

129 201



UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

DESARROLLO DE UN TRADUCTOR DE LENGUAJE ENSAMBLADOR A CODIGO DE MAQUINA PARA UN MICROPROCESADOR DE 16 BITS.

TESIS PROFESIONAL

QUE PARA OBTENER EL TITULO DE

TESIS CON
FALSA FE. CRGEN

S
:

ING. MECANICO ELECTRICISTA
P R E S E N T A
LUIS ALEJANDRO SAÑUDO CHAVEZ

Director de Tesis: ING. LUIS G. CORDERO BORBOA

MEXICO, D.F.

1990





UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

DESARROLLO DE UN TRADUCTOR DE LENGUAJE ENSEMBLADOR
A CODIGO DE MAQUINA PARA UN MICROPROCESADOR DE
16 BITS

	INTRODUCCION	1
I	DESCRIPCION DE ENSAMBLADORES	3
I.1	Tipos de instrucciones	
I.1.1	Instrucciones para el procesador	
I.1.2	Instrucciones para el ensamblador	
I.2	Macroinstrucciones	
I.3	Proceso de ensamble	
II	PRESENTACION DEL MC 6800	18
II.1	Arquitectura del microprocesador	
II.2	Señalización	
II.3	Registros	
II.4	Organización de la memoria	
II.5	Configuración de un sistema mínimo	
II.6	Elementos de programación del microprocesador	
II.7	Modos de direccionamiento	
II.8	Conjunto de instrucciones	
III	ANALISIS DEL SISTEMA	51
III.1	Conceptualización del problema	
III.2	Metodología de análisis	
IV	DISEÑO DEL SISTEMA	69
IV.1	Especificación del problema	
IV.2	Identificación de los módulos principales	
IV.3	Identificación entradas y salidas	
IV.4	Especificación de Estructuras de datos	
V	PROGRAMACION	80
VI	DEMONSTRACION DE RESULTADOS	136
	CONCLUSIONES	143
	BIOGRAFIA	145

INTRODUCCION

El presente trabajo tiene como objetivo facilitar el manejo de la programación del microprocesador MC 68000, através de un pequeño que traduce un programa escrito en lenguaje ensamblador a su correspondiente programa escrito en lenguaje de máquina para posteriormente introducirlo dentro del microcomputador basado en el microprocesador MC 68000.

Actualmente se cuenta con microcomputadores de este tipo en la facultad de Ingeniería, por lo que se plantea este proyecto como una herramienta más para agilizar la elaboración y el procesamiento de programas en lenguaje ensamblador.

El capítulo 1 resume la estructura interna desde el punto de vista de programación (software), tales como componentes de un sistema de programación, macros y definiciones importantes como que es un ensamblador, su elaboración y diseño.

El capítulo 2 incluye conceptos básicos del microprocesador MC 68000 tales como su arquitectura, contenido interno, ejemplo de un sistema mínimo y su funcionamiento. En este capítulo se incluyen también conceptos de programación, principalmente modos de direccionamiento.

El capítulo 3 contiene las características particulares a emplear en el desarrollo del traductor y análisis del sistema.

El capítulo 4 contiene un desarrollo sistemático del diseño del sistema, basándose en los conceptos manejados en los capítulos anteriores, carta de estructuras y diagrama de flujo de datos.

El capítulo 5 contiene el desarrollo del proyecto basado en el capítulo anterior y el pseudocódigo del diagrama de flujo de datos y la programación estructurada.

El capítulo 6 contiene ejemplos de traducciones ejecutadas con el sistema, manejo del mismo, así como las anotaciones que se deben tener en cuenta para el empleo del traductor.

CAPITULO 1
DESCRIPCION DE ENSAMBLADORES

1 DESCRIPCIÓN DE ENSAMBLADORES

En un principio, el programador de computadoras tenía a su disposición una máquina que interpretaba, a través de hardware, cierto tipo de instrucciones fundamentales. La programación de la computadora se hacía escribiendo una serie de unos y ceros (lenguaje de máquina) colocándolos en la memoria de la máquina, posteriormente presionaba un botón de consenso y de esta forma la máquina empezaba a ejecutarlos como instrucciones.

Los programadores encontraron difícil escribir y leer programas en lenguaje de máquina. En el intento por encontrar un lenguaje más conveniente comenzaron a usar mnemónicos (símbolos) para cada instrucción de máquina, los cuales serían convertidos posteriormente a lenguaje de máquina. Tal lenguaje de mnemónicos es llamado lenguaje ensamblador. Los programas conocidos como ensambladores fueron escritos para automatizar la traducción de lenguaje ensamblador a lenguaje de máquina. La entrada a un ensamblador se conoce como programa fuente, la salida en lenguaje de máquina es el programa objeto.

Hay cuatro ventajas para usar el lenguaje ensamblador en lugar del lenguaje de máquina:

1. Es mnemónico, esto es, escribimos ST en lugar de la configuración de bit 01010000.
2. Las direcciones son simbólicas, no absolutas.
3. La lectura del programa es más fácil.
4. El mantenimiento del programa es más fácil.

Una desventaja del lenguaje ensamblador es que requiere del uso de un ensamblador para traducir el programa fuente en programa objeto.

Con respecto al uso del computador la siguiente relación nos indica algunas características de los diferentes niveles de lenguaje.

BASIC, FORTRAN, PASCAL ;

Ventajas:

requiere de relativamente poco tiempo de programación, usa expresiones en inglés, cada instrucción individual es traducida en muchas instrucciones de máquina, es portable, puede fácilmente adaptarse a diferentes computadores.

Desventajas: Ineficiente uso de memoria comparado con lenguajes de más bajo nivel, dificultad de manejar bits, especialmente para operaciones de entrada salida, no puede ser optimizado respecto a su velocidad de ejecución.

MACROENSAMBLADORES :

Ventajas: Combina la velocidad de los ensambladores con cierta eficiencia de programación de los lenguajes de alto nivel.

Desventajas: Es más difícil de aprender y de usar que los lenguajes de alto nivel.

ENSAMBLADORES:

Ventajas: Corre mucho más rápido que los lenguajes de alto nivel, usa menos memoria que ellos y permite el control de cada bit y dirección.

Desventajas: Es más difícil de aprender y de usar que los lenguajes de alto nivel, requiere de más tiempo de programación, no es exportable, cada microprocesador usa diferente ensamblador.

LENGUAJE DE MÁQUINA:

Ventajas: Trabaja con cada bit dentro y fuera del microprocesador, es el de mayor velocidad al ejecutarse.

Desventajas: Muy difícil de trabajar, requiere de mucho tiempo de programación, no es exportable, diferentes lenguajes para cada sistema y microprocesador.

El lenguaje ensamblador es un lenguaje de símbolos. Para usar un lenguaje ensamblador es necesario aprender los símbolos o instrucciones que reconoce. Estas instrucciones incluyen instrucciones para el procesador, instrucciones para el ensamblador y macroinstrucciones; todo esto difiere de ensamblador a ensamblador. Antes de pasar a describir cada una de ellas, platiquemos brevemente sobre las reglas de sintaxis y formato de las instrucciones en forma general.

La SINTAXIS es un conjunto de reglas para colocar correctamente símbolos conjuntamente de manera que un ensamblador los pueda reconocer, trabajar y traducir apropiadamente. El punto que se debe recordar es que la sintaxis varía de ensamblador a ensamblador. Generalmente las instrucciones básicas tienen la misma representación, las instrucciones más avanzadas pueden no ser las mismas. Algunas reglas generales de sintaxis son :

1. Uso de símbolos para indicar en que sistema numérico se trabaja :

B ó b = Binario.
O ó o = Octal.
D = Decimal.
H ó h = Hexadecimal.

2. Escribir el código de operación de la instrucción al principio, posteriormente escribir los operandos.

3. Separa el operando fuente del operando destino por una coma.

4. Uso de una letra extensiva para indicar el tamaño del operando.

B = Byte.
W = Word.
L = Long-word.

5. Uso de paréntesis en los operandos para indicar modo indirecto. Un par de paréntesis alrededor de un operando muestra que el valor del operando será usado como una dirección indirecta.

6. Uso de signos para indicar direccionamiento post incrementado o predecrementado.

- Después del paréntesis para indicar postincremento.

- Antes del paréntesis para indicar predecremento.

7. Uso de dos puntos después de una etiqueta.

8. Uso de un espacio después del código de operación.

Respecto al **FORMATO** de las instrucciones, los ensambladores organizan los programas en lenguaje ensamblador en divisiones llamadas campos. No son en realidad parte del código; son estructuras que el ensamblador usa para simplificar la comunicación con el programador. Cuando se está programando se tiene que introducir la información en los campos correspondientes o el ensamblador se lo entenderá. Existen tres campos principales: Etiqueta, Instrucción y Comentario.

El siguiente es un ejemplo en un programa ensamblador cualquiera :

Etiqueta	Cod. Op.	Operandos	Comentarios
DATA	DD	\$0010000	Almacena DATA
PROGRAM	DD	\$0012000	Almacena PROGRAM
	ORG	DATA	
TEMP	DS.W	1	Valor a girar
COUNT	DS.W	1	Posiciones a girar.
	ORG	PROGRAM	
SHIFTER	MOVE.W	TEMP,DS	DS tiene el valor a girar.
	MOVE.W	COUNT,DI	Contador giros
	ROT.B	DS,DS	Giro
	MOVE.W	DS,VALUE	
	RTS		
	END	SHIFTER	

Esta es la organización del código fuente que el ensamblador traducirá en código objeto. El campo de la etiqueta contiene etiquetas simbólicas o direcciones de varias instrucciones. El ensamblador puede utilizar estas referencias simbólicas para especificar saltos o regresos. El siguiente campo es realmente el más importante. El campo de las instrucciones se divide en dos partes, el primero es el código de operación, el siguiente son los operandos. Finalmente se tiene el campo de los comentarios. Los comentarios son opcionales, y no cambian la operación del programa, pero deben ser usados.

Hay varios tipos de ensambladores. Un poderoso ensamblador conocido como **MACROENASMLADOR** permite definir macroinstrucciones y usarlas. Estas macros son instrucciones para el procesador o para el macroensamblador agrupadas en bloques a los cuales se les da un nombre. Una vez que se ha definido la macroinstrucción, lo único que se tiene que hacer es llamarla para que el macroensamblador sustituya el bloque de instrucciones por el nombre de la macroinstrucción, se hablará de esta macroinstrucción más adelante.

Otro importante tipo de ensamblador es el que se conoce como CROSS ASSEMBLER. Dado que la creación, edición y ensamble de un programa fuente en código objeto es abstracto (no requiere del microprocesador real), los programadores frecuentemente usan ensambladores que están escritos y corren en otras computadoras que no tienen el microprocesador para el cual el programa está dirigido, a tales ensambladores se los conoce como CROSS ASSEMBLERS.

1.1 Tipos de instrucciones

El siguiente subtema contempla los dos tipos de instrucciones contemplados para el sistema que son: instrucciones para el procesador e instrucciones para el ensamblador.

1.1.1 Instrucciones para el procesador.

Las operaciones fundamentales que un microprocesador puede ejecutar se conocen como instrucciones para el procesador, son las palabras en el lenguaje del microprocesador. El conjunto de instrucciones del microprocesador MC68000 está dado en el capítulo 11, todo ese conjunto forman las instrucciones para el procesador MC68000.

Las instrucciones fundamentales de cualquier microprocesador pueden ser divididas en grupos funcionales. Estos grupos son muy similares de circuito integrado a circuito integrado.

Instrucciones de movimiento de datos.

Muchas personas piensan en el computador como máquinas matemáticas, cuando en realidad su mayor función es la de almacenamiento y manipulación de datos. Cualquier programa tiene que mover BITS y BYTES entre la entrada y la salida, entre la memoria y el CPU.

Las instrucciones de movimientos de datos permiten mover información de registro a registro, registro a memoria o de memoria a memoria, la información que se mueve puede ser dirección o dato. El movimiento de los datos puede ser BYTE, palabra o palabras completas.

Instrucciones aritméticas.

En términos generales la mayoría de los microprocesadores pueden efectuar operaciones de suma y resta, los microprocesadores de 16 bits, tienen también multiplicación y división. Dentro de las operaciones aritméticas podemos mencionar, además, comparación, selección e inicialización de registros.

Instrucciones lógicas.

La mayoría de los computadores hacen uso extendido de las operaciones lógicas. Las operaciones generalmente manejadas son AND, OR, XOR, NOT.

Instrucciones de desplazamiento y giro.

Son instrucciones que permiten cambiar o mover el contenido de un operando. Las instrucciones de giro y desplazamiento mueven BITS en forma adyacente de una posición a la siguiente en un registro o localidad de memoria.

Instrucciones de salto condicional e incondicional.

Cualquiera de las instrucciones de salto condicional e incondicional obliga al programa a continuar la ejecución del programa en una nueva posición. Las instrucciones de salto incondicional ponen un nuevo valor en el CONTADOR DE PROGRAMA, de esta forma el procesamiento continúa en una nueva dirección.

1.1.2 Instrucciones para el ensamblador

Las DIRECTIVAS son instrucciones para el ensamblador, no son parte del conjunto de instrucciones para el procesador. Las directivas son usadas para especificar el contenido de un programa, inicializar variables, reservar espacio de memoria para ciertas estructuras de datos o definir MACROS (son una secuencia comprimida de instrucciones usadas para ahorrar tiempo en la programación), la siguiente es una lista de las principales directivas que usan la mayoría de los ensambladores:

DATA	Introduce datos en una memoria.
EQUATE	Relaciona nombres simbólicos con direcciones o datos.
END	Muestra el fin del programa.
ENTRY	Muestra qué nombre está disponible para usarlo.
EXTERNAL	Indica que el nombre está definida en otro programa.
LIST	Imprime programa fuente.
NAME	Imprime el nombre del programa en la cabecera de cada página.
ORIGIN	Especifica la localidad de memoria donde el programa o los datos quedara almacenado.
PAGE	Salta el listado a la siguiente página.
RESERVE	Asigna memoria.

Existen, también, directivas condicionales, es decir, directivas que seleccionan, mediante el valor de algún parámetro que parte del programa fuente será ensamblado. Generalmente estas directivas se encuentran asociadas a MACROS a las cuales nos referiremos más adelante.

1.3 Macroinstrucciones

El programador de lenguaje ensamblador suele encontrar necesario repetir bloques de instrucciones en un programa. El bloque puede consistir de instrucciones para salvar o intercambiar conjuntos de registros, por ejemplo, o una serie de operaciones aritméticas. Realmente el bloque puede agrupar conjunto de instrucciones para el procesador, directivas o una combinación de ambas. Las macroinstrucciones (generalmente llamadas MACROS) permiten referenciar a través de una sola instrucción un conjunto de instrucciones. Por cada referencia de la macro el ensamblador sustituirá todo el bloque de instrucciones.

Mediante la definición apropiada de macros, un programador en lenguaje ensamblador puede optimizar su uso. Puede hacer semejante su codificación a los lenguajes de alto nivel sin perder las ventajas básicas de la programación en lenguaje ensamblador. Integrar instrucciones en macros simplifica el desarrollo y mantenimiento de los programas, así como facilita la estandarización.

Para generar el programa objeto de un programa en lenguaje ensamblador es necesario usar un ensamblador, un programa en lenguaje ensamblador que use macros deberá ser traducido por un MACROENSAMBLADOR. Generalmente se considera a las macroinstrucciones como una extensión del lenguaje ensamblador básico, el macroensamblador es visto como extensión del ensamblador.

Características de las macroinstrucciones.

En su forma simple, una MACRO es una abreviatura para una secuencia de operaciones. Consideremos el siguiente programa (no se trata de algún programa escrito en cierto tipo de lenguaje ensamblador, véase solo para ilustrar la forma de las MACROS):

Una vez que la macro ha sido definida, el uso del nombre de la macro como el nombre de una operación es equivalente al uso de su correspondiente secuencia de instrucciones. Nuestro ejemplo queda escrito de la siguiente forma :

```

DATA      DS      2;
          .
          .
MACRO;
SUMA;
ADD      1, DATA;
ADD      2, DATA;
ADD      3, DATA;
MEND;
          .
          .
SUMA;
          .
          .
SUMA;
          .
          .

```

En este caso el macroensamblador reemplaza cada llamada a la macro por las líneas :

```

ADD      1, DATA;
ADD      2, DATA;
ADD      3, DATA;

```

Este proceso de reemplazo es llamado expansión de la macro. La ocurrencia en el programa fuente del nombre de la macro se conoce como llamada a la macro.

La estructura de la macro presentada hasta el momento es capaz de insertar bloques de instrucciones en el lugar de las llamadas de las macros. Todas las llamadas a una macro dada son reemplazadas por un bloque idéntico. Este tipo de estructura de las macros no es flexible ; no hay forma de modificar el código que reemplaza a la llamada de la macro.

Como resultado de esta restricción fué implementada la posibilidad de pasar **PARAMETROS** o **ARGUMENTOS** en la llamada a una macro. Correspondientes argumentos dados de la macro aparecen en su definición.

Ejemplo 2.

```

DATA1      DS  2;
DATA2      DS  2;
.
.
.
      ADD     1, DATA1;
      ADD     2, DATA1;
      ADD     3, DATA1;
.
.
.
      ADD     1, DATA2;
      ADD     2, DATA2;
      ADD     3, DATA2;
.
.
.

```

En este caso la secuencia de instrucciones son muy similares pero no idénticas. La primera secuencia ejecuta operaciones usando **DATA1** como operando. La segunda, usa **DATA2**. Realmente la secuencia difiere en un argumento variable. Una macro puede ser definida para permitir argumentos de la siguiente forma :

```

MACRO:
SUMA      @ ARG;
      ADD     1, @ ARG;
      ADD     2, @ ARG;
      ADD     3, @ ARG;
MEND;
.
.
.
SUMA      DATO1;
.
.
.
SUMA      DATO2;
.
.
.

```

Es posible proporcionar más de un argumento en una llamada a macro. Cada argumento debe corresponder con los argumentos definidos en el nombre de la macro. Cuando la macro es ensamblada, los argumentos dados son sustituidos por sus respectivos argumentos dados en la definición de la macro.

Ejemplo 1.

```

        DATA1    DS    2;
        DATA2    DS    2;
        DATA3    DS    2;
        .
        .
        .
CICLO1:  ADD      1, DATA1;
        ADD      2, DATA2;
        ADD      3, DATA3;
        .
        .
        .
CICLO2:  ADD      1, DATA1;
        ADD      2, DATA2;
        ADD      3, DATA3;
        .
        .
        .

```

En este caso, los operandos en el bloque de instrucciones son diferentes, así como las etiquetas. Con la facilidad de manejo de argumentos en las llamadas a macros, el ejemplo 3 sería escrito como :

```

        .
        .
        .
        MACRO    SUMA (ARG1, ARG2, ARG3);
        ADD     1, ARG1;
        ADD     2, ARG2;
        ADD     3, ARG3;
        MEND;
        .
        .
        .
CICLO1:  SUMA    DATA1, DATA2, DATA3;
        .
        .
        .

```

```
CICLO3:   SUMA      DATA3, DATA3, DATA3:
```

Cuando se recibe de instrucciones para el ensamblador se mencionarán las directivas condicionales, el uso en la definición de macroinstrucciones es el mismo que en el programa en lenguaje ensamblador sin el soporte de macroinstrucciones, es decir, proporcionar un metalingua para decir que parte del bloque de instrucciones que sobrevia la macro formará parte del programa ensamblado.

Consideremos el siguiente ejemplo:

Ejemplo 4.

```
DATA1    DS    1;
DATA2    DS    2;
DATA3    DS    3;

.
.
.
CICLO1:  ADD    1, DATA1;
        ADD    2, DATA2;
        ADD    3, DATA3;

.
.
.
CICLO2:  ADD    1, DATA1;
        ADD    2, DATA2;

.
.
.
CICLO3:  ADD    1, DATA1;
```

El programa puede ser escrito usando condicionales, de la siguiente forma :

```

MACRO:
VARIA      @BANDERA, @ARG1, @ARG2, @ARG3:
ADD        1, @ARG1:
-IF        (@BANDERA EQ 1) .FINI
ADD        2,@ARG2:
-IF        (@BANDERA EQ 2) .FINI
ADD        3,@ARG3:
.FINI     END

```

```

      .
      .
CICLO1: VARIA      3,DATA1,DATA2,DATA3:
      .
      .

```

```

CICLO2: VARIA      2,DATA1,DATA2:
      .
      .

```

```

CICLO3: VARIA      1,DATA1:
      .
      .

```

Las instrucciones que comienzan con un punto son directivas, esto es, no aparecen en la salida del macroensamblador. La instrucción al macroensamblador `.IF (@BANDERA EQ 1) .FINI` lo dirige para saltar a la etiqueta `.FINI` si el valor correspondiente a `BANDERA` es 1, de otra forma el macroensamblador continúa con la siguiente instrucción al `.IF`.

1.3 Proceso de ensamble

Cualquier ensamblador debe ejecutar los siguientes pasos :

1. Generar instrucciones :

- 1.1 Evaluar el número en el campo de operación para producir su código de máquina.
- 1.2 Evaluar los operandos : definir el valor de cada símbolo, procesar literales y asignar direcciones.

2. Ejecutar directivas.

Con esta información, las siguientes son las etapas en el proceso de ensamble.

1. Mantener un contador de localidades.
2. Almacenar variables y sus direcciones.
3. Ejecutar directivas.
4. Almacenar literales y sus valores.
5. Determinar la longitud de las instrucciones para el procesador.
6. Evaluar las instrucciones para el procesador.
7. Generar el código de la instrucción.

CAPITULO II
PRESENTACION DEL MC 68000

II. Presentación del MC 68000

Tomando en cuenta que existen varias versiones del MC 68000 a continuación, se resume un poco de esta familia.

La primera presentación de Motorola en el ámbito de los 16 bits fue el MC 68000, contiene 17 registros de datos y direcciones de 32 bits, 14 tipos de direccionamiento, capacidad para manejar 16 megabytes de memoria, 56 tipos de instrucción, utiliza segmentación y admite cinco tipos de datos. El bus de datos es de 16 bits y el bus de direcciones de 24 bits.

El microprocesador 68008, ofrece las características básicas del 68000, sumando efectividad de costo en su encapsulado, utiliza un bus de datos 8 bits que permite acoplarlo y acceder a memorias económicas organizadas en bytes y a periféricos más sencillos. El microprocesador 68010 añade el soporte de hardware para memoria virtual y máquinas virtuales, tablas de vectores múltiples e instrucciones para bucles de altas prestaciones, utiliza un bus de datos de 16 bits y un bus de direcciones de 24 bits. El microprocesador 68012 varía del 68010 en que su bus de direcciones es de 30 bits. El microprocesador 68020 contiene buses completos de 32 bits, capaz de direccionar 4 gigabytes, interfaz a coprocesador, 7 tipos de datos, 18 modos de direccionamiento y un acelerador (cache) de altas prestaciones para instrucciones en una pastilla.

II.1 Arquitectura del microprocesador

La arquitectura del microprocesador se entiende como su estructura interna, está incluye elementos como caracterización de las líneas de dirección, datos y control, así como su repercusión en los formatos de instrucción y los modos de direccionamiento.

Alineamiento

En los procesadores 68000, 68008, 68010, 68012, las palabras y palabras largas (incluyendo palabras instrucciones) deben comenzar en una dirección par. Esto es un requerimiento impuesto por la naturaleza del bus. Si se intenta acceder a una palabra o palabra larga en una dirección impar, el procesador inicia un procesamiento excepcional.

El 68020 solventa esta limitación comprobando los accesos de palabras impares; si así ocurre, hace dos o más accesos parciales a la palabra o palabra larga. Naturalmente, esto añade una sobrecarga a la ejecución, por lo que se debería alinear. Cuando sea posible, las palabras en fronteras pares (bit 0 igual a cero) y las palabras largas en fronteras de palabras largas (bit 0 y 1 iguales a cero). Para mejor funcionamiento, la pila debería ubicarse en una frontera de palabra larga.

Unidad aritmética y lógica

Datos de memoria o registros son ruteados através de la ALU, donde operaciones matemáticas y lógicas son ejecutadas, para después, los datos procesados sean enviados a su destino final.

El MC68020 tiene tres unidades aritméticas lógicas: una para procesar datos y dos para procesar direcciones. La ALU para datos tiene 16 bits para cálculos de datos y evaluación en un solo paso a los 16 bits de datos. Operaciones de 32 bits de datos son hechas en dos pasos, primero la parte baja de la palabra y después la parte alta de la palabra.

Otras dos ALU's son usadas conjuntamente para calcular direcciones. La dirección efectiva (EA) es el resultado final generado desde la instrucción de datos y modo de direccionamiento, para esto es necesario usar dos ALU's ya que las direcciones están formadas de 32 bits.

Decodificador

El decodificador es la parte del microprocesador que interpreta las instrucciones, este traduce las instrucciones en patrones de 1's y 0's para formar el lenguaje de máquina y decir al resto del microprocesador que hacer.

Los más avanzados microprocesadores como el 68000 están frecuentemente en microcódigo, tienen en efecto, un muy pequeño microprocesador que corre rutinas del microprocesador mayor.

Existe una memoria ROM que es programada con instrucciones "tiny" que dicen al procesador que hacer. El microprocesador "tiny" es llamado como microsecuenciador. Estas instrucciones son más simples que el lenguaje de máquina. Ellas envuelven acciones elementales (llamadas micropalabras) tales como enviar señales a ciertas compuertas o retener un bit particular del registro de estado.

Micropalabras (microwords) están construidas dentro de microrutinas que vienen a ser instrucciones en lenguaje ensamblador.

Existen dos tipos de microprogramación: horizontal y vertical. Horizontal es más directa; un solo bit de una micropalabra puede habilitar un registro. Micropalabras horizontales son más largas y requieren microbuses más anchos y facilidades de almacenamiento.

Micropalabras verticales codifican información para 16 registros de 4 bits. Este formato es mucho más lento, debido a que las micropalabras tienen que ser decodificadas por sí mismas lo cual requiere más área de circuito integrado.

Ambas formas de microcódigo son usadas en el procesador 68000, esto quiere decir un doble uso para lo que el procesador tiene un nanocódigo.

La información en microcódigo es puesta en microsubrutinas en nanocódigo, lo cual actualmente hace ruteo, selección y direccionamiento.

Líneas de direcciones

El MC 68000 tiene un bus de direcciones de 23 bits (A1-A23) y es capaz de direccionar una memoria de 8 mega-words (16 Mbytes), este bus de direcciones es de tres estados, unidireccional y asíncrono.

Líneas de datos

Es de 16 bits (D0-D15) permite la transferencia de datos con longitud de byte, palabra ó palabra larga (2 accesos). En el ciclo de reconocimiento de interrupción en D0-D7 está el vector de transferencia. El bus es de tres estados, bidireccional, asíncrono y no es multiplexado.

11.2 Señalización

Señales del MC 68008.

Control asíncrono del bus.
Controla la comunicación asíncrona con memorias y dispositivos periféricos.

Dirección válida en el bus, AS (por sus siglas en inglés).

Sentido de transferencia en el bus, S/W (por sus siglas en inglés).

Dato superior habilitado, USS (por sus siglas en inglés).

Dato inferior habilitado, LSD (por sus siglas en inglés).

Reconocimiento de transferencia de datos, DTACK (por sus siglas en inglés).

Señales de arbitraje de bus.

Petición de bus, BR (por sus siglas en inglés).

Cesión de bus, BG (por sus siglas en inglés).

Reconocimiento de cesión de bus, BG ACK (por sus siglas en inglés).

Señales de interrupción.

IP10, IP11, IP12 : Tienen definido el nivel de interrupción o prioridad para indicar al procesador que dispositivo interrumpe.

Señales de control del sistema.

Señal de entrada de error en el bus, BERR (por sus siglas en inglés).

Señal bidireccional de reinicio, RESET (por su significado en inglés).

Señal bidireccional de parada, HALT (por su significado en inglés).

Señales de la familia C800.

Señal disponible, E (por su sigla en inglés).

Señal de periférico válida, VPA (por sus siglas en inglés).

Señal de dirección de memoria válida, VMA (por sus siglas en inglés).

Códigos de función.

FC0, FC1 y FC2 (FUNCTION CODE): indican el estado normal o supervisor y el ciclo que se está ejecutando en el procesador.

Modos de ejecución.

El MC 68000 puede operar en uno de dos modos, el modo "usuario" o el modo "supervisor". Un bit del registro de estado determina el modo de funcionamiento del procesador.

El modo usuario es para cuando el procesador ejecuta operaciones a nivel de aplicaciones, mientras que el modo supervisor está proyectado para programas a nivel de sistema operativo.

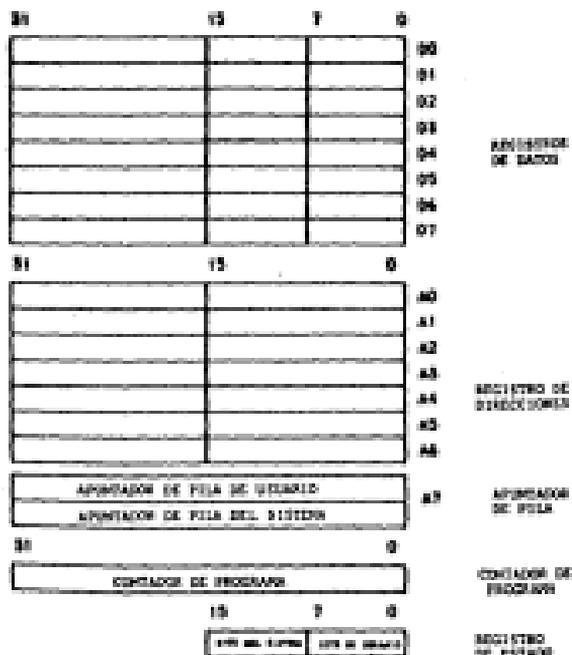
El modo supervisor tiene su propio puntero de pila, así como instrucciones adicionales, privilegiadas, que un programa ejecutándose en modo usuario no puede utilizar.

En base a los modos de ejecución definiremos los registros y su operación.

11.3 Registros

Los registros son localidades de memoria dentro del CPU, debido a que son direccionados fácilmente, leídos o escritos. Los registros pueden ser de propósito general o de propósito especial.

registro de bits



Registros en modo usuario

En modo usuario el MC 68000 tiene 8 registros de datos de 32 bits, 7 registros de dirección de 32 bits. Un apuntador (puntero) de pila (stacker pointer) de 32 bits, un contador de programa de 32 bits y un registro de código de condición de 8 bits.

Los registros de datos pueden usarse como datos de un solo bit, datos de 8 bits (byte), datos de 16 bits (palabra) y datos de 32 bits (palabra larga) como se muestra en la siguiente ilustración.



Cuando una instrucción utiliza un registro de datos como operando fuente o destino, sólo la porción correspondiente del registro es alterada; los bits de mayor orden del registro permanecen sin cambio.

Además de su función como fuente o destino de operandos en una operación, los registros de datos pueden funcionar como registros índices o contadores en bucles.

Registros de direcciones

El MC 68000 tiene 7 registros de direcciones de uso general, dependiendo del modo de direccionamiento estos registros pueden mantener direcciones de punteros a operandos, direcciones base e índices.

Los registros de dirección pueden mantener datos de 16 bits o de 32 bits. No hay operaciones directas con bytes en los registros de dirección.

Cuando se utiliza directamente como operando fuente, los valores de 16 bits son ampliados en signo antes de usarlos (la palabra de mayor orden no afecta ni es afectada por la operación). Cuando se usan directamente como destino de operandos, las fuentes de 16 bits están ampliadas en el signo y rellenan completamente los 32 bits del registro de dirección de destino.

El procesador utiliza un octavo registro de direcciones A7, como puntero de pila; el procesador tiene dos punteros de pila distintos, dependiendo del modo de procesamiento en curso. Estos registros (A7 y A7') son comúnmente más conocidos como puntero de pila del usuario (USP) y puntero de pila del sistema (SSP).

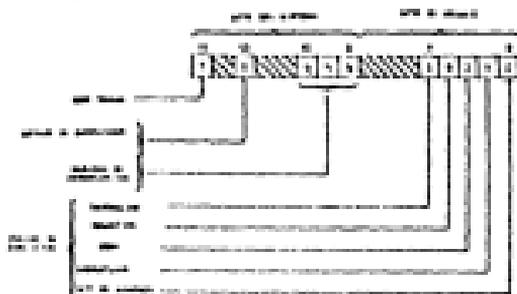
El procesador rellena una pila de posiciones altas a bajas de memoria predecrementando o post-incrementando los modos de direccionamiento indirectos para apilar (insertar) y desapilar (extraer), respectivamente.

Contador de programa

El contador de programa es un registro de uso especial de 12 bits. Apunta a la primera palabra de la siguiente instrucción a ser ejecutada (verificando instrucciones) o a la siguiente palabra de una instrucción durante la captación de instrucciones. Como las direcciones deben estar alineadas en direcciones frías (word boundaries), el contador de programa debe contener siempre un valor par.

Registro de estado

En modo usuario, el procesador tiene acceso al byte de menor orden del registro de estado. Este byte del registro de código de condición o CCR) mantiene bits indicadores cuyos valores dependen del resultado de ciertas operaciones, tales como suma, resta, desplazamiento y otras.



Registros para el modo sistema

Adicionalmente a los registros de datos direcciones y de código de condición disponibles en el modo usuario, los miembros de la familia MC 68000 tienen varios registros adicionales utilizables en el modo supervisor. El tipo y número de registros supervisor varía entre miembros individuales de la familia MC 68000 por lo que hacemos un enfoque hacia el MC 68000 debido a que el bit educativo está basado en este procesador. Contiene dos registros adicionales, el puntero de pila supervisor (SP) y el byte superior del registro de estado. El SP tiene una longitud de 12 bits y apunta descendientemente en memoria las referencias a opcodes bien estar alineadas a palabras (direcciones pares).

El byte del sistema del registro de estado junto con el registro de código de condición forman el registro de estado de 16 bits. El byte del sistema contiene dos bits indicadores y tres bits de máscara de interrupción.

El bit superior (5) especifica el modo de ejecución del procesador. Si es uno, el procesador está en modo supervisor; si es cero, el procesador está en modo usuario.

El bit de modo traza (7), cuando es uno especifica que el procesador se encontrará en modo traza. Este modo conduce a un modo paso a paso de ejecución. Esto permite a un programa depurador monitorizar los resultados de un programa de aplicación ejecutando instrucciones e instrucciones.

Los procesadores 68000 pueden operar en cualquiera de ocho niveles o prioridades de interrupción. Los dispositivos externos pueden intentar interrumpir al procesador con señales de validación según cualquier combinación en las líneas de petición de interrupción IPL0-IPL7. El valor binario de las líneas de interrupción representa la importancia del dispositivo que interrumpe: un dispositivo de prioridad alta (tal como el reloj) tiene un nivel de interrupción de valor superior al de un dispositivo de prioridad baja, tal como un controlador de terminal. La prioridad mayor es de 7 (en binario 111) y el nivel inferior es cero (en binario 000).

La máscara de interrupción en el registro de estado define el nivel de operación en curso del procesador. El código del modo normal del usuario, normalmente cero. Los controladores de dispositivos operan a niveles superiores. Cuando un dispositivo interrumpe al procesador, la lógica interna compare el valor de la máscara de interrupción con el valor del dispositivo contenido en IPL0-IPL7.

La máscara de interrupción, en conjunción con las líneas de interrupción administran un modelo sofisticado de jerarquizar las prioridades de ejecución.

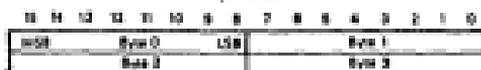
11.4 Organización de la memoria

La familia del MC68000 permite acceder a la memoria en grupos de bits constituyendo un byte, palabra o palabra larga. Una dirección especifica la posición del byte o, para datos de varios bytes, la posición del byte más significativo. El byte menos significativo de una palabra está en esa dirección más 1. El byte menos significativo de una palabra larga está en esa dirección más 1. La siguiente figura muestra como están alineados en memoria los bytes, palabras y palabras largas.

Datos bin
1 byte = 8 bits



Byte orden
1 byte = 8 bits



1 palabra = 16 bits



Bytes pares

Bytes impares

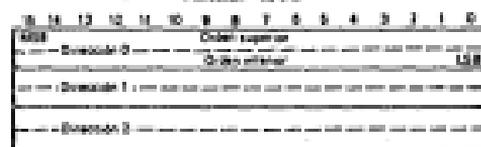


1 palabra larga = 32 bits



Hexadecimales

1 hexadecim = 16 bits



MSB = Bit más significativo

LSB = Bit menos significativo

Datos decimales

2 dígitos decimales codificados binario (BCD) = 1 byte



MSB = Bit más significativo

LSB = Bit menos significativo

Memoria Virtual

En muchos sistemas basados en MC68000 como procesador central, solo una parte de los 16 megabyte (2 gigabyte para el MC68020) de espacio de direccionamiento contendrán memoria físicamente, sin embargo, por medio del uso de técnicas de memoria virtual, el sistema puede aparentar al usuario que dispone de 16 megabytes de memoria físicamente. Estas técnicas han sido usadas por varios años en sistemas principales de gran capacidad (large main frame systems) y más recientemente en minicomputadores y ahora con el MC68000 pueden ser totalmente soportados por sistemas basados en microprocesadores.

11.5 Configuración de un sistema mínimo

Para fines prácticos y con el enfoque de el proyecto que este trabajo persigue, se tomará referencia del computador educativo. El cual es un sistema completo de microcomputadora con procesador de 4 Mhz (MC6800DL4).

Se muestra a continuación el diagrama a bloques del sistema basado en MC68000 para el computador educativo.

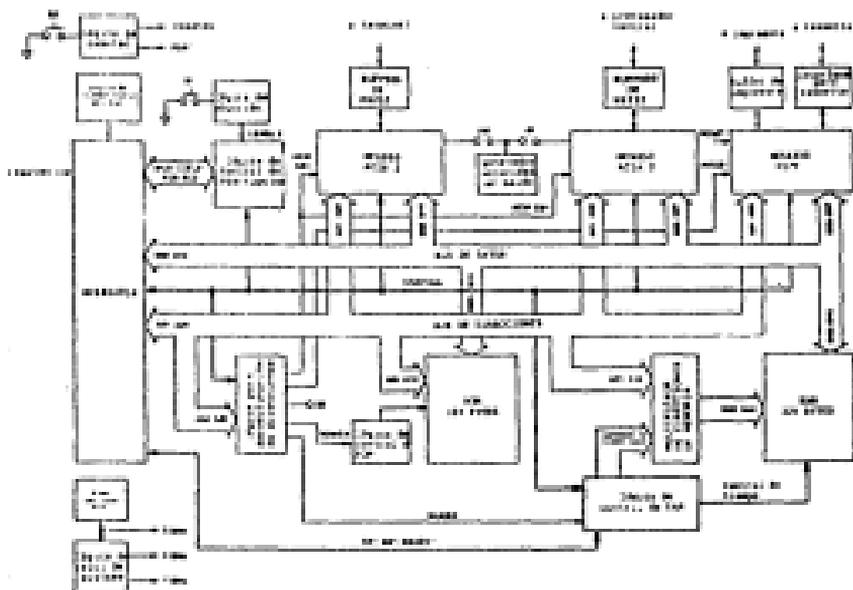


DIAGRAMA BLOQUES- COMPUTADOR EDUCACIONAL

Las áreas funcionales se describen a continuación.

MC680084 Microprocesador

Este procesador trabaja a razón de 4 Mhz como se comentó anteriormente. La ejecución de instrucciones consiste en una combinación de ciclos internos y ciclos de acceso al bus.

El microprocesador utiliza transferencias asíncronas para llevar datos fuera del procesador. Esto significa que el procesador y los periféricos utilizan líneas de conformidad para coordinar la transferencia de datos.

Usando transferencias asíncronas, el procesador permite transferencias de y hacia los periféricos de velocidades diversas.

Decodificador de direcciones

Para entrar a fondo con este elemento, nos basaremos en el mapa de memoria asignado al computador.

MAPA DE MEMORIA			
	Función		Dirección
memoria reservada del sistema	tabla de vectores	ROM/TPROM	000000-000007 (1)
		ROM	000008-00001F
memoria para usuarios	memoria del usuario	RAM	000020-00001F
dirección del ciclo de carga	no usada	ROM/TPROM	000020-00007F (1)
			000080-0000FF
registros I/O	no usada	PI/O	000000-00000F
		ACIA2 (para lazo de datos)	000010-00001F
		ACIA1 (para el bus de datos)	000020-00002F
no usada		registro	000030-00003F
no usada			000040-00007F
no usada			000080-0000FF
no usada			000100-0001FF

NOTA: (1) Datos en esta lista

La lógica de decodificación de direcciones habilita la generación de las señales RAMEN, ROMEN, ACIA CS1, y PUTES de acuerdo con el mapa de memoria. Las líneas de dirección A03-A23 son decodificadas y usadas con señales propias de control para generar estas señales.

La memoria RAM es direccionada en la parte inferior o más baja del mapa, excluyendo las ocho primeras localidades, las cuales contienen el apuntador de pila inicial y el contenido del contador de progreso grabado en memoria PROM.

La RAM es dividida dentro de dos áreas, esta es de 8000000 a 80003FF es una área reservada para el sistema básico (firmware). El cual contiene instrucciones inalterables básicas para la operación del equipo.

La memoria RAM comprendida dentro de las direcciones 8000900 a 80078FF es área disponible para el usuario.

Direcciones de 8000400 a 80008FF son usadas como memoria borrable (scratchpad memory) para el firmware TUTOR incluyendo secuenciador de datos(buffer), apuntadores, memoria temporal, etc.

Todos los dispositivos de entrada/salida son mapeados dentro de la misma página de 4 Kbyte en las direcciones 8010000 a 801FFFF.

El arreglo de memoria dinámica está constituido por 63 chips de memoria MC64100B (10K x 1). Estos pueden ser accedidos en cualquiera de dos formas; por byte o por palabras y la transferencia de datos hacia o desde el MC68000 es a través del bus de transferencia asíncrono.

El tiempo de acceso a memoria es aproximadamente de 450 nanosegundos y la señal RAM DIACK (reconocimiento de transferencia de datos) es generada en un tiempo de 500 a 625 nanosegundos después de un comienzo de ciclo de lectura o escritura.

Control lógico de RAM

Esto controla las operaciones de memoria RAM. Genera un control temporizado (timing system) para los dispositivos RAM así como señales de control para el multiplexor de direcciones de memoria.

El multiplexor genera direcciones en renglón y columna hacia las líneas A01-A14 durante ciclos de lectura y escritura.

Direcciones de refresco (refresh address) son también ruteadas a memoria por el multiplexor durante el refresco de memoria (refresh memory).

La memoria RAM es completamente refrescada cada 1.5 milisegundos en promedio (en el peor de los casos 1.9 milisegundos).

16KBYTE ROM

El sistema firmware (TUTOR) está grabado en dos ROMs de 8Kbits. El tiempo de acceso para las ROMs varía entre 150 a 400 nanosegundos dependiendo del dispositivo usado.

Puerto de comunicación serie

Está formado por dos MC1650 ACIA's que proveen un bus de interfase para puertos serie. ACIA1 es usado para comunicación con una terminal y está conectado a los bits D04-D15 del bus de datos.

ACIA2 es usado para comunicación con un procesador huésped (host processor).

Generador de velocidad en bauds

Los relojes de transmisión y recepción son suministrados por el generador de velocidad en bauds. La velocidad puede fijarse desde 110 a 3000 baud por medio de la combinación de los pines J9 y J10. Ambos puertos son RS-232c compatibles.

Los puertos ACIA son los únicos dispositivos del computador educacional que toman la ventaja del MC68000 y MC6800 de ser compatibles con terminal síncrona. Los ACIA solo pueden ser accedidos usando una transferencia de sus tipo síncrona. Estos dispositivos son manejados por una señal de reloj: Señal E clock, la cual es administrada por el MC68000 (E = 4Mhz dividida entre 10 = 400KHz).

Cada vez que las señales de dirección decodificadas en cualquiera de los ACIA está siendo accedida, el MC68000 sincroniza por sí mismo con la señal de reloj E clock y usa un ciclo de sus muestreo.

Interfases

El circuito integrado MC68030 PI/T provee varias características en el computador, contiene un reloj programable de 34 bits.

Los puertos paralelos A y B del PI/T son compensados (buffered) en cuanto a la razón de flujo de información para manejar impresoras Centronics compatibles. Además, el puerto C del PI/T es compensado también para trabajar con interfase de línea para cassette.

El MC68030 está conectado en las líneas D00-D07 del bus de datos y usa la interfase asíncrona del MC68000

Control lógico de interrupciones

Esta lógica monitorea las señales de código de función (F00-F01) y genera la señal de reconocimiento de interrupción para el dispositivo que interrumpe.

Señales de reloj del sistema

Un oscilador de 4 Mhz es la base de tiempo de cual se derivan todas las señales de reloj. Un contador es usado para dividir y obtener señales de 40Khz y 10Khz. La lógica de control y otros dispositivos en el sistema usan las tres frecuencias de reloj como base de tiempo.

Señal lógica de tiempo fuera del bus

Con una interfase asíncrona, una señal de fuera de tiempo debe ser proporcionada. El tiempo fuera finaliza un ciclo de bus si un dispositivo falla, para responder dentro de un tiempo asignado. Y un error en bus será señalado.

Un dispositivo puede fallar, direccionando una localidad no usada de memoria o atendiendo un ciclo de escritura en ROM. El tiempo fuera (bus timeout) en el computador tiene una duración de 10 microsegundos.

11.6 Elementos de programación del microprocesador

Como elementos de programación definimos a continuación los modos de direccionamiento.

Modos de direccionamiento

Los modos de direccionamiento son los caminos que sigue el procesador para localizar datos y manipularlos. Centro de los registros, una parte del registro especifica el código objeto para decirle cual registro usar. La dirección efectiva usa cualquier modo de direccionamiento excepto implícito ya que requiere un uso particular de registros.

Los modos de direccionamiento son:

1. Direccionamiento directo a registro
 - direccionamiento a registro de dato
 - direccionamiento a registro de direcciones
2. Indirecto con registro:
 - indirecto con registro de dirección
 - indirecto con registro de dirección y post-incrementado
 - indirecto con registro de dirección y pre-decrementado
 - indirecto con registro de dirección y desplazamiento
 - indirecto con registro de dirección e índice y desplazamiento de 8 bits.

3. Indirecto con memoria:

- Indirecto con memoria y post-índice*
- Indirecto con memoria y pre-índice*

4. Indirecto con contador de programa:

- Indirecto con PC y desplazamiento de 16 bits
- Indirecto con PC e índice y desplazamiento de 8 bits*
- Indirecto con PC e índice y desplazamiento de 16 ó 32* bits

5. Indirecto con memoria y con el contador de programa:

- Indirecto a memoria con PC y post-índice*
- Indirecto a memoria con PC y pre-índice*

6. Absoluto:

- Absoluto corto
- Absoluto amplio

7. Inmediato

Los modos marcados con un asterisco (*) o bien tienen funciones ampliadas en el 68020 o bien existen sólo en el 68020.

Codificación de direcciones

Las instrucciones del 68K codifican las direcciones de operandos de una manera específica. La primera palabra especifica la operación y, en muchos casos, la dirección de un operando. Las especificaciones de operandos adicionales van a continuación de esta primera palabra. En los primeros cuatro miembros de la familia 68K hay, como mucho, cuatro palabras de "ampliación". El 68020 permite hasta 10 palabras de ampliación (más que en los otros a causa de los modos adicionales de direccionamiento).

Las direcciones efectivas se codifican con ayuda de un campo modo y un campo registro. Cada uno de estos campos tiene una longitud de 3 bits y están embebidos en la primera palabra de instrucción. Cualquiera de las palabras de ampliación que sigan pueden ser sencillamente o bien direcciones de operandos, en forma de direcciones absolutas, o pueden ser formas bastante complejas con campos de direcciones embebidos o con modos de direccionamiento indirecto a memoria. Los campos de ampliación más complejos incluyen valores para indexación y direccionamiento indirecto, además de para la presencia o no y en su caso, tamaño de desplazamiento.

La siguiente tabla resume los campos del modo de direccionamiento tanto en el interior de la palabra instrucción como en el interior de las palabras de ampliación.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
código operación de la instrucción										modo			registro		

formas de direccionamiento efectivas básicas

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
%A	registro índice		%V	escala	0	desplazamiento									

palabras de ampliación

%A	registro índice	%V	escala	1	00	01	00	01	00	01	00	01	00	01	00
desplazamiento base (0, 1 ó 2 palabras)															
desplazamiento escala (0, 1 ó 2 palabras)															

palabras de ampliación, formato complejo (modo MCR20)

registro	registros de datos o direcciones	L16	Selección de registro/índice (modo MCR20)
modo	modo de direccionamiento	00 (base)	Tamaño de desplazamiento base (modo MCR20): 00 reservado 01 desplazamiento corto 10 desplazamiento de palabra 11 desplazamiento largo
código de operación	instrucción y posición referenciada sobre modo/registro para el desplazamiento que viene		
desplazamiento	valor con signo de 3 bits	01	Supresión de índice (modo MCR20)
escala	Índice de factor de escala (sólo en MCR20) 00=1X 01=2X 10=4X 11=8X	00 00L	Supresión de base (modo MCR20) 0: cualquier y offset registro base 1: cualquier registro base
		registro índice	Tamaño de registro índice 0: palabra estándar con signo 1: palabra larga con signo
		registro índice	registro de datos o de direcciones (000-111)
		D16	Tipo de registro índice 0: registro de datos 1: registro de índice

La siguiente tabla resume los valores modo/registro y sus funciones.

Clasificación modo/registro

Modo	Registro	Operación de direccionamiento
000	reg #	Directo a registro de datos
001	reg #	Directo a registro de dirección
010	reg #	Indirecto a registro de dirección
011	reg #	Indirecto a registro de dirección con post-incremento
100	reg #	Indirecto a registro de dirección con pre-incremento
101	reg #	Indirecto a registro de dirección con desplazamiento
110	reg #	Indirecto a memoria con registro de dirección con índice*
111	000	Abstracción corto
111	001	Abstracción largo
111	010	Indirecto con el contador de programa y desplazamiento
111	011	Indirecto a memoria con el contador de programa con índice*
111	100	Datos inmediatos
111	101-111	Reservado

* solo el BPCD

11.7 Modos de direccionamiento

Directo a registro de datos.

En el modo directo a registro de datos, un registro de datos contiene el operando.

Registro de datos

operando

Contiene el contenido de Dn

Directo a registro de direcciones.

En el modo directo a registro de direcciones, un registro de dirección contiene el operando.



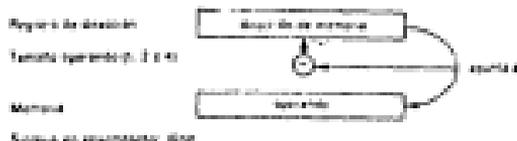
Indirecto con registro de dirección.

En el modo de direccionamiento indirecto con registro, un registro de dirección contiene la dirección del operando (esto es, "apunta" al operando).



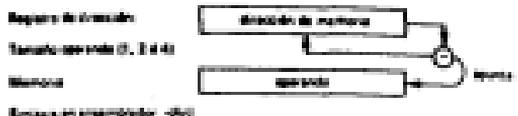
Indirecto con registro de dirección y post-incremento.

Como en el modo sencillo indirecto con registro de dirección, un registro de dirección, en el modo indirecto con post-incremento, contiene la dirección del operando. Sin embargo, después de usar el operando, el procesador suma al registro de dirección, la longitud del operando en bytes (esto es, 1 para una operación con 1 byte, 2 para una operación con palabra o 4 para una operación con palabra larga).



Indirecto con registro de dirección y predecremento.

Como en el caso sencillo del modo indirecto con registro de dirección, en el modo indirecto con pre-decremento, un registro de dirección contiene la dirección del operando.



Sin embargo, antes de usar el operando, el procesador resta la longitud del operando al registro de dirección (esto es, 1 para una operación de un byte, 2 para una operación con palabra o 4 para una operación con palabra larga).

Indirecto con registro de dirección y desplazamiento.

En el modo indirecto con desplazamiento, el operando reside en la dirección dada por la suma de los contenidos de un registro de dirección y un desplazamiento de 16 bits. El valor del desplazamiento es trasladado en signo a 32 bits para permitir desplazamientos negativos.

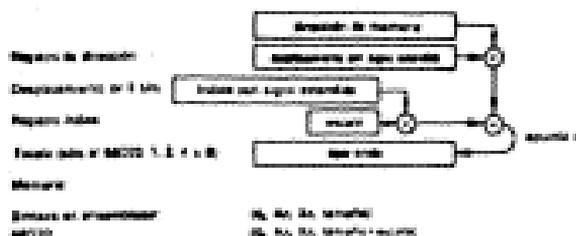


Indirecto con registro de dirección e índice y desplazamiento.

En el modo indirecto con índice y desplazamiento el operando reside en una dirección dada por la suma de los contenidos de un registro de dirección, un desplazamiento y el producto del registro índice por un factor de escala.

Si la base del desplazamiento es de 2 ó 16 bits, está el signo trasladado a 32 bits. (Hay que hacer notar que la versión de 8 bits de este tipo de direccionamiento sólo existe en el 68020.)

El valor del registro índice sólo puede ser un valor de 16 ó 32 bits; si es 16 bits, el procesador traslada el signo a 32 bits antes de añadirlo. El factor de escala puede ser 2D(1), 2E(2), 2E(4) ó 2E(8). Sólo el 68030 admite índices escalados.

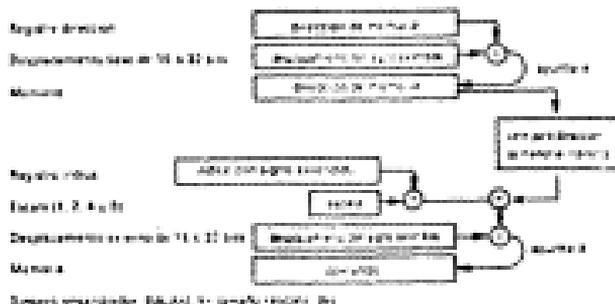


Indirecto con memoria y post-índice.

En el modo indirecto con memoria y post-índice, la dirección del operando se obtiene a partir de cuatro valores: el contenido de un registro de dirección, un desplazamiento base de 16 ó 32 bits, el producto del valor en un registro índice por la escala y un segundo desplazamiento (interno).

La base y el desplazamiento externo puede tener una longitud de 16 ó 32 bits, el procesador traslada el signo de los valores antes de añadirlos. El registro índice es también ampliado y trasladado el signo (cuando se requiera) antes de añadirlo a la dirección efectiva.

Los cuatro valores son opcionales; si se omiten, el procesador suma un valor cero de contribución al cálculo de la dirección efectiva. Al calcular la dirección efectiva, el procesador añade el valor del registro de dirección al desplazamiento base. Después utiliza este valor para obtener un segundo valor de dirección. Para este segundo valor, el procesador añade la escala del registro índice. Finalmente, añade el desplazamiento externo para obtener la dirección del operando.

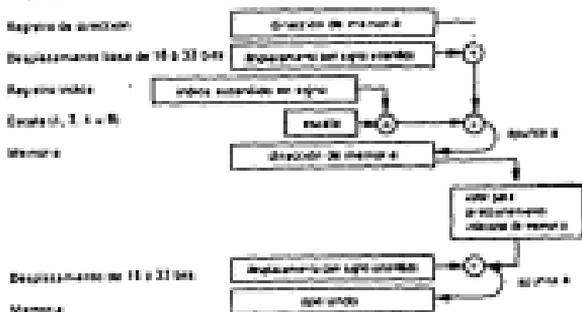


Indirecto con memoria y pre-índice.

En el modo indirecto con memoria y pre-índice (exclusivo del 68020), la dirección del operando se obtiene a partir de cuatro valores: el contenido de un registro de dirección, un desplazamiento base de 16 ó 32 bits, el producto de valor de un registro índice y la escala y un segundo desplazamiento externo.

La base y el desplazamiento externo puede ser de 16 ó 32 bits; si son de 16 bits el procesador traslada el signo antes de sumarlos. El registro índice es también ampliado y su signo trasladado (cuando sea necesario) antes de ser sumado a la dirección efectiva. Los cuatro valores son opcionales; si se omiten, el procesador supone un valor cero para esa contribución a la dirección efectiva.

Al evaluar la dirección efectiva, el procesador suma el valor del registro de dirección al desplazamiento base y el valor escala del registro índice. Después utiliza este valor para obtener una segunda dirección. Para calcular esta segunda dirección, el procesador suma el desplazamiento externo consiguiendo así la dirección del operando.



Indirecto con PC y desplazamiento.

En el modo indirecto con PC y desplazamiento, el procesador genera una dirección efectiva sumando el valor del contador de programa y el valor con el signo trasladado de esa palabra de ampliación de 16 bits.



Indirecto con PC e índice y desplazamiento.

En el modo indirecto con contador de programa (PC) y desplazamiento, la dirección efectiva es la suma de la dirección en el contador de programa el valor en la palabra de ampliación y el valor en el registro índice (multiplicado por la escala, en el 68020).

El valor en el contador de programa en el momento de la evaluación será la dirección de la palabra de ampliación. El procesador trasladará el signo del desplazamiento si es de 6 a 16 bits (la versión de ocho bits es sólo en el 68020).

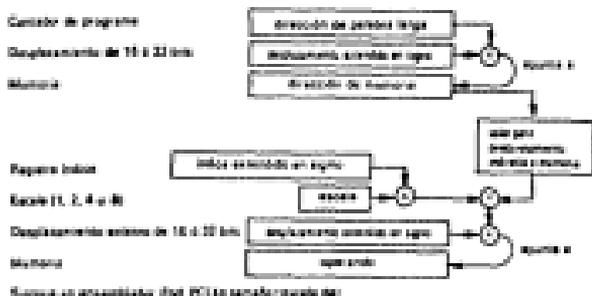
El índice del valor en registro índice puede ser trasladado, si es necesario, así como el del valor escalado, en el procesador 68020.



Indirecto a memoria con PC post-índice.

En el modo indirecto a memoria con PC y post-índice, la dirección efectiva se calcula a partir de cuatro valores. La suma del PC y un desplazamiento base se utiliza como una dirección. El procesador suma el valor de dicha dirección con una escala en el registro índice y con el desplazamiento externo, obteniendo así la dirección del operando.

El valor del contador de programa es el de la dirección de la palabra de ampliación. La base y el desplazamiento externo pueden ser de 16 ó 32 bits. El valor en el registro índice puede ser 1, 2, 4 u 8. Todos los componentes utilizados en el cálculo de la dirección del operando son opcionales; si se omiten, el procesador assume para ellos el valor cero.

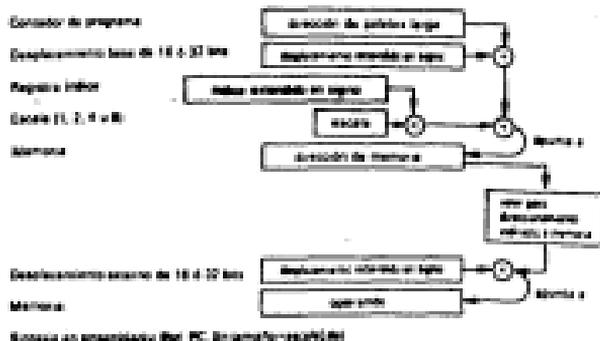


Indirecto a memoria con PC y pre-índice.

En el modo indirecto a memoria con PC y pre-índice, el cálculo de la dirección efectiva se hace con cuatro valores. La suma del PC, con el desplazamiento base y con la escala en el registro índice, se usa como una dirección de partida. El procesador suma el valor de esa dirección con el desplazamiento externo para dar la dirección del operando.

El valor del contador de programa es la dirección de la palabra de ampliación. Los desplazamientos base y externo pueden ser de 16 ó 32 bits; si es de una longitud de 16 bits, el procesador traslada el signo antes de usar el valor.

El valor en el registro índice puede estar en un byte, una palabra o una palabra larga y su signo se traslada si es necesario. El valor de escala puede ser 1, 2, 4 u 8. Todos los componentes utilizados en el cálculo de la dirección del operando son opcionales; si se omiten, el procesador supone un valor cero.



Absoluto corto.

En el direccionamiento absoluto corto la dirección del operando es el valor, con el signo trasladado, de la palabra de 16 bits de aplicación que sigue a la instrucción.



Absoluto largo.

En el direccionamiento absoluto largo, la dirección del operando está en la palabra de aplicación (32 bits) que sigue a la palabra instrucción.



Dato inmediato.

En el direccionamiento inmediato, el propio operando sigue a la propia instrucción. Dependiendo del tamaño del operando, la instrucción necesita una o dos palabras de ampliación.



11.8 Conjunto de instrucciones.

EL M68000 cuenta con un set de instrucciones largo y ortogonal, se dice que son ortogonales debido a que diferentes instrucciones de adición tienen el mismo modo de direccionamiento.

Captura anticipada de líneas de espera

Algo especial en el diseño del procesador M68000 es una instrucción de captura anticipada de líneas de espera (prefetch queue). Mientras una instrucción comienza a ejecutarse, otra puede ser decodificada y otra puede ser capturada.

Tipo de datos

El M68000 puede trabajar con bits, cuarteos (nibbles), bytes, palabra y palabra larga. Estos tipos de datos trabajan con bastantes instrucciones lo que provee una gran flexibilidad de programación.

Definiremos las instrucciones del MC 68000 en forma de grupos de instrucciones, de modo que sea más clara la función de cada instrucción y más concreta. Los grupos de instrucciones son los siguientes:

- Movimiento de datos
- Aritméticas enteras
- Decimales
- Lógicas
- Corrimiento y rotación
- Manipulación de bit
- Control de programa
- Control de sistema
- Instrucciones nulas

Instrucciones de movimiento de datos.

Estas instrucciones se describen primero debido a que se usan al principio de programas. Cualquier programa necesita mover varios bits y bytes hacia fuera o dentro del CPU y memoria. Las siguientes son las instrucciones de movimiento.

EXG, LEA, LINK, MOVE, MOVEM, MOVEP, MOVES, PEA, SHAP, UNLK.

EXG es una instrucción que intercambia el contenido del operando destino con el contenido del operando fuente.

LEA es una instrucción de carga de dirección efectiva, la dirección efectiva es información de dirección calculada desde el modo de direccionamiento y especifica un dato. Este es puesto en un registro de dirección y puede usarse para acceder una tabla en memoria.

LINK (enlace) y UNLINK (des-enlace) son instrucciones avanzadas, sirven para manipulación de pila en cuanto a organización de áreas de memoria, para subrutinas, funciones y módulos de programa.

MOVE cuando combinase con el modo de direccionamiento, byte, palabra y palabra larga, esta simple instrucción es capaz de mover cualquier pedazo de información donde quiera que las líneas del bus del microprocesador lleguen. MOVE puede mover información de un registro a otro registro, de un registro a memoria, o de memoria a memoria.

MOVEM esta instrucción mueve los datos en grupos de registros hacia o desde memoria. Es muy usada ya que ahorra trabajo de programación y en lenguajes de alto nivel donde es necesario salvar y restaurar información.

MOVEP esta instrucción sirve para mover datos con los periféricos, facilita la transferencia de información enviada o recibida por el CPU con los periféricos.

MOVES es una instrucción de movimiento rápido debido a esto solo puede mover bytes de datos inmediatos a registros de datos.

PEA sirve para empujar la dirección efectiva dentro de la pila, esta instrucción decrementa el apuntador de pila por dos (hay dos bytes en una palabra) y pone la palabra baja de la dirección efectiva en la pila, entonces el apuntador de pila es decrementado otra vez por dos y la palabra alta de la dirección efectiva es puesta o empujada dentro de la pila.

SWAP esta instrucción sirve para mover palabras, intercambiando palabra baja y alta de un registro de datos.

Instrucciones aritméticas.

Las instrucciones aritméticas son las siguientes

ADD, ADDX, CLR, CMP, DIVS, DIVU, EXT, MULL, MULD, NEG, NEGE, SUB, SUBX, TAG, TST.

Mientras el MC68000 puede ejecutar las mismas operaciones de adición y sustracción que los otros microprocesadores, además puede ejecutar instrucciones de multiplicación y división.

ADD es una instrucción de adición aritmética binaria, suma el operando fuente y operando destino, el resultado es almacenado en el operando destino, borrando el valor anterior de este operando.

ADDX es una instrucción que ejecuta suma extendida ya que puede sumar bytes, palabras y palabras largas. Hay dos formas de addx: registro a registro y memoria a memoria.

CLR es la instrucción más simple, pone cero en el destino. El operando destino pueda ser un registro de datos o una localidad de memoria y es llenada con tantos ceros como el tamaño de su especificación (byte, palabra o palabra larga).

CMP comparará un operando fuente con un operando destino (sustrayendo el contenido de la fuente con el contenido del destino).

DIVS y DIVU son instrucciones de división signada y no signada respectivamente. DIVU divide un operando destino de 32 bits entre un operando fuente de 16 bits usando números binarios no signados. El resultado es de 32 bits se almacenado en el destino (el cual debe ser un registro de datos). DIVS hace exactamente lo mismo pero usando números signados.

EXT extiende el bit de signo de un número binario que puede ser interpretado como un número signado o no signado. Cuando se interpreta como signado el bit más significativo es leído como el signo del número : un cero significa positivo y un uno significa negativo.

MULL y MULD son instrucciones de multiplicación, una de las ventajas de los 16 bits del 68000 es que tiene multiplicación y división, circuito integrado de 8 bits tienen que ejecutar varias operaciones con rutinas de movimiento, corrimiento y comparación. Para obtener una división o una multiplicación.

MULU multiplica no signados y MUIS multiplica signados. MUDU multiplica dos operandos de 16 bits no signados produciendo un resultado de 32 bits no signado. MUIS multiplica dos operandos de 16 bits signados y produce un resultado de 32 bits signado.

NEG es una instrucción de negación. El valor del operando destino es sustraído de cero usando complemento binario a dos, el resultado es almacenado en el destino.

NEGX es una instrucción de negación extendida y es una forma especial de NEG que afecta la bandera X. El operando y la bandera X son sustraídos de cero. NEGX intenta simplificar trabajo aritmético de múltiple-precisión.

SUB es una instrucción de sustracción binaria, el operando fuente es sustraído del operando destino.

SUBX es una instrucción de sustracción extendida.

TAS y TST son operaciones especiales de comparación, TST sustrae cero del contenido del destino y manipula las condiciones de código de banderas de acuerdo al resultado. TAS es más compleja, puede trabajar con un solo byte y puede direccionarse de ocho diferentes maneras.

Instrucciones decimales.

Aunque los computadores trabajen con 1's y 0's binarios en toda la aritmética esta se hace usando las reglas básicas.

ABCD es una instrucción de suma usando código BCD con extensión, puede sumar dos operandos que estén en memoria.

SBCD es una instrucción que sustrae datos en BCD y actúa de la misma forma que ABCD.

NEBCD es la negación BCD con extensión, el operando es negado por la sustracción de este y de la bandera X con cero.

Existen instrucciones avanzadas dentro de este grupo que son:

ABCD, B0AC, BCLR, BSET, CHK, DBC, DLEAD, LEA, LINK, NBCD, PEA, RESET, RTE, RTR, SBCD, Sec, SWAP, TAS, TRAP, TRAPV, TST, UNLK.

Instrucciones lógicas.

Todas las computadores digitales hacen uso pesado de operaciones lógicas, el MC68000 provee las siguientes instrucciones lógicas:

AND, OR, EOR, NOT.

Operaciones lógicas son usadas para limpiar, encender y probar posiciones específicas de bits dentro de bytes, palabras y palabras largas.

AND compara el operando fuente con el operando destino, bit por bit en cada posición siguiendo la regla de la operación AND. El resultado es almacenado dentro del operando destino. AND es usado generalmente para crear máscaras.

OR compara el operando fuente con el operando destino bit por bit de acuerdo a la operación lógica OR, el resultado es almacenado en el operando destino. OR también es usado para crear máscaras.

XOR compara el operando fuente con el operando destino bit por bit de acuerdo a la operación lógica XOR. El resultado es almacenado en el operando destino. Esta instrucción es usada para invertir bits.

MOT completa el valor del operando destino y almacena el nuevo valor en el operando destino. Esta instrucción hace el complemento a uno.

Instrucciones de corrimiento y rotación.

Estas instrucciones no corresponden con lenguajes de alto nivel. La habilidad de mover o cambiar el contenido de un operando con localidades de operandos no usadas. Las instrucciones de corrimiento y rotación son:

ASL, ASR, LSL, LSR, ROL, ROR, ROLX, RORX.

ASL es una instrucción que hace un corrimiento aritmético a la izquierda, el término aritmético es para diferenciarlo del lógico.

ASR es una instrucción que hace un corrimiento aritmético a la derecha, mueve el valor de bits hacia la derecha.

LSL es una instrucción que hace un corrimiento lógico a la izquierda poniendo un cero al bit menos significativo. El bit más significativo que sale es almacenado en las banderas C y X.

LSR es una instrucción que hace un corrimiento lógico a la derecha.

ROL es una instrucción de rotación a la izquierda, el bit menos significativo es copiado en la bandera C y copiado también en la posición del bit más significativo.

ROR es una instrucción de rotación hacia la derecha.

ROL es una instrucción de rotación hacia la izquierda extendida, el bit más significativo es copiado a la bandera Z y el contenido de esta bandera es copiado en la posición del bit menos significativo.

ROR es una instrucción de rotación hacia la derecha extendida similar a ROL.

Instrucciones de manipulación de bit.

Estas instrucciones pueden trabajar en bits individuales en lugar de bytes, palabras o palabras largas. Existen 4 tipos de manipulación de bit en el MCS8600:

BTEST, BSET, BCLR, BCHG.

BTEST es una instrucción de prueba de bit, simplemente prueba el valor de una localidad particular de bit y usa la bandera Z para comunicar el valor. Si la bandera Z está encendida, el bit es 1, si está apagada el bit es cero.

BSET es una instrucción de prueba y encendido de bit, después de probar un bit particular, BSET solo trabaja sobre la bandera Z. BSET prueba un bit particular, trabaja también sobre la bandera Z y después enciende el bit probado poniéndolo en el valor 1.

CAPITULO III
ANALISIS DEL SISTEMA

III ANALISIS DEL SISTEMA

El análisis es un proceso lógico, el objetivo de esta fase no es resolver el problema inmediatamente, pero sí determinar exactamente qué es lo que debe hacerse para resolverlo. Una metodología estructurada es muy útil ya que con una aproximación estructurada, el criterio de salida específico debe ser cumplido para cada paso del proceso. El objetivo básico de la etapa de análisis es desarrollar un modelo lógico de el sistema, usando herramientas tales como diagramas de flujo de datos, un diccionario elemental de datos, descripciones esquemáticas de los algoritmos relevantes. Este modelo lógico es sujeto de revisiones tanto del usuario como de los analistas, quienes deben concluir que el modelo refleja realmente lo que debe hacerse para resolver el problema.

El diagrama de flujo de datos tiene como objeto mostrar las transformaciones de los datos a medida que estos fluyen a través de los procesos del programa; es decir ayuda a analizar los cambios que ocurren a los datos de entrada a fin de lograr la salida deseada.

III.1 Conceptualización del problema

Definiremos primero el objetivo del sistema para hacerlo más claro. Se quiere desarrollar un sistema que partiendo de un programa en lenguaje ensamblador genere su correspondiente lenguaje de máquina. Y esto que quiere decir? Bueno comenzaremos por escribir lo que un programador necesita para ensamblar sus programas, el lenguaje ensamblador son un conjunto de instrucciones llamadas mnemónicos, estos tienen un código equivalente en lenguaje máquina que se arma a través de tablas en las cuales se toman muchos aspectos en cuenta, tales como: modo de direccionamiento, registros usados, tamaño de los operandos, cual de estos operandos es operando destino y cual es operando fuente, dirección efectiva, etc. Y ahora hablando del término macroensamblador, están más conceptos como: macro definición, llamadas a macros, literales, símbolos, etc. De este modo tenemos que a parte de ser complejo programar en lenguaje ensamblador es muy tardado ya que ensamblar las instrucciones a código máquina es una tarea muy laboriosa.

Por lo que el objetivo de este sistema es resolver la parte de ensamble y macrodefiniciones que existan dentro del programa en lenguaje mnemónico, para que de este modo, el usuario del sistema aplique su labor de prueba y ejecución de sus programas.

III.2 Metodología de análisis

Definiremos ahora una primera aproximación hacia la solución del problema, en la que se incluye todo el sistema de una forma general, y partiendo de esta empezar a modelar el sistema. Como entrada al sistema se tiene un programa en lenguaje ensamblador que se llamará LENGUAJE FUENTE, un proceso en el que se llevarán a cabo todas las transformaciones de la entrada, que se llamará MACROENSAMBLADOR, el cual tiene otras dos entradas que son DIRECTIVAS y CODIGO DE INSTRUCCION, la salida del proceso será un programa en lenguaje de máquina que se llamará PROGRAMA OBJETIVO. Empleando el método de análisis por diagramas de flujo de datos (DFD), la siguiente figura 1 muestra esta primer aproximación.

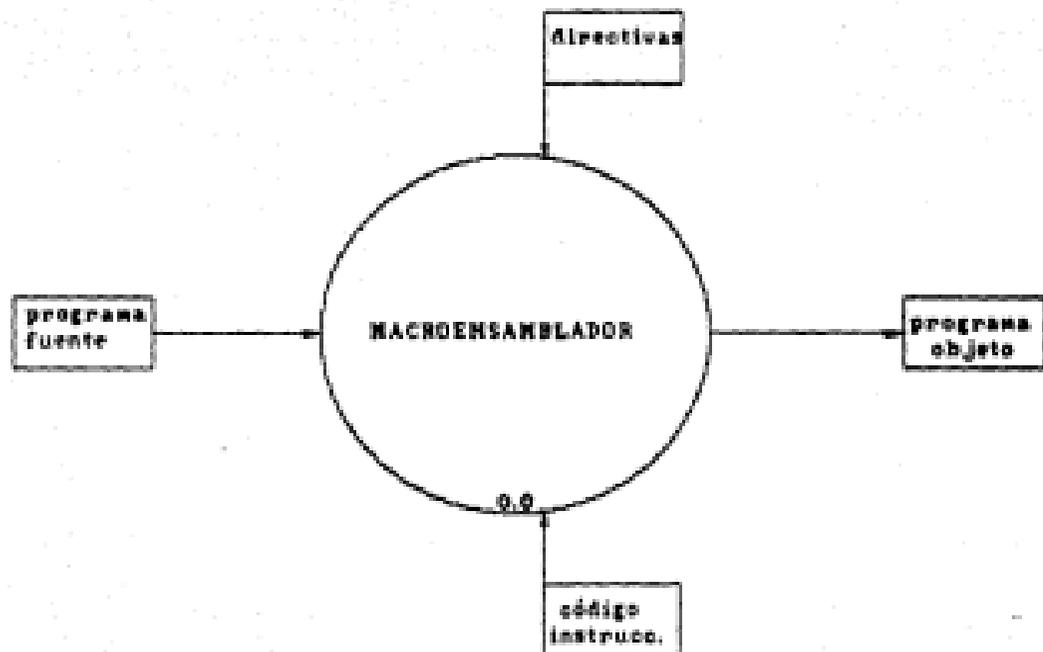


FIG. 1

La siguiente etapa consiste en dar un paso hacia adentro del sistema, para definirlo más a detalle. El macroensamblador se puede separar en dos procesos que llamaremos: MACROPROCESADOR y ENSAMBLADOR, el primero tiene como entrada el código fuente y lo transforma en código expandido, el segundo toma como entrada el código expandido y lo transforma en código objeto, más adelante veremos como se hacen estas acciones los procesos. La gráfica número 2 muestra esta segunda aproximación.

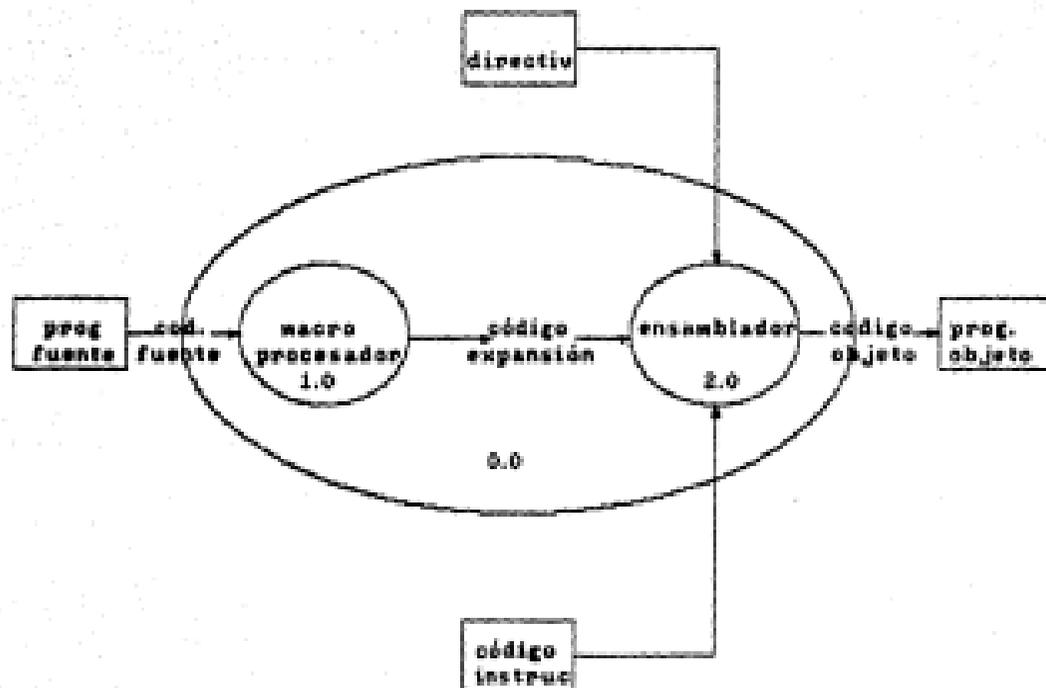


FIG. 2

La tercera aproximación del análisis del sistema es definir por separado el macroprocesador y el ensamblador.

Macroprocesador

El macroprocesador lo definiremos subdividiéndolo en dos procesos que llamaremos: RECOPIRAR MACRO DEFINICIONES y EXPANDIR MACRO LLAMADAS. En el primero se genera un contenedor de datos donde serán guardados los nombres y cuerpos de macros, en el segundo se sustituyen las llamadas a macros con los cuerpos correspondientes de estas macros, y de esta forma generar el código de expansión.

Recopilar Macro Definición

Definiremos ahora como estará formado el proceso de recopilar macro definiciones, lo podemos subdividir en dos procesos: IDENTIFICAR MACRO DEFINICION e IDENTIFICAR EXPANSION, el primero solo toma los nombres de las macros y los guarda en un contenedor, el segundo toma el cuerpo de la macro y lo guarda en otro contenedor de datos.

La gráfica 3 muestra el macroprocesador y la recopilación de macros.

MACROPROCESADOR

RECOPILA MACROS

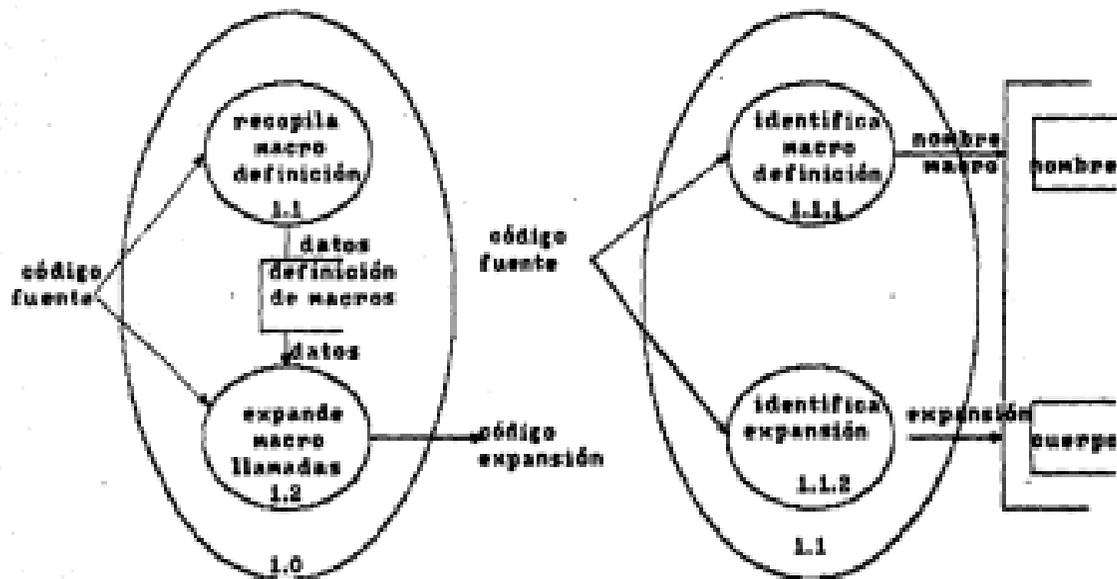
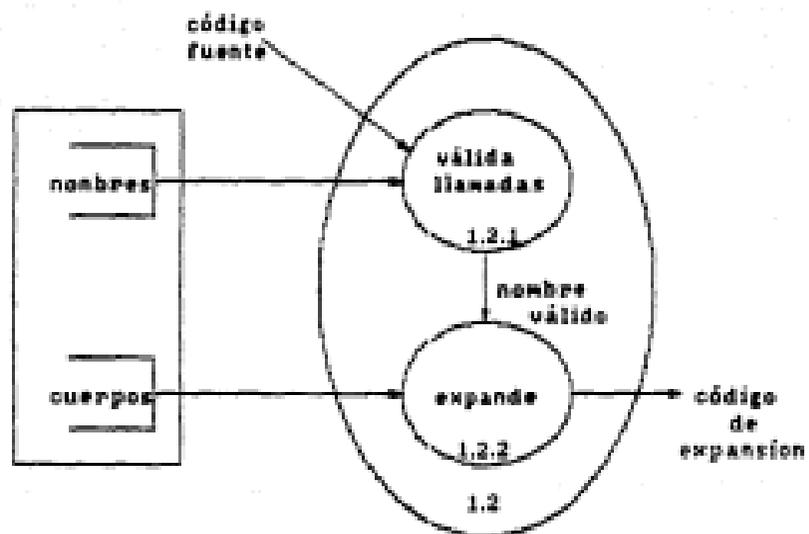


FIG. 3

Expandir macro llamadas.

El proceso de expandir macro llamadas, lo definiremos como otros dos procesos: VALIDAR LLAMADA y EXPANSIÓN. El proceso de validar llamada tiene como entrada los nombres de las macros y también el código fuente, y su función es verificar que el nombre de la macro recolectada existe dentro del código fuente y dar como salida los nombres de macros válidos.

El subproceso de expansión, tiene como entrada el cuerpo de la macro y el nombre válido, con este nombre válido localiza el cuerpo que le corresponde y lo sustituye en el lugar donde hizo la llamada a macro. Como salida del proceso de expansión tenemos el código de expansión. La gráfica 4 muestra este proceso.



EXPANDE MACROLLAMADAS

Ensamblador.

La otra parte o subproceso del macroensamblador que hace falta definir es el ensamblador. En este proceso la entrada son directivas, código fuente expandido y código de instrucción. La transformación de estos tres datos tiene como resultado o salida el código objeto. La gráfica 3 muestra este proceso.

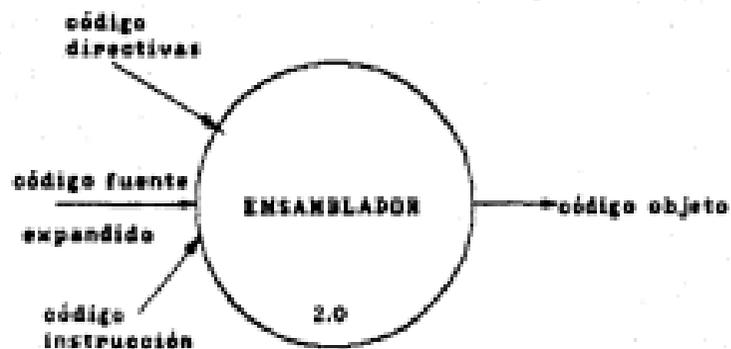


FIG. 5

El proceso de ensamble lo podemos definir o subdividir en dos procesos que llamaremos : **RECOPILAR SÍMBOLOS y LITERALES y GENERADOR DE INSTRUCCIONES.**

Recopilar símbolos y literales.

Este subproceso tiene como entradas el código de instrucción, el código fuente y las directivas. La función de este proceso es como su nombre lo indica, tomar solo los símbolos y las literales que serán la salida de este proceso.

Generador de instrucciones.

El generador de instrucciones tiene como entradas el código de instrucción, el código fuente , las directivas, literales y símbolos. Con todos estos elementos el generador de instrucción arma la instrucción en lenguaje máquina ya que la salida es el código objeto. La gráfica 6 muestra este proceso.

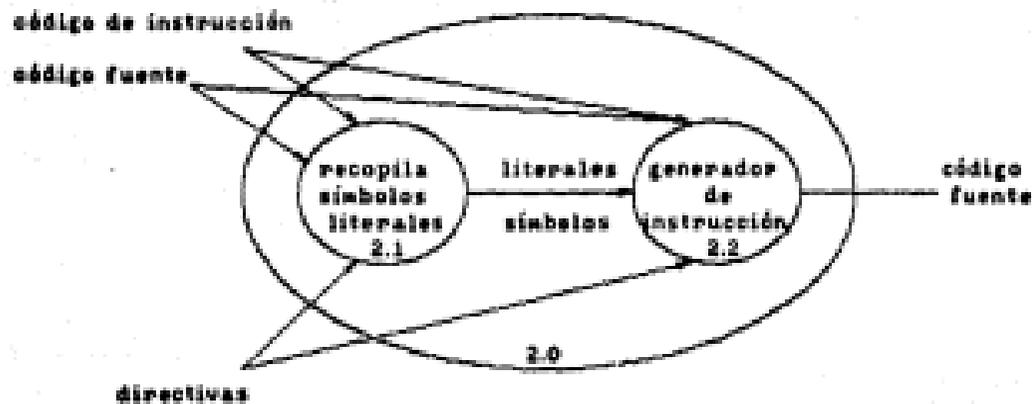


FIG. 6

El proceso de recolección de símbolos y literales, lo volvemos a subdividir en dos procesos, uno que IDENTIFIQUE SÍMBOLOS y otro que IDENTIFIQUE LITERALES.

El proceso generador de instrucción también se puede subdividir en dos procesos más : uno de VALIDACIÓN DE INSTRUCCION y otro de ARMAR OBJETO. La gráfica 7 muestra los procesos descritos.

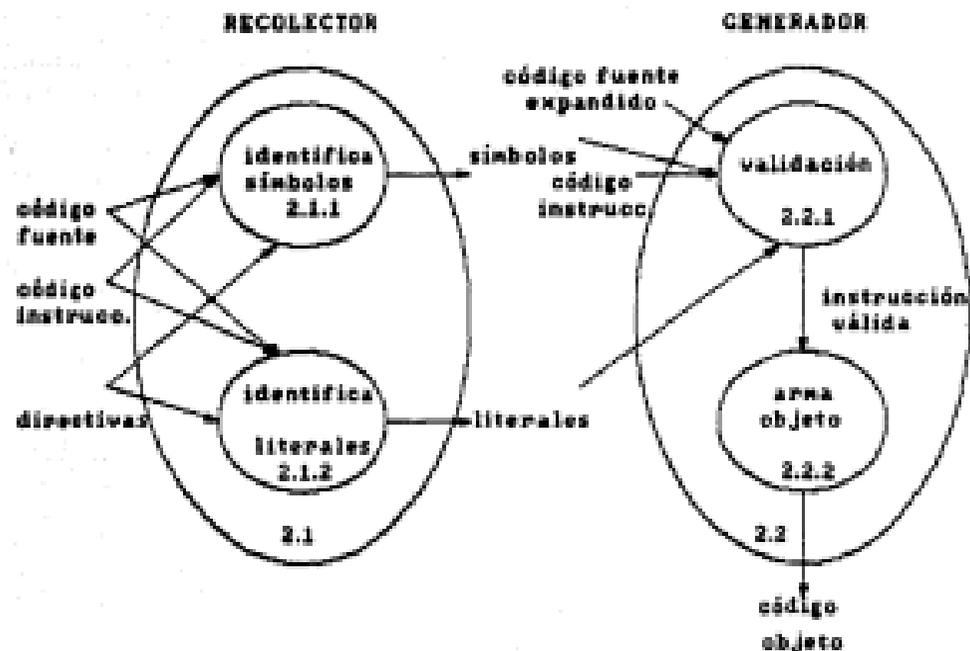


FIG. 7

Con todos estos diagramas estructurados se tiene una aproximación definida de lo el sistema hace.

CAPITULO IV
DISEÑO DEL SISTEMA

IV DISEÑO DEL SISTEMA

Una vez que la etapa de análisis ha quedado terminada, el siguiente paso es determinar, en forma de un esbozo amplio, como el problema puede ser resuelto. Durante el diseño del sistema, el objetivo es moverse de lo lógico a lo físico.

Pueden ser consideradas distintas alternativas de solución, la cuestión a ser resuelta durante el diseño del sistema es ¿cómo, en general, puede ser solucionado el problema?.

Varias son las etapas que debe cubrir el diseño de sistemas:

1. Especificación del problema.
2. Identificación de módulos.
3. Identificación de entradas y salidas.
4. Especificación de estructuras de datos.
5. Repetir los pasos del 1 al 3 para cada módulo.

IV.1 Especificación del problema

Para el desarrollo del ensamblador es necesario separar la problemática en una serie de etapas lógicas que en forma posterior servirán como módulos "naturales" del sistema.

Atacemos primeramente el problema. Tratamos de desarrollar un programa traductor de instrucciones en ensambladores a instrucciones de máquina, las instrucciones que vamos a traducir pueden ser macroinstrucciones, instrucciones con símbolos y literales, instrucciones simples yseudo instrucciones (directivas).

Existe una relación uno a uno entre las instrucciones simples llamadas por ensambladores y operandos que pueden ser constantes, registros de direccionamiento o registros de datos; y las instrucciones en código máquina. Un primer intento de diseño nos permite reconocer 2 etapas principales del problema:

- 1 Convertir las instrucciones "complejas" en instrucciones simples.
- 11 Convertir las instrucciones simples a su correspondiente código de máquina.

Por una instrucción compleja se entiende cualquier instrucción que agrupe bajo su nombre a un conjunto de instrucciones o bien una instrucción cuyos operandos son literales o símbolos sin un valor directo (no registros ni constantes), todas estas instrucciones son llamadas macroinstrucciones y pseudoinstrucciones. Conforme a la definición dada de macroinstrucción en los capítulos anteriores, los pasos a seguir para descomponerla en sus instrucciones simples serán:

- Recopilar las definiciones de Macros.
- Conservar el cuerpo de la Macro.
- Identificar la llamada de la Macro.
- Expandir o sustituir el cuerpo de la Macro.

Hasta este momento el conjunto de instrucciones que tiene nuestro programa "fuente" son instrucciones donde no aparecen llamadas a Macros. El siguiente paso es asignar valores a los literales y símbolos que pueden aparecer en el programa, es decir, procesar cierto tipo de pseudoinstrucciones:

- Identificar símbolos y guardar su valor.
- Identificar literales y guardar su valor.

Ahora se está en condiciones de efectuar la conversión de las instrucciones en ensamblador a las instrucciones de máquina, existen dos pasos que se deben efectuar para ello:

- Validar la sintaxis de la instrucción.
- Armar el código objeto.

IV.2 Identificación de los módulos principales.

Con las etapas obtenidas al especificar el problema parece natural que el sistema tenga tantos módulos como los pasos que se mencionaron, de manera que un diagrama como el de la figura 8 cubra todos los pasos. Esta figura se conoce como CARTA DE ESTRUCTURA.

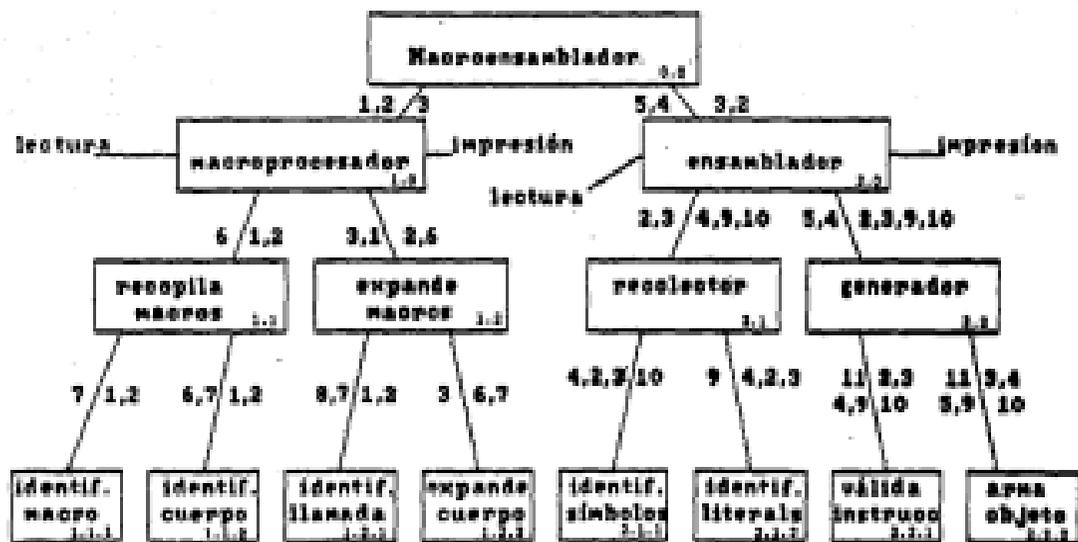


FIG. 8

IV.3 Identificación entradas y salidas

Con la identificación de los módulos principales, el siguiente paso en el diseño del sistema consiste en reconocer las entradas y salidas de cada uno de esos módulos, siguiendo un proceso lógico como el llevado a cabo para el desglose del sistema en sus módulos, (todos los números de entradas y salidas se refieren a la figura 8) tenemos:

-Módulo macroensamblador (0.0)

Este módulo representa al sistema como un todo, de manera que las entradas y salidas son funciones que como objetivo fueron planteadas para el sistema, es decir.

Entradas :

- 1 : Programa fuente.
- 2 : Directivas.
- 3 : Código de la instrucción.

Salidas :

- 4 : Código objeto.

El macroensamblador deberá efectuar dos funciones básicas:
Proceso de macroinstrucciones y ensamblado de instrucciones simples.

- Módulo de macroprocesador (1.0)

El objetivo del módulo es identificar las macroinstrucciones y sustituir su cuerpo en las líneas a estas macros que aparezcan en el programa.

Entradas:

- 1 : Programa fuente.
- 2 : Directivas.

Salidas:

- 3 : Código expandido.

- Módulo de ensamblador (2.0)

El objetivo de este módulo es traducir las instrucciones simples a instrucciones en lenguaje de máquina. Asignando valores a las literales y simbólicas, y sustituyendo estos valores en las instrucciones del programa fuente para formar instrucciones simples.

Entradas:

- 1 : Programa expandido.
- 2 : Directivas.
- 4 : Código de la instrucción.

Salida:
5 : Código objeto.

El nivel siguiente de módulos espera los objetivos de su módulo predecesor conforme a funciones claramente definidas:

-Módulo que recopila macrodefinición (1.1)

El propósito de este módulo es obtener la definición de la macroinstrucción y almacenarla junto con su cuerpo para el caso en que sea llamada en el programa.

Entradas:
1 : programa fuente.
2 : directivas.

Salidas:
6 : Tabla de macroinstrucciones recopiladas.

- Módulo expande macrosllamadas (1.2)

El propósito de este módulo es identificar las llamadas a macroinstrucciones, verificar que se encuentren en la tabla obtenida por el módulo recopila macrodefinición y sustituirla por el cuerpo de macroinstrucción en caso que sea válida.

Entradas:
1 : programa fuente.
2 : Directivas.
6 : Tabla de macroinstrucciones recopiladas.

Salidas:
3 : Código expandido.

-Módulo recolector (2.1)

El objetivo de este módulo es identificar símbolos y literales, asignarles valores y almacenarlos en una tabla para futuras referencias. El valor que tome una literal o símbolo puede quedar fijado por la pseudoinstrucción que lo declare o por la posición del símbolo en el programa (aquí hablamos de etiquetas).

Entradas:
3 : Código expandido.
2 : Directivas.

Salidas:
9 : Tabla de literales.
10 : Tabla de símbolos.

= Módulo generador (2.2)

El objetivo de este módulo es generar la instrucción de máquina correspondiente a la instrucción del código expandido, para lograr generarla, verificará que la sintaxis de la instrucción sea válida.

Entradas:

- 2 : Directivas.
- 3 : Código expandido.
- 9 : Tabla de literales.
- 10 : Tabla de símbolos.
- 4 : Código de instrucción.

Salidas:

- 5 Código objeto.

El nivel más bajo de diseño de módulos, es digamos, el nivel de programación inmediata, cada uno de estos módulos realiza funciones simples y perfectamente definidas.

-Módulo de identificación de macrodefinición (1.1.1)

Este módulo buscará en el programa fuente la declaración de alguna macrodefinición, validará su sintaxis y procesará directivas correspondientes.

Entradas:

- 1 : Programa fuente.
- 2 : Directivas.

Salidas:

- 7 : Nombre de la llamada.

-Módulo identifica cuerpo de la macroinstrucción (1.1.2)

con el nombre de la macro, el módulo de identificación del cuerpo de la macroinstrucción abrirá una entrada en la tabla de macros para almacenar el conjunto de instrucciones agrupadas por esta macro, verificará que el armado de macro sea correcto y procesará algunas directivas.

Entradas:

- 1 : Programa fuente.
- 2 : Directivas.
- 7 : Nombre de la macro.

Salidas:

- 6 : Tabla de macroinstrucciones.

-Módulo que identifica llamada a macro (1.2.1)

El módulo tomará el campo de código de operación de la instrucción y buscará en la tabla de definición de macroinstrucciones, en caso de encontrarlo, le dirá a su módulo predecesor que existe una llamada válida y que deberá proseguir a expandirla; si no, guardará la instrucción en el programa expandido.

Entradas:

- 1 : Programa fuente.
- 2 : Directivas.

Salidas:

- 7 : Nombre de la macro.
- 8 : Llamada válida.

- Módulo que expande el cuerpo de la macro (1.2.2)

El módulo tomará cada instrucción que forma a la macroinstrucción y la almacenará en el código expandido, así como su nombre.

Entradas:

- 6 : Tabla de macroinstrucciones.
- 7 : Nombres de las macros.

Salidas:

- 3 : Código expandido.

- Módulo identifica símbolos (2.1.1)

El programa fuente expandido es leído instrucción por instrucción, se toma el campo de código de operación y se analiza si corresponde a un mnenónico válido de operación, en caso de no ser válido, se almacena en la tabla de símbolos justo con el valor que le corresponde, el valor será la localidad de memoria contenida en el contador de localidades hasta ese momento.

Entradas:

- 2 : Directivas.
- 3 : Código expandido.
- 4 : Código de instrucción.

Salidas:

- 10 : Tabla de símbolos.

- Módulo que identifica literales (2.1.3)

El programa fuente expandido es leído instrucción por instrucción, se toma el código de operación, si este código representa una instrucción para el macroensamblador, entonces, se toma el valor del contador de localidades y se almacena este valor junto con la identificación de la literal en la tabla de literales.

Entradas:

- 1 : Código expandido.
- 2 : Directivas.
- 4 : Código de la instrucción.

Salidas:

- 9 : Tabla de literales.

- Módulo de validar instrucción (2.2.1)

El propósito de este módulo es verificar la sintaxis de la instrucción en términos del código de operación y modos de direccionamiento.

Entradas:

- 2 : Directivas.
- 3 : Código expandido.
- 4 : Código de la instrucción.
- 9 : Tabla de símbolos.
- 10: Tabla de literales.

Salida:

- 11 : Instrucción válida.

- Módulo arma objeto (2.2.2)

Después de haber verificado la sintaxis de la instrucción, el siguiente paso es traducirla a lenguaje de máquina. El módulo arma objeto toma la instrucción del programa fuente expandido, la busca en la tabla de código de la instrucción, la ensambla y la almacena en el programa objeto.

Entradas:

- 3 : Código expandido.
- 4 : Código de instrucción.
- 9 : Tabla de literales.
- 10: Tabla de símbolos.
- 11: Instrucción válida.

Salida:

- 5 : Código objeto.

IV.4 Especificación de Estructuras de datos

1. Programa fuente.

Contiene el conjunto de instrucciones que serán ensambladas. Las instrucciones pueden ser instrucciones para el microprocesador 68030 o directivas.

2. Tabla de nombres de macros.

Donde se almacenarán los nombres de las macros.

3. Tabla de argumentos de macros.

Donde se almacenarán los nombres de los argumentos de la macro. Consiste de dos partes, la primera conserva el nombre del argumento y la segunda, contiene un número consecutivo para el argumento.

4. Tabla de cuerpo de macro.

Contiene cada una de las instrucciones que forman a la macroinstrucción, los argumentos en cada instrucción son sustituidos por números asignados en la tabla de argumentos de macros.

5. Programa expandido.

Es el programa fuente pero eliminadas las declaraciones de macroinstrucción y sustituidas las llamadas a macro por sus instrucciones correspondientes.

6. Tabla de símbolos.

Es usada para almacenar cada etiqueta y su valor correspondiente.

7. Tabla de literales.

Es usada para almacenar cada literal encontrada y su correspondiente localización.

8. Tabla de códigos de la instrucción.

Contiene información que identifica el código de operación de la instrucción, la sintaxis válida, la longitud de la instrucción y el código de máquina correspondiente.

9. Tablas de validación de instrucción.

Es usada para validar los modos de direccionamiento.

10. Tabla de dirección efectiva.

Es usada para obtener la dirección efectiva de cada uno de los modos de direccionamiento.

CAPITULO V
PROGRAMACION

V PROGRAMACION

El siguiente capítulo contiene lo que se denomina pseudocódigo, que son los procedimientos que sigue el programa paso a paso, escritos en lenguaje estructurado incluyen la referencia numérica que guardan con los diagramas de las etapas anteriores de análisis y diseño, posteriormente se encuentran los programas escritos en lenguaje Pascal.

***MACROENSAMBLADOR (0.0)**

- MENU:**
1. Macroprocesador
 2. Ensamblador
 3. Salir del programa

Si la opción es un 1 haz

Pasa al bloque de macroprocesador

Si la opción es un 2 haz

Pasa al bloque de ensamblador

Fin

Fin

MACROPROCESADOR (1.6)

- MENU**
1. Recopilar macrodefiniciones
 2. Expandir macrolineas
 3. Regresar al menú anterior

Si la opción es 1 ejecuta

Ve al bloque de identificar macrodefinición y ejecuta.

Ve al bloque de identificar cuerpo macro y ejecuta

Regresa al menú anterior

Si no pregunta si es un 3 y ejecuta

Ve al bloque identificar llamada y ejecuta

Ve al bloque expande cuerpo y ejecuta

Pregunta si quiere ud. ensamblar

Ve al menú de ensamblador

Regresa al menú anterior

Si no regresa al menú de macroensamblador.

IDENTIFICAR MACRODEFINICION (3.1.1)

```
Mientras haya instrucciones ejecuta
  Lee instrucción
  Busca primer carácter diferente a blanco
  Si el primer carácter es punto
    Lee siguiente carácter hasta que sea diferente de blanco
    Si cadena restante de la instrucción es igual a MACRO
      Guarda el nombre en la tabla de MACRO
      Guarda número de argumentos y argumentos
      Llama a Identificar cuerpo MACRO
    Fin
  Fin
Fin
```

IDENTIFICAR CUERPO DE MACRO (3.1.2)

```
Mientras haya instrucciones e no encuentres MEND
  Lee instrucción
  Busca primer carácter diferente de blanco
  Si el primer carácter es diferente de punto
    Si hay argumentos
      Reemplaza argumentos por índices
    Fin
    Guarda instrucción en tabla de cuerpos de MACRO
  Si no
    Busca primer carácter diferente a blanco
    Si cadena restante es igual a MEND
      Índice que encontraste MEND
    Fin
  Fin
Fin
```

IDENTIFICAR LLAMADAS (1.2.1)

```
Mientras haya instrucción ejecuta
  Lee instrucción
  Separa por campos y toma código de operación
  Mientras haya elementos en la tabla de MACROS ejecuta
    Si código de operación es igual a nombre de MACRO
      Marca encontrado
      Llame a expandir cuerpo
    Si no
      Lee siguiente elemento de tabla de MACROS
  Fin
Si no lo encontré
  Graba la instrucción en el archivo expandido
Fin
```

EXPANDIR CUERPO (1.2.2)

```
Mientras haya instrucciones de la MACRO ejecuta
  Lee instrucción
  Sustituye argumentos de llamada
  Graba instrucción en código expandido
Fin
```

ENSAMBLADOR (2.0)

MENU: 1. Recolector
 2. Generador
 3. Regresa al menú anterior(principal)

Si la opción es un 1 y ejecuta

 Ve al bloque identificar símbolos y ejecuta

 Ve al bloque identificar literales y ejecuta

 Regresa al menú anterior

Si no pregunta si es un 2 y ejecuta

 Ve al bloque válida instrucción y ejecuta

 Ve al bloque arma directiva y ejecuta

 Ve al bloque arma instrucción y ejecuta

 Muestra el programa ya traducido a código máquina

Regresa al menú principal.

IDENTIFICAR SÍMBOLOS Y LITERALES (2.1.1) y (2.1.2)

```
Mientras haya instrucciones en el código expandido
  Leer instrucción
  Separa por campos y obtén código de operación
  Mientras haya mnemónicos de instrucción a comparar ejecuta
    Lee mnemónico
    Si código de operación es igual a mnemónico
      Indica que se trata de una instrucción
      No busques más en tabla de mnemónicos
    Fin
  Fin
Si no es una instrucción entonces
  Mientras haya directivas a comparar haz
    Lee directiva
    Si código de operación es igual a directiva
      Indica que se trata de una directiva
      No busques más en tabla de directivas
    Fin
  Fin
Fin
Si no es instrucción ni directiva
  Guarda código de operación encontrado en tabla de Símb.
Fin
Fin
```

VALIDA INSTRUCCION (2.2.1)

```
Mientras haya instrucciones o errores menor al máximo
  Leer instrucción
  Separa por campos y obtien el código de operación
  Llama a ARMAR DIRECTIVAS
  Si no es directiva
    Llama a ARMAR INSTRUCCIONES
  Fin
Fin
```

ARMA DIRECTIVA (2.2.2)

```
Mientras haya directivas en la tabla de directivas haz
  Lee directiva
  Si el código de operación es igual a la directiva
    Valida sintaxis de la directiva
    Si hay error
      Guarda instrucción junto con su error en la salida
      Incrementa número de errores
    Si no
      Ejecuta directiva
      Incrementa contador de localidades
  Fin
Fin
Fin
```

ANALISA INSTRUCCION (2.1.2)

```
Mientras haya anemónicos de instrucción ejecuta
  Lee anemónico de instrucción
  Si el código de operación es igual al anemónico
    Valida sintaxis de la instrucción
    Si hay error
      Guarda instrucción junto con su error en salida
      Incrementa número de errores
    Si no
      Genera código de la instrucción
      Incrementa contador de localidades
      Indica encontrado
  Fin
Fin
```

PROGRAMA ENSAMBLE

El siguiente listado contiene la parte del sistema que ensambla las instrucciones para el microprocesador y que contiene los siguientes procedimientos :

- a) Elimina exceso de blancos y compacta la instrucción.
- b) Separa por palabras la instrucción actual analizada.
- c) Guarda palabras que forman la instrucción.
- d) ~~Creación~~ ^{Creación} del diagrama de estados para los modos de direccionamiento.
- e) ~~Crea las~~ ^{Crea las} tablas de dirección efectiva.
- f) Crea la dirección efectiva dependiendo del modo de direccionamiento y del registro usado.
- g) Se mueve a lo largo del autómata para identificar el modo de direccionamiento del operando.
- h) Verifica la sintaxis de la instrucción, generando las posibles combinaciones de modos de direccionamiento.
- i) Verifica que los modos de direccionamiento usados en la instrucción sean válidos.
- j) Ensambla la instrucción a analizar.
- k) Sustituye en el código el número de registro de datos usado.
- l) Sustituye el registro de datos como destino.
- m) Sustituye en el código el número de registro de dirección usado.
- n) Sustituye en el código el número de registro de dirección usado como destino.
- ñ) Sustituye el código de la dirección efectiva del operando fuente.
- o) Sustituye el código de la dirección efectiva del operando destino.
- p) Sustituye el código binario de la instrucción por código hexadecimal.

PROGRAM ENSEMBLE (INPUT, OUTPUT, FUENTE, CODIGO);

CONST

LONGBIN= 16;
LONGWOOD= 34;
LEDOS= 33;
LONGOOD= 55;
LCARAC= 15;
LONGON= 12;
LININST= 55;
LINCTD= 7;
LONGSEP= 3;

TYPE

WORDS= PACKED ARRAY [1..LINCTD, 1..LONGON] OF CHAR;
WORD= PACKED ARRAY [1..LONGON] OF CHAR;
INSTRUC= PACKED ARRAY [1..LININST] OF CHAR;
DIRECCION= PACKED ARRAY [1..4] OF CHAR;
ADIRECCION= PACKED ARRAY [1..2, 1..6] OF CHAR;
TMOO= PACKED ARRAY [1..12,1..3] OF CHAR;
TREGISTER= PACKED ARRAY [1..13,1..3] OF CHAR;
FORMATO= PACKED ARRAY [1..2, 1..LONGWOOD] OF CHAR;
TSINARIO= PACKED ARRAY [1..2, 1..LONGBIN] OF CHAR;
TCAQUINA= PACKED ARRAY [1..LONGBIN] OF CHAR;
TINMEDIATO= PACKED ARRAY [1..4,1..4] OF CHAR;
TCAQUIE= PACKED ARRAY [1..4] OF CHAR;

VAR

FUENTE,
CODIGO, TEXT;
LINEAIN, LINEAOUT: INSTRUC;
I, J, K, L, LONG: INTEGER;
NUMREG, NUMFILAS: INTEGER;
ENCR: INTEGER;
NODO, NUMWORDS: INTEGER;
NUMINARIO: INTEGER;
PALABRAS: WORDS;
PALABRA: WORD;
ADICIONAL: PACKED ARRAY [1..LINCTD, 1..LONGSEP] OF CHAR;
TABAUTON: ARRAY [1..LEDOS, 1..LCARAC] OF INTEGER;
TABCARAC: ARRAY [1..LCARAC] OF CHAR;
DIRREFEC: DIRECCION;
ADIRREFEC: ADIRECCION;
TABNODO: TMOO;
TABREG: TREGISTER;
POSIBILIDADES: FORMATO;
NODO: ARRAY [1..2] OF INTEGER;
ABINARIO: TSINARIO;
CAQUINA: TCAQUINA;
CINMEDIATO: TINMEDIATO;
CAQUIE: TCAQUIE;
TIPOINSTRUC: CHAR;
APTEXT: INTEGER;

```
.....  
{* ..... *}  
{* ELIMINA EXCESO DE BLANCOS ..... *}  
{* Y COMPACTA LA INSTRUCCION ..... *}  
{* ..... *}
```

```

{*****}
PROCEDURE QUITARBLANCOS (VAR LONG: INTEGER;VAR LINEA:INSTRUC;
                        VAR LINEA2:-INSTRUC );
CONST
  BLANCO= ' ';
VAR
  SEPARADOR: SET OF CHAR;
  I,J,E,L,N: INTEGER;
  CARACTER: CHAR;
BEGIN
  SEPARADOR:= [ ]; SEPARADOR:=SEPARADOR + ['#'];
  SEPARADOR:= SEPARADOR + ['(']; SEPARADOR:= SEPARADOR + [')'];
  SEPARADOR:= SEPARADOR + [',']; SEPARADOR:= SEPARADOR + [';'];
  SEPARADOR:= SEPARADOR + [' ']; SEPARADOR:= SEPARADOR + ['\'];
  SEPARADOR:= SEPARADOR + ['\'];
  J:=0; E:=0; L:=0; N:=0;
  FOR I:=1 TO LONG DO
    BEGIN
      CARACTER:= LINEA[I];
      IF CARACTER IN SEPARADOR THEN
        N:=1;
      IF CARACTER <> BLANCO THEN
        BEGIN
          J:= J+1;
          LINEA2[J]:=LINEA[I];
          E:=0; L:=1;
        END
      ELSE
        BEGIN
          IF (E = 0) AND (L = 1) AND (N = 0) THEN
            BEGIN
              J:=J+1;
              LINEA2[J]:=LINEA[I];
              E:=1;
            END;
          END;
        END;
      LONG:=J;
    END;(*QUITARBLANCOS*)
  {*****}
  {
  *
  * SEPARA POR PALABRAS LA INSTRUCCION ACTUAL ANALIZADA
  *
  *
  {*****}
PROCEDURE FORWORDS( VAR PALABRAS: WORDS; LINEA: INSTRUC;
                   VAR NUMWORDS: INTEGER);
CONST
  BLANCO= ' ';
  PUNTOCOMA= ',';
VAR
  CARACTER: CHAR;
  I,J,E,W,MM: INTEGER;
  SEPARADOR: SET OF CHAR;
(*
(*
(* GUARDA PALABRAS QUE FORMAN LA INSTRUCCION
(*
(*

```

```

PROCEDURE GUARDAMODOS;
BEGIN
  E:= E+1;
  J:=0; WW:=0;
  REPEAT
    J:= J+1;
    PALABRAS[E,J]:= LINEA[I];
    I:= I+1
  UNTIL ( LINEA[I] IN SEPARADOR ) OR ( I > LONG );
  J:= J-1; (* PARA APUNTAR AL ULTIMO CARACTER VISTO *)
END;
(*
BEGIN
  SEPARADOR:=[]; SEPARADOR:= SEPARADOR + [' '];
  SEPARADOR:=SEPARADOR + [',']; SEPARADOR:= SEPARADOR + [':'];
  (*SEPARA POR PALABRAS*)
  FOR E:=1 TO LIMCTO DO
    BEGIN
      FOR I:=1 TO LONGSEP DO
        ADICIONAL[E,I]:= BLANCO;
      FOR I:=1 TO LONGCOM DO
        PALABRAS[E,I]:= BLANCO;
      END;
      E:=0; J:=0; WW:=0;
      WHILE (E <= LIMCTO) AND ( I < LONG ) DO
        BEGIN
          I:= I+1;
          CARACTER:= LINEA[I];
          IF CARACTER IN SEPARADOR THEN
            BEGIN
              WW:= WW+1;
              ADICIONAL[E,WW]:= CARACTER;
            END
          ELSE
            GUARDAMODOS;
          END;
          NUMMODOS:=E;
        END;
      (*NONWORDS*)
    )
  .....
  (*
  * CREACION DEL DIAGRAMA DE ESTADOS PARA LOS MODOS DE
  * DE DIRECCIONAMIENTO. LOS MODOS CONTEMPLADOS SON :
  *
  *
  *      1. DIRECTO A REGISTRO DE DATOS.
  *      2. DIRECTO A REGISTRO DE DIRECCION.
  *      3. INDIRECTO A REGISTRO DE DIRECCION.
  *      4. INDIRECTO A REGISTRO DE DIRECCION CON +
  *      5. INDIRECTO A REGISTRO DE DIRECCION CON -
  *      6. INDIRECTO CON DESPLAZAMIENTO.
  *      7. INDIRECTO CON EL PC Y DESPLAZAMIENTO
  *      12. INMEDIATO.
  *      9. ABSOLUTO CORTO.
  *      10. ABSOLUTO LARGO.
  *
  * .....
  *)
  PROCEDURE AUTOMATA;

```



```

PROCEDURE CREAEPFC;
BEGIN
TARMODO[1,1]:=0;TARMODO[1,2]:=0;TARMODO[1,3]:=0;
TARMODO[2,1]:=0;TARMODO[2,2]:=0;TARMODO[2,3]:=1;
TARMODO[3,1]:=0;TARMODO[3,2]:=1;TARMODO[3,3]:=0;
TARMODO[4,1]:=0;TARMODO[4,2]:=1;TARMODO[4,3]:=1;
TARMODO[5,1]:=1;TARMODO[5,2]:=0;TARMODO[5,3]:=0;
TARMODO[6,1]:=1;TARMODO[6,2]:=0;TARMODO[6,3]:=1;
TARMODO[7,1]:=1;TARMODO[7,2]:=1;TARMODO[7,3]:=0;
TARMODO[8,1]:=1;TARMODO[8,2]:=1;TARMODO[8,3]:=1;
TARMODO[9,1]:=1;TARMODO[9,2]:=1;TARMODO[9,3]:=1;
TARMODO[10,1]:=1;TARMODO[10,2]:=1;TARMODO[10,3]:=1;
TARMODO[11,1]:=1;TARMODO[11,2]:=1;TARMODO[11,3]:=1;
TARMODO[12,1]:=1;TARMODO[12,2]:=1;TARMODO[12,3]:=1;
TARREG[1,2]:=0;TARREG[1,3]:=0;TARREG[1,3]:=0;
TARREG[2,2]:=0;TARREG[2,3]:=0;TARREG[2,3]:=1;
TARREG[3,2]:=0;TARREG[3,3]:=1;TARREG[3,3]:=0;
TARREG[4,2]:=0;TARREG[4,3]:=1;TARREG[4,3]:=1;
TARREG[5,2]:=1;TARREG[5,3]:=0;TARREG[5,3]:=0;
TARREG[6,2]:=1;TARREG[6,3]:=0;TARREG[6,3]:=1;
TARREG[7,2]:=1;TARREG[7,3]:=1;TARREG[7,3]:=0;
TARREG[8,2]:=1;TARREG[8,3]:=1;TARREG[8,3]:=1;
TARREG[9,2]:=0;TARREG[9,3]:=0;TARREG[9,3]:=0;
TARREG[10,1]:=0;TARREG[10,2]:=0;TARREG[10,3]:=1;
TARREG[11,1]:=0;TARREG[11,3]:=1;TARREG[11,3]:=1;
TARREG[12,1]:=0;TARREG[12,3]:=1;TARREG[12,3]:=1;
TARREG[13,1]:=1;TARREG[13,2]:=0;TARREG[13,3]:=0;
END; (* CREAEPFC *)
*
*.....*
*
*
* CREA LA DIRECCION EFECTIVA DEPENDIENDO DEL MODO
* DE DIRECCIONAMIENTO Y DEL REGISTRO USADO.
*
*.....*
PROCEDURE EFECTIVA(OPERANDO:WORD; VAR MODO: INTEGER;
VAR DIREFEC: DIRECCION; NUMREG: INTEGER);
VAR
FLAG: BOOLEAN;
I: INTEGER;
BEGIN
FLAG:= FALSE;
IF NUMREG = 99 THEN (* MODO DE DIR NO USA REGISTROS *)
FLAG:= TRUE;
IF FLAG THEN
BEGIN
FOR I:=1 TO 3 DO
BEGIN
DIREFEC[I]:= TARMODO[MODO,I];
DIREFEC[1-3]:= TARREG[MODO-1,I];
END;
END;
END
ELSE
BEGIN

```

```

FOR I:=1 TO 3 DO
  BEGIN
    DIREFPC[I]:= TABREG[MODO,I];
    DIREFPC[I+3]:= TABREG[NUMREG+1,I];
  .END;
.END;
END; (* EFECTIVA *)
(*
*****
*)
(* SE MUEVE A LO LARGO DEL AUTOMATA PARA IDENTIFICAR
*)
(* EL MODO DE DIRECCIONAMIENTO DEL OPERANDO.
*)
*)
*****
*)
PROCEDURE VIAJA(OPERANDO: WORD; VAR MODO,NUMREG: INTEGER;
TIPO:CHAR;VAR APUNTEO: INTEGER);
CONST
  EDERROR= 35;
VAR
  I,INDICE,J,II,MM,JJ: INTEGER;
  AX: INTEGER;
  EDO,VEDO,SEEDO: INTEGER;
  COMEDO: INTEGER;
  DATO: CHAR;
  ERRO: BOOLEAN;
BEGIN
  I:=0;VEDO:=1; ERROR:= FALSE;
  COMEDO:=0;MODO:=0;NUMREG:=33;
  REPEAT
    I:=I+1; INDICE:= 0;
    EDO:= VEDO; COMEDO:=COMEDO+1;
    II:=0;
    DATO:= OPERANDO[I];
    FOR J:=1 TO LCHAR DO
      BEGIN
        IF TABCHAR[J] = DATO THEN
          INDICE:= J;
      END;
    IF INDICE = 0 THEN (* SE TRATA DE UN CARACTER NUMERICO *)
      BEGIN
        IF ( ORD(DATO) >= 48 ) AND ( ORD(DATO) <= 54) THEN
          BEGIN
            INDICE:= 3;
            NUMREG:= ORD(DATO)-48;
          END
        ELSE
          IF ( ORD(DATO) = 35 ) THEN
            BEGIN
              INDICE:= 4;
              NUMREG:= ORD(DATO)-48;
            END
          ELSE
            INDICE:= 11;
          END
        END;
      END;

```

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

```
VEDO:= TABAYTON(ESO,INDICE);
IF COMTEDO = 1 THEN
  SEGUIDO:= VEDO;
IF VEDO = EDCORROR THEN
  ERROR:= TRUE;
CASE VEDO OF
  4: MODO:= 1 ;
  7: MODO:= 2 ;
  12: CASE SEGUIDO OF
    8: MODO:= 3;
    15: MODO:= 5;
    16: MODO:= 4;
  ELSE
    ERROR:= TRUE;
  END;
14: MODO:= 4;
19: BEGIN
  IF TIPO = '2' THEN (* INTROCCION LARGA *)
    BEGIN
      II:=LONGCON; AUX:=APUNTEXT+2; NW:=5;
      REPEAT
        IF OPERANDO[II] <> ' ' THEN
          BEGIN
            NW:=NW-1;
            C[IMEDIATO[AUX,NW]]:= OPERANDO[II];
          END;
          II:= II-1;
        UNTIL (NW = 1) OR (II = 1);
        IF II > 1 THEN
          BEGIN
            AUX:=APUNTEXT+1; NW:=5;
            REPEAT
              IF OPERANDO[II] <> ' ' THEN
                BEGIN
                  NW:=NW-1;
                  C[IMEDIATO[AUX,NW]]:= OPERANDO[II];
                END;
                II:= II-1;
              UNTIL (NW = 1) OR (II = 1);
            END;
            APUNTEXT:= APUNTEXT+2;
          END;
        IF TIPO = '1' THEN
          BEGIN
            II:=LONGCON;
            APUNTEXT:=APUNTEXT+1; NW:=5;
            REPEAT
              IF OPERANDO[II] <> ' ' THEN
                BEGIN
                  NW:=NW-1;
                  C[IMEDIATO[APUNTEXT,NW]]:= OPERANDO[II];
                END;
                II:= II-1;
              UNTIL (NW = 1) OR (II = 1);
            END;
          END;
        END;
```


VAR ABINARIO: TERNARIO; VAR TIPO: CHAR;

VAR

H, I, W, MW: INTEGER;

LINEAIN: INSTRUCC;

BEGIN

ASSIGN(CODIGO, 'INSTRUCC.DAT');

RESET(CODIGO);

M:=0;

WHILE (NOT EOF(CODIGO)) DO

 BEGIN

 I:=0; H:=0; MW:=0;

 REPEAT

 I:= I+1;

 READ(CODIGO, LINEAIN[I]);

 UNTIL I=LONGCOD;

 FOR I:=1 TO LONGMOM DO

 BEGIN

 IF LINEAIN[I] = INSTRUCCION[I] THEN

 H:= H+1;

 END;

 IF H = LONGMOM THEN

 BEGIN

 M:=M+1;

 FOR J:=11 TO LONGMODO DO

 BEGIN

 MW:= MW+1;

 POSIBILIDAD(W, MW) := LINEAIN[J];

 END;

 MW:=0; TIPO:= LINEAIN[38];

 FOR J:=48 TO 53 DO

 BEGIN

 MW:=MW+1;

 ABINARIO(W, MW) := LINEAIN[J];

 END;

 END;

 READLN(CODIGO);

 END;

 CLOSE(CODIGO);

 IF M = 0 THEN (* NO EXISTE LA INSTRUCCIÓN *)

 ERROR:= 1;

 MUNPEL:= W;

 END; (* CERRARCODIGO *)

.....
(* VERIFICA QUE LOS MODOS DE DIRECCIONAMIENTO USADOS EN LA INSTWCC. *)
(* SEAN VALIDOS. *)
(* *)
.....

PROCEDURE VERMODO(VAR MUNPEL: INTEGER);

VAR

I, J, K, H: INTEGER;

AANDOS: ARRAY[1..12, 1..3] OF INTEGER;

TIPO: PACKED ARRAY[1..2] OF CHAR;

HASOPERANDOS: BOOLEAN;

ERROR_VALIDO: BOOLEAN;

COLUMNA: INTEGER;


```

        VAR APOINTER: INTEGER};

VAR
    POINTERCMAQ:    INTEGER;
    CARACTER:       CHAR;
    I,J:            INTEGER;
{*
*****
** SUBSTITUTE EN EL CODIGO EL NUMERO DE REGISTRO DE DATOS USADO **
**
*****
}
PROCEDURE DATAREGISTER;
BEGIN
    IF ANODO[I] = 1 THEN
        BEGIN
            POINTERCMAQ:= POINTERCMAQ+1;
            CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,4];
            POINTERCMAQ:= POINTERCMAQ+1;
            CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,5];
            POINTERCMAQ:=POINTERCMAQ+1;
            CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,6];
        END
    ELSE
        BEGIN
            POINTERCMAQ:= POINTERCMAQ+1;
            CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,4];
            POINTERCMAQ:= POINTERCMAQ+1;
            CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,5];
            POINTERCMAQ:=POINTERCMAQ+1;
            CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,6];
        END;
END; (* DATA REGISTER *)
{*
*****
** SUBSTITUTE EL DATA REGISTER COMO DESTINO **
**
*****
}
PROCEDURE DESTDATA;
BEGIN
    POINTERCMAQ:= POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,4];
    POINTERCMAQ:= POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,5];
    POINTERCMAQ:=POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,6];
END; (* DESTDATA *)

{*
*****
** SUBSTITUTE EN EL CODIGO EL NUMERO DE REGISTRO DE DIRECCION **
** USADO. **
*****
}
PROCEDURE ADDRESSREGISTER;
BEGIN
    POINTERCMAQ:= POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,4];
    POINTERCMAQ:= POINTERCMAQ+1;

```

```

CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,5];
POINTERCMAQ:=POINTERCMAQ+1;
CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,6];
END; (* ADDRESS REGISTER *)
{
*****
}
{** SUBSTITUTE EN EL CODIGO EL NUMERO DE REGISTRO DE DIRECCION **}
{** USADO COMO DESTINO. **}
{*****}
PROCEDURE DESTADDRESS;
BEGIN
    POINTERCMAQ:= POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,4];
    POINTERCMAQ:= POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,5];
    POINTERCMAQ:=POINTERCMAQ+1;
    CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,6];
END; (* DESTADDRESS *)
{
*****
}
{** SUBSTITUTE EN EL CODIGO EL EFECTIVE ADDRESS DEL OPERANDO **}
{** **}
{*****}
PROCEDURE EFECTIVEADDRESS;
VAR
    I: INTEGER;
BEGIN
    IF ARCCO[1] = 1 THEN
        BEGIN
            FOR I:=1 TO 6 DO
                BEGIN
                    POINTERCMAQ:= POINTERCMAQ+1;
                    CRAQUINA[POINTERCMAQ]:= ADIREFEC[2,I];
                END
            END
        ELSE
            BEGIN
                FOR I:=1 TO 6 DO
                    BEGIN
                        POINTERCMAQ:= POINTERCMAQ+1;
                        CRAQUINA[POINTERCMAQ]:= ADIREFEC[1,I];
                    END;
                END;
            END;
END; (* EFECTIVEADDRESS *)
{
*****
}
{** SUBSTITUTE EN EL CODIGO EL EFECTIVE ADDRESS DEL OPERANDO **}
{** FUENTE. **}
{*****}
PROCEDURE EFECTIVESOURCE;
VAR
    I: INTEGER;
BEGIN
    FOR I:=1 TO 6 DO
        BEGIN

```

```

        POINTERCMAQ:= POINTERCMAQ+1;
        CMAQUINA[POINTERCMAQ]:= ADIREFEC[1,I];
    END; (* EFECTIVE *)
{ *
{*****
{** SUSTITUYE EN EL CODIGO EL EFECTIVE-ADDRESS DEL OPERANDO **}
{** DESTINO. **}
{*****
PROCEDURE EFECTIVEDESTINO;
VAR
    I: INTEGER;
BEGIN
    FOR I:=1 TO 6 DO
        BEGIN
            POINTERCMAQ:= POINTERCMAQ+1;
            CMAQUINA[POINTERCMAQ]:= ADIREFEC[2,I];
        END;
    END; (* EFECTIVEDESTINO *)
{ *
{*****
{** SUSTITUYE EN EL CODIGO EL EFECTIVE ADDRESS DEL OPERANDO **}
{** DESTINO EN FORMA INVERTIDA. **}
{*****
PROCEDURE DESTINOEFECTIVE;
VAR
    I: INTEGER;
BEGIN
    FOR I:=4 TO 6 DO
        BEGIN
            POINTERCMAQ:= POINTERCMAQ+1;
            CMAQUINA[POINTERCMAQ]:= ADIREFEC[2,I];
        END;
    FOR I:=1 TO 3 DO
        BEGIN
            POINTERCMAQ:= POINTERCMAQ+1;
            CMAQUINA[POINTERCMAQ]:= ADIREFEC[2,I];
        END;
    END; (* DESTINOEFECTIVE *)
{ *
{*****
{** SUSTITUYE EN EL CODIGO EL CORREMIENTO. **}
{** **}
{*****
PROCEDURE SHIFTCOUNT( VAR APUNTER: INTEGER);
VAR
    I: INTEGER;
    CARACTER: CHAR;
BEGIN
    CARACTER:= CIMEDIATO[1,4];
    CASE CARACTER OF
        '1': BEGIN
            POINTERCMAQ:= POINTERCMAQ+1;
            CMAQUINA[POINTERCMAQ]:= '0';
            POINTERCMAQ:= POINTERCMAQ+1;

```

```

        CRAQUNA[POINTERCRAQ]:= '0';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
    END;
'2': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
    END;
'3': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
    END;
'4': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
    END;
'5': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
    END;
'6': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';
    END;
'7': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '1';
    END;
'8': BEGIN
        POINTERCRAQ:= POINTERCRAQ+1;
        CRAQUNA[POINTERCRAQ]:= '0';

```

```

        POINTERCMAQ:= POINTERCMAQ+1;
        CRAQUINA[POINTERCMAQ]:= '0';
        POINTERCMAQ:= POINTERCMAQ+1;
        CRAQUINA[POINTERCMAQ]:= '0';
    END;
END; (* CASE *)
APUNTA:=0;
END; (* SHIFTCOUNT*)
(*
BEGIN
    I:=1;
    POINTERCMAQ:=0;
    WHILE ( I <= LONGBIN ) DO
    BEGIN
        CARACTER:= ABINARIO[NUMBINARIO,I];
        CASE CARACTER OF

            'A': BEGIN
                    ADDRESSREGISTER;
                    I:=I+3;
                END;

            'C': BEGIN
                    DATAREGISTER;
                    I:=I+3;
                END;

            'F': BEGIN
                    EFFECTIVEADDRESS;
                    I:=I+6;
                END;

            'J': BEGIN
                    EFFECTIVESOURCE;
                    I:=I+6;
                END;

            'L': BEGIN
                    EFFECTIVEDESTINO;
                    I:=I+6;
                END;

            'S': BEGIN
                    DESTDATA;
                    I:=I+3;
                END;

            'P': BEGIN
                    DESTADDRESS;
                    I:=I+3;
                END;

            'Q': BEGIN
                    DESTINDIRECTIVE;
                    I:=I+6;
                END;

            'S': BEGIN
                    SHIFTCOUNT(APUNTA);
                    I:=I+3;
                END;

        ELSE;

```

*)

```

      BEGIN
        POINTER:= POINTER+1;
        CRACK:=CRACK+CHARACTER;
        I:=I+1;
      END;
    END; (* CASE *)
  END; (* WHILE *)
END; (* ASSEMBLA *)
*)
*****
** SUSTITUIR EL CODIGO BINARIO DE LA INSTRUCCION **
** POR SU DISEÑO EN HEXADECIMAL. **
*****
PROCEDURE HEXADECIMAL; CRACK:= CRACK+CRACK; CIRCULADO:= CIRCULADO;
APUNTA:= INICIO; VIA CRACK:= CRACK;
VAR
  I,I1,I2,POINTER: INTEGER;
  SUMA,I1,I2,HEX: INTEGER;
BEGIN
  POINTER:= 1;
  WHILE ( POINTER <= 4 ) DO
    BEGIN
      I:=POINTER * 4;
      SUMA:=0;
      I1:= I;
      REPEAT
        UNTIL I=1;
        I1:= ORD(CRACK(I)); I:= I-1;
        IF I1 > 0 THEN
          BEGIN
            FOR J:=1 TO I1 DO
              SUMA:= SUMA+I1;
            END;
            I1:=I1-1;
            I2:=I1;
            UNTIL ( I1 = 0 );
          CASE SUMA OF
            0: BEGIN
                  CRACK(I+POINTER):= '0';
                  POINTER:= POINTER+1;
                END;
            1: BEGIN
                  CRACK(I+POINTER):= '1';
                  POINTER:= POINTER+1;
                END;
            2: BEGIN
                  CRACK(I+POINTER):= '2';
                  POINTER:= POINTER+1;
                END;
            3: BEGIN
                  CRACK(I+POINTER):= '3';
                  POINTER:= POINTER+1;
                END;
            4: BEGIN

```

```

        CMAQHEK[ POINTER ] := '4';
        POINTER := POINTER + 1;
    END;
5: BEGIN
        CMAQHEK[ POINTER ] := '5';
        POINTER := POINTER + 1;
    END;
6: BEGIN
        CMAQHEK[ POINTER ] := '6';
        POINTER := POINTER + 1;
    END;
7: BEGIN
        CMAQHEK[ POINTER ] := '7';
        POINTER := POINTER + 1;
    END;
8: BEGIN
        CMAQHEK[ POINTER ] := '8';
        POINTER := POINTER + 1;
    END;
9: BEGIN
        CMAQHEK[ POINTER ] := '9';
        POINTER := POINTER + 1;
    END;
10: BEGIN
        CMAQHEK[ POINTER ] := 'A';
        POINTER := POINTER + 1;
    END;
11: BEGIN
        CMAQHEK[ POINTER ] := 'B';
        POINTER := POINTER + 1;
    END;
12: BEGIN
        CMAQHEK[ POINTER ] := 'C';
        POINTER := POINTER + 1;
    END;
13: BEGIN
        CMAQHEK[ POINTER ] := 'D';
        POINTER := POINTER + 1;
    END;
14: BEGIN
        CMAQHEK[ POINTER ] := 'E';
        POINTER := POINTER + 1;
    END;
15: BEGIN
        CMAQHEK[ POINTER ] := 'F';
        POINTER := POINTER + 1;
    END;
END; (* CASE *)
END; (* WHILE *)
FOR I := 1 TO 4 DO
    WRITE( CMAQHEK[ I ] );
    FOR J := 1 TO APOINTEX DO
        FOR K := 1 TO 2 DO
            WRITE( CINKKIDATO[ I, J ] );
        END;
    END;
END;

```

END; {+ HEXADECIMAL+}

```
(* *)
BEGIN
ASSIGN (FUENTE, 'FUENTE.DAT');
RESET(FUENTE);
AUTOMATE;
CREATEPEC;
WHILE NOT EOF(FUENTE) DO
BEGIN
I:=0;
REPEAT
I:= I+1;
READ(FUENTE, LINEAIN[1]);
UNTIL (LINEAIN[1] = ';' ) OR ( I = LINENET);
LONG:= I;
QUITABLANCOS(LONG, LINEAIN, LINEACOT);
FORWORDS(FALABRAS, LINEACOT, MINWORDS);
FOR J:=1 TO LONGROW DO
FALABRA[J]:= FALABRAS[1,J];
OSTERNCODIGO(FALABRA, POSIBILIDADES, ERROR, NUMPLE, ABINARIO, TIPOINSTR)

FOR I:=1 TO 4 DO
FOR J:=1 TO 4 DO
CIMEDIATO[1,J]:='0';
I:=2; AMODO[1]:=99; AMODO[2]:=99; APUNTEXT:=0;
WHILE I <= MINWORDS DO
BEGIN
FOR J:=1 TO LONGROW DO
FALABRA[J]:= FALABRAS[1,J];
VIAJA(FALABRA, MODO, NUMREG, TIPOINSTRUC, APUNTEXT);
AMODO[I-1]:=MODO;
EFFECTIVA(FALABRA, MODO, DIREFEC, NUMREG);
FOR J:=1 TO 6 DO
ADIREFEC[I-1,J]:= DIREFEC[J];
I:=I+1;
END;
VERMODOC(NUMBINARIO);
ENSAMBLA(NUMBINARIO, CRAQUINA, APUNTEXT);
HEXADECIMAL(CRAQUINA, CIMEDIATO, APUNTEXT, CRAQUES);
READLN(FUENTE);
END;
CLOSE(FUENTE);
END.
```

PROGRAMA MACRODISEMBLE

El siguiente listado contiene la parte del sistema que procesa MACROS o instrucciones para el procesador y que contiene los siguientes procedimientos principales:

- a | Elimina exceso de blancos y compacta la instrucción.
- b | Separa por palabras la instrucción actual analizada.
- c | Guarda el argumento de los macros.
- d | Identifica la declaración de MACRO y guarda su nombre y el cuerpo de esta.
- e | Reemplaza en el cuerpo de la macro los argumentos por índices.
- f | Procedimiento para recolectar nombres de MACROS y argumentos.
- g | Procedimiento para expandir el cuerpo de la MACRO.


```

CARACTER:= LEXEM(I);
IF CARACTER = SEPARADOR THEN
  M:=I;
IF CARACTER = FINDEL THEN
  BEGIN
    J:= J+1;
    LEXEM(J):=LEXEM(I);
    K:=0; L:=1;
  END
ELSE
  BEGIN
    IF (K = 0) AND (L = 1) AND (M = 0) THEN
      BEGIN
        J:=J+1;
        LEXEM(J):=LEXEM(I);
        K:=1;
      END;
    END;
    LEXEM:=S;
  END;
END;(*QUITAVALLADOS*)
.....
(*
(* SEPARA POR PALABRAS LA INSTRUCCION ACTUAL ANALIZADA
*)
*)
.....
PROCEDIMIENTO PARMONIS; VAR PALABRAS: WORDS; LEXEM: INSTRUCC;
VAR NIMONTO: INTEGER;
CONST
  SEPARA= ' ';
  FINDEL= '>';
VAR
  CARACTER: CHAR;
  I,J,K,M,NW: INTEGER;
  SEPARADOR: SET OF CHAR;
(*
(* GUARDA PALABRAS QUE FORMAN LA INSTRUCCION
*)
*)
C
PROCEDIMIENTO QUITAVALLADOS;
BEGIN
  K:= M+1; J:=0; NW:=0;
  BEGIN
    J:= J+1;
    PALABRAS[K,J]:= LEXEM[I];
    I:= I+1;
  UNTIL ( LEXEM[I] IN SEPARADOR ) OR ( I > LONG );
  I:= I-1; (* PARA APUNTA AL ULTIMO CARACTER VISTO *)
END;
C
(*
*)
BEGIN
  SEPARADOR:=[]; SEPARADOR:= SEPARADOR + [' '];
  SEPARADOR:=SEPARADOR + ['\']; SEPARADOR:= SEPARADOR + ['\'];
  (*SEPARA POR PALABRAS*)
  FOR M:=1 TO LIMITE DO
    BEGIN

```

```

FOR I:=1 TO LONGEXP DO
  ADICIONAL[K,I]:= BLANCO;
FOR J:=1 TO LONGNOM DO
  PALABRAS[K,J]:= BLANCO;
END;
K:=0 ; I:=0; NW:=0;
WHILE (K <= LONGEXP) AND ( I < LONG ) DO
  BEGIN
    I:= I+1;
    CARACTER:= LINEA[I];
    IF CARACTER IS SEPARADOR THEN
      BEGIN
        NW:= NW+1;
        ADICIONAL[K,NW]:= CARACTER;
      END
    ELSE
      GUARDESWORD;
  END;
NUMWORDS:=K;
END; {FORMWORDS}
{.....}
{*
{* GUARDA EL ARGUMENTO DE LAS MACROS, RELACIONANDOSLO
{* CON SU INDICE.
{*
{*
{*.....}
PROCEDERE ARGUMENTOS( PALABRAS: WORDS; NUMWORDS: INTEGER);
CONST
  AMPER= ' & ' ;
  BLANCO= ' ' ;
VAR
  BLANCO: PACKED ARRAY [1..LONGNOM] OF CHAR;
  I,K,LL: INTEGER;
  HAYARG: BOOLEAN;
BEGIN
  FOR I:=1 TO LONGNOM DO
    BLANCO[I]:=BLANCO;
  HAYARG:= TRUE;
  J:=1; { POSICION FIJA DESDE DONDE COMIENZAN LOS ARGUMENTOS }
  I:=0;
  LL:= 48; {= inicio del caso " }
  WHILE ( HAYARG ) AND ( J <= NUMWORDS ) DO
    BEGIN
      IF PALABRAS[J,I] = AMPER THEN
        BEGIN
          K:=K+1;
          FOR I:=1 TO LONGNOM DO
            ANCHO[K,I]:= PALABRAS[J,I];
          ANCHO[K,13]:= AMPER;
          LL:=LL+1;
          ANCHO[K,14]:= CHR(LL);
        END
      ELSE
        BEGIN
          WRITELN(' SÍMBOLO NO ASOCIADO A NINGUNA VARIABLE ');
        END
    END
  END;

```

```

        HAYARG:= FALSE;
    END;
    J:=J+1;
END; {WHILE}
APUNTAORIG:= K;
END; {ARGUMENTOS}

```

```

{.....}
{ *
{ * IDENTIFICA A LA DECLARACION DE MACRO Y GUARDA SU
{ * NOMBRE EN LA TABLA DE NOMBRES DE MACRO, INDICANDO
{ * QUE LAS SIGUIENTES LINEAS SERAN EL CUERPO DE LA MACRO *
{ *
{ *
{.....}

```

```

PROCEDURE MACRO (LONG: INTEGER; LINEA: INSTRUCC);

```

```

CONST

```

```

    N = 'N';
    A = 'A';
    C = 'C';
    R = 'R';
    O = 'O';
    BLANCO = ' ';
    SEIS = 6;

```

```

TYPE

```

```

    ALFA = 'A'..'Z';

```

```

VAR

```

```

    I,J,K: INTEGER;
    NUMORDS: INTEGER;
    PALABRAS: WORDS;
    CADENA: PACKED ARRAY[1..SEIS] OF CHAR;
    ALFABETO: SET OF ALFA;
    CARAC: ALFA;

```

```

BEGIN

```

```

    ALFABETO:= [ ];

```

```

    FOR CARAC:='A' TO 'Z' DO

```

```

        ALFABETO:=ALFABETO + [CARAC];

```

```

    FORWARD ( PALABRAS, LINEA, NUMORDS);

```

```

    CADENA[1]:= N; CADENA[2]:=A; CADENA[3]:=C;

```

```

    CADENA[4]:=R; CADENA[5]:=O; CADENA[6]:= BLANCO;

```

```

    J:=0;

```

```

    FOR I:=1 TO SEIS DO

```

```

        BEGIN

```

```

            IF PALABRAS[I,1] = CADENA[I] THEN

```

```

                J:= J+1;

```

```

            END;

```

```

        IF J = SEIS THEN (ENCONTRAMOS LA PALABRA MACRO *)

```

```

            BEGIN

```

```

                APUNTAORIG:= APUNTAORIG *I;

```

```

                APUNTAINDICE:= APUNTAINDICE +1;

```

```

                INDICADOR[APUNTAINDICE,1]:= APUNTAORIG +1;

```

```

IF PALABRAS[2,1] IN ALFABETO THEN
  FLAG
  NFORM:= 3;
  FOR I:= 1 TO LONGMACO DO
    BEGIN
      WORDSINDEX [APERTON,1]:= PALABRAS[2,1];
      MATRACRO:= TRUE;
    END;
  ARGUMENTOS{PALABRAS, MINWORDS};
  END;
  END; (* MACRO *)
{*****}
(*
(* REEMPLAZA EN EL CUERPO DE LA MACRO LOS ARGUMENTOS
(* POR INDICES.
(*
(*
{*****}
PROCEDURE REEMPLAZA( VAR LINEA: INSTRUCC; VAR LONG: INTEGER;
  ERROLESTA: BANDERA);
CONST
  BLANCO= ' ';
VAR
  PALABRAS: WORDS;
  MINWORDS: INTEGER;
  I,J,M: INTEGER;
  YAVI BLANCO: BANDERA;
{*****}
(*
(* COMPARA PALABRAS PARA INDICAR EL ARGUMENTO A SUSTITUIR.
(*
(*
{*****}
PROCEDURE COMPARAMOS;
VAR
  I,N,J,MN,W: INTEGER;
  ENCONTRADO: BANDERA;
BEGIN
  I:=0;
  WHILE ( I = SUFICIES ) TO
  BEGIN
    I:= I+1;N:=0; ENCONTRADO:= FALSE;
    REPEAT
      N:=N+1; W:=0;
      FOR J:=1 TO LONGMACO DO
        BEGIN
          IF PALABRAS[I,J] = ANSWER[N,J] THEN
            M:=M+1;
          END;
        IF M = LONGMACO THEN
          BEGIN
            FOR MN:=1 TO LONGMACO DO
              PALABRAS[I,MN]:= BLANCO;
              PALABRAS[2,1]:= ANSWER[N,1];
              PALABRAS[2,2]:= ANSWER[N,2];
              ENCONTRADO:=TRUE;
            END;

```

```

UNTIL ( ENCONTRADO = TRUE ) OR ( H = APUNTADECR )
END; (* WHILE *)
END; (* COMPARACIONES *)
.....
(*
* COMPARA PALABRAS PARA INDICAR EL AUMENTO A SUSTITUIR.
* PARA LA EXPANSION DE LA MACRO.
*)
.....
PROCEDURE EXPWORDS;
VAR
  I,H,J,NW,W: INTEGER;
  ENCONTRADO: BANDERA;
BEGIN
  I:=0;
  WHILE ( I < NUNWORDS ) DO
    BEGIN
      I:= I+1;H:=0; ENCONTRADO:= FALSE;
      REPEAT
        H:=H+1; W:=0;
        IF PALABRAS[I,1] = ARGCR[H,13] THEN
          W:=W+1;
        IF PALABRAS[I,1] = ARGCR[H,14] THEN
          W:= W+1;
        IF W = 2 THEN
          BEGIN
            FOR NW:=-1 TO LONGROW DO
              PALABRAS[I,NW]:= BLANCO;
            FOR NW:=-1 TO LONGROW DO
              PALABRAS[I,NW]:= ARGCR[H,NW];
            ENCONTRADO:=TRUE;
          END
        UNTIL ( ENCONTRADO = TRUE ) OR ( H = APUNTADECR )
      END; (* WHILE *)
    END; (* EXPWORDS *) ( *
    (****)
    ( *
    BEGIN
    FORMWORDS(PALABRAS,LINER,NUNWORDS);
    IF RECOLECTA THEN (* RECOLECTANDO LA DEFINICION DE MACRO *)
      COMPARACIONES
    ELSE
      EXPWORDS; (* EXPANDIENDO CUERPO DE MACRO *)
    H:=0;
    FOR I:=1 TO NUNWORDS DO
      BEGIN
        YAVINBLANCO:=FALSE;
        FOR J:=1 TO LONGROW DO
          BEGIN
            IF ( PALABRAS[I,J] <> BLANCO ) THEN
              BEGIN
                H:=H+1;
                LINER[H]:= PALABRAS[I,J];
              END
            END

```

```

ELSE
  BEGIN
    IF YAVIBLANCO = FALSE THEN
      BEGIN
        H:=H+1;
        LINEA[H]:=BLANCO;
        YAVIBLANCO:=TRUE;
      END;
    END;
  END;
FOR J:=1 TO LONGSEP DO
  BEGIN
    IF ADICIONAL[I,J] <> BLANCO THEN
      BEGIN
        H:=H+1;
        LINEA[H]:=ADICIONAL[I,J];
      END;
    END;
  END; (* FOR *)
LONG:=-H;
END; (* REEMPLAZA *)

```

```

.....
*
* ALMACENA EL CUERPO DE LAS MACROS Y ACTUALIZA LOS
* INDICES EN LA TABLA DE INDICES DE LAS MACROS.
*
*
.....
PROCEDURE CUERPO ( VAR LONG: INTEGER; LINEA: INSTRUCC );
CONST
  BLANCO= ' ';
  CINCO= 5;
VAR
  I,J,K,L: INTEGER;
  NUNWORDS: INTEGER;
  CARACTER: CHAR;
  PALABRAS: WORDS;
  RECOLECTA: BANDERA;
  CADENA: PACKED ARRAY[1..CINCO] OF CHAR;
BEGIN
  FORWORDS ( PALABRAS,LINEA,NUNWORDS);
  CADENA[1]:= 'M'; CADENA[2]:= 'E'; CADENA[3]:= 'N';
  CADENA[4]:= 'D'; CADENA[5]:= ' ';
  J:=0;
  FOR I:=1 TO CINCO DO
    BEGIN
      IF CADENA[I] = PALABRAS[1,I] THEN
        J:= J+1;
      END;
    IF J = CINCO THEN (* HAY NEMB*)

```

```

BEGIN
  HAYMACRO:= FALSE;
  IMCICENCR[APUNTCUERPO,2]:= APUNTCUERPO;
END
ELSE
  (*GUARDA CUERPO*)
  BEGIN
    APUNTCUERPO:= APUNTCUERPO +1;
    RECOLECTA:= TRUE;
    REEMPLAZA(LINEA, LONG, RECOLECTA);
    FOR I:=1 TO LONG DO
      CUERPOCR[APUNTCUERPO, I]:= LINEA[I];
    FOR I:=LONG+1 TO LIMNST DO
      CUERPOCR[APUNTCUERPO, I]:= BLANCO;
    END;
  END; (* CUERPO*)
*)
*) PROCEDIMIENTO PARA RECOLECTAR NOMBRES DE MACROS Y *)
*) ARGUMENTOS. *)
*) *)
*) ENTRADA : PROGRAMA FUENTE *)
*) *)
*) SALIDA : TABLA DE NOMBRES DE MACROS *)
*) TABLA DE INDICES DE MACROS *)
*) TABLA DE CUERPOS *)
*) *)
PROCEDURE RECOLECTA;
CONST
  BLANCO= ' ';
VAR
  EMP,EMCPA: PACKED ARRAY [1..5] OF CHAR;
BEGIN
  APUNTYMCR:= 0; APUNTCUERPO:=0; APUNTCICENCR:=0; HAYMACRO:= FALSE;
  ASSIGN (FUENTE, 'FUENTE.DAT');
  REWRITE (INTERMEDIO, 'INTERMEDIO.DAT');
  RESET (FUENTE);
  CONTINST:=0;
  WHILE (NOT EOF (FUENTE)) AND (EMCR = 0) DO
    BEGIN
      I:=0; CONTINST:=CONTINST + 1;
      REPEAT
        I:= I+1;
        READ (FUENTE, LINEAIN[I]);
      UNTIL (LINEAIN[I] = ' ') OR ( I = LIMNST);
      IF I = LIMNST THEN
        EMCR:= 1
      ELSE
        BEGIN
          LONG:= I;
          QUITABLANCOS (LONG, LINEAIN, LINEAOUT);
          FOR J:=1 TO 5 DO
            EMCPA[J]:= LINEAOUT[J];
          IF HAYMACRO THEN
            CUERPO (LONG, LINEAOUT);
          ELSE

```

```

BEGIN
  SACRO (LONG, LINEADU);
  IF MAYMACRO = FALSE THEN
    BEGIN
      FOR I:=1 TO LONG DO
        WRITE (INTERMEDIO, LINEADU(I));
      WRITELN (INTERMEDIO);
    END;
  END;
  READLN (FUENTE);
END;
[* WHILE *]
IF ERROR = 0 THEN
  BEGIN
    ENDA[1]:='E'; ENDA[2]:='N'; ENDA[3]:='D'; ENDA[4]:=' ';
    J:=0;
    FOR I:=1 TO 5 DO
      BEGIN
        IF (ENDA[I] = ENDDA[I]) OR (ENDA[I] = BLANCO) THEN
          J:=J+1;
        END;
        IF J = 4 THEN
          ERROR:=2;
        END;
      END;
    CLOSE (INTERMEDIO);
    CLOSE (FUENTE);
  END;
  [*
  * PROCEDIMIENTO PARA EXPANDIR EL CUERPO DE LA SACRO *]
  [*
  * ENTRADA : PROGRAMA FUENTE SIN DECLARACION *]
  [* DE MACROS. *]
  [*
  * SALIDA : PROGRAMA FUENTE COS LAS LLAMADAS A *]
  [* MACROS EXPANDIDAS. *]
  [*
  * *]
  [*
  * *]
  PROCEDURE LLAMAMACR ( PALABRAS: WORDS; VAR MAYLLAMADA: BANDERA;
    NUMWORDS: INTEGER);
  CONST
    PUNTOYCIMA='.';
    BLANCO=' ';
    APER='&';
  VAR
    I, J, L, LUGAR, K, LL: INTEGER;
    ESCONTRADO: BANDERA;
    RECOLECTA: BANDERA;
    DIFERENTE: BANDERA;
    LINEA: INSTRU;
  BEGIN
    MAYLLAMADA:= FALSE;
    ESCONTRADO:= FALSE;
    I:=0;
    WHILE ( I = APUNTON) AND ( ESCONTRADO = FALSE ) DO

```

```

BEGIN
  DIFERENTE:= FALSE;
  I:= 1;
  J:=0;L:=0;
  REPEAT
    J:= J+1;
    IF PALABRAS[I,J] = HOKKRESMCR[I,J] THEN
      L:= L+1
    ELSE
      BEGIN
        IF PALABRAS[I,J] = PUNTOYCOMA THEN
          L:= L+1
        ELSE
          DIFERENTE:= TRUE
        END
      END
    UNTIL [ DIFERENTE = TRUE ] OR [ J = LONGHOR ];
    IF L = LONGHOR THEN
      BEGIN
        ENCONTRADO:= TRUE;
        HAYLLAKADA:= TRUE;
        LOGAR:= I;
      END;
    END; (* WHILE *)
  IF ENCONTRADO THEN
    BEGIN
      LL:= 48; AFUNTARGMCR:=0;
      FOR I:=2 TO NUNWORDS DO
        BEGIN
          FOR J:=1 TO LONGHOR DO
            ARGMCR[I-1,J]:= PALABRAS[I,J];
            LL:= LL+1;
            AFUNTARGMCR:= AFUNTARGMCR+1;
            ARGMCR[AFUNTARGMCR,1]:= ARPER;
            ARGMCR[AFUNTARGMCR,14]:= CER(LL);
          END;
          J:= INDICMCR[LOGAR,1];
          K:= INDICMCR[LOGAR,2];
          FOR I:=J TO K DO
            BEGIN
              LL:=0;
              FOR L:=1 TO LIMIST DO
                BEGIN
                  LL:=LL+1;
                  LINEA[LL]:= CUERPMCR[I,L];
                  IF CUERPMCR[I,L] = PUNTOYCOMA THEN
                    LONG:=LL;
                END;
              RECOLECTA:= FALSE;
              REEMPLAZA(LINEA,LL,RECOLECTA);
              FOR L:=1 TO LL DO
                WRITE(EXPANDIDO,LINEA[L]);
              WRITELN(EXPANDIDO);
            END;(* IF *)
          END;
        END;
      END;
    END;
  
```

```

END; (* LLAMADA MACRO *)
(*
PROCEDURE EXPANDE;
VAR
  NUMWORDS: INTEGER;
BEGIN
  HAYLLAMADA:= FALSE;
  ASSIGN (INTERMEDIO,'INTERMEDIO.DAT');
  ASSIGN (EXPANDIDO,'EXPANDIDO.DAT');
  REWRITE(EXPANDIDO);
  RESET(INTERMEDIO);
  WHILE NOT EOF(INTERMEDIO) DO
    BEGIN
      I:=0;
      REPEAT
        I:= I+1;
        READ(INTERMEDIO,LINEAIN[I]);
      UNTIL LINEAIN[I] = ';';
      LONG:= I;
      FORMWORDS(PALABRAS, LINEAIN, NUMWORDS);
      LLAMADAOR(PALABRAS, HAYLLAMADA, NUMWORDS);
      IF HAYLLAMADA = FALSE THEN
        BEGIN
          FOR I:=1 TO LONG DO
            WRITE(EXPANDIDO,LINEAIN[I]);
          WRITELN(EXPANDIDO);
        END;
      READLN(INTERMEDIO);
    END;
  CLOSE(INTERMEDIO);
  CLOSE(EXPANDIDO);
END;
(*
PROGRAMA PRINCIPAL MACRO
(*
(*
BEGIN
FOR I:=1 TO 10 DO;
WRITELN;
WRITELN('***** PRINCIPIA PROGRAMA MACRO *****');
ERROR:= 0;
RECOLECTA;
IF ERROR <> 0 THEN
  BEGIN
    CASE ERROR OF
      1: WRITELN('ERR' INSTRUCCION ',CONTINIST,' FALTA ');
      2: WRITELN('ERR' PROGRAMA SIN END FINAL ');
      3: WRITELN('ERR' FALTA NOMBRE DE MACRO EN ',CONTINIST');
    END;
  END
ELSE
  EXPANDE;
WRITELN('***** TERMINO PROGRAMA MACRO *****');
END.

```

ARCHIVO DE INSTRUCCIONES

El siguiente archivo contiene las instrucciones válidas para el microprocesador MC16002 junto con una tabla que sirve para validar el modo de direccionamiento del operando, por ejemplo si el operando es fuente tiene que encontrar un "uno" en la intersección de la columna con el renglón de la instrucción, la posición en las columnas están relacionadas directamente con los modos de direccionamiento. La columna 11 contiene un uno si la instrucción es de tamaño palabra y un dos si es de tamaño palabra larga. Las columnas de la 33 en adelante son los forsetos de cada instrucción, que una vez procesados son divididos de cuatro en cuatro generando un número hexadecimal.

ADD_B	130302030202020203000000	1	1101CC0100FFFGG
ADD_B	0210101010101010101010	1	1101CC0000FFFGG
ADD_M	120202020202020202000000	1	1101CC0101FFFGG
ADD_M	0210101010101010101010	1	1101CC0001FFFGG
ADD_L	130302020202020203000000	2	1101CC0110FFFGG
ADD_L	0210101010101010101010	2	1101CC0000FFFGG
ADD_A_W	101200101010101010101010	1	1101AA0111JJJJK
ADD_A_L	101200101010101010101010	2	1101AA1111JJJJK
ADDI_B	020002020202020202000010	1	0000011000LLMMN
ADDI_B	020002020202020202000010	1	0000011001LLMMN
ADDI_L	020002020202020202000010	2	0000011010LLMMN
ADDI_B	120000000000000000000000	1	1101MM100000CCC
ADDI_B	000012000000000000000000	1	1101AA100001FFF
ADDI_W	120000000000000000000000	1	1101MM101000CCC
ADDI_W	000012000000000000000000	1	1101AA101001FFF
ADDI_L	120000000000000000000000	2	1101MM100000CCC
ADDI_L	000012000000000000000000	2	1101AA110001FFF
CLR_B	100010101010101010000000	1	010001000JJJJKK
CLR_M	100010101010101010000000	1	010001001JJJJKK
CLR_L	100010101010101010000000	2	010001010JJJJKK
CHP_B	121010101010101010101010	1	1011MM000JJJJKK
CHP_M	121010101010101010101010	1	1011MM001JJJJKK
CHP_L	121010101010101010101010	2	1011MM010JJJJKK
CHPA_W	101210101010101010101010	1	1011MM011JJJJKK
CHPA_L	101210101010101010101010	2	1011MM111JJJJKK
CMPI_B	020002020202020202000010	1	000011000LLMMN
CMPI_B	020002020202020202000010	1	000011001LLMMN
CMPI_L	020002020202020202000010	2	000011010LLMMN
CMPL_M	000000120000000000000000	1	1011AA100001FFF
CMPL_M	000000120000000000000000	1	1011AA101001FFF
CMPL_L	000000120000000000000000	2	1011AA110001FFF
DEVS_W	120010101010101010101010	1	1000MM111JJJJKK
DEVS_W	120010101010101010101010	1	1000MM011JJJJKK
EXT_M	100000000000000000000000	1	010010001000CCC
EXT_L	100000000000000000000000	2	0100100011000CC
HLS_W	120010101010101010101010	1	1100MM011JJJJKK
HLS_W	120010101010101010101010	1	1100MM010JJJJKK
HLS_B	100010101010101010000000	1	010001000JJJJKK
HLS_M	100010101010101010000000	1	010001001JJJJKK
HLS_L	100010101010101010000000	2	010001010JJJJKK
HSL_B	100010101010101010000000	1	010000000JJJJKK
HSL_M	100010101010101010000000	1	010000001JJJJKK
HSL_L	100010101010101010000000	2	010000010JJJJKK
SUB_B	131010101010101010101010	1	1001MM000JJJJKK
SUB_B	130302020202020202000000	1	1001CC0100LLMMN
SUB_M	131010101010101010101010	1	1001MM001JJJJKK
SUB_M	130202020202020202000000	1	1001CC0101LLMMN
SUB_L	131010101010101010101010	2	1001MM010JJJJKK
SUB_L	130302020202020202000000	2	1001CC0110LLMMN
SUBA_M	101210101010101010101010	1	1001AA0111JJJJK
SUBA_L	101210101010101010101010	2	1001AA1111JJJJK
SUBI_B	020002020202020202000010	1	0000011000LLMMN
SUBI_M	020002020202020202000010	1	0000011001LLMMN
SUBI_L	020002020202020202000010	2	0000011010LLMMN
SUBI_B	120000000000000000000000	1	1001MM100000CCC

SURE.B	000012020000000000000000	1	1001AAAL00001FFF
SURE.H	120000000000000000000000	1	1001AHE010000CCC
SURE.W	000012000000000000000000	1	1001AARL01001FFF
SURE.L	120000000000000000000000	2	1001AHML10000CCC
SURE.I	000012000000000000000000	2	1001AARL10001FFF
AND.D	120010101010101010101010	1	1100HN0000JJJKEK
AND.B	120202020202020202020000	1	1100CCC1001LLMH
AND.W	120010101010101010101010	1	1100HN001JJJKEK
AND.H	120202020202020202020000	1	1100CCC1011LLMH
AND.L	120000101010101010101010	2	1100HN001JJJKEK
AND.I	120202020202020202020000	2	1100CCC1111LLMH
AND.E	020002020202020202020000	1	00000010001LLMH
AND.M	020002020202020202020000	1	00000010011LLMH
AND.L	020002020202020202020000	2	00000010101LLMH
OR.B	120002020202020202020000	1	1011CCC1001LLMH
OR.H	120002020202020202020000	1	1011CCC1011LLMH
OR.L	120002020202020202020000	2	1011CCC1101LLMH
ORT.D	020002020202020202020000	1	00001010001LLMH
ORT.W	020002020202020202020000	1	00001010011LLMH
ORT.L	020002020202020202020000	2	00001010101LLMH
OT.D	100010101010101010000000	1	0100011000JJJKEK
OT.W	100010101010101010000000	1	0100011001JJJKEK
OT.L	100010101010101010000000	2	0100011010JJJKEK
OR.B	120010101010101010101010	1	1000HN0000JJJKEK
OR.B	120202020202020202020000	1	1000CCC1001LLMH
OR.W	120000101010101010101010	1	1000HN001JJJKEK
OR.W	120202020202020202020000	1	1000CCC1011LLMH
OR.L	120010101010101010101010	2	1000HN001JJJKEK
OR.L	120202020202020202020000	2	1000CCC1101LLMH
ORI.B	020002020202020202020000	1	00000010001LLMH
ORI.W	020002020202020202020000	1	00000010011LLMH
ORI.L	020002020202020202020000	2	00000010101LLMH
TST.B	100010101010101010000000	1	0100101000JJJKEK
TST.W	100010101010101010000000	1	0100101001JJJKEK
TST.L	100010101010101010000000	2	0100101010JJJKEK
EXG.L	120000000000000000000000	2	1100CCC101000HH
EXG.L	001200000000000000000000	2	1100FFF10100AAA
LEA.L	000210000000000000000000	2	0100AAA111JJJKEK
NOVE.B	121012121212121212101010	1	0001000000JJJKEK
NOVE.H	121012121212121212101010	1	0011000000JJJKEK
NOVE.L	121012121212121212101010	2	0011000000JJJKEK
NOVE.W	101210101010101010101010	1	0011AAA001JJJKEK
NOVE.L	101210101010101010101010	2	0011AAA001JJJKEK
NOVE.W	100000000002000000000000	1	000000010000AAA
NOVE.W	020000000100000000000000	1	000000011000AAA
NOVE.L	100000000002000000000000	2	000000010000AAA
NOVE.L	020000000100000000000000	2	000000011000AAA
NOVE.L	020000000000000000000000	2	0111MH0000000000
PSA.L	000010000010101010101010	2	0100100001JJJKEK
ASL.B	120000000000000000000000	1	1110000100100HH
ASL.B	020000000000000000000000	1	111000100000HH
ASL.W	120000000000000000000000	1	111000101100HH
ASL.W	020000000000000000000000	1	111000101100HH
ASL.H	000010101010101010000000	1	111000111JJJKEK
ASL.L	120000000000000000000000	2	111000111100HH

CAPÍTULO VI
DEMOSTRACION DE RESULTADOS

VI DEMOSTRACION DE RESULTADOS

Este capítulo contiene la prueba del sistema mostrando algunos ejemplos de programas en lenguaje ensamblador y su traducción a lenguaje máquina así como un tema relacionado con el manejo del sistema y sus anotaciones, como emplear la sintaxis y los requerimientos para obtener del sistema traductor los resultados satisfactorios.

Anotaciones para traducir programas con el sistema.

a) Todas las instrucciones deberán llevar una extensión de la siguiente forma : .B si la instrucción es de tamaño byte .W si la instrucción es de tamaño palabra y .L si es de tamaño palabra larga. ejemplo :

```
MOVE.L (A0), D0
```

b) En el caso de que las instrucciones se refieran a movimientos entre registros de direcciones o instrucciones inmediatas deberán estar bien definidas, llevando la letra en la instrucción como el siguiente ejemplo :

```
CMPI.B #0, D0 (en esta instrucción la I)
```

```
MOVEA.L A0,A1 (en esta instrucción la A)
```

c) Cuando una instrucción incluya números decimales deberán ser convertidos previamente a números hexadecimales. ejemplo :

si la instrucción es MOVE.L #65601, A1

deberá escribirse como MOVE.L #10041, A1

Manejo del sistema traductor

Una vez que se han hecho los cambios necesarios para que las instrucciones sean compatibles con las definidas en el sistema, se procederá a escribirlos en el archivo de datos llamado FUENTE.DAT, tenemos dos opciones para correr el traductor, una que nuestro programa incluya definiciones de MACROS, para lo cual se deberá correr el programa MACROENSAMBLE.EXE primero, posteriormente se deberá correr el programa ENSAMBLE.EXE. La segunda opción es que el programa a traducir no contenga definiciones de MACROS, con lo que bastará correr solamente el programa ENSAMBLE.EXE.

```

MACRO (MC)
MOVE.B    #0, (A0);
ADDL.W   #1, A0;
MOVE.B   #1, (A0);
ADDL.W   #1, A0;
MOVE.W   #10, D7;
MOVE.W   #10, D0;
RND ;
ADDL.W   #1, A0;
MOVEA.L  #1000, A0;
MOVE.W   #1000, D0;
MOVEA.L  #1000, A0;
USD;
COP1.B   #0, D0;
REQ.S    500000E;
ADDL.L   #1, A0;
REQ.S    5000010;
MOVE.W   #E, D7;
TRAP     #E;
USD ;
LEA.L    500001500, A0;
MOVE.W   #FD, D7;
TRAP     #E;
NOP;
NOP;
MOVE.W   #D, D7;
TRAP     #E;
MOVE.W   #E, D7;
TRAP     #E;
NOP;
END;

```

PROGRAMA CON DECLARACION DE MACRO Y SU
CORRESPONDIENTE EXPANSION.

```

ADDA.W    #1, A6;
MOVEB.L  #1004, A3;
MOVE.W   #1500, D0;
MOVEA.L  #1000, A0;
MOVEA.B  #10, (A6);
ADDA.W   #1, A6;
MOVE.B   #2, (A6);
ADDA.W   #1, A6;
MOVE.W   #FF, D7;
MOVE.W   #1B, D0;
CMPL.B   #0, D0;
BEQ.S    $00001E;
ADDA.L   #1, A0;
BRA.S    $000010;
MOVE.W   #24, D7;
TRAP     #2;
MOVEA.B  #10, (A6);
ADDA.W   #1, A6;
MOVE.B   #2, (A6);
ADDA.W   #1, A6;
MOVE.W   #FF, D7;
MOVE.W   #1B, D0;
LEA.L    $00001500, A0;
MOVE.W   #7D, D7;
TRAP     #2;
NOP;
NOP;
MOVE.W   #0, D7;
TRAP     #2;
MOVE.W   #24, D7;
TRAP     #2;
NOP;
END;

```

EXPANSION.

EJEMPLO 1

```
MOVE.L #10000, A0;
MOVE.W #0100, D6;
MOVE.P.W D6, $0000(A0);
MOVE.L $0004, A0;
ABCD.B D4, D0;
CMPI.B #60, D0;
BLTQ.B $000000;
CLR.W D0;
ABCD.B D4, D1;
CMPI.B #60, D1;
BLT.B $000000;
CLR.W D1;
MOVE.W D2, D5;
LSL.W #4, D5;
ROR.B #4, D5;
ORI.W #0010, D5;
MOVE.W D5, (A0);
MOVE.W D1, D9;
LSL.W #4, D9;
ROR.B #4, D9;
ORI.W #0010, D9;
ROR.W #8, D9;
MOVE.B D9, 7(A0);
MOVE.W D0, D8;
LSL.W #4, D8;
ROR.W #4, D8;
ORI.W #0010, D8;
MOVE.L #2500, A5;
MOVE.W #00F3, D7;
TRAP;
STR;
NOP;
END;
```

ASSEMBLY, ICE

000000	207000010015
000006	10700100
00000A	00000000
00000E	20700104
000012	0104
000014	00000000
000018	0010
00001A	4240
00001C	0304
00001E	00010000
000022	0010
000024	4241
000026	0504
000028	00000010
00002C	0000
00002E	4242
000030	1A02
000032	E940
000034	E010
000036	00450000
00003A	1005
00003C	1A01
00003E	E940
000040	E010
000042	00451000
000044	11450004
00004A	E050
00004C	11450003
000050	1A00
000052	E940
000054	E010
000056	00450000
00005A	11450000
00005E	2A702500
000062	3E100073
000066	4E4E
000068	4E77
00006A	4E71

12

EJEMPLO 2

```
MOVE.L #100, A5;
MOVE.W #256, D0;
MOVE.W #7, D7;
TRAP;
MOVE.B #10, (A6);
ADDR.W #1, A6;
MOVE.B #5, (A6);
ADDR.W #1, A6;
MOVE.W #78, D7;
MOVE.W #18, D0;
TRAP;
MOVE.W #42, D0;
TRAP;
MOVE.W #1, D1;
MOVE.W #2000, SR;
NOP;
NOP;
TST.W D1;
BNE.S #00001A;
NOP;
MOVE.W #700, SR;
MOVE.W #4, D7;
TRAP;
NOP;
MOVE.W #73, D7;
TRAP;
MOVE.W #0, D1;
NOP;
END;
```

A>DISASSEMBLE.KEX

000000	2A7C00001000
000004	7C7C00001000
000008	3E3C00F1
000010	4E4E
000012	1C8C000A
000014	0CFC0001
00001A	1C8C0010
00001E	0CFC0001
000022	3E3C00F8
000026	303C0018
00002A	4E4E
00002C	303C002A
000030	4E4E
000032	313C0001
000036	44FC2000
00003A	4E71
00003E	4E71
000042	4A41
000046	66F8
00004C	4E71
00004E	44FC2700
000048	3E3C00E4
00004C	4E4E
00004E	4E71
000050	3E3C00F3
000054	4E4E
000056	323C0000
00005A	4E71

A>

EJEMPLO 3

```

MOVEA.W #2000, A5;
MOVEA.L A5, A6;
MOVEA.L A5, $2504;
MOVE.L #0, $2508;
MOVEA.L #1, D4;
MOVE.W #F1, D7;
TRAP #E;
MOVE.W #TIB, (A6)+;
MOVE.W #2000, (A6)+;
MOVE.L #100020A2, -(A5);
SUBI.L #4, (A5);
MOVE.W #101A, -(A5);
SUBA.L #2, A5;
MOVE.L A5, $2500;
MOVEA.L $2504, A0;
MOVE.W (A0), D2;
ANDI.W #0F0F, D2;
LSL.B #4, D2;
ROR.W #4, D2;
MOVE.B 1(A0), D1;
LSL.W #8, D1;
MOVE.B 4(A0), D1;
ANDI.W #0F0F, D1;
LSL.B #4, D1;
ROR.W #4, D1;
MOVE.W 6(A0), D0;
ANDI.W #0F0F, D0;
LSL.B #4, D0;
ROR.W #4, D0;
MOVEA.L #10015, A0;
MOVE.L #10018, D3;
MOVEP.L D3, $0000(A0);
MOVEQ.W #-A140, D3;
MOVEA.L #10011, A0;
MOVEQ.W #-A140, D3;
MOVEP.W D3, $0000(A0);
MOVE.W #2000, SR;
BRA.B $00098A;
MOVE.W #84, D7;
TRAP #E;
NOP;

```

A->DSSAMPLE.EXE

000000	3A7C3008
000004	204D
000006	21CD2504
00000A	21FC000000002508
000012	280000000001
000018	3E0000F1
00001C	404E
00001E	3CFC0718
000022	3CFC2000
000026	280C18102842
00002C	0c0000000004
000032	703C182A
000036	56FC00000002
00003C	21CD2500
000040	207E1508
000046	3410
00004E	02420F0F
000054	E90A
00005C	E85A
000062	12200003
00006A	E149
000072	12200004
00007A	02410F0F
000082	E909
00008A	E859
000092	30200006
00009A	02400F0F
0000A2	E908
0000AA	E858
0000B2	207C00010025
0000BA	263C0001E848
0000C2	07C00000
0000CA	207C00010021
0000D2	363CA140
0000DA	07000000
0000E2	46FC2000
0000EA	60FA
0000F2	3E3C0004
0000FA	4E2E
000000	4E71
000008	4E71

A>

EJEMPLO 4

```
LEA.L    $0500, A0;
MOVE.W  #FD, D7;
TRAP    #E;
MOVE.L  A0, $0504;
MOVE.W  #DA, D7;
TRAP    #E;
NOP;
NOP;
MOVE.W  #1, D7;
TRAP    #E;
MOVE.W  #EA, D7;
TRAP    #E;
NOP;
MOVEA.L #1000, A0;
MOVEA.L A0, A1;
ADDA.L  #4, A1;
MOVE.W  {A0}, D2;
CMP.L   {A1}, D0;
BLE.S   $000000;
MOVE.L  -4(A1), (A0);
MOVE.L  D0, -4(A1);
CMPA.L  $2000, A1;
BNE.S   $00005E;
ADDA.L  #4, A0;
MOVEA.L A0, A1;
CMPA.L  $2000, A0;
BNE.S   $00005E;
RFS;
NOP;
END;
```

A>EXAMPLE.EXE

000000	41FD0500
000004	3E3C00FD
000008	4E4E
00000A	21C00504
00000E	3A3C0003
000012	4E4E
000014	4E71
000016	4E71
000018	3E3C0001
00001C	4E4E
00001E	3E3C0004
000022	4E4E
000024	4E71
000028	203C00001000
00002C	3240
000030	01FC00000004
000034	2030
000038	0000
00003C	5F00
000040	20A0FFFC
000044	3340FFFC
000048	03F02000
00004C	6600
000050	01FC00000004
000054	3240
000058	01F02000
00005C	6600
000060	4E75
000064	4E71

A>

EXAMPLE 5

```

MOVE.L #10001, A0;
MOVE.W #0, D0;
MOVEP.W D0, $0000(A0);
MOVE.L #10000, A0;
MOVE.W 0-1, D0;
MOVEP.W D0, $0000(A0);
MOVEP.L #10000, A0;
MOVE.W #7F00, D0;
MOVEP.W D0, $0000(A0);
MOVE.L #10000, A0;
MOVE.W #2020, D0;
MOVEP.W D0, $0000(A0);
MOVE.L #10001, A0;
MOVE.W #700, D0;
MOVEP.W D0, $0000(A0);
MOVE.L #10011, A0;
MOVE.L #1000, A1;
MOVE.L #3, D4;
MOVEP.W D0, $0000(A0);
JRR.S
NOP;
NOP;
DELT.L D4, $00000070;
JMP.S
NOP;
NOP;
NOP;
NOP;
NOP;
MOVE.L #1, D3;
MOVE.L #15, D0;
NOP;
DELT.L D3, $000004;
DELT.L D3, $00007E;
RTS;
NOP;
NOP;
END;

```

A> ENSEMBLE. CEE

000000	207000010001
000006	30300000
00000A	01000000
00000E	207000010005
000014	3030FFFF
000018	01000000
00001C	207000010009
000020	3030F700
000026	01000000
00002A	20700001000D
000030	30302020
000034	01000000
000038	207000010001
00003E	30300700
000042	01000000
000046	207000010001
00004C	227000010000
000052	203000000003
000058	01000000
00005C	4E000970
000060	4E71
000062	4E71
000064	53CCFF70
000068	4E70094C
00006C	4E71
00006D	4E71
000070	4E71
000072	4E71
000074	4E71
000076	343C00000000

A>

CONCLUSIONES

CONCLUSIONES

En el desarrollo del traductor se consideraron varias etapas, que comprenden desde como definir el marco teórico hasta la puesta en marcha de puntos técnicos en la programación. El marco teórico se logró a través de diferente bibliografía y la asesoría recibida por parte del director de tesis, enfocando la información hacia la arquitectura del microprocesador y conceptos de lenguajes ensambladores, análisis y diseño de sistemas.

En el diseño del traductor se siguieron dos procesos, el primero manejando la información de entrada como procesador de textos y el segundo usado como reconocimiento y conversión de textos a su correspondiente valor de máquina. Para ambos procesos se definieron algoritmos que tocan su base en autómatas finitos y se generó una tabla con entradas para cada instrucción del microprocesador, que almacena el código objeto, modos de direccionamiento válidos y la longitud de cada instrucción.

Los programas fuente que el sistema traduce tuvieron que ser anotados a una sintaxis para poderlos interpretar, estos se formatearon para las macroinstrucciones, para las instrucciones del procesador y las directivas.

El programa objeto generado se escribe en un archivo, que puede posteriormente conectarse con un cargador para transferirlo directamente hacia el computador educacional.

BIBLIOGRAFIA

Autor: Phillip R. Robinson.
 Obra: MASTERING THE 68000 MICROPROCESSOR.
 Editorial: Tab books inc. 1985.

Autor: William Cramer/Gerry Kane.
 Obra: Manual del microprocesador 68000.
 Editorial: McGraw Hill. 1987

Autor: Steve Wood.
 Obra: Turbo pascal.
 Editorial: McGraw Hill. 1987

Obra: Introducción a los microprocesadores
 usando el MC 68000.
 Editorial: Editado por la Facultad de Ingeniería.
 1989

Autor: Lance A. Levental.
 Obra: 6800 Assembly Language Programming.
 Editorial: Osborne/McGraw Hill. 1980

Autor: John J. Donovan.
 Obra: System Programming.
 Editorial: McGraw Hill. 1987