defecto no las muestra.
- **'M'** Función: Ver y modificar memoria.
  Argumentos: Dirección de memoria y valor para esa dirección de memoria
  Comentario: Si sólo se indica la dirección, se muestra el contenido de la misma. Si se indica además el valor, se actualiza la dirección con dicho valor.
- **'C'** Función: Compilar fichero.
  Argumentos: Nombre del fichero a compilar.
  Comentario: Realizará la compilación en dos pasadas para poder valorar todas las etiquetas. La compilación léxica se hará en la primera pasada, así como parte de la sintáctica y la semántica. La segunda pasada terminará de dar valores a las etiquetas y generará el código final. Este código se almacenará a partir de la posición 0 de memoria.
- **'T'** Función: Ver Tabla de Etiquetas.
  Tras compilar un fichero con código ensamblador la tabla de etiquetas queda en memoria. Con este comando se puede visualizar.
  Comentario: Cada compilación distinta creará una tabla nueva.
- **'F'** Función: Ver contenido fichero *ASCII*.
  Argumentos: El nombre del fichero.
- **'L'** Función: Cargar desde un fichero el contenido de la memoria. Se presupone que el contenido de la memoria fue almacenado previamente en dicho fichero.
  Argumentos: El nombre del fichero.
- **'S'** Función: Salvar en un fichero la memoria actual.
  Argumentos: El nombre del fichero.
  Comentario: No salva el estado de la máquina (los registros).
- **'H'** Función: Ver fichero de ayuda.
- **'X'** Función: Salir a un *shell* del *DOS*.
- **'Q'** Función: Salir del programa.

Limitaciones.

- Longitud de un identificador: 20 caracteres

2.5.3. Comentario acerca de las herramientas existentes

Como se puede observar tras la lectura de las características de ambas herramientas, se deduce que *ENS96* fue concebida como una mejora de *ASS*. De hecho, el propio autor cita en la documentación de la aplicación que "todo lo que hace *ASS* lo realiza también, y de manera más cómoda, *ENS96*. Además, no utilizan la misma sintaxis para el código ensamblador, ya que la de *ASS* no es estándar" [Rodríguez 1997].

De este breve estudio se extrae una lista de ventajas, que se tomarán como base para desarrollar la nueva herramienta y una serie de desventajas que se tratarán de eliminar o, al menos, corregir en la medida de lo posible.

Ventajas de las herramientas actuales:

- *ENS96* supera a su predecesor *ASS* adoptando la sintaxis del estándar *IEEE 694* y un subconjunto de instrucciones del mismo. Aún así, no todas las instrucciones se ajustan a la nomenclatura del estándar.
- *ENS96* permite su manejo mediante una sencilla interfaz a base de comandos (suficiente para la época en que fue diseñado).
- *ENS96* posee gran capacidad de seguimiento de los programas simulados, así como su depuración y el acceso a los componentes de la arquitectura simulada.
- La arquitectura simulada por *ENS96* añade a la propuesta por *ASS* varios registros de propósito general que facilitan el trabajo del programador en ensamblador.
- *ENS96* permite reubicar el código generado mediante la pseudoinstrucción `ORG`.

Desventajas de las herramientas actuales:

- La sintaxis del lenguaje ensamblador propuesto por *ASS* no es estándar.
- *ASS* posee una interactividad casi nula con el usuario, que se limita a invocar a la aplicación y esperar su respuesta.
- *ASS* sólo permite identificadores de 24 caracteres, líneas de código fuente de hasta 80 caracteres y un máximo de 256 etiquetas. A su vez, *ENS96* limita la longitud de los identificadores a 20 caracteres.
- *ENS96* sólo direcciona 15 bits de posiciones de memoria, lo cual rompe la homogeneidad del modelo simulado, ya que el ancho de palabra de los registros y las posiciones de memoria es de 16 bits.
- Se echan en falta en la arquitectura de *ENS96* algunos biestables de estado, como pueden ser los de acarreo o paridad.
- En *ENS96*, el formato de instrucción es único y, por tanto, se desaprovecha demasiado espacio.
- El juego de instrucciones de *ENS96* no es demasiado homogéneo en cuanto a las combinaciones de modos de direccionamiento permitidas para los operandos. Sería interesante permitir todas las combinaciones posibles, siempre que tengan sentido.
- En el juego de instrucciones de *ENS96*, la combinación de instrucciones de salto condicional que se propone es un poco críptica, siendo deseable tener una pareja de instrucciones de salto condicional para cada biestable de estado (una que se ejecute para el valor 0 y otra para el valor 1).
- El juego de instrucciones de *ENS96* posee una instrucción para escribir cadenas de caracteres, pero no para leerlas.
- En los modos de direccionamiento de *ENS96*, se echa en falta el direccionamiento relativo al contador de programa. Si bien a la hora de diseñar el formato de instrucción y optimizar espacio no es relevante, sí lo es a nivel conceptual.
- La interfaz de usuario de ambos programas se ha quedado obsoleta en la actualidad, prefiriéndose una interfaz gráfica basada en un entorno de ventanas, en la que el usuario pueda acceder fácilmente, tanto a las funciones de la aplicación, como a los componentes del simulador (memoria, registros, etc.). No obstante, esto no implica desterrar la posible utilidad de una interfaz de comandos, para usuarios que no empleen sistemas operativos con entorno gráfico.
- *ENS96* permite un único punto de ruptura durante la ejecución.
- No existe una versión para *Linux* de *ENS96* (aunque seguramente bastaría con recompilar la versión para *MS-DOS* si se dispusiera del código fuente).

3. Solución

Este es el capítulo más importante del Trabajo de Fin de Carrera. En él se expone el proceso software que se ha seguido para la construcción de la aplicación *ENS2001*, cuya motivación se ha visto en los dos capítulos anteriores.

3.1. Especificación de Requisitos

3.1.1. Introducción

En esta sección se realiza una breve introducción al documento de Especificación de Requisitos Software (ERS). Este documento ha sido redactado siguiendo el estándar IEEE 830-1998 [IEEE 1998].

3.1.1.1. Propósito

El propósito de este documento ERS consiste en definir las características de la aplicación que se va a construir, el ámbito en que se va a desarrollar, las funcionalidades que debe proporcionar a los usuarios y las restricciones que debe cumplir a la hora de ser diseñada.

Por tratarse de un Trabajo Fin de Carrera, en este caso el documento ERS no constituye el habitual contrato entre un cliente y un equipo de desarrollo. Sí que indica las directrices a las que se tiene que ajustar la persona que está realizando el proyecto, y sirve como indicación al tribunal que evalúe el trabajo para comprobar que se ajuste a unos objetivos prestablecidos. Como tal, este documento forma parte de la memoria correspondiente al TFC.

3.1.1.2. Ámbito del Sistema

El sistema que se va a construir se trata de un ensamblador-simulador de estándar *IEEE 694*. Se le ha dado el nombre de *ENS2001*, por su funcionalidad y por el año en que comenzó el proyecto.

ENS2001 es un sistema que permite al usuario generar código máquina a partir de programas escritos en lenguaje ensamblador de acuerdo al estándar y simular su ejecución, en el entorno de un ordenador personal.

3.1.1.3. Definiciones, Acrónimos y Abreviaturas

ERS. Especificación de Requisitos Software.

IEEE. *Institute of Electrical and Electronic Engineers* (Instituto de Ingenieros de Electricidad y Electrónica).

TFC. Trabajo Fin de Carrera.

3.1.1.4. Referencias

Las Referencias necesarias se encuentran en el capítulo 6 (Bibliografía).

3.1.1.5. Visión General del Documento

Este documento se divide en tres secciones bien diferenciadas:

- En la primera de ellas, la introducción, se expone el popósito y alcance del documento ERS, el ámbito del sistema que se va a construir y se introducen los conceptos y referencias que se vayan a utilizar a lo largo del documento ERS.
- En la segunda, se efectúa una descripción general del sistema que se va a construir, resumiendo a grandes rasgos las funciones que ofrece, las características de sus usuarios y las restricciones que se imponen sobre los desarrolladores del producto.
- Por último, en la tercera de ellas se especifican los requisitos concretos que debe cumplir el sistema, de forma que el equipo de desarrollo pueda diseñar y construir un sistema que satisfaga dichos requisitos. Se hablará de interfaces externas, funcionalidad, requisitos de rendimiento, restricciones de diseño y atributos del sistema.

3.1.2. Descripción General

3.1.2.1. Perspectiva del Producto

El sistema que se va a construir funciona con independencia de otras aplicaciones. Únicamente se valdrá de los servicios del sistema operativo sobre el que vaya a ejecutarse. Tan sólo será necesario el empleo de un programa externo que genere los ficheros de texto escritos en lenguaje ensamblador, pero su elección será a discrección del usuario.

3.1.2.2. Funciones del Producto

El sistema consta de dos partes diferenciadas:

- Por una parte el ensamblador, que acepta ficheros de texto y genera código máquina comprensible por el simulador. Ofrece la funcionalidad característica de un compilador, recibir su entrada de un fichero de texto y devolver una lista de errores o el código generado.
- Por otra parte el simulador, que proporciona funciones de ejecución, depuración y acceso a los componentes del sistema simulado.

3.1.2.3. Características de los Usuarios

En principio la aplicación se dirije a los estudiantes de Compiladores que requieran de un entorno simulado para probar sus prácticas. No obstante, también puede servir como herramienta de aprendizaje a cualquier estudiante de ensamblador o como introducción a la arquitectura de computadores. En cualquier caso, su perfil será eminentemente técnico, correspondiente a un estudiante de informática o personas que estén familiarizadas con el uso de un ordenador personal.

3.1.2.4. Restricciones

La aplicación debe ser desarrollada en un lenguaje que permita su portabilidad al mayor número de plataformas posible. Se recomienda el uso de *C++* estándar.

3.1.2.5. Suposiciones y Dependencias

Los requisitos se mantendrán salvo que las necesidades de desarrollo y evaluación de las prácticas de la asignatura de Compiladores sufran un cambio drástico.

3.1.2.6. Requisitos Futuros

Tras un período de uso se tratará con los usuarios la conveniencia de añadir alguna nueva funcionalidad a la herramienta o modificar alguna de las ya existentes de manera que se simplifique el trabajo con la misma.

3.1.3. Requisitos Específicos

En este apartado se incluye una relación de los requisitos con un nivel de detalle tal que permita al equipo de desarrollo diseñar un sistema que los satisfaga, y que permita al equipo de pruebas planificar y realizar las pruebas que verifiquen si el sistema satisface, o no, los requisitos.

3.1.3.1. Interfaces Externas

ERS-REQ01. Interfaces de usuario.

Se va a construir una misma herramienta con dos interfaces de usuario distintas.

La primera de ellas se trata de una interfaz en modo texto, que será ejecutable en la consola de *Windows* (32 bits) y en consolas *Linux*.

La segunda de ellas se trata de una interfaz gráfica basada en entorno *Windows* y ejecutable en todas sus versiones a partir de *Windows 95*.

3.1.3.2. Funciones

ERS-REQ02. Arquitectura del sistema.

El sistema deberá simular la siguiente arquitectura, basada en la del procesador *Z80*:

- Procesador con ancho de palabra de 16 bits.
- Memoria de 64 Kpalabras (de 16 bits).
- Banco de Registros:
    - PC (Contador de Programa)
    - SP (Puntero de Pila)
    - SR (Registro de Estado)
    - IX, IY (Registros Índices)
    - A (Acumulador)
    - R0..R9 (Registros de Propósito General)
- Biestables de estado (contenidos en el registro de estado SR):
    - Z (cero)
    - C (acarreo)
    - V (desbordamiento)
    - P (paridad)
    - S (signo)
    - H (fin de programa)

- Entrada/Salida por consola de números enteros de 16 bits y cadenas de caracteres.
- Unidad Aritmético-Lógica que operará sobre enteros con signo de 16 bits.
- Generación de Excepciones:
  - Instrucción no Implementada.
  - División por cero.
  - Final de memoria alcanzado.
  - Contador de programa invade la zona de pila (opcional).
  - Puntero de pila invade la zona de código (opcional).
  - Ejecución detenida por el usuario (salvo en la versión *Linux*).

ERS-REQ03. Instrucciones.

El lenguaje ensamblador que se defina, será análogo al de las herramientas existentes. En un anexo a este documento se especifica el juego de instrucciones completo, que contendrá instrucciones de los siguientes tipos:

- No operación.
- Parar la ejecución.
- Transferencia de Datos.
- Bifurcaciones condicionales e incondicionales.
- Manejo de Pila.
- Aritméticas.
- Comparación.
- Lógicas.
- Entrada/Salida.
- Manejo de Subrutinas.

ERS-REQ04. Operandos.

Cobra especial relevancia que los operandos de las instrucciones permitan el máximo número de modos de direccionamiento posibles, y todas sus combinaciones, siempre que tengan sentido.

ERS-REQ05. Ficheros fuente.

El código fuente de los programas que se van a ensamblar y ejecutar se introducirá en el sistema mediante un fichero de texto plano. El nombre y ubicación de este fichero fuente puede ser cualquiera. El usuario se encargará de proporcionárselo al sistema cuando éste último lo requiera.

ERS-REQ06. Ensamblado.

El código fuente se ensamblará de acuerdo al lenguaje descrito y el resultado del ensamblado se cargará en memoria, siempre que no se hayan producido errores durante el proceso, dejando el simulador preparado para la ejecución cuando el usuario estime oportuno.

ERS-REQ07. Visualización de los errores de ensamblado.

Si durante el proceso de ensamblado se produjeran errores, el sistema informará por pantalla de los mismos, tipo de error y número de línea donde se produjo, proporcionando al usuario la máxima cantidad de información de que disponga el sistema (esto es, devolviendo una pequeña descripción del error, no sólo un código). Obviamente, en caso de error, no se ensamblará nada. Los errores informados no sólo corresponderán a la corrección del programa, sino también a errores de acceso, como no encontrar el fichero fuente, no poder abrirlo, no disponer de memoria para ejecutar la aplicación, etc.

ERS-REQ08. Volcado de memoria a fichero.

El sistema proporcionará la opción de volcar el contenido íntegro de la memoria en un fichero a petición del usuario.

ERS-REQ09. Volcado de memoria desde fichero.

El sistema proporcionará la opción de recuperar el contenido íntegro de la memoria desde un fichero creado previamente por el propio sistema a petición del usuario.

ERS-REQ10. Base de Representación Numérica.

El usuario podrá elegir el formato de representación para los números enteros, que podrá ser en base decimal o hexadecimal.

ERS-REQ11. Consulta de una posición de memoria.

El sistema permitirá visualizar el contenido de una posición de memoria.

ERS-REQ12. Consulta de una zona de memoria.

El sistema permitirá visualizar el contenido de una zona de memoria.

ERS-REQ13. Edición de una posición de memoria.

El sistema permitirá editar el contenido de una posición de memoria.

ERS-REQ14. Reinicia el contenido de la memoria.

El sistema permitirá reiniciar el contenido de toda la memoria (poner todos los valores a cero).

ERS-REQ15. Consulta de la pila del sistema.

El sistema permitirá visualizar el contenido de una zona de la pila, a partir de la posición del puntero de pila.

ERS-REQ16. Consulta del valor de un registro.

El sistema permitirá visualizar el contenido de un registro.

ERS-REQ17. Consulta del Banco de Registros.

El sistema permitirá visualizar el contenido del banco de registros.

ERS-REQ18. Edición del valor de un registro.

El sistema permitirá editar el contenido de un registro.

ERS-REQ19. Reinicio del Banco de Registros.

El sistema permitirá reiniciar el contenido del banco de registros. Al reiniciar el banco de registros, el valor de todos ellos se pondrá a cero, y el del puntero de pila se actualizará a la última posición de memoria, si el funcionamiento de la pila es decreciente, o a la primera posición de memoria después del código, si el funcionamiento es creciente, o bien otro esquema análogo que se estime oportuno si el código no está ubicado a partir de la posición más baja de memoria.

ERS-REQ20. Desensamblado de una zona de memoria.

El sistema permitirá desensamblar el contenido de la memoria a partir de una posición indicada por el usuario.

ERS-REQ21. Dirección de Crecimiento de la Pila.

El usuario podrá optar entre dos opciones de funcionamiento de la pila del simulador, que crezca en sentido ascendente o descendente.

Para facilitar la labor al usuario, la herramienta calculará la posición inicial del puntero de pila basándose en el sentido de crecimiento y la posición del código ejecutable. El código ejecutable puede dejar espacio libre delante y detrás de él, dependiendo de su ubicación en memoria. La herramienta calcula cuál de ambos huecos es mayor. Así, si la pila crece en sentido ascendente, el puntero de pila se colocará al inicio del hueco mayor, mientras que si crece en sentido descendente, el puntero de pila se situará al final del hueco de mayor tamaño.

ERS-REQ22. Definición de puntos de ruptura.

El sistema permitirá establecer y anular puntos de ruptura ante los cuales el procesador virtual detendrá la simulación.

ERS-REQ23. Ejecución Paso a Paso.

Para depurar el programa ensamblado, el usuario podrá optar entre ejecutar el código paso a paso o hasta que se cumpla cualquier otra condición de parada.

ERS-REQ24. Ejecución del código.

El sistema proporcionará una opción de ejecutar (simular) el código.

ERS-REQ25. Condiciones para detener la ejecución del código.

La ejecución del código se detendrá en los siguientes casos:

- El procesador encontró una instrucción de parada y la ejecutó.
- Se produjo una excepción durante la ejecución.
- Se encontró un punto de ruptura en la dirección apuntada por el contador de programa.
- El usuario seleccionó el modo de ejecución paso a paso, con lo cual el procesador se detendrá tras ejecutar cada instrucción.

ERS-REQ26. Consola de Entrada/Salida.

El sistema mostrará una consola para las operaciones de entrada/salida. Dicha consola será la propia consola del sistema cuando se esté trabajando en la versión en línea de comandos, y una ventana independiente cuando se trate de la versión en modo gráfico.

ERS-REQ27. Comprobar si PC invade el espacio de Pila.

Opcionalmente, el sistema comprobará durante la ejecución que el contador de programa invade la zona definida como pila del sistema, lanzando una excepción que detendrá la ejecución.

ERS-REQ28. Comprobar si SP invade el espacio de Código.

Opcionalmente, el sistema comprobará durante la ejecución si el puntero de pila invade la zona definida como código, lanzando una excepción que detendrá la ejecución. Esta característica sólo funcionará cuando se ejecuten programas que han sido previamente ensamblados con la herramienta, y no con aquellos que sean introducidos por otros medios (volcado de memoria, edición manual, etc.), ya que la zona de código no podrá estar correctamente definida.

ERS-REQ29. Detención arbitraria de la simulación.

La herramienta permitirá al usuario detener la simulación en cualquier momento, independientemente del estado en que se encuentre la misma.

ERS-REQ30. Inicializar los registros al ejecutar.

Opcionalmente, el simulador reiniciará el contenido del banco de registros antes de comenzar una nueva ejecución, siempre que la ejecución anterior haya finalizado con una instrucción de parada.

ERS-REQ31. Consulta de las variables de configuración del simulador.

El usuario tendrá la posibilidad de consultar en cualquier momento cuál es la configuración actual del simulador. Se mostrará en pantalla una lista de las variables de configuración y el valor que tienen en ese momento.

ERS-REQ32. Modo de depuración.

El usuario tendrá la posibilidad de ignorar o no durante la simulación los puntos de ruptura definidos previamente, sin eliminar la definición de los mismos.

ERS-REQ33. Consulta de los puntos de ruptura en el código desensamblado.

El usuario podrá ver junto al código fuente los puntos de ruptura incondicionales definidos. No existirá la posibilidad de definir puntos de ruptura condicionales.

ERS-REQ34. Almacenamiento de la salida generada por la herramienta durante la sesión.

La versión gráfica de la herramienta permitirá que el usuario guarde en un fichero de texto el contenido de la sesión de trabajo.

ERS-REQ35. Fin de la ejecución de la herramienta.

El usuario podrá detener la ejecución de la herramienta en cualquier momento.

3.1.3.3. Requisitos de Rendimiento

Se trata de una herramienta monousuario y, teniendo en cuenta la potencia de cálculo de los ordenadores personales actuales y que no se va a simular la arquitectura en tiempo real, no se plantea ninguna restricción en cuanto al rendimiento que se espera obtener.

3.1.3.4. Restricciones de Diseño

El diseño de la herramienta, en las partes en las que se opte por el paradigma orientado a objetos, se efectuará siguiendo la metodología propuesta en [Rumbaugh 1996].

3.1.3.5. Atributos del Sistema

El sistema deberá ser portable a cualquier máquina con el mínimo coste posible. En principio, será portada a sistemas *Windows* y *Linux*.

3.1.4. Apéndice I. Juego de instrucciones

Mnemónico	Comportamiento
ADD	Suma de enteros.
AND	Y lógico.
BC	Bifurcación si acarreo.
BE	Bifurcación si paridad par.
BN	Bifurcación si negativo.
BNC	Bifurcación si no acarreo.
BNV	Bifurcación si no desbordamiento.
BNZ	Bifurcación si distinto de cero.
BO	Bifurcación si paridad impar.
BP	Bifurcación si positivo.
BR	Bifurcación incondicional.
BV	Bifurcación si desbordamiento.
BZ	Bifurcación si cero.
CALL	Llamada a subrutina.
CMP	Comparar dos enteros.
DEC	Decrementar.
DIV	Cociente de enteros.
HALT	Detener la ejecución de la máquina.
INC	Incrementar.
INCHAR	Lectura de carácter.
ININT	Lectura de entero.
INSTR	Lectura de cadena.

MOD	Resto de división de enteros.
MOVE	Copiar operando origen en operando destino.
MUL	Producto de enteros.
NEG	Cambiar de signo.
NOP	Instrucción de no operación.
NOT	Negación lógica.
OR	O lógico.
POP	Sacar de la pila.
PUSH	Poner en la pila.
RET	Retorno de subrutina.
SUB	Resta de enteros.
WRCHAR	Escritura de carácter.
WRINT	Escritura de entero.
WRSTR	Escritura de cadena.
XOR	O exclusivo lógico.

3.2. Análisis del Sistema

Se ha separado en tres partes el análisis del sistema *ENS2001*. Por una parte, se analizará el lenguaje ensamblador que va a usar la herramienta. Por otro, el módulo encargado de ensamblar el código fuente. Y, por último, el motor de simulación.

3.2.1. Análisis del Lenguaje Ensamblador

El lenguaje ensamblador diseñado para ser ensamblado y simulado por *ENS2001* se trata de un subconjunto del estándar *IEEE 694*, descrito en [De Miguel 1996], al que se han añadido algunas instrucciones de entrada/salida por consola especiales, de acuerdo a lo expuesto en la Especificación de Requisitos.

Se definen los siguientes conceptos:

- **Estructura del código fuente**. Cómo se construyen los programas en lenguaje ensamblador.
- **Modos de Direccionamiento**. Cómo se hace referencia a los operandos de las instrucciones del lenguaje.
- **Juego de Instrucciones**. Cuáles son las operaciones del lenguaje y cuál es su comportamiento.
- **Formatos de Instrucción**. Cómo se codifican las instrucciones del lenguaje dentro de la memoria del sistema.
- **Macroinstrucciones**. Cuáles son las instrucciones que proporciona el ensamblador para facilitar la escritura de código fuente.

3.2.1.1. Código Fuente

Los programas se introducirán en la aplicación a través de ficheros de código fuente. El fichero fuente es un fichero de texto plano que se compone de una sucesión de líneas, de acuerdo al siguiente formato:

```
[ [Etiqueta ':'] Instrucción] [';' Comentario]
```

No se limita el número de caracteres por línea (salvo restricciones impuestas por el sistema operativo o la memoria disponible).

De este formato se deduce que:

- Se permiten líneas en blanco.
- Se permiten líneas de comentario.
- Cada instrucción en sí ocupará una única línea.
- La longitud de las etiquetas no está limitada.

Adicionalmente, se permitirá que se inserten comentarios y líneas en blanco entre la etiqueta y la instrucción, en aras de aumentar la legibilidad del código.

El formato general que comparten todas las instrucciones es el siguiente:

```
Mnemónico [Operando1 [',' Operando2]]
```

Como se detalla más adelante, en cuanto al número de operandos existen tres grupos de instrucciones: sin operandos, con un operando y con dos operandos. Por tanto, aunque en general sólo el campo Mnemónico es obligatorio, en realidad la presencia de ninguno, uno o dos operandos depende del tipo de instrucción. Por lo tanto se podría desdoblar el formato general en los tres siguientes:

```
Mnemónico0
Mnemónico1 Operando1
Mnemónico2 Operando1 ',' Operando2
```

El ensamblador será sensible a mayúsculas/minúsculas. No obstante, se reconocerán tanto las instrucciones como los identificadores de registros de ambas formas (pero no con algunas letras en mayúsculas y otras en minúsculas).

3.2.1.2. Modos de Direccionamiento

Antes de pasar a enumerar el juego de instrucciones, sería conveniente estudiar los modos de direccionamiento que van a estar permitidos para los operandos. Según la Especificación de Requisitos, debería ser posible utilizar el máximo número de ellos, y todas las combinaciones posibles (siempre que tengan sentido). La información los modos de direccionamiento propuestos en el estándar *IEEE 694* ha sido extraída de [De Miguel 1996].

- **Direccionamiento inmediato**.

  El direccionamiento es inmediato cuando el valor final del operando está explícito en la instrucción. Por tanto, dadas las características de la máquina virtual, en la que el tamaño de palabra es de 16 bits, serán necesarios dichos 16 bits para representar internamente este tipo de direccionamiento.

  Para indicar este modo de direccionamiento, se empleará el carácter '#' (almohadilla) precediendo al operando, que en este caso indicará un valor entero de 16 bits en complemento a 2.

  Se pueden introducir valores decimales con signo, en el intervalo comprendido entre –32768 y 32767, o bien sin signo en el rango 0..65535. También se permite introducir valores hexadecimales, expresados en complemento a 2, precedidos por el prefijo '0x', en el rango 0h..FFFFh.

Ejemplos:

- La instrucción de carga `MOVE #1000,.R1`, transfiere el valor 1000d al registro R1 (Figura 3.2.1).

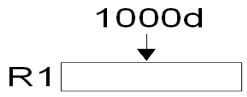


Figura 3.2.1. Direccionamiento Inmediato (`MOVE`)

- La instrucción de suma `ADD .R3,#0x00FF` suma el contenido de R3 con el valor FFh y lo almacena en el acumulador (Figura 3.2.2).

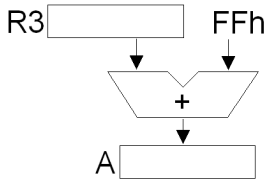


Figura 3.2.2. Direccionamiento Inmediato (`ADD`)

En la documentación se hace referencia a este tipo de direccionamiento, para abreviar, como *inmediato*.

- **Direccionamiento Directo a Registro**.

  El direccionamiento es directo a registro cuando expresa el nombre del registro donde se encuentra almacenado el objeto. Dado el banco de registros expuesto para la máquina virtual, serán necesarios 4 bits para representar internamente este tipo de direccionamiento.

  Para indicar este modo de direccionamiento, se empleará el carácter '.' (punto) precediendo al operando, que en este caso será el nombre de un registro del banco de registros.

  Por ejemplo, en la instrucción de suma `ADD .R3, #0xFF`, el primer operando emplea un direccionamiento directo a registro, ya que se está indicando el nombre del registro donde se encuentra almacenado. Dicha instrucción suma al contenido del registro R3 el valor inmediato FFh, y lo almacena en el acumulador, como se observa en la Figura 3.2.2.

  En la documentación se hace referencia a este tipo de direccionamiento, para abreviar, como *registro*.

- **Direccionamiento Directo a Memoria**.

  El direccionamiento es directo a memoria cuando se expresa la dirección absoluta de memoria donde se encuentra almacenado el operando. Como la memoria se compone de 64 KPalabras, serán necesarios 16 bits para representar internamente este tipo de direccionamiento.

  Para indicar este modo de direccionamiento, se empleará el carácter '/' (barra) precediendo al operando, que en este caso indicará una dirección de memoria.

Como en el caso del direccionamiento inmediato, se pueden introducir valores decimales con signo, en el intervalo –32768..32767, o bien sin signo en el rango 0..65535. También son válidos los valores hexadecimales, precedidos por el prefijo '0x', en el rango 0000..FFFFh. En el caso de que el usuario introduzca valores decimales con signo, se traducirán a la dirección que se correspondería con su representación sin signo, por ejemplo, -1 se corresponde con 65535, -2 con 65534, y así sucesivamente.

Por ejemplo, para la instrucción de incremento `INC /0x1000`, el resultado será que el valor contenido en la posición de memoria 1000h se habrá incrementado en una unidad (Figura 3.2.3).

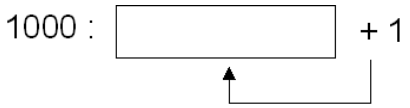


Figura 3.2.3. Direccionamiento Directo a Memoria

En la documentación se hace referencia a este tipo de direccionamiento, para abreviar, como *memoria*.

Otro modo de direccionamiento posible sería el **Directo Absoluto a Página Base**, pero no se va a implementar, ya que en el modelo que simula *ENS2001*, la memoria no está paginada, sino que consta de un único bloque direccionable en su totalidad.

- **Direccionamiento Relativo a Registro Índice**.

  El direccionamiento es relativo a un registro índice cuando la instrucción contiene un desplazamiento sobre una dirección marcada por un puntero. La dirección del operando, por tanto, se calcula sumando el desplazamiento al puntero de referencia.

  Para indicar este modo de direccionamiento, se empleará el carácter '#' (almohadilla) precediendo al operando, que en este caso indicará un desplazamiento (entero de 8 bits en complemento a 2), y detrás, en notación de indirección (entre corchetes), el nombre del registro índice. El valor del desplazamiento se puede escribir bien en base decimal, bien en base hexadecimal, siguiendo las mismas consideraciones que en los casos anteriores de direccionamiento inmediato y directo a memoria.

  En este caso, el sistema simulado consta de dos registros IX e IY que actúan como índices, permitiéndose desplazamientos restringidos a enteros de 8 bits en complemento a 2. Por tanto, para representar internamente este tipo de direccionamiento, se requieren 9 bits, 8 para el desplazamiento y uno adicional para indicar el registro índice. No obstante, para homogeneizar con el resto de modos de direccionamiento, se considerará que son dos modos distintos los relativos a IX y a IY, así sólo serán necesarios los 8 bits del desplazamiento.

  Ejemplos:

  - En la instrucción de carga `MOVE #6[.IX],.R1`, en el registro R1 se carga el contenido de la posición de memoria apuntada por el índice IX más 6 posiciones (Figura 3.2.4).

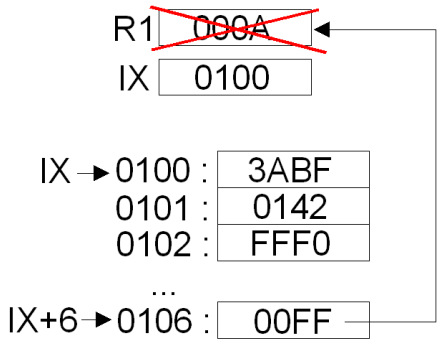


Figura 3.2.4. Direccionamiento Relativo a Índice

- Las siguientes instrucciones son análogas: `DEC #-1[.IY]`; `DEC #0xFF[.IY]`; `DEC #255[.IY]`, debido a la equivalencia entre las representaciones decimal y hexadecimal en complemento a 2.

En la documentación se hace referencia a este tipo de direccionamiento, para abreviar, como *relativo*.

- **Direccionamiento Relativo a Contador de Programa**.

Se trata de un caso particular de direccionamiento relativo. El registro puntero es el contador de programa y se indica el desplazamiento (restringido a un entero de 8 bits en complemento a 2). Por tanto, este modo de direccionamiento requiere 8 bits para su representación interna.

Así, la dirección del objeto, en este caso la próxima instrucción que se va a ejecutar, se obtiene sumando el desplazamiento al contador de programa (PC).

Para indicar este modo de direccionamiento, se empleará el carácter '$' (dólar) precediendo al operando, que en este caso indicará un desplazamiento (entero de 8 bits en complemento a 2). El formato del desplazamiento es idéntico al caso anterior (direccionamiento relativo a registro índice).

Por ejemplo, en la instrucción de salto incondicional `br $3`, la siguiente instrucción que se debe ejecutar se encuentra en la posición de memoria a la que apunte el contador de programa más 3 posiciones. Hay que recordar que el contador de programa apunta siempre a la siguiente instrucción a la que se está ejecutando. Por tanto, en el ejemplo presentado en la Figura 3.2.5, la siguiente instrucción que se ejecutará será `INC .R1`.

```
0000:   BR $3
0001:   BR /bucle
0003:   HALT
0004:   INC .R1
```

Figura 3.2.5. Direccionamiento Relativo a PC

En las instrucciones de salto es muy frecuente el empleo de etiquetas, tanto si el direccionamiento es directo a memoria como relativo a contador de programa. No obstante, en el caso de direccionamiento relativo, la etiqueta no se traduce por su valor, sino que se calcula el desplazamiento entre la posición a la que apuntaría PC

en el caso de seguir el flujo de ejecución y la posición que ocupa la etiqueta. Por ello, si se realiza un direccionamiento relativo a contador de programa hacia una etiqueta que está alejada más de 128 posiciones de memoria (127 hacia adelante, 128 hacia atrás), el ensamblador dará un error en tiempo de compilación, ya que no puede calcular un desplazamiento válido (debe representarlo únicamente con 8 bits).

En la documentación se hace referencia a este tipo de direccionamiento como *relativo a contador de programa* o *relativo a PC*.

- **Direccionamiento Relativo a Pila**.

  El direccionamiento a pila es un caso particular de direccionamiento relativo, muy empleado en los microprocesadores y minicomputadores. La máquina dispone de un registro SP, que realiza la función de puntero de la pila. Este registro contiene la dirección de la posición de memoria principal que actúa de cabecera de pila. La pila puede crecer según direcciones de memoria crecientes o decrecientes. Para el caso de direcciones crecientes, si el puntero SP contiene la cantidad 3721, por ejemplo, y se introduce un nuevo elemento en la pila, SP pasará a tener la dirección 3722. Si, por el contrario, se elimina un elemento deberá pasar a 3720. Los direccionamientos requeridos son:

  - Para insertar un nuevo elemento, se realiza un direccionamiento relativo al registro SP con preautoincremento.
  - Para extraer un elemento, se debe hacer postautodecremento.

  El direccionamiento a pila permite instrucciones muy compactas, puesto que, si sólo se dispone de un único registro SP (como en el caso de *ENS2001*) las instrucciones de manejo de pila (`PUSH` y `POP`) no requiere ninguna información de dirección.

- **Direccionamiento Indirecto**.

  Se trata de un direccionamiento directo, pero del que no se obtiene el operando, sino la dirección donde se encuentra el operando. Dicha dirección se encuentra almacenada en un registro. Por tanto, se requiere un acceso adicional a memoria para recuperar el objeto. Así las cosas, a la hora de representar internamente este modo de direccionamiento, serán necesarios 4 bits para indicar cuál es el registro a partir del que se recupera la dirección de memoria destino.

  Para indicar este modo de direccionamiento, se encierra entre corchetes el identificador del registro sobre el que se efectúa la indirección, precedido éste por el carácter '.' (punto).

  Por ejemplo, la instrucción de decremento `DEC [.R3]` decrementará en una unidad el contenido de la posición de memoria contenida en el registro R3 (Figura 3.2.6).

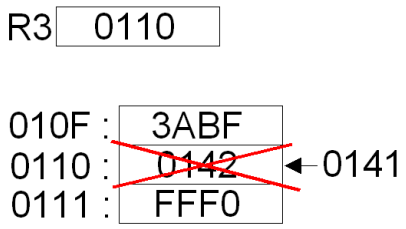


Figura 3.2.6. Direccionamiento Indirecto

En la documentación se hace referencia a este tipo de direccionamiento, para abreviar, como *indirecto*.

- **Direccionamiento Implícito**.

  En el direccionamiento implícito, la instrucción no contiene información sobre la ubicación del objeto, porque éste está en un lugar predeterminado (registro o posición de memoria).

  En el caso del lenguaje definido para *ENS2001*, se va a emplear en las instrucciones aritméticas y lógicas, en las que el resultado siempre se almacena en el acumulador (A), en instrucciones como `CALL` y `RET`, cuyo destino es el contador de programa (PC), y en la instrucción `HALT`, que activa un biestable prefijado. Por lo tanto, estas instrucciones no necesitan codificar información adicional acerca de los operandos con los que trabajan.

3.2.1.3. Juego de Instrucciones

A continuación se describe el juego de instrucciones completo de *ENS2001*. Consta de 37 instrucciones, que se agrupan en los siguientes tipos según su funcionalidad:

- Instrucciones de Transferencia de Datos.
  MOVE, PUSH, POP.
- Instrucciones Aritméticas (aritmética entera en complemento a 2).
  ADD, SUB, MUL, DIV, MOD, INC, DEC, NEG.
- Instrucciones de Comparación.
  CMP.
- Instrucciones Lógicas.
  AND, OR, XOR, NOT.
- Instrucciones de Bifurcación en la Ejecución.
  BR, BZ, BNZ, BP, BN, BV, BNV, BC, BNC, BE, BO.
- Instrucciones de Manejo de Subrutinas.
  CALL, RET.
- Instrucciones de Entrada/Salida.
  INCHAR, ININT, INSTR, WRCHAR, WRINT, WRSTR.
- Otras.
  NOP, HALT.

Antes de pasar a explicarlas, es preciso comentar que las únicas instrucciones que modifican el contenido de los biestables de estado son las instrucciones aritméticas y la instrucción de comparación, que lo hacen de la siguiente forma:

- **Z**. Si el resultado de la operación es 0, se activa. En caso contrario, se desactiva.
- **C**. Si el resultado excede los 16 bits de longitud, se activa. En caso contrario, se desactiva.
- **S**. Representa el signo del resultado de la operación. Indica el valor del bit más significativo (como se está trabajando en complemento a 2, 0 para valores positivos y 1 para valores negativos).
- **V**. Si el resultado de la operación excede el rango de representación de los enteros de 16 bits en complemento a 2 (-32768..32767), se activa. En caso contrario, se desactiva.
- **P**. Si el número de bits a 1 del resultado de la operación es par, entonces P=0. En otro caso, vale 1 (hace un XOR de todos los bits del resultado).

En el momento de ejecutar una instrucción, el contador de programa (PC) apunta a la dirección de memoria donde se ubica la siguiente instrucción de la secuencia. Este funcionamiento debe tenerse en cuenta especialmente a la hora de emplear operandos con direccionamiento relativo a PC o cualquier operación en la que este registo esté implicado.

Este es el listado exhaustivo del juego de instrucciones. Se indica el mnemónico (nombre de la instrucción), formato, código de operación, número de operandos necesarios, modos de direccionamiento permitidos y comportamiento (explicado en lenguaje natural).

Instrucción	NOP
Descripción	Instrucción de no operación.
Formato	NOP
Código de Operación	0
Número de Operandos	0
Modos de Direccionamiento	N/A
Comportamiento	Ninguno No modifica los biestables de estado.

Instrucción	HALT
Descripción	Detener la ejecución de la máquina.
Formato	HALT
Código de Operación	1
Número de Operandos	0
Modos de Direccionamiento	N/A
Comportamiento	Activa el biestable H de fin de programa y detiene el procesador virtual

Instrucción	MOVE
Descripción	Copiar operando origen en operando destino.
Formato	MOVE op1, op2
Código de Operación	2
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: registro, memoria, indirecto, relativo Para el operando 2 no se permite direccionamiento inmediato, ya que no tiene sentido como destino de un movimiento de datos.
Comportamiento	Lee el contenido del operando 1 (origen) y lo escribe en el operando 2 (destino). No modifica los biestables de estado.

Instrucción	PUSH
Descripción	Poner en la pila.
Formato	PUSH op1

Código de Operación	3
Número de Operandos	1
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Depende del modo de funcionamiento de la pila: • Para crecimiento hacia direcciones descendentes de memoria, almacena el contenido del operando 1 en la dirección apuntada por SP y decrementa el valor del puntero de pila. • Para crecimiento hacia direcciones ascendentes de memoria, incrementa el valor del puntero de pila y almacena el contenido del operando 1 en la dirección apuntada por SP. No modifica los biestables de estado.

Instrucción	POP
Descripción	Sacar de la pila.
Formato	POP op1
Código de Operación	4
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo Para el operando no se permite direccionamiento inmediato, ya que no tiene sentido como destino de un movimiento de datos.
Comportamiento	Depende del modo de funcionamiento de la pila: • Para crecimiento hacia direcciones descendentes de memoria, incrementa el valor del puntero de pila y almacena en el operando 1 el valor contenido en la dirección de memoria apuntada por SP. • Para crecimiento hacia direcciones crecientes de memoria, almacena en el operando 1 el valor contenido en la dirección de memoria apuntada por SP y decrementa el valor del puntero de pila. No modifica los biestables de estado.

Instrucción	ADD
Descripción	Suma de números enteros.
Formato	ADD op1, op2
Código de Operación	5
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Suma el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. Modifica los biestables de estado.

Instrucción	SUB
Descripción	Resta de números enteros.
Formato	SUB op1, op2
Código de Operación	6
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Resta el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. Modifica los biestables de estado.

Instrucción	MUL
Descripción	Producto de números enteros.
Formato	MUL op1, op2
Código de Operación	7

Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Multiplica el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. Modifica los biestables de estado.

Instrucción	DIV
Descripción	Cociente de la división de números enteros.
Formato	DIV op1, op2
Código de Operación	8
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Divide el contenido del operando 1 y el operando 2, y almacena el cociente de la operación en el registro acumulador. Si el operando 2 contiene el valor 0, se genera una excepción de tipo “división por cero”. Modifica los biestables de estado.

Instrucción	MOD
Descripción	Resto de la división de números enteros.
Formato	DIV op1, op2
Código de Operación	9
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Divide el contenido del operando 1 y el operando 2, y almacena el resto de la operación en el registro acumulador. Si el operando 2 contiene el valor 0, se genera una excepción de tipo “división por cero”. Modifica los biestables de estado.

Instrucción	INC
Descripción	Incremento unitario.
Formato	INC op1
Código de Operación	10
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Incrementa el contenido del operando en una unidad. Modifica los biestables de estado.

Instrucción	DEC
Descripción	Decremento unitario.
Formato	DEC op1
Código de Operación	11
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Decrementa el contenido del operando en una unidad. Modifica los biestables de estado.

Instrucción	NEG
Descripción	Cambio de signo.
Formato	NEG op1
Código de Operación	12

Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Cambia de signo el operando. Modifica los biestables de estado.

Instrucción	CMP
Descripción	Comparación.
Formato	CMP op1, op2
Código de Operación	13
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Resta el contenido del operando 1 y el operando 2 (pero no almacena el resultado de la operación en ningún sitio). Modifica los biestables de estado.

Instrucción	AND
Descripción	Y Lógico.
Formato	AND op1, op2
Código de Operación	14
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'y lógico' bit a bit entre el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. La tabla de verdad de la operación es: op1 AND op2 | 0 | 1 0 | 0 | 0 1 | 0 | 1 No modifica los biestables de estado.

Instrucción	OR
Descripción	O Lógico.
Formato	OR op1, op2
Código de Operación	15
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'o lógico' bit a bit entre el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. La tabla de verdad de la operación es: op1 OR op2 | 0 | 1 0 | 0 | 1 1 | 1 | 1 No modifica los biestables de estado.

Instrucción	XOR
Descripción	O Exclusivo Lógico.
Formato	XOR op1, op2
Código de Operación	16
Número de Operandos	2

Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'o exclusivo lógico' bit a bit entre el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. La tabla de verdad de la operación es: op1 XOR op2 | 0 | 1 0 | 0 | 1 1 | 1 | 0 No modifica los biestables de estado.

Instrucción	NOT
Descripción	Negación Lógica.
Formato	NOT op1
Código de Operación	17
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'negación lógica' bit a bit en el operando 1. La tabla de verdad de la operación es: op1 | NOT op1 0 | 1 1 | 0 No modifica los biestables de estado.

Instrucción	BR
Descripción	Bifurcación incondicional.
Formato	BR op1
Código de Operación	18
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Carga el PC con el valor contenido del operando 1.

Instrucción	BZ
Descripción	Bifuración si resultado igual cero.
Formato	BZ op1
Código de Operación	19
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable Z está activo (Z=1), carga el PC con el valor contenido del operando 1.

Instrucción	BNZ
Descripción	Bifurcación si resultado distinto de cero.
Formato	BNZ op1
Código de Operación	20
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable Z está inactivo (Z=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BP
Descripción	Bifurcación si resultado positivo.
Formato	BP op1

Código de Operación	21
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable S está inactivo (S=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BN
Descripción	Bifurcación si resultado negativo.
Formato	BN op1
Código de Operación	22
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable S está activo (S=1), carga el PC con el valor contenido en el operando 1.

Instrucción	BV
Descripción	Bifuración si hay desbordamiento.
Formato	BV op1
Código de Operación	23
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable V está activo (V=1), carga el PC con el valor contenido en el operando 1.

Instrucción	BNV
Descripción	Bifurcación si no hay desbordamiento.
Formato	BNV op1
Código de Operación	24
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable V está inactivo (V=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BC
Descripción	Bifurcación si hay acarreo.
Formato	BC op1
Código de Operación	25
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable C está activo (C=1), carga el PC con el valor contenido en el operando 1.

Instrucción	BNC
Descripción	Bifurcación si no hay acarreo.
Formato	BNC op1
Código de Operación	26
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable C está inactivo (C=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BE
Descripción	Bifurcación si el resultado tiene paridad par.
Formato	BE op1
Código de Operación	27

Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable P está inactivo (P=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BO
Descripción	Bifurcación si el resultado tiene paridad impar.
Formato	BO op1
Código de Operación	28
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable P está activo (P=1), carga el PC con el valor contenido en el operando 1.

Instrucción	CALL
Descripción	Llamada a subrutina.
Formato	CALL op1
Código de Operación	29
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Almacena en la pila el valor del contador de programa y salta a la dirección de destino indicada por el operando 1.

Instrucción	RET
Descripción	Retorno de subrutina.
Formato	RET
Código de Operación	30
Número de Operandos	0
Modos de Direccionamiento	N/A
Comportamiento	Rescata de la pila el contenido del contador de programa.

Instrucción	INCHAR
Descripción	Leer un carácter.
Formato	INCHAR op1
Código de Operación	31
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Lee un carácter de la consola y lo codifica en *ASCII*, dejando a cero los 8 bits superiores de la palabra de 16 bits. Almacena el carácter leído en el operando 1.

Instrucción	ININT
Descripción	Leer un entero.
Formato	ININT op1
Código de Operación	32
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Lee un entero de la consola. Si la entrada no puede ser convertida en un número entero (porque se sale de rango o no es un número), la instrucción supondrá que se ha leído un cero. Los enteros se pueden introducir en formato decimal o hexadecimal (precedidos por '0x').

Instrucción	INSTR
Descripción	Leer una cadena.
Formato	INSTR op1
Código de Operación	33

Número de Operandos	1
Modos de Direccionamiento	op1: memoria, indirecto, relativo
Comportamiento	Lee una cadena de caracteres de la consola y la almacena en posiciones consecutivas de memoria a partir de la dirección indicada por el operando 1, acabada con el carácter '\0'. No se comprueba previamente si hay espacio para almacenar la cadena, por lo que se puede producir una excepción si se sobrepasa el límite superior de la memoria.

Instrucción	WRCHAR
Descripción	Escribir un carácter.
Formato	WRCHAR op1
Código de Operación	34
Número de Operandos	1
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Escribe en la consola el carácter cuyo valor ASCII se corresponde con los 8 bits inferiores del valor del operando 1.

Instrucción	WRINT
Descripción	Escribir un entero.
Formato	WRINT op1
Código de Operación	35
Número de Operandos	1
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Escribe en la consola el valor del operando 1. El formato de escritura (decimal o hexadecimal), dependerá de la configuración de visualización de números enteros. En cualquier caso, el formato decimal siempre se escribe con signo (-32768 a 32767).

Instrucción	WRSTR
Descripción	Escribir una cadena.
Formato	WRSTR op1
Código de Operación	36
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, indirecto, relativo
Comportamiento	Escribe en la consola una cadena de caracteres, que estará almacenada en memoria a partir de la dirección indicada por el operando 1. Se escribirán caracteres hasta que se llegue al carácter '\0'. Por tanto, puede ocurrir que durante este proceso se genere una excepción si no se encuentra el carácter de fin de cadena antes de llegar al límite superior de la memoria.

3.2.1.4. Formatos de Instrucción

Una vez definido el juego de instrucciones y los modos de direccionamiento permitidos para sus operandos, es el momento de diseñar los formatos con los que se representarán las instrucciones en memoria. No se trata de convertir esta fase del proyecto en un ejercicio de arquitectura de computadores. Sin embargo, sí que se va a hacer un pequeño estudio para intentar obtener un formato lo más homogéneo posible, pero sin perder de vista el objetivo de ahorrar algo de memoria en la representación de las instrucciones.

Como se ha visto, se permiten 6 modos de direccionamiento. Desdoblando el modo de direccionamiento relativo a registro índice en dos, ya que el sistema cuenta con dos índices, IX e IY, y si se considera la ausencia de operando como un modo de direccionamiento más, serán necesarios 3 bits para codificar los 8 modos posibles, según la Tabla 3.2.1.

Modo	Ancho de representación interna	Codificación
Sin Operando	N/A	000 (0)
Inmediato	16 bits	001 (1)
Registro	4 bits	010 (2)
Memoria	16 bits	011 (3)
Indirecto	4 bits	100 (4)
Relativo a IX	8 bits	101 (5)
Relativo a IY	8 bits	110 (6)
Relativo a PC	8 bits	111 (7)

Tabla 3.2.1. Codificación de Modos de Direccionamiento

A la hora de expresar los operandos, los habrá de varios tipos: enteros, direcciones de memoria, desplazamientos e identificadores de registros. Enteros y direcciones se codifican con 16 bits, los desplazamientos con 8 bits y los registros con cuatro bits, según la Tabla 3.2.2.

Registro	Código	Registro	Código
R0	0000	R8	1000
R1	0001	R9	1001
R2	0010	A	1010
R3	0011	SR	1011
R4	0100	IX	1100
R5	0101	IY	1101
R6	0110	SP	1110
R7	0111	PC	1111

Tabla 3.2.2. Codificación de Registros

Para codificar las 37 instrucciones definidas se necesitan un mínimo de 6 bits. De esta forma, para construir una instrucción se necesitan como mínimo 12 bits (6 para codificar el mnemónico, 3 para el modo de direccionamiento del primer operando y otros 3 para codificar el modo de direccionamiento del segundo). Para que la estructura obtenida sea más regular, y de paso reservando espacio para futuras ampliaciones del juego de instrucciones, se ha tomado la decisión de emplear 10 bits para el código de operación, con lo cual se completa una palabra de 16 bits de la siguiente forma:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			Modo dir. op2		

A la hora de codificar los operandos, y ajustándose a la tabla anterior, se observa que según el modo de direccionamiento puede que no sea necesario ningún bit, o bien se necesiten hasta 32. Estudiando todas las combinaciones se puede ver fácilmente que se pueden combinar dos operandos en una sola palabra de 16 bits siempre y cuando ninguno de éstos venga indicado con direccionamiento inmediato o directo a memoria. En estos dos supuestos, el operando ocupará una palabra completa por sí mismo.

Puede ser que de 8 bits disponibles sólo sean necesarios 4. En ese caso, se rellenará el cuarteto (*nibble*) superior con ceros. En cambio, a la hora de dividir una palabra en 2 octetos, el octeto más significativo siempre se usará para el primer operando y el menos significativo para el segundo operando.

A modo de resumen, y según el número de operandos, se definen los siguientes formatos de instrucción:

- Instrucciones sin operandos.

  Sólo se emplean los bits correspondientes al código de operación (bits 15-6). El resto, se rellena con ceros.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										0	0	0	0	0	0

  Ejemplo:

```
RET = 07C0
```

  El código de operación de la instrucción `RET` es 30 (que en binario es 11110). Por tanto, se rellenan los bits 15 a 6 con dicho valor, y el resto se deja a cero.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0

- Instrucciones con un operando.

  Si el modo de direccionamiento es inmediato o directo a memoria, el operando necesita 16 bits para ser representado (corresponde con la segunda palabra). En la primera, el espacio correspondiente al modo de direccionamiento del segundo operando se rellena con ceros.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 1															

  Ejemplo:

```
INC /0x2000 = 0298 2000
```

  El código de operación de la instrucción `INC` es 10 (1010 en binario), y corresponde a los bits 15-6 de la primera palabra. Los bits 5-3 se corresponden con el modo de direccionamiento, que en este caso es directo a memoria, y se codifica con el valor 3 (001). En la segunda palabra va almacenado el valor del operando.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	1	1	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

  En otro caso, el operando sólo ocupará el byte superior de la segunda palabra. El resto se rellenará con ceros.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 1								0	0	0	0	0	0	0	0

Ejemplo:

```
PUSH .R1 = 00D0 0100
```

El código de operación de la instrucción `PUSH` es 3 (11 en binario), y corresponde a los bits 15-6 de la primera palabra. Los bits 5-3 se corresponden con el modo de direccionamiento, que en este caso es directo a registro, y se codifica con el valor 2 (010). En el byte superior de la segunda palabra va almacenado el valor del operando, que como es el registro R1 se codifica con el valor 1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

- Instrucciones con dos operandos.

  Si ambos modos de direccionamiento son ambos inmediatos o ambos directos a memoria, o uno de cada, en cualquier orden, las palabras segunda y tercera se corresponderán en su totalidad con el valor del primer y segundo operando, respectivamente. En este caso todos los campos del formato de instrucción toman valor.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			Modo dir. op2		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 1															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 2															

  Ejemplo:

```
ADD /0x2000,#0xFF = 0159 2000 00FF
```

  El código de operación de la instrucción `ADD` es 5 (101 en binario), y corresponde a los bits 15-6 de la primera palabra. Los bits 5-3 se corresponden con el modo de direccionamiento del primero operando, que en este caso es directo a memoria, y se codifica con el valor 3 (011). Los bits 2-0 se corresponden con el modo de direccionamiento del segundo operando, que en este caso es inmediato, y se codifica con el valor 1 (001). En la segunda palabra se almacena el valor del primer operando4 y en la tercera el valor del segundo operando.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Si alguno de los modos de direccionamiento es inmediato o directo a memoria, pero el otro no, entonces este último operando necesita una palabra para ser almacenado, si bien sólo ocupará un byte, el inferior si se trata del segundo operando:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			Modo dir. op2		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 1															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Operando 2							

Ejemplo:

```
SUB /0x2000,#0x10[.IX] = 019D 2000 0010
```

El código de operación de la instrucción `SUB` es 6 (110 en binario), y corresponde a los bits 15-6 de la primera palabra. Los bits 5-3 se corresponden con el modo de direccionamiento del primero operando, que en este caso es directo a memoria, y se codifica con el valor 3 (011). Los bits 2-0 se corresponden con el modo de direccionamiento del segundo operando, que en este caso es relativo a índice IX, y se codifica con el valor 5 (101). En la segunda palabra se almacena el valor del primer operando y, en la tercera, en el byte inferior, el valor del segundo operando.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	0	1	1	1	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

O bien se almacena en el byte superior si se trata del primer operando.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			Modo dir. op2		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 1								0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 2															

Ejemplo:

```
AND .R7,#0xFFFF = 0391 0700 FFFF
```

El código de operación de la instrucción `AND` es 14 (1110 en binario), y corresponde a los bits 15-6 de la primera palabra. Los bits 5-3 se corresponden con el modo de direccionamiento del primero operando, que en este caso es directo a registro, y se codifica con el valor 2 (010). Los bits 2-0 se corresponden con el modo de direccionamiento del segundo operando, que en este caso es inmediato, y

se codifica con el valor 1 (001). En el byte superior de la segunda palabra se almacena el valor del primer operando y, en la tercera palabra, se almacena el valor del segundo operando.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

En otro caso, es decir, cuando ninguno de los operandos poseen un direccionamiento inmediato o directo a memoria, se pueden almacenar ambos en una sola palabra, el primero en el byte superior y el segundo en el byte inferior.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código de operación										Modo dir. op1			Modo dir. op2		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operando 1								Operando 2							

Ejemplo:

```
MUL .R8,[.R3] = 01D4 0803
```

El código de operación de la instrucción `MUL` es 7 (111 en binario), y corresponde a los bits 15-6 de la primera palabra. Los bits 5-3 se corresponden con el modo de direccionamiento del primero operando, que en este caso es directo a registro, y se codifica con el valor 2 (010). Los bits 2-0 se corresponden con el modo de direccionamiento del segundo operando, que en este caso es indirecto, y se codifica con el valor 4 (100). En el byte superior de la segunda palabra se almacena el valor del primer operando y en el byte inferior se almacena el valor del segundo operando.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1

3.2.1.5. Etiquetas

Las etiquetas son identificadores textuales que sirven como nombres simbólicos. Deben comenzar por un carácter alfabético, seguido o no de caracteres alfanuméricos o el carácter subrayado (_).

A la hora de definir su valor, como se ha visto en el formato del código fuente, preceden a una instrucción y van seguidas del carácter dos puntos (:). En principio toman el valor de la posición de memoria que ocupa la instrucción a la que acompañan, aunque se les puede asignar un valor arbitrario mediante pseudoinstrucciones, como se contempla en el siguiente apartado.

En cuanto a su uso, pueden reemplazar a cualquier operando en el que se permita un valor entero (decimal o hexadecimal). Cuando sea ensamblada la instrucción en cuestión, o bien cuando se haya resuelto el valor de la etiqueta, se sustituirá dicho valor en el código máquina generado.

3.2.1.6. Pseudoinstrucciones del Ensamblador

Las pseudoinstrucciones (o macroinstrucciones) son órdenes dirigidas al programa ensamblador. Sirven para indicarle cómo se quiere que realice la traducción del programa, pero no forman parte de él. Sin embargo, algunas pseudoinstrucciones sirven para reservar posiciones de memoria, destinadas a los datos [De Miguel 1996].

El ensamblador de *ENS2001* soportará las instrucciones más importantes del *IEEE 694*, a excepción de las destinadas a importación y exportación de símbolos entre módulos, lo que no tiene sentido aquí ya que los listados de código fuente permitidos constan de un único módulo.

Pseudoinstrucción	ORG
Descripción	Origen del código
Formato	ORG dir
Comportamiento	Cada vez que se encuentre una pseudoinstrucción ORG en el código fuente, se proseguirá ensamblando a partir de la posición de memoria indicada por `dir`. Si no se indica la dirección, el código generado se almacenará a partir de la posición 0.

Pseudoinstrucción	EQU
Descripción	Equivalencia
Formato	etiqueta: EQU expresión
Comportamiento	Establece una equivalencia entre un nombre simbólico (`etiqueta`) y una constante numérica o expresión estática (`expresión`). Si no se indica ninguna etiqueta, la expresión será evaluada (pudiendo generarse errores), pero no será asignada a ningún nombre simbólico.

Pseudoinstrucción	END
Descripción	Fin del código
Formato	END
Comportamiento	Señala el final del código fuente. No es obligatoria, ya que si no aparece, se toma el final del fichero de texto (`EOF`) como final del código. Todo lo que aparezca en el fichero de código fuente tras una pseudoinstrucción END será ignorado.

Pseudoinstrucción	RES
Descripción	Reserva de Memoria
Formato	etiqueta: RES n
Comportamiento	Asigna `n` posiciones de memoria sin inicializar. La `etiqueta` hace referencia a la primera de ellas. Si no se indica ninguna etiqueta, se reservará memoria, pero no se asignará la dirección de comienzo de dicha zona a ningún nombre simbólico.

Pseudoinstrucción	DATA
Descripción	Definición de datos
Formato	etiqueta: DATA a, b, c...
Comportamiento	Asigna una serie de posiciones de memoria que va llenando con los datos `a`, `b`, `c`… Dichos datos pueden ser numéricos (enteros dentro del rango permitido) o cadenas de caracteres, en cuyo caso se deben entrecomillar. Las cadenas se almacenarán en memoria según el estilo C, un carácter por posición, finalizando con el carácter nulo '\0'. Las cadenas pueden estar compuestas de cualquier carácter imprimible, y pueden contener los caracteres especiales '\0', '\n' y '\t'. Si no se indica la etiqueta, se almacenarán los datos en memoria igualmente, pero no se asignará la dirección de comienzo de dicha zona de datos.

3.2.1.7. Comentarios

Está permitido (además de ser recomendable) introducir comentarios en el código fuente. Se considera comentario cualquier carácter escrito a partir del carácter punto y coma (`;`) hasta el final de la línea, y como tales, serán ignorados por el ensamblador.

3.2.2. Análisis del Ensamblador

El módulo ensamblador se encarga de convertir las sentencias de un programa escrito en lenguaje ensamblador, esto es, de un programa fuente, en el correspondiente programa en lenguaje máquina, esto es, en un programa objeto [De Miguel 1996].

Para comprender el funcionamiento del ensamblador, se ejemplifican esquemáticamente a continuación algunos de los problemas que debe resolver.

Ante una instrucción en el código fuente como:

```
SUB .R1,.R2
```

La traducción es sencilla, basta con buscar el código de operación correspondiente en una tabla, anotar que el direccionamiento de los operandos es directo a registro (lo que también está codificado), y dar a los operandos el valor correspondiente (el identificador de registro, que también está codificado). Así se obtendría el código objeto de forma inmediata:

```
01h 92h 01h 02h
```

El problema inicial queda resuelto consultando en unas tablas. Profundizando un poco más, sean estas dos instrucciones:

```
ADD .R2, #23
BR /2000
```

Se complica un poco la traducción. Además de buscar en la tabla los códigos de operación y los modos de direccionamiento, se deben convertir los enteros de decimal a binario, comprobar que las instrucciones permiten esta combinación de modos de direccionamiento en sus operandos y que se encuentran dentro del rango permitido. El resultado sería:

```
01h 51h 02h 00h 00h 17h
04h 98h 07h d0h
```

Sin embargo, tampoco ha sido muy complicado. La traducción se resuelve consultando en unas tablas, haciendo conversiones numéricas y comprobando ciertas condiciones. Para complicar el proceso un poco más, sean las instrucciones:

```
SUB .R3,#DAT1
BR $ETIQ3
```

En este caso se han empleado etiquetas (operandos simbólicos). El problema que suele aparecer es que, al ensamblar estas instrucciones, es posible que el ensamblador aún no conozca el valor que toman dichas etiquetas (referencias adelantadas). Por tanto, deberá llevar algún tipo de control sobre las etiquetas, y reservar espacio para actualizarlo con el valor correspondiente cuando se defina más adelante en el programa fuente.

Como añadido, el ensamblador permite macroinstrucciones de asignación de valores a etiquetas (`EQU`) en las que se pueden emplear constantes enteras o expresiones constantes. Por tanto, a la hora de evaluar y ejecutar las macroinstrucciones, también habrá que considerar este aspecto.

Teniendo en cuenta que las sentencias de código máquina son independientes del contexto, el análisis de las mismas se puede realizar una a una. De hecho, esto no es del todo cierto, pero la única relación que puede existir entre las distintas sentencias son las etiquetas, cuya resolución se deja para el final. El análisis de una sentencia de ensamblador se hace sobre cada uno de sus campos:

1. Comprobar si la sentencia está precedida por una etiqueta. En caso afirmativo, hay que comprobar que el nombre simbólico no se haya definido con anterioridad, y asignarle el valor correspondiente a la posición de memoria que ocupará la instrucción, dato o pseudoinstrucción que se está analizando. Se almacenará la etiqueta en una tabla de etiquetas.
2. Comprobar que el mnemónico es correcto y si corresponde a una instrucción o a una pseudoinstrucción. Aquí se obtiene la primera parte del código de operación. La otra se extraerá de los operandos, en concreto de su modo de direccionamiento.
3. Conocida la instrucción, habrá que determinar si los operandos son válidos. Hay que analizar si su número y modos de direccionamiento son correctos. Es posible que en este paso, el valor de los operandos o sus direcciones no sean conocidos, ya que pueden intervenir referencias adelantadas. Sin embargo, el tipo siempre puede ser determinado. Así se dispone de todo lo necesario para seleccionar el código de operación, así como el formato y la longitud de la instrucción máquina correspondiente. Se almacenarán las referencias adelantadas que haya en una tabla (Tabla de Referencias Adelantadas).

Tras analizar todas y cada una de las sentencias, las etiquetas habrán ido tomando valor. Por tanto, sólo restaría sustituir dicho valor en las referencias adelantadas. Este procedimiento se conoce como “ensamblador de pasada y media” [De Miguel 1996].

Ahora, cambiando de punto de vista, sería interesante ver el ensamblador como un caso particular de compilador. Un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto [Aho 1990]. Además, como parte importante de este proceso, el compilador informa de la presencia de errores en el código fuente.

Generalizando, un compilador consta de las siguientes fases, cada una de las cuales transforma al programa fuente de una representación en otra más cercana a la máquina:

- Analizador léxico.
- Analizador sintáctico.
- Analizador semántico.
- Generador de código intermedio.
- Optimizador de código.
- Generador de código.

Y dos módulos que se utilizan durante todo el proceso:

- Administrador de tabla de símbolos.
- Manejador de errores.

Se trata de adaptar la estructura general del compilador a la tarea de ensamblar código fuente en lenguaje ensamblador (Figura 3.2.7).

- Analizador léxico.
  Transforma la entrada, que no es otra cosa que un flujo de caracteres, en los elementos léxicos, como pueden ser instrucciones, operandos, comentarios, etiquetas, etc.

- Analizador Sintáctico.
  Comprueba que la secuencia de elementos léxicos que obtiene del analizador léxico se corresponde con la estructura predefinida como válida para el código fuente, esto es, que cada línea siga el esquema:

  ```
  [etiqueta ':'] [instrucción [operando [',' operando]] [';' comentario]
  ```

- Analizador Semántico.
  Comprueba que los operandos introducidos y sus modos de direccionamiento son correctos para cada instrucción. También comprueba que los operandos están definidos dentro de los rangos establecidos, que no hay etiquetas duplicadas, etc. Generalmente, este proceso se puede realizar simultáneamente al análisis sintáctico. En el caso de *ENS2001* se va a resolver así, completando la gramática del analizador sintáctico con las acciones semánticas oportunas.

- Generador de código intermedio.
  En este caso no procede, ya que se va a generar directamente código final.

- Optimizador de código.
  No procede tampoco, ya que el código escrito en ensamblador no se optimiza, se traduce tal cual está escrito.

- Generador de código.
  Con la información contenida en la tabla de símbolos (tabla de etiquetas) y la tabla de configuración (referencias adelantadas), construye el código final (código máquina).

Todos los errores que vayan surgiendo durante las sucesivas etapas del proceso se guardarán en una tabla de errores. Si hay errores, no se genera código máquina, y habrá que informar al usuario de los errores detectados y su ubicación en el código fuente. Se da por supuesto que el ensamblador recibe como entrada un archivo que existe y que se puede leer. También se considera que el ensamblador puede generar su salida sin ningún

problema. Por tanto, los errores de entrada/salida de disco y semejantes no se contemplan aquí.

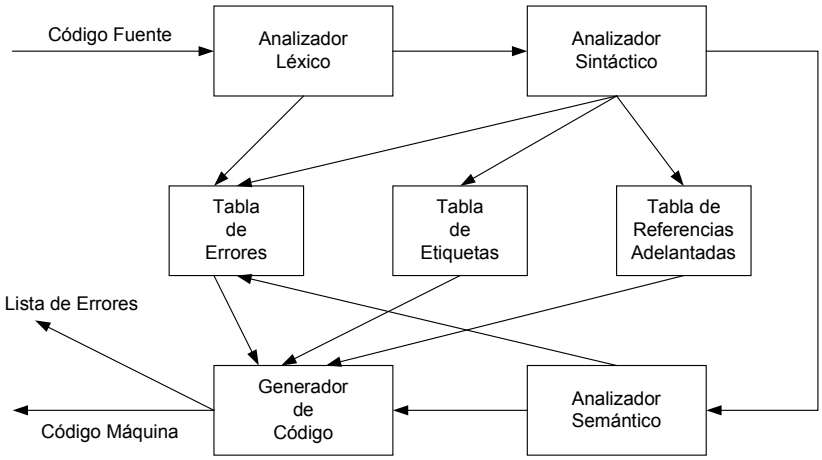


Figura 3.2.7. Esquema Modular del Ensamblador

3.2.2.1. Analizador Léxico

El comportamiento del analizador léxico está definido mediante una gramática regular que valida todos y cada uno de los elementos (*tokens*) que componen el lenguaje ensamblador. Con estos *tokens*, el analizador sintáctico comprobará la sintaxis del código fuente y comenzará la traducción a código máquina.

No obstante, no todos los caracteres que se encuentren a la entrada se comunicarán al analizador sintáctico:

- Se filtrarán los caracteres ESPACIO y TABULADOR.
- Se filtrarán los comentarios, esto es, todos los caracteres que aparezcan después de un indicador de comentario (punto y coma) hasta el final de la línea, con el indicador incluido (pero no el fin de línea).
- Los caracteres no válidos a la entrada se informarán como errores (indicando el número de línea en el que se encontraron), y se pasará a analizar la siguiente línea de código fuente.

Los *tokens* pueden llevar un atributo asociado, que contendrá información adicional sobre cada componente léxico. Esta información será de utilidad para el analizador sintáctico y el generador de código. En este caso concreto, el atributo podrá ser de dos tipos, bien un número entero, bien una cadena de caracteres.

La gramática, como suele ser habitual, comienza con el axioma S. Los símbolos no terminales se escriben con letras mayúsculas y los terminales con minúsculas. Para disminuir el número de reglas de la gramática, se asumen las siguientes equivalencias para símbolos terminales:

l = [a..zA..Z_]
d = [0..9]
d_l = [1..9]
d_h = [0..9a..f] $\cup$ [A..F]
c = cualquier carácter excepto comillas

Esta es la gramática del analizador léxico:

S → 'eof' | 'eol' | ',' | ':' | '+' | '-' | '*' | '%' | '(' | ')' | '#' A | '.' B | '/' C |
'[' D | '$' E | '"' F | l G | d_1 H | 0 I | ';' J | ' ' S | '\t' S

En este conjunto de reglas se reconocen *tokens* correspondientes a operandos cuyo modo de direccionamiento es inmediato:

A → d_1 A_1 | '0' A_2 | '-' A_3 | l A_4
A_1 → d A_1 | λ
A_2 → 'x' A_5 | 'X' A_5 | d A_1 | λ
A_3 → d A_1 | d
A_4 → l A_4 | d A_4 | λ
A_5 → d_h A_6
A_6 → d_h A_6 | λ

En este conjunto de reglas se reconocen *tokens* correspondientes a operandos cuyo modo de direccionamiento es directo a registro:

B → 'r' B_1 | 'R' B_1 | 'p' B_2 | 's' B_3 | 'i' B_4 | 'a' | 'P' B_5 | 'S' B_6 | 'I' B_7 | 'A'
B_1 → d
B_2 → 'c'
B_3 → 'p' | 'r'
B_4 → 'x' | 'y'
B_5 → 'C'
B_6 → 'P' | 'R'
B_7 → 'X' | 'Y'

En este conjunto de reglas se reconocen *tokens* correspondientes a operandos cuyo modo de direccionamiento es directo a memoria:

C → d_1 C_1 | '0' C_2 | '-' C_3 | l C_4
C_1 → d C_1 | λ
C_2 → 'x' C_5 | 'X' C_5 | d C_1 | λ
C_3 → d C_1 | d
C_4 → l C_4 | d C_4 | λ
C_5 → d_h C_6
C_6 → d_h C_6 | λ

En este conjunto de reglas se reconocen *tokens* correspondientes a operandos cuyo modo de direccionamiento es indirecto:

D → '.' D_1
D_1 → 'r' D_2 | 'R' D_2 | 'p' D_3 | 's' D_4 | 'i' D_5 | 'a' D_6 | 'P' D_7 | 'S' D_8 | 'I' D_9 | 'A' D_6
D_2 → d D_6
D_3 → 'c' D_6
D_4 → 'p' D_6 | 'r' D_6
D_5 → 'x' D_6 | 'y' D_6

$D_6 \rightarrow$ `']'`
$D_7 \rightarrow$ `'C'` D_6
$D_8 \rightarrow$ `'P'` D_6 | `'R'` D_6
$D_9 \rightarrow$ `'X'` D_6 | `'Y'` D_6

En este conjunto de reglas se reconocen *tokens* correspondientes a operandos cuyo modo de direccionamiento es relativo a PC:

$E \rightarrow$ $d_1\ E_1$ | `'0'` E_2 | `'-'` E_3 | l E_4
$E_1 \rightarrow$ d E_1 | λ
$E_2 \rightarrow$ `'x'` E_5 | d E_1 | λ
$E_3 \rightarrow$ d E_1 | d
$E_4 \rightarrow$ l E_4 | d E_4 | λ
$E_5 \rightarrow$ $d_h\ E_6$
$E_6 \rightarrow$ $d_h\ E_6$ | λ

Con esta regla se reconocen *tokens* correspondientes a cadenas:

$F \rightarrow$ c F | `'"'`

Con esta regla se reconocen *tokens* correspondientes a identificadores. Una vez reconocidos, se consultará en una tabla (Tabla 3.2.3) si son palabras reservadas, que generan distintos tipos de *tokens* y atributos:

$G \rightarrow$ l G | d G | λ

Con esta regla se reconocen *tokens* correspondientes a enteros decimales:

$H \rightarrow$ d H | λ

En este conjunto de reglas se reconocen *tokens* correspondientes a enteros hexadecimales (o bien decimales que comiencen por el dígito 0):

$I \rightarrow$ d I_1 | `'x'` I_2 | `'X'` I_2 | λ
$I_1 \rightarrow$ d I_1 | λ
$I_2 \rightarrow$ $d_h\ I_3$
$I_3 \rightarrow$ $d_h\ I_3$ | λ

Con esta regla se reconocen los comentarios en el código fuente:

$J \rightarrow$ c J | `'"'` J | `'eol'`

Partiendo del conjunto de palabras aceptadas por la gramática, se detallan los distintos *tokens* que se van a comunicar al analizador sintáctico:

- Las palabras reservadas correspondientes con instrucciones del lenguaje ensamblador pueden generar tres *tokens* diferentes dependiendo del número de operandos que estén permitidos para dicha instrucción. El atributo contendrá el código de operación de la instrucción: `<MNEMÓNICO0, cód_op>`, `<MNEMÓNICO1, cód_op>`, `<MNEMÓNICO2,`

<cód_op>. Para saber si un identificador constituye una palabra reservada (en este caso una instrucción), se consultará la Tabla 3.2.3.

- Las pseudoinstrucciones generarán un *token* diferente cada una de ellas. El atributo no representa ningún valor: <ORG, 0>, <EQU, 0>, <END, 0>, <RES, 0> y <DATA ,0>. Para saber si un identificador constituye una palabra reservada (en este caso una pseudoinstrucción), se consultará la Tabla 3.2.3.

- Los caracteres fin de línea y fin de fichero generan el *token* <EOL, 0>.

- Los caracteres separador (',') y fin de etiqueta (':') generan dos *tokens*, <SEPARADOR, 0> y <FIN_ETIQ, 0>, respectivamente.

- Los caracteres correspondientes a operadores aritméticos generan diferentes *tokens*: <SUMA, 0>, <RESTA, 0>, <PRODUCTO, 0>, <DIVISIÓN, 0>, <MÓDULO, 0>, <PARENT_ABRE, 0> y <PARENT_CIERRA, 0>.

- Los operandos y sus modos de direccionamiento generan distintos *tokens*. También se hace distinción si los operandos son enteros o etiquetas. Si el operando es un entero, el atributo contendrá su valor, si es un registro, contendrá un identificador numérico y, si se trata de una etiqueta, contendrá un identificador, el nombre de la etiqueta: <INMEDIATO_V, valor>, <INMEDIATO_E, identificador>, <REGISTRO, id_numérico>, <MEMORIA_V, valor>, <MEMORIA_E, identificador>, <INDIRECTO, id_numérico>, <RELAT_PC_V, valor>, <RELAT_PC_E, identificador>, <RELAT_IX_V, valor>, <RELAT_IX_E, identificador>, <RELAT_IY_V, valor>, <RELAT_IY_E, identificador>.

- Por último, se generarán los tokens correspondientes a cadenas de caracteres, identificadores y números enteros (decimales o hexadecimales). En el caso de las cadenas, el atributo contendrá la cadena de caracteres analizada, en el caso de las etiquetas, la cadena correspondiente al identificador y, en el caso de enteros, el valor correspondiente: <CADENA, texto>, <IDENTIFICADOR, texto>, <ENTERO, valor>.

Palabra clave	Tipo de Token	Atributo
nop	MNEMÓNICO0	0
halt	MNEMÓNICO0	1
move	MNEMÓNICO2	2
push	MNEMÓNICO1	3
pop	MNEMÓNICO1	4
add	MNEMÓNICO2	5
sub	MNEMÓNICO2	6
mul	MNEMÓNICO2	7
div	MNEMÓNICO2	8
mod	MNEMÓNICO2	9
inc	MNEMÓNICO1	10
dec	MNEMÓNICO1	11
neg	MNEMÓNICO1	12
cmp	MNEMÓNICO2	13
and	MNEMÓNICO2	14
or	MNEMÓNICO2	15
xor	MNEMÓNICO2	16
not	MNEMÓNICO1	17
br	MNEMÓNICO1	18

bz	MNEMÓNICO1	19
bnz	MNEMÓNICO1	20
bp	MNEMÓNICO1	21
bn	MNEMÓNICO1	22
bv	MNEMÓNICO1	23
bnv	MNEMÓNICO1	24
bc	MNEMÓNICO1	25
bnc	MNEMÓNICO1	26
be	MNEMÓNICO1	27
bo	MNEMÓNICO1	28
call	MNEMÓNICO1	29
ret	MNEMÓNICO0	30
inchar	MNEMÓNICO1	31
inint	MNEMÓNICO1	32
instr	MNEMÓNICO1	33
wrchar	MNEMÓNICO1	34
wrint	MNEMÓNICO1	35
wrstr	MNEMÓNICO1	36
org	ORG	0
end	END	0
equ	EQU	0
res	RES	0
data	DATA	0

Tabla 3.2.3. Palabras clave del lenguaje ensamblador

3.2.2.2. Analizador Sintáctico

El analizador léxico realiza un filtro previo de la entrada. En concreto eliminará los espacios y tabuladores, así como los comentarios inmersos entre el código. También evitará tratar con caracteres erróneos. Por tanto, el conjunto de entradas posibles (*tokens*) para el analizador sintáctico es el que se muestra en la Tabla 3.2.4.

Token	Tipo de Atributo
<MNEMÓNICO0,cód_op>	entero
<MNEMÓNICO1,cód_op>	entero
<MNEMÓNICO2,cód_op>	entero
<ORG,0>	ninguno
<END,0>	ninguno
<EQU,0>	ninguno
<RES,0>	ninguno
<DATA,0>	ninguno
<EOL,fin_de_fichero>	lógico
<SEPARADOR,0>	ninguno
<FIN_ETIQ,0>	ninguno
<SUMA,0>	ninguno
<RESTA,0>	ninguno
<PRODUCTO,0>	ninguno
<DIVISIÓN,0>	ninguno
<MÓDULO,0>	ninguno
<PARENT_ABRE,0>	ninguno
<PARENT_CIERRA,0>	ninguno
<INMEDIATO_V,valor>	entero
<INMEDIATO_E,identificador>	cadena de caracteres
<REGISTRO,id_numérico>	entero
<MEMORIA_V,valor>	entero
<MEMORIA_E,identificador>	cadena de caracteres
<INDIRECTO,id_numérico>	entero

<INDIRECTO,id_numérico>	entero
<RELAT_PC_V,valor>	entero
<RELAT_PC_E,identificador>	cadena de caracteres
<RELAT_IX_V,valor>	entero
<RELAT_IX_E,identificador>	cadena de caracteres
<RELAT_IY_V,valor>	entero
<RELAT_IY_E,identificador>	cadena de caracteres
<CADENA,texto>	cadena de caracteres
<IDENTIFICADOR,texto>	cadena de caracteres
<ENTERO,valor>	entero

Tabla 3.2.4. Tabla de tokens

Se procederá a continuación a construir paso a paso la gramática de contexto libre que define la estructura del lenguaje ensamblador. Los símbolos no terminales se muestran en mayúsculas y los *tokens* con un tipo de letra monoespaciado y entre los signos de menor que y mayor que.

En primera instancia, un programa puede estar vacío, contener líneas vacías o líneas con contenido. Así, del axioma (S) se derivarán las siguientes producciones:

S → λ
| `<EOL>` S
| LÍNEA S

Una línea puede contener una instrucción, la definición de una etiqueta y una instrucción o comentarios. Los comentarios los elimina el analizador léxico, por lo que no es necesario tenerlos en cuenta. Aunque se permite la inclusión de líneas en blanco entre la etiqueta y la instrucción, a efectos sintácticos, la definición de una etiqueta y la instrucción forman parte de la misma entidad (una sentencia), ya que no se pueden definir dos etiquetas seguidas. Por tanto, se propone la siguiente regla para definir las líneas de código fuente:

LÍNEA → `<IDENTIFICADOR> <FIN_ETIQ>` LÍNEA_VACÍA RESTO
| RESTO

Y esta regla permite la inclusión de líneas en blanco entre la etiqueta y la instrucción:

LÍNEA_VACÍA → λ
| `<EOL>` LÍNEA_VACÍA

Hay dos tipos de instrucciones, las del ensamblador propiamente dichas y las macroinstrucciones. Para distinguirlas, ya que las últimas no se traducen en general en código máquina, se propone la siguiente regla:

RESTO → INSTRUCCIÓN
| PSEUDOINSTRUCCIÓN

Las instrucciones se clasifican según el número de operandos que permiten, a saber, ninguno, uno o dos. Los operandos, cuando hay dos, se separan por una coma. En cualquier caso, las instrucciones siempre acaban con un carácter de salto de línea. Todo esto se recoge en la siguiente regla:

```
INSTRUCCIÓN → <MNEMÓNICO0> <EOL>
|             <MNEMÓNICO1> OPERANDO <EOL>
|             <MNEMÓNICO2> OPERANDO <SEPARADOR> OPERANDO
              <EOL>
```

El lenguaje permite siete modos de direccionamiento. En cinco de ellos, el operando puede ser un entero o una etiqueta (referencia), y en los otros dos, el operando será un identificador de registro. Por tanto, se contabilizan doce tipos distintos de operandos (según esta clasificación):

```
OPERANDO → <INMEDIATO_V>
|          <INMEDIATO_E>
|          <REGISTRO>
|          <MEMORIA_V>
|          <MEMORIA_E>
|          <INDIRECTO>
|          <RELAT_PC_V>
|          <RELAT_PC_E>
|          <RELAT_IX_V>
|          <RELAT_IX_E>
|          <RELAT_IY_V>
|          <RELAT_IY_E>
```

El lenguaje también posee una serie de pseudoinstrucciones. Tres de ellas tienen una expresión constante como operando, otra una lista de datos (enteros o cadenas de caracteres) separados por comas, y la última no admite operandos:

```
PSEUDOINSTRUCCIÓN → <ORG> EXPRESIÓN <EOL>
|                   <RES> EXPRESIÓN <EOL>
|                   <DATA> LISTA_DATOS <EOL>
|                   <EQU> EXPRESIÓN <EOL>
|                   <END> <EOL>
```

El análisis de expresiones sigue el esquema clásico, que da más precedencia a los paréntesis y el menos unario, luego a las operaciones de producto, división y módulo, y por último a las sumas y restas.

```
EXPRESION → EXPRESIÓN <SUMA> EXPRESIÓN2
|           EXPRESIÓN <RESTA> EXPRESIÓN2
|           EXPRESIÓN2

EXPRESIÓN2 → EXPRESIÓN2 <PRODUCTO> EXPRESIÓN3
|            EXPRESIÓN2 <DIVISIÓN> EXPRESIÓN3
|            EXPRESIÓN2 <MÓDULO> EXPRESIÓN3
|            EXPRESIÓN3

EXPRESIÓN3 → <PARENT_ABRE> EXPRESIÓN <PARENT_CIERRA>
|            <RESTA> EXPRESIÓN
|            <ENTERO>
```

La lista de datos, enteros y cadenas de caracteres encerradas entre comillas dobles separados por comas, se analiza según este par de reglas:

LISTA_DATOS → <ENTERO> RESTO_LISTA_DATOS
| <CADENA> RESTO_LISTA_DATOS

RESTO_LISTA_DATOS → λ
| <SEPARADOR> LISTA_DATOS

Cualquier *token* de entrada que no se pueda equiparar con la regla que se esté analizando generará un error sintáctico. Cuando se produzca un error, el analizador intentará recuperarse. Para ello, descartará la entrada hasta que encuentre un fin de línea, y entonces intentará equiparar la regla cuya parte izquierda es el no terminal *LÍNEA*.

El analizador sintáctico puede detectar estos errores:

- *Instrucción No Reconocida*. Ocurre cuando no se puede ajustar el primer *token* recibido a las reglas INSTRUCCIÓN o PSEUDOINSTRUCCIÓN.
- *Operando 1 incorrecto*. Ocurre al no poder ajustar el símbolo no terminal OPERANDO cuando se está analizando el primer operando de una instrucción (de uno o dos operandos).
- *Operando 2 incorrecto*. Ocurre al no poder ajustar el símbolo no terminal OPERANDO cuando se está analizando el segundo operando de una instrucción (de dos operandos).
- *Expresión errónea*. Ocurre cuando el analizador está resolviendo las reglas relativas a expresiones y no puede ajustar el *token* que le viene de la entrada.
- *Se esperaba el Operando 1*. Ocurre cuando hay un error analizando la regla OPERANDO cuando se trata del primero de la instrucción que está siendo analizada.
- *Se esperaba el Operando 2*. Ocurre cuando hay un error analizando la regla OPERANDO cuando se trata del segundo de la instrucción que está siendo analizada.
- *Se esperaba fin de línea*. Ocurre cuando se ha analizado una instrucción o una pseudoinstrucción completa, a falta del *token* EOL, y se recibe otro distinto.
- *Se esperaba separador (coma)*. Ocurre cuando se está analizando una instrucción de dos operandos y se recibe un *token* distinto de SEPARADOR.
- *Lista de datos errónea*. Ocurre cuando se está analizando la regla LISTA_DATOS y se recibe en la entrada un *token* que no se puede equiparar.

3.2.2.3. Analizador Semántico

El analizador semántico se encarga de comprobar la correción del programa fuente introducido. El analizador sintáctico comprueba que la estructura del programa fuente es correcta, pero para él los *tokens* no tienen significado. Por ejemplo, sea la línea de código:

```
POP #3
```

La instrucción `POP` no admite como operando un inmediato. Sin embargo, el flujo de *tokens*:

```
<MNEMÓNICO1,POP>
<INMEDIATO_V,3>
```

cumpliría las reglas sintácticas generando el árbol de análisis sintáctico que se observa en la Figura 3.2.8:

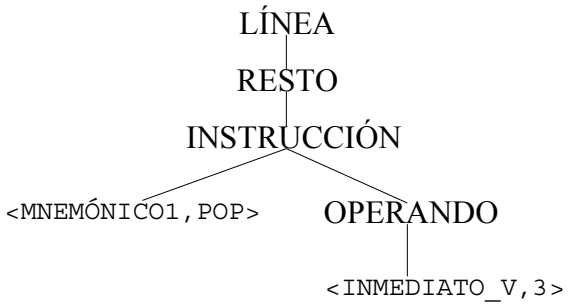


Figura 3.2.8. Ejemplo de árbol de análisis sintáctico

Es aquí donde entran en juego las comprobaciones semánticas, donde se tiene en cuenta el significado de los *tokens*. El analizador semántico debe comprobar que el modo de direccionamiento de los operandos introducidos está permitido para la instrucción que está siendo analizada. Para ver qué modos de direccionamiento se permiten en cada instrucción se puede consultar el juego de instrucciones, en el apartado 3.2.1.3, o la tabla contenida en el Manual de Usuario de *ENS2001*.

También debe comprobar el uso correcto de las pseudoinstrucciones. Por ejemplo, controlar que una pseudoinstrucción `ORG` no situa el comienzo del código fuera de la memoria, que una pseudoinstrucción `RES` no reserva más espacio en memoria que el restante hasta el final de la misma, que una pseudoinstrucción `EQU` no asigna a una etiqueta un valor mayor de 16 bits o que una pseudoinstrucción `DATA` no está almacenando datos fuera de la memoria. Asimismo, se debe controlar que no se definen etiquetas por duplicado.

Además de informar de los errores encontrados, se podría sugerir al usuario qué *token* estaba esperando el analizador cuando se produjo el error. Esto puede conseguirse también mediante el uso de acciones semánticas.

Por otro lado, las acciones semánticas también se emplean para calcular el valor de las expresiones aritméticas recogidas por la serie de reglas `EXPRESIÓN`, para reservar memoria, cambiar la dirección de memoria sobre la que se está generando el código máquina, detener el análisis, etc.

3.2.2.4. Generador de Código

El generador de código se va a encargar de recorrer la tabla de referencias adelantadas para resolver todas aquéllas que hayan quedado pendientes. Esto lo hará consultando la tabla de etiquetas que, una vez finalizado el análisis, y si el programa es correcto, contendrá el valor de todas ellas.

También será el encargado de devolver el código ensamblado o, en su defecto, la lista de errores encontrados en el código fuente.

3.2.2.5. Tabla de Etiquetas

La tabla de etiquetas sirve para almacenar los identificadores de las etiquetas que se encuentren definidas en el código fuente y el valor que vayan tomando en función de su

posición en el mismo o definido arbitrariamente mediante el uso de pseudoinstrucciones. De esta forma, cuando el generador de código vaya a resolver las referencias adelantadas que hayan quedado pendientes podrá encontrar el valor de las etiquetas en esta tabla.

3.2.2.6. Tabla de Referencias Adelantadas

La tabla de referencias adelantadas es una estructura en la que el ensamblador irá anotando posiciones del código en las cuales se encuentra una etiqueta cuyo valor es desconocido en ese momento. Al finalizar el proceso de análisis, el generador de código recorrerá esta estructura e irá completando la información, ya que en ese punto todas las etiquetas tienen (o deberían tener) valor.

Los datos relevantes para esta estructura son el identificador de la etiqueta que está sin resolver, la posición que ocupa dentro del código (número de instrucción), la posición que ocupa dentro de la instrucción (primer o segundo operando) y el tamaño de la referencia (como las etiquetas pueden hacer referencia a enteros o a desplazamientos, el ancho puede ser de 8 ó 16 bits). Es necesario guardar la posición de memoria y el desplazamiento ya que puede darse el caso de que en una palabra de 16 bits se almacenen dos operandos de 8 o menos bits.

3.2.2.7. Gestor de Errores

La tarea del gestor de errores será mantener una lista de los errores que se van produciendo, en la que almacenará el tipo de error que se produjo, una descripción del error, el carácter o conjunto de caracteres de entrada que lo produjo (si procede) y el número de línea en que se produjo el error (si procede). Al final del proceso, se devolverá el listado completo de errores producidos, si es que existen.

La responsabilidad de anotar los errores corresponde a todos los módulos implicados en el proceso de ensamblado.

Sólo se registrará un error por línea, lo cual parece adecuado teniendo en cuenta que las instrucciones del lenguaje ensamblador son cortas y no puede haber más de una instrucción por línea.

3.2.3. Análisis del Simulador

El análisis se dedica a la comprensión y modelado de la aplicación y del dominio en el cual funciona. La entrada inicial de la fase de análisis es una descripción del problema que hay que resolver y proporciona una visión general conceptual del sistema propuesto. Un diálogo subsiguiente con el cliente y un conocimiento de fondo del mundo real son entradas adicionales del análisis. La salida del análisis es un modelo formal que captura los tres aspectos esenciales del sistema: Los objetos y sus relaciones, el flujo dinámico de control, y la transformación funcional de datos que están sometidos a restricciones.

El proceso de construcción de dicho modelo no es lineal, sino que se va construyendo iterativamente, pasando de una parte a otra, completándolas mutuamente según se avanza en la comprensión del problema. No obstante, en este documento a la hora de presentar la solución final, se ha optado por detallar las tres visiones del modelo secuencialmente. Pero eso no quiere significar en ningún momento que se hayan construido en este orden.

Para efectuar el análisis del simulador se ha seguido la metodología descrita en [Rumbaugh 1996]. Todos los detalles de la misma han sido obtenidos de esta referencia.

3.2.3.1. Modelo de objetos

El primer paso para analizar los requisitos es construir un modelo de objetos, que muestra la estructura estática de datos correspondiente al sistema del mundo real, y la organiza en segmentos manejables describiendo clases de objetos del mundo real y sus relaciones entre sí. Para construir el modelo de objetos, se llevan a cabo los siguientes pasos, pudiéndose obviar alguno de ellos o simultanearlos si la complejidad que presentan es baja:

- Identificar los objetos y las clases.
- Preparar un diccionario de datos.
- Identificar las asociaciones entre objetos (incluyendo agregaciones).
- Identificar atributos de objetos y enlaces.
- Organizar y simplificar las clases de objetos empleando la herencia.
- Verificar la existencia de vías de acceso adecuadas para las posibles consultas.
- Iterar y refinar el modelo.
- Agrupar las clases en módulos.

A continuación se detallan las fases en las que se han tomado decisiones relevantes durante el análisis.

Identificación de las clases. Retención de las clases correctas

En un primer momento se efectúa una relación de objetos que participan en el modelo. En este caso, se trata de modelar el comportamiento de una computadora sencilla de arquitectura *Von Neumann* [De Miguel 1996], por tanto, una posible aproximación sería construir objetos que representen sus componentes significativos. Las características del modelo que se pretende simular vienen expuestas en el requisito ERS-REQ02.

El modelo se va a simular a nivel lógico, no a nivel físico. Directamente del requisito se pueden extraer las clases: `Procesador`, `Memoria`, `Banco de Registros`, `Biestables de Estado`, `Entrada/Salida`, `Unidad Aritmético-Lógica` (UAL) y `Generador de Excepciones`.

Sin embargo, no todos los objetos de esta lista formarán parte del modelo. Los biestables de estado pueden formar parte del banco de registros. La UAL y el generador de excepciones pueden estar integrados dentro del procesador, por lo que quedarían estas cuatro: `Procesador`, `Memoria`, `Banco de Registros` y `Entrada/Salida`.

La relación de multiplicidad entre los objetos es de 1 a 1, ya que sólo existe un objeto de cada clase en el modelo.

Las opciones de configuración determinan la forma en que se va a comportar la clase `Procesador` al efectuarse la simulación y, por tanto, serán atributos de dicha clase. Las variables de configuración que será necesario definir se extraen de los siguientes requisitos:

- ERS-REQ21. Dirección de Crecimiento de la Pila (`ModoPila`).

  Se trata de un selector que permite indicarle al simulador cuál será el comportamiento de la pila. Hay dos modos predefinidos:

  - Direcciones crecientes. La pila crece ascendentemente. Las instrucciones `PUSH` y `POP` se comportan de la siguiente manera:

PUSH	.SP ++ M(.SP) ← op1
POP	op1 ← M(.SP) .SP --

  - Direcciones decrecientes. La pila crece descendentemente. Las instrucciones `PUSH` y `POP` se comportan de la siguiente manera:

PUSH	M(.SP) ← op1 .SP --
POP	.SP ++ op1 ← M(.SP)

- ERS-REQ23. Ejecución Paso a Paso (`ModoEjecución`).

  Se trata de un selector que permite indicarle al simulador que ejecute las instrucciones una a una, deteniéndose después de cada una de ellas, o bien que comience a ejecutar hasta que encuentre un punto de ruptura (si están activos), una instrucción `HALT` en el código o bien se produzca una excepción.

  - Sí.
  - No.

- ERS-REQ27. Comprobar si PC invade el espacio de Pila (`ComprobarPC`).

  Se trata de un selector que permite indicarle al simulador que lance una excepción si el contador de programa se sitúa entre los límites superior e inferior de la pila o bien que ignore dicha situación.

  - Sí.
  - No.

- ERS-REQ28. Comprobar si SP invade el espacio de Código (`ComprobarPila`).

  Se trata de un selector que permite indicarle al simulador que lance una excepción si el puntero de pila se sitúa entre los límites superior e inferior de la zona perteneciente al código o bien que ignore dicha situación.

  - Sí.
  - No.

- ERS-REQ30. Inicializar los registros al ejecutar (`ReiniciarRegistros`).

  Se trata de un selector que permite indicarle al simulador si debe reiniciar el banco de registros al comenzar la ejecución de un programa. No obstante, sólo se reiniciará tras finalizar una ejecución anterior, esto es, cuando el biestable `H` esté activo (en otras palabras, si la última instrucción que se ejecutó fue `HALT`).

  - Sí.
  - No.

- ERS-REQ32. Modo de Depuración (`ModoDepuración`).

  Se trata de un selector que permite indicarle al simulador que se detenga cuando encuentre un punto de ruptura o bien que los ignore por completo.

  - Sí.
  - No.

Por último, el usuario ha de interactuar con el sistema a través de dos interfaces, una en modo textual, modelada por la clase `Interfaz Texto`, y otra gráfica, modelada por la clase `Interfaz Gráfica`. Una de las opciones de configuración está íntimamente ligada a su comportamiento. Se trata de la opción que permite definir la base numérica en la que se presentarán los datos enteros, extraída del siguiente requisito:

- ERS-REQ10. Base de Representación Numérica (`BaseNumérica`).

  Se trata de un selector que permite indicarle al simulador la base de representación numérica que empleará al visualizar los números enteros, bien decimal, bien hexadecimal.

  - Base 10 (decimal).
  - Base 16 (hexadecimal).

Diccionario de datos

Para construir el diccionaro de datos se debe escribir un párrafo que describa con precisión cada clase de objetos. Se describe el alcance de la clase dentro del problema estudiado. El diccionario de datos se presenta en la Tabla 3.2.5.

Procesador	Esta clase se encarga de modelar el comportamiento del procesador del sistema. Se encarga de simular la ejecución del juego de instrucciones, accediendo a los componentes del sistema (memoria, registros y entrada/salida) cuando sea necesario, así como de generar las posibles excepciones que se produzcan. También se encarga de almacenar las diferentes variables de configuración de las que dispone el simulador para definir su comportamiento.
Memoria	Esta clase se encarga de modelar el comportamiento de la memoria del sistema. Permite acceder y modificar los valores contenidos en las celdas de memoria, así como definir puntos de ruptura incondicionales en cada una de las posiciones.
Banco de Registros	Esta clase se encarga de modelar el comportamiento del banco de registros del sistema. Permite acceder y modificar los valores contenidos en los registros y consultar el valor de los biestables de estado por separado.

Entrada/Salida	Esta clase se encarga de modelar la interfaz de entrada/salida del sistema. Permite simular la interacción con el exterior del sistema mediante una consola de texto. Posee un *buffer* intermedio de lectura/escritura, que da soporte a distintos tipos de datos (enteros, caracteres y cadenas).
Interfaz Texto	Esta clase se encarga de modelar la interfaz entre el usuario y el simulador en modo texto. Permite acceder a toda la operativa de la aplicación.
Interfaz Gráfica	Esta clase se encarga de modelar la interfaz entre el usuario y el simulador en modo gráfico. Permite acceder a toda la operativa de la aplicación.

Tabla 3.2.5. Diccionario de Datos

Asociaciones entre clases. Retención de las asociaciones correctas

Toda dependencia entre dos o más clases es una asociación, y una referencia de una clase a otra también lo es. La implementación de las asociaciones debe mantenerse fuera del análisis, para mantener la libertad de diseño.

A continuación, se deben descartar las asociaciones innecesarias e incorrectas, como aquéllas entre clases eliminadas, irrelevantes o de implementación, acciones, asociaciones ternarias, asociaciones derivadas, asociaciones de nombre incorrecto, nombres de rol, asociaciones cualificadas, multiplicidad y asociaciones inexistentes.

Todas las asociaciones presentes en el modelo son de uso o acceso y todas son de multiplicidad uno a uno. Se identifican las siguientes relaciones:

- El objeto de clase `Procesador` accede al objeto de clase `Memoria` para leer y almacenar datos en las celdas de memoria.
- El objeto de clase `Procesador` accede al objeto de clase `Banco de Registros` para leer y almacenar datos en los registros y biestables de estado.
- El objeto de clase `Procesador` accede al objeto de clase `Entrada/Salida` para leer y escribir datos en la consola de usuario.
- El objeto de clase `Interfaz Texto` accede al resto de clases para realizar consultas, modificaciones de datos o lanzar la simulación.
- El objeto de clase `Interfaz Gráfica` accede al resto de clases para realizar consultas, modificaciones de datos o lanzar la simulación.

Atributos identificativos. Retención de los atributos correctos

A continuación, se identifican los atributos de los objetos. Los atributos son propiedades de objetos individuales. No deben ser objetos (se debe utilizar una asociación para mostrar relaciones entre objetos). Por otra parte, los atributos derivados deben ser omitidos o bien deben ser marcados claramente. Los atributos de enlace también deben ser identificados.

Se deben eliminar posteriormente aquellos atributos innecesarios o incorrectos, como aquéllos que resulten ser objetos, cualificadores, nombres, identificadores, atributos de enlace, valores internos, detalles finos y atributos discordantes.

Se identifican los atributos presentados en la Tabla 3.2.6:

Clase	Atributos
Procesador	ModoPila ModoEjecución ComprobarPC ComprobarPila ReiniciarRegistros ModoDepuración
Banco de Registros	Registros Biestables
Memoria	Celdas PuntosDeRuptura
Entrada/Salida	Buffer
Interfaz Texto	BaseNumérica
Interfaz Gráfica	BaseNumérica

Tabla 3.2.6. Atributos del modelo de objetos

Refinamiento mediante herencia

El siguiente paso consiste en organizar las clases empleando la herencia para compartir una estructura común, bien generalizando aspectos comunes en una superclase, bien refinando las clases existentes para dar lugar a subclases especializadas. En este caso, se agrupan las características comunes de las dos interfaces de usuario, textual y gráfica, en una superclase denominada `Interfaz Usuario`, de la que heredan `Interfaz Texto` e `Interfaz Gráfica`. Por tanto, la clase madre contendrá el atributo `BaseNumérica`, común para sus hijas.

Comprobación de vías de acceso

Hay que seguir las vías de acceso por el diagrama del modelo de objetos para ver si tiene unos resultados sensatos. El modelo debe responder a las preguntas que se propongan sobre él. Si no las responde, es posible que haya una falta de información. En este caso, todas las clases están conectadas entre sí, por lo que no se presentan problemas de accesibilidad.

Agrupamiento de clases en módulos

El último paso del modelado consiste en agrupar las clases en módulos. En la aproximación dada durante el análisis no es necesario.

Tras concluir todos los pasos de la fase de análisis, se llega al modelo de objetos que se muestra en la Figura 3.2.9.

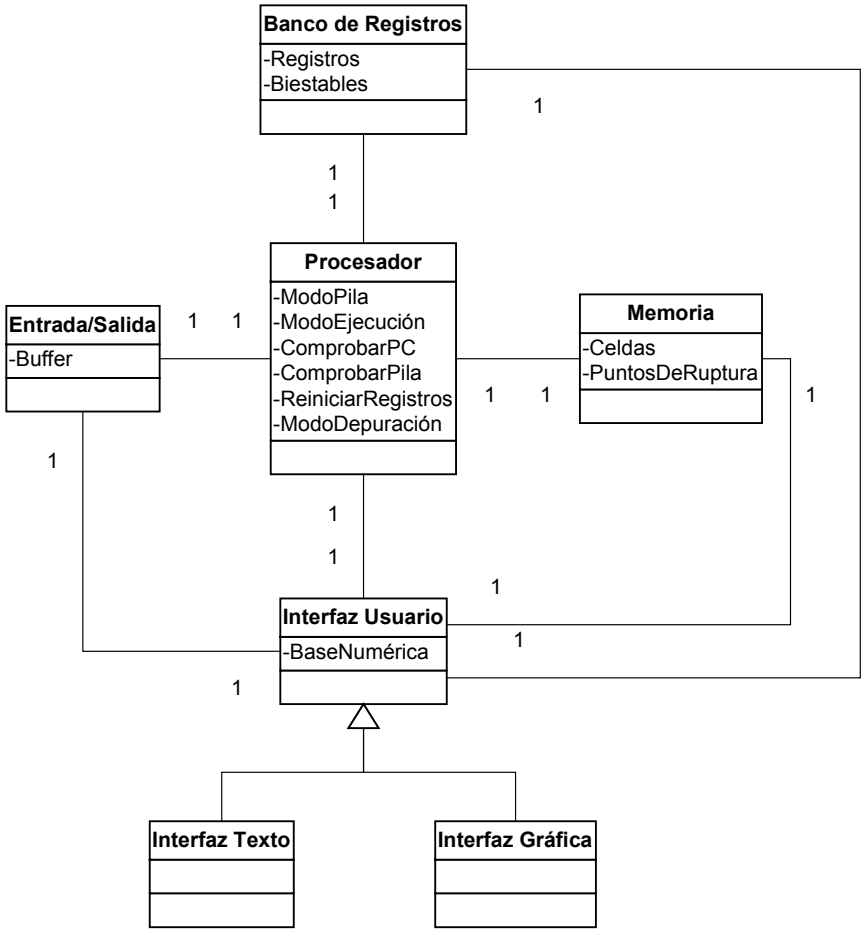


Figura 3.2.9. Modelo de Objetos del Simulador (Análisis)

3.2.3.2. Modelo dinámico

El modelo dinámico muestra la forma en que el comportamiento del sistema y de los objetos de que consta va variando con el tiempo. Es un modelo importante para los sistemas interactivos, como lo es *ENS2001*. Para construir el modelo dinámico se llevan a cabo los pasos siguientes:

- Se preparan escenarios de secuencias típicas de interacción.
- Se identifican sucesos que actúen entre objetos.
- Se prepara un seguimiento de sucesos para cada escenario.
- Se construye un diagrama de estados.
- Se comparan los sucesos intercambiados entre objetos para verificar la congruencia.

Preparación de escenarios

Un escenario es una secuencia de sucesos. Los sucesos se producen cuando se intercambia información entre un objeto del sistema y un agente externo, como puede ser el usuario. Los valores de información intercambiados son los parámetros del suceso.

Para cada suceso hay que identificar el actor que haya dado lugar al mismo y también los parámetros del suceso.

En esta visión del modelo se puede apreciar cómo se produce la interacción de los objetos en el tiempo. No se van a reproducir todos los casos de interacción posibles, pero sí los más significativos, de los que se ha extraído la información más relevante a la hora de construir el modelo.

Escenario 1. Operación Ejecutar

Desde la clase `Interfaz Usuario` se invocará al método `Ejecutar` de la clase `Procesador`, que desencadenará toda la acción, cuyos efectos se podrán apreciar al finalizar la ejecución (o durante la misma, si se produce alguna operación de Entrada/Salida), y que devolverá un código con la causa que la dio fin mediante un mensaje de `FinEjecución` (Figura 3.2.10).

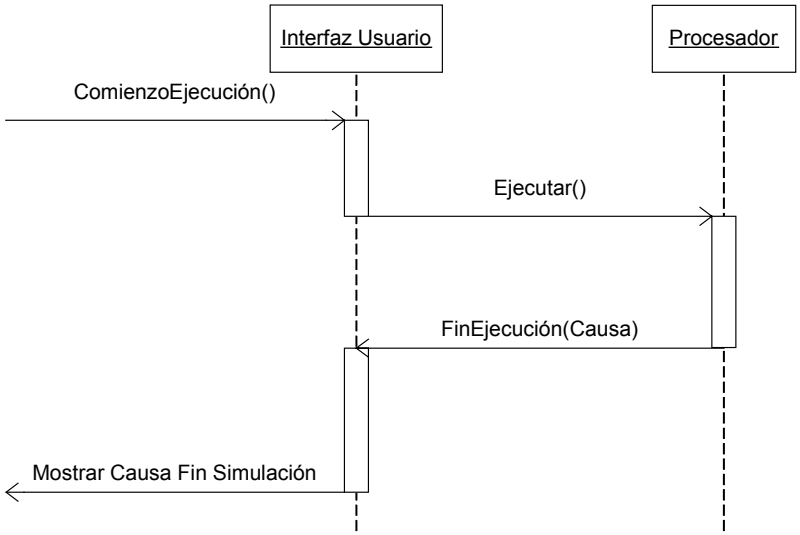


Figura 3.2.10. Operación Ejecutar

Otras operaciones, tan sencillas como ésta, que se pueden realizar desde la interfaz de usuario son aquéllas que permiten consultar o modificar el estado de la memoria, el banco de registros, o las propias opciones de configuración. Basta con invocar desde la interfaz de usuario a los métodos `Leer` o `Escribir` de la clase correspondiente.

Escenario 2. Operación Leer Memoria

La operación para leer una posición de memoria (Figura 3.2.11) consistirá en invocar al método `Leer` de la clase `Memoria` con la dirección cuyo valor se quiere consultar. Para leer bloques de memoria, bastará invocar a este método con direcciones correlativas. El método devuelve el valor almacenado, o bien un error si la dirección que se pretende consultar no es válida.

Esta operación puede efectuarse desde varias clases, como `Procesador` o `Interfaz Usuario`. Por ello, no se detalla ninguna en especial en el diagrama de secuencia que representa este escenario.

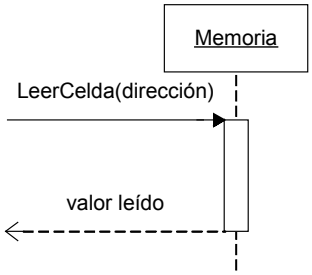


Figura 3.2.11. Operación Leer Memoria

Escenario 3. Operación Activar Punto de Ruptura

Desde la interfaz de usuario se pueden definir puntos de ruptura (Figura 3.2.12). Para ello, se invoca al método `EscribirPuntoRuptura` de la clase `Memoria`, indicando la dirección que se quiere editar, y si se quiere activar o desactivar el punto de ruptura.

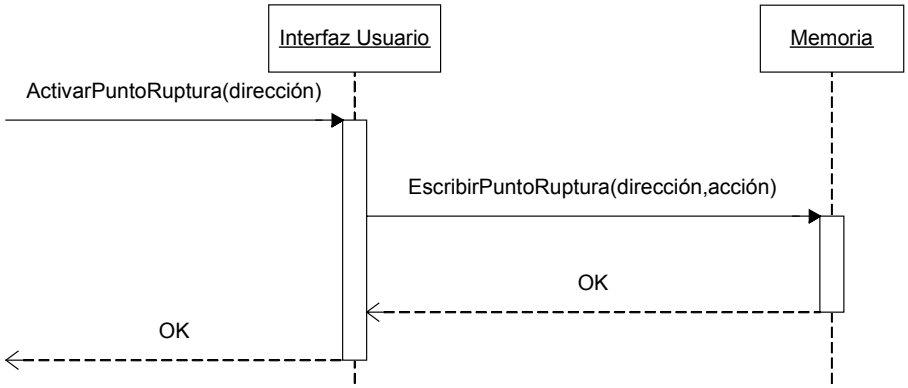


Figura 3.2.12. Operación Activar Punto de Ruptura

Escenario 4. Operación Escribir Registro

La operación para escribir un registro (Figura 3.2.13) consistirá en invocar al método `Escribir` de la clase `Banco de Registros`, indicando qué registro se quiere actualizar y su nuevo valor. Suponiendo que ambos datos son válidos, se realizará la acción correspondiente. Si no, se devolverá un error.

Esta operación puede efectuarse desde varias clases, como `Procesador` o `Interfaz Usuario`. Por ello, no se detalla ninguna en especial en el diagrama de secuencia que representa este escenario.

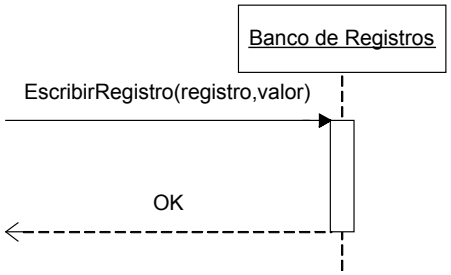


Figura 3.2.13. Operación Escribir Registro

Escenario 5. Operación Leer Variable de Configuración

La operación para leer una variable de configuración (Figura 3.2.14) es análoga a las anteriores de lectura de un registro o una posición de memoria. Desde la interfaz de usuario se invocará al método `LeerConfiguración`, indicando qué variable se quiere leer. Dependiendo de qué variable se trate, se invocará al método de lectura del atributo correspondiente de la clase `Procesador` o se recuperará directamente el valor si se trata de la variable `BaseNumérica`. En el ejemplo de la figura, se supone que `id_variable==ModoDepuración`, por lo que se lee dicho atributo de la clase `Procesador`.

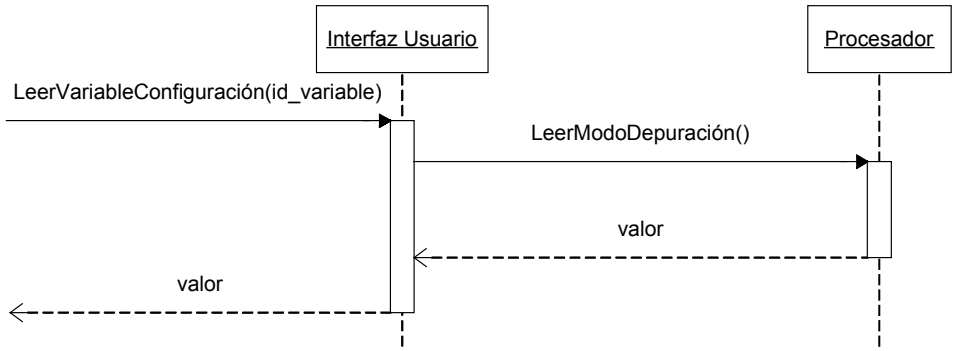


Figura 3.2.14. Operación Leer Variable de Configuración

Escenario 6. Operación Guardar Configuración en Disco

La operación para guardar la configuración en un fichero de disco (Figura 3.2.15) consiste en recuperar el valor de todas las variables de configuración desde la clase `Interfaz Usuario`, leer el propio atributo `BaseNumérica` y almacenar los valores leídos en un fichero en el disco.

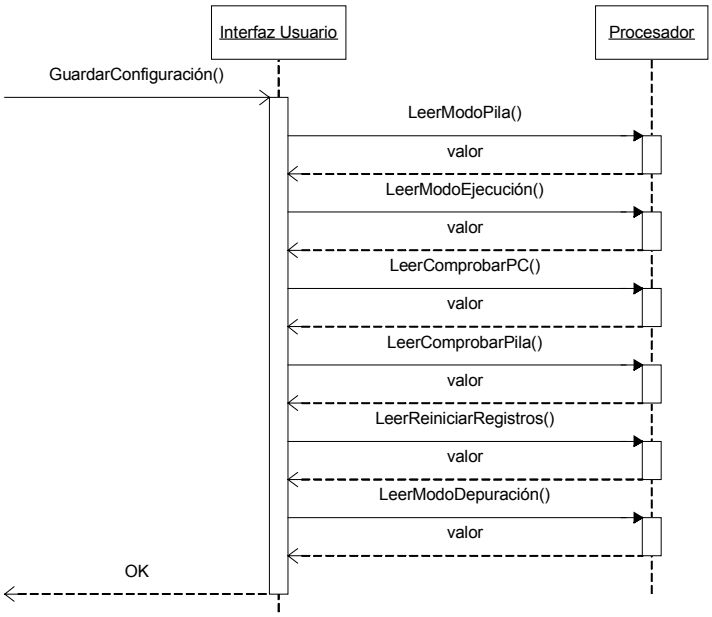


Figura 3.2.15. Operación Guardar Configuración en Disco

Escenario 7. Simulación. *Fetch* de la instrucción

El diagrama más complejo se obtiene al ilustrar el comportamiento de la clase `Procesador` al simular la ejecución de una instrucción. Se van a proponer a continuación varias posibilidades que pueden ocurrir durante la simulación.

La fase de *fetch* siempre es igual (Figura 3.2.16). El procesador lee el valor del contador de programa. A continuación, comprueba que en la dirección a la que apunta `PC` no hay ningún punto de ruptura activo. Después, lee la palabra contenida en la posición de memoria a la que apunta `PC`. En esa palabra está contenido el código de operación y los modos de direccionamiento de los operandos. Con esta información, se decodifica la instrucción.

Si se hubiera encontrado un punto de ruptura activo, la ejecución se detendría y se informaría que la causa ha sido un punto de ruptura activo.

La fase de *fetch* de los operandos varía, ya que pueden darse muy variados casos, desde que no haya ningún operando, hasta dos, teniendo en cuenta los posibles modos de direccionamiento para cada uno de ellos y todas sus combinaciones. A continuación, se detalla el diagrama para cada modo de direccionamiento concreto.

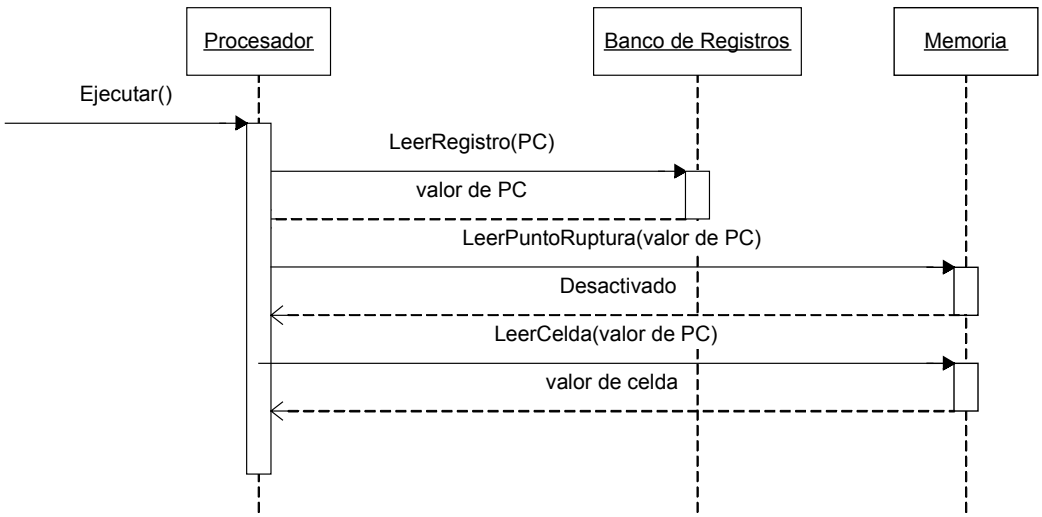


Figura 3.2.15. *Fetch* de la instrucción

Escenario 8a. Simulación. Acceso a operando con direccionamiento inmediato

El direccionamiento inmediato (Figura 3.2.17) consiste únicamente en leer de la memoria el valor del operando.

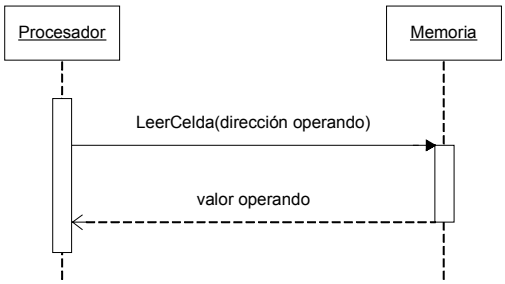


Figura 3.2.17. Direccionamiento inmediato

Escenario 8b. Simulación. Acceso a operando con direccionamiento directo a memoria

El direccionamiento directo a memoria (Figura 3.2.18) requiere dos accesos, el primero para recuperar la dirección de memoria donde se encuentra el operando (un puntero al operando) y el segundo para recuperar el operando en sí.

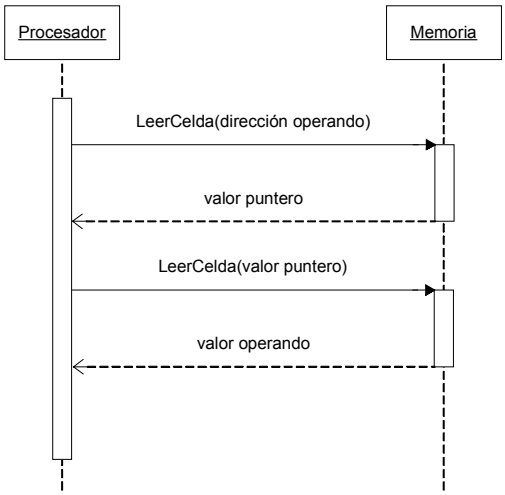


Figura 3.2.18. Direccionamiento directo a memoria

Escenario 8c. Simulación. Acceso a operando con direccionamiento directo a registro

El direccionamiento directo a registro (Figura 3.2.19) consta de un acceso a memoria, para recuperar el identificador de registro y un acceso al banco de registros para recuperar el valor del operando.

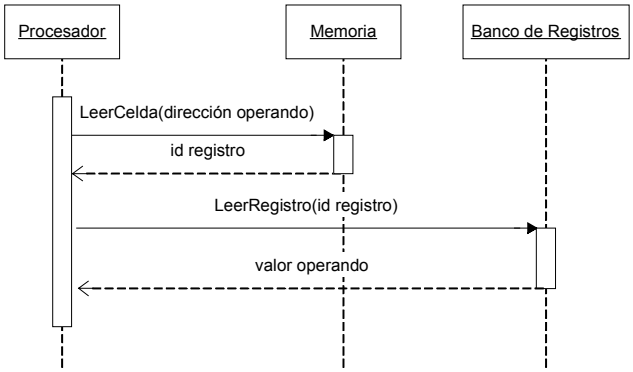


Figura 3.2.19. Direccionamiento directo a registro

Escenario 8d. Simulación. Acceso a operando con direccionamiento indirecto

El direccionamiento indirecto (Figura 3.2.20) consta de un acceso a memoria para recuperar el identificador del registro que ejerce función de puntero, otro al banco de registros para recuperar el valor del registro y un último acceso a memoria para leer el valor del operando.

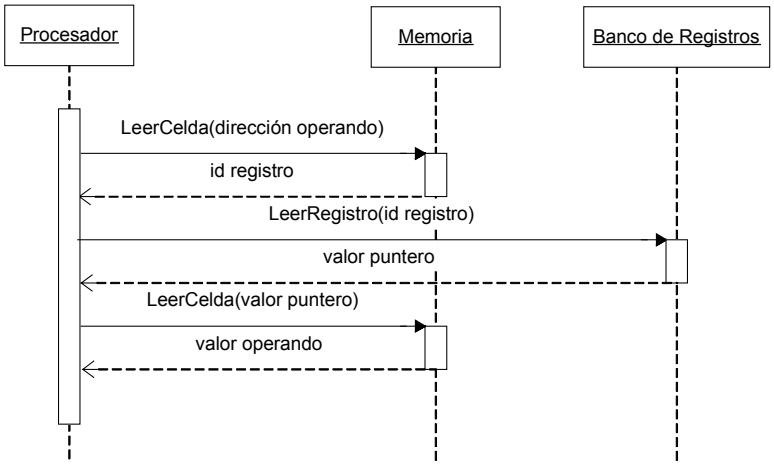


Figura 3.2.20. Direccionamiento indirecto

Escenario 8e. Simulación. Acceso a operando con direccionamiento relativo a índice

El direccionamiento relativo a registros índice (Figura 3.2.21) es semejante al direccionamiento indirecto. En el primer acceso a memoria se recupera el valor del desplazamiento y, más tarde, antes de acceder al valor del operando, hay que añadir dicho desplazamiento al valor del registro.

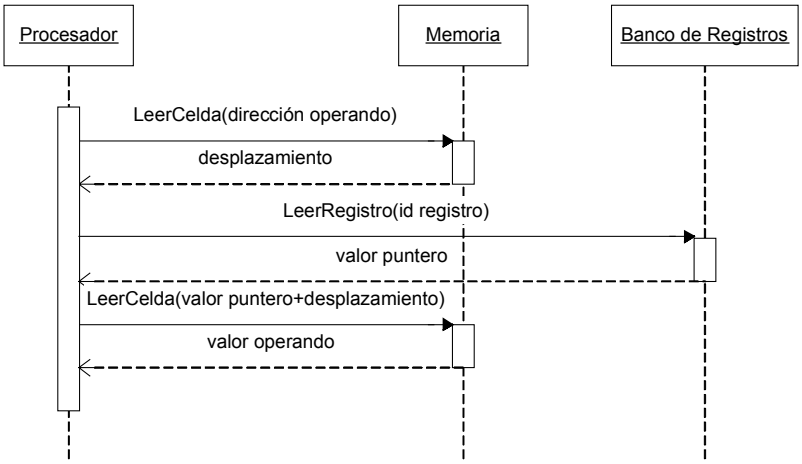


Figura 3.2.21. Direccionamiento relativo a registros índice

Escenario 8f. Simulación. Acceso a operando con direccionamiento relativo a PC

El direccionamiento relativo a contador de programa (Figura 3.2.22) consta de un acceso al banco de registros para recuperar el valor del contador de programa, un acceso a memoria para leer el valor del desplazamiento, añadir dicho desplazamiento al contador de programa y actualizar su valor.

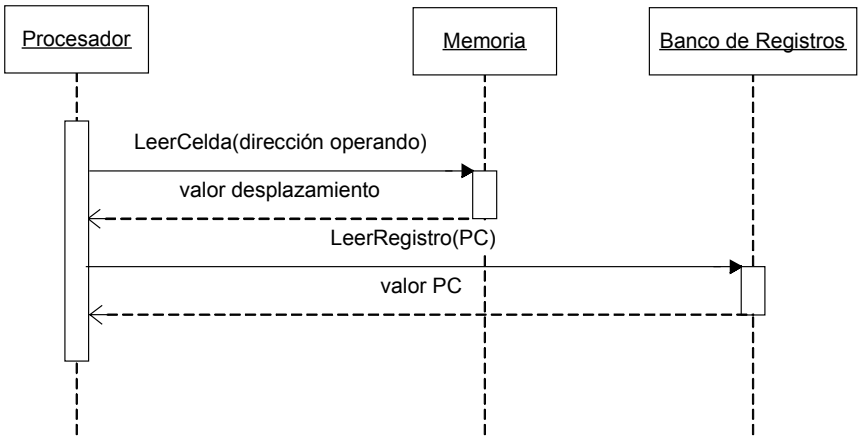


Figura 3.2.22. Direccionamiento relativo a PC

Escenario 9. Simulación. Actualización del contador de programa

Tras la fase de *fetch* de operandos, y antes de ejecutar la instrucción, se guarda en el banco de registros el nuevo valor del contador de programa (Figura 3.2.23). Este nuevo valor dependerá de la instrucción y los operandos que se hayan leído.

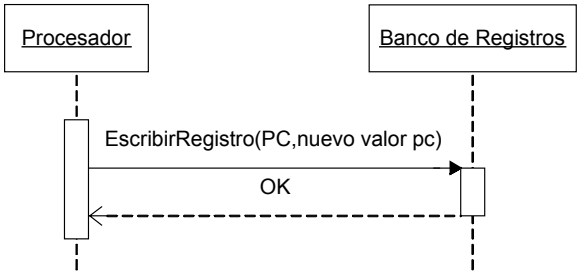


Figura 3.2.23. Actualización del contador de programa

Durante la fase de ejecución, el procesador efectuará cálculos con los operandos, como pueden ser operaciones aritméticas, lógicas, comparaciones, etc. En otras instrucciones, la ejecución requerirá que se escriban valores en memoria o en el banco de registros, siguiendo el mismo esquema que se acaba de desarrollar. Únicamente en la ejecución de las instrucciones de entrada/salida se va a seguir un esquema distinto, ya que se llama a los métodos de otra de las clases. A continuación se ilustra la ejecución de dos instrucciones de este grupo, la primera de entrada y la segunda de salida.

Escenario 10. Simulación. Instrucción `INCHAR`

Para simular la instrucción `INCHAR /0x1000` (Figura 3.2.24), primero se invoca al método `LeerBuffer` de la clase `Entrada/Salida`. El valor del *buffer* se deberá obtener desde la clase `Interfaz Usuario`, invocando el método `LeerConsola`. Se obtendrá una cadena de caracteres, que debería contener un único carácter. Se toman los 8 bits que conforman el

carácter, los 8 bits superiores se dejan a cero y se escribe el valor obtenido en la memoria en la posición 1000h.

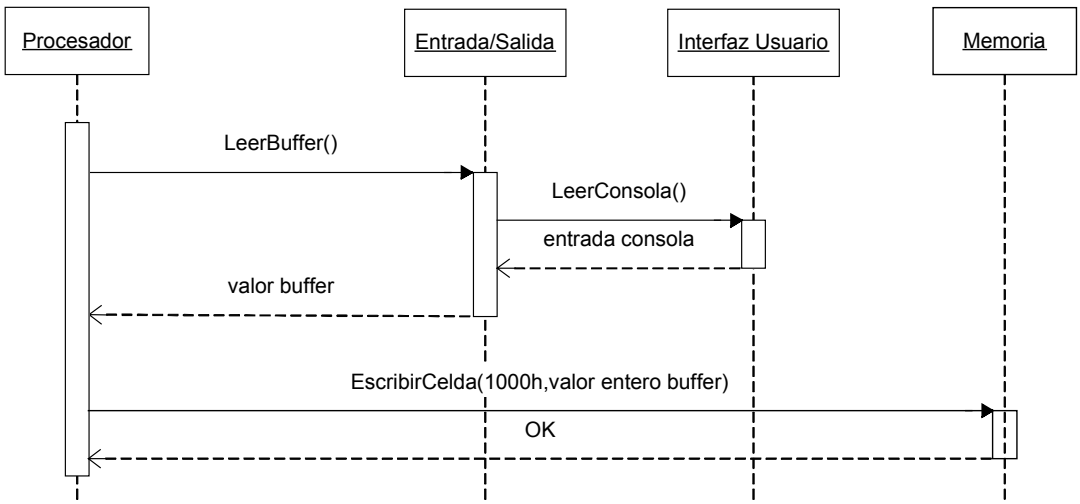


Figura 3.2.24. Instrucción `INCHAR /0x1000`

Escenario 11. Simulación. Instrucción `WRINT`

Para simular la instrucción `WRINT #0xFF` (Figura 3.2.25), se invoca al método `EscribirBuffer` de la clase `Entrada/Salida` con el entero que se quiere escribir como parámetro. Desde ahí, se invocará al método `EscribirConsola` de la clase `Interfaz Usuario` quien, según el valor del atributo `BaseNumérica`, mostrará por consola el valor en formato decimal o hexadecimal.

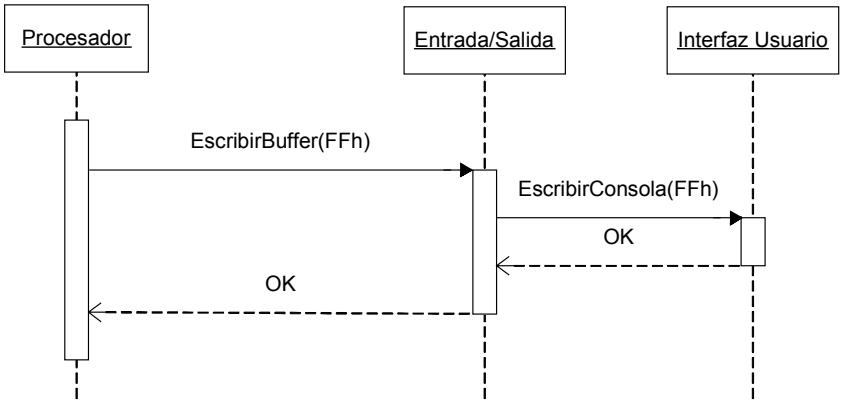


Figura 3.2.25. Instrucción `WRINT #0xFF`

Durante la ejecución, el comportamiento del método `Ejecutar` de la clase `Procesador` vendrá dado por el valor de sus atributos, que son las variables de configuración del simulador. Por ejemplo, antes de proceder a ejecutar la siguiente instrucción, siempre que no se hubiera producido ninguna condición de parada hasta ahora, el procesador debe verificar que en la siguiente posición de memoria no hay ningún punto de ruptura activo. Esta operación sería análoga al caso estudiado en el Escenario 8a, pero invocando al método `LeerPuntoRuptura(dirección)` en vez del método `LeerCelda(dirección)`.

Todo este tipo de comprobaciones no dan lugar a escenarios descriptivos en sí mismos, ya que se producen dentro de la misma clase.

Identificación de sucesos

Mediante un examen detallado de los escenarios se pueden identificar todos los sucesos externos. Deben utilizarse escenarios para hallar los sucesos normales, pero sin olvidar posibles condiciones de error y sucesos poco corrientes.

Se deben agrupar bajo un mismo nombre aquellos sucesos que tengan el mismo efecto sobre el flujo de control, incluso si los valores de sus parámetros son distintos.

En este caso, todos los sucesos externos serán recogidos por la clase `Interfaz Usuario`.

La lista completa de métodos identificados se muestra en la Tabla 3.2.7. Como se puede observar, la gran mayoría de ellos son métodos de acceso a los atributos de las clases (para lectura o escritura).

Clase	Métodos
Procesador	LeerModoPila LeerModoEjecución LeerComprobarPC LeerComprobarPila LeerReiniciarRegistros LeerModoDepuración EscribirModoPila EscribirModoEjecución EscribirComprobarPC EscribirComprobarPila EscribirReiniciarRegistros EscribirModoDepuración Ejecutar
Banco de Registros	LeerRegistro EscribirRegistro LeerBiestable EscribirBiestable
Memoria	LeerCelda EscribirCelda LeerPuntoRuptura EscribirPuntoRuptura
Entrada/Salida	LeerBuffer EscribirBuffer
Interfaz Usuario	ComienzoEjecución FinEjecución ActivarPuntoRuptura DesactivarPuntoRuptura LeerRegistro EscribirRegistro LeerBiestable EscribirBiestable LeerCeldaMemoria EscribirCeldaMemoria LeerVariableConfiguración EscribirVariableConfiguración GuardarConfiguración CargarConfiguración LeerConsola EscribirConsola

Tabla 3.2.7. Métodos del modelo de objetos

Construcción de un diagrama de estados

Se debe construir un diagrama de estados para todas aquellas clases cuyo comportamiento dinámico no sea trivial. Todo escenario se corresponde con un camino a través del diagrama de estados.

El diagrama de estados de una clase estará terminado cuando abarque todos los escenarios y cuando maneje todos los sucesos que puedan afectar a un objeto de esta clase en todos y cada uno de sus estados. No todas las clases necesitan un diagrama de estados.

En este caso tan sólo la clase `Procesador` presenta un comportamiento dirigido por una secuencia de estados. El diagrama de estados para dicha clase se muestra en la Figura 3.2.26.

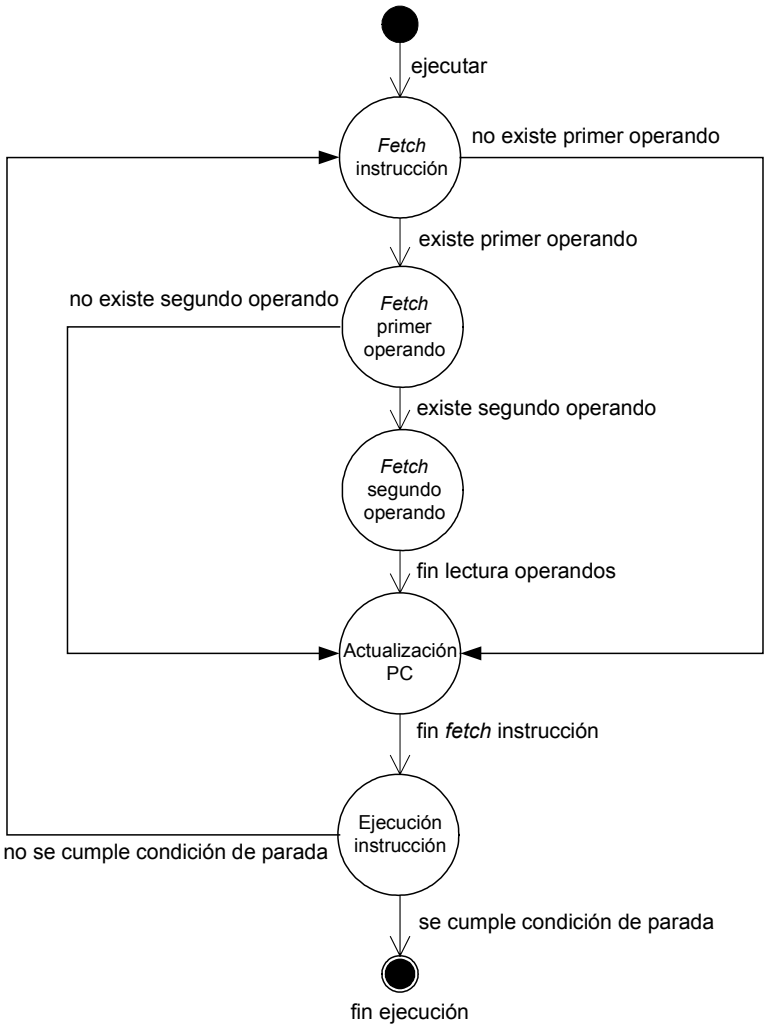


Figura 3.2.26. Diagrama de estados de la clase Procesador

3.2.3.3. Modelo funcional

El modelo funcional muestra la forma en que se calculan los valores, sin tener en cuenta las secuencias, decisiones o estructura de los objetos. El modelo funcional muestra qué valores dependen de qué otros valores y las funciones que los relacionan.

Los procesos de un *DFD* se corresponden con actividades o acciones de los diagramas de estados de las clases.

Para construir el modelo funcional se deberían seguir estos pasos:

- Identificar los valores de entrada y de salida
- Construir diagramas de flujo de datos que muestren las dependencias funcionales
- Describir las funciones
- Identificar las restricciones
- Especificar los criterios de optimización

Sin embargo, en la aplicación que se está construyendo no se produce apenas ninguna transformación de los datos. La gran parte de la simulación se encarga simplemente de mover datos de un lugar a otro. Por ello, el modelo dinámico tiene mayor importancia, mientras que los diagramas de flujo de datos se limitarían, si acaso, a representar los métodos como funciones de acceso a los atributos (Figura 3.2.27), bien para recuperar su valor, bien para modificarlo.

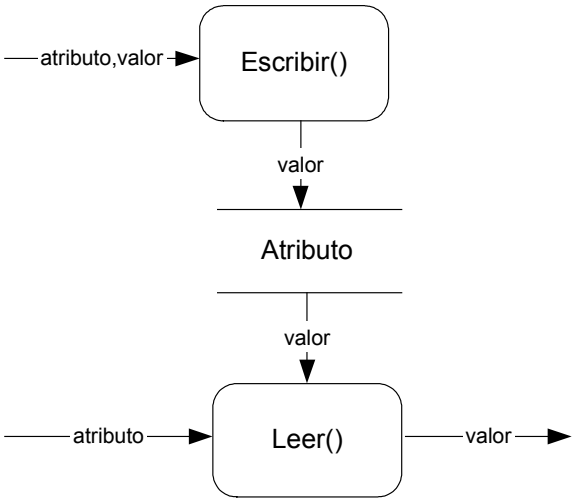


Figura 3.2.27. Acceso genérico a los atributos

El método `Ejecutar`, de la clase `Procesador`, consiste en una secuencia de métodos de acceso a los atributos del resto de las clases, y dicha secuencia se puede comprender perfectamente tras estudiar el modelo dinámico. La única transformación de datos se efectúa al simular instrucciones aritméticas o lógicas, pero son tan sencillas que no requieren el uso de almacenes. Simplemente se leen dos operandos, se aplica una operación y se almacena el resultado, bien en la memoria, bien en el banco de registros.

3.3. Diseño del Sistema

Nuevamente, se ha separado en varias partes el diseño del sistema *ENS2001*. En esta ocasión, los apartados se centran en el ensamblador, el simulador, la interfaz en modo texto y la interfaz gráfica.

3.3.1. Diseño del Ensamblador

En este apartado se detalla la labor de diseño para las distintas partes en los que se divide el ensamblador: analizador léxico, sintáctico, semántico, generador de código y las estructuras auxiliares que los apoyan. A diferencia de la fase de análisis, en este caso el análisis sintáctico, semántico y generación de código se tratará como una unidad, ya que su funcionamiento se halla entremezclado.

3.3.1.1. Analizador Léxico

A partir de la gramática construida durante el análisis se va a mostrar el Autómata Finito Determinista (en adelante AFD) que reconoce el lenguaje de entrada para el analizador léxico. A este autómata se le añadirán las acciones semánticas necesarias para generar la información que recibirá el analizador sintáctico. Por último, se mencionan los posibles errores que puede generar el ensamblador en esta fase.

El AFD se va a ir presentando por partes, ya que en su conjunto resulta bastante extenso. El estado 0 es siempre el estado inicial. Respecto a los caracteres de entrada, se representan mediante un tipo de letra monoespaciado, mientras que para los casos especiales se seguirá una simbología semejante a la empleada durante el análisis:

`letra = [a..zA..Z_]`
`dígito = [0..9]`
`dígito1 = [1..9]`
`dígito hex = [0..9a..f] ∪ [A..F]`
`carácter =` cualquier carácter imprimible menos el carácter comillas (`"`)

Los caracteres delimitadores son espacio, tabulador, fin de línea, fin de fichero, coma, dos puntos, +, -, *, /, %, (y), salvo que se indique expresamente lo contrario.

Después de cada parte del AFD se enumeran las acciones semánticas efectuadas en cada estado (indicado entre paréntesis). Se da por supuesto que los caracteres leídos se van concatenando en una variable llamada `palabra`. El número de líneas leídas se almacena en la variable `nlin` y se inicializa a 0 al comenzar el análisis. Por su parte, el carácter leído se almacena en la variable `entrada`.

También se da por supuesto que la operación + indica suma entera cuando los operandos son enteros y ⊕ indica concatenación de cadenas cuando son caracteres o cadenas de caracteres.

A continuación se describen las funciones auxiliares empleadas en las acciones semánticas del AFD:

- `ValorEntero(carácter)` devuelve el valor decimal del dígito representado por dicho carácter, que podrá ser un dígito decimal o hexadecimal

- `EnviarToken(palabra, atributos)` envía el *token* reconocido al analizador sintáctico.
- `EnviarError(id_error, línea, palabra)` envía al gestor de errores el error detectado, indicando en qué línea se encontró y la cadena de entrada que lo produjo.
- `EsPalabraClave(palabra)` devuelve `cierto` si la `palabra` es una palabra clave de la Tabla 3.2.3 (tabla de palabras clave).
- `TipoToken(texto)` devuelve el valor contenido en la columna Tipo de Token de la Tabla 3.2.3 para la fila cuya palabra clave coincide con `texto`.
- `Atributo(texto)` devuelve el valor contenido en la columna Atributo de la Tabla 3.2.3 para la fila cuya palabra clave coincide con `texto`.

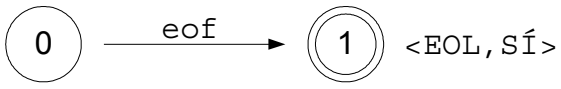
En el estado 0 se ejecutan las siguientes acciones semánticas:

```
valor = 0; negativo = falso; palabra = "";
```

Siempre que en una transición el símbolo terminal leído de la entrada no sea un carácter delimitador (del conjunto de caracteres que se ha definido anteriormente), se avanzará hacia el siguiente carácter de la entrada:

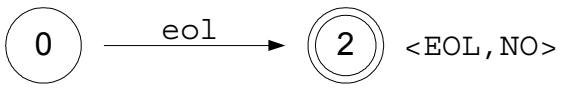
```
Leer(entrada);
```

- Fin de fichero.



(1)
```
línea ++;
EnviarToken(EOL,SÍ);
```

- Fin de línea.

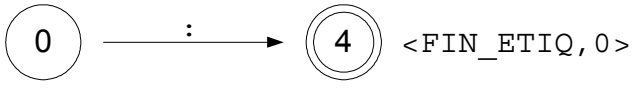


(2)
```
línea ++;
EnviarToken(EOL,NO);
```

- Coma (separador de operandos).

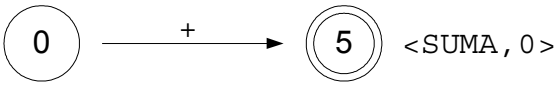
(3)
```
EnviarToken(SEPARADOR,0);
```

- Dos puntos (fin de etiqueta).



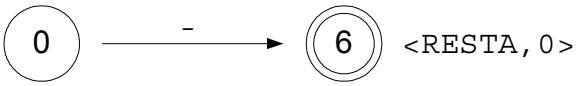
(4)
```
EnviarToken(FIN_ETIQ,0);
```

- Suma.



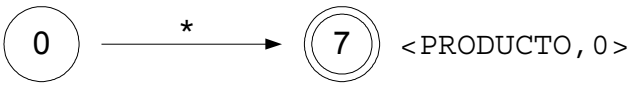
(5) `EnviarToken(SUMA,6);`

- Resta.



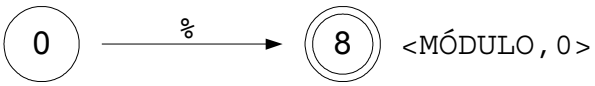
(6) `EnviarToken(RESTA,0);`

- Producto.



(7) `EnviarToken(PRODUCTO,0);`

- Módulo.

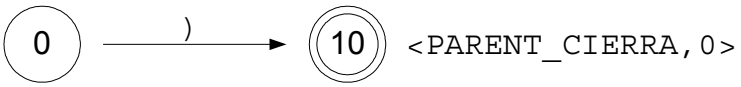


(8) `EnviarToken(MÓDULO,0);`

- Apertura de paréntesis.

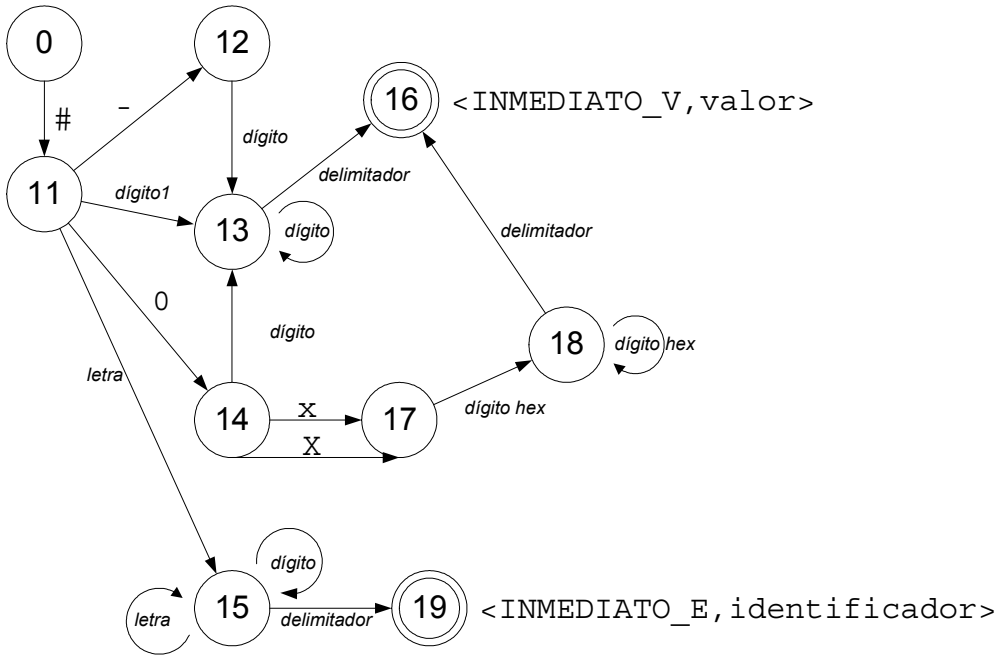
(9) `EnviarToken(PARENT_ABRE,0);`

- Cierre de paréntesis.



(10) `EnviarToken(PARENT_CIERRA,0);`

- Operandos con modo de direccionamiento inmediato.

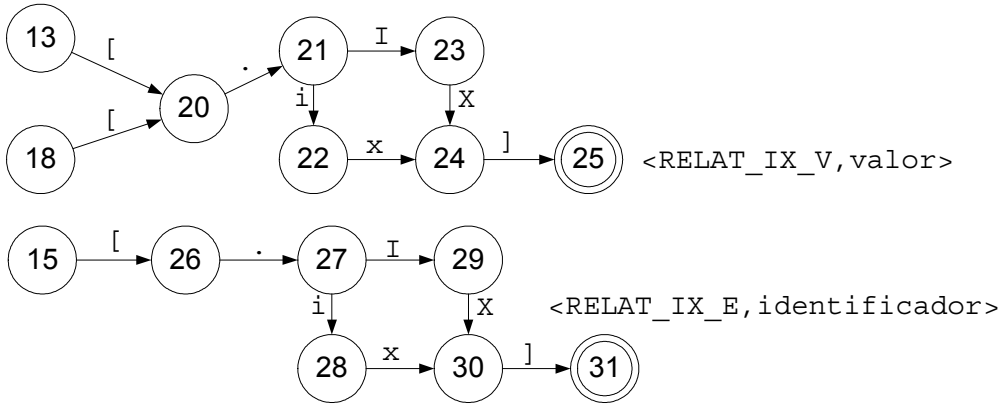


Transición	Acciones semánticas
11→12	`negativo = cierto;`
11→13 12→13 13→13 14→13	`valor = valor * 10 + ValorEntero(entrada);`
11→15 15→15	`identificador = identificador ⊕ entrada;`

13→16
18→16
	`SI (negativo) ENTONCES valor = - valor;`
	`SI (valor < -32768) O (valor > 65535)`
	`ENTONCES EnviarError (ENTERO_FUERA_DE_RANGO, línea+1, token);`
	`SI NO EnviarToken(INMEDIATO_V,valor);`

17→18 18→18	`valor = valor*16 + ValorEntero(entrada);`
15→19	`EnviarToken(INMEDIATO_E,identificador);`

- Operandos con modo de direccionamiento relativo a registro índice IX.

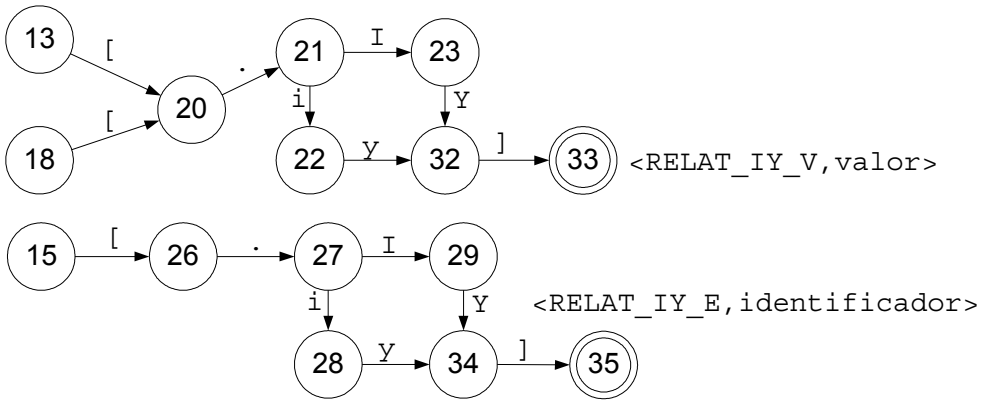


Transición	Acciones semánticas

24→25	`SI (negativo) ENTONCES valor = - valor;`
	`SI (valor < -128) O (valor > 127)`
	`ENTONCES EnviarError (ENTERO_FUERA_DE_RANGO, línea+1, token);`
	`SI NO EnviarToken(RELAT_IX_V,valor);`

30→31	`EnviarToken(RELAT_IX_E,identificador);`

- Operandos con modo de direccionamiento relativo a registro índice IY.

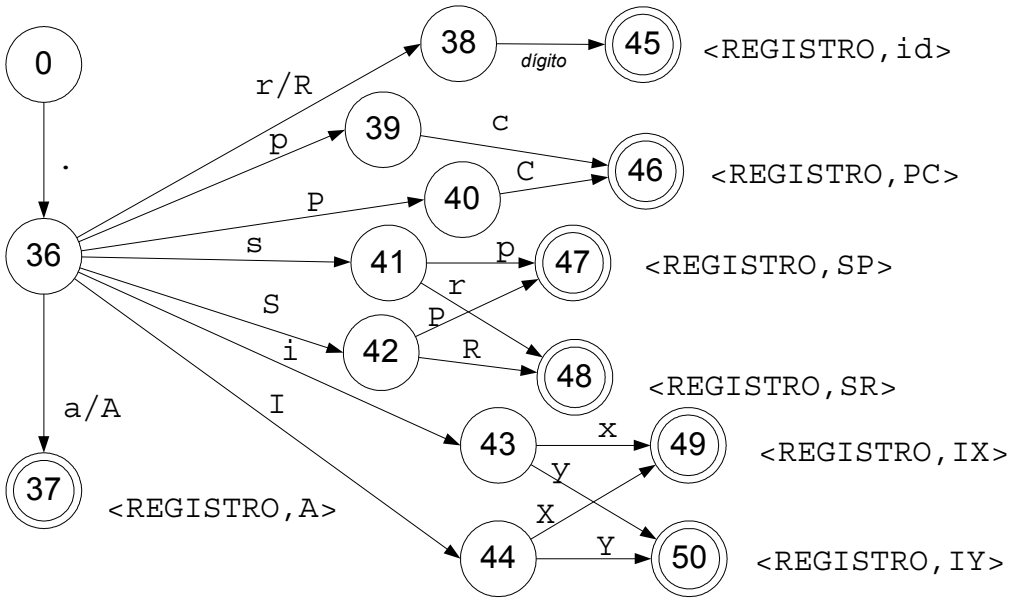


Transición	Acciones semánticas

32→33	`SI (negativo) ENTONCES valor = - valor;`
	`SI (valor < -128) O (valor > 127)`
	`ENTONCES EnviarError (ENTERO_FUERA_DE_RANGO, línea+1, token);`
	`SI NO EnviarToken(RELAT_IY_V,valor);`

34→35	`EnviarToken(RELAT_IY_E,identificador);`

- Operandos con modo de direccionamiento directo a registro.

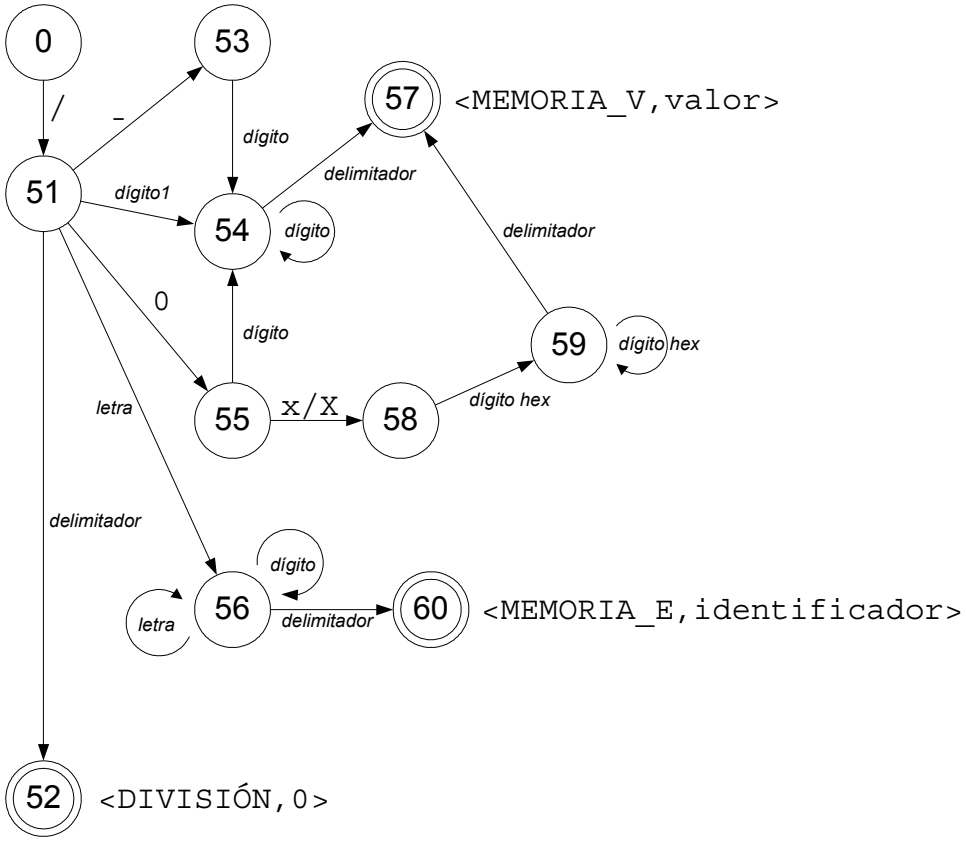


Transición	Acciones semánticas

38→45	`SI (entrada = = '0') ENTONCES id = R0;`
	`...`
	`SI (entrada = = '9') ENTONCES id = R9;`
	`EnviarToken(REGISTRO,id);`

36→37	`EnviarToken(REGISTRO,A);`
39→46 40→46	`EnviarToken(REGISTRO,PC);`
41→47 42→47	`EnviarToken(REGISTRO,SP);`
41→48 42→48	`EnviarToken(REGISTRO,SR);`
43→49 44→49	`EnviarToken(REGISTRO,IX);`
43→50 44→50	`EnviarToken(REGISTRO,IY);`

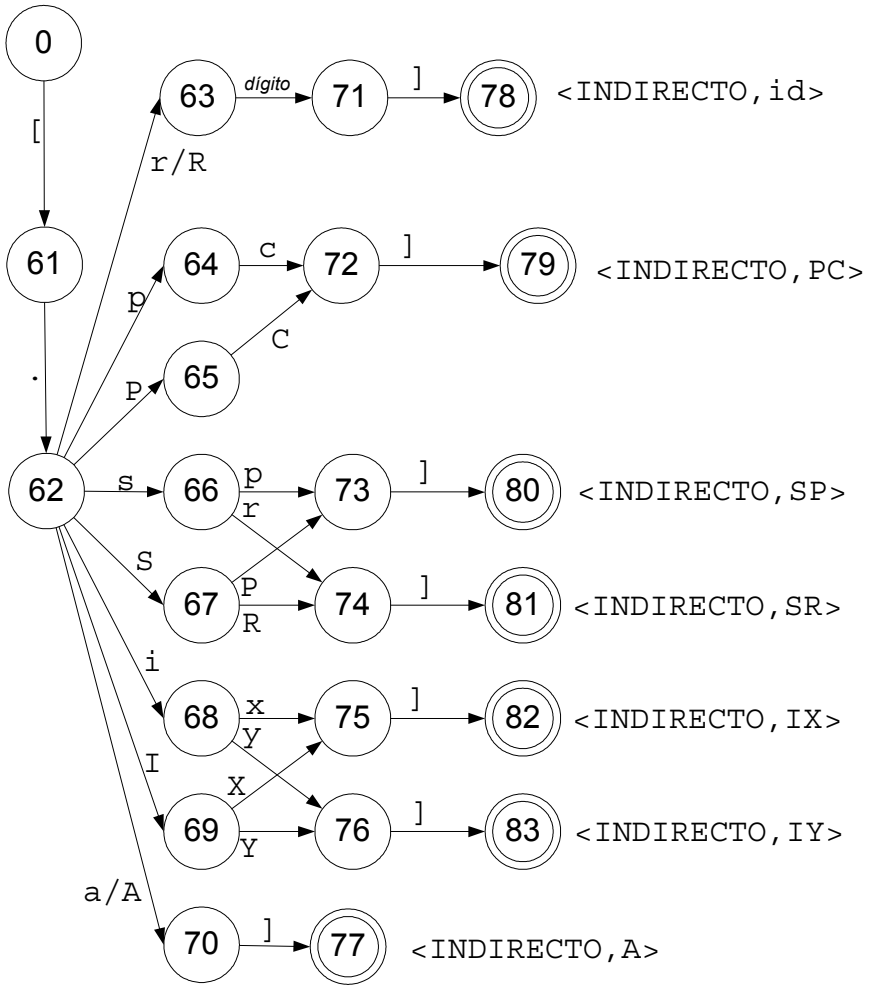
- Operandos con modo de direccionamiento directo a memoria.



Transición	Acciones semánticas
51→52	`EnviarToken(DIVISIÓN,0);`
51→53	`negativo = cierto;`
51→54 53→54 54→54 55→54	`valor = valor*10 + ValorEntero(entrada);`
51→56 56→56	`identificador = identificador ⊕ entrada;`
54→57	`SI (negativo) ENTONCES valor = - valor;`

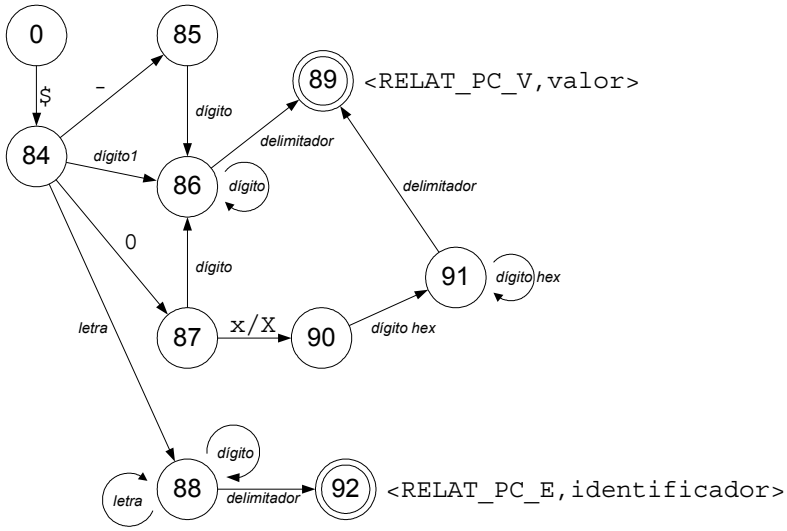
59→57	SI (valor < -32768) O (valor > 65535) ENTONCES EnviarError (ENTERO_FUERA_DE_RANGO, línea+1, token); SI NO EnviarToken(MEMORIA_V,valor);
58→59 59→59	valor = valor*16 + ValorEntero(entrada);
56→60	EnviarToken(MEMORIA_E,identificador);

- Operandos con modo de direccionamiento indirecto.



Transición	Acciones semánticas
63→71	SI (entrada = = '0') ENTONCES id = R0; ... SI (entrada = = '9') ENTONCES id = R9;
70→77	EnviarToken(INDIRECTO,A);
71→78	EnviarToken(INDIRECTO,id);
72→79	EnviarToken(INDIRECTO,PC);
73→80	EnviarToken(INDIRECTO,SP);
74→81	EnviarToken(INDIRECTO,SR);
75→82	EnviarToken(INDIRECTO,IX);
76→83	EnviarToken(INDIRECTO,IX);

- Operandos con modo de direccionamiento relativo a PC.

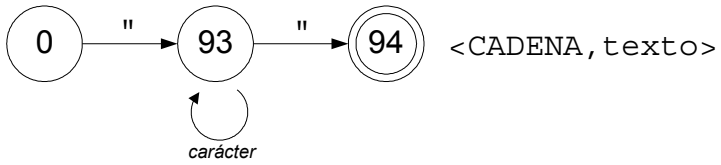


Transición	Acciones semánticas
84→85	`negativo = cierto;`
84→86 85→86 86→86 87→86	`valor = valor*10 + ValorEntero(entrada);`
84→88 88→88	`identificador = identificador ⊕ entrada;`

86→89	`SI (negativo) ENTONCES valor = - valor;`
	`SI (valor < -128) O (valor > 127)`
	`  ENTONCES EnviarError (ENTERO_FUERA_DE_RANGO, línea+1, token);`
	`  SI NO EnviarToken(RELAT_PC_V,valor);`

90→91 91→91	`valor = valor*16 + ValorEntero(entrada);`
88→92	`EnviarToken(RELAT_PC_E,identificador);`

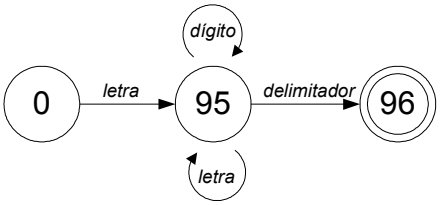
- Cadenas.



Transición	Acciones semánticas
93→93	`texto = texto ⊕ entrada;`
93→94	`EnviarToken(CADENA,texto);`

- Identificadores.

  Después de leer un identificador, hay que comprobar si es una palabra clave, según la Tabla 3.2.3, y de esta forma se enviará un *token* u otro.



<IDENTIFICADOR,texto>

<MNEMÓNICO0,cod_op>

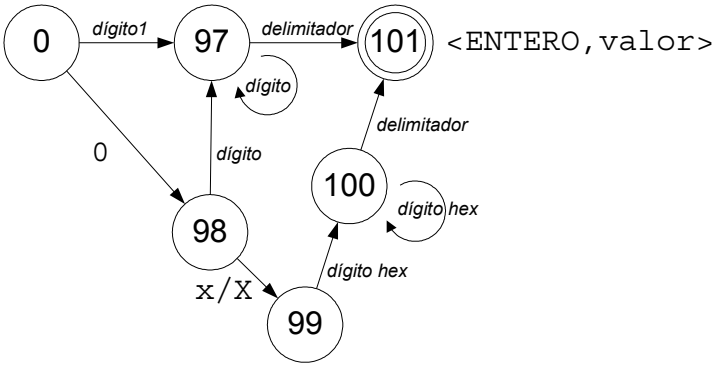
<MNEMÓNICO1,cod_op>

<MNEMÓNICO3,cod_op>

<ORG,0> … <DATA,0>

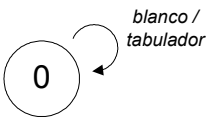
Transición	Acciones semánticas
0→95 95→95	texto = texto ⊕ entrada;
95→96	SI (EsPalabraClave(texto)) ENTONCES EnviarToken(TipoToken(texto),Atributo(texto)); SI NO EnviarToken(IDENTIFICADOR,texto);

- Enteros (decimales y hexadecimales).

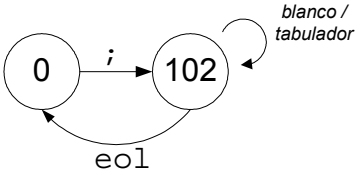


Transición	Acciones semánticas
0→97 97→97 98→97	valor = valor*10 + entrada;
99→100 100→100	valor = valor*16 + entrada;
97→101 100→101	EnviarToken(ENTERO,valor);

- Espacios y tabulaciones.



- Comentarios.



En cuanto a los errores que detecta el analizador léxico, son los dos siguientes:

- *Carácter no válido en la entrada*. Ocurre cuando el carácter leído en la entrada no se puede equiparar con ninguna regla, esto es, mirando el AFD, cuando el estado en el que está el autómata no dispone de ninguna transición con dicho carácter. En este caso se indicará qué carácter provocó el error y la línea de código fuente en la que se encontró. En el caso de producirse este error, el análisis prosigue en la siguiente línea (aunque sólo con el fin de detectar más errores, ya que se detiene la generación de código). Se invocará a la función `EnviarError (ENTRADA_ERRÓNEA, línea+1, token);`
- *Entero fuera de rango*. Ocurre cuando se lee un *token* operando con un entero (decimal o hexadecimal) cuyo valor excede del rango de representación de los enteros de 16 bits, o bien cuando se lee un *token* desplazamiento con un entero cuyo valor excede del rango de representación de los enteros de 8 bits. Este error es detectado por las acciones semánticas añadidas al autómata.

3.3.1.2. Analizador Sintáctico, Semántico y Generador de Código

La gramática propuesta para el analizador sintáctico se va a completar con una serie de acciones semánticas, que permitirán efectuar las comprobaciones vistas en el apartado correspondiente al análisis. Todas las referentes a cálculo de atributos y comprobación de errores se engloban dentro del análisis semántico. Por el contrario, aquellas encargadas de construir el código máquina pertenecen al generador de código. No obstante, se presentan entremezcladas ya que así van a actuar cuando se ejecute el ensamblador.

En primera instancia, las acciones semánticas requieren el uso de algunas funciones auxiliares, que son las siguientes:

- `ComprobarModoDir (operando, cód_op, modo_dir)`: devuelve si el modo de direccionamiento indicado es válido para el código de operación y el operando determinados.
- `EmitirInstrucción (cód_op, modo_dir_1, operando_1, modo_dir_2, operando_2, dirmem)`: genera el código máquina correspondiente a la instrucción indicada por los parámetros, a partir de la posición de memoria indicada por la variable `dirmem`, que es actualizada con la posición a partir de la que se ensamblará la siguiente instrucción. Dicha variable se incrementará en 1 posición si la instrucción no lleva operandos. Se incrementará en 2 posiciones si la instrucción lleva un operando, o dos, no siendo ninguno de ellos inmediato o directo a memoria. Y se incrementará en 3 posiciones si lleva dos operandos, y alguno de ellos o ambos son inmediatos o directos a memoria.
- `EnviarError (tipo, línea, token)`: informa al gestor de errores de un nuevo error, indicando el tipo de error, la línea de código fuente que se estaba analizando cuando se encontró y el *token* que lo provocó (si procede).
- `ExisteEtiqueta (identificador)`: devuelve si el identificador determinado existe o no en la tabla de etiquetas.
- `Lista (texto)`: devuelve una lista de valores enteros correspondientes según el código *ASCII* a la cadena de caracteres determinada, terminando con el carácter nulo. Por ejemplo, si `texto = "texto"`, entonces `Lista(texto)` devuelve {116, 101, 120, 116, 111, 0} (116 es el código *ASCII* del carácter ‘t’, 101 el del carácter ‘e’, y así sucesivamente).

- `Longitud (cadena)`: calcula la longitud de una cadena de caracteres (más el carácter de fin de cadena).
- `NuevaEtiqueta (identificador, valor)`: inserta un nuevo identificador y su valor entero en la Tabla de Etiquetas.
- `NuevaReferenciaAdelantada (identificador, posición, desplazamiento, modo_dir, línea)`: genera una nueva entrada en la tabla de referencias adelantadas con la información proporcionada.

Los atributos que acompañan a los *tokens* se pueden consultar en la Tabla 3.2.4, a los que se añaden estos nuevos para los símbolos no terminales de la gramática:

- `EXPRESIÓN.valor` contiene el valor de la expresión que se está analizando en esa regla.
- `EXPRESIÓN2.valor` contiene el valor de la expresión que se está analizando en esa regla.
- `EXPRESIÓN3.valor` contiene el valor de la expresión que se está analizando en esa regla.
- `INSTRUCCIÓN.cód_op` contiene el código de operación de la instrucción que se está analizando.
- `INSTRUCCIÓN.modo_dir_1` contiene el modo de direccionamiento del primer operando de la instrucción (si lo tiene).
- `INSTRUCCIÓN.modo_dir_2` contiene el modo de direccionamiento del segundo operando (si lo tiene).
- `INSTRUCCIÓN.operando_1` contiene el valor entero del primer operando (si lo tiene).
- `INSTRUCCIÓN.operando_2` contiene el valor entero del segundo operando (si lo tiene).
- `INSTRUCCIÓN.valor` contiene la dirección de memoria a partir de la cual se ha ensamblado la instrucción.
- `LISTA_DATOS.longitud` contiene la longitud de la lista de datos (de una pseudoinstrucción `DATA`).
- `LISTA_DATOS.datos` contiene la lista de datos, que es una lista de valores enteros (de una pseudoinstrucción `DATA`).
- `OPERANDO.modo_dir` contiene el modo de direccionamiento del operando que se está analizando.
- `OPERANDO.valor` contiene el valor entero del operando que se está analizando.
- `OPERANDO.etiqueta` contiene el identificador correspondiente cuando el operando es una etiqueta.
- `PSEUDOINSTRUCCIÓN.valor` contiene la dirección de memoria a partir de la cual se ensambla la pseudoinstrucción, o el valor de la expresión de una pseudoinstrucción `EQU`.
- `RESTO.valor` contiene la dirección de memoria cuyo valor debería tomar una etiqueta situada al comienzo de la línea.

Por último, se usarán las siguientes variables:

- `dirmem` contiene la dirección de memoria en la que se está insertando el código máquina generado que, inicialmente, es la posición 0.
- `línea` contiene el número de línea del fichero de código fuente que se está analizando.

A continuación se expone la Definición Dirigida por la Sintaxis resultante. En cursiva se muestran las reglas que se obtuvieron para el analizador sintáctico, mientras que las acciones semánticas se muestran con un tipo de letra monoespaciado.

Producción	Reglas semánticas
S → λ	
S → <EOL> *S*	
S → *LÍNEA S*	

LÍNEA → <IDENTIFICADOR> <FIN_ETIQ> *LÍNEA_VACÍA RESTO*	SI (ExisteEtiqueta(IDENTIFICADOR.texto))
	ENTONCES
	  EnviarError(ETIQUETA_DUPLICADA,nlin,IDENTIFICADOR.texto);
	SI NO
	  NuevaEtiqueta(IDENTIFICADOR.texto,RESTO.valor);

LÍNEA → *RESTO*	
LÍNEA_VACÍA → λ	
LÍNEA_VACÍA → <EOL> *LÍNEA_VACÍA*	
RESTO → *INSTRUCCIÓN*	RESTO.valor := dirmem;
RESTO → *INSTRUCCIÓN*	RESTO.valor := PSEUDOINSTRUCCIÓN.valor;

INSTRUCCIÓN → <*MNEMÓNICO0*> <*EOL*>	INSTRUCCIÓN.cód_op := MNEMÓNICO0.cód_op;
	INSTRUCCIÓN.modo_dir_1 := NINGUNO;
	INSTRUCCIÓN.operando_1 := NINGUNO;
	INSTRUCCIÓN.modo_dir_2 := NINGUNO;
	INSTRUCCIÓN.operando_2 := NINGUNO;
	INSTRUCCIÓN.valor := dirmem;
	EmitirInstrucción(INSTRUCCIÓN.cód_op,INSTRUCCIÓN.modo_dir_1,
	  INSTRUCCIÓN.operando_1,INSTRUCCIÓN.modo_dir_2,
	  INSTRUCCIÓN.operando_2,dirmem);

INSTRUCCIÓN → <MNEMÓNICO1> *OPERANDO* <EOL>	INSTRUCCIÓN.cód_op := MNEMÓNICO1.cód_op;
	INSTRUCCIÓN.modo_dir_1 := OPERANDO.modo_dir;
	INSTRUCCIÓN.operando_1 := OPERANDO.valor;
	INSTRUCCIÓN.modo_dir_2 := NINGUNO;
	INSTRUCCIÓN.operando_2 := NINGUNO;
	INSTRUCCIÓN.valor := dirmem;
	ComprobarModoDir1(INSTRUCCIÓN.instrucción,
	  INSTRUCCIÓN.modo_dir_1);
	SI (OPERANDO.etiqueta != "")
	ENTONCES
	  NuevaReferenciaAdelantada(OPERANDO.etiqueta,dirmem+1,0,
	  OPERANDO.modo_dir, línea);
	  EmitirInstrucción(INSTRUCCIÓN.cód_op,
	  INSTRUCCIÓN.modo_dir_1, INSTRUCCIÓN.operando_1,
	  INSTRUCCIÓN.modo_dir_2, INSTRUCCIÓN.operando_2,
	  dirmem);

INSTRUCCIÓN → <MNEMÓNICO2> $OPERANDO_1$ <SEPARADOR> $OPERANDO_2$ <EOL>	INSTRUCCIÓN.instrucción := MNEMÓNICO2.cód_op;
	INSTRUCCIÓN.modo_dir_1 := $OPERANDO_1$.modo_dir;
	INSTRUCCIÓN.operando_1 := $OPERANDO_1$.valor;
	INSTRUCCIÓN.modo_dir_2 := $OPERANDO_2$.modo_dir;
	INSTRUCCIÓN.operando_2 := $OPERANDO_2$.valor;
	INSTRUCCIÓN.valor := dirmem;
	ComprobarModoDir(INSTRUCCIÓN.instrucción,
	  INSTRUCCIÓN.modo_dir_1);
	ComprobarModoDir(INSTRUCCIÓN.instrucción,
	  INSTRUCCIÓN.modo_dir_2);
	SI (OPERANDO1.etiqueta != "")
	ENTONCES
	  NuevaReferenciaAdelantada($OPERANDO_1$.etiqueta,dirmem+1,0,
	  $OPERANDO_1$.modo_dir, línea);
	SI
	  ($OPERANDO_1$.modo_dir == INMEDIATO) O
	  ($OPERANDO_1$.modo_dir == MEMORIA) O
	  ($OPERANDO_2$.modo_dir == INMEDIATO) O
	  ($OPERANDO_2$.modo_dir == MEMORIA)
	ENTONCES
	  desplaz := 1;
	SI NO
	  desplaz := 0;
	SI ($OPERANDO_2$.etiqueta != "")
	ENTONCES
	  NuevaReferenciaAdelantada($OPERANDO_2$.etiqueta,desplaz,0,
	  $OPERANDO_2$.modo_dir, línea);
	  EmitirInstrucción(INSTRUCCIÓN.cód_op,
	  INSTRUCCIÓN.modo_dir_1, INSTRUCCIÓN.operando_1,
	  INSTRUCCIÓN.modo_dir_2, INSTRUCCIÓN.operando_2,
	  dirmem);

OPERANDO → <INMEDIATO_V>	OPERANDO.modo_dir := INMEDIATO; OPERANDO.valor := INMEDIATO_V.valor; OPERANDO.etiqueta := "";
OPERANDO → <INMEDIATO_E>	OPERANDO.modo_dir := INMEDIATO; OPERANDO.valor := 0; OPERANDO.etiqueta := INMEDIATO_E.identificador;
OPERANDO → <REGISTRO>	OPERANDO.modo_dir := REGISTRO; OPERANDO.valor := REGISTRO.id_numérico; OPERANDO.etiqueta := "";
OPERANDO → <MEMORIA_V>	OPERANDO.modo_dir := MEMORIA; OPERANDO.valor := MEMORIA_V.valor; OPERANDO.etiqueta := "";
OPERANDO → <MEMORIA_E>	OPERANDO.modo_dir := MEMORIA; OPERANDO.valor := 0; OPERANDO.etiqueta := MEMORIA_E.identificador;
OPERANDO → <INDIRECTO>	OPERANDO.modo_dir := INDIRECTO; OPERANDO.valor := INDIRECTO.id_numérico OPERANDO.etiqueta := "";
OPERANDO → <RELAT_PC_V>	OPERANDO.modo_dir := RELATIVO_PC; OPERANDO.valor := RELAT_PC_V.valor; OPERANDO.etiqueta := "";
OPERANDO → <RELAT_PC_E>	OPERANDO.modo_dir := RELATIVO_PC; OPERANDO.valor := 0; OPERANDO.etiqueta := RELAT_PC_E.identificador;
OPERANDO → <RELAT_IX_V>	OPERANDO.modo_dir := RELATIVO_IX; OPERANDO.valor := RELAT_IX_V.valor; OPERANDO.etiqueta := "";
OPERANDO → <RELAT_IX_E>	OPERANDO.modo_dir := RELATIVO_IX; OPERANDO.valor := 0; OPERANDO.etiqueta := RELAT_IX_E.identificador;
OPERANDO → <RELAT_IY_V>	OPERANDO.modo_dir := RELATIVO_IY; OPERANDO.valor := RELAT_IY_V.valor; OPERANDO.etiqueta := "";
OPERANDO → <RELAT_IY_E>	OPERANDO.modo_dir := RELATIVO_IY; OPERANDO.valor := 0; OPERANDO.etiqueta := RELAT_IY_E.identificador;

PSEUDOINSTRUCCIÓN → <ORG> *EXPRESIÓN* <EOL>	SI
	  (EXPRESIÓN.valor < 0) O (EXPRESIÓN.valor > 65535)
	ENTONCES
	  EnviarError(ORG_FUERA_DE_RANGO,línea,EXPRESIÓN.valor);
	SI NO
	  dirmem := EXPRESIÓN.valor;
	  PSEUDOINSTRUCCIÓN.valor := dirmem;

PSEUDOINSTRUCCIÓN → <RES> *EXPRESIÓN* <EOL>	SI ((EXPRESIÓN.valor + dirmem) > 65535)
	ENTONCES
	  EnviarError(RES_FUERA_DE_RANGO,línea,EXPRESIÓN.valor);
	SI NO
	  PSEUDOINSTRUCCIÓN.valor := dirmem;
	  dirmem = dirmem + EXPRESIÓN.valor;

PSEUDOINSTRUCCIÓN → <DATA> *LISTA_DATOS* <EOL>	SI (LISTA_DATOS.longitud + dirmem) > 65535)
	ENTONCES
	  EnviarError(RES_FUERA_DE_RANGO,línea,"");
	SI NO
	  PSEUDOINSTRUCCIÓN.valor := dirmem;
	  dirmem = dirmem + LISTA_DATOS.longitud;

PSEUDOINSTRUCCIÓN → <EQU> *EXPRESIÓN* <EOL>	PSEUDOINSTRUCCIÓN.valor := EXPRESIÓN.valor;
PSEUDOINSTRUCCIÓN → <END> <EOL>	

EXPRESIÓN → <RESTA> *EXPRESIÓN2*	EXPRESIÓN.valor := - EXPRESIÓN2.valor;
	SI
	  (EXPRESIÓN.valor < -32768) O (EXPRESIÓN.valor > 65535)
	ENTONCES
	  EnviarError(EXPR_FUERA_DE_RANGO,línea,EXPRESIÓN.valor);

EXPRESIÓN → *EXPRESIÓN*$_1$ <SUMA> *EXPRESIÓN2*	EXPRESIÓN.valor := EXPRESIÓN$_1$.valor + EXPRESIÓN2.valor; SI (EXPRESIÓN.valor < -32768) O (EXPRESIÓN.valor > 65535) ENTONCES EnviarError(EXPR_FUERA_DE_RANGO,línea,EXPRESIÓN.valor);
EXPRESIÓN → *EXPRESIÓN*$_1$ <RESTA> *EXPRESIÓN2*	EXPRESIÓN.valor := EXPRESIÓN$_1$.valor - EXPRESIÓN2.valor; SI (EXPRESIÓN.valor < -32768) O (EXPRESIÓN.valor > 65535) ENTONCES EnviarError(EXPR_FUERA_DE_RANGO,línea,EXPRESIÓN.valor);

EXPRESIÓN → *EXPRESIÓN2*	`EXPRESIÓN.valor := EXPRESIÓN2.valor;`
	`SI (EXPRESIÓN.valor < -32768) O (EXPRESIÓN.valor > 65535)`
	`ENTONCES`
	`  EnviarError(EXPR_FUERA_DE_RANGO,línea,EXPRESIÓN.valor);`

EXPRESIÓN2 → *EXPRESIÓN2*$_1$ <PRODUCTO> *EXPRESIÓN3*	`EXPRESIÓN2.valor := EXPRESIÓN2`$_1$`.valor * EXPRESIÓN3.valor;`

EXPRESIÓN2 → *EXPRESIÓN2*$_1$ <DIVISIÓN> *EXPRESIÓN3*	`SI (EXPRESION3.valor == 0)`
	`ENTONCES`
	`  EnviarError(EXPR_DIVISIÓN_POR_CERO,nlin,0);`
	`SI NO`
	`  EXPRESIÓN2.valor := EXPRESIÓN2`$_1$`.valor / EXPRESIÓN3.valor;`

EXPRESIÓN2 → *EXPRESIÓN2*$_1$ <MÓDULO> *EXPRESIÓN3*	`SI (EXPRESION3.valor == 0)`
	`ENTONCES`
	`  EnviarError(EXPR_DIVISIÓN_POR_CERO,nlin,0);`
	`SI NO`
	`  EXPRESIÓN2.valor := EXPRESIÓN2`$_1$`.valor % EXPRESIÓN3.valor;`

EXPRESIÓN2 → *EXPRESIÓN3*	`EXPRESIÓN2.valor := EXPRESIÓN3.valor;`
EXPRESIÓN3 → <PARENT_ABRE> *EXPRESIÓN* <PARENT_CIERRA>	`EXPRESIÓN3.valor := EXPRESIÓN.valor;`
EXPRESIÓN3 → <ENTERO>	`EXPRESIÓN3.valor := ENTERO.valor;`
LISTA_DATOS → <ENTERO> *RESTO_LISTA_DATOS*	`LISTA_DATOS.longitud := 1 + RESTO_LISTA_DATOS.longitud;` `LISTA_DATOS.datos := ENTERO.valor ⊕ LISTA_DATOS.datos;`

LISTA_DATOS → <CADENA> *RESTO_LISTA_DATOS*	`LISTA_DATOS.longitud := Longitud(CADENA.texto) +`
	`  RESTO_LISTA_DATOS.longitud;`
	`LISTA_DATOS.datos := Lista(CADENA.texto) ⊕  LISTA_DATOS.datos;`

RESTO_LISTA_DATOS → λ	`RESTO_LISTA_DATOS.longitud := 0;`
RESTO_LISTA_DATOS → <SEPARADOR> LISTA_DATOS	`RESTO_LISTA_DATOS.longitud := LISTA_DATOS.longitud;` `RESTO_LISTA_DATOS.datos := LISTA_DATOS.datos;`

Para completar la generación de código, se ejecutará el siguiente algoritmo:

- Mientras (no hay errores) y (quedan entradas en la tabla de configuración por procesar):
  - Leer la siguiente entrada de la tabla de configuración.
  - Si la etiqueta referida tiene valor, y es correcto, insertarlo en el código máquina. Si no, generar un error.
- Si (No Hay errores):
  - Devolver el código máquina generado
- Si (Hay errores):
  - Devolver la lista de errores producidos durante el proceso de ensamblado

Durante la etapa de análisis semántico se pueden detectar los siguientes errores:

- *Comienzo del código fuera de los límites de la memoria*. Ocurre cuando se está analizando una regla ORG y se recibe una expresión cuyo valor excede los límites de la memoria.
- *Definición de datos fuera de los límites de la memoria*. Ocurre cuando se está analizando una regla DATA, y al intentar ubicar la lista de datos, a partir de la posición actual de ensamblado, cae fuera de los límites de la memoria.

- *Etiqueta duplicada*. Ocurre cuando se encuentra un *token* `<IDENTIFICADOR>`, se va a la tabla de símbolos para anotarlo y ya hay otro anotado previamente con el mismo identificador.
- *Expresión fuera de rango*. Ocurre cuando se está analizando una expresión y el valor del resultado excede del rango de representación de los enteros de 16 bits.
- *Modo de direccionamiento del Operando 1 erróneo*. Ocurre cuando el modo de direccionamiento recogido por el primer operando no es válido para la instrucción que se está analizando.
- *Modo de direccionamiento del Operando 2 erróneo*. Ocurre cuando el modo de direccionamiento recogido por el segundo operando no es válido para la instrucción que se está analizando.
- *Reserva de espacio fuera de los límites de la memoria*. Ocurre cuando se está analizando una regla RES y se recibe una expresión cuyo valor, sumado a la posición actual de ensamblado, excede los límites de la memoria.

El generador de código puede detectar los siguientes errores:

- *Desplazamiento fuera de rango*. Ocurre cuando se está revisando la tabla de configuración, se encuentra una entrada referente a un desplazamiento y se encuentra en la tabla de etiquetas un valor que excede la representación de enteros en 8 bits.
- *Etiqueta no definida*. Ocurre cuando se está recorriendo la tabla de configuración para resolver las referencias adelantadas y se encuentra una anotación para un identificador de etiqueta que no se encuentra en la tabla de etiquetas.

3.3.1.3. Tabla de Etiquetas

En la tabla de etiquetas se irán almacenando todas aquellas etiquetas que se definan a lo largo del código fuente. La definición de etiquetas viene dada por la regla:

LÍNEA → `<IDENTIFICADOR>` `<FIN_ETIQ>` LÍNEA_VACÍA RESTO

en la que como atributo del *token* etiqueta se recupera el identificador de la etiqueta.

El único dato de interés que será necesario guardar de cada etiqueta es su valor. A la hora de generar código máquina, se sustituirá cada etiqueta por su valor correspondiente. Por tanto, la tabla de etiquetas será parecida a la que se muestra en la Tabla 3.3.1.

Identificador	Valor
Etiqueta_1	33
Etiqueta_2	¿?
Bucle	10

Tabla 3.3.1. Ejemplo de Tabla de Etiquetas

Debe permitirse que la columna *Valor* contenga valores indefinidos, ya que generalmente no se conocerá el valor que toma la etiqueta en el momento de su definición (inclusión en la tabla). Así que debe existir alguna forma de actualizar dicho valor basándose en un identificador dado.

A la hora de controlar errores, hay que resaltar que no puede haber dos etiquetas con el mismo identificador y cada etiqueta se corresponderá, al final del proceso de análisis, con un único valor.

En resumen, la tabla de etiquetas será una estructura que contenga pares (*Identificador*, *Valor*), teniendo en cuenta que no puede haber dos identificadores repetidos, y que proporcionará funciones para añadir pares, detectar si un identificador ya está definido, consultar un valor de un identificador dado y modificar un valor de un identificador dado. Por tanto, se deben definir funciones para crear la tabla, destruir la tabla, añadir etiquetas, dar valor a las etiquetas y consultar el valor de dichas etiquetas.

Estas son las funciones propuestas:

Función	**InicializarTablaEtiquetas**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Inicializa la estructura interna de la tabla de etiquetas, dejándola vacía. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si se produjo algún error durante la inicialización.

Función	**ConsultarEtiqueta**
Parámetros	IN etiqueta: `Cadena` OUT valor: `Entero`
Salida	`Entero`
Comportamiento	Consulta la tabla de etiquetas, buscando el identificador indicado por el parámetro de entrada `etiqueta`, y devuelve el valor asociado en el parámetro de salida `valor`. Devuelve 0 como valor de retorno.
Errores	Si la etiqueta no existe en la tabla de etiquetas, devuelve –1 como valor de retorno.

Función	**GuardarEtiqueta**
Parámetros	IN etiqueta: `Cadena` IN valor: `Entero`
Salida	`Entero`
Comportamiento	Almacena la etiqueta en la tabla de etiquetas, asociando al identificador `etiqueta` el valor indicado por el parámetro `valor`. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno, si hubo problemas de asignación de espacio dinámico. Devuelve –2 como valor de retorno, si la etiqueta ya existía en la tabla. Devuelve –3 como valor de retorno, si el identificador elegido coincide con una palabra reservada del lenguaje ensamblador que se está analizando. En caso de error, se deja la tabla como estuviera anteriormente.

Función	**DarValorEtiqueta**
Parámetros	IN etiqueta: `Cadena` IN valor: `Entero`
Salida	`Entero`
Comportamiento	Asocia en la tabla de etiquetas el valor indicado por el parámetro `valor` al identificador indicado por el parámetro `etiqueta`, que debe existir previamente en la tabla. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si la etiqueta no existe en la tabla.

Función	**LiberarTablaEtiquetas**
Parámetros	Ninguno
Salida	`Entero`

Comportamiento	Libera los recursos que se hubieran podido reservar para almacenar la tabla de etiquetas. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si hubo algún problema durante la operación.

3.3.1.4. Tabla de Referencias Adelantadas

La tabla de referencias adelantadas, al igual que la tabla de etiquetas, no hace ningún tipo de procesamiento sobre los datos que almacena. Simplemente se encarga de mantener la tabla, esta vez con las columnas identificador, dirección, desplazamiento y tamaño. También se va a guardar información sobre la línea de código que se estaba analizando cuando se introdujo la entrada en la tabla, para informar de los errores que pudieran surgir.

Las entradas de la tabla de referencias adelantadas vendrán generadas por las reglas del tipo:

OPERANDO → `<MEMORIA_E>`

En este caso, el operando viene expresado según un direccionamiento directo a memoria (lo que indica el tamaño de la referencia, 16 bits en este caso), pero el valor de la dirección de memoria viene dado por una etiqueta. En el momento de procesar la regla, se conoce qué operando se está tratando, el primero o el segundo (lo que aporta el dato relativo al desplazamiento), en qué dirección de memoria se está ensamblando (aporta la posición dentro del código) y el identificador de la etiqueta. El analizador sintáctico consultará la tabla de etiquetas, y puede actuar según dos estrategias posibles. La primera consiste en que si su valor es conocido, se sustituye y se continúa con el proceso, pero si no está definido todavía, se hace la anotación en la tabla de referencias adelantadas con todos estos datos. La segunda, que es la que se va a aplicar, consiste en dejar para el final la actualización de todos los valores anotados en la tabla.

La tabla de referencias adelantadas tendrá una estructura similar a la de la Tabla 3.3.2.

Identificador	Dirección	Desplazamiento	Tamaño
Etiqueta_1	20	0	16 bits
Etiqueta_2	24	8	8 bits
Etiqueta_3	26	0	8 bits

Tabla 3.3.2. Ejemplo de Tabla de Referencias Adelantadas

Estas son las funciones propuestas:

Función	**InicializarTablaReferenciasAdelantadas**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Inicializa la estructura interna de la tabla de referencias adelantadas, dejándola vacía. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si se produjo algún error durante la inicialización.

Función	**EscribirTablaReferenciasAdelantadas**
Parámetros	IN etiqueta: `Cadena` IN posición: `Entero` IN desplazamiento: `Entero` IN mododireccionamiento: `Entero` IN línea: `Entero`
Salida	`Entero`
Comportamiento	Almacena en la tabla una nueva entrada de referencia adelantada, tomando los valores de los parámetros de entrada. Devuelve 0 como valor de retorno si la operación se ejecutó correctamente.
Errores	Devuelve –1 como valor de retorno si hubo algún problema de asignación de espacio dinámico.

Función	**RevisarTablaReferenciasAdelantadas**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Recorre la tabla y va actualizando el código generado con las referencias pendientes. Devuelve 0 como valor de retorno.
Errores	Si hay algún error al resolver alguna referencia llama a la función `InformarError` del gestor de errores, indicando el tipo de error que se ha producido, la línea de código fuente en la que se ha producido y el *token* que lo ha provocado. Devuelve –1 como valor de retorno.

Función	**LiberarTablaReferenciasAdelantadas**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Libera los recursos que se hubieran podido reservar para almacenar la tabla de referencias adelantadas. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno, si hubo algún problema durante la operación. El resto de la tabla se mantiene como estuviera antes de invocar a la función.

3.3.1.5. Gestor de Errores

Siguiendo un diseño análogo al de la tabla de etiquetas y de referencias adelantadas, la tabla de errores mantiene un listado de los errores que vayan surgiendo durante las fases de análisis léxico, sintáctico, semántico y generación de código. Se trata de almacenarlos en una tabla con las columnas número secuencial, descripción, *token* que produjo el error y línea de código fuente en la que se detectó el error.

Podría ser una tabla con el formato de la Tabla 3.3.3.

Número	Descripción	Token	Nº Línea
1	Modo Direccionamiento Erróneo	#31[.ix]	4
2	Se esperaba Fin de Línea	.r3	12
3	Etiqueta Duplicada	Bucle	20

Tabla 3.3.3. Ejemplo de Tabla de Errores

Estas son las funciones que se proponen:

Función	**InicializarListaErrores**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Inicializa la estructura interna de la lista de errores, dejándola vacía. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si se produjo algún error durante la inicialización.

Función	**LiberarListaErrores**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Libera el espacio dinámico asignado a la lista de errores. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno, si hubo algún problema durante la operación.

Función	**InformarError**
Parámetros	IN código: `Entero` IN línea: `Entero` IN token: `Cadena`
Salida	`Entero`
Comportamiento	Introduce un nuevo error en la tabla, con los valores indicados por los parámetros. La descripción se crea a partir del tipo de error. Devuelve 0 como valor de retorno.
Errores	Devuelve –1, si hubo algún problema en la asignación de espacio dinámico.

Función	**VolcarFicheroErrores**
Parámetros	IN nombrefichero: `Cadena`
Salida	`Entero`
Comportamiento	Vuelca el contenido de la tabla de errores en el fichero indicado por el parámetro de entrada `nombrefichero`. Devuelve 0 como valor de retorno.
Errores	Devuelve –1, si no se pudo crear o escribir correctamente el fichero.

3.3.2. Diseño del Simulador

3.3.2.1. Diseño del sistema

El diseño de sistemas es la primera fase de diseño en la que se selecciona la aproximación básica para resolver el problema. Durante el diseño del sistema, se decide la estructura y el estilo global. La arquitectura es la organización global del sistema dividido en componentes llamados subsistemas. La arquitectura proporciona el contexto en el que se toman decisiones más detalladas en una fase posterior del diseño. De esta manera se posibilita el trabajo independiente de diseño sobre distintos subsistemas.

A la hora de diseñar el sistema se deben tomar las decisiones siguientes:

- Organizar el sistema en subsistemas.
- Identificar la concurrencia inherente al problema.
- Asignar los subsistemas a los procesadores y tareas.
- Seleccionar una aproximación para la administración de almacenes de datos.
- Manejar el acceso a recursos globales.
- Seleccionar la implementación de control en software.

- Manejar las condiciones de contorno.
- Establecer la compensación de prioridades.

Como siempre, sólo se van a detallar aquéllas que tengan alguna relevancia en el caso concreto del sistema *ENS2001*.

Descomposición del sistema en subsistemas

Cada subsistema abarca aspectos del sistema que compartan alguna propiedad común. Normalmente, un subsistema se caracteriza por los servicios que proporciona. Cada subsistema posee una interfaz bien definida con el resto del sistema. De la misma forma, es deseable que los subsistemas se definan de modo que la mayor parte de las interacciones se produzcan dentro de cada uno de ellos, con el fin de reducir las dependencias.

En este caso el sistema completo no es muy grande, pero hay una característica que invita a descomponer el simulador de *ENS2001* en, al menos, dos subsistemas, a modo de capas. Por una parte, estaría lo que se podría denominar el *nucleo* del simulador, que estaría compuesto por las clases que modelan la arquitectura simulada. Por otra parte, estaría la interfaz de usuario.

Así se dota al diseño de dos características cruciales: la independencia de la interfaz (de esta forma se consigue poder construir la interfaz textual y la gráfica sobre el mismo motor de simulación) y, como se verá en el siguiente apartado, la posibilidad de que motor e interfaz se ejecuten concurrentemente.

Identificación de la concurrencia

En este caso, la interfaz de usuario y el motor de simulación (el resto de los objetos) pueden y deben ejecutarse concurrentemente. De esta forma, mientras se esté efectuando la simulación no se bloqueará la interacción con el usuario.

No obtante, mientras el procesador esté ejecutando instrucciones, se deberán bloquear las acciones que impliquen escritura de atributos, ya que si se permitieran podrían producirse inconsistencias. Las acciones de consulta de valores no entrañan ningún peligro.

Asignación de subsitemas a procesadores y tareas

El sistema será implantado en ordenadores monoprocesador. Sin embargo, se usarán los recursos habituales de multiprogramación para simular la concurrencia de subsistemas (mediante distintos procesos o hilos de ejecución).

Administración de almacenes de datos

El volumen de datos manejados por la aplicación es mínimo, por lo que no es necesario contemplar la posibilidad de emplear ningún sistema de gestión de bases de datos.

Los únicos datos persistentes entre ejecuciones (las variables de configuración) se almacenan en ficheros de texto en disco.

Selección de una implementación de control de software

Claramente se trata de un sistema controlado por sucesos. De esta forma se permite nuevamente la independencia entre la interfaz y el motor de simulación, ya que la interfaz de usuario está siempre preparada para recibir nuevas peticiones sin quedarse bloqueada.

En el capítulo 9.10 de [Rumbaugh 1996] se enumeran algunos de los entornos de arquitectura más comunes. De ellos, la parte de interfaz de usuario se ajusta claramente al tipo Interfaz Interactiva, mientras que el núcleo del simulador se equipara con el tipo de Simulación Dinámica.

A la hora de diseñar la interfaz de usuario, especialmente la versión gráfica, tiene una especial importancia aislar las operaciones de la forma de invocarlas, ya que será posible ejecutar una misma acción de varias formas, por ejemplo, mediante el acceso al menú de la aplicación, pulsando un botón, mediante una combinación de teclas de acceso rápido, etc.

3.3.2.2. Diseño de objetos

Durante el diseño de objetos se ejecuta la estrategia seleccionada durante el diseño del sistema y se rellenan los detalles. Los objetos descubiertos durante el análisis sirven como esqueleto del diseño y, en este punto, se debe escoger entre distintas formas de implementarlos, con el objetivo de minimizar el tiempo de ejecución, la memoria y el coste. Las operaciones deben expresarse en forma de algoritmos. Las clases, atributos y asociaciones del análisis deben implementarse en forma de estructuras de datos específicas.

El modelo de objetos describe las clases de objetos que hay en el sistema, incluyendo sus atributos y las operaciones que admiten. El modelo funcional describe la forma en que responde el sistema frente a sucesos externos. La estructura de control del sistema se deriva fundamentalmente del modelo dinámico.

Durante el diseño de objetos se deben llevar a cabo los pasos siguientes:

- Combinar los tres modelos para obtener operaciones aplicables a clases.
- Diseñar algoritmos para implementar las operaciones.
- Optimar las vías de acceso a los datos.
- Implementar el control para interacciones externas.
- Ajustar la estructura de clases para incrementar la herencia.
- Diseñar asociaciones.
- Determinar la representación de los objetos.
- Empaquetar la clase y las asociaciones en módulos.

Nuevamente, sólo se detallarán aquellos apartados relevantes durante el diseño de *ENS2001* y, por último, se presentará una descripción detallada de cada una de las clases al término de la fase de diseño.

Se han renombrado las clases, para distinguir entre análisis y diseño, y se han añadido algunas nuevas. Estos cambios quedan recogidos en la Tabla 3.3.4.

Diseño	Análisis
CBancoRegistros	Banco de Registros
CConfiguración	
CEntradaSalida	Entrada/Salida
CGestorInstrucciones	
CGestorPuntosRuptura	
CInstrucción y derivadas	
CMemoria	Memoria
CProcesador	Procesador

Tabla 3.3.4. Clases en el Análisis y en el Diseño

Combinación de los tres modelos

Después del análisis, se tienen los modelos de objetos, dinámico y funcional. No obstante, el primero de ellos es el entorno principal alrededor del cual se construye el diseño. El modelo de objetos del análisis puede no mostrar operaciones. Se deben transformar las acciones y actividades del modelo dinámico y los procesos del modelo funcional en operaciones asociadas a las clases del modelo de objetos.

Como resultado de esta transformación, se completa el diseño de las clases ya con todos sus atributos y métodos, como se muestra en la Figura 3.3.1 (sólo se muestran los atributos, la descripción completa de las clases se detalla al final de este apartado).

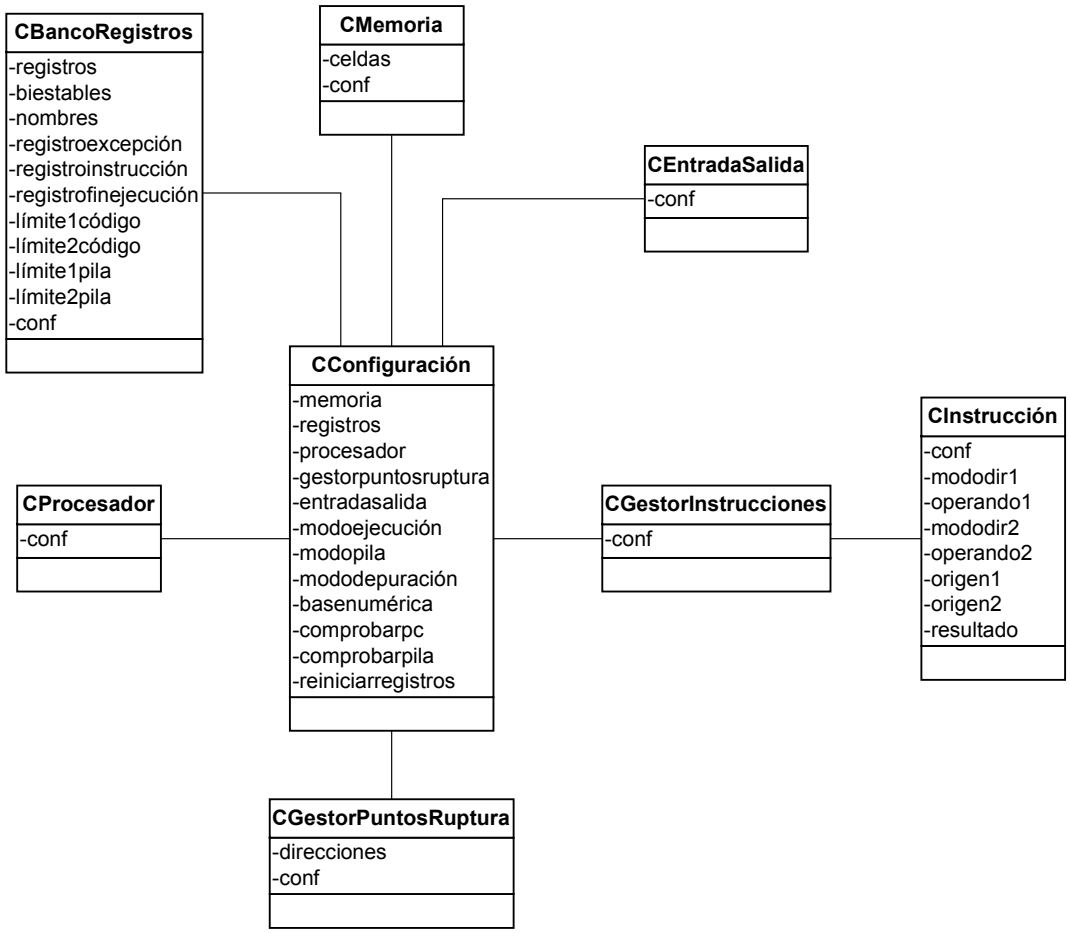


Figura 3.3.1. Modelo de Objetos del Simulador (Diseño)

El tratamiento de los puntos de ruptura se ha extraído de la clase `Memoria`, creándose una nueva clase `GestorPuntosRuptura` al efecto. De esta forma, si en el futuro se necesita cambiar dicho tratamiento (por ejemplo, para introducir la posibilidad de definir puntos de ruptura condicionales o cualquier otra mejora), no será necesario modificar la clase `Memoria`, que en realidad modeliza algo totalmente ajeno a los puntos de ruptura (aunque en esta primera aproximación, cada posible punto de ruptura esté asociado a una posición de memoria).

También se ha extraído de la clase `Procesador` toda la implementación de las instrucciones. Ahora las instrucciones son objetos independientes. El procesador, una vez haya decodificado la instrucción que haya leído de la memoria, solicitará al gestor de instrucciones (otra de las clases añadidas al modelo, `CGestorInstrucciones`) que le dé una referencia a un objeto que represente dicha instrucción, e invocará al método `Ejecutar` de dicho objeto.

Diseño de algoritmos

Cada operación especificada en el modelo funcional debe ser formulada como un algoritmo. En este caso, no hay ninguna operación que requiera diseñar un algoritmo complejo, ya que todas son operaciones básicas (leer, escribir, operaciones aritméticas, operaciones lógicas y comparaciones).

Los atributos especificados tampoco requieren del diseño de complejas estructuras de datos. Atributos como los registros, biestables o las posiciones de memoria se almacenarán en vectores de enteros, cuyos índices serán identificadores de registros, identificadores de biestables y direcciones de memoria, respectivamente. El resto de atributos serán enteros o cadenas de caracteres, dependiendo si almacenan valores numéricos o textuales.

Implementación del control

Tal y como se ha planteado en el análisis, el método `Ejecutar` de la clase `Procesador` es muy complejo, ya que debe albergar en su interior el comportamiento de todas y cada una de las instrucciones del lenguaje ensamblador. Esta forma de proceder es poco manejable a la hora del diseño, ya que, aparte de concentrar toda la funcionalidad de la simulación en una única clase, a la hora de introducir cambios en el ensamblador, como modificar el comportamiento de una instrucción o añadir una nueva, el coste de estas modificaciones es enorme.

Por tanto, se introduce el uso de una nueva clase `Instrucción`, de la que se derivarán por herencia todas y cada una de las instrucciones del lenguaje ensamblador. Cada una de ellas modelizará el comportamiento de una instrucción. Para modificar dicho comportamiento bastará modificar la clase concreta y añadir nuevas instrucciones será tan sencillo como crear nuevas clases que hereden de la clase genérica `Instrucción`.

Todas las instrucciones heredan de una clase base `CInstrucción`, que implementa atributos y métodos comunes a todas ellas (esta relación de herencia se muestra en la Figura 3.3.3). De esta forma, para extender el juego de instrucciones, o modificar el comportamiento de alguna de ellas, no es necesario modificar la clase `Procesador` en absoluto. El procesador sólo se ocupa de la tarea de leer la instrucción, decodificarla, mandar que se ejecute (ignorando por completo su implementación) y comprobar si debe parar o seguir ejecutando.

Para cada instrucción, el procesador ejecutará la secuencia *fetch* instrucción, *fetch* operandos, actualizar `PC`. A continuación, solicitará a una clase intermedia, llamada `GestorInstrucciones`, que instancie un objeto de la clase correspondiente a la instrucción que ha leído de memoria, a la que pasará como argumentos los operandos leídos y cederá el control para que simule la ejecución de la instrucción.

El modelo dinámico se extiende para mostrar el funcionamiento de esta solución propuesta (Figura 3.3.2). El procesador llama al método `ObtenerInstrucción` del gestor de instrucciones, con la instrucción y sus operandos leídos de la memoria. El gestor de instrucciones llama al constructor de la clase correspondiente a la instrucción que se quiere ejecutar y devuelve una referencia a dicho objeto. El procesador únicamente debe invocar entonces al método `Ejecutar` de la referencia que ha recibido.

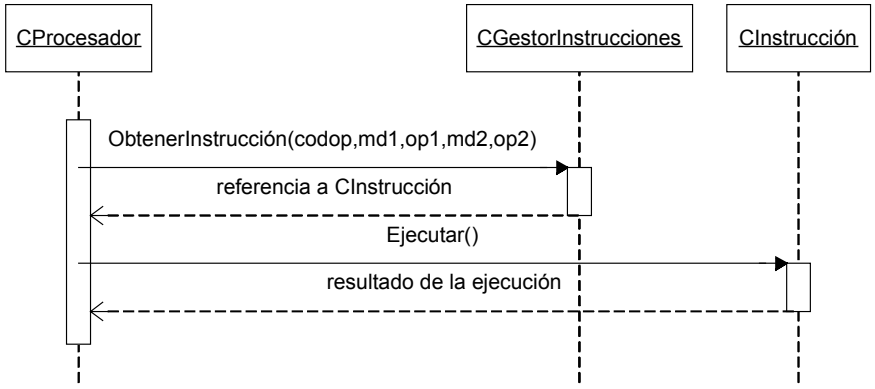


Figura 3.3.2. Extensión del Modelo Dinámico

Ajuste de la herencia

En este punto ya se han visto las dos clases genéricas que se van a emplear en el diseño. La primera de ellas se detectó durante el análisis, la clase `InterfazUsuario`, que aglutina las funciones de interfaz compartidas entre las dos versiones propuestas (textual y gráfica). La segunda se acaba de ver, y es la clase `Instrucción`, que proporciona el esqueleto de ejecución genérico para todas las instrucciones del lenguaje ensamblador que se está simulando (Figura 3.3.3).

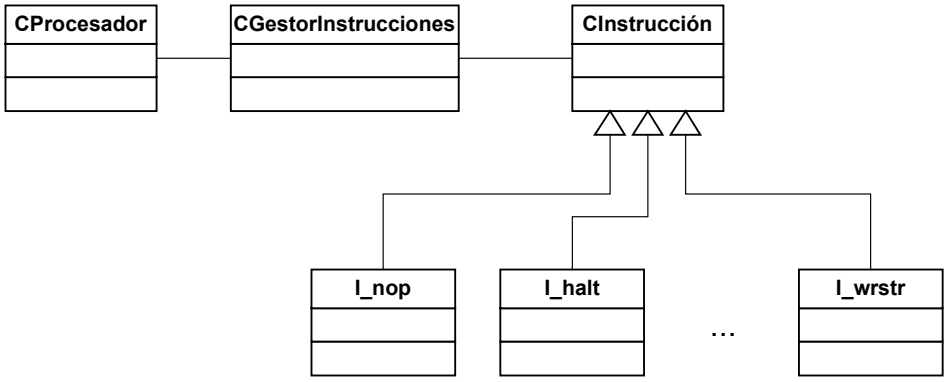


Figura 3.3.3. Gestión de Instrucciones

Diseño de asociaciones

Gran parte de los objetos que forman parte del sistema se instanciarán al comienzo de la ejecución y persistirán hasta el final de la misma. Las asociaciones entre ellos son todas de multiplicidad "uno a uno". Se ha optado por la creación de una nueva clase que guarde las referencias a todos los objetos del modelo, de forma que, desde cada uno de ellos, conociendo únicamente la referencia a este nuevo objeto, se pueda acceder a todos los demás.

Por otra parte, se ha optado por trasladar a esta nueva clase la responsabilidad de gestionar las variables de configuración del sistema. Así, se ha denominado a la nueva clase `CConfiguración`.

A continuación se va a efectuar un repaso exhaustivo de las clases, las características de sus atributos y la funcionalidad de sus métodos:

Clase CBancoRegistros

Esta clase es la que modela el comportamiento del banco de registros de la máquina virtual. Por tanto deberá almacenar el valor de todos y cada uno de los registros, así como de los biestables de estado, y deberá proveer métodos para la lectura y modificación de dichos valores.

El banco de registros se compone de los registros `PC`, `SP`, `IX`, `IY`, `SR`, `A`, `R0`...`R9`. Los biestables de estado son seis: `Z`, `C`, `V`, `P`, `S` y `H`. Los valores contenidos en los registros son enteros de 16 bits en complemento a 2, por tanto en el rango –32768 a 32767. Los valores que pueden tomar los biestables son 0 ó 1. El valor del registro de estado (`SR`) depende del valor de los biestables, calculándose según la Tabla 3.3.5.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	H	S	P	V	C	Z

Tabla 3.3.5. Biestables de Estado

Los métodos de esta clase pueden ser invocados desde la interfaz de usuario, si éste desea consultar el contenido de algún registro o modificar su valor.

CBancoRegistros
- registros: `Entero[16]`
- biestables: `Entero[6]`
- nombres: `Cadena[16]`
- registroexcepción: `Entero`
- registroinstrucción: `Entero`
- registrofinejecución: `Entero`
- límite1código: `Entero`
- límite2código: `Entero`
- límite1pila: `Entero`
- límite2pila: `Entero`
- conf: `CConfiguración`

+ CBancoRegistros(IN configuración: ^`CConfiguración`) + ~CBancoRegistros() + Escribir(IN registro: `Entero`, IN valor: `Entero`) + EscribirBiestableEstado(IN biestable: `Entero`, IN valor: `Entero`) + EscribirLímite1Código(IN límite: `Entero`) + EscribirLímite2Código(IN límite: `Entero`) + EscribirLímite1Pila(IN límite: `Entero`) + EscribirLímite2Pila(IN límite: `Entero`) + EscribirRegistroExcepción(IN valor: `Entero`) + EscribirRegistroFinEjecución(IN valor: `Entero`) + EscribirRegistroInstrucción(IN valor: `Entero`) + Leer(IN registro: `Entero`, OUT valor: `Entero`) + LeerBiestableEstado(IN biestable: `Entero`, OUT valor: `Entero`) + LeerIdRegistro(IN nombre: `Cadena`, OUT registro: `Entero`) + LeerNombreRegistro(IN registro: `Entero`, OUT nombre: `Cadena`) + Límite1Código(): `Entero` + Límite2Código(): `Entero` + Límite1Pila(): `Entero` + Límite2Pila(): `Entero` + RegistroExcepción(): `Entero` + RegistroFinEjecución(): `Entero` + RegistroInstrucción(): `Entero` + Reiniciar()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
registros	Almacena el contenido de los registros. El vector tiene tantos elementos como número de registros (16 en este caso)
biestables	Almacena el contenido de los biestables. El vector tiene tantos elementos como número de biestables (6 en este caso)
nombres	Almacena el nombre de cada registro. De esta forma se puede ampliar el banco de registros arbitrariamente o cambiar la denominación de alguno de ellos e, incluso, definir alias. El vector tiene tantos elementos como número de registros (16 en este caso), y cada elemento se corresponde con el de igual índice del vector registros.
registroexcepción	Almacena el contenido del registro de excepción.
registroinstrucción	Almacena la última instrucción leída por el procesador (código de operación + modos de direccionamiento + operandos).
registrofinejecución	Almacena el código de la causa que detuvo la simulación.
límite1código	Almacena el límite inferior de la zona de memoria correspondiente al código
límite2código	Almacena el límite superior de la zona de memoria correspondiente al código
límite1pila	Almacena el límite inferior de la zona de memoria correspondiente a la pila del sistema.
límite2pila	Almacena el límite superior de la zona de memoria correspondiente a la pila del sistema.
conf	Almacena, en el momento de crear el objeto de la clase, una referencia a un objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CBancoRegistros**
Parámetros	IN configuración: ^CConfiguración
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Guarda el valor dado por el parámetro configuración en el atributo conf. Inicializa los valores contenidos en los registros invocando al método Reiniciar.
Errores	Ninguno.

Método	**~CBancoRegistros**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Escribir**
Parámetros	IN registro: Entero IN valor: Entero
Salida	Ninguna
Comportamiento	Almacena en el registro indicado por el primer parámetro el valor indicado por el segundo parámetro. Si registro == SP, se actualizan los límites de la zona de pila: Si valor < límite1pila Entonces límite1pila = valor. Si valor > límite2pila Entonces límite2pila = valor. Si SP invade la zona de código y está activada dicha comprobación en la configuración de la herramienta, se generará una excepción (*SP invadió la zona de código*). Si registro == PC, se debe comprobar si el nuevo valor invade la zona de pila, siempre que esté activada dicha comprobación en la configuración de la herramienta. En caso afirmativo se generará una excepción (*PC invadió la zona de pila*).
Errores	Ninguno.

Método	**EscribirBiestableEstado**
Parámetros	IN biestable: Entero IN valor: Entero
Salida	Ninguna
Comportamiento	Almacena en el biestable indicado por el primer parámetro el valor indicado por el segundo parámetro.
Errores	Ninguno.

Método	**EscribirLímite1Código**
Parámetros	IN límite: Entero
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo límite1código.
Errores	Ninguno.

Método	**EscribirLímite2Código**
Parámetros	IN límite: Entero
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo límite2código.
Errores	Ninguno.

Método	**EscribirLímite1Pila**
Parámetros	IN limite: Entero
Salida	Ninguna

Comportamiento	Actualiza el valor del atributo `límite1pila`.
Errores	Ninguno.

Método	**EscribirLímite2Pila**
Parámetros	IN límite: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `límite2pila`.
Errores	Ninguno.

Método	**EscribirRegistroExcepción**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `registroexcepción`.
Errores	Ninguno.

Método	**EscribirRegistroFinEjecución**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `registrofinejecución`.
Errores	Ninguno.

Método	**EscribirRegistroInstrucción**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `registroinstrucción`.
Errores	Ninguno.

Método	**Leer**
Parámetros	IN registro: `Entero` OUT valor: `Entero`
Salida	Ninguna
Comportamiento	Devuelve en el parámetro de salida el valor contenido por el registro indicado en el parámetro de entrada.
Errores	Ninguno.

Método	**LeerBiestableEstado**
Parámetros	IN biestable: `Entero` OUT valor: `Entero`
Salida	Ninguna
Comportamiento	Devuelve en el parámetro de salida (`valor`) el valor contenido por el biestable de estado indicado en el parámetro de entrada (`biestable`).
Errores	Ninguno.

Método	**LeerIdRegistro**
Parámetros	IN nombre: `Cadena` OUT registro: `Entero`
Salida	Ninguna
Comportamiento	Devuelve en el parámetro de salida (`registro`) el identificador del registro cuyo nombre se corresponde con el parámetro de entrada (`nombre`).
Errores	Ninguno.

Método	**LeerNombreRegistro**
Parámetros	IN registro: `Entero` OUT nombre: `Cadena`
Salida	Ninguna

Comportamiento	Devuelve en el parámetro de salida (`nombre`) el nombre del registro cuyo indentificador señala el parámetro de entrada (`registro`).
Errores	Ninguno.

Método	**Límite1Código**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `límite1código`.
Errores	Ninguno.

Método	**Límite2Código**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `límite2código`.
Errores	Ninguno.

Método	**Límite1Pila**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `límite1pila`.
Errores	Ninguno.

Método	**Límite2Pila**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `límite2pila`.
Errores	Ninguno.

Método	**RegistroExcepción**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `registroexcepción`.
Errores	Ninguno.

Método	**RegistroFinEjecución**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `registrofinejecución`.
Errores	Ninguno.

Método	**RegistroInstrucción**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `registroinstrucción`.
Errores	Ninguno.

Método	**Reiniciar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Pone a cero el contenido de todos los registros, salvo el puntero de pila. La zona de código puede dejar dos huecos en la memoria, por encima y por debajo. Se calcula cuál es el mayor y, si la pila funciona de forma creciente, se coloca en la primera posición del hueco mayor. Si funciona de forma decreciente, se coloca en la última posición del hueco mayor.
Errores	Ninguno.

Clase CConfiguración

Esta clase se encarga de mantener las opciones de configuración acerca del comportamiento del simulador.

Para simplificar las relaciones entre todos los objetos del modelo, se guarda una referencia de cada uno en esta clase, de forma tal que basta con que un objeto mantenga la referencia a la clase `CConfiguración` para tener acceso a todos los objetos.

CConfiguración
- memoria: ^CMemoria - registros: ^CBancoRegistros - procesador: ^CProcesador - gestorpuntosruptura: ^CGestorPuntosRuptura - entradasalida: ^CEntradaSalida - modoejecución: Entero - modopila: Entero - mododepuración: Booleano - basenumérica: Entero - comprobarpc: Booleano - comprobarpila: Booleano - reiniciarregistros: Booleano
+ BancoRegistros(): ^CBancoRegistros + BaseNumérica(): Entero + CargarConfiguración(): Entero + ComprobarPC(): Booleano + ComprobarPila(): Booleano + CConfiguración() + ~CConfiguración() + EntradaSalida(): ^CEntradaSalida + EscribirBancoRegistros(IN regs: ^CBancoRegistros) + EscribirBaseNumérica(IN valor: Entero) + EscribirComprobarPC(IN valor: Booleano) + EscribirComprobarPila(IN valor: Booleano) + EscribirEntradaSalida(IN es: ^CEntradaSalida) + EscribirGestorPuntosRuptura(IN gpr: ^CGestorPuntosRuptura) + EscribirMemoria(IN mem: ^CMemoria) + EscribirModoDepuración(IN valor: Booleano) + EscribirModoEjecución(IN valor: Entero) + EscribirModoPila(IN valor: Entero) + EscribirProcesador(IN proc: ^CProcesador) + EscribirReiniciarRegistros(IN valor: Booleano) + GestorPuntosRuptura(): ^CGestorPuntosRuptura + GuardarConfiguración(): Entero + Memoria(): ^CMemoria + ModoDepuración(): Booleano + ModoEjecución(): Entero + ModoPila(): Entero + Procesador(): ^CProcesador + ReiniciarRegistros(): Booleano

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
memoria	Almacena la referencia a un objeto de la clase `CMemoria`.
registros	Almacena la referencia a un objeto de la clase `CBancoRegistros`.
procesador	Almacena la referencia a un objeto de la clase `CProcesador`.
gestorpuntosruptura	Almacena la referencia a un objeto de la clase `CGestorPuntosRuptura`.
entradasalida	Almacena la referencia a un objeto de la clase `CEntradaSalida`.
modoejecución	Almacena el valor de la opción de configuración relativa al modo de ejecución.
modopila	Almacena el valor de la opción de configuración relativa al modo de funcionamiento de la pila del simulador.
mododepuración	Almacena el valor de la opción de configuración relativa al modo de depuración.
basenumérica	Almacena el valor de la opción de configuración relativa a la base de representación de los números enteros en el simulador.
comprobarpc	Almacena el valor de la opción de configuración relativa a la comprobación de si el contador de programa invade la zona de pila.
comprobarpila	Almacena el valor de la opción de configuración relativa a la comprobación de si el puntero de pila invade la zona de código.
reiniciarregistros	Almacena el valor de la opción de configuración relativa a reiniciar el banco de registros antes de cada ejecución.

Método	**BancoRegistros**
Parámetros	Ninguno
Salida	`^CBancoRegistros`
Comportamiento	Devuelve el valor del atributo `bancoregistros`.
Errores	Ninguno.

Método	**BaseNumérica**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `basenumérica`.
Errores	Ninguno.

Método	**CargarConfiguración**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Carga el valor de las variables de configuración desde el fichero de configuración (`ens2001.cfg`). Devuelve 0 como valor de retorno. Si el fichero no existe, se crea uno con los valores de configuración por defecto indicados en el apartado 3.2.3 (Análisis del Simulador).
Errores	Devuelve –1 como valor de retorno, si hubo algún problema en la lectura del fichero de configuración.

Método	**ComprobarPC**
Parámetros	Ninguno
Salida	`Booleano`
Comportamiento	Devuelve el valor del atributo `comprobarpc`.
Errores	Ninguno.

Método	**ComprobarPila**
Parámetros	Ninguno
Salida	`Booleano`
Comportamiento	Devuelve el valor del atributo `comprobarpila`.
Errores	Ninguno.

Método	**CConfiguración**
Parámetros	Ninguno

Salida	Ninguna
Comportamiento	Es el constructor de la clase. Inicializa las variables de configuración a los valores por defecto, que son los indicados en el apartado 3.2.3 (Análisis del Simulador).
Errores	Ninguno.

Método	**~CConfiguración**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**EntradaSalida**
Parámetros	Ninguno
Salida	`^CEntradaSalida`
Comportamiento	Devuelve el valor del atributo `entradasalida`.
Errores	Ninguno.

Método	**EscribirBancoRegistros**
Parámetros	IN regs: `^CBancoRegistros`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `registros`.
Errores	Ninguno.

Método	**EscribirBaseNumérica**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `basenumérica`.
Errores	Ninguno.

Método	**EscribirComprobarPC**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `comprobarpc`.
Errores	Ninguno.

Método	**EscribirComprobarPila**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `comprobarpila`.
Errores	Ninguno.

Método	**EscribirEntradaSalida**
Parámetros	IN es: `^CEntradaSalida`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `entradasalida`.
Errores	Ninguno.

Método	**EscribirGestorPuntosRuptura**
Parámetros	IN gpr : `^CGestorPuntosRuptura`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `gestorpuntosruptura`.
Errores	Ninguno.

Método	**EscribirMemoria**
Parámetros	IN mem: `^CMemoria`
Salida	Ninguna

Comportamiento	Actualiza el valor del atributo `memoria`.
Errores	Ninguno.

Método	**EscribirModoDepuración**
Parámetros	IN valor: `Booleano`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `mododepuración`.
Errores	Ninguno.

Método	**EscribirModoEjecución**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `modoejecución`.
Errores	Ninguno.

Método	**EscribirModoPila**
Parámetros	IN valor: `Entero`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `modopila`.
Errores	Ninguno.

Método	**EscribirProcesador**
Parámetros	IN proc: `^CProcesador`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `procesador`.
Errores	Ninguno.

Método	**EscribirReiniciarRegistros**
Parámetros	IN valor: `Booleano`
Salida	Ninguna
Comportamiento	Actualiza el valor del atributo `reiniciarregistros`.
Errores	Ninguno.

Método	**GestorPuntosRuptura**
Parámetros	Ninguno
Salida	`^CGestorPuntosRuptura`
Comportamiento	Devuelve el valor del atributo `gestorpuntosruptura`.
Errores	Ninguno.

Método	**GuardarConfiguración**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda el valor de las variables de configuración en el fichero de configuración (`ens2001.cfg`). Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si hubo algún problema al escribir el fichero de configuración.

Método	**Memoria**
Parámetros	Ninguno
Salida	`^CMemoria`
Comportamiento	Devuelve el valor del atributo `memoria`.
Errores	Ninguno.

Método	**ModoDepuración**
Parámetros	Ninguno
Salida	`Booleano`

Comportamiento	Devuelve el valor del atributo `mododepuración`.
Errores	Ninguno.

Método	**ModoEjecución**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `modoejecución`.
Errores	Ninguno.

Método	**ModoPila**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `modopila`.
Errores	Ninguno.

Método	**Procesador**
Parámetros	Ninguno
Salida	`^CProcesador`
Comportamiento	Devuelve el valor del atributo `procesador`.
Errores	Ninguno.

Método	**ReiniciarRegistros**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `reiniciarregistros`.
Errores	Ninguno.

Clase CEntradaSalida

Esta clase es la que modela el comportamiento del módulo de entrada/salida del simulador. Se encarga de interactuar con la consola, a petición de las instrucciones de lectura y escritura.

Las funciones están sobrecargadas, de manera que existen tres de lectura y tres de escritura, según sean los parámetros, enteros, caracteres o cadenas de caracteres.

La lectura se hace mediante el método auxiliar `LeerEntrada`, que se encarga de encapsular el tratamiento de la consola, devolviendo la entrada capturada en una cadena de caracteres.

CEntradaSalida
- conf : `^CConfiguración`
+ CEntradaSalida(IN configuración: `^CConfiguración`) + ~CEntradaSalida() + Escribir(IN cadena: `Cadena`) + Escribir(IN carácter: `Carácter`) + Escribir(IN entero: `Entero`) + Leer(OUT cadena: `Cadena`): `Entero` + Leer(OUT carácter: `Carácter`) : `Entero` + Leer(OUT entero: `Entero`) : `Entero` - LeerEntrada(OUT entrada: `Cadena`)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, el objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CEntradaSalida**
Parámetros	IN configuración: `^CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Guarda el valor dado por el parámetro `configuración` en el atributo `conf`.
Errores	Ninguno.

Método	**~CEntradaSalida**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Escribir**
Parámetros	IN cadena: `Cadena`
Salida	Ninguna
Comportamiento	Escribe en la consola la cadena indicada por el parámetro de entrada.
Errores	Ninguno.

Método	**Escribir**
Parámetros	IN carácter: `Carácter`
Salida	Ninguna
Comportamiento	Escribe en la consola el carácter indicado por el parámetro de entrada.
Errores	Ninguno.

Método	**Escribir**
Parámetros	IN entero: `Entero`
Salida	Ninguna
Comportamiento	Escribe en la consola el entero indicado por el parámetro de entrada. Lo escribirá en formato decimal o hexadecimal según la configuración del simulador en ese momento. Para ello, se enviará el mensaje `conf.BaseNumérica`.
Errores	Ninguno.

Método	**Leer**
Parámetros	OUT cadena: `Cadena`
Salida	`Entero`
Comportamiento	Lee una cadena de caracteres desde la consola y la devuelve en el parámetro de salida. Se devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Leer**
Parámetros	OUT carácter: `Carácter`
Salida	`Entero`
Comportamiento	Lee un carácter desde la consola y lo devuelve en el parámetro de salida. Se devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Leer**
Parámetros	OUT entero: `Entero`
Salida	`Entero`

Comportamiento	Lee un entero desde la consola y lo devuelve en el parámetro de salida. Se devuelve 0 como valor de retorno.
Errores	Si la cadena de entrada no corresponde con un entero válido, se asumirá que se ha leído el entero 0, y se devolverá –1 como valor de retorno.

Método	**LeerEntrada**
Parámetros	OUT entrada: `Cadena`
Salida	Ninguna
Comportamiento	Lee la consola hasta que se pulsa la tecla INTRO y devuelve los caracteres leídos en el parámetro de salida, en forma de cadena de caracteres.
Errores	Ninguno.

Clase CGestorInstrucciones

Esta clase es responsable de manejar los objetos de tipo `CInstrucción` que el procesador ejecutará. Para ello, cuenta con un único método responsable de seleccionar el objeto de la clase adecuada partiendo del código de operación proporcionado por el procesador.

CGestorInstrucciones
- conf: ^`CConfiguración`
+ CGestorInstrucciones(configuración: ^`CConfiguración`) + ~CGestorInstrucciones() + ObtenerInstrucción(IN codop: `Entero`, IN md1: `Entero`, IN op1: `Entero`, IN md2: `Entero`, IN op2: `Entero`): ^`CInstrucción`

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CGestorInstrucciones**
Parámetros	IN configuración: ^`CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Inicializa el valor del atributo `conf` con el valor del parámetro de entrada `configuración`.
Errores	Ninguno.

Método	**~CGestorInstrucciones**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**ObtenerInstrucción**
Parámetros	IN codop: `Entero` (código de operación) IN md1: `Entero` (modo de direccionamiento del primer operando) IN op1: `Entero` (valor del primer operando) IN md2: `Entero` (modo de direccionamiento del segundo operando) IN op2: `Entero` (valor del segundo operando)
Salida	^`CInstrucción`

Comportamiento	A partir de los parámetros de entrada devuelve un puntero a un objeto de la clase genérica `CInstrucción`, que implementa la instrucción solicitada con los operandos indicados. Para saber qué objeto debe devolver, se basa en el contenido del parámetro de entrada `codop` (que indica el código de operación de la instrucción que se pretende ejecutar). Una vez averiguado, llama al constructor de dicha clase con el resto de parámetros (`md1`, `op1`, `md2` y `op2`), y devuelve un puntero al objeto obtenido. Para añadir nuevas instrucciones, bastará con modificar este método del gestor de instrucciones (y crear las nuevas clases derivadas de `CInstrucción`, por supuesto).
Errores	Si el código de instrucción no es válido, devuelve un puntero nulo.

Clase CGestorPuntosRuptura

Esta clase tiene la responsabilidad de manejar la información acerca de los puntos de ruptura definidos en el simulador.

Se decide que el atributo sea un vector de lógicos ya que, aunque se malgasta un poco más de memoria, el acceso de consulta y modificación es mucho más ágil.

CGestorPuntosRuptura
- conf: ^`CConfiguración` - direcciones: `Booleano[65536]`
+ CGestorPuntosRuptura(IN configuración: ^`CConfiguración`) + ~CGestorPuntosRuptura() + Consultar(IN dirección: `Entero`): `Booleano` + Intercambiar(IN dirección: `Entero`): `Booleano` + Reiniciar()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, el objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
direcciones	Almacena una relación de las direcciones de memoria en las que se han activado puntos de ruptura incondicionales.

Método	**CGestorPuntosRuptura**
Parámetros	IN configuración: ^`CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Inicializa el valor del atributo `conf` con el valor del parámetro de entrada `configuración`. Al crear el objeto no hay ningún punto de ruptura definido, por lo que todos los elementos del vector `direcciones` se inicializarán a cero.
Errores	Ninguno.

Método	**~CGestorPuntosRuptura**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Consultar**
Parámetros	IN dirección: `Entero`
Salida	`Booleano`
Comportamiento	Devuelve `cierto` si hay un punto de ruptura activo en la dirección indicada por el parámetro de entrada y `falso` en caso contrario.
Errores	Ninguno.

Método	**Intercambiar**
Parámetros	IN dirección: `Entero`
Salida	`Booleano`
Comportamiento	Si existe un punto de ruptura incondicional activo en la dirección indicada por el parámetro de entrada, lo desactiva y, si no existe, lo activa. Devuelve `cierto` si se ha activado un punto de ruptura y `falso` si se ha desactivado.
Errores	Ninguno.

Método	**Reiniciar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Elimina todos los puntos de ruptura activos.
Errores	Ninguno.

Clase CInstrucción

Esta clase es la superclase de la que heredan el resto de clases que representan cada instrucción del lenguaje ensamblador. Define los operandos que mantienen en común, así como una serie de métodos auxiliares que son comunes a bastantes de las instrucciones.

CInstrucción
- conf: ^`CConfiguración` - mododir1: `Entero` - operando1: `Entero` - mododir2: `Entero` - operando2: `Entero` - origen1: `Entero` - origen2: `Entero` - resultado: `Entero`
Aritmética(IN operación: `Entero`, IN op1: `Entero`, IN op2: `Entero`, OUT resultado: `Entero`): `Entero` + CInstrucción(IN configuración: ^`CConfiguración`, IN md1: `Entero`, IN op1: `Entero`, IN md2: `Entero`, IN op2: `Entero`) + ~CInstrucción() + Ejecutar(): `Entero` # Entrada(): `Entero` # Entrada2Operandos(): `Entero` # EntradaSalto(): `Entero` # Lógica(IN operación: `Entero`, IN op1: `Entero`, IN op2: `Entero`, OUT resultado: `Entero`): `Entero` # PopPila(OUT operando: `Entero`): `Entero` # Proceso(): `Entero` # PushPila(IN operando: `Entero`): `Entero` # Salida(): `Entero`

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, el objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
mododir1	Almacena el modo de direccionamiento del primer operando.
operando1	Almacena el valor del primer operando.
mododir2	Almacena el modo de direccionamiento del segundo operando.
operando2	Almacena el valor del segundo operando.
origen1	Es un atributo auxiliar. Sirve para almacenar el valor entero del primer operando cuando sea necesario.
origen2	Es un atributo auxiliar. Sirve para almacenar el valor entero del segundo operando cuando sea necesario.
resultado	Es un atributo auxiliar. Sirve para almacenar el valor entero del resultado de la instrucción cuando sea necesario.

Las operaciones aritméticas disponibles, a saber, suma, resta, multiplicación, división, módulo, incremento, decremento, cambio de signo y comparación, mantienen todas un mismo esquema, representado por el método `Aritmética`.

Las operaciones lógicas disponibles, a saber, and, or, xor y not, también mantienen un esquema común, que se sintetiza en el método `Lógica`.

Para modelar el comportamiento de las instrucciones, la clase `CInstrucción` define tres métodos virtuales: `Entrada`, `Proceso` y `Salida`.

- `Entrada` modela la toma de datos para ejecutar la instrucción, leyendo los operandos de memoria o registros cuando corresponda.
- `Proceso` modela la funcionalidad de la instrucción (por ejemplo, una suma, o una comparación.
- `Salida` modela la generación de nuevos datos por parte de la instrucción, por ejemplo, grabar en el acumulador el resultado de una operación.

Estos tres métodos tienen un comportamiento predefinido en esta clase, que es no hacer nada. Cada instrucción (cada subclase), según su funcionamiento, deberá redefinir estos métodos (alguna deberá redefinir los tres, alguna otra, como `NOP`, no necesita redefinir ninguno).

Método	**Aritmética**
Parámetros	IN operación: `Entero` IN op1: `Entero` IN op2: `Entero` OUT resultado: `Entero`
Salida	`Entero`
Comportamiento	Realiza todas las operaciones aritméticas, dependiendo del parámetro de entrada `operación`. Almacena el resultado en el parámetro de salida, así como los biestables de estado. Si la operación lógica seleccionada es unaria, se ignora el valor del parámetro `op2`. Devuelve 0 como valor de retorno.
Errores	En caso de que se produzca una operación de división por cero (en la operación de división o módulo), se lanza una excepción, que debe ser capturada y tratada por el objeto de la clase `CProcesador`. Si el parámetro `operación` no indica una de las operaciones aritméticas permitidas, se lanza una excepción, que deberá ser capturada y tratada por el objeto de la clase `CProcesador`.

Método	**CInstrucción**
Parámetros	IN configuracion: ^`CConfiguración` IN md1: `Entero` IN op1: `Entero` IN md2: `Entero` IN op2: `Entero`
Salida	Ninguna

Comportamiento	Es el constructor de la clase.
	Carga el valor de los atributos con los parámetros de entrada, esto es:
	`mododir1 = md1;`
	`operando1 = op1;`
	`mododir2 = md2;`
	`operando2 = op2;`
	`conf = configuración;`
	El resto de los atributos se inicializan con el valor 0.

Errores	Ninguno.

Método	**~CInstrucción**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Ejecutar**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta los métodos `Entrada`, `Proceso` y `Salida`, en este orden, recogiendo si se produce alguna condición de parada, para devolverla como tal al procesador. Devuelve el valor entero correspondiente a la condición de parada (0 si no se ha producido ninguna).
Errores	Ninguno.

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Su comportamiento por defecto es no hacer nada. En cada subclase estará definido convenientemente. Su finalidad consiste en calcular los operandos de entrada para ejecutar la instrucción. No se verifica ninguna condición de parada, luego en este caso devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Entrada2Operandos**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Es un método auxiliar empleado para calcular el valor entero de los operandos en aquellas instrucciones con dos operandos de entrada. Aplica el modo de direccionamiento guardado en `mododir1` al operando almacenado en el atributo `operando1` para calcular el valor entero del primer operando y lo almacena en el atributo `origen1`. De la misma forma, aplica el modo de direccionamiento contenido en `mododir2` al operando contenido en `operando2` para calcular el valor entero del segundo operando, almacenándolo en el atributo `origen2`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**EntradaSalto**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Es un método auxiliar empleado para calcular el valor del operando de una instrucción de salto, en las que uno de los modos de direccionamiento permitidos es el relativo a contador de programa. Calcula el valor entero del primer operando a partir del modo de direccionamiento y el valor numérico contenidos en los atributos `mododir1` y `operando1`, y lo almacena en `origen1`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Lógica**
Parámetros	IN operación: `Entero` IN op1: `Entero` IN op2: `Entero` OUT resultado: `Entero`
Salida	`Entero`
Comportamiento	Realiza todas las operaciones lógicas, dependiendo del parámetro de entrada `operación`. Almacena el resultado en el parámetro de salida `resultado`. Las operaciones lógicas no modifican los biestables de estado, como se indica en la especificación de requisitos. Si la operación lógica seleccionada es unaria, se ignora el valor del parámetro `op2`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**PopPila**
Parámetros	OUT operando: `Entero`
Salida	`Entero`
Comportamiento	Lee del banco de registros el valor del puntero de pila y, según el comportamiento definido en la configuración del simulador, recupera desde la pila el valor para el parámetro de salida `operando`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Su comportamiento por defecto es no hacer nada. En cada subclase estará definido convenientemente. Su finalidad consiste en ejecutar la instrucción en si. No se verifica ninguna condición de parada, luego en este caso devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**PushPila**
Parámetros	IN operando: `Entero`
Salida	`Entero`
Comportamiento	Lee del banco de registros el valor del puntero de pila y, según el comportamiento definido en la configuración del simulador, almacena en la pila el valor del parámetro de entrada `operando`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Su comportamiento por defecto es no hacer nada. En cada subclase estará definido convenientemente. Su finalidad consiste en almacenar los datos de salida generados por la instrucción. No se verifica ninguna condición de parada, luego en este caso devuelve 0 como valor de retorno.
Errores	Ninguno.

Clases correspondientes al juego de instrucciones

Sólo se citan los métodos que se redefinen. El constructor llama al constructor de la clase madre con los mismos parámetros en todos los casos.

Clase I_nop

No se redefine ningún método.

Clase I_halt

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Escribe el valor 1 en el biestable de estado `H`. Devuelve *"Instrucción HALT"* como condición de parada (valor de retorno).
Errores	Ninguno.

Clase I_move

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda en el atributo `resultado` el valor entero del primer operando, calculándolo a partir de los atributos `mododir1` y `operando1`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el atributo `resultado` y lo almacena como valor del segundo operando, según el modo de direccionamiento. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_push

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda en el atributo `resultado` el valor entero del primer operando, calculándolo a partir de los atributos `mododir1` y `operando1`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Llama al método `PushPila` con el valor contenido en el atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_pop

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda en el atributo `resultado` el valor devuelto por el método `PopPila`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el atributo `resultado` y lo almacena como valor del primer operando, según el modo de direccionamiento. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clases I_add, I_sub, I_mul, I_div, I_mod

Los métodos `Entrada` y `Salida` son iguales. Sólo cambia el método `Proceso`.

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Entrada2Operandos`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Aritmética`, variando el parámetro `operación` para cada clase (`add` – suma, `sub` – resta, `mul` – producto, `div` – división y `mod` – módulo), usando como operandos los atributos `origen1` y `origen2` y guardando la salida en el atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda el valor del atributo `resultado` en el registro `A`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clases I_inc, I_dec, I_neg

Los métodos `Entrada` y `Salida` son iguales. Sólo cambia el método `Proceso`.

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda en el atributo `origen1` el valor entero del primer operando, calculado a partir del valor de los atributos `operando1` y `mododir1`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Aritmética`, variando el parámetro `operación` para cada clase (`inc` – incrementar, `dec` – decrementar, `neg` – cambiar de signo), usando como operando el atributo `origen1` y guardando la salida en el atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el atributo `resultado` y lo almacena como valor del primer operando, según el modo de direccionamiento. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_cmp

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Entrada2Operandos`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Aritmética (resta, origen1, origen2, resultado)`. Por tanto, realiza una resta. Sin embargo, el valor devuelto en el atributo `resultado` no se almacenará en ningún sitio, ya que lo que se persigue como finalidad es actualizar el valor de los biestables de estado. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clases I_and, I_or, I_xor

Los métodos `Entrada` y `Salida` son iguales. Sólo cambia el método `Proceso`.

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Entrada2Operandos`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Lógica`, variando el parámetro `operación` para cada clase (`and` – y lógico, `or` – or lógico, `xor` – or exclusivo lógico), usando como operandos los atributos `origen1` y `origen2`. La salida se guarda en el atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda el valor del atributo `resultado` en el registro `A`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_not

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda en el atributo `origen1` el valor entero del primer operando, calculándolo a partir de los valores de los atributos `mododir1` y `operando1`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `Lógica (not, origen1, origen2, resultado)`, con lo que la negación lógica del valor contenido en el atributo `origen1` se almacenará en el atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el atributo `resultado` y lo almacena como valor del primer operando, según el modo de direccionamiento. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_br

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bz

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `Z`=1, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bnz

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `Z` =0, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bp

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `S`=0, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bn

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `S`=1, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bv

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `V`=1, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bnv

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `V`=0, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bc

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`

Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `C`=1, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bnc

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `C`=0, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_be

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `P`=0, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_bo

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si el biestable `P`=1, escribe en el registro `PC` el contenido del atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_call

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta el método auxiliar `EntradaSalto`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el contenido del registro `PC` y lo almacena en el atributo `origen1`. Llama al método auxiliar `PushPila` con el atributo `origen1` como entrada. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el valor del atributo `resultado` y lo escribe en el registro `PC`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_ret

Método	**Proceso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Llama al método auxiliar `PopPila` y almacena el valor devuelto en el atributo `resultado`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el valor del atributo `resultado` y lo escribe en el registro `PC`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_inchar

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`

Comportamiento	Llama al método `Leer(carácter)` del objeto de la clase `CEntradaSalida`. Codifica el carácter leído en los 8 bits inferiores de una palabra de 16 bits y lo almacena en el primer operando según el modo de direccionamiento. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_inint

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Llama al método `Leer(entero)` del objeto de la clase `CEntradaSalida`. Almacena el entero leído en el primer operando según el modo de direccionamiento. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_instr

Método	**Entrada**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Llama al método `Leer(cadena)` del objeto de la clase `CEntradaSalida`. Almacena la cadena leída carácter a carácter, a partir de la posición de memoria indicada por el primer operando y añade el carácter nulo para acabar la cadena. Si escribiendo valores en memoria se sobrepasa el límite superior, se genera la excepción *Sobrepasado el límite de memoria*. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_wrchar

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el valor correspondiente al primer operando, según el modo de direccionamiento. Toma los 8 bits inferiores y calcula el carácter correspondiente. Llama al método `Escribir(carácter)` del objeto de la clase `CEntradaSalida`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_wrint

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee el valor correspondiente al primer operando, según el modo de direccionamiento. Llama al método `Escribir(entero)` del objeto de la clase `CEntradaSalida`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Clase I_wrstr

Método	**Salida**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Lee sucesivamente valores de la memoria, a partir de la dirección indicada por el primer operando, y los va añadiendo en una cadena hasta encontrar el valor correspondiente al carácter nulo (`\0`). Llama al método `Escribir(cadena)` del objeto de la clase `CEntradaSalida`. Si leyendo valores de memoria se sobrepasa el límite superior, se genera la excepción *Sobrepasado el límite de memoria*. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Se puede apreciar que todas las instrucciones (menos `HALT`), no devuelven condición de parada. Sin embargo, podría ser interesante introducir algún otro tipo de puntos de ruptura en el futuro (por ejemplo, al ejecutar determinada instrucción o con determinados parámetros. Por ello, se mantiene en todas ellas que el tipo de datos del valor devuelto sea `Entero`.

Clase CMemoria

Esta clase es la que modela el comportamiento de la memoria de la máquina virtual. Por tanto, deberá almacenar el valor de las posiciones de memoria y deberá proveer métodos para la lectura y modificación de dichos valores.

El rango de direcciones está comprendido entre la 0 y la 65535. Los valores contenidos en dichas posiciones de memoria son enteros de 16 bits en complemento a 2, por tanto en el rango –32767 a 32768.

Los métodos de esta clase pueden ser invocados desde la interfaz de usuario, si éste desea consultar el contenido de alguna posición de memoria o modificar su valor.

No se efectúa ningún control de errores ya que se da por supuesto que los parámetros siempre van a estar dentro del rango permitido (0..65535 si se consideran los enteros siempre sin signo).

CMemoria
- celdas: `Entero[65536]` - conf: `^CConfiguración`
+ CMemoria(IN configuración: `^CConfiguración`) + ~CMemoria() + Escribir(IN dirección: `Entero`, IN valor: `Entero`) + Leer(IN dirección: `Entero`, OUT valor: `Entero`) + Reiniciar()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
celdas	Almacena el contenido de las celdas de memoria
conf	Almacena, en el momento de crear el objeto de la clase, el objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CMemoria**
Parámetros	IN configuración: `^CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Inicializa el valor del atributo `conf` con el valor del parámetro de entrada `configuración`. Los valores contenidos en el vector `celdas` se inicializan llamando al método `Reiniciar`.
Errores	Ninguno.

Método	**~CMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Escribir**
Parámetros	IN dirección: `Entero` IN valor: `Entero`
Salida	Ninguna
Comportamiento	Almacena en la posición de memoria indicada por el primer parámetro el valor contenido en el segundo parámetro.
Errores	Ninguno.

Método	**Leer**
Parámetros	IN dirección: `Entero` OUT valor: `Entero`
Salida	Ninguna
Comportamiento	Devuelve en el parámetro de salida el contenido de la posición de memoria indicada por el parámetro de entrada
Errores	Ninguno.

Método	**Reiniciar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Reinicia a cero el contenido de todas las posiciones de memoria, esto es, almacena el valor cero en todas las posiciones del vector `celdas`.
Errores	Ninguno.

Clase CProcesador

Esta clase es la que modela el comportamiento del procesador virtual. Se compone de un único método público (además del constructor). Este método es el encargado de controlar el flujo de ejecución. Funciona de la siguiente forma:

Mientras el resultado de ejecutar la anterior instrucción no implique detenerse, esto es, no se ha producido una excepción, el biestable `H` está desactivado y no está seleccionado el modo paso a paso:

- Se lee la instrucción de memoria.
- Se descodifica la instrucción.
- Se leen los operandos que proceda.
- Se ejecuta la instrucción.

Cuando la ejecución se detiene, siempre se informa el motivo de la parada.

A la hora de ejecutar la instrucción, con el código de operación y los operandos recuperados se invoca al gestor de instrucciones. Éste devolverá una referencia a la instrucción que se debe ejecutar. De esta forma, el comportamiento del procesador es homogéneo para todas las instrucciones (de hecho no sabe qué instrucción está ejecutando). Simplemente, se encarga de recoger el resultado de la ejecución.

Los dos métodos privados se emplean en el primer y tercer paso del bucle de simulación.

CProcesador
- conf: ^CConfiguración
+ CProcesador(IN configuración: ^CConfiguración) + ~CProcesador() + Ejecutar(): Entero - FetchInstrucción(): Entero - FetchOperandos(IN mododireccionamiento1: Entero, IN mododireccionamiento2: Entero, OUT operando1: Entero, OUT operando2: Entero)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase CConfiguración que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CProcesador**
Parámetros	IN configuración: ^CConfiguración
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Inicializa el valor del atributo conf con el valor del parámetro de entrada configuración y anota una referencia a sí mismo invocando el método EscribirProcesador del objeto configuración.
Errores	Ninguno.

Método	**~CProcesador**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Ejecutar**
Parámetros	Ninguno
Salida	Entero
Comportamiento	Ejecuta instrucciones hasta que se produce una condición de parada. Devuelve un código que indica el resultado de la ejecución.
Errores	Ninguno.

Método	**FetchInstrucción**
Parámetros	Ninguno
Salida	Entero
Comportamiento	Lee el contador de programa. Lee de la memoria la posición apuntada por el contador de programa, lo incrementa, extrae el código de operación y los modos de direccionamiento de la instrucción que se va a simular, y los devuelve como salida.
Errores	Ninguno.

Método	**FetchOperandos**
Parámetros	IN mododireccionamiento1: `Entero` IN mododireccionamiento2: `Entero` OUT operando1: `Entero` OUT operando2: `Entero`
Salida	Ninguna
Comportamiento	Lee de la memoria los operandos, dependiendo de los modos de direccionamiento indicados por los parámetros de entrada, actualiza el contador de programa tras la lectura y devuelve los valores leídos en los parámetros de salida.
Errores	Ninguno.

Generación de Excepciones.

Durante la simulación se pueden producir situaciones excepcionales, que obligan al procesador a detener la ejecución. El procesador almacenará en el registro de excepción el código de la excepción que se ha producido e informará a la interfaz de usuario como causa de finalización 'excepción'. Las excepciones que pueden producirse son las siguientes:

- División por cero.
  Se produce al ejecutar la instrucción `DIV` o `MOD`, si el segundo operando es cero.
- Instrucción no implementada.
  Se produce si el código de operación leído por el procesador no se corresponde con ninguna de las instrucciones del juego de instrucciones.
- Sobrepasado el límite de memoria.
  Se produce si la instrucción `INSTR` trata de escribir más allá de la última posición de memoria o la instrucción `WRSTR` trata de leer de más allá de la última posición de memoria.
- Detenido por usuario.
  Se produce cuando el usuario aborta la ejecución (en la interfaz de usuario se indica de qué forma puede hacerse).
- `PC` invade la zona de pila.
  Se produce, opcionalmente, si el usuario desea que se compruebe este hecho.
- `SP` invade la zona de código.
  Se produce, opcionalmente, si el usuario desea que se compruebe este hecho.

Fichero de configuración.

El fichero de configuración del simulador `ens2001.cfg`, es un fichero de texto compuesto por pares `VARIABLE = VALOR`, que se corresponden con las características configurables, y que permite mantener su valor entre sesión y sesión. Las variables que se encuentran en el fichero y los diferentes valores que pueden tomar se resumen en la Tabla 3.3.6 (en negrita los valores por defecto).

Si el fichero de configuración no existe, se deberá crear uno con los valores por defecto.

Variable	Valores
ModoEjecucion	**NORMAL** PASOAPASO
ModoPila	ASCENDENTE **DESCENDENTE**
ModoDepuracion	**SI** NO
BaseNumerica	**DECIMAL** HEXADECIMAL
ComprobarPC	SI **NO**
ComprobarPila	SI **NO**
ReiniciarRegistros	**SI** NO

Tabla 3.3.6. Variables de Configuración

3.3.3. Diseño de la Interfaz de Línea de Comandos

3.3.3.1. Descripción de la interfaz

La interfaz textual será semejante a la que presenta *ENS96* y, en definitiva, a la de cualquier aplicación manejada mediante línea de comandos, con una sintaxis particular y un juego de instrucciones predefinidos. No en vano, la existencia de cada comando responde a un requisito o un conjunto de requisitos del sistema.

Se va a construir una clase que dé soporte a la interfaz. Deberá contener métodos para analizar la entrada, determinando qué comando ha sido introducido, y ejecutar dicho comando, informando de los resultados o los posibles errores que se produzcan. Por otra parte, los mensajes de presentación, hasta que comienza la entrada de comandos por parte del usuario, serán responsabilidad del programa principal.

Al arrancar, la aplicación mostrará los siguientes mensajes:

```
ENS2001 para consola Win32 (32 bits)
Versión versión - fecha de publicación
```

Donde *versión* y *fecha de publicación* dependerán de la versión del sistema. A continuación mostrará un mensaje dependiendo de si el fichero de configuración se ha podido cargar o no:

```
Fichero de configuración (ens2001.cfg) cargado con éxito
```

O bien:

```
Fichero de configuración (ens2001.cfg) no encontrado o erróneo
Se usarán las opciones por defecto
```

Respectivamente. Después, se muestran una serie de mensajes que informan al usuario de cuál es la configuración activa en este momento. Por ejemplo:

```
Representación en base hexadecimal
La pila crece en direcciones crecientes
Se detendrá la ejecución si se encuentran puntos de ruptura
Ejecución Normal
No se comprobará si el PC invade el espacio de pila
```

```
No se comprobará si la pila invade el espacio del código
Se inicializarán los registros antes de ejecutar
```

Tras presentar toda esta información, la aplicación queda a la espera de que el usuario introduzca los comandos. Siempre que esté esperando instrucciones, se mostrará el indicador:

```
ENS2001>
```

Ya que la herramienta no admite un número exagerado de instrucciones, cada comando se corresponderá con un único carácter, en mayúsculas o minúsculas indistintamente, seguido de argumentos en los casos que los precisen.

Tanto los comandos como la salida que produzcan los mismos y la propia consola del simulador (tanto entrada como salida), se mostrarán entremezclados en la misma sesión de texto. La interfaz con el usuario se encarga de ir leyendo comandos, determinando a qué función corresponden y ejecutándolos, hasta que introduzca el comando que finaliza la ejecución.

Ésta es la lista completa de comandos, por orden alfabético. Se indica el requisito o requisitos que cumplen, su funcionalidad y los mensajes que muestran en la consola.

'a' (Activar/Desactivar modo Paso a Paso).

Cumple el requisito **ERS-REQ23**.

Se trata de un selector que irá alternando entre dos estados, cada vez que el usuario ejecute este comando:

- Ejecutar paso a paso. El procesador detendrá la ejecución después de cada instrucción. Si el usuario selecciona esta opción, se mostrará en la consola el mensaje:

  ```
  Ejecución Paso A Paso
  ```

- No ejecutar paso a paso. El procesador sólo detendrá la ejecución cuando encuentre un punto de ruptura (si están activos), una instrucción HALT dentro del código, o bien se produzca una excepción. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  Ejecución Normal
  ```

'b' (Activar/Desactivar Puntos de Ruptura).

Cumple el requisito **ERS-REQ32**.

Se trata de un selector que irá alternando entre dos estados, cada vez que el usuario ejecute este comando:

- Detenerse en los puntos de ruptura. El procesador detendrá la ejecución cuando encuentre un punto de ruptura definido previamente por el usuario. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  Se detendrá la ejecución si se encuentra un punto de ruptura
  ```

- No detenerse en los puntos de ruptura. El procesador ignorará cualquier punto de ruptura que encuentre. Sin embargo, se detendrá ante cualquier otra condición de parada posible (modo paso a paso, instrucción HALT o excepción). Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

```
Se ignorarán los puntos de ruptura establecidos
```

'b' dir (Activar/Desactivar un Punto de Ruptura).

Cumple el requisito **ERS-REQ22**.

Activa un punto de ruptura en la dirección de memoria indicada por el parámetro *dir* y mostrará en la consola el siguiente mensaje:

```
dir: Punto de Ruptura Activado
```

Si ya existía un punto de ruptura activo en dicha dirección, entonces se desactivará y se mostrará en la consola el siguiente mensaje:

```
dir: Punto de Ruptura Desactivado
```

'c' fichero (Cargar y Ensamblar).

Cumple los requisitos **ERS-REQ05**, **ERS-REQ06** y **ERS-REQ07**.

Con este comando, la herramienta lee el código fuente contenido en el archivo indicado por el parámetro *fichero* y lo ensambla. Si el proceso de ensamblado es correcto, coloca el código en memoria. Si no, muestra un listado con los errores producidos. Este proceso es responsabilidad única del módulo ensamblador.

El ensamblador necesita crear unos ficheros temporales de trabajo, que son *memoria.tmp*, *fuente.tmp* y *errores.tmp*. Si no pudiera crear alguno de ellos, se mostrará el siguiente error, y no se completará la operación:

```
Error generando fichero nombre_fichero
El directorio de trabajo está lleno o protegido contra escritura
```

Esto podría ocurrir, por ejemplo, si se intenta ejecutar la aplicación desde un CD-ROM, o una unidad de red con acceso exclusivo de lectura.

'd' dir num (Desensamblar).

Cumple los requisitos **ERS-REQ20** y **ERS-REQ33**.

Este comando permite desensamblar el código contenido en la memoria. En concreto, comienza a desensamblar a partir de la dirección de memoria indicada por el parámetro *dir* y desensamblará tantas instrucciones como indique el parámetro *num*.

El código desensamblado se irá mostrando en la consola hasta que se haya desensamblado la cantidad de instrucciones indicada, según el siguiente formato:

```
dir_1: instrucción_1
dir_2: instrucción_2
...
dir_num: instrucción_num
```

Al desensamblar, puede ocurrir que se lean datos en memoria que no se correspondan con ninguna instrucción (por ejemplo, al desensamblar una zona de datos). En este caso, se mostrará en el lugar de la instrucción la cadena '*NO IMPLEMENTADA*'.

```
0x4231: *NO IMPLEMENTADA*
```

En el caso de que en la instrucción que se está desensamblando se haya definido un punto de ruptura, se notificará precediendo a la línea actual con un asterisco '*'.

```
*       0x1224: PUSH .r3
```

También se indicará la dirección a la que apunta el contador de programa, si procede. Para ello, se mostrarán a la izquierda los caracteres 'PC->':

```
* PC-> 0x1224: PUSH .r3
```

Puede ocurrir que el usuario pretenda desensamblar más instrucciones de las existentes desde la dirección de comienzo hasta el final de la memoria. Si esto ocurre, se mostrará el siguiente mensaje:

```
AVISO: Se sobrepasa el límite de memoria
```

'e' (Ejecutar).

Cumple el requisito **ERS-REQ24**.

Invoca al procesador virtual para lanzar la simulación. El procesador ejecutará instrucciones hasta que se cumpla una de las siguientes condiciones:

- Ejecución Paso a Paso activa (se detiene nada más ejecutar una instrucción).
- Punto de Ruptura Activado (se detiene antes de ejecutar la instrucción).
- Excepción durante la ejecución.
- Instrucción HALT

En el caso de la ejecución paso a paso, se desensamblará y se mostrará en la consola la próxima instrucción que se va a ejecutar, con el mismo formato que en el comando 'd' (Desensamblar).

Al finalizar la simulación, la herramienta mostrará en pantalla cuál de estas causas fue la que detuvo al procesador, con alguno de los siguientes mensajes respectivamente:

```
Modo Paso A Paso. Se detuvo la ejecución
Punto de Ruptura Incondicional. Se detuvo la ejecución
Se ha generado una excepción
Fin del programa. Instrucción HALT
```

Si se ha producido una excepción, se indicará también la causa de la misma, que podrá ser alguna de las siguientes:

```
Instrucción No Implementada
División Por Cero
Contador de Programa Sobrepasa el Final de la Memoria
Contador de Programa Invade la zona de Pila
Puntero de Pila Invade la zona de Código
Ejecución Detenida por el Usuario
```

Si durante la ejecución, el procesador encuentra una instrucción de entrada por consola, esperará a la introducción del dato por parte del usuario para continuar, como se mostró en el diseño de la clase CEntradaSalida. Asimismo, si encuentra una instrucción de salida, el resultado aparecerá en la pantalla, en el punto donde esté situado el cursor en ese momento, intercalado con la salida de la propia interfaz de *ENS2001*.

El usuario puede detener la ejecución en cualquier momento, pulsando la tecla ESCAPE.

'g' (Activar/Desactivar comprobación si PC invade la zona de Pila).

Cumple el requisito **ERS-REQ27**.

Se trata de un selector que irá alternando entre dos estados, cada vez que el usuario ejecute este comando:

- Comprobar si el contador de programa invade la zona de pila. El procesador generará una excepción si se produce esta situación durante la ejecución. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  Se comprobará si PC invade el espacio de pila
  ```

- No comprobar si el contador de programa invade la zona de pila. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  No se comprobará si PC invade el espacio de pila
  ```

'h' (Ayuda).

Muestra por pantalla el contenido del fichero 'ayuda.txt' o, lo que es lo mismo, un listado con todos los comandos que admite la interfaz de usuario:

```
Referencia Rápida de Comandos Modo Consola
--------------------------------------------------------------------------
a                Activa/Desactiva Modo Paso A Paso
b                Activa/Desactiva Puntos de Ruptura
b dir            Pone/Quita un Punto de Ruptura en la dirección dir
c fichero        Carga y Ensambla el fichero
d dir num        Desensambla num instrucciones a partir de dir
e                Ejecuta el código
g                Activa/Desactiva comprobación si PC invade la zona de pila
h                Visualiza la ayuda
i                Activa/Desactiva comprobación si SP invade la zona de código
k                Modifica el Funcionamiento de la pila (Creciente/Decreciente)
l fichero        Carga una Imagen de Memoria desde fichero
m dir [valor]    Visualiza/Modifica el contenido de la Direccion dir
n base           Base de Representación Numerica (10 o 16)
o                Configuración del Simulador
p num            Vuelca num posiciones de la pila
q                Salir del Programa
r                Visualiza el Banco de Registros
r reg [valor]    Visualiza/Modifica el Contenido del Registro reg
s fichero        Guarda una Imagen de Memoria desde fichero
t                Inicializa/No Inicializa Registros al ejecutar programa
v dir num        Vuelca num posiciones de memoria a partir de la direccion dir
--------------------------------------------------------------------------
```

'i' (Activar/Desactivar comprobación si SP invade la zona de Código).

Cumple el requisito **ERS-REQ28**.

Se trata de un selector que irá alternando entre dos estados, cada vez que el usuario ejecute este comando:

- Comprobar si el puntero de pila invade la zona de código. El procesador generará una excepción si se produce esta situación durante la ejecución. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  Se comprobará si SP invade el espacio de código
  ```

- No comprobar si el puntero de pila invade la zona de código. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  No se comprobará si SP invade el espacio de código
  ```

'k' (Seleccionar Modo de Funcionamiento de la Pila).

Cumple el requisito **ERS-REQ21**.

Se trata de un selector que irá alternando entre dos estados, cada vez que el usuario ejecute este comando:

- Direcciones crecientes. La pila crece en sentido ascendente. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  La pila crece en sentido ascendente
  ```

- Direcciones decrecientes. La pila crece en sentido descendente. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

  ```
  La pila crece en sentido descendente
  ```

'l' (Cargar una Imagen de Memoria).

Cumple el requisito **ERS-REQ09**.

Este comando permite recuperar el contenido íntegro de la memoria desde un archivo que el usuario habrá creado previamente con la propia aplicación. Como la lectura se hace en formato binario, si el usuario intenta leer un fichero cualquiera que no haya sido generado por la aplicación los resultados son inesperados. En concreto, se espera leer un archivo de 128 Kbytes de longitud, con las palabras de memoria almacenadas en formato *big-endian* (esto es, primero el *byte* más significativo y a continuación el menos significativo).

Si existe algún problema a la hora de leer el archivo, se mostrará el siguiente mensaje:

```
ERROR: Leyendo el fichero nombre_de_fichero
```

'm reset' (Reiniciar el contenido de la memoria).

Este comando permite al usuario reiniciar, previa confirmación, el contenido de todas las posiciones de memoria al valor cero.

'm' dir (Visualizar el contenido de una dirección de memoria).

Cumple el requisito **ERS-REQ11**.

Este comando muestra el contenido de una posición de memoria. La posición de memoria viene indicada por el parámetro *dir*. Como resultado mostrará en la consola una línea con el siguiente formato:

```
dir: valor
```

Tanto la dirección como el valor serán expresados en decimal o hexadecimal, según la configuración actual de la herramienta.

Si el parámetro *dir* toma un valor incorrecto, es decir, no es un entero de 16 bits, comprendido en el rango 0..65535, expresado en decimal o hexadecimal, se mostrará el siguiente mensaje de error:

```
ERROR: Dirección incorrecta o fuera de rango
```

NOTA: Es importante recordar que la representación numérica que maneja el simulador es binario en complemento a 2, por lo tanto los valores enteros decimales comprendidos entre –32768 y –1 son válidos, y equivalentes a los positivos entre 32768 y 65535.

'm' dir valor (Modificar el contenido de una dirección de memoria).

Cumple el requisito **ERS-REQ13**.

Este comando edita el contenido de una posición de memoria. En concreto, introducirá en la posición de memoria indicada por el parámetro *dir* el entero indicado por el parámetro *valor*. Por tanto, ambos parámetros deberán ser enteros de 16 bits, expresados en decimal o hexadecimal, en el rango 0..65535, o bien en el caso de los decimales, también en el rango –32768..-1, al igual que en el comando anterior.

Si alguno de los parámetros, *dir* o *valor*, no contienen valores correctos se mostrará alguno de los siguientes mensajes, respectivamente:

```
Parámetro incorrecto - dir
Parámetro incorrecto - valor
```

'n' base (Seleccionar la Base de Representación Numérica).

Cumple el requisito **ERS-REQ10**.

El parámetro base indica la base de representación con la que el usuario desea que la herramienta presente todos los números enteros, tanto de direcciones, como de datos, salida por consola... Sólo se permiten los valores 10 (base decimal) y 16 (base hexadecimal). Se mostrarán los siguientes mensajes, según sea la base de representación seleccionada:

- Base decimal (base=10).

```
Representación en base decimal
```

- Base hexadecimal (base=16).

```
Representación en base hexadecimal
```

- Cualquier otra base (base distinta de 10 y 16).

```
Base de representación incorrecta – Valores posibles: 10, 16
```

'o' (Consultar la configuración actual).

Cumple el requisito **ERS-REQ31**.

Este comando muestra en la consola la configuración actual del simulador. Cada variable de configuración puede tomar dos valores. Por tanto, esta lista se compondrá escogiendo un valor de cada una de las siete parejas posibles, en el orden que se muestra a continuación:

- Representación en base decimal
  Representación en base hexadecimal
- La pila crece en sentido ascendente
  La pila crece en sentido descendente
- Se detendrá la ejecución si se encuentra un punto de ruptura
  Se ignorarán los puntos de ruptura establecidos
- Ejecución Paso a Paso
  Ejecución Normal
- Se comprobará si PC invade el espacio de pila
  No se comprobará si PC invade el espacio de pila
- Se comprobará si SP invade el espacio de código
  No se comprobará si SP invade el espacio de código
- Se inicializarán los registros antes de ejecutar
  No se inicializarán los registros antes de ejecutar

'p' num (Acceso a la Pila).

Cumple el requisito **ERS-REQ15**.

El comportamiento de este comando es análogo al comando 'v' (Visualizar un bloque de memoria), sólo que como límite del bloque tomará la dirección a la que apunte el puntero de pila, y mostrará posiciones en sentido ascendente o descendente según esté configurado el funcionamiento de la pila.

Esto quiere decir que si la pila está configurada en dirección ascendente, se mostrará el contenido de las direcciones sp, sp+1... sp+(num-1). Sin embargo, si está configurada en dirección descendente, se mostrará el contenido de las direcciones sp-(num-1), sp-(num-2) ... sp. Además, se indicará la posición del puntero de pila en la columna izquierda con los caracteres 'SP->':

```
SP-> dir: valor
     dir+1: valor
     ...

     dir+num-1: valor
```

Si el parámetro *num* no contiene un valor correcto, se mostrará el siguiente mensaje de error:

```
Parámetro incorrecto – num
```

Si el usuario pretende visualizar posiciones más allá de los límites de memoria, tanto superior como inferior, simplemente no se mostrarán.

'q' (Salir de la herramienta).

Cumple el requisito **ERS-REQ35**.

Este comando permite abandonar la ejecución de la herramienta. Mostrará en pantalla la pregunta de confirmación:

```
¿Seguro que desea salir (S/N)?
```

Si el usuario responde afirmativamente ('S' o 's'), se devuelven todos los recursos ocupados, se guarda la configuración actual en el archivo 'ens2001.cfg' y se retorna a la línea de comandos del sistema operativo. Si contesta cualquier otra cosa, se retoma la ejecución de la herramienta desde el instante en que se introdujo el comando 'q'.

'r reset' (Reiniciar el banco de registros).

Este comando permite reiniciar el contenido de todos los registros del banco de registros. Todos tomarán un valor cero, excepto el puntero de pila, cuyo valor de inicialización dependerá del modo de funcionamiento de la pila que esté seleccionado.

La zona de código puede dejar un hueco anterior y otro posterior a ella. Se calcula cuál es el mayor hueco. Entonces, si la pila crece en sentido ascendente, el puntero de pila ocupará la primera posición del hueco mayor, mientras que si crece en sentido descendente, ocupará la última posición de dicho hueco.

'r' reg (Visualizar el contenido de un registro).

Cumple el requisito **ERS-REQ16**.

Este comando muestra el contenido de un registro. El registro al que se hace referencia vienen indicado por el parámetro reg, que debe ser un identificador de registro válido, en mayúsculas o minúsculas, pero no una combinación (ejemplos: PC es válido, pc es válido, pero Pc no es válido). Como resultado mostrará en la consola una línea con el siguiente formato:

```
id_registro: valor
```

El valor será expresado en decimal o hexadecimal, según la configuración de la herramienta en ese momento.

Si no se introdujo un identificador de registro válido, se mostrará el siguiente mensaje de error:

```
ERROR: Registro inexistente
```

'r' (Visualizar el contenido del Banco de Registros).

Cumple el requisito **ERS-REQ17**.

Este comando muestra en la consola el contenido del banco de registros completo, así como los biestables de estado y los intervalos de direcciones definidos para las zonas de código y pila. Todo ello en el siguiente formato (suponiendo que todos los valores estén a cero):

Base decimal:

```
 A:      0 PC:      0 SP:      1 IX:      0 IY:      0 SR:      0
R0:      0 R1:      0 R2:      0 R3:      0 R4:      0 R5:      0
R6:      0 R7:      0 R8:      0 R9:      0  Z:0 C:0 V:0 P:0 S:0 H:0
Código [0,0] Pila [0,0]
```

Base hexadecimal:

```
 A: 0x0000 PC: 0x0000 SP: 0x0001 IX: 0x0000 IY: 0x0000 SR: 0x0000
R0: 0x0000 R1: 0x0000 R2: 0x0000 R3: 0x0000 R4: 0x0000 R5: 0x0000
R6: 0x0000 R7: 0x0000 R8: 0x0000 R9: 0x0000  Z:0 C:0 V:0 P:0 S:0 H:0
Código [0x0000,0x0000] Pila [0x0000,0x0000]:
```

‘r’ reg valor (Modificar el contenido de un registro).

Cumple el requisito **ERS-REQ18**.

Este comando edita el contenido de un registro. En concreto, introducirá en el registro indicado por el parámetro *reg* el entero indicado por el parámetro *valor*. Por tanto, el primer parámetro deberá ser un identificador de registro válido, en mayúsculas o minúsculas, pero no una combinación (ejemplos: PC es válido, pc es válido, pero Pc no es válido), mientras que el segundo deberá ser un entero de 16 bits, expresado en decimal o hexadecimal, en el rango 0..65535, o bien en el caso de los decimales, también en el rango equivalente –32768..32767.

Si alguno de los parámetros, *reg* o *valor*, no contienen valores correctos se mostrará alguno de los siguientes mensajes, respectivamente:

```
ERROR: Registro inexistente
ERROR: Valor fuera de rango
```

‘s’ fichero (Guardar una Imagen de Memoria).

Cumple el requisito **ERS-REQ08**.

Este comando permite guardar el contenido íntegro de la memoria en un fichero, cuyo nombre indica el parámetro *fichero*. Se creará un nuevo fichero con una longitud de 128Kbytes. Si el fichero ya existía, se sobrescribirá. El fichero es una imagen exacta de la memoria, almacenado en formato *big-endian*.

Si existe algún problema a la hora de grabar el archivo, se mostrará el mensaje:

```
ERROR: Guardando el fichero nombre_de_fichero
```

‘t’ (Activar/Desactivar Inicializar Registros al comenzar la ejecución).

Cumple el requisito **ERS-REQ30**.

Se trata de un selector que irá alternando entre dos estados, cada vez que el usuario ejecute este comando:

- Reiniciar registros al comienzo de la ejecución. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

```
Se inicializarán los registros antes de ejecutar
```

- No reiniciar registros al comienzo de la ejecución. Si el usuario selecciona esta opción, se mostrará en la consola el siguiente mensaje:

```
No se inicializarán los registros antes de ejecutar
```

'v' dir num (Visualizar un bloque de memoria).

Cumple el requisito **ERS-REQ12**.

Este comando permite visualizar en la consola el contenido de un bloque de memoria. El bloque comienza en la dirección de memoria indicada por el parámetro *dir* y se visualizarán un total de *num* posiciones. Por tanto, ambos parámetros deberán ser enteros de 16 bits, expresados en decimal sin signo o hexadecimal, en el rango 0..65535. En el caso de los decimales con signo, se permite el rango equivalente –32768..32767.

El resultado del comando será análogo a emplear el comando 'm dir' tantas veces como posiciones se quieran mostrar, incrementando *dir* a cada paso. Además, en el margen izquierdo, en la posición indicada por la letra Z en el ejemplo, puede aparecer la letra C si la dirección de memoria pertenece a la zona de código, y la letra P si la dirección de memoria pertenece a la zona de pila:

```
Z dir: valor
Z dir+1: valor
  ...
Z dir+num-1: valor
```

Si alguno de los parámetros, *dir* o *num*, no contiene valores correctos se mostrará alguno de los siguientes mensajes, respectivamente:

```
Parámetro incorrecto – dir
Parámetro incorrecto – num
```

Puede ocurrir que mostrando posiciones sucesivas se llegue al final de la memoria. En ese caso, se mostrarán hasta el final, y luego el mensaje:

```
AVISO: Se sobrepasa el límite de memoria
```

3.3.3.2. Modelo de clases

A continuación se presenta el modelo de clases que da soporte a toda esta funcionalidad. A la ya vista, `CInterfazConsola`, se añaden 2 nuevas clases, `CInterfazDisco` y `CDesensamblador`. La primera de ellas da soporte a la interfaz con el usuario y a las operaciones que realiza la herramienta. Las segunda permite guardar en disco y cargar desde disco el contenido de la memoria. La última sirve de desensamblador, para leer el código almacenado en la memoria. El resto, son las mismas que modelizan tanto el ensamblador como el simulador. Sólo se muestran las clases, sus atributos y las relaciones

que participan directamente en esta parte de la solución (Figura 3.3.4). Por otra parte, como se ha visto en apartados anteriores de diseño, las clases se relacionan entre sí físicamente a través de la clase `CConfiguración`, pero en este caso se están representando las relaciones entre clases a nivel conceptual.

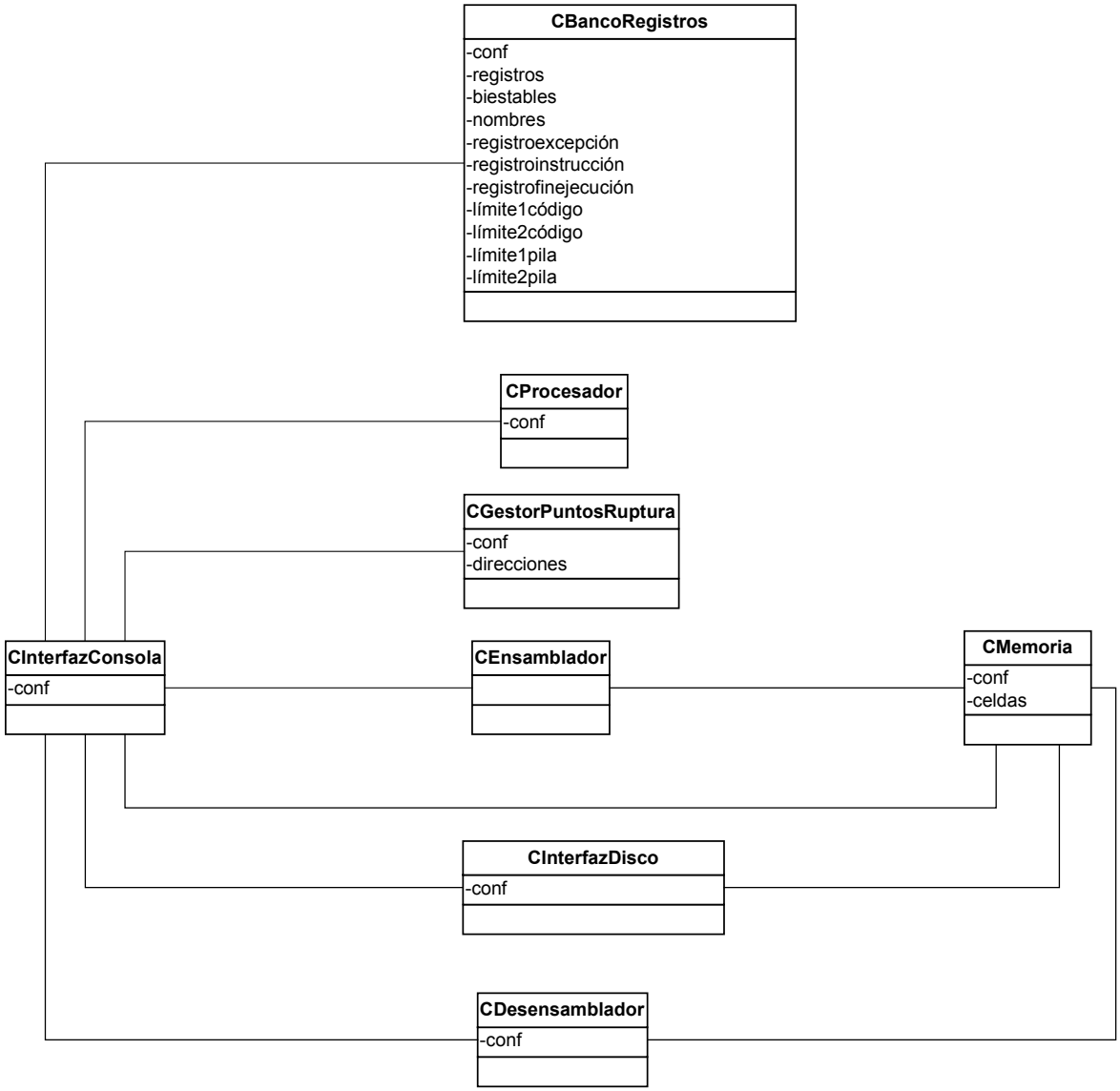


Figura 3.3.4. Diagrama de Clases de la Interfaz de Usuario en Línea de Comandos

Clase CInterfazConsola

Esta clase da soporte para todas las operaciones que puede introducir el usuario desde la línea de comandos, e interactúa con el resto de clases del modelo para conseguir efectuar dichas operaciones.

CInterfazConsola
- conf: ^`CConfiguración`
- Ayuda(): `Entero` - Base(IN valor: `Cadena`): `Entero` - CargarMemoria(IN fichero: `Cadena`): `Entero` + CInterfazConsola(IN configuración: ^`CConfiguración`) + ~CInterfazConsola()

- Desensamblar(IN dir: Entero, IN núm: Entero): Entero
- Ejecutar(): Entero
- Ensamblar(IN fichero: Cadena): Entero
+ EnsamblarLíneaComando(fichero: Cadena): Entero
- EscribirMemoria(IN dirección: Entero, IN valor: Entero): Entero
- EscribirRegistro(IN registro: Cadena, IN valor: Entero): Entero
+ EstadoInicialSimulador(): Entero
- EstadoSimulador(): Entero
- ExtraerComando(IN entrada: Cadena, OUT comando: Carácter, OUT parámetro1: Cadena, OUT parámetro2: Cadena, OUT nparámetros: Entero)
- GuardarMemoria(IN fichero: Cadena): Entero
- IntercambiarComprobarPC(): Entero
- IntercambiarComprobarPila(): Entero
- IntercambiarModoPasoAPaso(): Entero
- IntercambiarModoPila(): Entero
- IntercambiarPararPuntosRuptura(): Entero
- IntercambiarPuntoRuptura(): Entero
- IntercambiarReiniciarRegistros(): Entero
- LeerBancoRegistros()
+ LeerComandos(): Entero
- LeerMemoria(IN dirección: Cadena)
- LeerRegistro(IN registro: Cadena)
- Salir(): Entero
- VolcarBloqueMemoria(IN dir: Cadena, IN núm: Cadena): Entero
- VolcarPila(IN núm: Cadena): Entero

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase CConfiguración que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**ExtraerComando**
Parámetros	IN entrada: Cadena OUT comando: Carácter OUT parámetro1: Cadena OUT parámetro2: Cadena OUT nparámetros: Entero
Salida	Ninguna
Comportamiento	Analiza la entrada y la descompone en un comando y, si procede, los argumentos que le siguen. Rellena los campos correspondientes en los parámetros de salida.
Errores	Si se introdujo un comando no válido, devuelve comando = COMANDO_NO_VÁLIDO. Si no se introdujo ningún comando, devuelve comando = COMANDO_VACÍO.

Método	**Ayuda**
Parámetros	Ninguno
Salida	Entero
Comportamiento	Muestra por consola el contenido del archivo de ayuda (ayuda.txt). Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 como valor de retorno si no se encuentra o no se puede abrir el archivo de ayuda (ayuda.txt).

Método	**Base**
Parámetros	IN valor: `Cadena`
Salida	`Entero`
Comportamiento	Si `valor` == 10, cambia la base de representación del simulador a decimal. En cambio, si `valor` == 16, cambia la base de representación del simulador a hexadecimal. Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 si valor es distinto de 10 y 16 (es decir, se introdujo una base de representación no contemplada por la aplicación).

Método	**CargarMemoria**
Parámetros	IN fichero: `Cadena`
Salida	`Entero`
Comportamiento	Abre el fichero indicado y carga su contenido en la memoria del simulador. Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 si hubo algún problema al abrir o leer el fichero.

Método	**CInterfazConsola**
Parámetros	IN configuración: `^CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`.
Errores	Ninguno.

Método	**~CInterfazConsola**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Desensamblar**
Parámetros	IN dir: `Entero` IN núm: `Entero`
Salida	`Entero`
Comportamiento	A partir de la dirección indicada por el parámetro `dir`, desensambla un total de `núm` instrucciones y muestra el resultado en pantalla, según el formato descrito anteriormente. Si se llega al final de la memoria, se deja de desensamblar, pero no se produce un error. Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 como valor de retorno si alguno de los parámetros no es correcto. Tanto `dir` como `núm` deben ser enteros de 16 bits (entre 0 y 65535).

Método	**Ejecutar**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Si está activada la opción de reiniciar el banco de registros y el biestable `H` está activado, se reinician los registros. Si está activada la opción de ejecución paso a paso, se muestra por la consola la siguiente instrucción que se va a ejecutar. Se invoca al método `Ejecutar` del objeto procesador, y se recoge el valor de retorno cuando finalice la ejecución. Se muestra por consola un mensaje indicando la causa de la finalización de la ejecución, como se describió anteriormente. Devuelve el valor obtenido como retorno del método `Ejecutar` del objeto procesador.
Errores	Ninguno como tal.

Método	**Ensamblar**
Parámetros	IN fichero: `Cadena`
Salida	`Entero`
Comportamiento	Invoca al ensamblador para que procese el fichero indicado, y recoge el resultado de la operación. Si todo ha ido bien, carga el código máquina en la memoria del simulador, muestra por consola el número de líneas procesadas y un mensaje indicando que la operación ha ido correctamente y devuelve 0 como valor de retorno
Errores	Devuelve –1 si hubo algún error en el proceso de ensamblado, y muestra por consola información acerca del problema (ya sean errores en el código fuente, en cuyo caso se mostrará el número de líneas procesadas y la lista de errores, o bien errores en la propia operación de ensamblado).

Método	**EnsamblarLíneaComando**
Parámetros	IN fichero: `Cadena`
Salida	`Entero`
Comportamiento	Encapsula el uso del método `Ensamblar` para cuando desde la línea de comando se indica el fichero que se quiere ensamblar.
Errores	Devuelve el valor retornado por el método `Ensamblar`.

Método	**EscribirMemoria**
Parámetros	IN dirección: `Entero` IN valor: `Entero`
Salida	`Entero`
Comportamiento	Llama al método `Escribir (dirección, valor)` del objeto memoria, es decir, actualiza la dirección de memoria indicada por el primer parámetro (`dirección`) con el valor indicado por el segundo parámetro (`valor`). Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 si hubo algún error en alguno de los parámetros (deben ser enteros de 16 bits).

Método	**EscribirRegistro**
Parámetros	IN registro: `Cadena` IN valor: `Entero`
Salida	`Entero`
Comportamiento	Llama al método `Escribir (registro, valor)` del objeto banco de registros, es decir, actualiza la el contenido del registro indicado por el primer parámetro (`registro`) con el valor indicado por el segundo parámetro (`valor`). Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 si hubo algún error en alguno de los parámetros (el registro debe existir y el valor debe ser un entero de 16 bits).

Método	**EstadoInicialSimulador**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Proporciona acceso al método `EstadoSimulador`. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**EstadoSimulador**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Muestra por consola la configuración del simulador, según se vio anteriormente. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**GuardarMemoria**
Parámetros	IN fichero: `Cadena`
Salida	`Entero`
Comportamiento	Vuelca el contenido de la memoria del simulador en el `fichero` indicado. Devuelve 0 como valor de retorno si todo ha ido bien.
Errores	Devuelve –1 como valor de retorno si hubo algún problema al crear o abrir el fichero.

Método	**IntercambiarComprobarPC**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Modifica la configuración del simulador, en concreto la opción de comprobar si el contador de programa invade la zona de pila. Si está activa, la desactiva y viceversa. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**IntercambiarComprobarPila**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Modifica la configuración del simulador, en concreto la opción de comprobar si el puntero de pila invade la zona de código. Si está activa, la desactiva y viceversa. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**IntercambiarModoPasoAPaso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Modifica la configuración del simulador, en concreto la opción de ejecutar instrucción a instrucción. Si está activa, la desactiva y viceversa. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**IntercambiarModoPila**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Modifica la configuración del simulador, en concreto la opción de que la pila crezca hacia direcciones ascendentes o descendentes. Si está configurada como ascendente, la cambia a descendente y viceversa. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**IntercambiarPararPuntosRuptura**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Modifica la configuración del simulador, en concreto la opción de detener el procesador cuando se encuentre un punto de ruptura definido por el usuario. Si está activa, la desactiva y viceversa. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**IntercambiarReiniciarRegistros**
Parámetros	Ninguno
Salida	`Entero`

Comportamiento	Modifica la configuración del simulador, en concreto la opción de reiniciar los valores de los registros tras finalizar una ejecución y antes de comenzar la siguiente. Si está activa, la desactiva y viceversa. Devuelve 0 como valor de retorno.
Errores	Ninguno.

Método	**LeerBancoRegistros**
Parámetros	Ninguno
Salida	Ninguno
Comportamiento	Lee el contenido de todos los registros del banco de registros, los biestables de estado y los límites de las zonas de pila y código y las muestra en pantalla, en formato decimal o hexadecimal según sea la configuración.
Errores	Ninguno.

Método	**LeerComandos**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Se muestra el indicador `ENS2001>` en la salida estándar. Se leen comandos introducidos por el usuario en la entrada estándar, mientras éste no introduzca el comando para acabar la sesión (`q`). Devuelve 0 como valor de retorno.
Errores	Si algún comando de los introducidos no se reconoce, se mostrará un mensaje de error por la salida estándar.

Método	**LeerMemoria**
Parámetros	IN dirección: `Cadena`
Salida	`Entero`
Comportamiento	Lee el contenido de la posición de memoria indicada por el parámetro de entrada y lo muestra por pantalla, en formato decimal o hexadecimal según sea la configuración. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 si el parámetro dirección está fuera de rango.

Método	**LeerRegistro**
Parámetros	IN registro: `Cadena`
Salida	`Entero`
Comportamiento	Lee el contenido del registro indicado por el parámetro de entrada y lo muestra por pantalla, en formato decimal o hexadecimal según sea la configuración. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 si el registro no existe.

Método	**Salir**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Pide confirmación al usuario sobre si desea salir. Si el usuario introduce un carácter '`S`' o '`s`', guarda la configuración actual en el archivo correspondiente y devuelve 1 como valor de retorno. En otro caso, devuelve 0 como valor de retorno.
Errores	En caso de que no se pueda guardar el archivo de configuración se informa por pantalla, pero se sigue devolviendo 1 como valor de retorno.

Método	**VolcarBloqueMemoria**
Parámetros	IN dir: `Cadena` IN núm: `Cadena`
Salida	`Entero`

Comportamiento	Muestra en pantalla el contenido del bloque de memoria de `núm` posiciones a partir de la dirección `dir`, en formato decimal o hexadecimal según sea la configuración. Si el bloque indicado sobrepasa el límite superior de memoria, se informa mediante un mensaje por pantalla. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 si alguno de los parámetros es incorrecto (fuera de rango).

Método	**VolcarPila**
Parámetros	IN núm: `Cadena`
Salida	`Entero`
Comportamiento	Muestra en pantalla el contenido de las últimas `núm` posiciones de la pila, siempre que esto sea posible sin exceder los límites de la memoria. La cima se mostrará en última o en primera posición según la configuración de crecimiento de la pila sea ascendente o descendente respectivamente. Los valores se mostrarán en formato decimal o hexadecimal según la configuración. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 si el parámetro de entrada está fuera de rango (entero de 16 bits).

Clase CInterfazDisco

Esta clase sirve para encapsular las operaciones de cargar y guardar en disco el contenido de la memoria del simulador.

CInterfazDisco
- conf: `^CConfiguración`
+ CInterfazDisco(IN configuración: `^CConfiguración`) + ~CInterfazDisco + VolcarDiscoAMemoria(IN fichero: `Cadena`) + VolcarMemoriaADisco(IN fichero: `Cadena`)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CInterfazDisco**
Parámetros	IN configuración: `^CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`.
Errores	Ninguno.

Método	**~CInterfazDisco**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**VolcarDiscoAMemoria**
Parámetros	IN fichero: `Cadena`
Salida	`Entero`

Comportamiento	Abre el fichero indicado por el parámetro de entrada y carga su contenido de forma secuencial y en formato *big-endian* en la memoria del simulador.
Errores	Devuelve –1 si se produjo algún error al abrir el fichero o durante la lectura.

Método	**VolcarMemoriaADisco**
Parámetros	IN fichero: `Cadena`
Salida	`Entero`
Comportamiento	Guarda el contenido de la memoria del simulador en el fichero indicado por el parámetro de entrada, de forma secuencial y en formato *big-endian*.
Errores	Devuelve –1 si se produjo algún error al abrir el fichero para escritura o al guardar los datos.

Clase CDesensamblador

Esta clase ofrece la funcionalidad de desensamblar código máquina. Interactúa directamente con la memoria del simulador. Hay que indicar la dirección a partir de la cual se quiere desensamblar.

CDesensamblador
- conf: ^`CConfiguración`
+ CDesensamblador(IN configuración: ^`CConfiguración`) + ~CDesensamblador() + Desensamblar(INOUT dir: `Entero`, OUT instrucción: `Cadena`) - FetchOperandos(INOUT dir: `Entero`, OUT md1: `Entero`, OUT op1: `Entero`, OUT md2: `Entero`, OUT op2: `Entero`)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**CDesensamblador**
Parámetros	IN configuración: ^`CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`.
Errores	Ninguno.

Método	**~CDesensamblador**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**Desensamblar**
Parámetros	INOUT dir: `Entero` OUT instrucción: `Cadena`
Salida	Ninguna

Comportamiento	Desensambla la instrucción almacenada en memoria a partir de la dirección indicada por el parámetro `dir` y almacena la cadena resultante en el parámetro de salida `instrucción`. En el parámetro de entrada/salida `dir` devuelve la posición de la siguiente instrucción.
Errores	Ninguno.

Método	**FetchOperandos**
Parámetros	INOUT dir: `Entero` OUT md1: `Entero` OUT op1: `Entero` OUT md2: `Entero` OUT op2: `Entero`
Salida	Ninguna
Comportamiento	Desensambla una instrucción a partir de la dirección indicada por el parámetro de entrada dir y la almacena en los parámetros de salida. En `dir` devuelve la posición de la siguiente instrucción.
Errores	Ninguno.

3.3.4. Diseño de la Interfaz en Modo Gráfico

3.3.4.1. Descripción de la interfaz

La finalidad de construir una interfaz en modo gráfico para la herramienta consiste en disponer de una mayor accesibilidad a los elementos que componen el simulador, manteniendo la misma funcionalidad que en el caso de la interfaz textual.

El principal problema que presenta la interfaz por consola es que la información aparece entremezclada, y deja de ser visible conforme transcurre la simulación de los programas o bien el usuario va solicitando nueva información.

Por tanto, como solución se propone el diseño de una serie de ventanas que muestren los distintos tipos de información que el usuario pueda solicitar. Serán las siguientes:

- **Ventana Principal** (clase `TfrmPrincipal`).

  Contendrá un menú desde el que se puede acceder a todas las operaciones disponibles. También contendrá una barra de herramientas con las acciones más comunes y una ventana en la que se mostrarán al usuario mensajes de aviso o error durante el transcurso de la sesión.

- **Consola** (clase `TfrmConsolaEjecución`).

  Se trata de la consola del simulador, donde el procesador ejecutará las instrucciones de entrada/salida.

- **Memoria** (clase `TfrmMemoria`).

  Muestra el contenido de una zona de memoria. Permite al usuario indicar qué zona quiere que se le muestre, alterar el contenido de las posiciones de memoria y reiniciar el valor contenido en todas ellas. Se apoya en una ventana auxiliar (clase TfrmValorMemoria) desde la cual el usuario puede editar las celdas de memoria.

- **Pila** (clase `TfrmPila`).

  En principio, muestra el contenido de la cima de la pila y las últimas posiciones, aunque el usuario puede desplazarse a su antojo por el resto de la memoria. También permite alterar el contenido de las celdas de memoria, al igual que la ventana Memoria.

- **Banco de Registros** (clase `TfrmBancoRegistros`).

  Muestra el contenido del banco de registros, permite alterar su valor y reiniciarlo. También muestra los biestables de estado, los límites de las zonas de código y pila, y desensambla la instrucción a la que está apuntando el contador de programa. Se apoya en una ventana auxiliar (TfrmValorRegistro) desde la cual el usuario puede editar el valor de los registros.

- **Código Fuente** (clase `TfrmFuente`).

  Desensambla el contenido de la memoria a partir de una dirección determinada, bien la apuntada por el contador de programa o bien una arbitrariamente seleccionada por el usuario.

- **Configuración** (clase `TfrmConfiguración`).

  Muestra todas las opciones de configuración del simulador y permite al usuario su modificación.

La ventana principal, como su nombre indica, lleva el peso de la ejecución. Cuando se cierre esta ventana, se mostrará un mensaje de confirmación y, si el usuario acepta, se dará por finalizada la aplicación. El resto de las ventanas pueden aparecer visibles u ocultas a voluntad del usuario, excepto la ventana de consola, que se hará visible automáticamente al ejecutarse una instrucción de entrada/salida.

Todas las ventanas pueden cerrarse pulsando la tecla ESCAPE o bien el icono de Cerrar Ventana, situado en la parte superior derecha de todas ellas.

La funcionalidad que ofrece la aplicación es la misma que para la versión en modo texto, con un par de añadidos. Existe una nueva opción, denominada ‘Guardar Sesión’, que permite al usuario almacenar en un fichero de texto el contenido del cuadro de mensajes y la consola. Por otra parte, la herramienta almacena la posición y estado de todas las ventanas de la interfaz en el archivo `wens2001.cfg`, para restaurarlo en la siguiente sesión.

El usuario interactuará con la aplicación según la forma habitual de trabajo con el entorno *Windows*. Cuando tenga que introducir valores enteros (en la interfaz, no durante la simulación de programas), podrá hacerlo en formato decimal o hexadecimal (añadiendo el prefijo ‘0x’) indistintamente, en el rango 0 a 65535 (o –32768 a 32767). Si no es así, la herramienta debe verificarlo e informar del error.

Clase TfrmPrincipal

Esta clase recoge la Ventana Principal y, con ella, toda la funcionalidad básica de la aplicación. El usuario puede acceder a la mayoría de las opciones que ofrece *ENS2001* desde el menú de esta ventana.

Se trata de una ventana normal de *Windows*, con el menú de control y botones para Maximizar, Minimizar y Cerrar. Su tamaño puede ser modificado a voluntad del usuario, pero con unas medidas mínimas de 180 puntos de alto y 230 puntos de ancho. El tamaño del cuadro de mensajes se modificará en consonancia para mantener la distancia a los bordes de la ventana. El menú y la barra de botones conservarán siempre su posición en la parte superior. La barra de estado siempre permanecerá en el borde inferior (Figura 3.3.5).

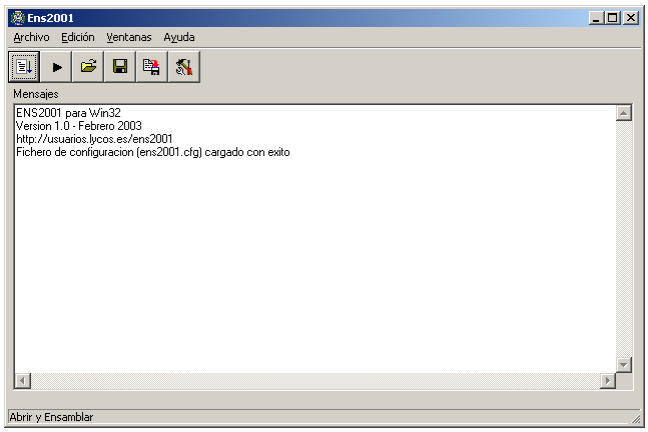
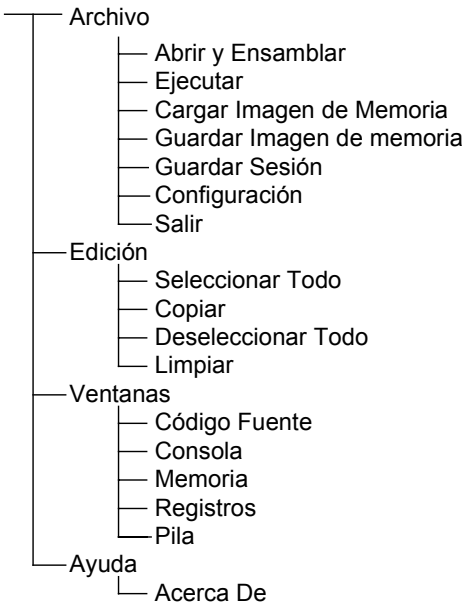


Figura 3.3.5. Ventana Principal *ENS2001*-Win32

La ventana principal consta de los siguientes elementos, que se detallan a continuación:

- Menú.
- Barra de botones.
- Cuadro de texto.
- Barra de estado.

El menú tiene la siguiente estructura:



- El submenú Archivo agrupa las opciones relativas al manejo principal de la herramienta.

- El submenú Edición, que también es activable haciendo clic con el botón derecho del ratón sobre el cuadro de mensajes, ofrece cuatro sencillas opciones de edición para dicho cuadro.
- El submenú Ventanas permite que el usuario pueda mostrar u ocultar las cinco ventanas flotantes de las que dispone la herramienta. En el menú se muestra un *tick* a la izquierda de aquellas ventanas que estén visibles en el momento de entrar en el menú.
- El submenú Ayuda contiene la opción Acerca De.

A continuación se va a explicar brevemente el funcionamiento de todas las opciones del menú.

Archivo>Abrir y Ensamblar (atajo Ctrl+E).

Cumple los requisitos **ERS-REQ05**, **ERS-REQ06** y **ERS-REQ07**.

Con este comando, la herramienta abre un cuadro de diálogo para que el usuario seleccione un fichero.

Una vez seleccionado, llama al ensamblador para que realice su trabajo y recoge los resultados para mostrarlos en el cuadro de mensajes.

El ensamblador necesita crear unos ficheros temporales de trabajo, que son *memoria.tmp*, *fuente.tmp* y *errores.tmp*. Si no pudiera crear alguno de ellos, se mostrará el siguiente error en el cuadro de mensajes y no se completará la operación:

```
Error generando fichero nombre_fichero
El directorio de trabajo está lleno o protegido contra escritura
```

Esto podría ocurrir, por ejemplo, si se intenta ejecutar la aplicación desde un *CD-ROM*, o una unidad de red con acceso exclusivo de lectura.

Archivo>Ejecutar (atajo Ctrl+J).

Cumple el requisito **ERS-REQ24**.

Invoca al procesador virtual para lanzar la simulación. El procesador ejecutará instrucciones hasta que se cumpla una de las siguientes condiciones:

- Ejecución Paso a Paso activa (se detiene nada más ejecutar una instrucción).
- Punto de Ruptura Activado (se detiene antes de ejecutar la instrucción).
- Excepción durante la ejecución.
- Instrucción `HALT`.

Al finalizar la simulación, la herramienta mostrará en el cuadro de mensajes cuál de estas causas fue la que detuvo al procesador, respectivamente:

```
Modo Paso A Paso. Se detuvo la ejecución
Punto de Ruptura Incondicional. Se detuvo la ejecución
Se ha generado una excepción
Fin del programa. Instrucción HALT
```

En el caso de que se haya producido una excepción, se acompañará el mensaje con otro que indica qué excepción se ha producido, alguna de las vistas en el apartado de diseño del simulador:

```
Instrucción No Implementada
División Por Cero
Contador de Programa Sobrepasa el Final de la Memoria
Contador de Programa Invade la zona de Pila
Puntero de Pila Invade la zona de Código
Ejecución Detenida por el Usuario
```

Si durante la ejecución el procesador encuentra una instrucción de entrada o salida por consola, se mostrará la ventana de la consola y esperará a la introducción del dato por parte del usuario para continuar. En el caso de que el procesador se encuentre esperando por un dato de entrada, el usuario podrá detener la ejecución pulsando la tecla ESCAPE. En cualquier otro momento, el usuario podrá detener la simulación pulsando el botón Detener.

Archivo>Cargar Imagen de Memoria (atajo Ctrl+L).

Cumple el requisito **ERS-REQ09**.

Este comando permite recuperar el contenido íntegro de la memoria desde un fichero en el que se habrá guardado previamente. La herramienta mostrará un cuadro de diálogo donde el usuario seleccionará el fichero que desea cargar. Una vez seleccionado, se invocará a la interfaz de disco para que lea el fichero y lo cargue en la memoria.

Como la lectura se hace en formato binario, si el usuario intenta leer un fichero cualquiera que no haya sido generado por la aplicación los resultados son inesperados. En concreto, se espera leer un archivo de 128Kbytes de longitud, con las palabras de memoria almacenadas en formato *big-endian*.

Archivo>Guardar Imagen de Memoria (atajo Ctrl+S).

Cumple el requisito **ERS-REQ08**.

Este comando permite guardar el contenido íntegro de la memoria en un fichero. La herramienta mostrará un cuadro de diálogo para que el usuario escoja el nombre del fichero. Una vez seleccionado, se invocará a la interfaz de disco para que genere el fichero.

Si no existe, se creará un nuevo fichero, que tendrá una longitud de 128Kbytes. En caso de que existiera, la aplicación preguntará si se desea sobrescribir y, en caso negativo, dará la opción de introducir otro nombre.

El fichero es una imagen exacta de la memoria, almacenado en formato *big-endian*.

Archivo>Guardar Sesión (atajo Ctrl+G).

Cumple el requisito **ERS-REQ34**.

Este comando, que es el único que sólo está disponible para la versión gráfica de la aplicación, permite guardar en un fichero el contenido del cuadro de mensajes y la consola. Para ello, como es habitual, presentará un cuadro de diálogo donde el usuario puede escoger en qué fichero desea guardar la sesión.

Igual que en la operación anterior, si el fichero ya existe la aplicación dará a elegir entre sobrescribirlo o introducir un nombre nuevo. A continuación, recogerá el contenido del cuadro de mensajes y la consola y los almacenará en disco de la siguiente forma:

```
---- Mensajes ----
```
Contenido del cuadro de mensajes
```
---- Consola ----
```
Contenido de la consola

Archivo>Configuración (atajo Ctrl+F).

Cumple el requisito **ERS-REQ31**.

Este comando se estudiará en profundidad al hablar de la ventana de configuración. Muestra dicha ventana, en la que se presentan las diferentes opciones de configuración de la herramienta.

Archivo>Salir (atajo Ctrl+X).

Cumple el requisito **ERS-REQ35**.

Este comando sirve para salir de la aplicación. Tiene el mismo efecto que pulsar el icono de cerrar (X) en la esquina superior derecha, o pulsar la tecla ESCAPE cuando el foco lo tiene la ventana principal. Al salir, tras pedir confirmación al usuario (Figura 3.3.6), se guardará la configuración actual y la posición y estado de las ventanas flotantes.

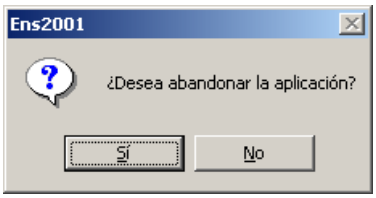


Figura 3.3.6. Confirmar comando Archivo>Salir

Edición>Seleccionar Todo (atajo Ctrl+A).

Selecciona todo el texto contenido en el cuadro de mensajes.

Edición>Copiar (atajo Ctrl+C).

Copia en el portapapeles el texto del cuadro de mensajes que se encuentre seleccionado en ese momento.

Edición>Deseleccionar Todo (atajo Ctrl+N).

Descarta la selección actual del cuadro de mensajes.

Edición>Limpiar (atajo Ctrl+W).

Borra el contenido del cuadro de mensajes.

Ventanas>Código Fuente (atajo Ctrl+F1).

Si no está marcada, muestra la ventana de código fuente, y si está marcada con un *tick* (‘✓’), la oculta.

Ventanas>Consola (atajo Ctrl+F2).

Si no está marcada, muestra la ventana de la consola, y si está marcada con un *tick* (‘✓’), la oculta.

Ventanas>Memoria (atajo Ctrl+F3).

Si no está marcada, muestra la ventana de memoria, y si está marcada con un *tick* (‘✓’), la oculta.

Ventanas>Registros (atajo Ctrl+F4).

Si no está marcada, muestra la ventana de banco de registros, y si está marcada con un *tick* (‘✓’), la oculta.

Ventanas>Pila (atajo Ctrl+F5).

Si no está marcada, muestra la ventana de pila, y si está marcada con un *tick* (‘✓’), la oculta.

Ayuda>Acerca De.

Muestra un cuadro de diálogo que informa sobre la versión de *ENS2001* que se está ejecutando (Figura 3.3.7).

Figura 3.3.7. Cuadro de Diálogo Acerca de

La barra de botones contiene accesos directos a las operaciones más comunes. El comportamiento es el mismo que si el usuario selecciona en el menú principal las opciones correspondientes, como se indica a continuación:

Archivo>Abrir y Ensamblar

Archivo>Ejecutar

Archivo>Cargar Imagen de Memoria

Archivo>Guardar Imagen de Memoria

Archivo>Guardar Sesión

Archivo>Configuración

El botón Ejecutar cambiará de icono y de función, dependiendo de las siguientes condiciones:

- Cuando haya un programa ejecutándose, su función será **Detener** la ejecución, y se mostrará de esta forma:

- Cuando en la configuración esté seleccionado el Modo de Ejecución **Paso a Paso**, tendrá esta forma:

El cuadro de mensajes es un cuadro de texto, como se aprecia en la Figura 3.3.8, en el que se irán mostrando los distintos mensajes generados por la herramienta, como consecuencia del proceso de ensamblado o ejecución. Ejemplos de estos mensajes pueden ser el número de líneas procesadas por el ensamblador, los errores producidos (caso de que existan), o la condición de detención del simulador tras la ejecución.

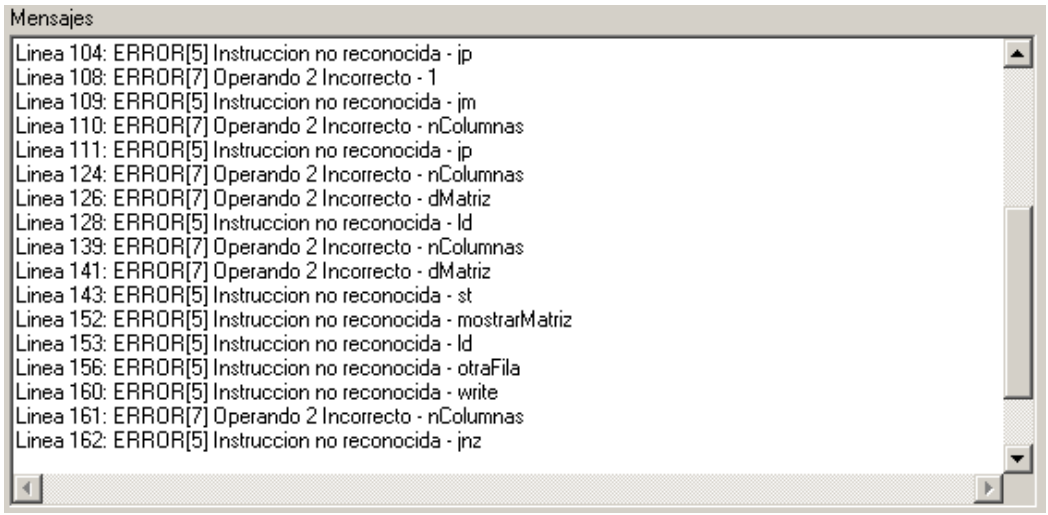


Figura 3.3.8. Cuadro de Mensajes

Haciendo clic con el botón derecho del ratón encima de él se accede al menú Edición (Figura 3.3.9).

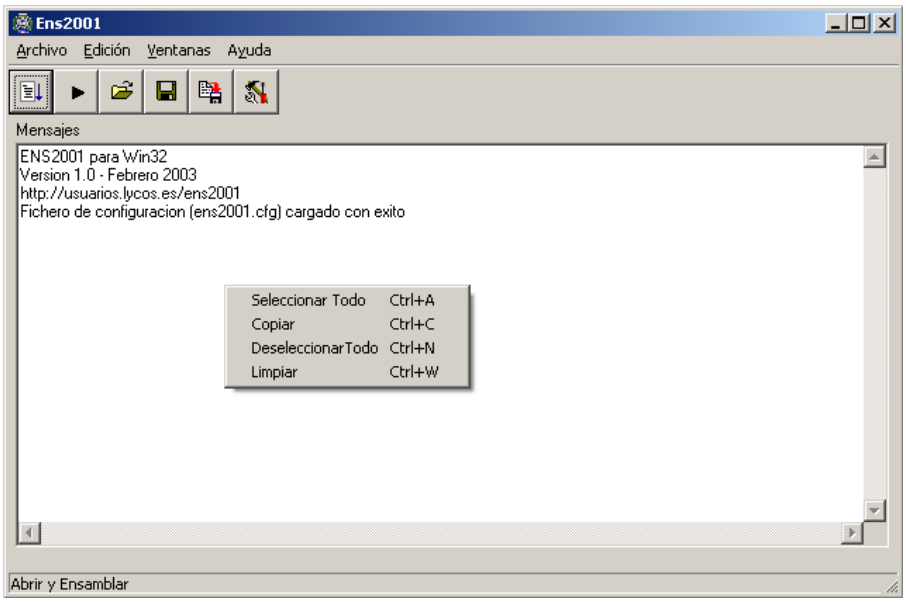


Figura 3.3.9. Menú Edición flotante

La barra de desplazamiento vertical se activará cuando el contenido del cuadro no quepa dentro de los límites definidos para el mismo.

Clase TfrmFuente.

Cumple los requisitos **ERS-REQ20** y **ERS-REQ33**.

A través de esta ventana (Figura 3.3.10), el usuario tiene acceso al contenido de la memoria, pero una vez desensamblado, con lo que se puede leer fácilmente el código contenido en ella. El código se desensambla a partir de una posición de memoria dada, con la ayuda de la clase `CDesensamblador`. La funcionalidad es análoga a la del comando 'd' de la interfaz en modo texto.

Se trata de una ventana de tipo *ToolBox* con una anchura fija de 275 puntos y una altura mínima de 290 puntos.

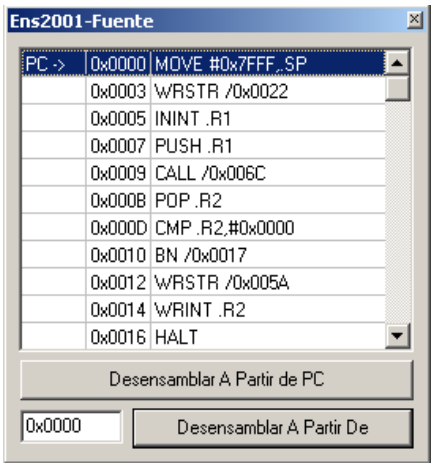


Figura 3.3.10. Ventana de Código Fuente

La ventana consta de un cuadro en el que se muestran más instrucciones desensambladas cuanto mayor es la altura de la ventana. La primera columna contiene información acerca

de los puntos de ruptura habilitados y la posición a la que apunta el contador de programa, la segunda columna indica la dirección de memoria y la tercera, la instrucción desensamblada.

Bajo dicho cuadro hay un botón, **Desensamblar A Partir de PC**, que sirve, como su propio nombre indica, para desensamblar código a partir de la dirección a la que apunta el contador de programa.

Por último, la ventana contiene un cuadro de texto en el que el usuario puede introducir una dirección de memoria a partir de la que se desensamblará código pulsando en el botón **Desensamblar A Partir De**.

Teniendo en cuenta que no todas las instrucciones ocupan el mismo tamaño en memoria (las hay de 1, 2 y 3 palabras de longitud), se pueden obtener resultados erróneos al desensamblar a partir de una dirección arbitraria. También puede leerse código extraño al desensamblar las zonas de datos de los programas, por ejemplo. No se ejerce ningún control sobre este tipo de situaciones. El usuario debería desensamblar siempre a partir del comienzo del código, o bien a partir de la posición ocupada por el contador de programa o una instrucción desensamblada anteriormente.

En esta ventana también se pueden activar y desactivar los puntos de ruptura incondicionales, que servirán de ayuda al usuario a la hora de depurar sus programas. Para ello, basta con hacer doble clic en la posición de memoria que se quiera editar, y se irá alternando entre punto de ruptura activado y desactivado. Los puntos de ruptura activados se marcan con un asterisco "*" en la columna de la izquierda, como se puede observar en la Figura 3.3.11.

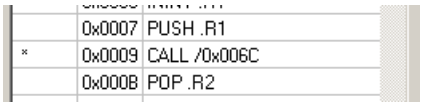


Figura 3.3.11. Marca de Punto de Ruptura Activado

Como se ha citado anteriormente, se puede conocer la posición a la que apunta el Contador de Programa mediante el indicador "PC →" que aparecerá en la columna de la izquierda (Figura 3.3.12).

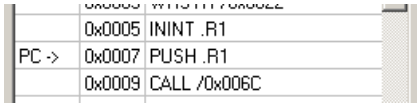


Figura 3.3.12. Indicador de Contador de Programa

Clase TfrmConsolaEjecución.

A través de esta ventana se realizan las operaciones de entrada/salida durante la simulación.

Se trata de una ventana normal de *Windows*, con el menú de control y botones para Maximizar, Minimizar y Cerrar. Su tamaño puede ser modificado a voluntad del usuario, pero con unas medidas mínimas de 160 puntos de alto y 200 puntos de ancho. El tamaño del cuadro de mensajes se modificará en consonancia para mantener la misma distancia a los bordes de la ventana (Figura 3.3.13). El espacio en el borde inferior es necesario para

los casos en los que se pide un dato de entrada, ya que el comportamiento de la consola es el siguiente:

- Como salida, visualizará el resultado de las instrucciones `WRCHAR`, `WRINT` y `WRSTR`.
- Como entrada, en ella el usuario deberá introducir los datos de entrada para las instrucciones correspondientes (`INCHAR`, `ININT` e `INSTR`) cuando se ejecute alguna de ellas. Además, en el borde inferior izquierdo se mostrará el tipo de dato que se espera que el usuario introduzca (Carácter, Entero o Cadena, respectivamente).

En el momento que el procesador ejecute cualquiera de las instrucciones de entrada/salida, la ventana pasará a primer plano. Además, en el caso de que se trate de una instrucción de entrada, el simulador quedará a la espera de que el usuario introduzca información, o bien detenga la ejecución pulsando la tecla ESCAPE. Si esto ocurre, se generará la excepción *Ejecución Detenida por el Usuario*.

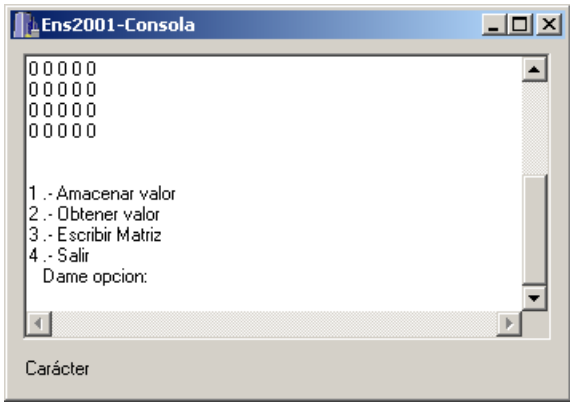


Figura 3.3.13. Consola

Si el usuario pretende cerrar la ventana durante la ejecución de una instrucción de entrada, se mostrará un mensaje que indique que debe introducir un valor, o bien abortar la simulación pulsando ESCAPE.

Análogamente al cuadro de mensajes de la ventana principal, la consola consta de un menú de edición (Figura 3.3.14), accesible haciendo clic con el botón derecho del ratón encima de la misma. Ofrece las siguientes opciones:

Seleccionar Todo (atajo Ctrl+A).

Selecciona todo el texto contenido en la consola.

Copiar (atajo Ctrl+C).

Copia en el portapapeles el texto de la consola que se encuentre seleccionado en ese momento.

Deseleccionar Todo (atajo Ctrl+N).

Descarta la selección actual de la consola.

Limpiar (atajo Ctrl+W).

Borra el contenido de la consola.

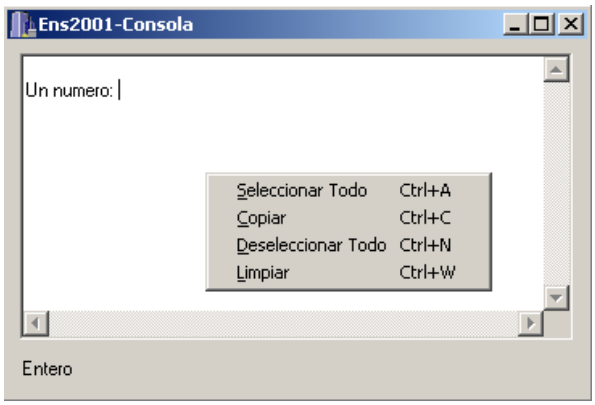


Figura 3.3.14. Menú Edición de la Consola

Clase TfrmMemoria.

Cumple los requisitos **ERS-REQ11** y **ERS-REQ12**.

La funcionalidad que ofrece esta ventana es análoga a la ofrecida por los comandos 'm' y 'v' de la interfaz en modo texto.

Se trata de una ventana de tipo *ToolBox* con una anchura fija de 195 puntos y una altura mínima de 300 puntos, como se observa en la Figura 3.3.15.

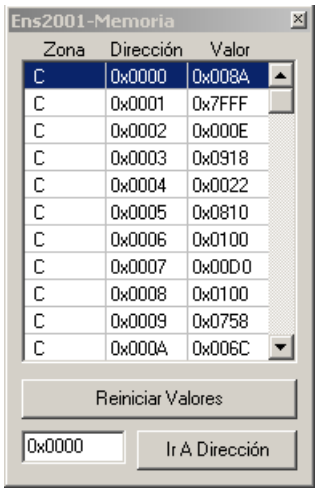


Figura 3.3.15. Ventana de Memoria de ENS2001

La ventana consta de una cuadro desplazable verticalmente en el que el usuario puede consultar el valor de tantas posiciones de memoria correlativas como quepan en la ventana. Dicha información se presenta en tres columnas:

- Zona: zona a la que pertenece la posición de memoria ('C' código, 'P' pila o en blanco si no pertenece a ninguna de las dos), según se observa en la Figura 3.3.16.
- Dirección: dirección de memoria.
- Valor: contenido de la posición de memoria.

Los datos de la dirección y el valor se mostrarán en formato decimal o hexadecimal según la configuración del simulador.

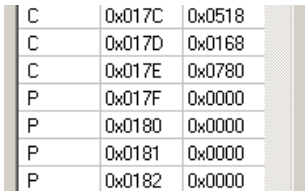


Figura 3.3.16. Indicadores de Zona Código y Pila

Si el usuario hace doble clic en la fila correspondiente a una dirección cualquiera, se abrirá una ventana de tipo `TfrmValorMemoria`, que le permitirá editar el valor allí contenido.

Debajo se encuentra un botón etiquetado como **Reiniciar Valores**. Si el usuario lo pulsa, se mostrará un cuadro de diálogo para solicitar que confirme la operación y, si la respuesta es afirmativa, reinicia el contenido de todas las posiciones de memoria (Figura 3.3.17).

Figura 3.3.17. Confirmar Reiniciar Valores

Por último, un cuadro de texto y un botón etiquetado como **Ir A Dirección**, que desplaza el contenido del cuadro superior, mostrando en primer lugar la dirección indicada por el usuario en el cuadro de texto.

Clase TfrmValorMemoria.

Se trata de un pequeño cuadro de diálogo, como el que se puede ver en la Figura 3.3.18, que dispone de tres elementos:

- Una etiqueta en la que se indica la posición de memoria que se va a editar
- Un cuadro de texto con el contenido de la posición de memoria.
- Un botón para Aceptar la edición.

La edición se puede cancelar pulsando la tecla ESCAPE o cerrando la ventana.

En caso de que el usuario acepte la operación, se validará el dato introducido. Si es correcto, se editará en la memoria del simulador y se cerrará la ventana. Si es erróneo, se avisará al usuario y se le permitirá introducir un nuevo valor.

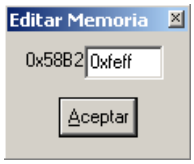


Figura 3.3.18. Editar Posición de Memoria

Clase TfrmRegistros.

Cumple los requisitos **ERS-REQ16**, **ERS-REQ17** y **ERS-REQ18**.

La ventana consta de cinco zonas de información: Banco de Registros, Biestables, Siguiente instrucción, Pila y Código. Se trata de la funcionalidad que ofrece el comando 'r' en la interfaz en modo texto.

Se trata de una ventana de tipo *ToolBox* cuyas dimensiones son fijas, siendo de 430 puntos de alto y 170 puntos de ancho, como se puede apreciar en la Figura 3.3.19.

El usuario puede modificar el valor de los registros haciendo doble clic encima de cualquiera de ellos (en la etiqueta o en el cuadro de texto, indistintamente). El sistema mostrará una ventana del tipo `TfrmValorRegistro`, donde tendrá la posibilidad de alterarlo.

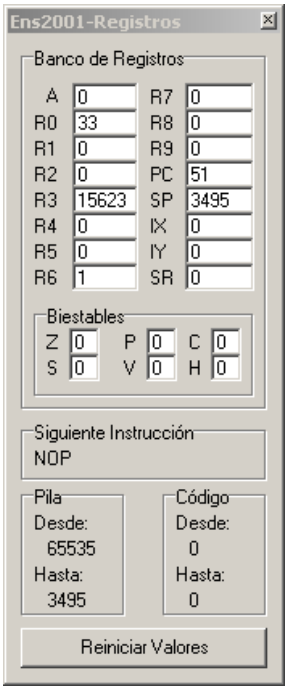


Figura 3.3.19. Ventana de Banco de Registros

En los recuadros Pila y Código se muestran los límites de ambas zonas de la memoria, respectivamente. El recuadro de Pila sólo se actualiza al detenerse la ejecución. Los valores del recuadro de Código se calculan tras ensamblar un programa.

Por último, aparece el botón etiquetado como **Reiniciar Valores**. Al pulsarlo, el sistema mostrará un cuadro de diálogo para confirmar la operación. Si el usuario acepta, se reiniciará el valor de todos los registros, de la siguiente forma:

- Registros generales, índices, contador de programa y registro de estado: a cero
- Puntero de pila: se calcula la ubicación del mayor hueco de espacio libre dejado por el código tras el ensamblado. Si la pila crece en direcciones ascendentes, el puntero de pila tomará el valor de la primera dirección de dicho hueco, mientras que si crece en direcciones descendentes, tomará el valor de la última dirección.

Clase TfrmValorRegistro.

Se trata de un pequeño cuadro de diálogo, como el que se aprecia en la Figura 3.3.20, que dispone de tres elementos:

- Una etiqueta en la que se indica el registro que se va a editar
- Un cuadro de texto con el contenido del registro.
- Un botón para Aceptar la edición.

La edición se puede cancelar pulsando la tecla ESCAPE o cerrando la ventana.

En caso de que el usuario acepte la operación, se validará el dato introducido. Si es correcto, se editará en el banco de registros del simulador y se cerrará la ventana. Si es erróneo, se avisará al usuario y se le permitirá introducir un nuevo valor.

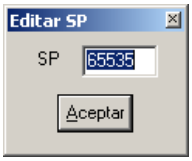


Figura 3.3.20. Editar Registro

Clase TfrmPila.

Cumple el requisito **ERS-REQ15**.

Es una ventana muy parecida en funcionalidad a la ventana de memoria. En este caso, la analogía se establece con el comando 'p' de la interfaz textual.

Se trata de una ventana de tipo 'ToolBox' cuya anchura es fija y de 170 puntos, y con una altura mínima de 300 puntos (Figura 3.3.21).

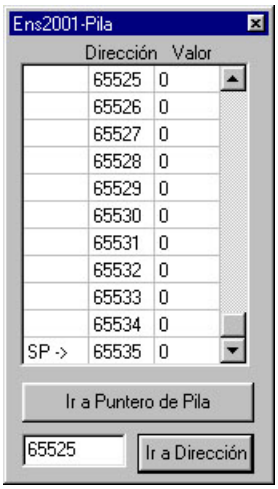


Figura 3.3.21. Ventana de Pila

La parte superior consta de un cuadro de desplazamiento vertical en el que se muestran tantas posiciones de memoria consecutivas como quepan en la ventana. En este caso, la columna de la izquierda únicamente sirve para indicar la posición de la cima de la pila ('SP→'). Las dos restantes son semejantes a las de la ventana de memoria. En la segunda se muestra la dirección y en la tercera el valor (ambas en formato decimal o hexadecimal según la configuración).

Si el usuario hace doble clic en la fila correspondiente a una dirección cualquiera, se abrirá una ventana de tipo `TfrmValorMemoria`, que le permitirá editar el valor allí contenido.

Debajo se encuentra un botón etiquetado como **Ir a Puntero de Pila**. Si el usuario lo pulsa, en el cuadro superior se mostrarán las últimas posiciones de la pila, lo que quiere decir que si la pila crece ascendentemente la cima aparecerá en el extremo inferior, y si crece descendentemente, aparecerá en el extremo superior.

Por último, un cuadro de texto y un botón etiquetado como **Ir a Dirección**, que desplaza el contenido del cuadro superior, mostrando en primer lugar la dirección indicada por el usuario en el cuadro de texto.

Clase TfrmConfiguración.

Esta ventana se muestra cuando el usuario selecciona en el menú la opción Archivo>Configuración o pulsa el botón correspondiente en la barra de botones.

Se trata de una ventana de tipo *ToolBox* cuyas dimensiones no se pueden alterar, siendo de 440 puntos de alto y 280 puntos de ancho (Figura 3.3.22).

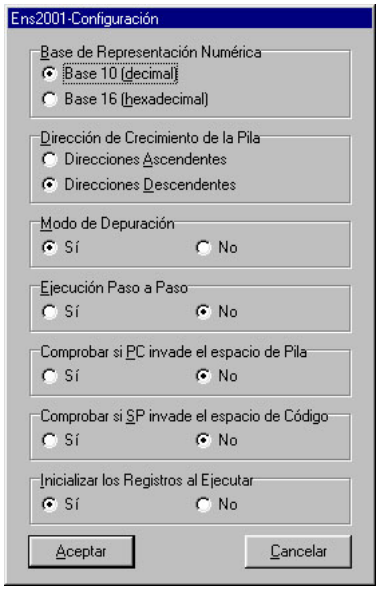


Figura 3.3.22. Ventana de Configuración

Consta de siete grupos de opciones, cada una de las cuales puede tomar dos valores posibles. Son las siguientes:

- Base de Representación Numérica.
  - Base 10 (decimal).
  - Base 16 (hexadecimal).
- Dirección de Crecimiento de la Pila.
  - Direcciones Crecientes.
  - Direcciones Decrecientes.
- Modo de Depuración.
  - Sí.
  - No.
- Ejecución Paso a Paso.
  - Sí.
  - No.

- Comprobar si PC invade el espacio de Pila.
  - Sí.
  - No.
- Comprobar si SP invade el espacio de Código.
  - Sí.
  - No.
- Inicializar los Registros al Ejecutar.
  - Sí.
  - No.

En la parte inferior se muestran dos botones, **Aceptar** y **Cancelar**. Si el usuario pulsa Aceptar, o la tecla INTRO, se guarda la configuración actual en el fichero `ens2001.cfg`. Si pulsa Cancelar o la tecla ESCAPE, los cambios se ignoran.

3.3.4.2. Modelo de Clases

A continuación se muestra el modelo de clases que da soporte a toda esta funcionalidad. Se introducen las 9 clases que se acaban de proponer, `TfrmPrincipal`, `TfrmConsolaEjecución`, `TfrmFuente`, `TfrmMemoria`, `TfrmPila`, `TfrmRegistros`, `TfrmConfiguración`, `TfrmValorMemoria` y `TfrmValorRegistro`. Por supuesto, también se emplean las clases `CInterfazDisco` y `CDesensamblador` (añadidas durante el diseño de la interfaz por consola) y las que modelizan tanto el ensamblador como el simulador. Sólo se muestran las clases y las relaciones que participan directamente en esta parte de la solución (Figura 3.3.23). En esta ocasión, sólo se representan las clases, sin mencionar sus atributos ni sus métodos, que se detallan posteriormente.

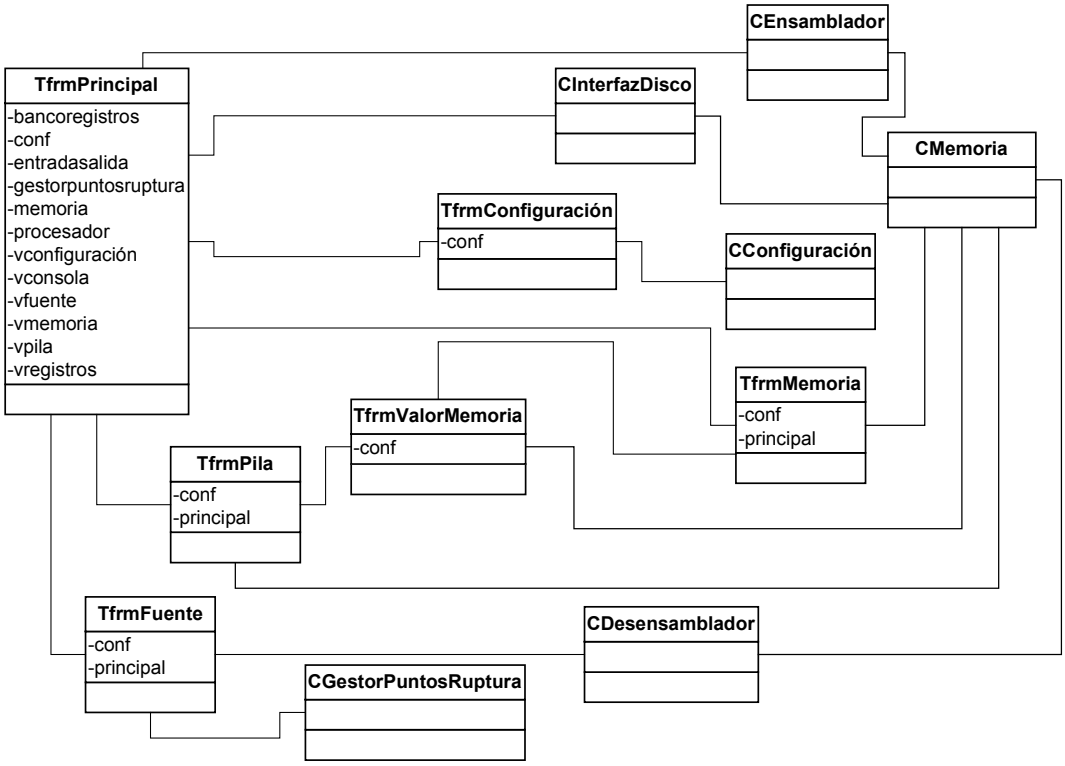


Figura 3.3.23. Diagrama de Clases de la Interfaz de Usuario en modo Gráfico

Clase TfrmConfiguración

Esta clase implementa la ventana de configuración de la herramienta, cuyo diseño visual (Figura 3.3.22) y funcional se ha detallado anteriormente.

TfrmConfiguración
- conf: ^CConfiguración
+ BotónAceptarClick() + BotónCancelarClick() + LeerTecla(IN key: Tecla) + MostrarConfiguración() + TfrmConfiguración(IN configuración: ^CConfiguración) + ~TfrmConfiguración()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase CConfiguración que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**BotónAceptarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnClick del botón Aceptar. Almacena las opciones de configuración en el objeto configuración e invoca al método que la almacena en disco. Cierra la ventana.
Errores	Si hubo algún problema al guardar el fichero de configuración, muestra un cuadro de diálogo informando del error.

Método	**BotónCancelarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnClick del botón Cancelar. Cierra la ventana. Por tanto, se pierden todos los cambios efectuados en la configuración.
Errores	Ninguno.

Método	**LeerTecla**
Parámetros	IN key: Tecla
Salida	Ninguna
Comportamiento	Responde al evento OnKeyDown de la ventana. Si la tecla pulsada ha sido ESCAPE, se cierra la ventana, perdiéndose todos los cambios efectuados en la configuración. Si se ha pulsado cualquier otra tecla, se ignora.
Errores	Ninguno.

Método	**MostrarConfiguración**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnShow de la ventana. Recupera los valores del objeto configuración y selecciona en la ventana el valor activo de cada pareja.
Errores	Ninguno.

Método	**TfrmConfiguración**
Parámetros	IN configuración: ^CConfiguración
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo conf el valor del parámetro de entrada configuración.
Errores	Ninguno.

Método	**~TfrmConfiguración**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Clase TfrmConsolaEjecución

Esta clase implementa la ventana correspondiente a la consola del simulador, cuyo diseño visual (Figura 3.3.13) y funcional se ha detallado anteriormente.

TfrmConsolaEjecución
- buffer: Cadena - modal: Booleano - principal: ^TfrmPrincipal - terminar: Booleano
+ Cerrar() + ConfirmarCerrar(OUT canclose: Booleano) + ConsolaResize() + CopiarExecute() + Copiar1Click() + DeseleccionarTodoExecute() + DeseleccionarTodo1Click() + EscribirModal(IN valor: Booleano) + EscribirTerminar(IN valor: Booleano) + LeerBuffer(): Cadena + LeerCaracteres(IN tecla: Carácter) + LeerTeclado(IN key: Tecla) + LimpiarExecute() + Limpiar1Click() + Modal(): Booleano + Mostrar() + MostrarCadena(IN cadena: Cadena) + SeleccionarTodoExecute() + SeleccionarTodo1Click() + Terminar(): Booleano + TfrmConsolaEjecución(IN ventana_principal: ^TfrmPrincipal) + ~TfrmConsolaEjecución()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
buffer	Almacena el texto escrito en la ventana de consola.
modal	Indica si la consola requiere entrada de datos por parte del usuario, lo que quiere decir que la ventana de la consola se representará en forma modal, no dejando acceder al resto de ventanas hasta que se complete la acción requerida.
principal	Almacena un puntero al objeto de la clase TfrmPrincipal que representa la ventana principal de la aplicación.

terminar	Indica si se ha de detener la ejecución a petición del usuario (el usuario pulsó la tecla ESCAPE).

Debido a que algunas funciones pueden ser accedidas de varias formas, se definen una serie de Acciones. Cuando se invoque alguna de estas acciones desde los distintos puntos de acceso (menú principal, menú contextual, botón, teclas de acceso rápido...), se lanzará un evento `OnExecute`, que será tratado por el método correspondiente según la siguiente relación:

Acción	Método que la implementa
Copiar	CopiarExecute
DeseleccionarTodo	DeseleccionarTodoExecute
Limpiar	LimpiarExecute
SeleccionarTodo	SeleccionarTodoExecute

Método	**Cerrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClose` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal. `modal = falso`. Cierra la ventana.
Errores	Ninguno.

Método	**ConfirmarCerrar**
Parámetros	OUT canclose: `Booleano`
Salida	Ninguna
Comportamiento	Responde al evento `OnCloseQuery` de la ventana. Si se está ejecutando una instrucción de entrada (`modal == cierto`), la ventana no se puede cerrar, se informa mediante un cuadro de diálogo y se devuelve `CanClose = falso`. Si no, `CanClose = cierto`.
Errores	Ninguno.

Método	**ConsolaResize**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnResize` de la ventana. Cambia el tamaño del cuadro Consola para mantener las proporciones.
Errores	Ninguno.

Método	**CopiarExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Copiar`. Copia en el portapapeles el texto seleccionado en el cuadro Consola.
Errores	Ninguno.

Método	**Copiar1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú Copiar. Invoca la ejecución de la acción `Copiar`.
Errores	Ninguno.

Método	**DeseleccionarTodoExecute**
Parámetros	Ninguno

Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `DeseleccionarTodo`. Elimina la selección de texto del cuadro Consola.
Errores	Ninguno.

Método	**DeseleccionarTodo1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú DeseleccionarTodo. Invoca la ejecución de la acción `DeseleccionarTodo`.
Errores	Ninguno.

Método	**EscribirModal**
Parámetros	IN valor: `Booleano`
Salida	Ninguna
Comportamiento	Modifica el valor del atributo `modal`.
Errores	Ninguno.

Método	**EscribirTerminar**
Parámetros	IN valor: `Booleano`
Salida	Ninguna
Comportamiento	Modifica el valor del atributo `terminar`.
Errores	Ninguno.

Método	**LeerBuffer**
Parámetros	Ninguno
Salida	`Cadena`
Comportamiento	Devuelve el valor del atributo `buffer`.
Errores	Ninguno.

Método	**LeerCaracteres**
Parámetros	IN tecla: `Carácter`
Salida	Ninguna
Comportamiento	Respuesta al evento `OnKeyPress` del cuadro Consola. Muestra la tecla en la consola. Hay que tratar especialmente la tecla de RETROCESO, borrando el último carácter introducido en la consola.
Errores	Ninguno.

Método	**LeerTeclado**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Respuesta al evento `OnKeyDown` del cuadro Consola. Sólo se hace tratamiento si `modal == cierto`. Si se lee la tecla ESCAPE, `terminar = cierto` (se detiene la ejecución si está en marcha) y se cierra la ventana. Si se lee la tecla ENTER, `terminar = cierto` (no se detiene la ejecución si está en marcha), se añade una línea al cuadro Consola y se cierra la ventana
Errores	Ninguno.

Método	**LimpiarExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Respuesta al evento `OnExecute` de la acción Limpiar. Borra el contenido del cuadro Consola.
Errores	Ninguno.

Método	**Limpiar1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Respuesta al evento `OnClick` de la opción de menú Limpiar. Invoca la ejecución de la acción `Limpiar`.
Errores	Ninguno.

Método	**Modal**
Parámetros	Ninguno
Salida	`Booleano`
Comportamiento	Devuelve el valor del atributo `modal`.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Respuesta al evento `OnShow` de la ventana. Pone el tick en la opción de menú Consola de la ventana principal. `terminar = falso`. Si `modal == cierto`, hay que borrar el atributo `buffer`.
Errores	Ninguno.

Método	**MostrarCadena**
Parámetros	IN cadena: `Cadena`
Salida	Ninguna
Comportamiento	Imprime en el cuadro Consola la cadena indicada por el parámetro de entrada `cadena`.
Errores	Ninguno.

Método	**SeleccionarTodoExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Respuesta al evento `OnExecute` de la acción `SeleccionarTodo`. Selecciona todo el texto contenido en el cuadro Consola.
Errores	Ninguno.

Método	**SeleccionarTodo1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Respuesta al evento `OnClick` de la opción de menú SeleccionarTodo. Invoca la ejecución de la acción `SeleccionarTodo`.
Errores	Ninguno.

Método	**Terminar**
Parámetros	Ninguno
Salida	`Booleano`
Comportamiento	Devuelve el valor del atributo `terminar`.
Errores	Ninguno.

Método	**TfrmConsolaEjecución**
Parámetros	IN ventana_principal: `^TfrmPrincipal`
Salida	Ninguno

Comportamiento	Es el constructor de la clase.
	Almacena en el atributo `principal` el valor del parámetro de entrada `ventana_principal`. Asimismo, inicializa el resto de los atributos:
	`buffer = "";`
	`modal = falso;`
	`terminar = falso;`

Errores	Ninguno.

Método	**~TfrmConsolaEjecución**
Parámetros	Ninguno
Salida	Ninguno
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Clase TfrmFuente

Esta clase implementa la ventana de visualización de código fuente, cuyo diseño visual (Figura 3.3.10) y funcional se ha detallado anteriormente.

TfrmFuente
- conf: ^`CConfiguración` - principal: ^`TfrmPrincipal`
+ Actualizar() + BotónCentrarClick() + BotónIrAClick() + Cerrar() + EditarPuntoRuptura() + LeerTecla(IN key: `Tecla`) + Mostrar() + Ocultar() + TfrmFuente(IN conf: ^`CConfiguración`, IN ventana_principal: ^`TfrmPrincipal`) + ~TfrmFuente() + VolcarFuente(IN dirección: `Entero`)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
principal	Almacena un puntero al objeto de la clase `TfrmPrincipal` que representa la ventana principal de la aplicación.

Método	**Actualizar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnPaint` de la ventana. Desensambla código a partir de la dirección indicada en la primera línea del cuadro de Código Fuente. Escribe dicha dirección en el cuadro de texto inferior.
Errores	Ninguno.

Método	**BotónCentrarClick**
Parámetros	Ninguno
Salida	Ninguna

Comportamiento	Responde al evento `OnClick` del botón Desensamblar A Partir de PC. Desensambla código a partir de la dirección contenida en `PC`.
Errores	Ninguno.

Método	**BotónIrAClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Desensamblar A Partir De. Desensambla código a partir de la dirección contenida en el cuadro de texto inferior, siempre que sea una dirección válida.
Errores	Si la dirección contenida en el cuadro de texto inferior no es válida, se muestra un cuadro de diálogo informando del error (*Valor Incorrecto*).

Método	**Cerrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClose` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal. `modal = falso;` Cierra la ventana.
Errores	Ninguno.

Método	**EditarPuntoRuptura**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnDblClick` del cuadro de Código Fuente. Comprueba la dirección sobre la que se ha hecho doble clic. Si había activo un punto de ruptura, se desactiva, y viceversa. Se invoca al método `Intercambiar` del gestor de puntos de ruptura.
Errores	Ninguno.

Método	**LeerTecla**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Responde al evento `OnKeyDown` de la ventana Si se ha pulsado la tecla ESCAPE, se cierra la ventana. Si se ha pulsado cualquier otra, se ignora.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnShow` de la ventana. Pone el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**Ocultar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnHide` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**TfrmFuente**
Parámetros	IN conf: `^CConfiguración` IN ventana_principal: `^TfrmPrincipal`
Salida	Ninguna

Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`, y en el atributo `principal` el valor del parámetro de entrada `ventana_principal`. Fija el ancho de las columnas del cuadro de Código Fuente (40, 40 y 165 puntos respectivamente).
Errores	Ninguno.

Método	**~TfrmFuente**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**VolcarFuente**
Parámetros	IN dirección: `Entero`
Salida	Ninguna
Comportamiento	Desensambla código a partir de la dirección indicada por el parámetro de entrada `dirección` y lo muestra en el cuadro de Código Fuente.
Errores	Ninguno.

Clase TfrmMemoria

Esta clase implementa la ventana de la memoria del simulador, cuyo diseño visual (Figura 3.3.15) y funcional se ha detallado anteriormente.

TfrmMemoria
- conf: ^`CConfiguración` - principal: ^`TfrmPrincipal`
+ Actualizar() + BotónIrAClick() + BotónReiniciarClick() + Cerrar() + EditarMemoria() + LeerTecla(IN key: `Tecla`) + Mostrar() + Ocultar() + TfrmMemoria(IN conf: ^`CConfiguración`, IN ventana_principal: ^`TfrmPrincipal`) + ~TfrmMemoria() + VolcarMemoria(IN dirección: `Entero`)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, el objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
principal	Almacena un puntero al objeto de la clase `TfrmPrincipal` que representa la ventana principal de la aplicación.

Método	**Actualizar**
Parámetros	Ninguno
Salida	Ninguna

Comportamiento	Responde al evento `OnPaint` de la ventana. Muestra el contenido de la memoria a partir de la dirección indicada por la fila superior del cuadro de Memoria.
Errores	Ninguno.

Método	**BotónIrAClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Ir A Dirección. Muestra el contenido de la memoria a partir de la dirección indicada en el cuadro de texto inferior.
Errores	Si la dirección es incorrecta, se muestra un cuadro de diálogo informando del error (*Valor Incorrecto*).

Método	**BotónReiniciarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Reiniciar Valores. Muestra un cuadro de diálogo con la pregunta *"¿Reiniciar la Memoria?"* y botones Aceptar y Cancelar. Si el usuario pulsa Aceptar, se invoca al método `Reiniciar` del objeto `memoria` y se invoca al método `RefrescarVentanas` de la ventana principal. Si el usuario pusa Cancelar, o cierra el cuadro de diálogo, no se hace ninguna modificación en la memoria del simulador.
Errores	Ninguno.

Método	**Cerrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClose` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**EditarMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnDblClick` del cuadro de Memoria. Captura la dirección de memoria sobre la que el usuario ha hecho doble clic y crea una ventana `TfrmValorMemoria` en la que el usuario puede editar el valor de dicha posición.
Errores	Ninguno.

Método	**LeerTecla**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Responde al evento `OnKeyDown` de la ventana. Si se ha pulsado la tecla ESCAPE, se cierra la ventana. Si se ha pulsado la tecla ENTER, se lanza un evento `Click` al botón Ir A Dirección. Si se ha pulsado cualquier otra tecla se ignora.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnShow` de la ventana. Pone el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno

Método	**Ocultar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnHide` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno

Método	**TfrmMemoria**
Parámetros	IN conf: ^`CConfiguración` IN ventana_principal: ^`TfrmPrincipal`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`, y en el atributo `principal` el valor del parámetro de entrada `ventana_principal`.
Errores	Ninguno.

Método	**~TfrmMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**VolcarMemoria**
Parámetros	IN dirección: `Entero`
Salida	Ninguna
Comportamiento	Muestra el contenido de la memoria a partir de la dirección indicada por el parámetro de entrada `dirección`.
Errores	Ninguno.

Clase TfrmPila

Esta clase implementa la ventana de la pila del simulador, cuyo diseño visual (Figura 3.3.21) y funcional se ha detallado anteriormente.

TfrmPila
- conf: ^`CConfiguración` - principal: ^`TfrmPrincipal`
+ Actualizar() + BotónCentrarClick() + BotónIrAClick() + Cerrar() + EditarMemoria() + LeerTecla(IN key: `Tecla`) + Mostrar() + Ocultar() + TfrmPila(IN conf: ^`CConfiguración`, IN ventana_principal: ^`TfrmPrincipal`) + ~TfrmPila() + VolcarMemoria(IN dirección: `Entero`)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, el objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
principal	Almacena un puntero al objeto de la clase `TfrmPrincipal` que representa la ventana principal de la aplicación.

Método	**Actualizar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnPaint` de la ventana. Muestra el contenido de la memoria a partir de la dirección indicada por la fila superior del cuadro de Pila.
Errores	Ninguno.

Método	**BotónCentrarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Ir A Puntero de Pila. Muestra el contenido de la memoria a partir de la dirección a la que apunta `SP`.
Errores	Ninguno.

Método	**BotónIrAClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Ir A Dirección. Muestra el contenido de la memoria a partir de la dirección indicada en el cuadro de texto inferior.
Errores	Ninguno

Método	**Cerrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClose` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**EditarMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnDblClick` del cuadro de Memoria. Captura la dirección de memoria sobre la que el usuario ha hecho doble clic y crea una ventana `TfrmValorMemoria` en la que el usuario puede editar el valor de dicha posición.
Errores	Ninguno.

Método	**LeerTecla**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Responde al evento `OnKeyDown` de la ventana. Si se ha pulsado la tecla ESCAPE, se cierra la ventana. Si se ha pulsado la tecla ENTER, se lanza un evento `Click` al botón Ir A Dirección. Si se ha pulsado cualquier otra tecla, se ignora.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnShow` de la ventana. Pone el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**Ocultar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnHide` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**TfrmPila**
Parámetros	IN conf: ^`CConfiguración` IN ventana_principal: ^`TfrmPrincipal`
Salida	Ninguno
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`, y en el atributo `principal` el valor del parámetro de entrada `ventana_principal`.
Errores	Ninguno.

Método	**~TfrmPila**
Parámetros	Ninguna
Salida	Ninguno
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Método	**VolcarMemoria**
Parámetros	IN dirección: `Entero`
Salida	Ninguna
Comportamiento	Muestra el contenido de la memoria a partir de la dirección indicada por el parámetro de entrada `dirección`.
Errores	Ninguno.

Clase TfrmPrincipal

Esta clase implementa la ventana principal de la herramienta, cuyo diseño visual (Figura 3.3.5) y funcional se ha detallado anteriormente.

TfrmPrincipal
- bancoregistros: ^`CBancoRegistros` - conf: ^`CConfiguración` - entradasalida: ^`CEntradaSalida` - gestorpuntosruptura: ^`CGestorPuntosRuptura` - memoria: ^`CMemoria` - procesador: ^`CProcesador` - vconfiguración: ^`TfrmConfiguración` - vconsola: ^`TfrmConsolaEjecución` - vfuente: ^`TfrmFuente` - vmemoria: ^`TfrmMemoria` - vpila: ^`TfrmPila` - vregistros: ^`TfrmRegistros`

+ AbrirEnsamblarExecute() + BotónEjecutarClick() + CargarImagenMemoriaExecute() + CargarPosiciónVentanas() : `Entero` + CodigoFuente1Click() + ConfiguraciónExecute() + ConfirmarCerrar(OUT canclose: `Booleano`) + Consola1Click() + CopiarExecute() + DeseleccionarTodoExecute() + EjecutarExecute() + EstadoAbrirEnsamblar() + EstadoCargarImagenMemoria() + EstadoConfiguración() + EstadoEjecutar() + EstadoGuardarImagenMemoria() + EstadoGuardarSesión() + EstadoMensajes() + GuardarImagenMemoriaExecute() + GuardarPosiciónVentanas() : `Entero` + GuardarSesiónExecute() + LeerTeclado(IN key: `Tecla`) + LimpiarExecute() + Memoria1Click() + MenúAcercaDeClick() + Mostrar() + MostrarEstado(IN estado: `Cadena`) + MostrarMensaje(IN mensaje: `Cadena`) + MostrarMensajeDesdeFichero(IN fichero: `Cadena`) + PararExecute() + PermisoEscribir() : `Booleano` + Pila1Click() + PrincipalResize() + RefrescarVentanas() + Registros1Click() + SalirExecute() + SeleccionarTodoExecute() + TfrmPrincipal() + ~TfrmPrincipal()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
bancoregistros	Almacena un puntero al objeto de la clase `CBancoRegistros`, que modeliza el comportamiento del banco de registros del sistema.
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
entradasalida	Almacena un puntero al objeto de la clase `CEntradaSalida`, que modeliza el comportamiento del módulo de entrada/salida del sistema.
gestorpuntosruptura	Almacena un puntero al objeto de la clase `CGestorPuntosRuptura`, que modeliza el comportamiento del gestor de puntos de ruptura del sistema.
memoria	Almacena un puntero al objeto de la clase `CMemoria`, que modeliza el comportamiento de la memoria del sistema.
procesador	Almacena un puntero al objeto de la clase `CProcesador`, que modeliza el comportamiento del procesador del sistema.

vconfiguración	Almacena un puntero al formulario correspondiente a la Ventana de Configuración (de la clase `TfrmConfiguración`).
vconsola	Almacena un puntero al formulario correspondiente a la Ventana de Consola de Ejecución (de la clase `TfrmConsolaEjecución`).
vfuente	Almacena un puntero al formulario correspondiente a la Ventana de CódigoFuente (de la clase `TfrmFuente`).
vmemoria	Almacena un puntero al formulario correspondiente a la Ventana de Memoria (de la clase `TfrmMemoria`).
vpila	Almacena un puntero al formulario correspondiente a la Ventana de Pila (de la clase `TfrmPila`).
vregistros	Almacena un puntero al formulario correspondiente a la Ventana de Banco de Registros (de la clase `TfrmRegistros`).

Debido a que algunas funciones pueden ser accedidas de varias formas, se definen una serie de Acciones. Cuando se invoque alguna de estas acciones desde los distintos puntos de acceso (menú principal, menú contextual, botón, teclas de acceso rápido...), se lanzará un evento `OnExecute`, que será tratado por el método correspondiente según la siguiente relación:

Acción	Método que la implementa
CargarImagenMemoria	CargarImagenMemoriaExecute
Copiar	CopiarExecute
Deseleccionar	DeseleccionarExecute
Ejecutar	EjecutarExecute
GuardarImagenMemoria	GuardarImagenMemoriaExecute
GuardarSesión	GuardarSesiónExecute
Limpiar	LimpiarExecute
Parar	PararExecute
Salir	SalirExecute
SeleccionarTodo	SeleccionarTodoExecute

Método	**AbrirEnsamblarExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `AbrirEnsamblar`. Muestra un diálogo de Abrir Fichero para que el usuario seleccione el fichero que quiere ensamblar. Crea una instancia del ensamblador e invoca su método `Ensamblar`. Recoge los resultados del proceso y muestra la información correspondiente en el cuadro de mensajes. Se refresca el contenido de todas las ventanas.
Errores	Ninguno.

Método	**BotónEjecutarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Ejecutar. Si se está ejecutando un programa, se invoca la acción `Parar`. Si no, se invoca la acción `Ejecutar`.
Errores	Ninguno.

Método	**CargarImagenMemoriaExecute**
Parámetros	Ninguno
Salida	Ninguna

Comportamiento	Responde al evento `OnExecute` de la acción `CargarImagenMemoria`. Muestra un diálogo de Abrir Fichero para que el usuario seleccione el fichero que quiere cargar. Crea una instancia de la interfaz de disco e invoca el método `VolcarDiscoAMemoria`. Recoge los resultados del proceso y muestra la información correspondiente en el cuadro de mensajes.
Errores	Ninguno.

Método	**CargarPosiciónVentanas**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Carga la posición, el tamaño y el estado de las ventanas de la herramienta desde el fichero `wens2001.cfg`. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si se produjo un error al abrir el fichero. En este caso, toma los valores de diseño para la posición, tamaño y estado de las ventanas.

Método	**CódigoFuente1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú Ventanas>CódigoFuente. Si la opción de menú está marcada con un *tick*, cierra la ventana de código fuente. Si no, la abre.
Errores	Ninguno.

Método	**ConfiguraciónExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Configuración`. Muestra la ventana de configuración.
Errores	Ninguno.

Método	**ConfirmarCerrar**
Parámetros	OUT canclose: `Booleano`
Salida	Ninguna
Comportamiento	Responde al evento `OnCloseQuery` de la ventana. Muestra un cuadro de diálogo con el texto *¿Desea abandonar la aplicación?* y sendos botones de Aceptar y Cancelar. Si el usuario acepta, se guarda la posición y estado de las ventanas, se liberan los recursos usados, se borran los archivos temporales y se hace `CanClose = cierto`. Si no, se hace `CanClose = falso`.
Errores	Ninguno.

Método	**Consola1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú Ventanas>Consola. Si la opción de menú está marcada con un *tick*, cierra la ventana de consola. Si no, la abre.
Errores	Ninguno

Método	**CopiarExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Copiar`. Copia en el portapapeles el texto seleccionado en el cuadro de mensajes.
Errores	Ninguno.

Método	**DeseleccionarTodoExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Deseleccionar`. Elimina la selección de texto del cuadro de mensajes.
Errores	Ninguno.

Método	**EjecutarExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Ejecutar`. Invoca al método `Ejecutar` del objeto `procesador` para que comience la simulación. Cuando ésta acabe, se recoge el resultado y se muestra en el cuadro de mensajes.
Errores	Ninguno.

Método	**EstadoAbrirEnsamblar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del botón AbrirEnsamblar. Escribe en la barra de estado el texto *Abrir y Ensamblar*.
Errores	Ninguno.

Método	**EstadoCargarImagenMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del botón CargarImagenMemoria. Escribe en la barra de estado el texto *Cargar Imagen de Memoria*.
Errores	Ninguno.

Método	**EstadoConfiguración**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del botón Configuración. Escribe en la barra de estado el texto *Configuración*.
Errores	Ninguno.

Método	**EstadoEjecutar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del botón Ejecutar. Escribe en la barra de estado el texto *Ejecutar*.
Errores	Ninguno.

Método	**EstadoGuardarImagenMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del botón GuardarImagenMemoria. Escribe en la barra de estado el texto *Guardar Imagen de Memoria*.
Errores	Ninguno.

Método	**EstadoGuardarSesión**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del botón GuardarSesión. Escribe en la barra de estado el texto *Guardar Sesión*.
Errores	Ninguno.

Método	**EstadoMensajes**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnEnter` del cuadro de mensajes. Borra el texto de la barra de estado.
Errores	Ninguno.

Método	**GuardarImagenMemoriaExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `GuardarImagenMemoria`. Muestra un diálogo de Guardar Fichero para que el usuario seleccione el fichero que quiere salvar. Crea una instancia de la interfaz de disco e invoca el método `VolcarMemoriaADisco`. Recoge los resultados del proceso y muestra la información correspondiente en el cuadro de mensajes.
Errores	Ninguno.

Método	**GuardarPosiciónVentanas**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Guarda la posición, el tamaño y el estado de las ventanas de la herramienta en el fichero `wens2001.cfg`. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si se produjo un error al guardar el fichero.

Método	**GuardarSesiónExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `GuardarSesión`. Muestra un cuadro de diálogo Guardar Fichero para que el usuario seleccione el fichero que quiere salvar. Guarda la cabecera ---- *Mensajes* ----. A continuación, añade el contenido del cuadro de mensajes. Después, inserta la cabecera ---- *Consola* ----. Por último, copia el contenido del cuadro de consola.
Errores	Si hay un error al guardar el fichero, se muestra un cuadro de diálogo con el texto *Error guardando fichero*.

Método	**LeerTeclado**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Si se ha pulsado la tecla ESCAPE, se inicia el proceso de cierre de la aplicación. Si se ha pulsado cualquier otra tecla se ignora.
Errores	Ninguno.

Método	**LimpiarExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Limpiar`. Borra el contenido del cuadro de mensajes.
Errores	Ninguno.

Método	**Memoria1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú Ventanas>Memoria. Si la opción de menú está marcada con un *tick*, cierra la ventana de memoria. Si no, la abre.
Errores	Ninguno.

Método	**MenúAcercaDeClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú AcercaDe. Muestra el cuadro de diálogo Acerca De.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnShow` de la ventana. Actualiza el gráfico del botón Ejecutar según sea el estado del simulador (parado, paso a paso o ejecutando).
Errores	Ninguno.

Método	**MostrarEstado**
Parámetros	IN estado: `Cadena`
Salida	Ninguno
Comportamiento	Muestra la cadena indicada por el parámetro de entrada en la barra de estado.
Errores	Ninguno.

Método	**MostrarMensaje**
Parámetros	IN mensaje: `Cadena`
Salida	Ninguna
Comportamiento	Añade en una nueva línea del cuadro de mensajes la cadena indicada por el parámetro de entrada.
Errores	Ninguno.

Método	**MostrarMensajeDesdeFichero**
Parámetros	IN fichero: `Cadena`
Salida	Ninguna
Comportamiento	Añade al cuadro de mensajes líneas de texto desde el fichero indicado por el parámetro de entrada.
Errores	Ninguno.

Método	**PararExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Parar`. Detiene la simulación.
Errores	Ninguno.

Método	**PermisoEscribir**
Parámetros	Ninguno
Salida	`Booleano`
Comportamiento	Devuelve `cierto` si la simulación está detenida y `falso` en caso contrario.
Errores	Ninguno.

Método	**Pila1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú Ventanas>Pila. Si la opción de menú está marcada con un *tick*, cierra la ventana de pila. Si no, la abre.
Errores	Ninguno.

Método	**PrincipalResize**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnResize` de la ventana. Actualiza el tamaño y la posición del cuadro de mensajes y la barra de estado para mantener las proporciones en el diseño de la ventana.
Errores	Ninguno.

Método	**RefrescarVentanas**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Invoca a los métodos `Refrescar` de todas las ventanas activas.
Errores	Ninguno.

Método	**Registros1Click**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` de la opción de menú Ventanas>Registros. Si la opción de menú está marcada con un tick, cierra la ventana de registros. Si no, la abre.
Errores	Ninguno

Método	**SalirExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `Salir`. Inicia el proceso de cierre de la aplicación.
Errores	Ninguno.

Método	**SeleccionarTodoExecute**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnExecute` de la acción `SeleccionarTodo`. Selecciona todo el texto contenido en el cuadro de mensajes.
Errores	Ninguno.

Método	**TfrmPrincipal**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Crea un objeto de la clase `CConfiguración` y almacena su referencia en el atributo `conf`. Crea objetos para los componentes del simulador (memoria, banco de registros, procesador, entrada/salida y gestor de puntos de ruptura) y almacena las referencias en sus respectivos atributos. Reinicia el contenido de la memoria y el banco de registros. Crea las ventanas (configuración, banco de registros, memoria, pila, código fuente y consola de entrada/salida) y almacena las referencias en sus respectivos atributos. Invoca al método `CargarConfiguración` del objeto conf. Muestra u oculta las ventanas según la configuración que acaba de leer.
Errores	Ninguno

Método	**~TfrmPrincipal**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno

Clase TfrmRegistros

Esta clase implementa la ventana del banco de registros del simulador, cuyo diseño visual (Figura 3.3.19) y funcional se ha detallado anteriormente.

TfrmRegistros
- conf: ^CConfiguración - principal: ^TfrmPrincipal
+ Actualizar() + BotónReiniciarClick() + Cerrar() + EditarRegistro(In registro: Entero) + LeerTecla(IN key: Tecla) + Mostrar() + Ocultar() + TfrmRegistros(IN conf: ^CConfiguración, IN ventana_principal: ^TfrmPrincipal) + ~TfrmRegistros()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase CConfiguración que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
principal	Almacena un puntero al objeto de la clase TfrmPrincipal que representa la ventana principal de la aplicación.

Método	**Actualizar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnPaint de la ventana. Actualiza el valor de todos los campos que contiene la ventana.
Errores	Ninguno.

Método	**BotónReiniciarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnClick del botón Reiniciar. Muestra un cuadro de diálogo con el mensaje *¿Reiniciar el Banco de Registros?* y botones de Aceptar y Cancelar. Si el usuario pulsa Aceptar, se invoca al método Reiniciar del objeto banco de registros, y se actualiza la ventana. Si el usuario pulsa el botón Cancelar, o cierra el cuadro de diálogo, no se efectúa ningún cambio en el banco de registros.
Errores	Ninguno.

Método	**Cerrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnClose de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**EditarRegistro**
Parámetros	IN registro: Entero
Salida	Ninguno

Comportamiento	Responde al evento `OnDblClick` del cuadro de registros. Crea una ventana `TfrmValorRegistro` en la que el usuario puede editar el valor del registro indicado por el parámetro de entrada.
Errores	Ninguno.

Método	**LeerTecla**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Responde al evento `OnKeyDown` de la ventana. Si se ha pulsado la tecla ESCAPE, se cierra la ventana. Si se ha pulsado cualquier otra tecla, se ignora.
Errores	Ninguno

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnShow` de la ventana. Pone el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**Ocultar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnHide` de la ventana. Quita el *tick* en la opción de menú Consola de la ventana principal.
Errores	Ninguno.

Método	**TfrmRegistros**
Parámetros	IN configuración: `^CConfiguración` IN ventana_principal: `^TfrmPrincipal`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`, y en el atributo `principal` el valor del parámetro de entrada `ventana_principal`.
Errores	Ninguno.

Método	**~TfrmRegistros**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Clase TfrmValorMemoria

Esta clase implementa el cuadro de diálogo que permite modificar los valores de la memoria del simulador, cuyo diseño visual (Figura 3.3.18) y funcional se ha detallado anteriormente.

TfrmValorMemoria
- conf: `^CConfiguración`
+ BotónAceptarClick() + LeerTecla(IN key: `Tecla`) + Mostrar() + TfrmValorMemoria(IN configuración: `^CConfiguración`) + ~TfrmValorMemoria()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.

Método	**BotónAceptarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnClick` del botón Aceptar. Si el valor introducido en el cuadro de texto es un entero de 16 bits correcto, actualiza el contenido de la dirección de memoria indicada en la etiqueta. A continuación, cierra la ventana.
Errores	Si el valor introducido en el cuadro de texto es incorrecto, muestra un cuadro de diálogo con el mensaje *"Valor Incorrecto"*.

Método	**LeerTecla**
Parámetros	IN key: `Tecla`
Salida	Ninguna
Comportamiento	Responde al evento `OnKeyDown` de la ventana. Si se ha pulsado la tecla ESCAPE, se cierra la ventana. Si se ha pulsado cualquier otra tecla, se ignora.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento `OnShow` de la ventana. Da el foco al cuadro de texto.
Errores	Ninguno.

Método	**TfrmValorMemoria**
Parámetros	IN configuración: `^CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`
Errores	Ninguno.

Método	**~TfrmValorMemoria**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

Clase TfrmValorRegistro

Esta clase implementa la ventana que permite modificar los valores del banco de registros del simulador, cuyo diseño visual (Figura 3.3.20) y funcional se ha detallado anteriormente.

TfrmValorRegistro
- conf: ^CConfiguración - reg: Entero
+ BotónAceptarClick() + EscribirReg(IN valor: Entero) + LeerTecla(IN key: Tecla) + Mostrar() + Reg() : Entero + TfrmValorRegistro(IN configuración: ^CConfiguración) + ~TfrmValorRegistro()

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
conf	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase CConfiguración que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
reg	Almacena el identificador del registro cuyo valor se está editando.

Método	**BotónAceptarClick**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnClick del botón Aceptar. Si el valor introducido en el cuadro de texto es un entero de 16 bits correcto, actualiza el contenido del registro indicado por el atributo reg. A continuación, cierra la ventana.
Errores	Si el valor introducido en el cuadro de texto es incorrecto, se muestra un cuadro de diálogo con el mensaje *"Valor Incorrecto"*. Si el registro seleccionado es SP, el valor introducido pertenece a la zona de código y está activada la comprobación, se muestra un cuadro de diálogo con el mensaje *"Valor Incorrecto: SP invade la zona de código"*. Si el registro seleccionado es PC, el valor introducido pertenece a la zona de pila y está activada la comprobación, se muestra un cuadro de diálogo con el mensaje *"Valor Incorrecto: PC invade la zona de pila"*.

Método	**EscribirReg**
Parámetros	IN valor: Entero
Salida	Ninguna
Comportamiento	Modifica el valor del atributo reg.
Errores	Ninguno.

Método	**LeerTecla**
Parámetros	IN key: Tecla
Salida	Ninguna
Comportamiento	Responde al evento OnKeyDown de la ventana. Si se ha pulsado la tecla ESCAPE, se cierra la ventana. Si se ha pulsado cualquier otra tecla, se ignora.
Errores	Ninguno.

Método	**Mostrar**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Responde al evento OnShow de la ventana. Da el foco al cuadro de texto.
Errores	Ninguno.

Método	**Reg**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Devuelve el valor del atributo `reg`.
Errores	Ninguno.

Método	**TfrmValorRegistro**
Parámetros	IN configuración: `^CConfiguración`
Salida	Ninguna
Comportamiento	Es el constructor de la clase. Almacena en el atributo `conf` el valor del parámetro de entrada `configuración`
Errores	Ninguno.

Método	**~TfrmValorRegistro**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno.

3.4. Implementación

En este capítulo se efectúan comentarios acerca de los aspectos más relevantes de la fase de implementación. No se trata de hacer una descripción del código línea a línea. Por tanto, sólo se van a describir aquellos detalles que no puedan derivarse directamente de aplicar la información extraída de los capítulos dedicados al diseño de la herramienta.

3.4.1. Detalles generales

Tanto las constantes numéricas como las cadenas de caracteres se definen en ficheros aparte, con directivas `#define` del compilador de *C*/*C*++. Con esto se consigue clarificar el código, en el caso de las constantes numéricas, y facilitar la compilación de versiones multilenguaje o con distintas tablas de códigos, en el caso de las cadenas de caracteres.

A la hora de escribir código, ya sea *C* o *C*++, se separa la parte de declaraciones de la parte de implementaciones. Además, en el caso de la parte escrita en *C*, se agrupa el código por funcionalidades, mientras que en la parte *C*++ cada clase se escribe en ficheros separados.

Como añadido a los ficheros de código fuente *C*/*C*++, se encuentran los ficheros de definición de gramática para *Flex* y *Bison*, `ens.l` y `ens.y` respectivamente. Por tanto, el proceso de compilado de la aplicación constará de los siguientes pasos:

- Generación del fuente en *C* para el analizador léxico, usando la herramienta *Flex*. Se generan los ficheros `ens_lex.h` y `ens_lex.c`.
- Generación del fuente en *C* para el analizador sintáctico, usando la herramienta *Bison*. Se generan los ficheros `ens_tab.h` y `ens_tab.c`.
- Se compilan todos los ficheros. Dependiendo de la versión, se incluirán en esta compilación los ficheros de interfaz consola o interfaz gráfica.

También hay que indicar que la pantalla de ayuda mostrada por la versión consola se encuentra dentro del fichero `ayuda.txt`.

Por último, se debe señalar que la interfaz gráfica va a construirse mediante la herramienta *Borland C++ Builder*. Cada ventana constará de una clase que dará soporte a las operaciones y de una descripción gráfica de su aspecto.

3.4.2. Comunicación entre Módulos

Como se ha visto a lo largo del proceso de análisis y diseño, se ha dividido la aplicación en dos módulos bien diferenciados, ensamblador y simulador. El primero de ellos, generado automáticamente en su mayor parte, está escrito en lenguaje *C*. El segundo está escrito en *C++*. Ambos módulos comparten información. En concreto, el ensamblador debe transmitir al simulador el código máquina generado, si el programa fuente era correcto, o bien una lista de errores si el programa era incorrecto.

El problema de la comunicación se ha resuelto mediante el uso de dos ficheros temporales.

El primero de ellos, `memoria.tmp`, es un fichero de 128 Kb de longitud en el que se almacena una imagen exacta de la memoria del simulador después de un ensamblado correcto. Como el acceso al fichero se hace *byte* a *byte*, las palabras se colocan en disposición *big-endian* dentro del mismo. El fichero no contiene ningún tipo de información adicional de control, por lo que en principio se puede leer cualquier fichero, que la herramienta interpretará como una sucesión de palabras de 16 bits y así las almacenará en la memoria del simulador.

El segundo fichero, `errores.tmp`, se genera cuando el ensamblador ha encontrado errores en el código fuente. Se trata de un fichero de texto plano en el que se graban los errores, uno por línea, con el formato visto durante el diseño del ensamblador:

```
Línea n: ERROR[código] Descripción – token erróneo
```

Los ficheros temporales serán generados por el ensamblador a lo largo de la sesión según sea conveniente, y se eliminarán cuando el usuario salga de la aplicación. Sin embargo, de cara a las pruebas del ensamblador, podría ser interesante mantener los ficheros generados, lo que permitiría la automatización de las pruebas. Podrían generarse *scripts* que lanzaran los casos de prueba y que al terminar el proceso compararan el resultado con lo esperado de manera automática.

3.4.3. Implementación del Ensamblador

Como se mencionó en los capítulos de introducción, la construcción del módulo ensamblador se va a realizar con ayuda de dos herramientas de generación automática de analizadores, como son *Flex* y *Bison* (versiones de *Lex* y *Yacc* respectivamente). Por tanto, en este apartado, la tarea consiste en que, a partir de las gramáticas generadas en el proceso de análisis, se crearán unos ficheros de definición del analizador léxico y sintáctico, respectivamente, que con el apoyo de otras funciones den forma al conjunto del módulo ensamblador.

Flex es una herramienta que traduce la especificación de un analizador léxico a un programa escrito en *C*, que lo implementa. Para especificarlo, se emplean expresiones regulares (la parte derecha de las reglas) a las que se puede asociar acciones escritas en *C*. Cada vez que el analizador encuentre en la cadena de entrada una secuencia que encaje en una de las expresiones regulares especificadas, ejecutará la acción que se le haya asociado.

Por otra parte, *Bison* es una herramienta que traduce la especificación de una gramática de contexto libre a un programa en *C*, implementando un analizador *LALR(1)* que reconoce frases de esa gramática. El analizador *LALR(1)* generado es un analizador ascendente de desplazamiento-reducción. Además de reconocer frases se puede asociar código *C* a las reglas de la gramática, lo que permite especificar acciones semánticas, que se ejecutarán cada vez que se aplique la regla correspondiente.

Se puede consultar más información acerca del diseño y funcionamiento de *Flex* y *Bison* en [Levine 1995].

3.4.3.1. Analizador Léxico

A partir de la gramática ya definida, se va a generar el fichero de entrada para *Flex*. Su contenido será casi definitivo, quizás a falta de algún que otro detalle de implementación. Este fichero se compone básicamente de un conjunto de reglas. En la parte izquierda se encuentran las expresiones regulares que definen cadenas del lenguaje. En la parte derecha se ubica la lista de acciones que se ejecutarán cuando se cumpla la parte izquierda de la regla.

A la hora de inicializar el analizador léxico, hay que tener en cuenta que se va a trabajar con el binomio *Flex*-*Bison*, por lo que es importante saber cómo se le debe indicar cuál es el fichero de entrada, cómo se van a devolver los *tokens* y sus atributos, qué tipo de errores se van a controlar, cómo se informará de que se ha encontrado el final del archivo de entrada y cómo se dará por concluido el proceso de análisis.

El analizador sintáctico se encarga de abrir el fichero de entrada y de inicializar las variables. Cada vez que necesite un *token*, llamará a la función `yylex()`, que en realidad invoca al analizador léxico. Éste comenzará a leer caracteres de la entrada hasta que forme con ellos una cadena que se ajuste a alguno de los patrones definidos en la parte derecha de las reglas, y en ese momento ejecutará las acciones que se hayan asociado a cada regla.

Los *tokens* se pasan como valor de retorno de la función `yylex()`. Dicho valor de retorno es un entero. Para dar valor a los atributos de los *tokens*, se dispone de la variable `yylval`, cuyo tipo se puede definir arbitrariamente. En este caso, será necesario poder definir atributos como enteros, lógicos y cadenas de caracteres. Concretamente, sólo se devuelven cadenas de caracteres en un par de ocasiones, para el *token* `<IDENTIFICADOR>` y los *tokens* del tipo `<Modo_Direccionamiento_E>`. Se ha tomado la decisión de que, en estos dos casos, se almacenará el atributo en dos variables auxiliares, `etiqueta` y `cadena`, respectivamente. Los lógicos pueden representarse como enteros 0 y 1, así que la variable `yylval` se definirá de tipo entero. Otra variable interesante que aporta la herramienta es `yytext`, que almacena la cadena de caracteres con la que se ha equiparado la regla, esto es, que ha dado lugar al *token*.

Además, será necesario definir una variable que sirva de contador de líneas (`nlin`).

El conjunto de reglas se extrae directamente de la gramática. Para simplificar la lectura, se considerará que se han definido constantes enteras para cada uno de los *tokens* (`<MNEMÓNICO0>`, `<MNEMÓNICO1>`...), cada código de operación (`NOP`=0, `HALT`=1...), cada pseudoinstrucción (`ORG`, `EQU`...) y cada nombre de registro según su codificación (`R0`=0, `R1`=1...).

Para ignorar espacios, tabuladores y comentarios, se incluyen reglas cuya parte derecha se ajuste con ellos, pero que no definan ninguna acción.

```
[ |t]+
;.*
```

La primera de ellas se equipara a cualquier combinación de espacios y tabuladores, con la única condición de que la longitud del patrón sea al menos de un carácter. La segunda se equipara a una cadena de longitud indeterminada que comience por el carácter punto y coma, y que esté compuesta por cualquier carácter menos el de fin de línea (es lo que significa el carácter `'.'` para *Flex*).

Las reglas que no tienen el axioma en la parte izquierda conforman una serie de patrones auxiliares.

```
decimal           -?[0-9][0-9]*
decimalsinsigno   [0-9][0-9]*
hexadecimal       0x[0-9a-fA-F][0-9a-fA-F]*
ident             [a-zA-Z][a-zA-Z0-9_]*
cadena            \"[^\n^\"]*\"
```

Estos patrones, que funcionan a modo de *alias* se definen en el primer bloque del fichero de definición del analizador léxico y se pueden usar en la parte derecha de las reglas del segundo bloque.

El carácter fin de línea tiene un tratamiento especial, ya que además de enviar el *token* `<EOL>`, incrementará el contador interno de líneas (que se usará más adelante para ubicar los errores en el código fuente).

```
\n          {nlin++;
             return(EOL);}
```

El carácter fin de fichero tiene el mismo tratamiento que el fin de línea, sólo que además habrá que indicar de alguna manera que se ha acabado la entrada de caracteres, y que se deberá finalizar el análisis. Esto se hace empleando la función `yyterminate()`.

```
<<EOF>>     {yyterminate();
             nlin ++;
             return(EOL);}
```

Por cada mnemónico se define un patrón. El *token* devuelto dependerá del número de operandos que permita cada mnemónico, como se dispuso en el análisis. El atributo devuelto es el código de operación. No se van a enumerar todos los casos ya que son todos idénticos en su comportamiento y no aportan información adicional. Sirvan a modo de ejemplo los siguientes:

```
nop|NOP     {yyval = NOP;
             return(MNEMONICO0);}
move|MOVE   {yyval = MOVE;
             return(MNEMONICO2);}
inc|INC     {yyval = INC;
             return(MNEMONICO1);}
...
org|ORG     {yylval = ORG;
             return(ORG);}
...
```

Los signos de puntuación y operadores aritméticos dan lugar a un *token* cada uno de ellos. No se transmite más información hacia el analizador sintáctico.

```
,           {return(SEPARADOR);}
:           {return(FIN_ETIQ);}
"           {return(COMILLAS);}
+           {return(SUMA);}
...
```

Ahora restan aquellos *tokens* que se refieren a operandos. Teniendo en cuenta los patrones auxiliares que se definieron al comienzo, la traducción desde la gramática es prácticamente inmediata. Eso sí, serán necesarias funciones que calculen el valor entero para enviarlo como atributo, cuando la cadena de entrada sea un número decimal o hexadecimal.

```
#{decimal}        {yylval = Decimal(yytext+1);
                   return(INMEDIATO_V);}
#{hexadecimal}    {yylval = Hexadecimal(yytext+1);
                   return(INMEDIATO_V);}
#{etiqueta}       {etiqueta = yytext+1;
                   return(INMEDIATO_E);}
...
```

En este punto, el analizador léxico es capaz de determinar si se ha producido un error en la entrada, en concreto, detectar si se ha introducido un entero fuera de rango. Por tanto, al calcular el valor entero, ha de comprobar que es correcto, de la siguiente forma:

```
#{entero}         {yylval = Decimal(yytext+1);
                   if(FueraDeRango(yylval)){
                        GestorErrores(ENTERO_FUERA_DE_RANGO,
                                      nlin+1,yytext);
                   }
                   return(INMEDIATO_V);}
```

Para equiparar cualquier otra entrada que no se haya ajustado a ninguno de los patrones anteriores, se incluirá como última regla del segundo bloque aquélla que tiene el patrón . (punto) a la izquierda, que se ajusta a cualquier carácter menos el salto de línea, y se informará del error, ya que se ha encontrado en el fichero de código fuente un carácter no válido para el lenguaje.

```
.           {GestorErrores(ENTRADA_INCORRECTA,nlin+1,yytext);}
```

Hay que recordar que la variable `nlin` lleva la cuenta del número de líneas completas leídas y analizadas, luego se informará del error en la línea `nlin+1`, ya que la línea actual aún no se ha leído por completo. También se indicará al gestor de errores del carácter o conjunto de caracteres que produjo el error.

Ya se ha traducido toda la gramática previa a un formato a partir del cual *Flex* puede realizar su trabajo y generar el analizador. Sin embargo, surge un pequeño problema. Debido a la forma de trabajar de *Flex*, tal y como se han definido las reglas existe la posibilidad de obtener un resultado no deseado ante una entrada como:

```
etiq1 : equ 10/2
```

La secuencia de *tokens* debería ser:

```
<IDENTIFICADOR>, <FIN_ETIQ>, <EQU>, <DECIMAL>, <DIVISIÓN>, <DECIMAL>
```

Pero se obtendrá:

```
<IDENTIFICADOR>, <FIN_ETIQ>, <EQU>, <DECIMAL>, <MEMORIA_V>
```

Para resolver ambigüedades, *Flex* dispone de algunos mecanismos. El que se aplica de forma más inmediata consiste en que los patrones se intentan ajustar en orden según aparecen en el fichero de definición, por lo que, si dos patrones son válidos, la entrada se equiparará con el que aparezca antes en el fichero de definición. Pero cuando este mecanismo no es suficiente, se debe usar otra característica de *Flex*, la posibilidad de crear estados para el analizador. ¿En qué consiste? El analizador léxico generado por *Flex* es capaz de guardar un estado que recordará de una invocación a otra de la función `yylex()`. Existe al menos un estado, que se denomina `INITIAL`, y es posible definir cuantos se necesiten, que pueden ser exclusivos o no.

En este caso particular, los problemas aparecen al analizar expresiones, por lo que se definirá un estado aparte en el que se encontrará durante el análisis de una expresión, y se empleará el inicial para el resto del tiempo. Esto se consigue incluyendo en el primer bloque la línea:

```
%x EXPR
```

Ahora, si se desea que un patrón se equipare en cualquier estado, se deberá anteponer la expresión `<INITIAL,EXPR>`. En cambio, si se desea que sólo se equipare durante el análisis de expresiones, se antepondrá `<EXPR>`. Para el resto de los casos, no es necesario añadir nada.

Para cambiar de estado se utiliza la sentencia `BEGIN`. Por tanto, se añadirá en la parte derecha de aquellos patrones tras los cuales se espera leer una expresión:

```
org|ORG          {yylval=ORG;
                  BEGIN(EXPR);
                  return(ORG);}
equ|EQU          {yylval = EQU;
                  BEGIN(EXPR);
                  return(EQU);}
res|RES          {yylval = RES;
                  BEGIN(EXPR);
                  return(RES);}
```

Y se retornará al estado inicial tras leer un salto de línea (si no se estaba leyendo una expresión, el cambio de estado no tiene efecto).

```
\n               {nlin++;
                  BEGIN(INITIAL);
                  return(EOL);}
```

Además, se puede aprovechar esta modificación para detectar errores dentro de las expresiones, y proporcionar una información un poco más detallada al usuario, si se hace un cambio en el patrón . (punto), de forma que se comunique al gestor de errores un error distinto si se estaba analizando una expresión.

```
.           {if(ESTADO==EXPR){
                  InformarError(EXPRESIÓN_INCORRECTA,
                                    nlin+1,yytext);}
             else{
                  InformarError(ENTRADA_INCORRECTA,
                                    nlin+1,yytext);}}
```

Otro estado que puede ser interesante diferenciar es la lectura de la lista de datos de la pseudoinstrucción DATA. En ese caso sólo se pueden leer números enteros, cadenas de caracteres entrecomilladas y el carácter separador (coma), amén del carácter de fin de línea. Por tanto, se va a definir un nuevo estado:

```
%x DATOS
```

Habrá que cambiar de estado cuando se lea la pseudoinstrucción DATA:

```
data|DATA         {yylval=DATA;
                   BEGIN(DATOS);
                   return(DATA);}
```

E, igual que en el caso anterior, volver al estado inicial tras leer el carácter de fin de línea. Además, se puede aprovechar para detectar otro tipo de error y, nuevamente, que el analizador sea más explícito a la hora de informar de los errores:

```
.           {if(ESTADO==EXPR){
                  InformarError(EXPRESIÓN_INCORRECTA,
                                    nlin+1,yytext);}
             else if(ESTADO==DATOS){
                  InformarError(LISTA_DE_DATOS_INCORRECTA,
                                    nlin+1,yytext);}
             else{
                  InformarError(ENTRADA_INCORRECTA,
                                    nlin+1,yytext);}}
```

Por último, para llevar un control de los *tokens* leídos en cada línea, se pueden almacenar en una lista, que se irá borrando al comienzo de la siguiente línea. Así, a la hora de informar los errores, se pueden recuperar *tokens* leídos previamente dentro de la misma instrucción.

3.4.3.2. Analizador Sintáctico, Semántico y Generador de Código

La construcción del analizador sintáctico está basada en el complemento de *Flex*, que es *Bison*. Al igual que en el caso anterior, se va a construir un fichero que defina el analizador, de manera que la herramienta lo genere de forma automática.

En la parte exclusivamente sintáctica, el proceso consiste en una traducción directa de la gramática construida durante la fase de análisis. En cuanto a las acciones semánticas, la traducción también es bastante inmediata, no obstante, se irán comentando los detalles que vayan surgiendo hasta llegar al resultado final.

En el primer bloque del fichero de definición del analizador sintáctico se enumeran los *tokens* que se van a generar:

```
%token MNEMÓNICO0
%token MNEMÓNICO1
```

```
%token MNEMÓNICO2
...
```

A continuación se indica qué regla constituye el axioma de la gramática:

```
%start s
```

Y para cerrar este primer bloque, se definen las variables necesarias. Es necesario controlar los siguientes aspectos:

- Número de línea que se está analizando.
- Operando dentro de una instrucción que se está analizando.
- Indicador de si se produjo algún error analizando (no se debe generar código).
- Lugar donde almacenar internamente la instrucción que se está analizando (útil a la hora de generar código).
- Dirección de memoria en la que se está ensamblando en cada momento.
- Indicador de si la instrucción viene precedida o no por una etiqueta.
- Lugar donde almacenar la lista de datos de una pseudoinstrucción `DATA` como lista de palabras de 16 bits (útil a la hora de generar código).
- Regla que se está analizando.
- *Token* que se está analizando.

En el segundo bloque es donde se definen las reglas de la gramática vista en el capítulo dedicado al análisis. Es una mera traducción. Por ejemplo, las producciones del axioma:

S → λ
| <EOL> S
| LÍNEA S

Se traducen en:

```
s :
          | EOL s
          | línea s
```

Incorporar al analizador sintáctico las acciones semánticas tampoco requiere demasiado esfuerzo creativo, ya que se definieron con detalle durante el análisis. Básicamente son de dos tipos. El primer tipo consiste en copiar el valor de los atributos de los *tokens* a los de los no terminales. Para ello, *Bison* permite hacer referencia a los símbolos de la gramática mediante la notación `$n`, donde `n` es la posición que ocupa el símbolo en la producción, siendo `$$` el símbolo de la parte izquierda de la regla. Por ejemplo, la siguiente regla:

EXPRESIÓN → EXPRESIÓN$_1$ <SUMA> EXPRESIÓN2
{EXPRESIÓN.valor = EXPRESIÓN$_1$.valor+EXPRESIÓN2.valor;}

Se correspondería con:

```
expresión : expresión SUMA expresión_2
            {$$=$1+$3;}
```

El segundo tipo consiste en comparar el valor de los atributos con ciertas condiciones, y ejecutar determinadas acciones según sea el caso. Por ejemplo, la segunda regla de la gramática:

```
LÍNEA → <IDENTIFICADOR> <FIN_ETIQ>
       {SI
         ExisteEtiqueta(IDENTIFICADOR.texto)
        ENTONCES
         EnviarError(ETIQUETA_DUPLICADA,línea,IDENTIFICADOR.texto);
        SI NO
         NuevaEtiqueta(ETIQUETA.texto,dirmem);}
            LÍNEA_VACÍA RESTO
|           RESTO
```

se expresaría de la siguiente forma:

```
línea : IDENTIFICADOR FIN_ETIQ
        {if(ExisteEtiqueta($1)
         {
             EnviarError(ERR_ETIQUETA_DUPLICADA,nlin+1,$1);
         }
         else
         {
             NuevaEtiqueta($1,dirmem);
         }
        }
        línea_vacía resto
      | resto
```

Si no se ha producido ningún error en el análisis, se tendrá en la variable `instruccion` la instrucción leída en el código fuente, dividida en los siguientes campos:

- Código de operación.
- Modo de direccionamiento del primer operando.
- Primer operando.
- Modo de direccionamiento del segundo operando.
- Segundo operando.

El analizador debe comprobar que la instrucción es correcta, esto es, que el número de operandos y sus modos de direccionamiento están permitidos para dicho código de operación. En ese caso, ya se dispone de la información suficiente para saber cuánto espacio ocupa la instrucción en la memoria, e incluso se pueda codificar la instrucción completa, si es que ninguno de los operandos es una referencia adelantada, por lo que se procederá a escribir en la memoria con los datos que se conozcan. Resolver las referencias adelantadas será tarea del generador de código.

En caso de que la instrucción sea errónea, se invocará al gestor de errores con el error correspondiente, y ya no se generará código ni para esta instrucción ni para las siguientes.

La regla RESTO es importante a la hora de detectar los errores semánticos, ya que es donde se ubica el *token* especial `<ERROR>`, que es generado automáticamente por *Bison* cuando detecta un error en el análisis. La detección y recuperación de errores se realiza a partir de la regla que se estaba analizando, y el *token* dentro de la misma que produjo el error, de esta forma:

```
resto : instrucción
      | pseudoinstrucción
      | error EOL
```

```
{
    switch(numregla)
    {
        case 11 :
            if(numtoken==2)
            {
            //se ha leído el token <MNEMÓNICO0>, se espera un fin
            //de línea
            InformarError(ERR_SE_ESPERABA_EOL,nlin,
                          ConsultarToken(2));
             }
             break;
         case 12 :
             if(numtoken==2)
             {
                 //se ha leído el token <MNEMÓNICO1>
                 //se espera un token de tipo OPERANDO
                 if(strcmp(ConsultarToken(2),"<EOL>")==0)
                 {
                     //Se ha leído un fin de línea
                      //antes de tiempo
                      InformarError(ERR_SE_ESPERABA_OP1,
                                    nlin,"<EOL>");
                 }
                 else
                 {
                     //El token leído no es de tipo operando
                      InformarError(ERR_OP1_INCORRECTO,nlin,
                                    ConsultarToken(2));
                 }
             }
             else if(numtoken==3)
             {
                 //Falta por leer el token fin de línea
                 InformarError(ERR_SE_ESPERABA_EOL,nlin,
                               ConsultarToken(3));
             }
             break;
         case 13 :
             if(numtoken==2)
             {
                  //Se ha leído el token <MNEMÓNICO2>
                  //Se espera un token de tipo OPERANDO
                 if(strcmp(ConsultarToken(2),"<EOL>")==0)
                 {
                     //Se ha leído un fin de línea
                     //antes de tiempo
                    InformarError(ERR_SE_ESPERABA_OP1,
                                  nlin,"<EOL>");
                 }
                 else
                 {
                     //El token leído no es de tipo OPERANDO
                     InformarError(ERR_OP1_INCORRECTO,nlin,
                                   ConsultarToken(2));
                 }
             }
             else if(numtoken==3)
             {
                 //No se ha leído una coma
                 InformarError(ERR_SE_ESPERABA_SEPARADOR,nlin,
```

```
                        ConsultarToken(3));
            }
            else if(numtoken==4)
            {
                //Se espera un token de tipo OPERANDO
                if(strcmp(ConsultarToken(4),"<EOL>")==0)
                {
                    //Se ha leído un fin de línea
                    //antes de tiempo
                    InformarError(ERR_SE_ESPERABA_OP2,
                                  nlin,"<EOL>");
                }
                else
                {
                    //El token leído no es de tipo OPERANDO
                    InformarError(ERR_OP2_INCORRECTO,nlin,
                                  ConsultarToken(4));
                }
            }
            else if(numtoken==5)
            {
                //Falta por leer el token fin de línea
                InformarError(ERR_SE_ESPERABA_EOL,
                              nlin,ConsultarToken(5));
            }
            break;
        case 26 :
            if(numtoken==2)
            {
                //Se espera obtener una EXPRESIÓN
                if(strcmp(ConsultarToken(0),"<EOL>")==0)
                {
                    //Se ha leído un fin de línea
                    //antes de tiempo
                    InformarError(ERR_EXPRESION_ERRONEA,
                                  nlin-1,"<EOL>");
                }
                else
                {
                    //No se ha leído una EXPRESIÓN correcta
                    InformarError(ERR_EXPRESION_ERRONEA,
                                  nlin,"");
                }
            }
            else if(numtoken==3)
            {
                //Falta por leer el token fin de línea
                InformarError(ERR_SE_ESPERABA_EOL,
                              nlin,ConsultarToken(0));
            }
        case 27 :
            if(numtoken==2)
            {
                if(strcmp(ConsultarToken(0),"<EOL>")==0)
                {
                    //Se ha leído un fin de línea
                    //antes de tiempo
                    InformarError(ERR_EXPRESION_ERRONEA,
                                  nlin-1,"<EOL>");
                }
```

```
            else
            {
                //No se ha leído una EXPRESIÓN correcta
                InformarError(ERR_EXPRESION_ERRONEA,
                              nlin,"");
            }
        }
        else if(numtoken==3)
        {
            //Falta por leer el token fin de línea
            InformarError(ERR_SE_ESPERABA_EOL,
                          nlin,ConsultarToken(0));
        }
    case 28 :
        if(numtoken==2)
        {
            if(strcmp(ConsultarToken(0),"<EOL>")==0)
            {
                //Se ha leído un fin de línea
                //antes de tiempo
                InformarError(ERR_EXPRESION_ERRONEA,
                              nlin-1,"<EOL>");
            }
            else
            {
                 //No se ha leído una EXPRESIÓN correcta
                InformarError(ERR_EXPRESION_ERRONEA,
                              nlin,"");
            }
        }
        else if(numtoken==3)
        {
            //Falta por leer el token fin de línea
            InformarError(ERR_SE_ESPERABA_EOL,
                          nlin,ConsultarToken(0));
        }
    case 29 :
        if(numtoken==2)
        {
            if(strcmp(ConsultarToken(0),"<EOL>")==0)
            {
                //Se ha leído un fin de línea
                //antes de tiempo
                InformarError(ERR_EXPRESION_ERRONEA,
                              nlin-1,"<EOL>");
            }
            else
            {
                //No se ha leído una EXPRESIÓN correcta
                InformarError(ERR_EXPRESION_ERRONEA,
                              nlin,"");
            }
        }
        else if(numtoken==3)
        {
            //Falta por leer el token fin de línea
            InformarError(ERR_SE_ESPERABA_EOL,
                          nlin,ConsultarToken(0));
        }
    case 30 :
        if(numtoken==2)
```

```
                {
                    //Falta por leer el token fin de línea
                    InformarError(ERR_SE_ESPERABA_EOL,
                                  nlin,ConsultarToken(0));
                }
            default :
                //La instrucción introducida no es correcta
                InformarError(ERR_INSTRUCCION_NO_RECONOCIDA,nlin,
                              ConsultarToken(1));
        }
        generarcodigo=0;
        yyerrok;
    }
```

La variable `numregla` contiene el número de regla que se está analizando. La variable `numtoken` contiene el número de *token* que se espera encontrar a la entrada. En esta parte de la detección de errores que se está comentando, las reglas implicadas son las relativas a instrucciones y pseudoinstrucciones:

- 11. INSTRUCCIÓN → `<MNEMÓNICO0>` `<EOL>`
- 12. INSTRUCCIÓN → `<MNEMÓNICO1>` OPERANDO `<EOL>`
- 13. INSTRUCCIÓN → `<MNEMÓNICO2>` OPERANDO `<SEPARADOR>` OPERANDO `<EOL>`
- 26. PSEUDOINSTRUCCIÓN → `<ORG>` EXPRESIÓN `<EOL>`
- 27. PSEUDOINSTRUCCIÓN → `<RES>` EXPRESIÓN `<EOL>`
- 28. PSEUDOINSTRUCCIÓN → `<DATA>` LISTA_DATOS `<EOL>`
- 29. PSEUDOINSTRUCCIÓN → `<EQU>` EXPRESIÓN `<EOL>`
- 30. PSEUDOINSTRUCCIÓN → `<END>` `<EOL>`

De esta manera se puede detectar cuál es el *token* que se esperaba cuando se produjo el error y aportar al usuario una información lo más detallada posible.

Llegados a este punto, ya casi está todo el código generado. Lo único que resta, en caso de que no se haya producido ningún error hasta el momento, consiste en resolver las referencias adelantadas que no se pudieron completar durante el análisis sintáctico y semántico. Para ello, irá recorriendo la tabla de configuración e irá rellenando los huecos en el código máquina generado, o bien informará de los posibles errores que pudieran darse. Una vez hecho esto, se habrá completado el proceso de ensamblado con uno de los dos resultados posibles: el código máquina correspondiente al programa fuente o bien una lista de errores encontrados en el mismo.

El procedimiento es el siguiente:

- Para cada entrada de referencias adelantadas:
  - Leer la etiqueta.
  - Buscar el valor de la etiqueta en la tabla de etiquetas.
  - Si el valor está definido y es correcto, actualizar el código máquina.
  - Si no es correcto o no está definido, informar del error.

3.4.3.3. Tabla de Etiquetas

La tabla de etiquetas es una región de memoria donde se almacenan estructuras como la de la Figura 3.4.1. Cada una de las estructuras representa una entrada de la tabla y se define de la siguiente manera:

```
typedef struct EntradaTablaEtiquetas{
        char *etiqueta;
        int valor;};
```

La memoria ocupada por la tabla se ubicará dinámicamente. Sin embargo, no se va a reservar memoria cada vez que se introduzca una nueva entrada, sino que se irá reservando en bloques de 4096 entradas. De esta forma, se descarga el trabajo del gestor de memoria, aumentando el rendimiento. Se ha elegido este tamaño de bloque buscando un compromiso entre el tamaño inicial de la tabla y el número de llamadas al gestor de memoria para reservar nuevos recursos.

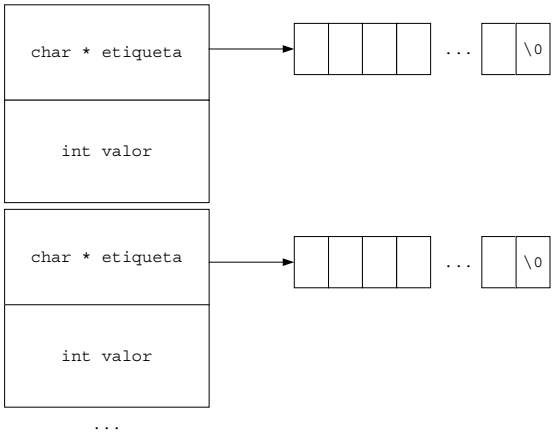


Figura 3.4.1. Tabla de etiquetas

3.4.3.4. Tabla de Referencias Adelantadas

La tabla de referencias adelantadas se construye de la misma forma que la tabla de etiquetas, tal y como se aprecia en la Figura 3.4.2. Esta es la estructura de cada una de sus entradas:

```
typedef struct EntradaTablaConfiguracion{
        char *etiqueta;
        int posicion;
        int desplazamiento;
        int mododireccionamiento;
        int linea;};
```

Análogamente al caso anterior, la memoria se ubica dinámicamente, solicitándose un nuevo espacio de memoria para 4096 entradas cuando sea necesario.

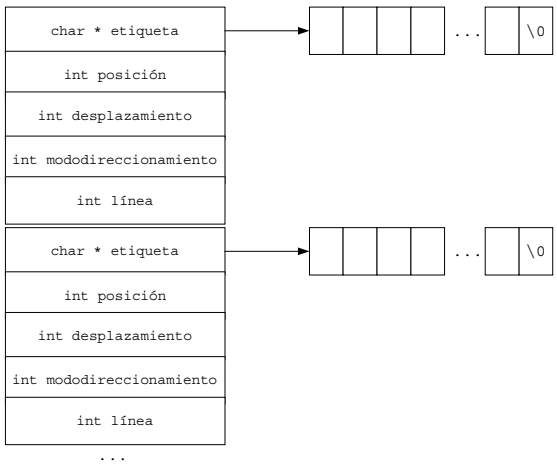


Figura 3.4.2. Tabla de referencias adelantadas

3.4.3.5. Gestor de Errores

La lista de errores es una lista enlazada que se va construyendo dinámicamente según se introducen nuevos errores, según se observa en la Figura 3.4.3. Cada elemento es una estructura definida de la siguiente manera:

```
typedef struct Error{
        int codigo;
        char *descripcion;
        int linea;
        char *token;
        struct Error *siguiente;};
```

En este caso no se solicita memoria por bloques, sino que al añadir cada nuevo error se reserva espacio para la nueva estructura y se enlaza con el que hasta ese momento era el último error de la lista. El valor del campo siguiente del último error de la lista valdrá NULL. Es de esperar que los programas no contengan un número de errores excesivo (como mucho contendrían uno por línea de código).

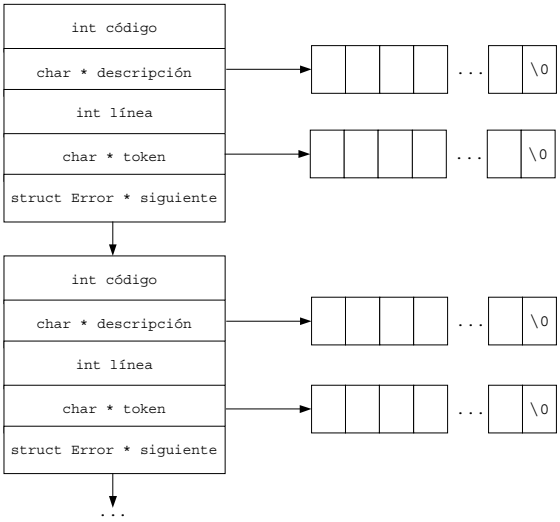


Figura 3.4.3. Lista de errores

Ya que en la interfaz de usuario hay definida una función para visualizar el contenido de la lista de errores, se va a añadir una función que permita grabar el contenido de la lista completa en un fichero, llamado `errores.tmp`.

Función	**VolcarFicheroErrores**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Vuelca el contenido de la lista de errores en un fichero llamado `errores.tmp`. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si hubo algún problema al crear o escribir el fichero.

Y ya que desde la interfaz de consola se va a mostrar el contenido de la tabla de errores por pantalla (o lo que es lo mismo, el contenido del fichero temporal `errores.tmp`), se añade una función que permita leer el contenido de dicho fichero y mostrarlo por la consola.

Función	**ListarErrores**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Abre el fichero `errores.tmp` y lo muestra por consola. Devuelve 0 como valor de retorno.
Errores	Devuelve –1 como valor de retorno si hubo algún problema al leer el fichero.

3.4.4. Implementación del Simulador

Partiendo de todo lo visto en los apartados de análisis y diseño se puede llegar a codificar la parte del simulador sin mayor dificultad. No obstante, hay que hacer mención a dos aspectos.

El primero es que se crean dos clases de objetos nuevas para facilitar la programación: una para manejar cadenas de caracteres (`CCadena`) y otra para manejar enteros de 16 bits (`CEntero16b`). La primera de ellas ya existe en la biblioteca de clases de *Borland C++ Builder*, pero es necesario crearla para la versión *Linux*, así que se le dará uso en todas las versiones. La segunda se introduce por claridad en la programación, ya que así se evita tener que hacer comprobaciones sobre el rango de los enteros manejados en toda la aplicación.

El segundo aspecto versa sobre la problemática que plantea la necesidad de que la ejecución de la simulación no bloquee la aplicación, ya que debe permitirse que el usuario la detenga en cualquier momento. Este problema se soluciona de distintas formas en las distintas versiones de la herramienta. Más adelante, en el apartado 3.4.6, se verá cómo se ha resuelto el problema.

Clase CCadena

Se trata de una clase que encapsula el tratamiento de cadenas de caracteres. De esta forma se evita tener que trabajar con punteros `char *`, lo que resulta mucho más engorroso y propenso a errores.

La implementación no es complicada: se basa en las funciones de biblioteca de *C* para manejo de cadenas de caracteres (de tipo `char *`), como pueden ser `strcpy`, `strcat` y `strcmp`. La ventaja es que libera al usuario (en este caso al programador del resto del

sistema) de la clase de toda la gestión de memoria, reservar espacio para el resultado de las concatenaciones, calcular las longitudes, etc.

CCadena
- char *texto - int longitud
+ CCadena(void) + CCadena(char *cadena) + CCadena(const CCadena &cadena) + ~CCadena() + CCadena &operator=(const CCadena &cad) + friend CCadena operator+(const CCadena &cad1,const CCadena &cad2) + friend CCadena operator+(const CCadena &cad1,const char *cad2) + friend CCadena operator+(const char *cad1,const CCadena &cad2) + friend CCadena operator+(const CCadena &cad1,const char cad2) + friend CCadena operator+(const char cad1,const CCadena &cad2) + friend ostream &operator<<(ostream &os,const CCadena &cadena) + char operator[](int índice) + int Longitud(void) + char *Cadena(void) + int Comparar(CCadena cad2) + int Comparar(char *cad2)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
texto	Es la zona de memoria gestionada internamente donde se guarda la cadena de caracteres a la que se corresponde el objeto.
longitud	Indica el tamaño de la cadena de caracteres, sin contar el carácter fin de cadena.

Método	**CCadena**
Parámetros	`void`
Salida	Ninguna
Comportamiento	Es el constructor por defecto. Crea un objeto `CCadena` correspondiente a una cadena vacía, de longitud cero (`texto = ""` y `longitud = 0`).
Errores	Ninguno.

Método	**CCadena**
Parámetros	`char *`cadena
Salida	Ninguna
Comportamiento	Es otro constructor de la clase. Crea un objeto `CCadena` a partir de una cadena de caracteres del tipo de las empleadas en lenguaje *C*, reservando espacio dinámicamente para el atributo `texto`.
Errores	Ninguno.

Método	**CCadena**
Parámetros	`const CCadena &`cadena
Salida	Ninguna
Comportamiento	Es el constructor de copia de la clase. Se emplea para crear un nuevo objeto que es copia del que se pasa como parámetro (`cadena`).
Errores	Ninguno.

Método	**~CCadena**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase. Libera el espacio ocupado por el atributo `texto`.
Errores	Ninguno.

Método	**operator=**
Parámetros	`const CCadena &`cadena
Salida	`CCadena &`
Comportamiento	Es el operador de asignación para la clase `CCadena`.
Errores	Ninguno.

Método	**operator+**
Parámetros	`const CCadena &`cad1 `const CCadena &`cad2
Salida	`CCadena`
Comportamiento	Es el operador de concatenación para objetos de la clase `CCadena`.
Errores	Ninguno.

Método	**operator+**
Parámetros	`const CCadena &`cad1 `const char *`cad2
Salida	`CCadena`
Comportamiento	Es el operador de concatenación entre un objeto de la clase `CCadena` y una cadena de caracteres del tipo de las empleadas en lenguaje *C*.
Errores	Ninguno.

Método	**operator+**
Parámetros	`const char *`cad1 `const CCadena &`cad2
Salida	`CCadena`
Comportamiento	Es el operador de concatenación entre una cadena de caracteres del tipo de las empleadas en lenguaje *C* y un objeto de la clase `CCadena`.
Errores	Ninguno.

Método	**operator+**
Parámetros	`const CCadena &`cad1 `const char` cad2
Salida	`CCadena`
Comportamiento	Es el operador de concatenación entre un objeto de la clase `CCadena` y un carácter.
Errores	Ninguno.

Método	**operator+**
Parámetros	`const char` cad1 `const CCadena &`cad2
Salida	Ninguna
Comportamiento	Es el operador de concatenación entre un carácter y un objeto de la clase `CCadena`.
Errores	Ninguno.

Todos los operadores de concatenación generan un objeto nuevo resultado de la operación. Su comportamiento es análogo al de la función de biblioteca de *C* `strcat`.

Método	**operator<<**
Parámetros	`ostream &`os `const CCadena &`cadena
Salida	`ostream &`
Comportamiento	Es el operador de inserción en un *stream* de salida para un objeto de la clase `CCadena`. Su comportamiento es análogo al operador de inserción de *C++* para el tipo de datos `char *`.
Errores	Ninguno.

Método	**operator[]**
Parámetros	`int` índice
Salida	`char`
Comportamiento	Es el operador de indexación para un objeto de la clase `CCadena`. Devuelve el carácter contenido en la posición de la cadena indicada por el índice.
Errores	Si el índice es incorrecto, devuelve el carácter nulo.

Método	**Longitud**
Parámetros	`void`
Salida	`int`
Comportamiento	Devuelve el valor del atributo `longitud`.
Errores	Ninguno.

Método	**Cadena**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve el valor del atributo `texto`, que es un puntero a caracteres acabados por un carácter nulo.
Errores	Ninguno.

En vez de implementar operadores de comparación (`<`,`>`,`==`...), se ha optado por definir un método `Comparar`, ya que sólo interesa comparar si dos cadenas son iguales o distintas, y es más clara la notación de método que la de operadores.

Método	**Comparar**
Parámetros	`CCadena` cadena
Salida	`int`
Comportamiento	Es la función de comparación entre dos objetos de la clase `CCadena`. Su comportamiento es análogo al de la función de biblioteca de *C* `strcmp`. Dicha función devuelve la relación lexicográfica entre ambas cadenas, esto es, <0 si la primera es "menor" que la segunda, 0 si ambas son idénticas y >0 si la primera es "mayor" que la segunda.
Errores	Ninguno.

Método	**Comparar**
Parámetros	`char` *cadena
Salida	`int`
Comportamiento	Es la función de comparación entre un objeto de la clase `CCadena` y una cadena de caracteres del tipo de las usadas en lenguaje *C*. Su comportamiento es análogo al de la función de biblioteca de *C* `strcmp`. Dicha función devuelve la relación lexicográfica entre ambas cadenas, esto es, <0 si la primera es "menor" que la segunda, 0 si ambas son idénticas y >0 si la primera es "mayor" que la segunda.
Errores	Ninguno.

Clase CEntero16b

Se trata de una clase que encapsula el tratamiento de números enteros de 16 bits. Permite efectuar operaciones aritméticas y lógicas, pudiendo visualizarse el resultado en base decimal (con o sin signo) y hexadecimal. También calcula el contenido de los biestables de estado para el valor contenido en el objeto. La principal ventaja que aporta es que, al trabajar continuamente con enteros de 16 bits, se pueden omitir en el código todas las comprobaciones de rango, y se tienen precalculados los valores de los biestables de estado para los resultados de todas las operaciones entre enteros de 16 bits.

En el caso de los operadores de autoincremento y autodecremento sólo se implementa la forma postfija, ya que la prefija no es necesaria.

CEntero16b
- int valor - int valor_Z - int valor_S - int valor_V - int valor_C - int valor_P
+ CEntero16b(void) + CEntero16b(int núm) + CEntero16b(const char *cadena_entero) + CEntero16b(const CEntero16b &entero) + ~CEntero16b() + CEntero16b operator++ (int) + CEntero16b operator-- (int) + CEntero16b operator+ (const CEntero16b &e+ nt2) + CEntero16b operator- (const CEntero16b &ent2) + CEntero16b operator- (void) + CEntero16b operator* (const CEntero16b &ent2) + CEntero16b operator/ (const CEntero16b &ent2) + CEntero16b operator% (const CEntero16b &ent2) + CEntero16b operator& (const CEntero16b &ent2) + CEntero16b operator| (const CEntero16b &ent2) + CEntero16b operator^ (const CEntero16b &ent2) + CEntero16b operator~ (void) + char *Decimal(void) + char *DecimalCorto(void) + char *DecimalFormateado(void) + char *DecimalSinSigno(void) + char *DecimalSinSignoFormateado(void) + char *Hexadecimal(void) + char *HexadecimalCorto(void) + int Valor(void) + int Z(void) + int S(void) + int V(void) + int C(void) + int P(void)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
valor	Almacena el valor entero del objeto.
valor_Z	Almacena el valor correspondiente al biestable Z.
valor_S	Almacena el valor correspondiente al biestable S.
valor_V	Almacena el valor correspondiente al biestable V.
valor_C	Almacena el valor correspondiente al biestable C.
valor_P	Almacena el valor correspondiente al biestable P.

Método	**CEntero16b**
Parámetros	`void`
Salida	Ninguna
Comportamiento	Es el constructor por defecto de la clase. Inicializa todos los valores de los atributos a cero.
Errores	Ninguno

Método	**CEntero16b**
Parámetros	`int num`
Salida	Ninguna
Comportamiento	Es otro constructor de la clase. Almacena en el atributo valor el contenido del parámetro `num`. Calcula el valor correspondiente para los atributos `Z`, `S` y `P`. Los atributos `V` y `C` toman el valor 0.
Errores	Si `num` está fuera del rango de los enteros de 16 bits, lanza una excepción *"Entero fuera de rango"*.

Método	**CEntero16b**
Parámetros	`const char * cadena_entero`
Salida	Ninguna
Comportamiento	Es otro constructor de la clase. Calcula el valor del entero a partir de una cadena de caracteres. La cadena puede contener un número en formato decimal o hexadecimal (comenzando por los caracteres `0x`). Almacena en el atributo `valor` el resultado de calcular el entero a partir de la cadena de entrada `cadena_entero`. Calcula el valor correspondiente para los atributos `Z`, `S` y `P`. Los atributos `V` y `C` toman el valor 0.
Errores	Si `num` está fuera del rango de los enteros de 16 bits, lanza una excepción *"Entero fuera de rango"*.

Método	**CEntero16b**
Parámetros	`const CEntero16b &entero`
Salida	Ninguna
Comportamiento	Es el constructor de copia de la clase. Se emplea para crear un nuevo objeto que es copia del que se pasa como parámetro (`entero`).
Errores	Ninguno.

Método	**CEntero16b**
Parámetros	Ninguno
Salida	Ninguna
Comportamiento	Es el destructor de la clase.
Errores	Ninguno

Método	**operator++**
Parámetros	`int`
Salida	`CEntero16b`

Comportamiento	Incrementa en una unidad el valor del objeto y recalcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Ninguno.

Método	**operator--**
Parámetros	`int`
Salida	`CEntero16b`
Comportamiento	Decrementa en una unidad el valor del objeto y recalcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Ninguno.

Método	**operator+**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de suma entre dos objetos de la clase `Centero16b`. Calcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Ninguno.

Método	**operator-**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de resta entre dos objetos de la clase `Centero16b`. Calcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Ninguno.

Método	**operator-**
Parámetros	`void`
Salida	`Centero16b`
Comportamiento	Efectua la operación de cambio de signo del objeto de la clase `Centero16b`. Calcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Ninguno.

Método	**operator***
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de multiplicación entre dos objetos de la clase `Centero16b`. Calcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Ninguno.

Método	**operator/**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de división entre dos objetos de la clase `Centero16b`. Calcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Si el segundo objeto (ent2) tiene el valor 0, se lanza una excepción de división por cero, que será recogida por el objeto de la clase `CProcesador`.

Método	**operator%**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`

Comportamiento	Efectua la operación de módulo entre dos objetos de la clase `Centero16b`. Calcula el valor de los biestables de estado (pudiéndose producir acarreo o desbordamiento).
Errores	Si el segundo objeto (ent2) tiene el valor 0, se lanza una excepción de división por cero, que será recogida por el objeto de la clase `CProcesador`.

Método	**operator&**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de y lógico entre dos objetos de la clase `Centero16b`. No modifica el valor de los biestables de estado.
Errores	Ninguno.

Método	**operator|**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de o lógico entre dos objetos de la clase `Centero16b`. No modifica el valor de los biestables de estado.
Errores	Ninguno.

Método	**operator^**
Parámetros	`const CEntero16b &ent2`
Salida	`Centero16b`
Comportamiento	Efectua la operación de o exclusivo entre dos objetos de la clase `Centero16b`. No modifica el valor de los biestables de estado.
Errores	Ninguno.

Método	**operator~**
Parámetros	`void`
Salida	`Centero16b`
Comportamiento	Efectua la operación de negación lógica del objeto de la clase `Centero16b`. No modifica el valor de los biestables de estado.
Errores	Ninguno.

Método	**Decimal**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve una cadena de caracteres representando el valor entero almacenado en el objeto en formato decimal de 16 bits con signo (en el intervalo de –32768 a 32767).
Errores	Ninguno.

Método	**DecimalCorto**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve una cadena de caracteres representando el valor entero almacenado en el objeto en formato decimal de 8 bits (en el intervalo de –128 a 127).
Errores	Ninguno.

Método	**DecimalFormateado**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve una cadena de 6 caracteres representando el valor entero almacenado en el objeto en formato decimal de 16 bits con signo (en el intervalo de –32768 a 32767), rellenando con espacios por la izquierda si es preciso.
Errores	Ninguno.

Método	**DecimalSinSigno**
Parámetros	void
Salida	char *
Comportamiento	Devuelve una cadena de caracteres representando el valor entero almacenado en el objeto en formato decimal de 16 bits (en el intervalo de 0 a 65535).
Errores	Ninguno.

Método	**DecimalSinSignoFormateado**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve una cadena de 6 caracteres representando el valor entero almacenado en el objeto en formato decimal de 16 bits (en el intervalo de 0 a 65535), rellenando con espacios por la izquierda si es preciso.
Errores	Ninguno.

Método	**Hexadecimal**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve una cadena de caracteres representando el valor entero almacenado en el objeto en formato hexadecimal de 16 bits (en el intervalo de 0x0000 a 0xFFFF).
Errores	Ninguno.

Método	**HexadecimalCorto**
Parámetros	`void`
Salida	`char *`
Comportamiento	Devuelve una cadena de caracteres representando el valor entero almacenado en el objeto en formato hexadecimal de 8 bits (en el intervalo de 0x00 a 0xFF).
Errores	Ninguno.

Método	**Valor**
Parámetros	`void`
Salida	`int`
Comportamiento	Devuelve el contenido del atributo `valor`.
Errores	Ninguno.

Método	**Z**
Parámetros	`void`
Salida	`int`
Comportamiento	Devuelve el contenido del atributo `Z`.
Errores	Ninguno.

Método	**S**
Parámetros	`void`
Salida	`int`
Comportamiento	Devuelve el contenido del atributo `S`.
Errores	Ninguno.

Método	**V**
Parámetros	`void`
Salida	`int`
Comportamiento	Devuelve el contenido del atributo `V`.
Errores	Ninguno.

Método	**C**
Parámetros	`void`
Salida	`int`

Comportamiento	Devuelve el contenido del atributo `C`.
Errores	Ninguno.

Método	**P**
Parámetros	`void`
Salida	`int`
Comportamiento	Devuelve el contenido del atributo `P`.
Errores	Ninguno.

3.4.5. Entrada/Salida

La clase que modela la entrada/salida de la máquina virtual (`CEntradaSalida`) es distinta en la versión gráfica. Mientras que en la versión para consola, la interacción se produce con la entrada y salida estándar del sistema, en la versión gráfica se produce en una ventana que se crea al efecto.

La interfaz de la clase se mantiene, pero el comportamiento de los métodos `Leer` y `Escribir` es diferente y se detalla en el siguiente apartado, junto con lo visto en el diseño de la clase `TfrmConsolaEjecución`, a la que se encuentran estrechamente ligados.

3.4.6. Simulación

A la hora de efectuar la simulación, se encuentra el problema de que la máquina virtual está aislada del exterior, lo que quiere decir que una vez arrancado el procesador, no se detiene hasta que se encuentre una instrucción de parada o se produzca una excepción. Como herramientas de depuración, se incluyó la posibilidad de ejecutar instrucciones de una en una y ubicar puntos de ruptura en la memoria. Ambas posibilidades son controladas por el procesador virtual, como se ha visto.

La dificultad aparece cuando el usuario desea detener la simulación en un momento arbitrario. En un sistema real, esto se podría implementar fácilmente generando una interrupción, pero el modelo de *ENS2001* no dispone de manejo de interrupciones. Así que habrá que "simularlo" de alguna forma.

La solución adoptada es distinta para las tres versiones de la aplicación y se ha visto influida en gran medida por los recursos de programación que los distintos compiladores usados han ofrecido.

La versión para *Linux* en consola ni siquiera contempla esta posibilidad. El usuario puede abortar la simulación (y de hecho la aplicación completa) de la forma habitual (Ctrl+C). Esto es necesario si el procesador se ha quedado simulando un bucle infinito, pero no permite al usuario seguir con su sesión de trabajo.

La versión para *Win32* en consola hace uso de la función `kbhit`, que es una función de la biblioteca `conio` proporcionada por el compilador de *Borland* (no es *C* estándar, por tanto no es portable a otros sistemas operativos directamente). Esta función devuelve el valor `cierto` si se ha pulsado una tecla y `falso` en caso contrario. Por tanto, el procesador, además de comprobar el resto de condiciones de parada, comprobará el valor de `kbhit` después de simular cada instrucción, y si el usuario ha pulsado la tecla ESCAPE, detendrá la ejecución. Con este mínimo cambio en el código se soluciona el problema.

La versión para *Windows* Gráfica implementa la solución más elegante. El procesador va a ejecutar la simulación en un *thread* aparte, haciendo uso de la biblioteca de *threads* que proporciona el compilador de *Borland* (por tanto, tampoco es portable directamente).

Para ello, se define una nueva clase, `CTProcesador`. Es una clase que hereda de `TThread`. Incluye métodos para leer y escribir, ya que debe canalizar la interacción con la consola de usuario.

Debe redefinir el método `Execute`. Este método contendrá un bucle en el que se efectuará la simulación de cada instrucción y, en cada vuelta, se comprobará si ha ocurrido alguna condición para detener la simulación y si el atributo `Terminated` del *thread* es cierto. De esta forma, desde la interfaz principal de la herramienta se puede detener la simulación simplemente llamando al método `Terminate` del *thread*.

Todas las llamadas a métodos que actualicen el contenido de alguna ventana han de hacerse a través del método especial `Synchronize`. Así se efectuarán las llamadas a los métodos `LeerConsola` y `EscribirConsola` de la ventana principal, que sirven de "interfaz" entre el *thread* y la ventana de consola de ejecución.

Como último comentario, hay que citar que cuando haya que simular una instrucción de entrada/salida, el *thread* necesita dibujar en pantalla la consola. Esto lo hace a través de dos métodos definidos en la ventana principal (`TfrmPrincipal`), llamados `LeerEntrada` y `EscribirSalida`.

CTProcesador
- CConfiguración *configuración - TfrmPrincipal *principal
+ CTProcesador(bool CreateSuspended) + int Escribir(Centero16b entero) + int Escribir(char carácter) + int Escribir(CCadena cadena) + void EscribirConfiguración(CConfiguración *conf)
+ void EscribirPrincipal(TfrmPrincipal *ventana) # Execute() + int Leer(CEntero16b &entero) + int Leer(char &carácter) + int Leer(CCadena &cadena)

A continuación se muestra la descripción detallada de los atributos y métodos que componen la clase:

Atributo	Función
configuración	Almacena, en el momento de crear el objeto de la clase, un puntero al objeto de la clase `CConfiguración` que guarda el estado del simulador y punteros al resto de objetos integrantes del modelo.
principal	Almacena un puntero al objeto de la clase `TfrmPrincipal` que representa la ventana principal de la aplicación.

Método	**CTProcesador**
Parámetros	`bool CreateSuspended`
Salida	Ninguna
Comportamiento	Invoca al constructor de la clase madre `TThread(CreateSuspended)`. Almacena el valor `true` en el atributo `FreeOnTerminate` (heredado de `TThread`), lo que permite liberar recursos cuando la ejecución del hilo finalice.
Errores	Ninguno.

Método	**Escribir**
Parámetros	`CEntero16b entero`
Salida	`int`
Comportamiento	Crea una cadena con el valor indicado por el parámetro `entero`, en formato decimal o hexadecimal según sea la configuración en ese momento (consultando el método `BaseNumérica` del atributo `conf`). Invoca al método `EscribirSalida` de la ventana principal con la cadena calculada anteriormente. Invoca, mediante `Synchronize`, al método `EscribirConsola` de la ventana principal.
Errores	Ninguno.

Método	**Escribir**
Parámetros	`char carácter`
Salida	`int`
Comportamiento	Crea una cadena con el carácter indicado por el parámetro `carácter`. Invoca al método `EscribirSalida` de la ventana principal con la cadena calculada anteriormente. Invoca, mediante `Synchronize`, al método `EscribirConsola` de la ventana principal.
Errores	Ninguno.

Método	**Escribir**
Parámetros	`CCadena cadena`
Salida	`int`
Comportamiento	Crea una cadena con el valor indicado por el parámetro `cadena`, e invoca al método `EscribirSalida` de la ventana principal con dicha cadena. Invoca, mediante `Synchronize`, al método `EscribirConsola` de la ventana principal.
Errores	Ninguno.

Método	**EscribirConfiguración**
Parámetros	`CConfiguración *`conf
Salida	`void`
Comportamiento	Actualiza el valor del atributo `configuración` con el contenido del parámetro `conf`.
Errores	Ninguno.

Método	**EscribirPrincipal**
Parámetros	`TfrmPrincipal *`ventana
Salida	`void`
Comportamiento	Actualiza el valor del atributo `principal` con el contenido del parámetro `ventana`.
Errores	Ninguno.

Método	**Execute**
Parámetros	Ninguno
Salida	`void`

Comportamiento	Invoca, mediante `Synchronize`, al método `ComenzarEjecución` de la ventana principal. Reinicia el banco de registros, si es necesario (el biestable `H` está activo y la opción de configuración también). Mientras el atributo `Terminated` (que obliga la terminación de la ejecución del *thread*) valga `false` y no se produzca ninguna condición de parada, invoca al método `EjecutarPaso` del procesador. Cuando se verifique una condición de parada, se escribe la causa en el registro de fin de ejecución. Si el atributo `Terminated` vale `true`, eso indica que el usuario ha abortado la ejecución, por lo que se genera una excepción de tipo *"Ejecución detenida por el usuario"*. Invoca, mediante `Synchronize`, al método `FinEjecución` de la ventana principal.
Errores	Ninguno.

Método	**Leer**
Parámetros	`Centero16b &entero`
Salida	`int`
Comportamiento	Invoca al método `EscribirTipoEntrada ("Entero")` de la ventana principal. Invoca al método `LeerEntrada (texto)` de la ventana principal, convierte la cadena devuelta en el parámetro `texto` en un valor entero y lo almacena en el parámetro `entero`. Si falla la conversión, almacena el valor `CEntero16b (0)` en el parámetro `entero`. Invoca, mediante `Synchronize`, al método `LeerConsola` de la ventana principal.
Errores	Ninguno.

Método	**Leer**
Parámetros	`char &carácter`
Salida	`int`
Comportamiento	Invoca al método `EscribirTipoEntrada ("Carácter")` de la ventana principal. Invoca al método `LeerEntrada (texto)` de la ventana principal, toma el primer carácter de la cadena devuelta en el parámetro `texto` y lo almacena en el parámetro `carácter`. Si la cadena leída está vacía, almacena el valor `'\n'` en el parámetro `carácter`. Invoca, mediante `Synchronize`, al método `LeerConsola` de la ventana principal.
Errores	Ninguno.

Método	**Leer**
Parámetros	`CCadena &cadena`
Salida	`int`
Comportamiento	Invoca al método `EscribirTipoEntrada ("Cadena")` de la ventana principal. Invoca al método `LeerEntrada (texto)` de la ventana principal, toma la cadena devuelta en el parámetro `texto` y la almacena en el parámetro `cadena`. Invoca, mediante `Synchronize`, al método `LeerConsola` de la ventana principal.
Errores	Ninguno.

Para poder llevar el control de la ejecución, es necesario definir un nuevo método en la clase `CProcesador` que permita ejecutar una única instrucción y comprobar todas las condiciones de parada cada vez, ya que si se usara el método `Ejecutar` no se podría detener la ejecución a voluntad del usuario. Este método se llama `EjecutarPaso`. Su funcionamiento es idéntico al del método `Ejecutar`, pero sólo ejecuta una instrucción.

Método	**EjecutarPaso**
Parámetros	Ninguno
Salida	`Entero`
Comportamiento	Ejecuta la siguiente instrucción a la que apunta el contador de programa. Devuelve un código que indica el resultado de la ejecución.
Errores	Ninguno.

Por otra parte, el procesador será el encargado de recoger las excepciones que se puedan producir durante la ejecución, ya sean causadas por las propias instrucciones (división por cero), como las ocasionadas por falta de memoria a la hora de crear los diferentes objetos.

Para modelar este mecanismo, se emplea la implementación ofrecida por *C++*. De esta forma, los objetos que ejecutan las instrucciones sólo tienen que lanzar una excepción y almacenar el código en el registro de instrucción. El método `Ejecutar`() de la clase `CProcesador` se encarga de recogerlas, detener la ejecución e informar a la interfaz de usuario.

La clase `TfrmPrincipal` también se ve afectada, ya que se introducen los siguientes atributos y métodos:

Atributo	Función
ejecutando	Es un *flag* que indica si el procesador está ejecutando instrucciones o no.
entrada	Almacena una cadena de caracteres leída desde la consola de ejecución.
salida	Almacena una cadena de caracteres que va a ser escrita en la consola de ejecución.
tipoentrada	Almacena una cadena de caracteres que indica el tipo de datos que se espera como entrada del usuario desde la consola de ejecución.

Método	**ComenzarEjecución**
Parámetros	`void`
Salida	`void`
Comportamiento	Deshabilita los botones y opciones del menú que permiten modificar valores en los objetos del simulador (banco de registros, memoria), así como la configuración del mismo. Activa el *flag* ejecutando.
Errores	Ninguno.

Método	**EscribirConsola**
Parámetros	`void`
Salida	`void`
Comportamiento	Sirve como interfaz entre el hilo del procesador y la consola de ejecución. Si la ventana de consola de ejecución no está visible la muestra e invoca al método `MostrarCadena (salida).`
Errores	Ninguno.

Método	**EscribirEntrada**
Parámetros	`String valor`
Salida	`void`

Comportamiento	Modifica el valor del atributo `entrada`.
Errores	Ninguno.

Método	**EscribirSalida**
Parámetros	`String valor`
Salida	`void`
Comportamiento	Modifica el valor del atributo `salida`.
Errores	Ninguno.

Método	**EscribirTipoEntrada**
Parámetros	`String valor`
Salida	`void`
Comportamiento	Modifica el valor del atributo `tipoentrada`.
Errores	Ninguno.

Método	**FinEjecución**
Parámetros	`void`
Salida	`void`
Comportamiento	Habilita los botones y opciones del menú que permiten modificar valores en los objetos del simulador (banco de registros, memoria), así como la configuración del mismo. Desactiva el *flag* ejecutando.
Errores	Ninguno.

Método	**LeerConsola**
Parámetros	`void`
Salida	`void`
Comportamiento	Sirve como interfaz entre el hilo del procesador y la consola de ejecución. Muestra la ventana de consola de ejecución de forma modal (obliga al usuario a introducir un valor o cancelar la ejecución), indicando el tipo de datos que se pretende leer (que está almacenado en el atributo `tipoentrada`) y guarda el valor leído en el atributo entrada. Si el usuario cancela la ejecución, invoca al método `Terminate` del hilo en el que se ejecuta el procesador.
Errores	Ninguno.

Método	**LeerEntrada**
Parámetros	`String &valor`
Salida	`void`
Comportamiento	Devuelve el valor del atributo `entrada`.
Errores	Ninguno.

Método	**LeerSalida**
Parámetros	`String &valor`
Salida	`void`
Comportamiento	Devuelve el valor del atributo `salida`.
Errores	Ninguno.

Método	**LeerTipoEntrada**
Parámetros	`String &valor`
Salida	`void`
Comportamiento	Devuelve el valor del atributo `tipoentrada`.
Errores	Ninguno.

3.5. Plan de Pruebas

En este capítulo se va a efectuar una descripción del proceso de pruebas de la aplicación. Se van a definir los distintos niveles de detalle de las pruebas y se propondrán una serie de casos de prueba que cubran la mayor parte posible de aspectos de la herramienta *ENS2001*.

El proceso de pruebas va desde las pruebas de más bajo nivel (pruebas unitarias), hasta las de más alto nivel (pruebas de sistema). Una vez superadas éstas, se puede decir que el producto está listo para su implantación.

El plan de pruebas se completa con información acerca del resultado de las pruebas. Al final del proceso se habrán pasado la totalidad de los casos. Iterativamente, cada prueba fallida da lugar a una corrección en el código, y así sucesivamente hasta que todos los casos de prueba obtienen un resultado correcto.

En este punto no se puede garantizar la corrección de la aplicación al 100%, pero sí en un gran porcentaje si la batería de pruebas diseñada es lo suficientemente amplia.

En los casos que proceda, se indica depués de cada prueba el fallo o fallos que se detectaron, así como una breve explicación de las causas de los mismos. Además, el código fuente de las pruebas se adjunta en un anexo a este documento.

3.5.1. Pruebas Unitarias

Las pruebas unitarias componen el primer banco de pruebas de la aplicación. Tras la etapa de implementación, se obtiene como resultado una serie de ficheros de código fuente cuya compilación dará como resultado *ENS2001*.

En las pruebas unitarias se va a tomar el código fuente función a función (en el caso de *C*) y clase a clase (en el caso de *C++*) y se van a realizar pruebas exhaustivas para comprobar que se cumple la funcionalidad de cada una de ellas.

Como se ha tendido a la creación de métodos y funciones muy simples, con propósitos muy determinados, en la realización de las pruebas se van a emplear técnicas de caja negra. Esto quiere decir que las pruebas se centrarán en comprobar que la salida de las funciones se corresponde con unos estímulos de entrada concretos, sin entrar a valorar cómo están construidas internamente. No obstante, para depurar errores sí que es posible seguir la ejecución del código paso a paso para ver dónde se produce el fallo.

A la hora de realizar este tipo de pruebas se emplearán todas las opciones que proporciona el compilador para tareas de depuración. También se va a utilizar la herramienta *Borland CodeGuard*, que permiten controlar errores difíciles de detectar por otros medios tales como accesos incorrectos a memoria, uso de variables antes de su inicialización, etc.

En este documento no se van a enumerar la totalidad de las pruebas unitarias realizadas al igual que no se ha seguido el proceso de implementación al detalle. Pero sí que se van a presentar aquéllas que puedan resultar más significativas para ilustrar el proceso que se ha seguido. En concreto se van a mostrar las pruebas unitarias de la clase `CEntero16b` (simulador) y del conjunto de funciones de gestión de memoria (ensamblador).

ENS-PRU-06. CEntero16b.

Se van a probar uno a uno todos los métodos de la clase, que son los definidos en el apartado 3.4.4.

Las pruebas se van a efectuar por grupos. Primeramente se probarán los constructores y los métodos de acceso a los atributos. La clase dispone de 3 constructores y el constructor copia. Se pueden crear objetos de la clase a partir de un entero o bien una cadena de caracteres.

Estímulo	Se crean objetos válidos de la clase, por tanto, con valores iniciales comprendidos entre 0 y 65535. También se crean con el constructor por defecto y el constructor copia.
Respuesta	Se crean los objetos. Puede comprobarse el valor de sus atributos mediante los métodos de acceso.

Estímulo	Se crean objetos no válidos, es decir, con valores iniciales menores que 0 ó mayores que 65535.
Respuesta	Se lanza una excepción `runtime_error`.

Errores detectados y corregidos
El constructor `CEntero16b(const char *valor)` no construye bien los enteros cuando el valor es un número negativo mayor o igual que –32768. Se revisa el constructor y se corrige el error.

Operadores aritméticos y lógicos.

Estímulo	Se crean objetos y se les aplica la operación que se esté probando. Interesan como operandos y como resultado valores límite, como puedan ser 0, 65535, 32767 y 32768.
Respuesta	El resultado de la operación es correcto y, en el caso de los operadores aritméticos, los biestables de estado del resultado se han actualizado también de acuerdo al resultado de cada operación.

Métodos de visualización.

Estímulo	Se crean objetos con diferentes valores y se muestran en los diferentes formatos permitidos por los métodos de la clase.
Respuesta	Se comprueba que el formateo y la visualización son correctos.

ENS-PRU-29. Gestión de Memoria.

Este módulo se compone de cuatro funciones. Se va a crear un programa de prueba que haga uso todas ellas, de la siguiente forma:

- Se inicializa la memoria.

Primeramente se van a probar casos correctos.

Estímulo	Se rellena toda la memoria en un bucle, desde la dirección 0 hasta la 65535, escribiendo como dato el mismo valor que para la dirección (`EscribirMemoria(0,0)`, `EscribirMemoria(1,1)`...). A la vez se escribe en pantalla el resultado de la llamada a la función.
Respuesta	En pantalla debe aparecer el valor escrito (el valor de retorno de la función), que es cero en todos los casos.

Estímulo	Se lee el contenido de toda la memoria desde un bucle, desde la dirección 0 hasta la 65535, y se escribe en pantalla el valor leído.
Respuesta	Se comprueba que los valores concuerdan con los introducidos en el caso anterior.

Estímulo	Se vuelca el contenido de la memoria en un fichero.
Respuesta	Se comprueba que el contenido del fichero concuerda con los valores introducidos en la memoria.

Errores detectados y corregidos
En el fichero todos los datos aparecen con el *byte* más significativo a cero. Hay un error en la función que vuelca los datos a disco, en concreto no está escribiendo correctamente el *byte* alto de cada posición de memoria. Se corrige y se verifica su buen funcionamiento.

A continuación se prueban los casos erróneos.

Estímulo	Se intenta escribir en la memoria en direcciones menores que 0 y mayores que 65535, en concreto son interesantes las posiciones límite –1 y 65536
Respuesta	La llamada a la función devuelve el valor –1.

Estímulo	Se intenta escribir en la memoria en direcciones permitidas valores menores que 0 y mayores que 65535, en concreto son interesantes los valores límite –1 y 65536.
Respuesta	La llamada a la función devuelve el valor –1.

Estímulo	Se intenta leer la memoria de posiciones menores que 0 y mayores que 65535, en concreto son interesantes las posiciones límite –1 y 65536
Respuesta	La llamada a la función devuelve el valor –1.

Estímulo	Se intenta volcar el contenido de la memoria en un archivo que ya exista y esté protegido contra escritura.
Respuesta	La llamada a la función devuelve el valor –1.

3.5.2. Pruebas de Integración

Una vez probadas todas las clases y todas las funciones de manera particular, se procede a efectuar un conjunto de pruebas de más alto nivel. En concreto, se definen dos bancos de pruebas, uno de ensamblado y otro de simulación. Esto es, todavía se van a probar los dos módulos principales de la aplicación por separado, pero cada uno de ellos ya construido de manera completa.

Pruebas de Integración del Ensamblador.

A la hora de probar el ensamblador, interesa centrarse en los siguientes aspectos, que constituirán cada uno de ellos un paquete de pruebas:

- Ensamblado de todas y cada una de las instrucciones, con operandos cualesquiera.
- Ensamblado de todas y cada una de las combinaciones de modos de direccionamiento, con instrucciones representativas (esto es, de uno y dos operandos).
- Ensamblado de todas y cada una de las pseudoinstrucciones del ensamblador.
- Ensamblado con etiquetas, para comprobar que toman los valores correctos.
- Ensamblado de listados erróneos, que contengan al menos un par de ejemplos de cada tipo de error.

ENS-PRI-001. Ensamblado del juego de instrucciones.

Estímulo	Listado con todas las instrucciones del juego de instrucciones. No es necesario incluir todas las combinaciones de modos de direccionamiento posibles, pero sí que las incluidas sean correctas.
Respuesta	Se desensambla la memoria y se comprueba que el código contenido se corresponde con el listado fuente.

Errores detectados y corregidos
En ocasiones, cuando la última línea de código fuente no acaba con un carácter fin de línea, dicha línea no se ensambla. Había un problema con el analizador generado por *Flex*. Para asegurar que no se reproduzca este problema, se añade internamente al código un salto de línea al final del mismo.
El código de operación de la instrucción `WRSTR` está mal. Había un "baile" de cifras en la definición del código de operación de la instrucción.

ENS-PRI-002. Ensamblado de combinaciones de modos de direccionamiento.

Estímulo	Se escoge un conjunto de instrucciones de 1 operando que entre todas cubran todos los posibles modos de direccionamiento (por ejemplo, `PUSH` y `BR`). Se escoge un conjunto de instrucciones de 2 operandos que entre todas cubran todos los posibles modos de direccionamiento (por ejemplo `ADD`).
Respuesta	Se comprueba que los operandos se han ensamblado correctamente.

Errores detectados y corregidos
El modo de direccionamiento relativo a registro índice no se está ensamblando bien: no está leyendo bien el desplazamiento y siempre lo ensambla como desplazamiento cero. Se modifica el código que implementa la acción semántica correspondiente para que lea bien el desplazamiento.

ENS-PRI-003. Ensamblado de pseudoinstrucciones.

Pseudoinstrucción `ORG`.

Estímulo	Se introducen pseudoinstrucciones `ORG` marcando sucesivamente distintos puntos de origen de código, seguidas de pequeños fragmentos de código, comprendidos entre las direcciones de memoria 0 y 65535.
Respuesta	Se comprueba en la memoria que los fragmentos de código se han ubicado en las direcciones indicadas

Pseudoinstrucción `EQU`.

Estímulo	Se introducen varias pseudoinstrucciones `EQU`, precedidas de etiquetas, para darles distintos valores válidos, comprendidos entre 0 y 65535.
Respuesta	Se comprueba que las etiquetas han tomado los valores indicados en el código fuente.

Pseudoinstrucción `END`.

Estímulo	Se introduce una pseudoinstrucción `END`, seguida de código.
Respuesta	Todo el contenido del fichero fuente tras la pseudoinstrucción `END` se ignora.

Pseudoinstrucción `RES`.

Estímulo	Se introducen varias pseudoinstrucciones `RES`, precedidas de etiquetas, y se les dan distintos valores válidos, entre 0 y 65535, pero sin salirse del espacio de memoria.
Respuesta	Se comprueba que se ha reservado el espacio en memoria y las etiquetas han tomado valores correctos.

Errores detectados y corregidos
Se está reservando una posición de memoria menos de las que deberían reservarse. Había un error en dicho cálculo en la parte del ensamblador.

Pseudoinstrucción `DATA`.

Estímulo	Se introducen varias pseudoinstrucciones `DATA`, precedidas de etiquetas, y alternando en la lista de datos enteros y cadenas de caracteres, conteniendo todos los caracteres válidos, con cuidado de no salirse del espacio de memoria (recordando que las cadenas acaban en un carácter nulo).
Respuesta	Se comprueba que los datos se han ubicado correctamente en la memoria y que las etiquetas han tomado valores correctos.

Errores detectados y corregidos
No se está añadiendo el carácter nulo a las cadenas. Se soluciona y se comprueba que funciona correctamente.

ENS-PRI-004. Ensamblado de etiquetas.

Estímulo	Se da valor a una serie de etiquetas, bien precediendo a instrucciones, bien precediendo a pseudoinstrucciones `EQU`, `RES` o `DATA`. Se introducen las etiquetas como operandos en las instrucciones. Al menos una por cada modo de direccionamiento: inmediato, directo a memoria y como desplazamiento en direccionamientos relativos a registros índice y a contador de programa.
Respuesta	Se comprueba que las etiquetas han tomado los valores correctos y se han sustituido adecuadamente en los operandos.

ENS-PRI-005. Generación de errores durante el ensamblado.

Error 01. Modo de direccionamiento del operando 1 erróneo.

Estímulo	Para cada instrucción, de uno o dos operandos, se introduce en un listado fuente una línea con cada modo de direccionamiento no permitido para el operando 1.
Respuesta	Se obtiene el error 01 en el ensamblado en cada una de las instrucciones introducidas.

Error 02. Modo de direccionamiento del operando 2 erróneo.

Estímulo	Para cada instrucción, de dos operandos, se introduce en un listado fuente una línea con cada modo de direccionamiento no permitido para el operando 2.
Respuesta	Se obtiene el error 02 en el ensamblado en cada una de las instrucciones introducidas.

Errores detectados y corregidos
El ensamblador está permitiendo introducir un direccionamiento inmediato en el segundo operando de una instrucción `MOVE`. Se modifica el código que implementa la acción semántica que hace esta comprobación para subsanar el error.

Error 03. Instrucción no reconocida.

Estímulo	Se introducen algunos mnemónicos de instrucciones mal escritos, con errores o inventados.
Respuesta	Se obtiene el error 03 en el ensamblado en cada una de las instrucciones introducidas.

Error 04. Operando 1 incorrecto.

Estímulo	Se introducen instrucciones de al menos un operando y, en el primero operando, se introducen operandos incorrectos, pero que son *tokens* del lenguaje (como por ejemplo enteros, cadenas, etc.).
Respuesta	Se obtiene el error 04 en el ensamblado en cada una de las instrucciones introducidas.

Error 05. Operando 2 incorrecto.

Estímulo	Se introducen instrucciones de al menos dos operandos y, en el segundo operando, se introducen operandos incorrectos, pero que son *tokens* del lenguaje (como por ejemplo enteros, cadenas, etc.).
Respuesta	Se obtiene el error 05 en el ensamblado en cada una de las instrucciones introducidas.

Error 06. Etiqueta duplicada.

Estímulo	Se introducen etiquetas con el mismo identificador repartidas por el código fuente.
Respuesta	Se obtiene el error 06 en el ensamblado cada vez que aparece una etiqueta anteriormente definida.

Error 07. Etiqueta no definida.

Estímulo	Se introducen instrucciones cuyos operandos son etiquetas que no están definidas en el código.
Respuesta	Se obtiene el error 07 en el ensamblado en cada una de las etiquetas no definidas.

Errores detectados y corregidos
No se está produciendo este error al ensamblar y, a la hora de ejecutar, las etiquetas no definidas están tomando el valor 0. Se revisa el gestor de etiquetas para reparar este error.

Error 08. Entrada incorrecta.

Estímulo	Se introducen caracteres no válidos en el código fuente, en cualquier parte de las instrucciones.
Respuesta	Se obtiene el error 08 en el ensamblado en cada una de las instrucciones introducidas.

Error 09. Expresión errónea.

Estímulo	Se construyen expresiones mal formadas, por ejemplo, no balanceando los paréntesis, escribiendo operadores al final o introduciendo cualquier cosa que no sean enteros, operadores o paréntesis.
Respuesta	Se obtiene el error 09 en el ensamblado en cada una de las pseudoinstrucciones en las que aparece una expresión mal formada.

Error 10. ORG define el comienzo fuera de la memoria.

Estímulo	Se introduce como valor de una pseudoinstrucción ORG varias expresiones que exceden de los límites de la memoria (0..65535).
Respuesta	Se obtiene el error 10 en para cada una de las pseudoinstrucciones introducidas.

Error 11. RES reserva espacio fuera de la memoria.

Estímulo	Se introducen pseudoinstrucciones RES que provoquen que la suma de la posición actual de ensamblado más la cantidad de memoria que se debe reservar excedan los límites de la memoria.
Respuesta	Se obtiene el error 11 en el ensamblado en cada una de las pseudoinstrucciones introducidas.

Errores detectados y corregidos
Al reservar espacio fuera de la memoria no se está generando el error correspondiente. Se modifica el ensamblador para subsanar este fallo.

Error 12. DATA ubica datos fuera de la memoria.

Estímulo	Se introducen pseudoinstrucciones DATA con una lista de datos tal que su ubicación en memoria excede del límite superior de la memoria. En otras palabras, la suma de la posición actual de ensamblado más la cantidad de memoria ocupada por los datos de la lista exceden dicho límite.
Respuesta	Se obtiene el error 12 en el ensamblado en cada una de las pseudoinstrucciones introducidas.

Errores detectados y corregidos
Al almacenar datos fuera de la memoria no se está generando el error correspondiente, sino que se están almacenando de nuevo al principio de la memoria (como si fuera circular). Se añade esta comprobación en el ensamblador.

Error 13. Expresión fuera de rango.

Estímulo	Se introducen varias expresiones con sintaxis correcta cuyo resultado excede los límites de representación de enteros en 16 bits. Al menos dos de ellas deben tomar los valores límite (-32769 y 65536).
Respuesta	Se obtiene el error 13 en el ensamblado en cada una de las expresiones introducidas.

Error 14. Nombre de etiqueta reservado.

Estímulo	Se introducen como operandos etiquetas cuyo identificador es un mnemónico de alguna instrucción del juego de instrucciones, indiferentemente en el primer o el segundo operando.
Respuesta	Se obtiene el error 14 en el ensamblado en cada una de las instrucciones introducidas.

Error 15. Entero fuera de rango.

Estímulo	En los operandos con modo de direccionamiento inmediato y directo a memoria, se introducen valores que superan los límites de los enteros de 16 bits. En los modos de direccionamiento relativo a índices se introducen valores que superan los límites de los enteros de 8 bits. Al menos se crea un ejemplo de cada caso con los valores frontera (-32769, 65536 para 16 bits, y –129, 256 para 8 bits).
Respuesta	Se obtiene el error 15 en el ensamblado en cada una de las instrucciones introducidas.

Errores detectados y corregidos
No se están tratando bien los rangos para enteros de 8 bits, ya que se está ignorando la cantidad a partir del noveno bit y no se está notificando el error existente. Se detecta el error en el ensamblador y se corrige.

Error 16. Se esperaba el operando 1.

Estímulo	Se introducen instrucciones de al menos un operando y se introduce un retorno de carro después del mnemónico de la instrucción.
Respuesta	Se obtiene el error 16 en el ensamblado en cada una de las instrucciones introducidas.

Error 17. Se esperaba el operando 2.

Estímulo	Se introducen instrucciones de al menos dos operandos y se introduce un retorno de carro después del carácter separador.
Respuesta	Se obtiene el error 17 en el ensamblado en cada una de las instrucciones introducidas.

Error 18. Se esperaba el carácter de fin de línea.

Estímulo	Se introducen instrucciones de 0, 1 y 2 operandos, y se les añade un operando más (o cualquier cosa) a cada una de ellas.
Respuesta	Se obtiene el error 18 en el ensamblado en cada una de las instrucciones introducidas.

Error 19. Se esperaba el carácter separador.

Estímulo	Se introducen instrucciones de dos operandos pero sin el carácter separador de operandos.
Respuesta	Se obtiene el error 19 en el ensamblado en cada una de las instrucciones introducidas.

Error 20. Lista de datos errónea.

Estímulo	Se introducen pseudoinstrucciones `DATA` seguidas de listas de datos, en las que los datos no se separan por comas o se introducen otro tipo de datos que no sean enteros o cadenas de caracteres entrecomilladas.
Respuesta	Se obtiene el error 20 en el ensamblado en cada una de las pseudoinstrucciones introducidas.

Error 21. Desplazamiento fuera de rango.

Estímulo	Se introducen instrucciones de salto o llamada a subrutina empleando el modo de direccionamiento relativo a contador de programa. El operando será una etiqueta que esté más alejada de la instrucción actual una cantidad de posiciones representable por un entero de 8 bits.
Respuesta	Se obtiene el error 21 en el ensamblado en cada una de las instrucciones introducidas.

Error 22. Error en asignación de memoria.

Estímulo	Se ejecuta el proceso de ensamblado en condiciones de baja disponibilidad de memoria, la cual se puede acentuar introduciendo una gran cantidad de etiquetas y referencias a etiquetas.
Respuesta	Se obtiene el error 22 en el ensamblado, producido por la falta de memoria a la hora de almacenar las estructuras de datos internas del ensamblador. Este error ocurre porque *ENS2001* no puede reservar espacio para almacenar sus estructuras de datos internas.

Pruebas de Integración del Simulador.

Para probar el simulador, se ejecutarán instrucciones y se comprobará que el efecto que producen sea el esperado, ejecutando paso a paso y viendo a través de la interfaz que todo funciona correctamente. En concreto, se van a agrupar las pruebas según las instrucciones implicadas. Por tanto, se podría proponer un esquema como el siguiente:

- Simulación de instrucciones básicas (`NOP` y `HALT`).
- Simulación de instrucciones de transferencia de datos.
- Simulación de instrucciones de manejo de pila.
- Simulación de instrucciones aritméticas.
- Simulación de instrucciones lógicas.
- Simulación de instrucciones de control de flujo de ejecución.
- Simulación de instrucciones de manejo de subrutinas.
- Simulación de instrucciones de entrada/salida.
- Simulación de código que genere todas y cada una de las excepciones definidas.

ENS-PRI-006. Instrucciones básicas.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: `NOP`, `HALT`.

Estímulo	Ejecutar la instrucción `NOP`.
Respuesta	El único cambio que se observa en el simulador es que el valor del contador de programa (`PC`) ha aumentado en una unidad.

Estímulo	Ejecutar la instrucción `HALT`.
Respuesta	Se esperan los siguientes cambios: • El contador de programa ha aumentado en una unidad. • Se ha activado el biestable de fin de ejecución (H).

ENS-PRI-007. Instrucciones de transferencia de datos.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de la instrucción `MOVE`. No es necesario probar todas las combinaciones de modos de direccionamiento, pero sí que todos los permitidos aparezcan en ambos operandos.

Estímulo	Ejecutar instrucciones `MOVE op1, op2`, con las combinaciones de operandos que se muestran en la Tabla 3.5.1.
Respuesta	El dato origen (primer operando) se ha almacenado en el destino (segundo operando).

Operando 1	Operando 2
inmediato - #10	registro - .r1
inmediato - #10	memoria - /1000
inmediato - #10	indirecto - [.r3]
inmediato - #10	relativo a índice ix - #20[.ix]
inmediato - #10	relativo a índice iy - #-10[.iy]
registro - .r4	memoria - /2000
memoria - /3000	memoria - /2000
indirecto - [.r6]	memoria - /2000
relativo a índice ix – #0xff[.ix]	memoria - /2000
relativo a índice iy - #127[.iy]	memoria - /2000

Tabla 3.5.1. Casos de prueba de las instrucciones de transferencia de datos

ENS-PRI-008. Instrucciones de manejo de pila.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: PUSH, POP.

Estímulo	Se ejecutan instrucciones PUSH con todos los modos de direccionamiento permitidos. Se ejecutan instrucciones POP cambiando los modos de direccionamiento. Se repite la prueba para las dos configuraciones de funcionamiento de la pila (creciente y decreciente).
Respuesta	Se comprueba que en cada instrucción PUSH se almacena el dato en memoria y se incrementa (o decrementa) el valor del puntero de pila. Se comprueba que en cada instrucción POP se recupera el dato desde la memoria y se decrementa (o incrementa) el valor del puntero de pila.

Errores detectados y corregidos
No se está comprobando la configuración y la pila siempre crece en direcciones descendentes. Se corrigen las instrucciones PUSH y POP para que efectúen esta comprobación.

ENS-PRI-009. Instrucciones aritméticas.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: ADD, SUB, MUL, DIV, MOD, INC, DEC, NEG, CMP.

Estímulo	Se ejecutan las instrucciones de dos operandos de manera que entre todos los casos se cubran todas las combinaciones permitidas. Se repite para las de un operando. Interesan especialmente aquellas operaciones cuyos operandos y resultado sean los límites de los enteros de 16 bits (-32768, 0, 32767 y 65535) y aquéllas en las que se produzcan condiciones de desbordamiento y acarreo.
Respuesta	Se comprueba que para cada operación se ha almacenado el valor correcto en el registro A y se han modificado adecuadamente los biestables de estado.

Errores detectados y corregidos
No se están actualizando correctamente los biestables de estado de acarreo y desbordamiento. Se modifica el método Aritmética de la clase CInstrucción para subsanar este error.

ENS-PRI-010. Instrucciones lógicas.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: AND, OR, XOR, NOT.

Estímulo	Las instrucciones lógicas operan bit a bit. Por tanto, basta con ejecutar las de dos operandos con los operandos 1100b y 1010b para comprobar que se verifican todos los casos de sus tablas de verdad. Para la instrucción `NOT`, basta ejecutarla con el operando 01.
Respuesta	Se verifica que: • 1100b and 1010b = 1000b • 1100b or 1010b = 1110b • 1100b xor 1010b = 0110b • not 01b = 10b

ENS-PRI-011. Instrucciones de control de flujo de ejecución.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: `BR`, `BZ`, `BNZ`, `BP`, `BN`, `BV`, `BNV`, `BC`, `BNC`, `BE`, `BO`.

Estímulo	Para cada instrucción de salto condicional, ejecutarla dos veces, una en la que se cumpla la condición y otra en la que no.
Respuesta	Se comprueba que los saltos sólo se producen cuando la condición por la que se pregunta es cierta. Lógicamente, el salto incondicional se cumple siempre.

Errores detectados y corregidos
La instrucción `BNC` se está comportando como la instrucción `BNV` (está fallando la consulta al biestable de estado). Se corrige la instrucción `BNC`, en la que se introdujo el fallo al teclear.

ENS-PRI-012. Instrucciones de manejo de subrutinas.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: `CALL`, `RET`.

Estímulo	Se crean subrutinas en varias posiciones de memoria. Las subrutinas van finalizadas por la instrucción `RET`. En diferentes puntos del código, se hacen llamadas a las subrutinas, con la instrucción `CALL`. La prueba se realiza dos veces, una con la pila configurada en modo ascendente y la otra en modo descendente.
Respuesta	Al ejecutar una instrucción `CALL`, se comprueba que se guarda en la pila la dirección correcta de memoria y se salta a la dirección indicada. Al ejecutar una instrucción `RET`, se comprueba que se restaura la dirección de retorno desde la pila y se salta a la misma.

ENS-PRI-013. Instrucciones de entrada/salida.

En ese grupo se incluyen casos de prueba para comprobar el funcionamiento de las siguientes instrucciones: `INCHAR`, `ININT`, `INSTR`, `WRCHAR`, `WRINT`, `WRSTR`.

Estímulo	Se ejecutan parejas de instrucciones lectura/escritura, con el mismo operando en ambas y para todos los modos de direccionamiento permitidos.
Respuesta	Se imprime en pantalla lo mismo que se introdujo previamente.

ENS-PRI-014. Generación de excepciones.

Excepción 01. Instrucción no implementada.

Estímulo	Se ensambla una zona de definición de datos. Se comprueba mediante el desensamblador de la herramienta que se están almacenando códigos de instrucción no implementados y se inicia la ejecución en dichas posiciones de memoria.
Respuesta	Se genera la excepción, se muestra un mensaje indicándolo y se detiene la ejecución.

Excepción 02. División por cero.

Estímulo	Se ejecuta una instrucción `DIV`, en la que el segundo operando vale 0. Se realiza una prueba por cada modo de direccionamiento permitido.
Respuesta	Se genera la excepción, se muestra un mensaje indicándolo y se detiene la ejecución.

Excepción 03. Contador de programa sobrepasa el final de la memoria.

Estímulo	Se ejecuta código de manera que no se produce ningún salto y no se encuentra ninguna instrucción `HALT`. Consecuentemente, el valor del contador de programa se irá incrementando.
Respuesta	Se genera la excepción, se muestra un mensaje indicándolo y se detiene la ejecución.

Errores detectados y corregidos
No se está generando la excepción. El contador de programa, al llegar al final de la memoria, vuelve al principio, como si la memoria fuera circular. Se modifica la clase `CBancoRegistros` para hacer efectiva esta comprobación.

Excepción 04. Contador de programa invade la zona de pila.

Estímulo	Se activa la comprobación de esta excepción en las opciones de configuración. Se hace que el contador de programa salte a la zona ocupada por la pila del sistema.
Respuesta	Se genera la excepción, se muestra un mensaje indicándolo y se detiene la ejecución.

Excepción 05. Puntero de pila invade la zona de código.

Estímulo	Se activa la comprobación de esta excepción en las opciones de configuración. Se hace que la pila crezca tanto como para invadir la zona en la que está almacenada el código.
Respuesta	Se genera la excepción, se muestra un mensaje indicándolo y se detiene la ejecución.

Errores detectados y corregidos
Se ha probado también desactivando la comprobación en la configuración y se sigue generando la excepción, lo cual es incorrecto. Se localiza el error en la clase `CBancoRegistros` y se subsana.

Excepción 06. Ejecución detenida por el usuario.

Estímulo	Mientras se está ejecutando un programa cualquiera, se pulsa la tecla ESCAPE en la versión para consola o el botón de STOP en la versión gráfica.
Respuesta	Se genera la excepción, se muestra un mensaje indicándolo y se detiene la ejecución.

3.5.3. Pruebas de Sistema

Las pruebas de sistema constituyen el último paso antes de liberar el producto. Se trata de pruebas en las que se va a comprobar la funcionalidad completa de la aplicación, desde la perspectiva más cercana al usuario posible.

Se han diseñado cinco programas en ensamblador, que además de servir como eje de las pruebas de sistema, se acompañarán como ejemplos de uso en la distribución de la aplicación. El desarrollo de las pruebas comprende actividades relacionadas con el manejo de la herramienta desde el punto de vista del usuario, como puede ser abrir y guardar ficheros, diferentes ejecuciones, consulta y modificación de los elementos de la arquitectura (memoria y registros), etc. Es decir, a este nivel se supone que tanto el ensamblado como la simulación son correctos, y las pruebas se centran en la integración de ambos módulos con la interfaz, en modo consola o gráfico según la versión de la aplicación.

ENS-PRS-001. Programa cálculo de factorial (`fact.ens`).

Esta prueba está adaptada de uno de los ejemplos incluidos con *ENS96*.

Se trata de un programa que solicita la introducción de un número entero positivo y a continuación calcula el factorial y lo muestra en pantalla. Si se introduce un número negativo, el programa devuelve un mensaje de error (número no válido). Si al calcular el factorial se produce una condición de desbordamiento, el programa devuelve un mensaje de error (número demasiado grande). En resumen, los números válidos a la entrada están en el intervalo entre 0 y 7.

En este programa se trata de probar las llamadas a subrutinas, operadores aritméticos, entrada/salida por consola y bifurcaciones condicionales. Esta versión no utiliza la recursividad.

Para realizar esta prueba, la pila debe estar configurada para crecer hacia direcciones descendentes.

Errores detectados y corregidos
No se estaba tratando adecuadamente el carácter especial `\n` en las cadenas de caracteres. Se corrige el error y se comprueba que el funcionamiento es el deseado.

ENS-PRS-002. Programa manejo de matriz 5x5 (`matriz.ens`).

Esta prueba está adaptada de uno de los ejemplos incluidos con *ENS96*.

Se trata de un programa que permite el manejo de una matriz de 5x5 posiciones. Muestra un menú con las siguientes opciones:

- Introducir un valor en una posición determinada.
- Consultar el valor de una posición determinada.
- Mostrar el contenido de la matriz (las 25 posiciones).
- Abandonar el programa.

A la hora de consultar datos o introducirlos en la matriz, el programa pedirá que se introduzca la fila y la columna deseadas. Si el número introducido no es válido, se repetirá la pregunta hasta que se introduzca uno correcto.

En este programa se trata de probar la entrada/salida por consola y las bifurcaciones condicionales, así como la gestión de espacio en memoria.

Errores detectados y corregidos
Se ha hecho una prueba exhaustiva de funcionamiento y todas las opciones se comportan adecuadamente.

ENS-PRS-003. Programa cálculo de máximo y mínimo (`maxmin.ens`).

Se trata de un programa que permite al usuario introducir una lista de números enteros distintos de 0, que se toma como indicador de fin de lista. A continuación, muestra por pantalla cuál ha sido el mayor y cuál el menor de los introducidos.

Este ejemplo hace uso de la instrucción de comparación para calcular el máximo y el mínimo. Se emplean estos dos criterios:

r = a – b	r>0	r<0
Desbordamiento	a<b	a>b
No desbordamiento	a>b	a<b

En este programa se trata de probar las bifurcaciones condicionales y el uso de los biestables de estado, así como la instrucción de comparación y entrada/salida por consola.

Errores detectados y corregidos
El biestable de desbordamiento no se actualizaba correctamente. Se corrige el error y se comprueba que el funcionamiento es el deseado.

ENS-PRS-004. Programa manejo de cadenas (`cadenas.ens`).

Se trata de un programa que pide al usuario que introduzca dos cadenas de caracteres, tras lo cual las concatena y las muestra por pantalla del derecho y del revés (invirtiendo el orden de los caracteres).

Como no hay forma de anticipar el tamaño de las cadenas introducidas, se reserva un máximo de 500 posiciones para cada una de ellas (lo que implica una longitud máxima de 499 caracteres).

En este programa se trata de probar el manejo de datos en memoria y la entrada/salida por consola.

Errores detectados y corregidos
Se ha hecho una prueba exhaustiva de funcionamiento y todas las opciones se comportan adecuadamente.

ENS-PRS-005. Programa calculadora (`calculadora.ens`).

Se trata de un programa que permite al usuario efectuar todas las operaciones implementadas en la unidad aritmética del simulador. Para ello, muestra un menú de opciones con todas ellas, luego solicita al usuario que introduzca los dos operandos, y por

último muestra el resultado de la operación y si se produjo desbordamiento o acarreo. El usuario podrá repetir el proceso cuantas veces desee hasta que elija la opción de salir.

En este programa se trata de probar las operaciones aritméticas y lógicas, las llamadas a subrutinas y la entrada/salida por consola.

Errores detectados y corregidos
La instrucción `MOD` no controlaba si el divisor era igual a cero. Se corrige el error y se comprueba que el funcionamiento es el deseado.

ENS-PRS-006. Programa cálculo de factorial recursivo (`fact_rec.ens`).

Esta prueba es una adaptación de la versión no recursiva en la que la rutina de cálculo del factorial sí emplea recursividad.

Se trata de un programa que solicita la introducción de un número entero positivo y a continuación calcula el factorial y lo muestra en pantalla. Si se introduce un número negativo, el programa devuelve un mensaje de error (número no válido). Si al calcular el factorial se produce una condición de desbordamiento, el programa devuelve un mensaje de error (número demasiado grande). En resumen, los números válidos a la entrada están en el intervalo entre 0 y 7. Para realizar esta prueba, la pila debe estar configurada para crecer hacia direcciones descendentes.

En este programa se trata de probar las llamadas a subrutinas, entrada/salida por consola y bifurcaciones condicionales, así como la creación de un espacio de marco de pila para la gestión de llamadas a subrutinas y permitir la recursividad.

Errores detectados y corregidos
Se ha hecho una prueba exhaustiva de funcionamiento y todas las opciones se comportan adecuadamente.

3.5.4. Fase de beta-test y pruebas de regresión

Por último, para asegurar aún más la corrección de la aplicación, previamente al lanzamiento definitivo de la versión 1.0 se ha iniciado una fase de *beta-test*. Se ha liberado una versión beta de la aplicación, etiquetada como 0.9, que se pone a disposición de los usuarios. Durante este proceso, se van anotando y resolviendo las incidencias que los usuarios detecten.

A la hora de corregir una incidencia, es posible que los cambios en el código den lugar a nuevos problemas, por tanto, antes de entregar el *software* corregido se deben efectuar pruebas de regresión. En este caso particular, se definen según la Tabla 3.5.2.

Módulo donde se efectúan los cambios	Pruebas de regresión
Módulo ensamblador	• Pruebas unitarias de los programas modificados. • Pruebas de integración del módulo ensamblador. • Pruebas de sistema
Módulo simulador	• Pruebas unitarias de los programas modificados. • Pruebas de integración del módulo simulador. • Pruebas de sistema
Interfaz de usuario	• Pruebas unitarias de los programas modificados. • Pruebas de sistema.

Tabla 3.5.2. Módulos afectados por las pruebas de regresión

La aplicación ha sido utilizada por un número aproximado de 20 grupos hasta la versión 0.9.6 beta en entorno *Windows*. Posteriormente a la fecha de entrega de la práctica de Compiladores de Junio de 2002 se liberó la versión 0.9.6 beta para *Linux*.

Durante la convocatoria de septiembre han usado la herramienta alrededor de 40 grupos, 7 de ellos en la versión consola, y el resto la gráfica para *Windows*, sin haberse registrado ninguna incidencia. La versión disponible hasta esa fecha seguía siendo la 0.9.6 beta.

El 31 de diciembre de 2002 se liberó la versión 0.9.7 beta para todos los sitemas.

Durante la práctica de Compiladores de Febrero de 2003 la versión gráfica para *Windows* fue usada por 9 grupos. Otros 2 grupos emplearon la versión consola para *Windows*. Por último, un grupo se sirvió de la versión para *Linux*. Este grupo manifestó su satisfacción al encontrar una versión para dicho sistema operativo y, además, comentó que no había encontrado ningún tipo de problema durante la realización de la práctica.

A continuación se muestra la relación de incidencias aparecidas hasta la fecha de entrega de este documento para cada una de las versiones liberadas (entre paréntesis se indica la fecha de liberación de cada versión).

ENS-BETA-001 (22.04.2002)

Problema: La herramienta genera un error de protección general en su versión para Windows al invocar al módulo ensamblador.

Módulo afectado: Ensamblador.

Solución: Se ha detectado un problema con el compilador en la invocación a una función de tipo `extern "C"` en el módulo ensamblador. El problema se soluciona modificando las opciones de compilación, dejando modo *Debug* para los fuentes `ens_lex.c` y `ens_tab.c`, y modo *Release* para el resto de fuentes.

ENS-BETA-002 (30.04.2002)

Problema: La herramienta genera un error de protección general al ensamblar los programas fuente `ejem5.ens` y `ejem13.ens`, entregados por uno de los usuarios.

Módulo afectado: Ensamblador.

Solución: Los programas no contienen ningún tipo de detalle especial, salvo su longitud. Se trata del mismo problema que en la incidencia anterior. Al invocar a la función `yyparse()` se produce el error de protección. Se soluciona modificando las opciones de compilación, dejando todo el proyecto en modo *Debug*.

ENS-BETA-003 (07.05.2002)

Problema: El simulador lanza una excepción no definida al ejecutar una instrucción `NOT`.

Módulo afectado: Simulador.

Solución: La instrucción `NOT` está mal implementada y falla cuando el operando es un entero negativo. Se corrige la implementación.

ENS-BETA-004 (15.05.2002)

Problema: El ensamblador se cuelga al procesar una pseudoinstrucción `DATA` con una cadena que contenga el carácter coma ‘,’.

Módulo afectado: Ensamblador.

Solución: Se había tomado la decisión en tiempo de diseño de que las cadenas de caracteres no pudieran contener dicho carácter, no obstante no se informaba del error, y la aplicación se quedaba colgada. Se ha corregido el ensamblador de forma que ahora acepta cadenas de caracteres conteniendo cualquier carácter imprimible.

ENS-BETA-005 (21.05.2002)

Problema: El gestor de errores tiene un fallo en la inserción en la lista de errores, que a veces provoca que la rutina que vuelca el fichero de errores entre en un bucle infinito.

Módulo afectado: Ensamblador.

Solución: Se ha corregido la operación de inserción, de forma que opere adecuadamente. Ahora se efectúa correctamente la inserción ordenada por número de línea en la lista de errores.

Problema: La pseudoinstrucción `RES` devuelve en ocasiones un error de ensamblado aún cuando esté siendo utilizada de forma correcta.

Módulo afectado: Ensamblador.

Solución: Se corrige la implementación de la pseudoinstrucción para que opere de forma adecuada.

ENS-BETA-006 (24.05.2002)

Problema: El comportamiento de la ventana de código fuente al hacer *scroll* no era el adecuado en determinadas circunstancias.

Módulo afectado: Interfaz Gráfica.

Solución: Se revisa la implementación de la ventana de código fuente para que la visualización sea la correcta.

Problema: El comando para leer el contenido de los registros no funciona.

Módulo afectado: Interfaz Consola.

Solución: Se corrige la implementación del comando para que opere de forma adecuada.

Problema: La herramienta no procesa programas fuente de más de 2000 líneas.

Módulo afectado: Ensamblador.

Solución: Se aumenta la capacidad de la pila del ensamblador, de forma que pueda procesar programas mucho más extensos (hasta unas 20000 líneas de código).

ENS-BETA-007 (31.12.2002)

Problema: No se está controlando que en una expresión se produzca una división por cero.

Módulo afectado: Ensamblador.

Solución: Se revisa la implementación del ensamblador de manera que dicha comprobación se haga efectiva, devolviéndose un error de tipo "expresión incorrecta".

Además de corregir los problemas aparecidos, se han ido incorporando algunas pequeñas mejoras a la herramienta, especialmente en su versión gráfica. La más relevante es la posibilidad de conservar el directorio de trabajo entre una sesión y otra.

En febrero de 2003, vista la estabilidad de la última versión beta de la herramienta, se decide lanzar la versión 1.0.

4. Conclusiones

En este capítulo se va a hacer una reflexión sobre lo que ha constituido el proceso de desarrollo de la aplicación, desde su concepción hasta la fase de pruebas. También se van a valorar la consecución de los objetivos pretendidos en los capítulos introductorios y las sensaciones experimentadas durante las distintas fases de desarrollo.

La primera impresión que cabe destacar es la utilidad didáctica de la aplicación. De hecho, durante la fase de *beta-test* ya ha sido empleada por bastantes grupos de la práctica de Compiladores del curso 2001-2002. Han mostrado su interés embarcándose en el riesgo que supone desarrollar y evaluar sus prácticas con una herramienta todavía en fase de prueba. Todo ello demuestra que los alumnos que la han usado consideran que presenta ventajas frente a sus predecesoras, *ASS* y *ENS96*.

El proceso de desarrollo ha sido muy completo, puesto que se han tocado diferentes aspectos de lo que puede llegar a ser la concepción e implementación de un sistema, teniendo que realizar una serie de actividades diversas:

- Desarrollo orientado a objetos. El motor de simulación está íntegramente programado en *C++*.
- Desarrollo estructurado. El ensamblador está diseñado según la estructura general de un compilador.
- Desarrollo basado en herramientas generadoras de código. Se han usado las herramientas *Flex* y *Bison*, que han generado código *C* para el analizador léxico y sintáctico respectivamente.
- Desarrollo de la interfaz gráfica basada en eventos.
- Desarrollo de una parte multihilo para ejecutar el simulador en un hilo aparte. De esta forma se libera la interfaz de usuario, especialmente en caso de bucles infinitos en la simulación.
- Diseño de la interfaz gráfica usando una herramienta de desarrollo visual. Se ha empleado *C++ Builder* de *Borland* para el diseño de la interfaz gráfica.
- Depuración mediante herramientas avanzadas. Se han empleado herramientas de *Borland*, como *Turbo Debugger* o *CodeGuard* para detección y corrección de errores.
- Proceso de pruebas, desde pruebas unitarias hasta pruebas de sistema. Gestión de cambios en el código fuente y políticas de *backup*, de forma simple pero efectiva.
- Período de *beta-test*, con resolución de incidencias "al vuelo". La arquitectura y el diseño de la aplicación han resultado lo suficientemente claros como para poder detectar y corregir los errores prácticamente de un día para otro.

Técnicamente, se han cumplido las expectativas creadas respecto al desarrollo. Se ha construido una interfaz gráfica totalmente nueva que facilita en gran medida el trabajo del usuario, manteniendo otra versión con una interfaz textual que sigue en la línea de sus predecesores, principalmente *ENS96*. Por otro lado, se ha insistido en un desarrollo multiplataforma, aislando el motor de simulación de la interfaz de usuario y empleando funciones estándar de *C++*, lo que permite recompilarlo en cualquier sistema que disponga de compilador de dicho lenguaje

A la hora de valorar las dificultades encontradas durante la realización del proyecto, la parte más ardua ha sido la relativa al ensamblador y, en concreto, a la detección de errores

en el código fuente. Ha resultado complicado establecer un sistema de gestión de errores que dé información útil al usuario, como qué error se produjo, qué se esperaba encontrar en la entrada y dónde se produjo el error.

Otro escollo importante que ha habido que superar ha sido el uso conjunto de código en *C* y *C++*. Se han encontrado bastantes dificultades a la hora de compilar el código, que han sido finalmente superadas. Esta restricción ha venido motivada por el uso de *Flex* y *Bison* para la generación automática de los analizadores. Las versiones que se han utilizado generan código *C*, por lo que se optó por construir la parte correspondiente al ensamblador en ese lenguaje. Evaluando todo el proceso de desarrollo de *ENS2001*, cabe preguntarse si ha merecido la pena, o si hubiera sido una mejor solución afrontar el desarrollo completo orientado a objetos y tomando *C++* como lenguaje fuente exclusivo.

En cambio, la parte del simulador ha sido más sencilla de analizar y desarrollar desde el principio. El funcionamiento ha quedado muy simple y es fácilmente mantenible y actualizable, como se puede apreciar en la documentación.

En resumen, desde el punto de vista del usuario, la aparición de *ENS2001* supone un avance en el panorama de este tipo de herramientas: desde el punto de vista de facilidad de manejo, con interfaces de texto o gráficas (en su versión *Windows*); desde el punto de vista de disponibilidad en múltiples plataformas (en principio *Linux*) y desde el punto de vista de capacidad de ampliación (es fácil retomar el proyecto y adaptarlo a sus necesidades concretas).

Desde el punto de vista del desarrollador, como se ha venido comentando, el gran interés que tiene *ENS2001* radica en la variedad de campos que toca dentro del mundo del desarrollo *software*, para ser un proyecto de tamaño relativamente pequeño. Durante el desarrollo de este proyecto se han puesto en práctica multitud de conocimientos adquiridos durante el plan de estudios de la carrera y, lo que es más importante, permite pasar de un plano puramente teórico a la consecución de resultados prácticos.

ENS2001 no pretende ser la herramienta definitiva para los alumnos de Compiladores. Se trata más bien de la puesta al día de conceptos de diseño, tanto del sistema como de la interfaz de usuario. La intención es la de proporcionar al estudiante una herramienta cómoda, sencilla y fácil de manejar, para que centre toda su atención en adquirir conocimientos acerca de la creación de compiladores y el lenguaje ensamblador.

5. Futuras Líneas de Trabajo

La aparición de *ENS2001* supone un avance respecto a las herramientas existentes hasta ahora en su campo de aplicación, como pueden ser *ASS* y *ENS96*. No obstante, aún queda un largo camino por recorrer y muchos aspectos que mejorar.

Para tratar de identificarlos mejor, se van a seguir dos caminos diferenciados. Por una parte, están aquellas características de *ENS2001* que pueden ser directamente corregidas o mejoradas. Por otro lado, hay una posible evolución de la herramienta hacia metas más ambiciosas, tratando de abarcar un rango más amplio en el campo de la simulación de sistemas informáticos con fines didácticos.

Ambas metas pueden ser objetivo de próximos Trabajos Fin de Carrera.

Mejoras sobre el panorama actual

Decir que *ENS2001* ha ganado en facilidad de uso respecto a sus predecesores es tan evidente como que cualquier usuario que lo haya sido de *ASS* y *ENS96* va a notar la diferencia en su primer contacto. Sin embargo, quizás en el diseño ha pesado demasiado la voluntad de que la versión en modo texto fuera semejante a *ENS96*, para que la transición entre herramientas fuera lo más suave posible. Este hecho puede haber condicionado la interfaz gráfica, que si bien presenta el aspecto de cualquier otra aplicación de *Windows*, sí que está claramente inspirada en su hermana textual.

Por tanto, quizás en la próxima versión sea la hora de apartar definitivamente la interfaz en modo texto y centrarse en una apariencia totalmente visual, ya que se disponen en la actualidad herramientas de desarrollo rápido de interfaces, tanto para *Windows* como para *Linux*.

Una buena forma de conseguir una interfaz gráfica homogenea para ambos sistemas podría ser rescribir la aplicación usando el lenguaje *Java*. *Java* ya está lo suficientemente evolucionado y se ejecuta con la suficiente rapidez como para implementar un sistema de este tipo (en el que la velocidad de ejecución tampoco es un factor vital). Y daría la posibilidad de ejecutar *ENS2001* en cualquier ordenador que disponga de una versión de la *Máquina Virtual Java*.

Otro punto importante, y que quizás eche en falta el usuario, es la incorporación dentro de *ENS2001* de un editor de texto para poder crear y corregir código fuente directamente desde la aplicación. En esta versión no se ha incluido por dos motivos. El primero de ellos es que, dada la utilidad básica que se plantea (simular código generado por compiladores), es raro que el usuario cree sus programas directamente escribiendo código ensamblador, sino que cargará el código creado por su compilador en la herramienta y procederá a simularlo. El otro motivo se basa en que al ser *ENS2001* una aplicación *Windows*, puede colaborar con otras, en concreto con un editor de texto externo (como podría ser el propio *Bloc de Notas*), en el que el usuario escribirá sus programas y tener ambas aplicaciones abiertas al mismo tiempo.

No obstante, en el futuro se podría plantear la inclusión de un editor propio, que resalte con colores la sintaxis del código fuente y que, incluso, permitiera marcar sobre el editor los errores en el fuente y, por otra parte, seguir la ejecución paso a paso y marcar los puntos de

ruptura sobre el mismo editor, al estilo de los entornos integrados de trabajo para *C++*, *Java* o *Delphi*.

Como mejoras en cuanto a la simulación, quizás se podría ampliar la homogeneidad del juego de instrucciones y permitir algún otro modo de direccionamiento, como podría ser el direccionamiento directo a registro en instrucciones de control de flujo.

Otra mejora que podría introducirse sería proveer un mayor soporte a las excepciones, informando qué instrucción fue la que generó la excepción, e incluso permitir crear una tabla de rutinas para tratar excepciones.

También cabe la posibilidad de permitir que las opciones de configuración del simulador fueran modificadas directamente desde el fichero de código fuente a través de directivas del ensamblador definidas al efecto.

Y por último, un añadido interesante podría ser incluir soporte para números reales, análogo al que se tiene ahora para números enteros, trabajando con los formatos definidos por el estándar *IEEE 754* y dando la posibilidad al usuario de activar o desactivar este soporte.

El futuro

Durante la carrera de Informática se estudia el comportamiento de los sistemas informáticos desde muchos puntos de vista. En algunas asignaturas se hacen prácticas con *hardware* real, pero en otras se emplea el trabajo con simuladores. De hecho, el uso de simuladores está ganando cada día mayor peso ya que permiten reproducir fielmente el comportamiento de sistemas reales, pero haciendo énfasis en las características más interesantes de aquéllos, lo que permite a los alumnos profundizar más y mejor en los fundamentos teóricos que los sustentan.

Una posible evolución de *ENS2001* sería encaminarlo hacia la simulación genérica de una máquina con arquitectura *Von Neumann*. La idea consistiría en seguir los pasos iniciados por *ENS2001* a la hora de construir un simulador totalmente modular, con un diseño orientado a objetos. El sistema podría contar con las siguientes características:

- Capacidad de memoria y ancho de palabra configurable. Gestión de paginación y segmentación.
- Banco de registros configurable. Posibilidad de definir el número de registros, tipo y uso de los mismos (registros de propósito general, cableados a un valor determinado, índices, punteros de pila y funcionamiento de los mismos).
- Unidad aritmético-lógica configurable, para trabajar con números en distintos formatos de representación. Posibilidad de definir el comportamiento de los biestables de estado.
- Juego de instrucciones configurable. Posibilidad de restringir el uso de instrucciones una a una o por grupos. Esto sería útil, por ejemplo, para restringir el uso de instrucciones de coma flotante en prácticas de Fundamentos de los Computadores, o habilitar instrucciones de gestión de interrupciones, excepciones, acceso directo a memoria, etc. También sería interesante tener la posibilidad de definir los modos de direccionamiento permitidos para cada instrucción o grupo de instrucciones.

- Simulación de una unidad de control microprogramada. Posibilidad de definir el nivel de la simulación (con microinstrucciones o con instrucciones).
- Simulación de dispositivos externos, incluso a través de los propios periféricos del ordenador donde se ejecute la herramienta (pantalla, teclado, ratón, impresora, puertos, etc.).
- Gestión de interrupciones. Se podría añadir al procesador un módulo de control de interrupciones. Las interrupciones podrían generarse por hardware (dispositivos externos simulados) o por software (habría que ampliar el lenguaje ensamblador con instrucciones INT que permitan generar interrupciones). Al generarse una interrupción, el procesador interrumpiría el flujo de ejecución para pasar a ejecutar la subrutina encargada de manejar dicha interrupción.
- Gestión de Acceso Directo a Memoria. Al incluir dispositivos externos, se podría proveer a los mismos la posibilidad de acceder directamente a la memoria del sistema sin intervención del procesador, de manera semejante a como actúan los sistemas reales.
- Gestión de Excepciones. En la versión actual, todas las excepciones que se producen detienen la ejecución. Podría implementarse un mecanismo de gestión de excepciones con el cual, mediante la definición de una tabla de punteros, al generarse una excepción se saltaría a una subrutina encargada del manejo de la misma.
- Representación gráfica de todos los componentes del sistema simulado. Se podría diseñar una presentación en pantalla en la que se pudiera ver cómo se interconectan los distintos componentes, memoria, unidad aritmética, procesador, registros, periféricos, etc. De esta forma, no sólo se tendría acceso a los datos almacenados, sino a las rutas que van recorriendo de unos componentes a otros.

De esta manera *ENS2001* se convertiría en un banco de pruebas perfecto para el estudio y aprendizaje de materias relacionadas con la Arquitectura de Computadores, los Compiladores e incluso los Sistemas Operativos.

Las dimensiones de este proyecto pueden parecer monstruosas, pero sin embargo, con estos objetivos en mente y buenos hábitos de diseño y planificación, se podría acometer un enfoque modular para ir construyendo el sistema poco a poco, en sucesivas iteraciones, partiendo de los módulos básicos (procesador, memoria, unidad aritmética y banco de registros totalmente parametrizables) y añadiendo sucesivamente el resto de funcionalidades.

En cuanto a la interfaz para este nuevo sistema, el diseño de *ENS2001* ha demostrado que al separar totalmente el motor de simulación de la interfaz de usuario, el sistema se vuelve demasiado lento a la hora de monitorizar todos los aspectos de la simulación. Por ejemplo, se intentó que durante la ejecución de un programa se actualizara el contenido de todas las ventanas flotantes (memoria, registros, código fuente...) y se observó que el resultado era muy lento. La causa es que el simulador no está pensado para mostrar resultados al usuario constantemente, y era la interfaz la responsable de ir consultando al simulador el estado de todos y cada uno de sus componentes para mostrarlos en pantalla.

Sería más fluido desde el punto de vista del usuario, e interesante desde el punto de vista del diseñador, que cada componente fuera responsable de su interfaz con el usuario, de forma que se pensara en diseñarlos de manera conjunta. Así se podría conseguir una mejora drástica del rendimiento del sistema.

6. Bibliografía

[Aho 1990] Aho, A.V.; Sethi, R.; Ullman, J. D.: *Compiladores. Principios, Técnicas y Herramientas*. Edición Española. Ed. Addison-Wesley Iberoamericana, 1990.

[ASS 1992] *Ensamblador Simbólico ASS 1.0*. http://www-lt.ls.fi.upm.es/compiladores/Software/ass.zip. Descargado el 16-01-2002. 1992.

[Booch 1991] Booch, G.: *Object Oriented Design With Applications*. Ed. The Bejamin/Cummings Publishing Company, Inc., 1991.

[Booch 1999] Booch, G.; Rumbaugh, J.; Jacobson, I.: *El Lenguaje Unificado de Modelado*. Ed. Addison-Wesley Iberoamericana, 1999.

[Borland 2002] *Borland Turbo Assembler 5.0*. Borland Corporation. http://www.borland.com/cbuilder/tass/index.html. Descargado el 03-07-2002.

[Ceballos 1993] Ceballos, F. J.: *Programación Orientada a Objetos con C++*. Ed. RA-MA, Madrid, 1993.

[De Miguel 1996] Miguel, P.: *Fundamentos de los Computadores*. Quinta Edición Revisada. Ed. Paraninfo, 1996.

[Estrada 2002] Estrada, M.: *VisualOS*. http://visualos.sourceforge.net/. Descargado el 27-07-2002.

[Fairclough 1982] Fairclough, D. A.: *A Unique Microprocessor Instruction Set*. IEEE Micro, mayo 1982, páginas 8 a 18.

[Holmes 1995] Holmes, J.: *Object-Oriented Compiler Construction*. Ed. Prentice Hall, 1995.

[IEEE 1998] IEEE-SA Standads Board. *IEEE Recommended Practice for Software Requierments Specifications*. The Institute of Electrical and Electronics Engineers, Inc. 1998.

[Larus 2002] Larus, J.: *SPIM MIPS Simulator*. http://www.cs.wisc.edu/~larus/spim.html. Descargado el 10-06-2002.

[Leblanc 1997] Leblanc, G.: *Borland C++ Builder*. Edición Española. Ed. Gestión 2000, S.A. Barcelona, 1997.

[Levine 1995] Levine, J. R.; Mason, T.; Brown D.: *Lex & Yacc*. Editorial O'Reilly, 1995.

[Microsoft 2002] *Microsoft Macro Assembler 6.1*. Microsoft Corporation http://www.microsoft.com/catalog/display.asp?subid=14&site=239. Descargado el 3-07-2002.

[Minsky 1967] Minsky, M. L.: *Computation of Finite and Infinite Machines*. Ed. Prentice Hall, 1967, páginas 2 a 7.

[Mott 1998] Mott, B. W.: *BSVC. A Microprocessor Simulator Framework*. North Carolina State University. http://www.redlinelabs.com/bsvc/. Descargado el 03-09-2002. Noviembre de 1998.

[Muller 1997] Muller, P. A.: *Modelado de Objetos con UML*. Ed. Gestión 200, Barcelona, 1997.

[RAE 2001] *Diccionario de la Lengua Española. Vigésima segunda edición*. Real Academia Española. http://www.rae.es. Descargado el 18-10-2001.

[Rodríguez 1997] Rodríguez, R.: *ENS 96 versión 2.0.1*. http://www-lt.ls.fi.upm.es/compiladores/Software/ens96201.zip. Descargado el 16-01-2002. 1997.

[Rumbaugh 1996] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Modelado y Diseño Orientados a Objetos (Metodología OMT)*. Edición Española. Ed. Prentice Hall, 1996.

[Schildt 1996] Schildt, H.: *C Manual de Referencia*. Tercera Edición. Ed. McGraw-Hill, 1996.

[Yepes 2002] Yepes, V.: *SimuProc versión 1.3.6.3*. http://orbita.starmedia.com/vlaye/software/simuproc/index2.html. Descargado el 02-09-2002. 25-08-2002.

ANEXO A. Listados de pruebas

A continuación se incluyen los listados fuente de los programas que se han empleado para realizar las pruebas de integración y de sistema de la herramienta.

ENS-PRI-001. Ensamblado del juego de instrucciones

```
;ENS-PRI-001
;Ensamblado del juego de instrucciones

NOP
HALT
MOVE
PUSH
POP
ADD
SUB
MUL
DIV
MOD
INC
DEC
NEG
CMP
AND
OR
XOR
NOT
BR
BZ
BNZ
BP
BN
BV
BNV
BC
BNC
BE
BO
CALL
RET
ININT
INCHAR
INSTR
WRINT
WRCHAR
WRSTR
```

ENS-PRI-002. Ensamblado de combinaciones de modos de direccionamiento

```
;ENS-PRI-002
;Ensamblado de combinaciones de modos de direccionamiento
;Instrucciones de 1 operando

PUSH #33
PUSH .R7
PUSH /0x1234
PUSH [.R5]
```

```
PUSH #10[.IX]
PUSH #40[.IY]
BR $-10

;Instrucciones de 2 operandos

ADD #0xFFFF,#1
ADD #0xFFFF,.R3
ADD #0xFFFF,/65535
ADD #0xFFFF,[.IX]
ADD #0xFFFF,#20[.IX]
ADD #0xFFFF,#-20[.IY]
ADD .R1,#0xABCD
ADD /0x10F2,#0xABCD
ADD [.A],#0xABCD
ADD #0[.IX],#0xABCD
ADD #0xFF[.IY],#0xABCD
```

ENS-PRI-003. Ensamblado de pseudoinstrucciones

```
;ENS-PRI-003
;Ensamblado de pseudoinstrucciones

;ORG
            ORG 0
            ;código de prueba
            MOVE #142,.R3
            ADD .R3,/200
            NOP
            WRCHAR .R3

            ORG 100
            ;código de prueba
            MOVE #142,.R3
            ADD .R3,/200
            NOP
            WRCHAR .R3

            ORG 0xFF
            ;código de prueba
            MOVE #142,.R3
            ADD .R3,/200
            NOP
            WRCHAR .R3

            ORG 65500
            ;código de prueba
            MOVE #142,.R3
            ADD .R3,/200
            NOP
            WRCHAR .R3

;EQU

etiq1 :     EQU 0
etiq2 :     EQU -1
etiq3 :     EQU 65535
etiq4 :     EQU 32767
etiq5 :     EQU -32768
etiq6 :     EQU 0xFFFF
etiq7 :     EQU 0x7FFF
```

```
etiq8 :     EQU 0x8000

            WRINT #etiq1
            WRINT #etiq2
            WRINT #etiq3
            WRINT #etiq4
            WRINT #etiq5
            WRINT #etiq6
            WRINT #etiq7
            WRINT #etiq8

;END
            ;código de prueba
            MOVE #142,.R3
            ADD .R3,/200
            NOP
            WRCHAR .R3
            END
            ;a partir de aquí se ignora todo
            SUB .R3,/200
            HALT

;RES

zona1 :     RES 100
            HALT
zona2 :     RES 0xFF
            RET
zona3 :     RES 32767
            HALT

;DATA

mezcla :    DATA "cadena1\tcadena2",7,"pepito",6,"hola\0adios\n"
enteros :   DATA 1,2,3,4,5,0x6,7,8,9,0xA,0xB
cadenas :   DATA "una cadena","y otra","la ultima"
```

ENS-PRI-004. Ensamblado de etiquetas

```
;ENS-PRI-004
;Ensamblado de etiquetas


                ORG 20

bucle :         MOVE #etiq1,.R1
                MOVE #etiq1,.R2
                MOVE #etiq3[.IX],/datos2
                ADD #etiq4[.IY],
                CMP /datos3,/datos4
                BR $bucle
                HALT

; dando valores a las etiquetas

etiq1 :         EQU 32452
etiq2 :         EQU 0xFAB3
etiq3 :         EQU 44*2+3
etiq4 :         7423%127

espacio :       RES 20
```

```
datos :         DATA 1,2,3,4,5
datos2 :        DATA 6,7,8,9,0
datos3 :        DATA "hola","que","tal"
datos4 :        DATA "cadena\ncompuesta"

                END
```

ENS-PRI-005. Generación de errores durante el ensamblado

```
;ENS-PRI-005
;Generación de errores durante el ensamblado

;01. Modo de direccionamiento del operando 1 erróneo

                NOP #3
                HALT .R4
                MOVE $345,/33
                BR #0x415
                BZ .R6
                BNZ #0xFF[.IX]
                BC #-32[.IY]
                RET /444
                RET [.R6]
                RET #11
                RET #-32[.IX]
                RET #0x11[.IY]
                RET $52
                RET .R1
                INCHAR #142
                ININT $0x634
                INSTR .R4

;02. Modo de direccionamiento del operando 2 erróneo

                MOVE #42,#52
                MOVE [.R6],$25
                CMP #0[.IX],$674

;03. Instrucción no reconocida

                MOVER #4,.R8
                COMPARAR [.R3],[.R3]
                RESTAR /3000,#42[.IX]

;04. Operando 1 incorrecto

                MOVE 1,.R5
                SUB entero1,[.R1]
                NEG "operando"

;05. Operando 2 incorrecto

                ADD #4,425
                MUL /0xFFFF,"operando2"

;06. Etiqueta duplicada

etiq1 :         EQU 0
                ...
etiq1 :         RES 100
```

```
;07. Etiqueta no definida

                POP .R2
                CMP .R2,#0
                BN /ERROR
                WRSTR /EL_FACT_ES
                WRINT .R2
                HALT
ERROR:          CMP .R2, #COD_NEGATIVO
                BZ /ERROR1
                WRSTR /NUMERO_GRANDE
                HALT
ERROR1:         WRSTR /NUMERO_NO_VALIDO
                HALT

;08. Entrada incorrecta

                NOP
                MOVE %R1,[.R1]
zona1 :         RES 400.25

;09. Expresión errónea

etiq1 :         EQU (31+2))
etiq2 :         EQU (0xff*24)-(100.4%3)

;10. ORG define el comienzo fuera de la memoria

                ORG -50000
                ORG 65536

;11. RES reserva espacio fuera de la memoria

                ORG 65000
zona1 :         RES 536

;12. DATA ubica datos fuera de la memoria

                ORG 65530
datos :         DATA "esta es una cadena muy larga para meter en memoria"

;13. Expresión fuera de rango

expresion :     EQU 65535+1
expresion2 :    EQU 30000*4

;14. Nombre de etiqueta reservado

                XOR [.R1],[.R1]
                POP /4000
ret :           RET ; retorno de la función

;15. Entero fuera de rango

                MOVE #65536,/314
                MUL [.R3],/65536
                DIV -32769,.R0
                NOT #-129[.IX]
                NEG #128[.IY]

;16. Se esperaba el operando 1
```

```
                MOVE
                CALL
                BP
                ADD ,[.R6]

;17. Se esperaba el operando 2

                AND #33,
                OR /0xFF

;18. Se esperaba el carácter de fin de línea

                NOP HALT
                ADD #1,#2,#3

;19. Se esperaba el carácter separador

                MOVE .R3 .R6
                SUB #145 #0xFF[.IX]

;20. Lista de datos errónea

datos :         DATA "cadena1,"cadena2,cadena3
enteros :       DATA 1 2 3 4 5 6

;21. Desplazamiento fuera de rango

                ORG 0
                BR $256
                BZ $rutina
                ORG 10000
rutina :        PUSH .R1
                PUSH .R2
                ...
                RET
```

ENS-PRI-006. Instrucciones básicas

```
;ENS-PRI-006
;Instrucciones básicas

            ORG 0
            NOP
            NOP
            NOP
            HALT
            END
```

ENS-PRI-007. Instrucciones de transferencia de datos

```
;ENS-PRI-007
;Instrucciones de transferencia de datos

            ORG 0
            ;preparación de los datos
            MOVE #0xFF,.R4
            MOVE #0xFE,/3000
            MOVE #0xFD,[.R6]
            MOVE #0xFC,#0xFF[.IX]
```

```
            MOVE #0xFB,#127[.IY]

            ;instrucciones de transferencia
            ;ver tabla 3.5.1
            MOVE #10,.R1
            MOVE #10,/1000
            MOVE #10,[.R3]
            MOVE #10,#20[.IX]
            MOVE #10,#-10[.IY]
            MOVE .R4,/2000
            MOVE /3000,/2000
            MOVE [.R6],/2000
            MOVE #0xFF[.IX],/2000
            MOVE #127[.IY],/2000

            HALT
            END
```

ENS-PRI-008. Instrucciones de manejo de pila

```
;ENS-PRI-008
;Instrucciones de manejo de pila

            ORG 0
            ;preparación de los datos
            MOVE #0xFE,.R4
            MOVE #0xFD,/3000
            MOVE #0xFC,[.R6]
            MOVE #0xFB,#0xFF[.IX]
            MOVE #0xFA,#127[.IY]

            MOVE #0xFFFF,.SP ; pila descendente
            ;MOVE #0x7FFF,.SP ; pila ascendente

            ;cargando la pila
            PUSH #0xFF
            PUSH .R4
            PUSH /3000
            PUSH [.R6]
            PUSH #0xFF[.IX]
            PUSH #127[.IY]

            ;descargando de la pila
            POP .R4
            POP /3000
            POP [.R6]
            POP #0xFF[.IX]
            POP #127[.IY]

            HALT
            END
```

ENS-PRI-009. Ensamblado del juego de instrucciones

```
;ENS-PRI-009
;Instrucciones aritméticas

            ORG 0

            ;preparación de los datos
            MOVE #0xFE,.R4
```

```
MOVE #0xFD,/3000
MOVE #0xFC,[.R6]
MOVE #0xFB,#0xFF[.IX]
MOVE #0xFA,#127[.IY]

;prueba de modos de direccionamiento
;1 operando
INC /3000
INC .R4
DEC [.R6]
NEG #0xFF[.IX]
NEG #127[.IY]

;2 operandos
ADD #255,#255
ADD #255,/3000
ADD #255,.R4
ADD #255,[.R6]
ADD #255,#0xFF[.IX]
ADD #255,#127[.IY]
SUB /3000,#255
SUB /3000,/3000
SUB /3000,.R4
SUB /3000,[.R6]
SUB /3000,#0xFF[.IX]
SUB /3000,#127[.IY]
MUL .R4,#255
MUL .R4,/3000
MUL .R4,.R4
MUL .R4,[.R6]
MUL .R4,#0xFF[.IX]
MUL .R4,#127[.IY]
DIV [.R6],#255
DIV [.R6],/3000
DIV [.R6],.R4
DIV [.R6],[.R6]
DIV [.R6],#0xFF[.IX]
DIV [.R6],#[.IY]
MOD #0xFF[.IX],#255
MOD #0xFF[.IX],/3000
MOD #0xFF[.IX],.R4
MOD #0xFF[.IX],[.R6]
MOD #0xFF[.IX],#0xFF[.IX]
MOD #0xFF[.IX],#127[.IY]
CMP #127[.IY],#255
CMP #127[.IY],/3000
CMP #127[.IY],.R4
CMP #127[.IY],[.R6]
CMP #127[.IY],#0xFF[.IX]
CMP #127[.IY],#127[.IY]

;prueba de biestables y resultados
CMP #89,#89 ;Z
MOVE .R2,#0x7FFF
INC .R2 ;NZ
ADD #-425,#3333 ; P
SUB #-425,#4444 ; N
MUL #0xFFFF,#3 ; V
DIV #64,#4 ; NV
ADD #32767,#1 ; C
SUB #32767,#1 ; NC
```

```
            ADD #0,#0x6666 ; PE
            SUB #0x6666,#1 ; PO

            HALT
            END
```

ENS-PRI-010. Instrucciones lógicas

```
;ENS-PRI-010
;Instrucciones lógicas

            ORG 0
            AND #6,#5
            WRINT .A
            WRSTR /eol
            OR #6,#5
            WRINT .A
            WRSTR /eol
            XOR #6,#5
            WRINT .A
            WRSTR /eol
            MOVE #1,.A
            NOT .A
            WRINT .A
            WRSTR /eol
            HALT

eol :       DATA "\n"
            END
```

ENS-PRI-011. Instrucciones de control de flujo de ejecución

```
;ENS-PRI-011
;Instrucciones de control de flujo de ejecución

            ORG 0
            BR /e_br
n_br :      CMP #0xFF,#0xFF
            BZ /e_bz
n_bz :      ADD #0,#1
            BNZ /e_bnz
n_bnz :     SUB #30,#20
            BP /e_bp
n_bp :      SUB #20,#30
            BN /e_bn
n_bn :      ADD #0xFFFF,#3
            BV /e_bv
n_bv :      MUL #0xFF,#2
            BNV /e_bnv
n_bnv :     ADD #0x7FFF,#1
            BC /e_bc
n_bc :      DIV #50,#5
            BNC /e_bnc
n_bnc :     ADD #0,#0xFFFF
            BE /e_be
n_be :      ADD #0,#0xFFFE
            BO /e_bo
n_bo :      HALT

e_br :      WRSTR /s_br
            BR /n_br
```

```
e_bz :      WRSTR /s_bz
            BR /n_bz
e_bnz :     WRSTR /s_bnz
            BR /n_bnz
e_bp :      WRSTR /s_bp
            BR /n_bp
e_bn :      WRSTR /s_bn
            BR /n_bn
e_bv :      WRSTR /s_bv
            BR /n_bv
e_bnv :     WRSTR /s_bnv
            BR /n_bnv
e_bc :      WRSTR /s_bc
            BR /n_bc
e_bnc :     WRSTR /s_bnc
            BR /n_bnc
e_be :      WRSTR /s_be
            BR /n_be
e_bo :      WRSTR /s_bo
            BR /n_bo

s_br :      DATA "BR\n"
s_bz :      DATA "BZ\n"
s_bnz :     DATA "BNZ\n"
s_bp :      DATA "BP\n"
s_bn :      DATA "BN\n"
s_bv :      DATA "BV\n"
s_bnv :     DATA "BNV\n"
s_bc :      DATA "BC\n"
s_bnc :     DATA "BNC\n"
s_be :      DATA "BE\n"
s_bo :      DATA "BO\n"

            END
```

ENS-PRI-012. Instrucciones de manejo de subrutinas

```
;ENS-PRI-012
;Instrucciones de manejo de subrutinas

            ORG 0
            MOVE #0xFFFF,.SP ; Pila decreciente
            ;MOVE #0x7FFF,.SP ; Pila creciente
            CALL /sub1
            CALL /sub2
            CALL /sub3
            HALT

sub1 :      WRSTR /id1
            WRSTR /eol
            WRSTR /rsp
            WRINT .SP
            WRSTR /eol
            CALL /sub2
            RET

            ORG 20000

sub2 :      WRSTR /id1
            WRSTR /eol
            WRSTR /rsp
```

```
            WRINT .SP
            WRSTR /eol
            CALL /sub2
            CALL /sub3
            RET

            ORG 40000

sub3 :      WRSTR /id1
            WRSTR /eol
            WRSTR /rsp
            WRINT .SP
            WRSTR /eol
            RET

            ORG 60000
id1 :       DATA "Subrutina 1"
id2 :       DATA "Subrutina 2"
id3 :       DATA "Subrutina 3"
rsp :       DATA "Valor de SP="
eol :       DATA "\n"
            END
```

ENS-PRI-013. Instrucciones de entrada/salida

```
;ENS-PRI-013
;Instrucciones de entrada/salida

            ORG 0

            WRSTR /cad01
            ININT /entero
            WRSTR /cad11
            WRINT /entero
            WRSTR /eol

            WRSTR /cad02
            INCHAR /caracter
            WRSTR /cad12
            WRCHAR /caracter
            WRSTR /eol

            WRSTR /cad03
            INSTR /cadena
            WRSTR /cad13
            WRSTR /cadena
            WRSTR /eol

            HALT

cad01 :     DATA "Introduzca un entero"
cad02 :     DATA "Introduzca un carácter"
cad03 :     DATA "Introduzca una cadena de caracteres"
cad11 :     DATA "El entero leído es: "
cad12 :     DATA "El carácter leído es: "
cad13 :     DATA "La cadena leída es: "
eol :       DATA "\n"
entero :    RES 1
caracter :  RES 1
cadena :    RES 0xFF
            END
```

ENS-PRI-014. Generación de excepciones

```
;ENS-PRI-014
;Generación de excepciones

;Excepción 01. Instrucción no implementada

            ORG 0
            DATA "elemento1","elemento2","elemento3"
            DATA 345,125,2096,41,5286
            END

;Excepción 02. División por cero

            ;inmediato
            DIV #42,#0

            ;memoria
            MOVE #0,/0xFF00
            DIV #52,/0xFF00

            ;registro
            MOVE #0,.R5
            DIV #11,.R5

            ;indirecto
            MOVE #0x100,.R3
            MOVE #0,[.R3]
            DIV #52,[.R3]

            ;relativo a ix
            MOVE #0x200,.IX
            MOVE #0,#22[.IX]
            DIV #0xFF,#22[.IX]

            ;relativo a iy
            MOVE #0x300,.IY
            MOVE #0,#-10[.IY]
            DIV #0xFF,#-10[.IY]

            END

;Excepción 03. Contador de programa sobrepasa el final de la memoria

            ORG 0
            BR /final
            ORG 0xFFFF
final :     NOP
            END

;Excepción 04. Contador de programa invade la zona de pila

            ORG 0
pila :      EQU 0xFFFF
            MOVE #pila,.SP
            PUSH #0xFFFF
            PUSH #0xFFFF
            PUSH #0xFFFF
            BR /pila
            END
```

```
;Excepción 05. Puntero de pila invade la zona de código

            ORG 0
            MOVE #0xFFFF,.SP
bucle :     MOVE #0xFF,.R2
            PUSH .R2
            DEC .R2
            BNZ $bucle
            BR /final
            ORG 0xFFFA
final :     HALT

            END
```

ENS-PRS-001. fact.ens

```
;fact.ens - ENS2001 Ejemplo de uso - Abril 2002
;Adaptado del original para ENS96, por Raul Rodriguez Barrio (Octubre 1996)
;NOTA: La ausencia de acentos es intencionada, para permitir la
;legibilidad tanto en entornos Windows como Linux

;El programa pide un entero y a continuacion calcula el factorial y
;lo muestra en pantalla.
;Es necesario que la pila este configurada hacia direcciones decrecientes,
;si no, el ejemplo no funcionara correctamente.

                ; coloco el puntero de pila en la cima de la memoria
                        MOVE #65535,.SP
                ; Escribo un mensaje y solicito un numero
                        WRSTR /UN_NUMERO
                        ININT .R1
                ; preparo el argumento de la funcion y la llamo
                        PUSH .R1
                        CALL /SUB_FACT
                ; recupero el valor y lo trato segun lo que devolvio
                        POP .R2
                        CMP .R2,#0
                        BN /ERROR
                        WRSTR /EL_FACT_ES
                        WRINT .R2
                        HALT
ERROR:                  CMP .R2, #COD_NEGATIVO
                        BZ /ERROR1
                        WRSTR /NUMERO_GRANDE
                        HALT
ERROR1:                 WRSTR /NUMERO_NO_VALIDO
                        HALT

                ; almacenamiento de la cadenas de caracteres con su referencias

UN_NUMERO:              DATA "\nUn numero: "
NUMERO_NO_VALIDO:       DATA "\nNumero no valido"
NUMERO_GRANDE:          DATA "\nNumero demasiado grande"
EL_FACT_ES:             DATA "\nEl factorial es "

COD_NEGATIVO:           EQU -1
COD_OVERFLOW:           EQU -2


; ************************************************************************
; Rutina SUB_FACT
;
;       Calcula el factorial de un numero que le viene en la pila
;
;
; ************************************************************************
```

```
                ; salvo los registros en el marco de pila
SUB_FACT:               PUSH .IY
                        PUSH .R1
                        PUSH .R2
                ; coloco el marco de pila
                        MOVE .SP, .IY
                ; recojo el argumento
                        MOVE #5[.IY],.R1
                        CMP .R1,#0
                        BP /SIGUE1
                ; es negativo: devuelvo un codigo de error de numero negativo.
                        MOVE #COD_NEGATIVO,.R2
                        BR /VOLVER
                ; valor no negativo
SIGUE1:                 BP /SIGUE2
                        MOVE #1,.R2
                        BR /VOLVER
                ; valor positivo en R1
SIGUE2:                 MOVE #1,.R2
SIGUEMUL:               MUL .R2, .R1
                        BV /OVERFLOW
                        MOVE .A, .R2
                        DEC .R1
                        CMP .R1, #0
                        BZ /VOLVER
                        BR /SIGUEMUL
                ; se produjo overflow, devuelvo codigo de error de OVERFLOW
OVERFLOW:               MOVE #COD_OVERFLOW,.R2
                ; retorno  el valor en la misma posicion del argumento
VOLVER:                 MOVE .R2, #5[.IY]
                ; restauro los valores y retorno
                        POP .R2
                        POP .R1
                        POP .IY
                        RET
```

ENS-PRS-002. matriz.ens

```
;matriz.ens - ENS2001 Ejemplo de uso - Abril 2002
;Adaptado del original para ENS96, por Raul Rodriguez Barrio (Octubre 1996)
;NOTA: La ausencia de acentos es intencionada, para permitir la
;legibilidad tanto en entornos Windows como Linux

;Muestra un menu que ofrece opciones para manejar una matriz 5x5
;1.Introducir un valor
;2.Consultar un valor
;3.Mostrar el contenido de la matriz
;4.Salir

nFilas:                 EQU 5    ; numero de filas de la matriz
nColumnas:              EQU 5    ; numero de columnas de la matriz
nOpciones:              EQU 4    ; num. de opciones
opcAlmacenar:           EQU 1
opcObtener:             EQU 2
opcMostrar:             EQU 3
opcSalir:               EQU 4

                        MOVE #0x7FFF,.SP ; inicializar la pila
bucleGeneral:           CALL /funcMenu
                        MOVE .R1, .R5 ; guardar la opcion en R5
                        CMP .R5, #opcSalir ; comprobar que es la opcion
                                           ; de salir
                        BZ /fin
                        CMP .R5, #opcMostrar ; comprobar que es la opcion de
                                             ; mostrar
                        BZ /opcion3
```

```
                        CALL /solicitarPosicion ; solicitar par de posicion en
                                                ; la matriz
                        CMP .R5 , #opcAlmacenar ; comprobar si es opcion 1 o 2
                        BNZ /opcion2
opcion1:                WRSTR /sPedirValor ; solicitar un valor
                        ININT .R3
                        CALL /ponerValor ; colocar valor en la memoria
                        BR /bucleGeneral
opcion2:                CALL /dameValor ; solicitar valor de la matriz
                        WRSTR /sDecirValor ; sacar valor por pantalla
                        WRINT .R3
                        BR /bucleGeneral
opcion3:                CALL /mostrarMatriz ; llamar a procedimiento de mostrar
                                            ; matriz
                        WRCHAR #13
                        BR /bucleGeneral
fin:                    HALT

longAlmacenMatriz:      EQU 5*5
dMatriz:                RES 25

sRetCarro:              DATA "\n"
sAlmacenar:             DATA "\n1 .- Amacenar valor"
sObtener:               DATA "\n2 .- Obtener valor"
sMostrar:               DATA "\n3 .- Escribir Matriz"
sSalir:                 DATA "\n4 .- Salir"
sPedirOpcion:           DATA "\n   Dame opcion:"
sPedirFila:             DATA "\n Dame número de fila: "
sPedirColumna:          DATA "\n Dame número de columna: "
sDecirValor:            DATA "\n El valor es "
sPedirValor:            DATA "\n Valor a almacenar: "

funcMenu:               WRSTR /sRetCarro
                        WRSTR /sAlmacenar
                        WRSTR /sObtener
                        WRSTR /sMostrar
                        WRSTR /sSalir
                        WRSTR /sPedirOpcion
menuDeNuevo:            INCHAR .R1 ; lee la opcion
                        SUB .R1, #48 ; resta el valor ascii del '0'
                        CMP .A,#1 ; comprueba límite inferior
                        BN /menuDeNuevo ; si no valido, repetir lectura
                        CMP #nOpciones,.A ; comprueba límite superior
                        BN /menuDeNuevo ; si no valido, repetir lectura
                        MOVE .A, .R1 ; colocar salida en .R1
                        RET

solicitarPosicion:      WRSTR /sPedirFila ; mensaje de solicitud
                        ININT .R1
                        CMP .R1, #1 ; comprueba límite inferior
                        BN /solicitarPosicion ; si no valido, repetir
                        CMP .R1, #nFilas ; comprueba límite superior
                        BP /solicitarPosicion ; si no valido, repetir
otraColumna:            WRSTR /sPedirColumna ; mensaje de solicitud
                        ININT .R2
                        CMP .R2, #1 ; comprueba límite inferior
                        BN /otraColumna ; si no valido, repetir
                        CMP .R2, #nColumnas ; comprueba límite superior
                        BP /otraColumna ; si no valido, repetir
                        RET

dameValor:              DEC .R2
                        DEC .R1
                        MUL .R1, #nColumnas
                        ADD .A, .R2
                        ADD .A, #dMatriz
                        MOVE .A, .IX
                        MOVE #0[.IX], .R3
```

```
                        RET
ponerValor:             DEC .R2
                        DEC .R1
                        MUL .R1, #nColumnas
                        ADD .A, .R2
                        ADD .A, #dMatriz
                        MOVE .A, .IX
                        MOVE .R3, #0[.IX]
                        RET
mostrarMatriz:          MOVE #nFilas, .R1
                        MOVE #dMatriz, .IX
                        WRSTR /sRetCarro
otraFila:               MOVE #0, .R2
sigueFila:              WRINT #0[.IX]
                        INC .R2
                        INC .IX
                        WRCHAR #32
                        CMP .R2, #nColumnas
                        BNZ /sigueFila
                        WRSTR /sRetCarro
                        DEC .R1
                        BNZ /otraFila
                        RET
```

ENS-PRS-003. maxmin.ens

```
;maxmin.ens - ENS2001 Ejemplo de uso - Abril 2002
;NOTA: La ausencia de acentos es intencionada, para permitir la
;legibilidad tanto en entornos Windows como Linux

;Calculo del maximo y minimo de una lista de numeros enteros
;El programa pide una lista de numeros que acabara cuando el
;usuario introduzca un cero o bien un numero no valido
;Para comparar numeros se sigue este criterio:
;   r = a - b
;   si r>0 -> si desbordamiento -> a < b
;          -> si no desbordamiento -> a > b
;   si r<0 -> si desbordamiento -> a > b
;          -> si no desbordamiento -> a < b

        ;R1 almacenara el valor mayor
        ;R2 almacenara el valor menor
        ;R0 almacenara el valor leido
        WRSTR /PROMPT

        ;lee el primero, si es cero ha terminado
        ININT .R0
        MOVE .R0,.R1
        MOVE .R0,.R2
        CMP .R0,#0
        BZ $FIN

        ;lee numeros hasta encontrar un cero
LECTURA:
        WRSTR /PROMPT
        ININT .R0
        CMP .R0,#0
        BZ $FIN

        ;si R0>R1 -> R0 es el mayor hasta ahora
        CMP .R0,.R1
        BN $SIGNO11
SIGNO10:
        BNZ $MAYOR
        BR $SIG
SIGNO11:
        BV $MAYOR
```

```
        ;si R0<R2 -> R0 es el menor hasta ahora
SIG:
        CMP .R0,.R2
        BZ $SIGNO21
SIGNO20:
        BV $MENOR
        BR $OTRO
SIGNO21:
        BNV $MENOR

        ;leer otro numero
OTRO:
        BR $LECTURA
        ;fin del programa
        ;muestra maximo y minimo
FIN:
        WRSTR /MAXIMO
        WRINT .R1
        WRSTR /EOL
        WRSTR /MINIMO
        WRINT .R2
        WRSTR /EOL
        HALT
MAYOR:
        MOVE .R0,.R1
        BR $SIG
MENOR:
        MOVE .R0,.R2
        BR $OTRO
        ;cadenas
PROMPT: DATA "Introduzca un numero: "
MAXIMO: DATA "El valor maximo introducido es "
MINIMO: DATA "El valor minimo introducido es "
EOL:    DATA "\n"
        END
```

ENS-PRS-004. cadenas.ens

```
;cadenas.ens - ENS2001 Ejemplo de uso - Abril 2002
;NOTA: La ausencia de acentos es intencionada, para permitir la
;legibilidad tanto en entornos Windows como Linux

;Lectura de dos cadenas, concatenar y dar la vuelta.
;El programa pide al usuario que introduzca dos cadenas cualesquiera.
;A continuacion, las concatena y muestra el resultado en pantalla al
;derecho y al reves

                ;leer cadena1
                WRSTR /mens1
                INSTR /cadena1
                ;leer cadena2
                WRSTR /mens2
                INSTR /cadena2
                ;concatenar
                MOVE #cadena1,.R1
                MOVE #cadena2,.R2
                MOVE #resu1,.R0
                MOVE #resu2,.R3
cad1:           CMP [.R1],#0 ; fin de cadena?
                BZ $cad2 ; pasamos a la siguiente
                MOVE [.R1],[.R0] ; copiamos un caracter
                INC .R0 ; incrementamos ambos
                INC .R1 ; punteros
                BR $cad1
cad2:           CMP [.R2],#0 ; fin de cadena?
                BZ $fin
```

```
                MOVE [.R2],[.R0] ; copiamos un caracter
                INC .R0 ; incrementamos ambos
                INC .R2 ; punteros
                BR $cad2
fin:            MOVE #0,[.R0]
                ;mostrar al derecho
                WRSTR /derecho
                WRSTR /resu1
                WRSTR /eol
                ;mostrar al reves
                CMP .R0,#resu1
                BZ $fin2
rev:            DEC .R0
                CMP [.R0],#0
                BZ $fin2
                MOVE [.R0],[.R3]
                INC .R3
                BR $rev
fin2:           MOVE #0,[.R3]
                WRSTR /reves
                WRSTR /resu2
                WRSTR /eol
                HALT
                ;cadenas
mens1:          DATA "Introduzca la primera cadena: "
mens2:          DATA "Introduzca la segunda cadena: "
derecho:        DATA "Cadena al derecho: "
reves:          DATA "Cadena al reves: "
eol:            DATA "\n"
cadena1:        RES 500
cadena2:        RES 500
resu1:          RES 1000
resu2:          RES 1000
                END
```

ENS-PRS-005. calculadora.ens

```
;calculadora.ens - ENS2001 Ejemplo de uso - Abril 2002
;NOTA: La ausencia de acentos es intencionada, para permitir la
;legibilidad tanto en entornos Windows como Linux

;en R0 se guarda la direccion de la funcion de operacion
;en R1 se guarda el primer operando
;en R2 se guarda el segundo operando
;en A se guarda el resultado de la operacion
;en R3 se guarda la opcion elegida por el usuario

ORG 0

INICIO:
;Muestra las operaciones por la consola
             WRSTR /menu1
             WRSTR /menu2
             WRSTR /menu3
             WRSTR /menu4
             WRSTR /menu5
             WRSTR /menu6
             WRSTR /menu7
             WRSTR /menu8
             WRSTR /menu0
;Lectura de la operacion
             WRSTR /cad1
             ININT .R3
;Se calcula la direccion de la funcion correspondiente
             CMP .R3,#0
             BZ /SALIR
             CMP .R3,#1
```

```
            BZ $OP1
            CMP .R3,#2
            BZ $OP2
            CMP .R3,#3
            BZ $OP3
            CMP.R3,#4
            BZ $OP4
            CMP .R3,#5
            BZ $OP5
            CMP .R3,#6
            BZ $OP6
            CMP .R3,#7
            BZ $OP7
            CMP .R3,#8
            BZ $OP8
            BZ $OPERROR
OPERANDOS:
;primer operando - se guarda en la variable OPERANDO1
            WRSTR /cad2
            ININT /OPERANDO1
;segundo operando - se guarda en la variable OPERANDO2
            WRSTR /cad3
            ININT /OPERANDO2
;se ponen los parametros en la pila
            PUSH /RESULTADO
            PUSH /OPERANDO1
            PUSH /OPERANDO2
;se llama a la funcion operar
            CALL /OPERAR
;se recuperan los valores de retorno de la funcion
            POP /OPERANDO2
            POP /OPERANDO1
            POP /RESULTADO
;se muestra el resultado
            WRSTR /cad4
            WRINT /RESULTADO
            WRSTR /saltolin
ACARREO:
            BNC $DESBORDAMIENTO
            WRSTR /cad7
DESBORDAMIENTO:
            BNV $ACABAR
            WRSTR /cad6
;vuelta a empezar
ACABAR:
            BR /INICIO

;direcciones de las funciones
OP1:
            MOVE #SUMA,.R0
            BR /OPERANDOS
OP2:
            MOVE #RESTA,.R0
            BR /OPERANDOS
OP3:
            MOVE #PRODUCTO,.R0
            BR /OPERANDOS
OP4:
            MOVE #DIVISION,.R0
            BR /OPERANDOS
OP5:
            MOVE #MODULO,.R0
            BR /OPERANDOS
OP6:
            MOVE #Y,.R0
            BR /OPERANDOS
OP7:
            MOVE #O,.R0
```

```
            BR /OPERANDOS
OP8:
            MOVE #OEX,.R0
            BR /OPERANDOS
OPEROR:
            MOVE #ERROR,.R0
            BR /OPERANDOS

;Funciones (paso de parametros por registros)
SUMA:
            ADD.R1,.R2
            RET
RESTA:
            SUB .R1,.R2
            RET
PRODUCTO:
            MUL .R1,.R2
            RET
DIVISION:
            DIV .R1,.R2
            RET
MODULO:
            MOD .R1,.R2
            RET
Y:
            AND .R1,.R2
            RET
O:
            OR .R1,.R2
            RET
OEX :
            XOR .R1,.R2
            RET
ERROR :
            WRSTR /cad5
            RET
SALIR :
            HALT

;funcion operar     (paso de parametros por pila)
OPERAR :
            MOVE .SP,.IX
            MOVE #3[.IX],.R1 ;se recupera el primer argumento
            MOVE #2[.IX],.R2 ;se recupera el segundo argumento
            CALL [.R0] ;se llama a la operacion pertinente
            MOVE .A,#4[.IX] ;se devuelve el resultado
            RET

;cadenas de texto
cad1:       DATA "Introduzca la operacion: "
cad2:       DATA "Introduzca el primer operando: "
cad3:       DATA "Introduzca el segundo operando: "
cad4:       DATA "El resultado de la operacion es: "
cad5:       DATA "La operacion introducida no es correcta.\n"
cad6:       DATA "La operacion produjo desbordamiento.\n"
cad7:       DATA "La operacion produjo acarreo.\n"
menu1:      DATA "1.Suma\n"
menu2:      DATA "2.Resta\n"
menu3:      DATA "3.Producto\n"
menu4:      DATA "4.Division\n"
menu5:      DATA "5.Modulo\n"
menu6:      DATA "6.And\n"
menu7:      DATA "7.Or\n"
menu8:      DATA "8.Xor\n"
menu0:      DATA "0.Salir\n"
saltolin:   DATA "\n"

;variables
```

```
OPERANDO1:    RES 1
OPERANDO2:    RES 1
RESULTADO:    RES 1

              END
```

ENS-PRS-006. fact_rec.ens

```
;fact_rec.ens - ENS2001 Ejemplo de uso - Noviembre 2002
;version recursiva del programa fact.ens (calculo del factorial)
;NOTA: La ausencia de acentos es intencionada, para permitir la
;legibilidad tanto en entornos Windows como Linux

;El programa pide un entero y a continuacion calcula el factorial y
;lo muestra en pantalla.
;Es necesario que la pila este configurada hacia direcciones
decrecientes,
;si no, el ejemplo no funcionara correctamente.

                ; coloco el puntero de pila en la cima de la memoria
                        MOVE #65535,.SP
                ; Escribo un mensaje y solicito un numero
                        WRSTR /UN_NUMERO
                        ININT .R1
                ; preparo el argumento de la funcion y la llamo
                        PUSH .R1
                        CALL /SUB_FACT
                ; recupero el valor y lo trato segun lo que devolvio
                        POP .R2
                        CMP .R2,#0
                        BN /ERROR
                        WRSTR /EL_FACT_ES
                        WRINT .R2
                        HALT
ERROR:                  CMP .R2, #COD_NEGATIVO
                        BZ /ERROR1
                        WRSTR /NUMERO_GRANDE
                        HALT
ERROR1:                 WRSTR /NUMERO_NO_VALIDO
                        HALT

                ; almacenamiento de la cadenas de caracteres con su
referencias

UN_NUMERO:              DATA "\nUn numero: "
NUMERO_NO_VALIDO:       DATA "\nNumero no valido"
NUMERO_GRANDE:          DATA "\nNumero demasiado grande"
EL_FACT_ES:             DATA "\nEl factorial es "

COD_NEGATIVO:           EQU -1
COD_OVERFLOW:           EQU -2


;
;*********************************************************************
; Rutina SUB_FACT
;
;       Calcula el factorial de un numero que le viene en la pila
;
;
```

```
;
**************************************************************************

                ; salvo los registros en el marco de pila
SUB_FACT:               PUSH .IY
                        PUSH .R1
                        PUSH .R2
                ; coloco el marco de pila
                        MOVE .SP, .IY
                ; recojo el argumento
                        MOVE #5[.IY],.R1
                        CMP .R1,#0
                        BP /SIGUE1
                ; es negativo: devuelvo un codigo de error de numero
negativo.
                        MOVE #COD_NEGATIVO,.R2
                        BR /VOLVER
                ; valor no negativo
SIGUE1:                 BP /SIGUE2
                ; cero
CASOBASICO:             MOVE #1,.R2
                        BR /VOLVER
                ; valor positivo en R1
SIGUE2:                 CMP .R1,#1
                        BZ /CASOBASICO
                ; preparo el argumento de la funcion y la llamo
                ; (llamada recursiva)
                        SUB .R1,#1
                        PUSH .A
                        CALL /SUB_FACT
                ; recupero el valor y lo trato segun lo que devolvio
                        POP .R2
                        MUL .R2,.R1
                ; compruebo el overflow
                        BV /OVERFLOW
                        MOVE .A,.R2
                        BR /VOLVER
                ; se produjo overflow, devuelvo codigo de error de
OVERFLOW
OVERFLOW:               MOVE #COD_OVERFLOW,.R2
                ; retorno  el valor en la misma posicion del argumento
VOLVER:                 MOVE .R2, #5[.IY]
                ; restauro los valores y retorno
                        POP .R2
                        POP .R1
                        POP .IY
                        RET
```

ANEXO B. Eventos de la interfaz gráfica de *Windows*

En este anexo se detallan los eventos de la interfaz gráfica de *Windows* manejados por la versión gráfica de *ENS2001*.

OnClick

El evento `OnClick` se envía cuando el usuario hace clic con el botón principal del ratón sobre un componente de la interfaz. Si el componente tiene una acción asociada y la acción tiene definido un método `OnExecute`, dicho método responderá al evento `OnClick`, salvo que se haya definido un manejador específico para este último.

Este evento también se puede enviar cuando:

- El usuario pulsa la barra espaciadora mientras un botón o *check box* tiene el foco.
- El usuario pulsa INTRO cuando el formulario activo tiene un botón por defecto (especificado en el atributo `Default`).
- El usuario pulsa ESCAPE cuando el formulario activo tiene un botón de cancelar (espeficicado por el atributo `Cancel`).
- El usuario pulsa una combinación de teclas de acceso rápido para un botón o *check box*.
- En un formulario, el evento `OnClick` se envía cuando el usuario hace clic en cualquier área desocupada del formulario o en un componente deshabilitado dentro del mismo.

OnClose

El evento `OnClose` se envía cuando un formulario se cierra invocando al método `Close` o cuando el usuario elige Cerrar desde el menú de sistema del formulario.

OnCloseQuery

El evento `OnCloseQuery` se envía cuando se pide a un formulario que se cierre. Existe un atributo lógico, llamado `CanClose`, que determina si se permite cerrar el formulario o no. Se puede emplear un manejador de este evento para preguntar al usuario si está seguro de que quiere cerrar un formulario.

OnExecute

La mayor parte de las ocasiones, el evento `OnExecute` se envía cuando el usuario hace clic sobre un componente que tiene una acción definida.

OnHide

El evento `OnHide` se envía cuando se oculta un formulario, esto es, cuando el atributo `Visible` toma el valor `false`.

OnKeyDown

El evento `OnKeyDown` se envía cuando se pulsa una tecla. Este evento puede responder a cualquier pulsación en el teclado, incluyendo las teclas de función y las combinaciones de

telcas (con Shift, Alt y Ctrl, incluso con botones del ratón). El parámetro `Key` almacena el código *virtual key* de la tecla o combinación que se ha pulsado. El parámetro `Shift` indica si las teclas Shift, Alt o Ctrl forman parte de la combinación.

OnKeyPress

El evento `OnKeyPress` se envía en el momento que se produce la pulsación de cualquier tecla que tenga representación *ASCII*. El parámetro `Key`, de tipo `char`, registra el carácter *ASCII* correspondiente a la tecla pulsada. Si la tecla no posee representación *ASCII* este evento no se envía.

OnPaint

El evento `OnPaint` se envía cuando cualquier control del formulario se tiene que pintar.

OnResize

El evento `OnResize` se envía cuando un control cambia de tamaño (bien por acción del usuario, bien por modificación de sus atributos).

OnShow

El evento `OnShow` se envía cuando un formulario se muestra, es decir, cuando su atributo `Visible` toma el valor `true`.

ANEXO C. Estructura de los ficheros de configuración

En este anexo se detalla el formato de los ficheros de configuración de *ENS2001*. La herramienta consta de dos ficheros de configuración. El primero de ellos (`ens2001.cfg`) almacena las opciones del simulador. El segundo (`wens2001.cfg`) está presente únicamente en la versión gráfica para *Windows*, y almacena la posición y estado de las distintas ventanas que componen la interfaz de *ENS2001*.

- fichero `ens2001.cfg`

  Se trata de un fichero de texto que almacena una lista de variables emparejadas a sus valores correspondientes con el siguiente formato:

```
ModoEjecucion=NORMAL
ModoPila=CRECIENTE
ModoDepuracion=SI
BaseNumerica=DECIMAL
ComprobarPC=NO
ComprobarPila=NO
ReiniciarRegistros=SI
```

  Sólo se almacena un par variable-valor por cada línea. Los valores posibles para cada variable se muestran a continuación (en negrita se marcan los valores por defecto):

ModoEjecucion	**NORMAL**
	PASOAPASO
ModoPila	CRECIENTE
	DECRECIENTE
ModoDepuracion	**SI**
	NO
BaseNumerica	**DECIMAL**
	HEXADECIMAL
ComprobarPC	SI
	NO
ComprobarPila	SI
	NO
ReiniciarRegistros	**SI**
	NO

- fichero `wens2001.cfg`

  Se trata de un fichero de texto que contiene información acerca de la posición (respecto al origen de coordenadas, situado en la esquina superior izquierda de la pantalla), tamaño y visibilidad de cada una de las ventanas de la interfaz gráfica, a saber, `PRINCIPAL`, `CONSOLA`, `REGISTROS`, `MEMORIA`, `PILA` y `FUENTE`, con el siguiente formato para cada una de ellas:

```
VENTANA
Coordenada X de la esquina superior izquierda
Coordenada Y de la esquina superior izquierda
Anchura
Altura
Visibilidad (0=invisible, 1=visible)
```

La primera vez que se arranca la aplicación se genera el fichero con este contenido:

```
PRINCIPAL
246
129
640
423
1
CONSOLA
276
199
473
358
0
REGISTROS
259
177
168
425
0
MEMORIA
590
176
195
300
0
PILA
695
77
170
300
0
FUENTE
310
182
275
300
0
```

ANEXO D. Manual de Usuario

1. Introducción

ENS2001 es una aplicación que integra la función de **Ensamblador** de un subconjunto de instrucciones del estándar *IEEE 694* [De Miguel 1996] y la función de **Simulador**, ya que es capaz de ejecutar programas ensamblados para dicha implementación particular del estándar. Se trata de una mejora de las herramientas disponibles hasta ahora, como son *ASS* y *ENS96*, a las que añade una arquitectura de máquina virtual mejorada, una implementación del lenguaje más homogénea y, principalmente, una nueva y cómoda interfaz gráfica.

El **Ensamblador** permite cargar desde un fichero el código fuente que será ejecutado, informando de la corrección del mismo o de los errores que pudiera contener, según el caso, y como se puede ver en la Figura 1.1.

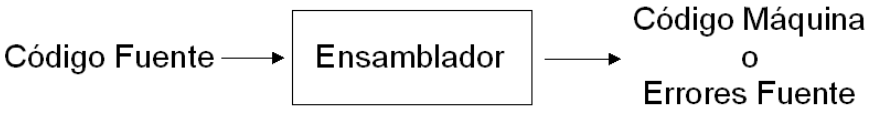


Figura 1.1. El Ensamblador de *ENS2001*

El **Simulador**, aparte de las diversas opciones de configuración y ejecución, proporciona las típicas funciones de acceso a memoria, pila, banco de registros, código fuente y consola, tanto para consultar datos como para alterarlos manualmente. En la Figura 1.2 se muestra el diagrama de bloques simplificado para esta parte de la herramienta.

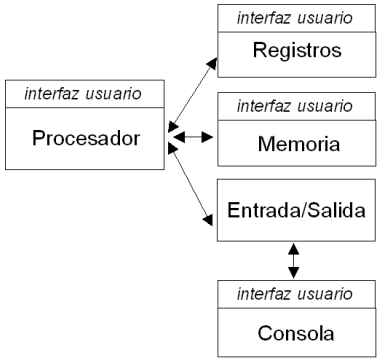


Figura 1.2. El Simulador de *ENS2001*

En principio, la aplicación está dirigida a los alumnos de Compiladores, que deberán probar el código generado por sus respectivas prácticas, si bien también puede ser de utilidad, sobre todo didáctica, para todo aquél que esté interesado en iniciarse en el mundo de la programación en ensamblador, sin profundizar en ningún lenguaje para ningún microprocesador en concreto. En esta herramienta hallará el complemento ideal para comprobar en un entorno simulado la ejecución de los programas que vaya creando durante las etapas de aprendizaje.

A continuación se describe el sistema que la herramienta simula y el lenguaje ensamblador ideado para dicho sistema. Posteriormente se ahondará en los detalles acerca de las características de las dos versiones disponibles de la herramienta, entorno gráfico y

consola. En cada apartado se describe completamente cada versión. De esta manera se evita al usuario tener que leer ambos. Si está interesado únicamente en la versión gráfica, puede ignorar el Apartado 9, mientras que si está interesado solamente en la versión textual, puede ignorar el Apartado 8.

La herramienta se distribuye en tres ficheros comprimidos que contienen cada una de las tres distintas versiones disponibles de manera independiente. Los ficheros están comprimidos en formato zip para las versiones de *Windows* y en un tar.gz para la de *Linux*.

No es necesario ningún tipo de instalación. Basta con descomprimir el fichero de la versión que se desee emplear en un directorio cualquiera del disco duro. También se adjuntan el manual de usuario y algunos programas de ejemplo. Además, la versión para consola *Linux* se distribuye con el código fuente.

2. Máquina Virtual

La Máquina Virtual que simula la aplicación posee las siguientes características:

- **Procesador** con ancho de palabra de 16 bits.
- **Memoria** de 64 Kpalabras (de 16 bits cada una). Por tanto, el direccionamiento es de 16 bits, coincidiendo con el ancho de palabra, desde la dirección 0 a la 65535 (FFFFh).
- **Banco de Registros**. Todos ellos son de 16 bits.
  - **PC** (Contador de Programa): Indica la posición en memoria de la siguiente instrucción que se va a ejecutar.
  - **SP** (Puntero de Pila): Indica la posición de memoria donde se encuentra la cima libre de la pila.
  - **SR** (Registro de Estado): Almacena el conjunto de los biestables de estado, según la Tabla 2.1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	H	S	P	V	C	Z

Tabla 2.1. Biestables de Estado

  - **IX**, **IY** (Registros Índices): Se emplean para efectuar direccionamientos relativos.
  - **A** (Acumulador): Almacena el resultado de las operaciones aritméticas y lógicas de dos operandos.
  - **R0**..**R9** (Registros de Propósito General): Son registros cuyo uso decidirá el programador en cada momento. Los registros se codifican internamente con 4 bits, valores de 0 a 15, según se muestra en la Tabla 2.2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PC	SP	IY	IX	SR	A	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

Tabla 2.2. Registros del Simulador

- **Biestables de estado:** Se actualizan después de que el procesador ha ejecutado una operación aritmética o de comparación. Su significado y condiciones de activación son los siguientes:

  - **Z (cero).** Si el resultado de la operación es 0, se activa. En caso contrario, se desactiva.
  - **C (acarreo).** Si el resultado excede los 16 bits de longitud, se activa. En caso contrario, se desactiva.
  - **V (desbordamiento)**. Si el resultado de la operación excede el rango de representación de los enteros de 16 bits en complemento a 2 (desde –32768 hasta 32767), se activa. En caso contrario, se desactiva.
  - **P (paridad)**. Es el resultado de realizar un O Lógico Exclusivo con todos los bits del resultado. Así, se activa si el número de bits que valen 1 es impar y se desactiva si el número de bits que valen 1 es par.
  - **S (signo)**. Se activa cuando el resultado de la operación es negativo y se desactiva si es positivo. En otras palabras, indica el valor del bit más significativo del resultado de la operación (como la ALU opera en complemento a 2, será 0 para valores positivos y 1 para valores negativos).
  - **H (fin de ejecucion)**. Se activa únicamente después de que el procesador haya ejecutado una instrucción de parada (`HALT`).

- **Entrada/Salida** por consola: Se proveen instrucciones para leer y escribir enteros, caracteres y cadenas de caracteres acabadas en el carácter nulo (‘\0’), según el estilo convencional del lenguaje *C*.

- **Unidad Aritmético-Lógica** sobre enteros en complemento a 2 de 16 bits: El juego de instrucciones proporciona operaciones de suma, resta, multiplicación, división, módulo, cambio de signo, incremento, decremento, *or* lógico, *and* lógico, *or* exclusivo lógico y negación lógica.

  Al operar en complemento a 2, los enteros negativos (en el rango –32768..–1) se transforman en los correspondientes positivos (32768..65535 respectivamente). Por lo tanto, al introducirlos, bien en el código fuente, bien durante la ejecución, serán tratados indistintamente por la herramienta, tanto en el ensamblador como durante la simulación.

- Generación de **Excepciones**: Durante la simulación se pueden producir las siguientes condiciones de excepción, que detendrán la ejecución:

  - *Instrucción no implementada*. Ocurre cuando el procesador lee un código de operación que no corresponde con ninguna instrucción de su juego de instrucciones. Esto puede ocurrir si el usuario introduce código directamente editando la memoria, o no se posiciona el contador de programa adecuadamente antes de lanzar la ejecución, por ejemplo, en zonas de datos o en una dirección de memoria intermedia dentro de una instrucción que ocupe varias posiciones.
  - *División por cero*. Ocurre cuando el segundo operando de una operación de división (el divisor) toma el valor cero.
  - *Sobrepasado el límite de la memoria*. Ocurre cuando, tras ejecutar una instrucción, el Contador de Programa toma un valor más alto que el límite superior de memoria. O bien cuando se está ejecutando una instrucción `INSTR` o `WRSTR` y la cadena sobrepasa dicho límite.

- *Contador de programa invade la zona de pila*. Esta comprobación es opcional, se puede activar o desactivar a voluntad del usuario. Cuando está activada, se genera una excepción si PC va a tomar un valor comprendido dentro de los límites de la pila del sistema.
- *Puntero de pila invade la zona de código*. Esta comprobación es opcional, se puede activar o desactivar a voluntad del usuario. Cuando está activada, se genera una excepción si SP va a tomar un valor comprendido dentro de los límites del código almacenado en memoria.
- *Ejecución detenida por el usuario*. El usuario puede detener la simulación en cualquier momento, generándose esta excepción.

3. Estructura del Código Fuente

El código fuente se lee desde un **fichero de texto plano**. Esta es la forma común de alimentar a la aplicación de programas para ejecutar, si bien es posible introducir el código ensamblado a mano modificando directamente las posiciones de memoria, lo cual resulta bastante más tedioso y peligroso ya que no se comprobará si se está cometiendo algún error (además no se actualizarán los límites de la zona de código). En la Figura 3.1 se muestra la arquitectura del módulo ensamblador. Se parte de un fichero donde está almacenado el código fuente y se genera en dos pasadas el código máquina, o bien una lista de errores en el programa introducido.

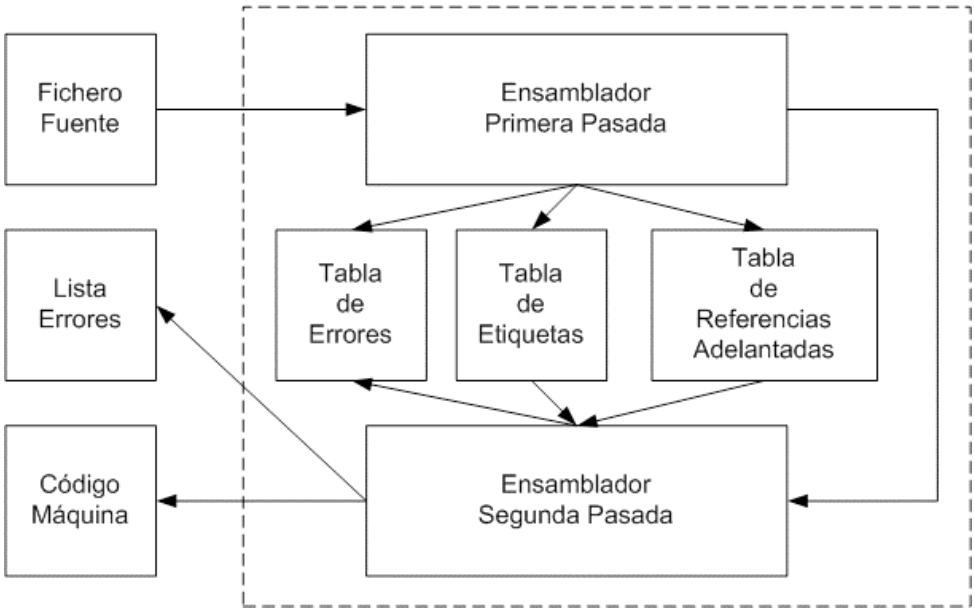


Figura 3.1. Arquitectura del Ensamblador

El código fuente se compone de una sucesión de líneas que obedecen al siguiente formato:

```
[[Etiqueta ':'] Instrucción] [';' Comentario]
```

No existe límite para el número de caracteres que pueda ocupar una línea (salvo las restricciones propias del sistema operativo o la memoria disponible). Son perfectamente posibles combinaciones tales como introducir líneas en blanco, comentarios tras las instrucciones, líneas únicamente con comentarios, etc.

Además, excepcionalmente, se permite introducir saltos de línea y comentarios entre etiquetas e instrucciones, en aras de aumentar la legibilidad del código. La única restricción "real" es que cada instrucción estará contenida en una única línea, ni más ni menos (es decir, una instrucción sólo ocupa una línea y en cada línea sólo hay una instrucción como mucho). Por ejemplo:

```
BZ .R3 ; saltar si el contador es cero          Correcto
; esta línea sólo tiene comentarios             Correcto
ADD #300 ; primer op , [.R0] ; segundo op       Erróneo
```

El ensamblador distingue entre mayúsculas y minúsculas (lo cual tiene especial importancia a la hora de definir etiquetas). En los casos especiales de las instrucciones y los identificadores de registros, serán válidos en uno u otro formato (pero sin mezclar ambos). Por ejemplo:

```
NOP     Correcto
nop     Correcto
Nop     Erróneo (se tomará como una etiqueta)
```

El formato de las instrucciones es el siguiente:

```
Mnemónico [Operando1 [Operando2]]
```

Un mnemónico es una **palabra** o una **abreviatura** en inglés que hace referencia a la operación.

Los operandos pueden ser **enteros**, **nombres de registros** o **etiquetas**. En cualquier caso, se deben ajustar a alguno de los modos de direccionamiento que se analizan en el siguiente apartado. No todas las instrucciones admiten todos los modos de direccionamiento posibles. Las combinaciones permitidas se verán al estudiar con detalle el juego de instrucciones y pueden consultarse en la Tabla 10.5.

4. Modos de Direccionamiento

El estándar *IEEE 694* define un conjunto de modos de direccionamiento para determinar los distintos operandos o su ubicación. El modelo simulado por *ENS2001* permite **siete** de ellos (aunque en realidad son seis, pero uno de ellos se desdobla por comodidad). Dependiendo de cada instrucción, se pueden usar unos u otros en los operandos. Los modos de direccionamiento son los siguientes:

4.1. Direccionamiento Inmediato

El direccionamiento es inmediato cuando el operando se encuentra **contenido** en la propia instrucción. Por tanto, dadas las características de la máquina virtual, internamente serán necesarios 16 bits para este tipo de direccionamiento.

Para indicar este modo de direccionamiento, se empleará el carácter **'#'** (almohadilla) precediendo al operando, que en este caso indicará un valor entero de 16 bits en complemento a 2.

Se pueden introducir valores decimales con signo, en el intervalo comprendido entre –32768 y 32767, o bien sin signo en el rango 0..65535. También se permite introducir valores hexadecimales, precedidos por el prefijo '0x', en el rango 0..FFFFh.

Ejemplos:

- La instrucción de carga `MOVE #1000,.R1`, transfiere el valor 1000d al registro R1, como se muestra en la Figura 4.1.1.

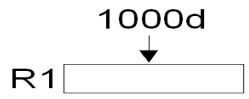


Figura 4.1.1. Direccionamiento Inmediato (ejemplo 1)

- La instrucción de suma `ADD .R3,#0x00FF` suma el contenido de R3 con el valor FFh y lo almacena en el acumulador, tal y como se aprecia en la Figura 4.1.2.

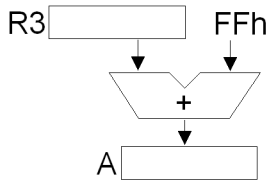


Figura 4.1.2. Direccionamiento Inmediato (ejemplo 2)

4.2. Direccionamiento Directo a Registro

El direccionamiento es directo a registro cuando expresa la **dirección** (en este caso el **nombre del registro**) donde se encuentra almacenado el objeto. Dado el banco de registros definido para la máquina virtual, internemente serán necesarios 4 bits para este tipo de direccionamiento (ya que el banco de registros contiene 16 registros en total).

Para indicar este modo de direccionamiento, se empleará el carácter **'.'** (punto) precediendo al operando, que en este caso será el nombre de un registro del banco de registros.

Por ejemplo, en la instrucción de resta `SUB .R3, #0xFF`, el primer operando emplea un direccionamiento directo a registro, ya que se está indicando el nombre del registro donde se encuentra almacenado. El segundo operando emplea direccionamiento inmediato, de la misma forma que el ejemplo anterior (Figura 4.1.2). Dicha instrucción resta al contenido del registro R3 el valor inmediato FFh, y lo almacena en el acumulador.

4.3. Direccionamiento Directo a Memoria

El direccionamiento es directo a memoria cuando se expresa la **dirección absoluta de memoria** donde se encuentra almacenado el operando. Como la

memoria consta de 64 KPalabras, internamente se necesitan 16 bits para este tipo de direccionamiento.

Para indicar este modo de direccionamiento, se empleará el carácter **'/'** (barra) precediendo al operando, que en este caso indicará una dirección de memoria. Como en el caso del direccionamiento inmediato, se permiten valores decimales con signo, en el intervalo –32768..32767 o bien sin signo en el rango 0..65535. También se permite introducir valores hexadecimales, precedidos por el prefijo '0x', en el rango 0..FFFFh. En el caso de que se introduzcan decimales con signo, se traducirán a la dirección que se correspondería con su representación en complemento a 2 (en 16 bits); por ejemplo, -1 se corresponde con 65535, -2 con 65534 y así sucesivamente.

Por ejemplo, para la instrucción de incremento `INC /0x1000`, el resultado será que el valor contenido en la posición de memoria 1000h se habrá incrementado en una unidad, tal y como se muestra en la Figura 4.3.1.

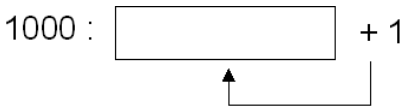


Figura 4.3.1. Direccionamiento Directo a Memoria

En el caso de haber introducido `INC /-30`, por ejemplo, el ensamblador haría la conversión pertinente, y el operando se correspondería con la celda de memoria ubicada en la dirección 65506 (que es la representación en complemento a 2 con 16 bits del valor –30). Por tanto, dicha instrucción es la misma que `INC /65506`.

4.4. Direccionamiento Indirecto

Se trata de un direccionamiento directo, pero del que no se obtiene el operando, sino la **dirección donde se encuentra el operando**. Dicha dirección se encuentra almacenada en un registro. Por tanto, se requerirá un acceso a memoria adicional (el primero al banco de registros y el segundo a memoria principal) para recuperar el objeto. Así las cosas, internamente se requieren 4 bits para indicar cuál es el registro a partir del que se recupera la dirección de memoria destino.

Para indicar este modo de direccionamiento, se encierra entre corchetes **'[ ]'** el identificador del registro sobre el que se efectúa la indirección en notación de registro, esto es, precedido por el carácter '**.**' (punto).

Por ejemplo, la instrucción de decremento `DEC [.R3]` decrementará en una unidad el contenido de la posición de memoria contenida en el registro R3. Se puede ver con claridad en la Figura 4.4.1.

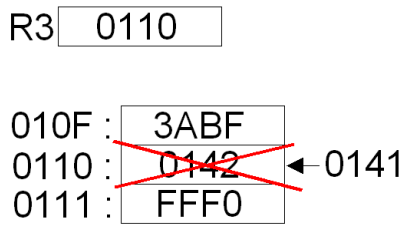


Figura 4.4.1. Direccionamiento Indirecto

4.5. Direccionamiento Relativo a Registro Índice

El direccionamiento es relativo a un registro índice cuando la instrucción contiene un **desplazamiento** sobre una **dirección** marcada por un **puntero**. La dirección del operando, por tanto, se calcula sumando el desplazamiento al puntero de referencia.

Para indicar este modo de direccionamiento, se empleará el carácter **'#'** (almohadilla) precediendo al operando, que en este caso indicará un desplazamiento (entero de 8 bits en complemento a 2, esto es, valores comprendidos entre –128 y 127), y detrás, en **notación de indirección '[ ]'** (entre corchetes), el nombre del registro índice. El valor del desplazamiento se puede introducir, como cualquier otro entero, bien en base decimal, bien en base hexadecimal, siguiendo las mismas consideraciones que en los casos anteriores de direccionamiento inmediato y directo a memoria, atendiendo al detalle adicional de que en este caso sólo se permiten valores de 8 bits.

ENS2001 incluye dos registros IX e IY que actúan como índices, y se van a permitir desplazamientos restringidos a enteros de 8 bits en complemento a 2. Por tanto, internamente, este tipo de direccionamiento requiere de 9 bits, 8 para el desplazamiento y uno adicional para indicar el registro índice. A efectos prácticos, se considerarán el relativo a IX y el relativo a IY como dos modos de direccionamiento distintos.

Ejemplos:

- En la instrucción de carga `MOVE #6[.IX],.R1`, se carga en el registro R1 el contenido de la posición de memoria a la que hace referencia el índice IX más 6 posiciones (Figura 4.5.1).

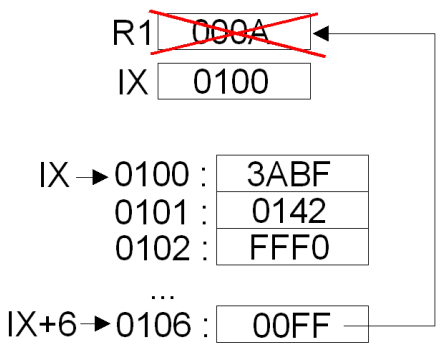


Figura 4.5.1. Direccionamiento Relativo a Índice

- Las siguientes instrucciones son análogas: `DEC #-10[.IY]; DEC #0xF6[.IY]; DEC #246[.IY],` debido a la equivalencia entre las representaciones decimal y hexadecimal en complemento a 2.

4.6. Direccionamiento Relativo a Contador de Programa

Se trata de un caso particular de direccionamiento directo relativo. El registro **puntero** es el **contador de programa (PC)**, y se indica el desplazamiento, que como en el caso general se trata de un valor entero de 8 bits en complemento a 2. Por tanto, este modo de direccionamiento requiere 8 bits para su representación interna.

Así, la dirección del objeto, en este caso, la próxima instrucción que se va a ejecutar, se obtiene sumando el desplazamiento al registro puntero. En la práctica, el desplazamiento indica una cantidad entera que se va a sumar al contador de programa para reubicarlo antes de la ejecución de la siguiente instrucción.

Para indicar este modo de direccionamiento, se empleará el carácter **'$'** (símbolo de dólar) precediendo al operando, que se trata del desplazamiento. El formato del desplazamiento es idéntico al caso general (enteros comprendidos entre –128 y 127).

Por ejemplo, en la instrucción de salto incondicional `BR $3` de la Figura 4.6.1, la siguiente instrucción que ejecutará el procesador se encuentra en la posición de memoria a la que apunte el contador de programa más 3 posiciones. Hay que recordar que el contador de programa apunta siempre a la siguiente instrucción de la que se está ejecutando.

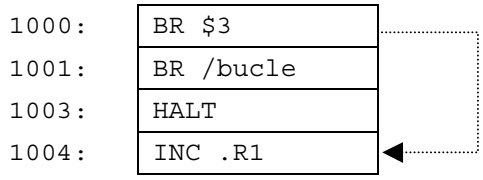


Figura 4.6.1. Direccionamiento Relativo a PC

La siguiente instrucción que se ejecutaría sería `INC .R1` (y no `HALT` como cabría esperar, ya que en el momento de ejecutar `BR $3`, PC vale 1001, así que PC = 1001+3).

En las instrucciones de salto es muy frecuente el empleo de etiquetas, tanto si el direccionamiento es directo a memoria como relativo a contador de programa. No obstante, en el caso de direccionamiento relativo, la etiqueta no se traduce por su valor, si no que se calcula el desplazamiento entre la posición a la que apuntaría PC en el caso de seguir el flujo de ejecución y la posición que ocupa la etiqueta. Por ello, si se introduce en el código fuente un direccionamiento relativo a contador de programa hacia una etiqueta que está alejada más de 128 posiciones de memoria (127 hacia adelante, 128 hacia atrás, cantidades representables con 8 bits), el ensamblador devolverá un error en tiempo de compilación, ya que no puede calcular un desplazamiento válido.

El caso anterior, representado en la Figura 4.6.2, pero empleando etiquetas en vez de números enteros, sería el siguiente. La etiqueta `SEGUIR` tomaría el valor 1004, y el valor del desplazamiento seguiría siendo 3.

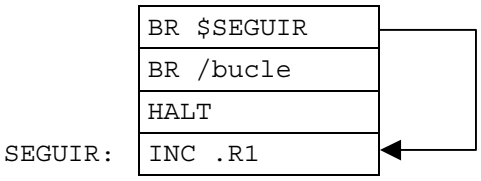


Figura 4.6.2. Direccionamiento Relativo a PC con etiquetas

5. Juego de Instrucciones

A continuación se enumeran todas las instrucciones que componen el juego de instrucciones, ordenadas por código de operación. En la Tabla 10.5 se puede consultar la relación completa de instrucciones y los modos de direccionamiento que soportan.

Instrucción	NOP
Descripción	Instrucción de no operación.
Formato	NOP
Código de Operación	0
Número de Operandos	0
Modos de Direccionamiento	N/A
Comportamiento	Ninguno No modifica los biestables de estado.

Instrucción	HALT
Descripción	Detener la ejecución de la máquina.
Formato	HALT
Código de Operación	1
Número de Operandos	0
Modos de Direccionamiento	N/A
Comportamiento	Activa el biestable H de fin de programa y detiene el procesador virtual

TRANSFERENCIA DE DATOS.

Instrucción	MOVE
Descripción	Copiar operando origen en operando destino.
Formato	MOVE op1, op2
Código de Operación	2
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: registro, memoria, indirecto, relativo Para el operando 2 no se permite direccionamiento inmediato, ya que no tiene sentido como destino de un movimiento de datos.
Comportamiento	Lee el contenido del operando 1 (origen) y lo escribe en el operando 2 (destino). No modifica los biestables de estado.

Instrucción	PUSH
Descripción	Poner en la pila.
Formato	PUSH op1
Código de Operación	3
Número de Operandos	1

Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Depende del modo de funcionamiento de la pila: • Para crecimiento hacia direcciones descendentes de memoria, almacena el contenido del operando 1 en la dirección apuntada por SP y decrementa el valor del puntero de pila. • Para crecimiento hacia direcciones ascendentes de memoria, incrementa el valor del puntero de pila y almacena el contenido del operando 1 en la dirección apuntada por SP. No modifica los biestables de estado.

Instrucción	POP
Descripción	Sacar de la pila.
Formato	POP op1
Código de Operación	4
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo Para el operando no se permite direccionamiento inmediato, ya que no tiene sentido como destino de un movimiento de datos.
Comportamiento	Depende del modo de funcionamiento de la pila: • Para crecimiento hacia direcciones descendentes de memoria, incrementa el valor del puntero de pila y almacena en el operando 1 el valor contenido en la dirección de memoria apuntada por SP. • Para crecimiento hacia direcciones crecientes de memoria, almacena en el operando 1 el valor contenido en la dirección de memoria apuntada por SP y decrementa el valor del puntero de pila. No modifica los biestables de estado.

ARITMÉTICAS.

De dos operandos.

Instrucción	ADD
Descripción	Suma de números enteros.
Formato	ADD op1, op2
Código de Operación	5
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Suma el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. Modifica los biestables de estado.

Instrucción	SUB
Descripción	Resta de números enteros.
Formato	SUB op1, op2
Código de Operación	6
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Resta el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. Modifica los biestables de estado.

Instrucción	MUL
Descripción	Producto de números enteros.
Formato	MUL op1, op2
Código de Operación	7
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Multiplica el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. Modifica los biestables de estado.

Instrucción	DIV
Descripción	Cociente de la división de números enteros.
Formato	DIV op1, op2
Código de Operación	8
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Divide el contenido del operando 1 y el operando 2, y almacena el cociente de la operación en el registro acumulador. Si el operando 2 contiene el valor 0, se genera una excepción de tipo “división por cero”. Modifica los biestables de estado.

Instrucción	MOD
Descripción	Resto de la división de números enteros.
Formato	DIV op1, op2
Código de Operación	9
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Divide el contenido del operando 1 y el operando 2, y almacena el resto de la operación en el registro acumulador. Si el operando 2 contiene el valor 0, se genera una excepción de tipo “división por cero”. Modifica los biestables de estado.

De un operando.

Instrucción	INC
Descripción	Incremento unitario.
Formato	INC op1
Código de Operación	10
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Incrementa el contenido del operando en una unidad. Modifica los biestables de estado.

Instrucción	DEC
Descripción	Decremento unitario.
Formato	DEC op1
Código de Operación	11
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Decrementa el contenido del operando en una unidad. Modifica los biestables de estado.

Instrucción	NEG
Descripción	Cambio de signo.
Formato	NEG op1
Código de Operación	12
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Cambia de signo el operando. Modifica los biestables de estado.

Comparaciones.

Instrucción	CMP
Descripción	Comparación.
Formato	CMP op1, op2
Código de Operación	13
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Resta el contenido del operando 1 y el operando 2 (pero no almacena el resultado de la operación en ningún sitio). Modifica los biestables de estado.

LÓGICAS.

De dos operandos.

Instrucción	AND
Descripción	Y Lógico.
Formato	AND op1, op2
Código de Operación	14
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'y lógico' bit a bit entre el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. La tabla de verdad de la operación es: op1 AND op2 | 0 | 1 0 | 0 | 0 1 | 0 | 1 No modifica los biestables de estado.

Instrucción	OR
Descripción	O Lógico.
Formato	OR op1, op2
Código de Operación	15
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo

Comportamiento	Efectúa la operación 'o lógico' bit a bit entre el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. La tabla de verdad de la operación es: op1 OR op2 | 0 | 1 0 | 0 | 1 1 | 1 | 1 No modifica los biestables de estado.

Instrucción	XOR
Descripción	O Exclusivo Lógico.
Formato	XOR op1, op2
Código de Operación	16
Número de Operandos	2
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo op2: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'o exclusivo lógico' bit a bit entre el contenido del operando 1 y el operando 2, y almacena el resultado en el registro acumulador. La tabla de verdad de la operación es: op1 XOR op2 | 0 | 1 0 | 0 | 1 1 | 1 | 0 No modifica los biestables de estado.

De un operando.

Instrucción	NOT
Descripción	Negación Lógica.
Formato	NOT op1
Código de Operación	17
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Efectúa la operación 'negación lógica' bit a bit en el operando 1. La tabla de verdad de la operación es: op1 | NOT op1 0 | 1 1 | 0 No modifica los biestables de estado.

BIFURCACIONES.

Todas ellas producen un salto a la posición de memoria indicada por op1, dependiendo de la instrucción y el contenido de los biestables de estado.

Instrucción	BR
Descripción	Bifurcación incondicional.
Formato	BR op1
Código de Operación	18
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Carga el PC con el valor contenido del operando 1.

Instrucción	BZ
Descripción	Bifuración si resultado igual cero.
Formato	BZ op1
Código de Operación	19
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable Z está activo (Z=1), carga el PC con el valor contenido del operando 1.

Instrucción	BNZ
Descripción	Bifurcación si resultado distinto de cero.
Formato	BNZ op1
Código de Operación	20
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable Z está inactivo (Z=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BP
Descripción	Bifurcación si resultado positivo.
Formato	BP op1
Código de Operación	21
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable S está inactivo (S=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BN
Descripción	Bifurcación si resultado negativo.
Formato	BN op1
Código de Operación	22
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable S está activo (S=1), carga el PC con el valor contenido en el operando 1.

Instrucción	BV
Descripción	Bifuración si hay desbordamiento.
Formato	BV op1
Código de Operación	23
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable V está activo (V=1), carga el PC con el valor contenido en el operando 1.

Instrucción	BNV
Descripción	Bifurcación si no hay desbordamiento.
Formato	BNV op1
Código de Operación	24
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable V está inactivo (V=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BC
Descripción	Bifurcación si hay acarreo.
Formato	BC op1
Código de Operación	25
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable C está activo (C=1), carga el PC con el valor contenido en el operando 1.

Instrucción	BNC
Descripción	Bifurcación si no hay acarreo.
Formato	BNC op1
Código de Operación	26
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable C está inactivo (C=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BE
Descripción	Bifurcación si el resultado tiene paridad par.
Formato	BE op1
Código de Operación	27
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable P está inactivo (P=0), carga el PC con el valor contenido en el operando 1.

Instrucción	BO
Descripción	Bifurcación si el resultado tiene paridad impar.
Formato	BO op1
Código de Operación	28
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Si el biestable P está activo (P=1), carga el PC con el valor contenido en el operando 1.

CONTROL DE SUBRUTINAS.

Instrucción	CALL
Descripción	Llamada a subrutina.
Formato	CALL op1
Código de Operación	29
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, relativo a contador de programa, indirecto
Comportamiento	Almacena en la pila el valor del contador de programa y salta a la dirección de destino indicada por el operando 1.

Instrucción	RET
Descripción	Retorno de subrutina.
Formato	RET
Código de Operación	30
Número de Operandos	0
Modos de Direccionamiento	N/A
Comportamiento	Rescata de la pila el contenido del contador de programa.

ENTRADA/SALIDA.

Instrucción	INCHAR
Descripción	Leer un carácter.
Formato	INCHAR op1
Código de Operación	31
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Lee un carácter de la consola y lo codifica en *ASCII*, dejando a cero los 8 bits superiores de la palabra de 16 bits. Almacena el carácter leído en el operando 1.

Instrucción	ININT
Descripción	Leer un entero.
Formato	ININT op1
Código de Operación	32
Número de Operandos	1
Modos de Direccionamiento	op1: registro, memoria, indirecto, relativo
Comportamiento	Lee un entero de la consola. Si la entrada no puede ser convertida en un número entero (porque se sale de rango o no es un número), la instrucción supondrá que se ha leído un cero. Los enteros se pueden introducir en formato decimal o hexadecimal (precedidos por '0x').

Instrucción	INSTR
Descripción	Leer una cadena.
Formato	INSTR op1
Código de Operación	33
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, indirecto, relativo
Comportamiento	Lee una cadena de caracteres de la consola y la almacena en posiciones consecutivas de memoria a partir de la dirección indicada por el operando 1, acabada con el carácter '\0'. No se comprueba previamente si hay espacio para almacenar la cadena, por lo que se puede producir una excepción si se sobrepasa el límite superior de la memoria.

Instrucción	WRCHAR
Descripción	Escribir un carácter.
Formato	WRCHAR op1
Código de Operación	34
Número de Operandos	1
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Escribe en la consola el carácter cuyo valor ASCII se corresponde con los 8 bits inferiores del valor del operando 1.

Instrucción	WRINT
Descripción	Escribir un entero.
Formato	WRINT op1
Código de Operación	35
Número de Operandos	1
Modos de Direccionamiento	op1: inmediato, registro, memoria, indirecto, relativo
Comportamiento	Escribe en la consola el valor del operando 1. El formato de escritura (decimal o hexadecimal), dependerá de la configuración de visualización de números enteros. En cualquier caso, el formato decimal siempre se escribe con signo (-32768 a 32767).

Instrucción	WRSTR
Descripción	Escribir una cadena.
Formato	WRSTR op1
Código de Operación	36
Número de Operandos	1
Modos de Direccionamiento	op1: memoria, indirecto, relativo
Comportamiento	Escribe en la consola una cadena de caracteres, que estará almacenada en memoria a partir de la dirección indicada por el operando 1. Se escribirán caracteres hasta que se llegue al carácter '\0'. Por tanto, puede ocurrir que durante este proceso se genere una excepción si no se encuentra el carácter de fin de cadena antes de llegar al límite superior de la memoria.

6. Macroinstrucciones del Ensamblador

Las macroinstrucciones (también conocidas como pseudoinstrucciones) son órdenes dirigidas al programa ensamblador. Sirven para indicarle cómo se quiere que realice la traducción del programa, pero no forman parte de él. Sin embargo, algunas pseudoinstrucciones sirven para reservar posiciones de memoria, destinadas a los datos.

Algunas de ellas van seguidas de una *expresión*. En este contexto, se entiende que una expresión consta de enteros (en base decimal o hexadecimal) combinados con los operadores clásicos: suma, resta, producto, división, módulo y paréntesis, y con las reglas de precedencia de operadores habituales. En el caso de la división, el resultado se trunca (3/2=1). Por ejemplo:

```
(3+7*(16-1)/(4-1))%27          (el resultado es 11)
```

1. ORG expresión

Ensambla el código a partir de la posición de memoria resultado de evaluar la expresión. Se pueden usar macroinstrucciones `ORG` a lo largo del código fuente todas las veces que se precise. Sin embargo, si no se emplean con orden, se pueden obtener resultados inesperados en el código máquina. Por ejemplo, en este fragmento de código:

```
            ORG 0
bloque_1:   MOVE .R2,.R1
            ...
            ...
            ORG 0
bloque_2:   PUSH #0xFF
            ...
```

Se lee la pseudoinstrucción `ORG 0`, por tanto, la etiqueta `bloque_1` tomará el valor 0 y se ensamblarán las siguientes instrucciones a partir de dicha dirección. Luego se vuelve a leer otra pseudoinstrucción `ORG 0`, y nuevamente, la etiqueta `bloque_2` tomará el valor 0 y se ensamblarán las siguientes instrucciones a partir de dicha dirección, sobrescribiendo a las del primer bloque que se ensambló.

Si no se incluye ninguna pseudoinstrucción `ORG` en el código fuente, se comenzará a ensamblar a partir de la posición de memoria 0.

2. EQU expresión
Equivalencia. Sirve para dar valor numérico a una etiqueta, empleándola de la siguiente forma:

```
etiqueta: EQU expresión
```
(etiqueta=expresión)

Por tanto, cada vez que aparezca la etiqueta como operando en una instrucción, será sustituida por el valor que ha tomado en la pseudoinstrucción `EQU`. Si no se indica la etiqueta, la expresión se calcula, pero su valor se pierde.

3. END
Marca el final del código. Después de la macroinstrucción `END`, el ensamblador da por finalizado el proceso de traducción, ignorando cualquier instrucción posterior.

4. RES expresión
Reserva tantas posiciones de memoria como indica el resultado de evaluar la expresión. Para marcar el inicio de la zona de memoria, se emplea de la siguiente forma:

```
etiqueta: RES expresión
```
(etiqueta toma el valor de la primera posición reservada)

Por tanto, cada vez que aparezca la etiqueta como operando en una instrucción, será sustituida por el valor que ha tomado en la pseudoinstrucción `RES`. Si no se indica ninguna etiqueta, la zona de memoria se reservará igualmente, pero se pierde la referencia a su dirección de comienzo.

5. DATA a, b, c...
Define un conjunto de datos en memoria. Los datos a, b, c... pueden ser enteros (en decimal o hexadecimal) o cadenas de caracteres delimitadas entre comillas dobles. Para marcar el inicio de la zona de datos se emplea de la siguiente forma:

```
etiqueta: DATA a, b, c...
```
(etiqueta toma el valor de la primera posición de la zona de datos)

Por tanto, cada vez que aparezca la etiqueta como operando en una instrucción, será sustituida por el valor que ha tomado en la pseudoinstrucción `DATA`. Si no se indica ninguna etiqueta, los datos se almacenarán en memoria igualmente, pero se pierde la referencia al primero de ellos.

Las cadenas de caracteres definidas en esta pseudoinstrucción pueden contener los caracteres especiales '\0' (carácter nulo), '\t' (tabulador) y '\n' (fin de línea), y cualquiera de los caracteres imprimibles.

Un posible ejemplo de uso de esta pseudoinstrucción sería:

```
ORG 0
DATA "texto\n", 33
END
```

Este listado fuente produce el resultado que se observa en la Figura 6.1.

0000 :	't'
0001 :	'e'
0002 :	'x'
0003 :	't'
0004 :	'o'
0005 :	'\n'
0006 :	'\0'
0007 :	33d

Figura 6.1. Resultado de la pseudoinstrucción DATA

7. Detección de Errores en el Código Fuente

Tras el proceso de ensamblado, si todo ha ido bien, la herramienta habrá ubicado en memoria el código máquina correspondiente al fuente introducido. Sin embargo, si había errores en el listado fuente, es el momento en que se comunicarán al usuario. *ENS2001* presenta una lista de errores con este formato:

```
Línea m: ERROR[n] Descripción del error – Token que causó el error
```

Donde `m` es el número de línea donde se encontró el error y `n` es el código del error. A continuación se muestra una somera descripción y, si procede, la entrada que produjo el error. Sólo se muestra un error por línea (aunque existan varios). Obviamente, si hay errores no se genera código máquina (y no se altera el contenido de la memoria que existiera antes del ensamblado).

Estos son los errores detectados por *ENS2001*:

- **Error 01**. *Modo de direccionamiento del Operando 1 erróneo*.
  El modo de direccionamiento del primer operando no se encuentra dentro de los permitidos para la instrucción. Ejemplo:

  ```
  CALL #10[.IX]
  ```

- **Error 02**. *Modo de direccionamiento del Operando 2 erróneo*.
  El modo de direccionamiento del segundo operando no se encuentra dentro de los permitidos para la instrucción. Ejemplo:

  ```
  MOVE .R2,#1000
  ```

- **Error 03**. *Instrucción no reconocida*.
  La instrucción introducida no pertenece al juego de instrucciones de la máquina virtual. Ejemplo:

  ```
  JP $bucle
  ```

- **Error 04**. *Operando 1 incorrecto*.
  El primer operando introducido no es un operando válido, pero sí que es un *token* del lenguaje. Ejemplo:

  ```
  WRINT 33
  ```

- **Error 05**. *Operando 2 incorrecto*.
  El segundo operando introducido no es un operando válido, pero sí que es un *token* del lenguaje. Ejemplo:

```
MOVE .r4,"cadena"
```

- **Error 06**. *Etiqueta duplicada*.
  La etiqueta ya se había definido previamente en el código. Ejemplo:

```
datos: DATA 1,2,3,4,5
...
datos: MOVE #0,.r5
```

- **Error 07**. *Etiqueta no definida*.
  La etiqueta usada como operando no se ha definido en ningún lugar del código. Ejemplo:

```
CALL /rutina
```

- **Error 08**. *Entrada incorrecta*.
  Se ha encontrado un *token* no válido en el código fuente. Ejemplo:

```
símbolo: EQU 10
```

- **Error 09**. *Expresión errónea*.
  La expresión no está bien formada, contiene operadores u operandos no válidos, o los paréntesis no están balanceados. Ejemplo:

```
datos: RES 10*(2-(256/4)
```

- **Error 10**. *Pseudoinstrucción ORG define el comienzo del código fuera de la memoria*.
  El comienzo del código viene definido por una expresión cuyo valor excede el límite superior de la memoria (65535d). Ejemplo:

```
ORG 65535+1
```

- **Error 11**. *Pseudoinstrucción RES reserva espacio fuera de la memoria*.
  La cantidad de espacio reservado por la pseudoinstrucción viene definida por una expresión cuyo valor excede el límite superior de la memoria a partir de la posición actual de ensamblado. Ejemplo:

```
RES 32768*2
```

- **Error 12**. *Pseudoinstrucción DATA ubica datos fuera de la memoria*.
  La ubicación de la lista de datos ha excedido el límite superior de la memoria. Ejemplo:

```
ORG 65535
DATA "p"
```

- **Error 13**. *Expresión fuera de rango*.
  El valor de la expresión excede la representación de 16 bits. Ejemplo:

```
etiqueta: EQU 32768*2
```

- **Error 14**. *Nombre de etiqueta reservado*.
  Se ha utilizado como etiqueta el mnemónico de una instrucción del procesador. Ejemplo:

```
BNZ /MOVE
```

- **Error 15**. *Entero fuera de rango*.
  Se ha introducido como operando un entero que excede la representación en 16 bits o bien en 8 bits para los modos de direccionamiento relativos a índices y a PC. Ejemplo:

```
WRINT 65536
```

- **Error 16**. *Se esperaba el Operando 1*.
  No se encontró el primer operando. Ejemplo:

```
CALL
```

- **Error 17**. *Se esperaba el Operando 2*.
  No se encontró el segundo operando. Ejemplo:

```
ADD #-10[.IY],
```

- **Error 18**. *Se esperaba carácter de fin de línea*.
  No se ha encontrado el carácter fin de línea donde debería estar. Ejemplo:

```
MUL .R1,.R2,.R3
```

- **Error 19**. *Se esperaba carácter separador*.
  No se ha encontrado el separador (coma) entre los dos operandos de una instrucción. Ejemplo:

```
DIV #0x7000 [.R0]
```

- **Error 20**. *Lista de datos errónea*.
  La lista de datos que acompaña a la pseudoinstrucción *equ* no se compone de enteros o cadenas encerradas entre comillas dobles, separados por comas. Ejemplo:

```
datos: DATA 'cadena',1,2,3
```

- **Error 21**. *Desplazamiento fuera de rango*.
  El desplazamiento viene indicado por el valor de una etiqueta que excede los 8 bits permitidos. Ejemplo:

```
bucle: RES 256
...
BR $bucle
```

- **Error 22**. *Error en asignación de memoria*.
  Ha habido un error de asignación de espacio en memoria para las estructuras internas del ensamblador. El usuario debería salir de la aplicación.

8. Interfaz Gráfica

En la Figura 8.1 se muestra la ventana principal de la aplicación *ENS2001* en su versión *Windows*. A partir de aquí, se efectuará un repaso exhaustivo por todas las funcionalidades que ofrece.

Para ejecutar la aplicación, se hará a través del archivo `wens2001.exe`.

Se observan varias partes bien diferenciadas.

- **Menú de la aplicación**. A través de él se puede acceder a la totalidad de opciones que presenta la herramienta.
- **Barra de botones** con los comandos más comunes.
- **Cuadro de mensajes**. En él aparecerán los mensajes de error o confirmación de operaciones que la herramienta vaya generando a lo largo de la sesión.
- **Barra de estado.** Ofrece una descripción del botón activo, cuando se está a la espera de una orden del usuario, o bien indica qué operación se está procesando.

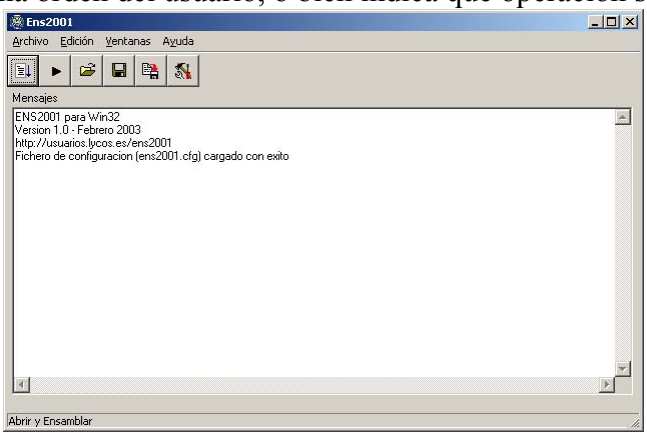


Figura 8.1. Ventana Principal *ENS2001-Win32*

Además de esta ventana principal, la aplicación dispone de otras cinco ventanas flotantes, accesibles desde el menú o con las combinaciones de acceso rápido (Ctrl+F1 a Ctrl+F5) y una sexta ventana donde se muestran las opciones de configuración.

8.1. Menú de la Aplicación

El menú principal de la aplicación (Figura 8.1.1) consta de cuatro opciones, que conforman cuatro submenús:

Figura 8.1.1. Menú Principal

- Archivo.
- Edición.
- Ventanas.
- Ayuda.

Submenú Archivo.

El submenú Archivo (Figura 8.1.2) agrupa las opciones relativas al manejo principal de la herramienta. Son las siguientes:

Figura 8.1.2. Menú Archivo

Abrir y Ensamblar (atajo Ctrl+E).

Con este comando, la herramienta abre un cuadro de diálogo para que el usuario seleccione un fichero.

Una vez seleccionado, lee el código fuente contenido en él, lo ensambla y, si el proceso de ensamblado fue correcto, coloca el código en memoria. Si no, muestra en el cuadro de mensajes un listado con los errores producidos.

Al ensamblar un fichero, se crean los archivos temporales `memoria.tmp` (contenido de la memoria tras ensamblar el código fuente) y `errores.tmp` (listado de los errores producidos al ensamblar, si los hubiera), que se eliminarán tras el proceso. No obstante, si se produce algún error en la creación o acceso a estos archivos, no se podrá completar la operación. Posiblemente habrá que salir de la aplicación y solucionar el error (liberando espacio en disco, porque se haya ejecutado *ENS2001* desde una unidad de red con acceso exclusivo de escritura o un *CD-ROM*, etc.).

Ejecutar (atajo Ctrl+J).

Invoca al procesador virtual para lanzar la simulación. El procesador ejecutará instrucciones hasta que se cumpla una de las siguientes condiciones:

- Ejecución Paso a Paso activa (se detiene nada más ejecutar una instrucción).
- Punto de Ruptura Activado (se detiene antes de ejecutar la instrucción).
- Excepción durante la ejecución.
- Instrucción `HALT`

Al finalizar la simulación, la herramienta mostrará en el cuadro de mensajes cuál de estas causas fue la que detuvo al procesador.

Si durante la ejecución el procesador encuentra una instrucción de entrada o salida por consola, se mostrará la ventana de la consola y esperará a la introducción del dato por parte del usuario para continuar. En el caso de que el procesador se encuentre esperando por un dato de entrada, el usuario podrá detener la ejecución pulsando la tecla ESCAPE.

Cargar Imagen de Memoria (atajo Ctrl+L).

Este comando permite recuperar el contenido íntegro de la memoria desde un fichero en el que se habrá guardado previamente. La herramienta mostrará un cuadro de diálogo donde el usuario seleccionará el fichero que desee cargar.

Como la lectura se hace en formato binario, si se intenta leer un fichero cualquiera que no haya sido generado previamente mediante el comando *Guardar Imagen de Memoria*, los resultados son inesperados.

Guardar Imagen de Memoria (atajo Ctrl+S).

Este comando permite guardar el contenido íntegro de la memoria en un fichero. La herramienta mostrará un cuadro de diálogo para que el usuario escoja el nombre del fichero.

Si no existe, se creará un nuevo fichero con una longitud de 128 Kbytes, y formato *big-endian*. En caso de que existiera, la aplicación preguntará si se desea sobrescribir o bien da la opción de introducir otro nombre.

NOTA: *big-endian* es una forma de almacenar los números de 16 bits en palabras de 8 bits, de forma que primero se almacena la palabra más significativa, y después la menos significativa. La forma contraria de hacerlo se denomina *little-endian*.

Guardar Sesión (atajo Ctrl+G).

Este comando, que es el único que sólo está disponible para la versión gráfica de la aplicación, permite guardar en un fichero el contenido del cuadro de mensajes y la consola. Para ello, como es habitual, presentará un cuadro de diálogo donde el usuario puede escoger en qué fichero desea guardar la sesión.

Igual que en la operación anterior, si el fichero ya existe, la herramienta dará a elegir entre sobrescribirlo o introducir un nombre nuevo.

Configuración (atajo Ctrl+F).

Este comando se estudiará en profundidad al hablar de la ventana de configuración. Muestra dicha ventana, en la que se presentan las diferentes opciones de configuración de la herramienta.

Salir (atajo Ctrl+X).

Este comando sirve para salir de la aplicación. Tiene el mismo efecto que pulsar el icono de cerrar (X) en la esquina superior derecha de la ventana principal. Al salir, tras pedir confirmación al usuario, se guardará la configuración actual y la posición y estado de las ventanas flotantes.

Submenú Edición.

Este submenú (Figura 8.1.3), que también es activable haciendo clic con el botón derecho del ratón sobre el cuadro de mensajes, ofrece cuatro sencillas opciones de edición para dicho cuadro:

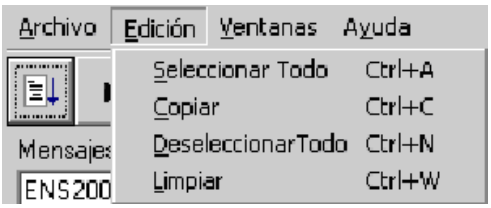


Figura 8.1.3. Menú Edición

Seleccionar Todo (atajo Ctrl+A).

Selecciona todo el texto contenido en el cuadro de mensajes.

Copiar (atajo Ctrl+C).

Copia en el portapapeles el texto del cuadro de mensajes que se encuentre seleccionado en ese momento.

Deseleccionar Todo (atajo Ctrl+N).

Descarta la selección actual del cuadro de mensajes.

Limpiar (atajo Ctrl+W).

Borra el contenido del cuadro de mensajes.

Submenú Ventanas.

En este submenú (Figura 8.1.4) el usuario puede mostrar u ocultar las cinco ventanas flotantes de las que dispone la herramienta. También puede conseguirse el mismo efecto mediante los atajos de teclado. En el menú se muestra un *tick* a la izquierda de aquellas ventanas que estén visibles (en la figura sería la ventana de consola).

Figura 8.1.4. Menú Ventanas

- **Código Fuente** (atajo Ctrl+F1).
- **Consola** (atajo Ctrl+F2).
- **Memoria** (atajo Ctrl+F3).

- **Registros** (atajo Ctrl+F4).
- **Pila** (atajo Ctrl+F5).

Haciendo clic en cada opción se muestra la ventana si estaba oculta, o se oculta si era visible. También es posible ocultar las ventanas haciendo clic en su icono de cerrar respectivo, en la esquina superior derecha de todas ellas.

Más adelante se describe con detenimiento cada una de las ventanas. Tanto la posición como el tamaño y el estado de las ventanas se guarda en el fichero `wens2001.cfg` cada vez que se finalice el uso de la herramienta, y se intentará recuperar cuando se inicie la siguiente sesión. Si no se puede recuperar el fichero, se creará de nuevo con los valores predeterminados.

Submenú Ayuda.

Figura 8.1.5. Menú Ayuda

Como se puede ver en la Figura 8.1.5, sólo contiene una opción, **Acerca de** (atajo Alt+D), que muestra información acerca de la versión de *ENS2001* que se está ejecutando, la fecha de compilación, y la *URL* del proyecto (Figura 8.1.6).

Figura 8.1.6. Cuadro de Diálogo Acerca de

8.2. Barra de Botones

La barra de botones sirve de acceso rápido a las acciones más frecuentes de la herramienta. Su comportamiento es análogo a seleccionar la misma acción desde el menú de la aplicación. Son los siguientes (de izquierda a derecha):

Abrir y Ensamblar

Ejecutar

Cargar Imagen de Memoria

Guardar Imagen de Memoria

Guardar Sesión

Configuración

El botón Ejecutar cambiará de icono y de función, dependiendo de las siguientes condiciones:

- Cuando haya un programa ejecutándose, su función será **Detener** la ejecución, y se mostrará de esta forma:

- Cuando en la configuración esté seleccionado el Modo de Ejecución **Paso a Paso**, tendrá esta forma:

8.3. Cuadro de Mensajes

En esta ventana (Figura 8.3.1) se irán mostrando los distintos mensajes generados por la herramienta, como consecuencia del proceso de ensamblado o ejecución. Ejemplos de estos mensajes pueden ser el número de líneas procesadas por el ensamblador, los errores producidos (caso de que existan), o la condición de detención del simulador tras la ejecución.

Figura 8.3.1. Cuadro de Mensajes

Además, como se comentó anteriormente, haciendo clic con el botón derecho del ratón, se puede acceder al menú de edición, pero esta vez de forma flotante.

8.4. Ventana de Código Fuente

Se trata de una ventana en la que el usuario tiene acceso al contenido de la memoria, pero una vez desensamblado, con lo que se puede leer fácilmente el código contenido en ella, como se aprecia en la Figura 8.4.1. El contenido de la ventana se actualiza automáticamente tras ensamblar con éxito un programa fuente y tras concluir la ejecución de un programa contenido en memoria. En este último caso, además, la ventana mostrará el código ensamblado a partir de la posición a la que apunte el contador de programa en ese instante.

El tamaño de la ventana es variable, por lo que el número de instrucciones mostradas dependerá del mismo.

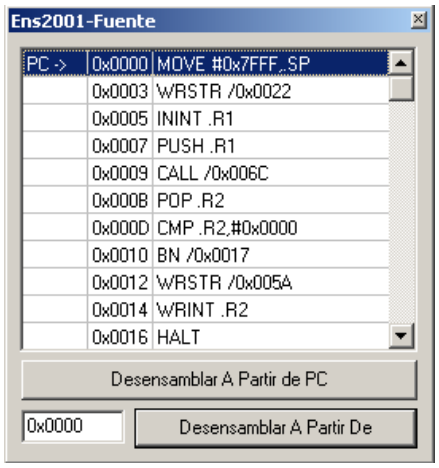


Figura 8.4.1. Ventana de Código Fuente

Hay que reseñar que, debido a que todas las instrucciones no ocupan el mismo tamaño en memoria (las hay de 1, 2 y 3 palabras de longitud), se pueden obtener resultados erróneos al desensamblar a partir de una dirección arbitraria. También puede leerse código extraño al desensamblar las zonas de datos de los programas, por ejemplo. Para evitarlo, se debería desensamblar siempre a partir del comienzo del código o bien a partir de la posición ocupada por el contador de programa o una instrucción desensamblada anteriormente.

Si se desplaza la ventana hacia abajo una posición, se saltará una instrucción, mientras que si se desplaza hacia arriba, se saltará una posición de memoria. Esto es así porque, partiendo de una posición cualquiera, no es posible saber en qué dirección comienza la instrucción anterior. Por ejemplo, si en la ventana de la Figura 8.4.1, el usuario presiona la flecha hacia abajo, la primera posición de memoria pasará a ser la 3, pero si a continuación presiona la flecha hacia arriba, no volverá a la 0, sino que la primera será la 2.

Para desensamblar código a partir de una dirección arbitraria, se puede introducir dicha dirección de comienzo en el cuadro de texto situado al lado del botón **Desensamblar A Partir De**, y a continuación pulsar dicho botón. Para desensamblar a partir de la dirección a la que apunta el contador de programa (PC), se pulsará el botón **Desensamblar A Partir De PC**.

En esta ventana también se pueden activar y desactivar los puntos de ruptura incondicionales, que servirán de ayuda al usuario a la hora de depurar sus programas. Para ello, basta con hacer doble clic en la posición de memoria que se quiera editar, y se irá alternando entre punto de ruptura activado y desactivado. Los puntos de ruptura activos se marcan con un asterisco "*" en la columna de la izquierda, como se observa en la Figura 8.4.2, en la que hay un punto de ruptura activo en la dirección 0009h.

Figura 8.4.2. Marca de Punto de Ruptura Activado

Como información adicional, se puede consultar la posición a la que apunta el contador de programa. Se trata del indicador "PC →" que aparecerá en la columna de la izquierda. En la Figura 8.4.3, al desensamblar esta zona de la memoria, el contador de programa apuntaba a la dirección 0007h.

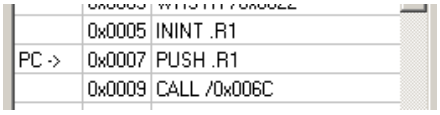


Figura 8.4.3. Indicador de contador de programa

8.5. Ventana de Consola

A través de esta ventana (Figura 8.5.1) se efectuarán las operaciones de entrada/salida durante la simulación. En ella se visualizará el resultado de las instrucciones de salida (`WRCHAR`, `WRINT` y `WRSTR`) y en ella el usuario deberá introducir los datos de entrada para las instrucciones correspondientes (`INCHAR`, `ININT` e `INSTR`) cuando se ejecute alguna de ellas.

Aunque esté oculta, en el momento en que se ejecute cualquiera de las seis instrucciones citadas, la ventana de consola pasará a un primer plano. Además, si se trata de una instrucción de entrada, la ejecución se detendrá hasta que el usuario introduzca el dato solicitado, tras lo que continuará la ejecución, o bien pulse la tecla ESCAPE, con lo que se generará la excepción *Ejecución Detenida por el Usuario*.

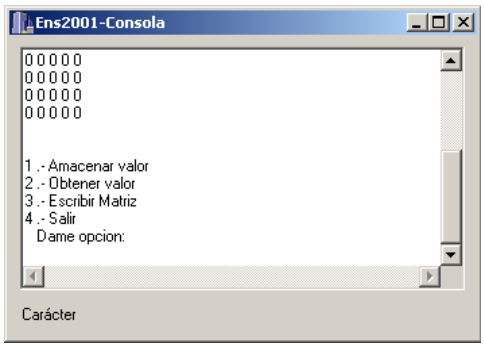


Figura 8.5.1. Consola

Cuando se esté ejecutando una instrucción de entrada de datos, en la parte inferior izquierda de la ventana aparecerá una indicación sobre el tipo de datos que se está leyendo (carácter, entero o cadena). En la Figura 8.5.1 se está ejecutando una instrucción `INCHAR`.

La consola consta de un menú de edición análogo al del cuadro de mensajes, accesible haciendo clic con el botón derecho del ratón encima de la misma, tal y como se observa en la Figura 8.5.2. Ofrece las siguientes opciones:

Seleccionar Todo (atajo Ctrl+A).

Selecciona todo el texto contenido en la consola.

Copiar (atajo Ctrl+C).

Copia en el portapapeles el texto de la consola que se encuentre seleccionado en ese momento.

Deseleccionar Todo (atajo Ctrl+N).

Descarta la selección actual de la consola.

Limpiar (atajo Ctrl+W).

Borra el contenido de la consola.

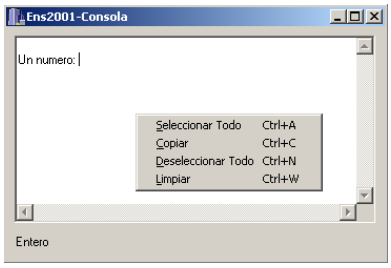


Figura 8.5.2. Menú Edición de la consola

8.6. Ventana de Memoria

En esta ventana (Figura 8.6.1) es posible consultar el valor de tantas celdas de memoria como quepan en ella, dependiendo de su tamaño. También es posible editar el contenido de las celdas, haciendo doble clic en cualquiera de ellas.

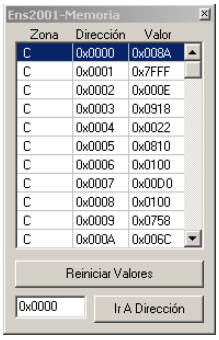


Figura 8.6.1. Ventana de Memoria de *ENS2001*

Además, aparecerán marcadas con una 'C' en la columna de la izquierda aquellas posiciones de memoria que pertenezcan a la zona de código, mientras que aparecerán marcadas con una 'P' aquellas posiciones que ocupe la pila, tal y como se aprecia en la Figura 8.6.2, en la que las posiciones 017Ch a 017Eh están ocupadas por código y las posiciones 017Fh a 0182h están ocupadas por la pila del sistema.

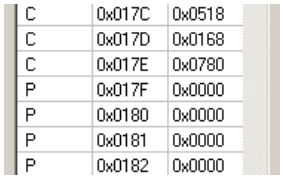


Figura 8.6.2. Indicadores de Zona Código y Pila

Se dispone de dos botones (Figura 8.6.1). El primero de ellos, **Reiniciar Valores**, sirve para poner a cero el contenido de todas las posiciones de memoria. El segundo, **Ir A**

Dirección, permite visualizar en la ventana la memoria a partir de la dirección introducida en el cuadro de texto anexo.

8.7. Ventana de Registros

En esta ventana se obtiene una visión global del banco de registros de la máquina virtual, con los valores que toma cada uno de ellos y la posibilidad de modificarlos haciendo doble clic encima de aquél cuyo valor se desee editar. Puede observarse su aspecto en la Figura 8.7.1.

También muestra información sobre los biestables de estado (éstos no se pueden editar directamente, pero se modificarán al editar el registro `SR`), la siguiente instrucción que se va a ejecutar (a la que apunta `PC`), y el rango de direcciones ocupado por la pila del sistema y el código ensamblado.

Por último, también se dispone de un botón para **Reiniciar Valores**, que inicializará el contenido del banco de registros. Al pulsarlo, se pedirá confirmación, y si el usuario pulsa el botón **Aceptar**, todos los registros tomarán el valor 0, excepto el puntero de pila, que tiene un tratamiento especial: La zona de código puede dejar un hueco anterior y otro posterior a ella. Se calcula cuál es el mayor hueco. Entonces, si la pila crece en sentido ascendente, el puntero de pila ocupará la primera posición del hueco mayor, mientras que si crece en sentido descendente, ocupará la última posición de dicho hueco.

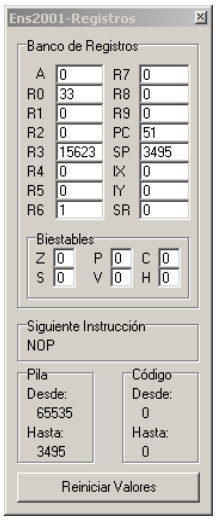


Figura 8.7.1. Ventana de Banco de Registros

8.8. Ventana de Pila

En esta ventana (Figura 8.8.1) se visualizará la zona de memoria correspondiente a la pila. El manejo de la misma, en cuanto a la edición de posiciones de memoria, es idéntico al de la ventana de memoria, por lo que no se abundará más en el tema. En este caso no aparece información acerca de las zonas de código y pila, sólo un indicador de la posición de memoria a la que apunta el puntero de pila (`SP`).

Igual que en la ventana de memoria, la cantidad de posiciones de memoria visibles simultáneamente dependerá del tamaño de la ventana. Adicionalmente, posee dos botones. El primero, **Ir A Dirección**, que centra la visualización en la dirección de memoria

introducida en el cuadro de texto a la izquierda del botón, y el segundo, **Ir A Puntero de Pila**, que muestra la Pila hasta la posición de memoria a la que apunta `SP`. Se trata del indicador "SP →" que aparecerá en la columna de la izquierda. Si la pila crece ascendentemente, `SP` aparecerá en la parte inferior de la ventana. Si crece descendentemente, `SP` aparecerá en la parte superior. De esta forma, es posible visualizar las posiciones cercanas a la cima.

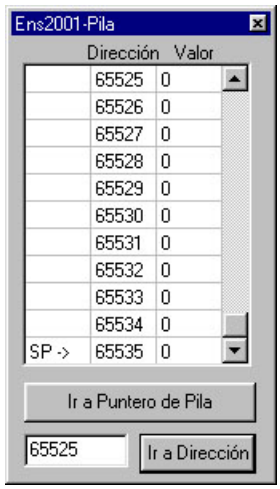


Figura 8.8.1. Ventana de Pila

8.9. Ventana de Configuración

En esta ventana se encuentran accesibles todas las opciones de configuración, como se muestra en la Figura 8.9.1. Se trata de siete grupos de opciones, cada una de las cuales puede tomar dos valores:

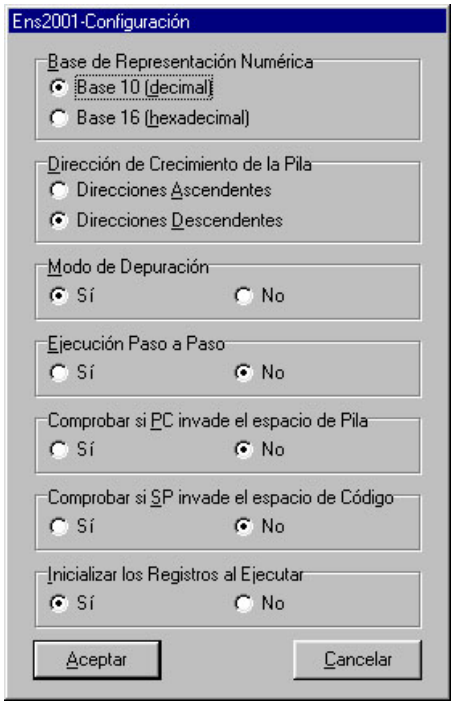


Figura 8.9.1. Ventana de Configuración

Base de Representación Numérica:

Se trata de un selector que permite indicarle al simulador la base de representación numérica que empleará al visualizar los números enteros, bien decimal, bien hexadecimal.

- Base 10 (decimal).
- Base 16 (hexadecimal).

Dirección de Crecimiento de la Pila:

Se trata de un selector que permite indicarle al simulador cuál será el comportamiento de la pila. Hay dos modos predefinidos:

- Direcciones crecientes. La pila crece ascendentemente. Las instrucciones `PUSH` y `POP` se comportan de la siguiente manera:

`PUSH`	`.sp ++` `M(.sp) ← op1`
`POP`	`op1 ← M(.SP)` `.sp --`

- Direcciones decrecientes. La pila crece descendentemente. Las instrucciones `PUSH` y `POP` se comportan de la siguiente manera:

`PUSH`	`M(.sp) ← op1` `.sp --`
`POP`	`.sp ++` `op1 ← M(.SP)`

Modo de Depuración:

Se trata de un selector que permite indicarle al simulador que se detenga cuando encuentre un punto de ruptura, o bien que los ignore por completo.

- Sí.
- No.

Ejecución Paso a Paso:

Se trata de un selector que permite indicarle al simulador que ejecute las instrucciones una a una, deteniéndose después de cada una de ellas, o bien que comience a ejecutar hasta que encuentre un punto de ruptura (si están activos), una instrucción `HALT` en el código, o bien se produzca una excepción.

- Sí.
- No.

Comprobar si PC invade el espacio de Pila:

Se trata de un selector que permite indicarle al simulador que lance una excepción si el contador de programa se sitúa entre los límites superior e inferior de la pila, o bien que ignore dicha situación.

- Sí.
- No.

Comprobar si SP invade el espacio de Código:

Se trata de un selector que permite indicarle al simulador que lance una excepción si el puntero de pila se sitúa entre los límites superior e inferior de la zona perteneciente al código, o bien que ignore dicha situación.

- Sí.
- No.

Inicializar los registros al ejecutar:

Se trata de un selector que permite indicarle al simulador si debe reiniciar el banco de registros al comenzar la ejecución de un programa. No obstante, sólo se reiniciará tras finalizar una ejecución anterior, esto es, cuando el flag `H` esté activo (en otras palabras, si la última instrucción que se ejecutó fue `HALT`).

- Sí.
- No.

Pulsando el botón **Aceptar** se validarán las opciones marcadas, mientras que al pulsar el botón **Cancelar**, los cambios que se hayan introducido no tendrán efecto.

Al salir de la herramienta se guardarán las opciones de configuración en el fichero `ens2001.cfg`, para que sean restauradas en el inicio de una nueva sesión.

9. Interfaz Consola

Salvo la posibilidad de guardar la sesión en un fichero, el resto de funcionalidades son las mismas que para la versión gráfica. Sin embargo, por motivos obvios el manejo es totalmente distinto.

La invocación de la herramienta desde la consola se hace introduciendo en la línea de comandos:

```
ens2001 [nombre_fichero]
```

Donde `nombre_fichero` es el fichero fuente que se ensamblará y colocará en memoria nada más arrancar. Este parámetro es opcional. Si no se indica, se entrará en la herramienta sin más.

En el proceso de arranque, la herramienta intentará cargar la configuración de la sesión anterior, almacenada en el fichero `ens2001.cfg`. En este caso, se mostrará el mensaje:

```
Fichero de configuración (ens2001.cfg) cargado con éxito
```

Si el fichero no existe, o es inválido, se creará uno nuevo y válido con las opciones por defecto, y aparecerá este otro mensaje:

```
Fichero de configuración (ens2001.cfg) no encontrado o erróneo
Se usarán las opciones por defecto
```

Tras esto, se muestra la configuración actual, cualquiera que sea. La configuración por defecto que aplicará la herramienta, si no se encontró el fichero de configuración, es la siguiente:

- Representación en base decimal.
- La pila crece en sentido descendente.
- La ejecución se detiene al encontrar un punto de ruptura.
- La ejecución se efectúa hasta que se dé una condición de parada (no paso a paso).
- No se comprobará si PC invade el espacio de Pila.
- No se comprobará si SP invade el espacio de Código.
- Se inicializará el Banco de Registros al comenzar la ejecución (después de la finalización de una ejecución anterior).

Si se indicó que se ensamblara directamente un fichero (mediante el parámetro opcional de la línea de comandos), a continuación aparecerá el resultado de dicha operación (ver comando *'c' Abrir y Ensamblar*).

Para finalizar el proceso de arranque, se muestra el indicador de la herramienta:

```
ENS2001>
```

Esto quiere decir que está preparada para recibir órdenes.

Todos los comandos de *ENS2001* constan de una letra, seguida en su caso de ninguno, uno o dos argumentos. Los comandos se aceptan tanto en mayúsculas como en minúsculas. A continuación se enumeran todos los comandos disponibles, agrupados por funcionalidades.

'h' (Ayuda).

Muestra una lista de los comandos permitidos, igual a la que se muestra a continuación.

```
Referencia Rapida de Comandos Modo Consola
--------------------------------------------------------------------------
a               Activa/Desactiva Modo Paso A Paso
b               Activa/Desactiva Puntos de Ruptura
b dir           Pone/Quita un Punto de Ruptura en la dirección dir
c fichero       Carga y Ensambla el fichero
d dir num       Desensambla num instrucciones a partir de dir
e               Ejecuta el código
g               Activa/Desactiva comprobación si PC invade la zona de pila
h               Visualiza la ayuda
i               Activa/Desactiva comprobación si SP invade la zona de código
k               Modifica el Funcionamiento de la pila (Creciente/Decreciente)
l fichero       Carga una Imagen de Memoria desde fichero
m dir [valor]   Visualiza/Modifica el contenido de la Dirección dir
n base          Base de Representación Numerica (10 o 16)
o               Configuración del Simulador
p num           Vuelca num posiciones de la pila
q               Salir del Programa
r               Visualiza el Banco de Registros
r reg [valor]   Visualiza/Modifica el Contenido del Registro reg
s fichero       Guarda una Imagen de Memoria desde fichero
t               Resetea Registros/No Resetea Registros al ejecutar programa
v dir num       Vuelca num posiciones de memoria a partir de la dirección dir
--------------------------------------------------------------------------
```

Configuración del simulador.

'a' (Activar/Desactivar modo Paso a Paso).

Se trata de un selector que permite indicarle al simulador que ejecute las instrucciones una a una, deteniéndose después de cada una de ellas, o bien que comience a ejecutar hasta que encuentre un punto de ruptura (si están activos), una instrucción `HALT` en el código, o bien se produzca una excepción.

'b' (Activar/Desactivar Puntos de Ruptura).

Se trata de un selector que permite indicarle al simulador que se detenga cuando encuentre un punto de ruptura, o bien que los ignore por completo.

'g' (Activar/Desactivar comprobación si PC invade la zona de Pila).

Se trata de un selector que permite indicarle al simulador que lance una excepción si el contador de programa se sitúa entre los límites superior e inferior de la pila, o bien que ignore dicha situación.

'i' (Activar/Desactivar comprobación si SP invade la zona de Código).

Se trata de un selector que permite indicarle al simulador que lance una excepción si el puntero de pila se sitúa entre los límites superior e inferior de la zona perteneciente al código, o bien que ignore dicha situación.

Los límites del código los define el ensamblador, lo que quiere decir que si se modifica el código por otros medios (carga de imagen de memoria, edición manual, etc.), dichos límites no se corresponderán con la realidad y los resultados pueden ser inesperados.

'k' (Seleccionar Modo de Funcionamiento de la Pila).

Se trata de un selector que permite indicarle al simulador cuál será el comportamiento de la pila. Hay dos modos predefinidos:

- Direcciones crecientes. La pila crece ascendentemente. Las instrucciones `PUSH` y `POP` se comportan de la siguiente manera:

PUSH	.sp ++ M(.sp) ← op1
POP	op1 ← M(.SP) .sp --

- Direcciones decrecientes. La pila crece descendentemente. Las instrucciones `PUSH` y `POP` se comportan de la siguiente manera:

PUSH	M(.sp) ← op1 .sp--
POP	.sp ++ op1 ← M(.SP)

'n' base (Seleccionar la Base de Representación Numérica).

El parámetro base indica la base de representación con la que la herramienta presentará todos los números enteros, tanto de direcciones, como de datos, salida por consola, etc. Sólo se permiten los valores 10 (base decimal) y 16 (base hexadecimal).

't' (Activar/Desactivar Reiniciar Registros al comenzar la ejecución).

Se trata de un selector que permite indicarle al simulador si debe reiniciar el banco de registros al comenzar la ejecución de un programa. No obstante, sólo se reiniciará tras finalizar una ejecución anterior, esto es, cuando el biestable `H` esté activo (en otras palabras, si la última instrucción que se ejecutó fue `HALT`).

'o' (Mostrar la Configuración Actual).

Muestra una lista con las opciones de configuración seleccionadas actualmente.

A continuación se enumeran los comandos que permiten consultar y modificar individualmente los valores contenidos tanto en la memoria como en los registros del simulador.

'm' (Acceso a memoria).

Este comando tiene tres posibles usos. Si se emplea la sintaxis:

```
m dir
```

presenta en pantalla el contenido de la posición de memoria `dir`. En cambio, si se emplea este otro formato:

```
m dir valor
```

modifica el contenido de la posición de memoria dir, actualizándolo con el entero indicado por `valor`. Se puede producir un error si `dir` o `valor` no son valores válidos para una dirección de memoria o un entero, respectivamente. Por último, introduciendo el comando:

```
m reset
```

se reinician a cero todas las posiciones de memoria.

'r' (Acceso al Banco de Registros).

Este comando también tiene varios usos, en concreto cuatro, según la sintaxis empleada. Si se introduce únicamente:

```
r
```

muestra en pantalla el contenido de todos los registros del banco de registros, así como los biestables de estado y los límites definidos para las zonas de código y pila. Si se quiere consultar únicamente el contenido de un registro en concreto, se usará la sintaxis:

```
r reg
```

donde `reg` es el nombre del registro (sin el punto delante) cuyo valor se va a consultar. Si se indica un nombre que no se corresponde con ninguno de los registros de la máquina, devolverá un error.

Para modificar el contenido de un registro se empleará el siguiente formato:

```
r reg valor
```

con lo que se actualizará el contenido del registro `reg` con el entero indicado por `valor`. Tanto si `reg` no se corresponde con ninguno de los registros de la máquina, como si `valor` no entra dentro del rango permitido de los enteros, devolverá un error.

Por último, existe la posibilidad de reiniciar el contenido de todos los registros, introduciendo el comando:

```
r reset
```

Todos los registros tomarán el valor 0, excepto el puntero de pila, que tiene un tratamiento especial. La zona de código puede dejar un hueco anterior y otro posterior a ella. Se calcula cuál es el mayor hueco. Entonces, si la pila crece en sentido ascendente, el puntero de pila ocupará la primera posición del hueco mayor, mientras que si crece en sentido descendente, ocupará la última posición de dicho hueco.

La herramienta pone a disposición del usuario algunos comandos para visualizar la información contenida en bloques de memoria, incluso para desensamblar código.

'v' dir num (Acceso a Bloques de Memoria).

Este comando permite visualizar un bloque de memoria a partir de la dirección `dir` y de tantas posiciones de memoria como las indicadas por el parámetro `num`. Dado el caso, informará si hay algún error en los parámetros.

'p' num (Acceso a la Pila).

El comportamiento de este comando es análogo al anterior, sólo que como inicio del bloque tomará la dirección a la que apunte el puntero de pila, y centrará la visualización hacia posiciones ascendentes o descendentes, según esté configurado el funcionamiento de la pila, de manera que permita visualizar el contenido de las posiciones cercanas a la cima.

'd' dir num (Desensamblar).

Este comando permite desensamblar `num` instrucciones a partir de la dirección de memoria `dir`. Hay que reseñar que, debido a que todas las instrucciones no ocupan el mismo tamaño en memoria (las hay de 1, 2 y 3 palabras de longitud), se pueden obtener resultados erróneos al desensamblar a partir de una dirección arbitraria. También puede leerse código extraño al desensamblar las zonas de datos de los programas, por ejemplo. Para evitarlo, siempre se debería desensamblar a partir del comienzo del código o bien a partir de la posición ocupada por el contador de programa o una instrucción desensamblada anteriormente.

A continuación se muestran los comandos que permiten cargar y grabar información:

'c' fichero (Cargar y Ensamblar).

Con este comando, la herramienta lee el código fuente contenido en `fichero`, lo ensambla y, si el proceso de ensamblado fue correcto, coloca el código en memoria. Si no, muestra un listado con los errores producidos.

Al ensamblar un fichero, se crean los archivos temporales `memoria.tmp` (contenido de la memoria tras ensamblar el código fuente) y `errores.tmp` (listado de los errores producidos al ensamblar, si los hubiera), que se eliminarán una vez finalizado el proceso. Si no pudiera crear alguno de ellos, se mostrará el siguiente error y no se completará la operación:

```
Error generando fichero nombre_fichero
El directorio de trabajo está lleno o protegido contra escritura
```

Esto podría ocurrir, por ejemplo, si se intenta ejecutar la aplicación desde un *CD-ROM* o una unidad de red con acceso exclusivo de lectura.

's' (Guardar una Imagen de Memoria).

Este comando permite guardar el contenido íntegro de la memoria en un fichero. Se creará un nuevo fichero con una longitud de 128 Kbytes, en formato *big-endian*. Si el fichero ya existía, se sobrescribirá.

'l' (Cargar una Imagen de Memoria).

Este comando permite recuperar el contenido íntegro de la memoria desde un fichero en el que se habrá guardado previamente mediante el comando *'s' Guardar una Imagen de Memoria*. Como la lectura se hace en formato binario, si se intenta leer un fichero cualquiera que no haya sido generado por la aplicación los resultados son inesperados.

Y, por último, pero no ello menos importantes, se muestran los comandos que lanzan la simulación y dan control sobre las partes que se van a ejecutar.

'b' dir (Activar/Desactivar un Punto de Ruptura).

Activa un punto de ruptura en la dirección `dir`, o bien, si ya estaba activo, lo desactiva.

'e' (Ejecutar).

Invoca al procesador virtual para lanzar la simulación. El procesador ejecutará instrucciones hasta que se cumpla una de las siguientes condiciones:

- Ejecución Paso a Paso activa (se detiene nada más ejecutar una instrucción).
- Punto de Ruptura Activado (se detiene antes de ejecutar la instrucción).
- Excepción durante la ejecución.
- Instrucción `HALT`

Al finalizar la simulación, la herramienta mostrará en pantalla cuál de estas causas fue la que detuvo al procesador.

Si durante la ejecución el procesador encuentra una instrucción de entrada por consola, esperará a la introducción del dato por parte del usuario para continuar. Si encuentra una instrucción de salida, el resultado aparecerá en la consola, intercalado con la salida de la propia interfaz.

Para detener la ejecución en cualquier momento, incluso cuando se están ejecutando instrucciones de entrada/salida, bastará con que el usuario pulse la tecla ESCAPE. Se generará una excepción que detendrá al procesador. Esta última característica no está disponible en la versión consola para *Linux*.

10. Tablas Resumen

TABLA 10.1. RESUMEN DE FORMATOS DE INSTRUCCIÓN

Instrucción sin operandos:

15 6	5 3	2 0
Código de operación	000	000

Instrucción con un operando inmediato o directo a memoria:

15 6	5 3	2 0
Código de operación	Modo Dir op1	000

15 0
Operando 1

Instrucción con un operando ni inmediato ni directo a memoria:

15 6	5 3	2 0
Código de operación	Modo Dir op1	000

15 8	7 0
Operando 1	00000000

Instrucción con dos operandos, ambos o bien inmediatos o bien directos a memoria:

15 6	5 3	2 0
Código de operación	Modo Dir op1	Modo Dir op2

15 0	15 0
Operando 1	Operando 2

Instrucción con dos operandos, el primero inmediato o directo a memoria y el segundo no.

15 6	5 3	2 0
Código de operación	Modo Dir op1	Modo Dir op2

15 0
Operando 1

15 8	7 0
00000000	Operando 2

Instrucción con dos operandos, el segundo inmediato o directo a memoria y el primero no.

15 6	5 3	2 0
Código de operación	Modo Dir op1	Modo Dir op2

15 8	7 0
Operando 1	00000000

15 0
Operando 2

Instrucción con dos operandos, ninguno de los cuales es inmediato ni directo a memoria.

15 6	5 3	2 0
Código de operación	Modo Dir op1	Modo Dir op2

15 8	7 0
Operando 1	Operando 2

Nota: los operandos que usen sólo 4 bits rellenarán los 4 bits superiores con ceros.

TABLA 10.2. MODOS DE DIRECCIONAMIENTO, ANCHO Y CODIFICACIÓN

Modo	Ancho	Codificación
Sin Operando	N/A	000 (0)
Inmediato	16 bits	001 (1)
Registro	4 bits	010 (2)
Memoria	16 bits	011 (3)
Indirecto	4 bits	100 (4)
Relativo a IX	8 bits	101 (5)
Relativo a IY	8 bits	110 (6)
Relativo a PC	8 bits	111 (7)

TABLA 10.3. BANCO DE REGISTROS Y CODIFICACIÓN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PC	SP	IY	IX	SR	A	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

TABLA 10.4. POSICIÓN DE LOS BIESTABLES DE ESTADO DENTRO DEL REGISTRO DE ESTADO

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	H	S	P	V	C	Z

TABLA 10.5. INSTRUCCIONES Y DIRECCIONAMIENTOS PERMITIDOS

Mnemónico	Cód. Oper.	Modo Dir. Op1							Modo Dir. Op2						
		#	.	/	[ ]	#[.ix]	#[.iy]	$	#	.	/	[ ]	#[.ix]	#[.iy]	$
NOP	0														
HALT	1														
MOVE	2	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	
PUSH	3	✓	✓	✓	✓	✓	✓								
POP	4		✓	✓	✓	✓	✓								
ADD	5	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
SUB	6	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
MUL	7	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
DIV	8	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
MOD	9	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
INC	10		✓	✓	✓	✓	✓								
DEC	11		✓	✓	✓	✓	✓								
NEG	12		✓	✓	✓	✓	✓								
CMP	13	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
AND	14	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
OR	15	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
XOR	16	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	
NOT	17		✓	✓	✓	✓	✓								
BR	18			✓	✓			✓							
BZ	19			✓	✓			✓							
BNZ	20			✓	✓			✓							
BP	21			✓	✓			✓							
BN	22			✓	✓			✓							
BV	23			✓	✓			✓							
BNV	24			✓	✓			✓							
BC	25			✓	✓			✓							
BNC	26			✓	✓			✓							
BE	27			✓	✓			✓							
BO	28			✓	✓			✓							
CALL	29			✓	✓			✓							
RET	30														
INCHAR	31		✓	✓	✓	✓	✓								
ININT	32		✓	✓	✓	✓	✓								
INSTR	33			✓	✓	✓	✓								
WRCHAR	34	✓	✓	✓	✓	✓	✓								
WRINT	35	✓	✓	✓	✓	✓	✓								
WRSTR	36			✓	✓	✓	✓								