UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

"SEC: un Sistema para la Enseñanza de Computación"

T E S I S

Que para obtener el título de

A C T U A R I O

p r e s e n t a

CELIA TERAMOTO MATSUBARA

Director de tesis: Mat. Salvador López Mendoza

FALLA DE ORIGEN

México, 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.

INDICE

INDICE

INTRODUCCION 6

CAPITULO 1. HARDWARE

Recursos bàsicos de un sistema de cômputo 9

Hardware 10

1.1 Memoria 12

1.1.1 Tipos de memoria 12
1.1.2 Organizaciòn de la memoria 14
1.1.3 Funcionamiento de la memoria 17

1.2 Procesador o UCP 19

1.2.1 Tipos de procesador 19
1.2.2 Organizaciòn del procesador de un solo acumulador 20
1.2.3 Funcionamiento del procesador 22
1.2.4 Instrucciones de màquina 24
1.2.4.1 Movimiento de datos 25
1.2.4.2 Operaciones aritmèticas y lògicas 26
1.2.4.3 Transferencia de control 26
1.2.4.4 Entrada/Salida 27
1.2.5 Registro de condiciòn 27
1.2.5.1 Bits de condiciòn 28
1.2.5.2 Control del programa 28
1.2.6 Modos de direccionamiento 29
1.2.6.1 Direccionamiento directo 29
1.2.6.2 Direccionamiento indirecto 30
1.2.7 Datos 31

1.3 Entrada/Salida (E/S) 33

1.3.1 Dispositivos perifèricos 34
1.3.1.1 Terminal en linea 34
1.3.1.2 Disco flexible (diskette) 35

1.3.2 Funcionamiento de los dispositivos de Entrada/Salida 38
1.3.3 Instrucciones de Entrada/Salida 38

CAPITULO 2. SOFTWARE

Software 42

2.1 Lenguajes de programación 43

2.2 Lenguaje de máquina 44

2.3 Lenguaje ensamblador y el ensamblador 45

2.3.1 Lenguaje ensamblador 45
2.3.1.1 Mnemónicos 47
2.3.1.2 Pseudo-instrucciones 47
2.3.1.3 Formato del lenguaje 48
2.3.1.4 Subrutinas 50
2.3.2 El ensamblador 51
2.3.2.1 Ensamblador de dos pasos 54
2.3.2.2 Errores detectados por un ensamblador 56
2.3.2.3 Literales 57
2.3.2.4 Tiempo de ensamble vs. tiempo de ejecución 59

2.4 Lenguaje de alto nivel y el compilador 60

2.4.1 Lenguaje de alto nivel 60
2.4.1.1 Definición del lenguaje 62
2.4.2 El compilador 64
2.4.2.1 Estructura del compilador 66
2.4.2.2 Análisis léxico 68
2.4.2.3 Análisis sintáctico 69
2.4.2.4 Análisis semántico y generación de código 72

2.5 Sistema operativo 74

2.6 Integración del sistema 79

2.6.1 Implementación técnica 84

CAPITULO 3. IMPACTO Y USO DEL SISTEMA "SEC" EN LA ENSEÑANZA DE COMPUTACION 87

CONCLUSIONES 101

APENDICE A

Conjunto de instrucciones 105

APENDICE B

Representación de los datos 109

APENDICE C

Diagrama de flujo del ensamblador de dos pasos 117

APENDICE D

Sintaxis del lenguaje 120

APENDICE E

Listado del programa compilador 124

APENDICE F

Manual de uso del sistema "SEC" 138

BIBLIOGRAFIA 145

GLOSARIO DE TERMINOS 148

INTRODUCCION

INTRODUCCION

La computadora nació con la Segunda Guerra Mundial. La Universidad de Harvard y la empresa IBM en colaboración con la Marina de los Estados Unidos produjeron la MARK I; ésta ocupaba centenares de metros cúbicos y su funcionamiento producía un ruido enloquecedor.

El Ejército Estadounidense también destinó fondos para la construcción de una computadora cuyo objetivo era el de calcular con mayor precisión las trayectorias balísticas.

La ENIAC (Electronic Numerical Integrator And Calculator) que fue el resultado del proyecto militar se terminó hasta el año 1946, o sea, varios meses después de terminada la guerra. Era rápida pero torpe en algunos aspectos, su memoria era pequeña y cada cálculo necesitaba un cambio completo de los cables.

John Von Neuman transformó dichas calculadoras electrónicas en cerebros electrónicos, el concepto real de la computadora.

Por la cantidad de información que existe sobre todos los aspectos que rodean la vida (deporte, política, ciencia, arte, religión, comunicación, mercado de valores, diversión, historia, impuestos, educación, petróleo, etc.) surgió la necesidad de un dispositivo dedicado a almacenar, clasificar, modificar, seleccionar, comparar y presentar la información a una alta velocidad. Y éste ha sido la computadora.

Así, la era en que las computadoras sólo se consideraban como dispositivos para realizar cálculos matemáticos y tareas tediosas, cansadas, monótonas y repetitivas, ha pasado. En la actualidad, la computación es una parte importante en la vida de todos, afectando los patrones de pensamiento y la forma de vida de todo individuo, y en un futuro próximo no habrá área alguna con la que no se encuentre relacionada.

Esta tesis se llama "SEC : un Sistema para la Enseñanza de Computación" por el propósito que se busca, el cual consiste en diseñar una herramienta que se pueda utilizar en los cursos de computación para dar un panorama general de la computadora, explicando su arquitectura, funcionamiento y programación.

Lo anterior se llevará a cabo mediante la simulación de una computadora, dando una idea de todos los conceptos fundamentales que encierra. Se enfatizará en los principios básicos en lugar de los detalles propios de una computadora particular.

Hacer la introducción a esta área tan facinante y apasionada como es la computación, de manera fácil y amena es otro de los objetivos trazados.

En el capítulo 1 se describen los componentes del HARDWARE y el funcionamiento de cada uno de ellos. En el capítulo 2 se tratan los términos y conceptos fundamentales que encierran los lenguajes de programación, sus traductores y el Sistema Operativo.

En ambos capítulos, conforme se van desarrollando los temas, se describe su aplicación de cada uno al sistema "SEC".

En el capítulo 3 se discute el impacto y el uso del sistema "SEC" como una herramienta en la enseñanza de computación, describiendo sus características y aplicaciones.

CAPITULO 1

HARDWARE

RECURSOS BASICOS DE UN SISTEMA DE COMPUTO

La mayoría de las computadoras en la actualidad son digitales, èsto es, almacenan y manipulan entidades representadas por dígitos.

Las computadoras en primera instancia, fueron desarrolladas como respuesta a la necesidad de formas màs ràpidas y màs eficientes de realizar cálculos numéricos. Después de este uso inicial de las computadoras, se exploraron otras àreas de aplicaciòn, lo cual puede ser constatado por la gran cantidad de àreas donde se encuentra involucrada.

Tomando como un todo a la computadora, es un sistema increíblemente complejo, formado por dos grandes niveles. El nivel bajo es el HARDWARE, los circuitos electrònicos y los dispositivos electromecànicos que constituyen el sistema, es decir, la parte física de una computadora, la cual proporciona la capacidad de efectuar las operaciones bàsicas.

El nivel alto es el SOFTWARE, la parte lògica constituida por la secuencia de instrucciones o programas que hacen a la computadora útil para el trabajo, los cuales le indican què hacer y cuàndo hacerlo. El HARDWARE y SOFTWARE juntos constituyen el sistema de cómputo.

Cada uno de los dos niveles mencionados puede ser descompuesto en niveles adicionales.

A continuaciòn estos niveles serán descritos con mayor detalle y aplicados a una computadora simulada (COSI de COmputadora SImple) la cual nos ayudarà con los objetivos trazados en esta tesis. COSI ha sido cuidadosamente diseñada para incluir características generales de cualquier computadora simple, evitando complejidades no usuales e irrelevantes para el objetivo de este trabajo.

HARDWARE

Los componentes del HARDWARE de un sistema se clasifican en tres grupos, como se muestra en la fig. 1 :

- Memoria
- Unidad Central de Procesamiento (UCP) o Procesador
- Entrada/Salida (E/S)

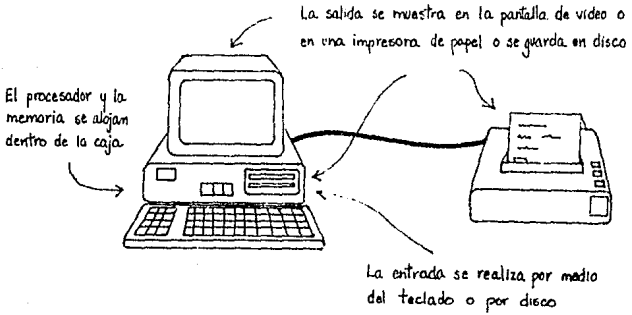


Fig. 1 Componentes del HARDWARE de una computadora.

La conexión de estos subsistemas se muestra en el siguiente diagrama :

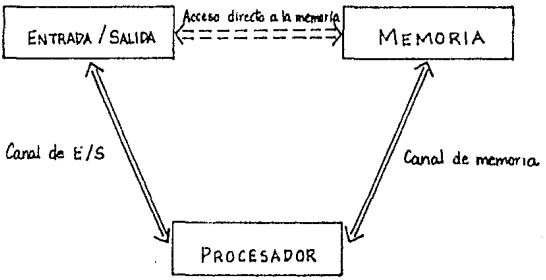


Diagrama de una computadora simple

1.1 MEMORIA

La memoria es el dispositivo utilizado para almacenar la información (instrucciones y datos) y se encuentra enlazada al procesador via el canal de memoria. Un canal es un medio fisico para transferir la información.

1.1.1 TIPOS DE MEMORIA

Existen varios tipos de memoria. La memoria conocida como MLE es una memoria de lectura y escritura. Puede almacenar datos en cualquier dirección y leerlos en cualquier momento. También es conocida como memoria principal y en general, es aquí donde se almacenan los programas. Es la forma de acceso del programador a una computadora.

La memoria de lectura exclusiva (MLEX), puede leer el contenido de cualquier dirección en cualquier momento pero la información sólo es almacenada una vez, al momento de su fabricación.

La diferencia entre ellas es que una se puede leer y escribir con igual facilidad (MLE) mientras que la otra sólo puede ser leída (MLEX) (ver fig. 2). Desafortunadamente la MLE es volátil, ésto es, pierde todo cuando se inactiva; a diferencia, la MLEX es no volátil.

También existe la memoria de lectura exclusiva/programable (MLEX/P), que se puede programar de acuerdo a las especificaciones del usuario.

La memoria de lectura exclusiva reprogramable (MLEX/REP) es similar a la MLEX/P sólo que ésta puede ser borrada. No entraremos en más detalle.

Muchas de las computadoras actuales tienen una MLE de gran capacidad, pero en la mayoría de los casos no resulta suficiente para almacenar toda la información leída (además de ser volátil).

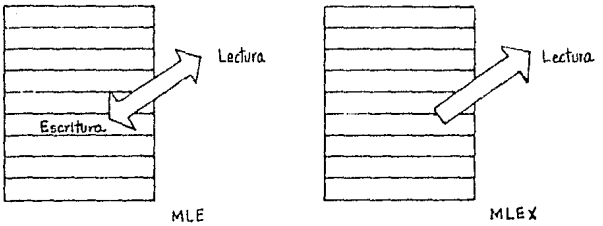


Fig. 2 Memoria de lectura y escritura (MLE) y Memoria de lectura exclusiva (MLEX).

Como respuesta a esta necesidad existen los dispositivos de almacenamiento masivo también conocidos como memoria secundaria, que pueden almacenar una gran cantidad de información. Como ejemplo tenemos cintas y discos magnèticos, diskettes, etc., que se ilustran en la fig. 3.

Fig. 3 Las cintas y discos magnéticos, así como los discos flexibles son dispositivos conocidos como memoria secundaria.

1.1.2 ORGANIZACION DE LA MEMORIA

La memoria de una computadora consta de una área de almacenamiento, dos registros llamados RDAM (Registro de Datos de Memoria) y RDIM (Registro de Direcciones de Memoria) y tres controles de entrada : ocupado/desocupado, lectura/escritura y byte/palabra, la cual se ilustra en la fig. 4.

El área de almacenamiento es simplemente un arreglo de bytes (en muchas ocasiones, a ésta se le conoce como la memoria sin considerar a los registros). El byte es una unidad de almacenamiento compuesto generalmente de 8 bits y equivalente a un caracter de información. Los bytes se utilizan para hacer referencia al tamaño de la información, por ejemplo, un texto de 100,000 bytes equivale a uno de 100,000 caracteres.

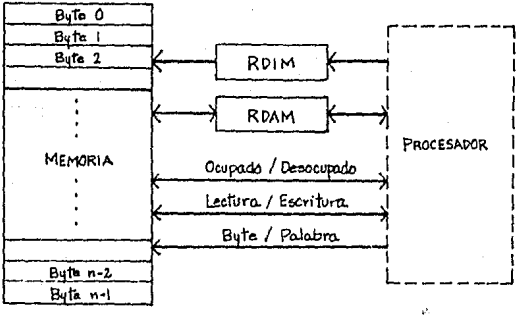


Fig. 4 La memoria consta de una área de almacenamiento, dos registros : RDIM y RDAM, y tres controles : ocupado / desocupado, lectura / escritura y byte / palabra.

Bit es la unidad bàsica de almacenamiento, el cual representa dígitos binarios, es decir, un bit sólo toma valores 0 ò 1. Toda informaciòn (incluyendo las instrucciones) se convierte en nùmeros binarios, èsto porque la mayoría de los dispositivos físicos usados en las màquinas sólo pueden retener uno de dos estados (apagado o prendido, positivo o negativo).

Los bits se agrupan en unidades que son procesadas y almacenadas por la computadora. Ocho bits forman un byte pero tambièn pueden formar palabras, que son unidades internas de almacenamiento. Una palabra puede representar un nùmero o uno o màs caracteres de informaciòn no numèrica. El largo de las palabras difiere de una màquina a otra. Sin embargo, una palabra consiste de un nùmero de bits que comùnmente va de 8 a 64 bits (1 byte a 8 bytes). (ver fig. 5)

Asi, cada bit de una palabra de b bits puede tener como valor 0 ò 1 independientemente, por lo tanto, una palabra puede asumir 2^b estados diferentes.

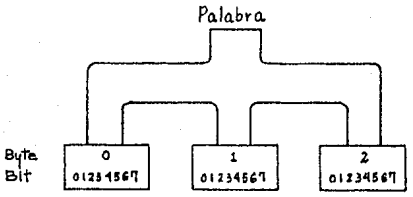


Fig. 5 Los bits se combinan para formar bytes y los bytes se agrupan para formar palabras. Este ejemplo muestra una palabra de tres bytes (24 bits).

Cada sucesiòn de bits (byte) en memoria (de ahora en adelante asi llamaremos a la àrea de almacenamiento, memoria) se conoce como localidad de memoria y cada localidad tiene asociada una direcciòn ùnica, la cual es un nùmero binario.

Si la memoria tiene n localidades , entonces el intervalo de las direcciones va de 0 a n-1 y las direcciones serían 0,1,2,3,....n-1. Esto se puede observar en la fig. 6.

Naturalmente, las computadoras reales no pueden tener memorias infinitas, por lo que existe un límite superior sobre la cantidad de información que puede ser almacenada en una memoria.

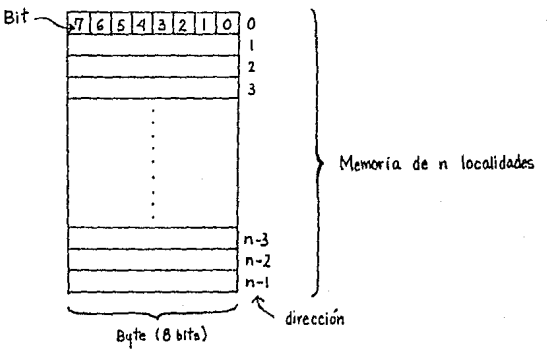


Fig. 6 Organización de la memoria, la cual es un arreglo de bytes.

En la fig. 7 se ilustra la organización de la memoria de COSI, la cual consiste de bytes de 8 bits; tres bytes consecutivos forman una palabra (24 bits). Las direcciones son direcciones de bytes y las palabras se direccionan por medio de la localidad o dirección de su primer byte.

Por el momento, para el propósito de enseñanza que se tiene, la memoria tiene un total de 2^9 bytes, ésto es, 512 localidades cuyo rango irá de 0 a 511; ésto cubre el espacio requerido para almacenar los programas con los que se trabajará puesto que no serán ni complejos ni muy largos.

Sin embargo, la memoria puede crecer hasta tener un total de 2^{15} bytes, es decir, 32,768 localidades, ya que las instrucciones y los registros (que se verán más adelante) manejan un campo de dirección de 15 bits.

Se incluye también los registros RDAM y RDIM.

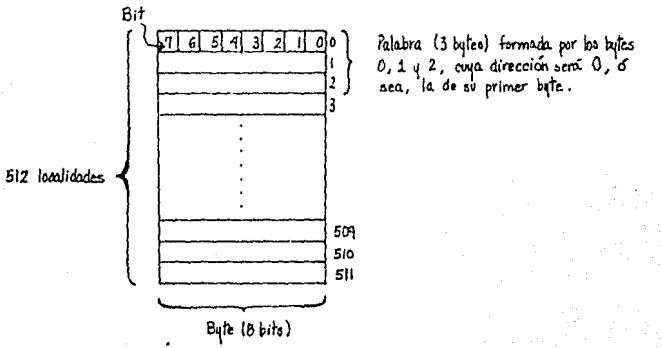


Fig. 7 Organización de la memoria de COSI

1.1.3 FUNCIONAMIENTO DE LA MEMORIA

Las instrucciones provenientes de un dispositivo periférico se copian en la memoria y el procesador extrae una por una para su ejecución (el procesador se explicará posteriormente).

El acceso a la memoria es un acceso aleatorio, ésto implica que el tiempo requerido para accesar una determinada localidad es invariable, no depende de las localidades anteriormente accesadas, ni del tipo de información que es leída o escrita.

El proceso de acceso a la memoria principal es muy sencillo. Para almacenar una palabra en un lugar particular en la memoria, la dirección de la palabra se coloca en RDIM y la información en RDAM. El control de entrada lectura/escritura se activa al estado escritura, el control byte/palabra a palabra y el control ocupado/desocupado a ocupado.

Cuando la memoria capta la señal de ocupado, comienza su ciclo de operación; la señal palabra indica que el RDAM contiene una palabra de información y que el RDIM contiene una dirección, la señal escritura causa que se sobreescriba la información almacenada en RDAM en una palabra de memoria, la cual comienza en la dirección almacenada en RDIM. Una vez realizado el proceso, el control ocupado/desocupado pasa a desocupado. El almacenamiento de un byte es similar al almacenamiento de una palabra, excepto que el control byte/palabra es activado a byte.

La lectura de una palabra se ejecuta colocando la dirección de la palabra en RDIM, el control byte/palabra indicando palabra y el control lectura/escritura indicando lectura. Cuando la señal ocupado se emite, se copia la palabra de información que comienza en la dirección almacenada en RDIM, en el RDAM. La lectura de un byte es similar a la lectura de una palabra, excepto que el control byte/palabra indicará byte.

Una forma màs compacta de expresar el funcionamiento de la memoria (en caso de tratarse de palabras) es dada por el siguiente simbolismo :

Escritura : Memoria [RDIM..RDIM+2] <--- RDAM

Lectura : RDAM <--- Memoria [RDIM..RDIM+2]

En el caso de nuestra màquina COSI, para facilitar la comprensión, tanto la lectura como la escritura se harà en bytes.

1.2 PROCESADOR O UCP

El procesador o UCP es el corazòn de la computadora. Es el dispositivo encargado de manipular los datos almacenados en la memoria principal bajo el control de un programa tambièn almacenado en ella (ver fig. 8). Un programa es una serie de instrucciones, cada una de las cuales le indica a la computadora ejecutar una de sus funciones bàsicas -sumar, restar, multiplicar, dividir, comparar, copiar, etc.

Un procesador simple contiene circuitos de control (unidad de control) para mandar, traer y ejecutar instrucciones, una unidad aritmètica y lògica (UAL) para la manipulaciòn de datos (ejecuciòn de las instrucciones aritmèticas y lògicas) y registros que llevan el estado o condiciòn del procesador.

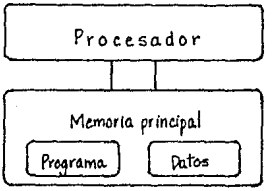


Fig. 8 El procesador manipula los datos almacenados en la memoria principal bajo el control de un programa también almacenado en ella.

1.2.1 TIPOS DE PROCESADOR

La organizaciòn del procesador màs simple tiene uno o dos registros llamados acumuladores, donde las operaciones aritmèticas y lògicas y la transferencia de datos toman lugar.

Existen por lo tanto, procesadores de un solo acumulador y procesadores de dos acumuladores. El tercer tipo es el llamado procesador de registros generales que contiene màs de dos acumuladores.

Nuestro estudio se enfocarà a màquinas con procesador de un solo acumulador; sobre esta base serà construido el procesador de COSI.

Se escogiò este tipo de procesador por ser el màs sencillo y fàcil de comprender, ademàs de contener todos los elementos bàsicos de un procesador.

1.2.2 ORGANIZACION DEL PROCESADOR DE UN SOLO ACUMULADOR

El procesador està formado de varios registros y unidades de funciòn. A continuaciòn se ilustra la organizaciòn interna de un procesador simple de un solo acumulador :

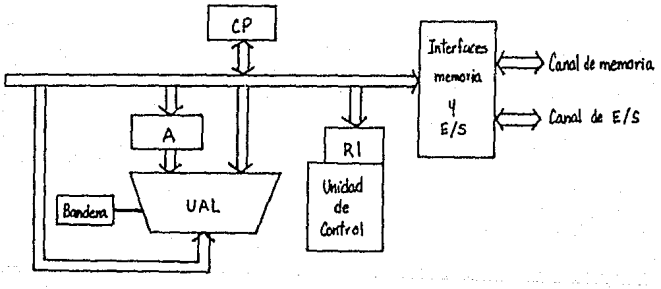


Diagrama de un procesador simple de un solo acumulador.

Brevemente se describirán los registros y unidades. Se define además para cada uno la longitud, el número de registro y el mnemónico con el que se maneja para su representación en la estructura del procesador de COSI.

-REGISTRO DE INSTRUCCION (24 bits , 1 , RI)
contiene la instrucción que va a ser ejecutada, la cual debe decodificarse.

-REGISTRO DE DIRECCION EFECTIVA (16 bits , 2 , RDE)
contiene la dirección real de memoria de la cual el procesador lee o escribe durante la ejecución de una instrucción.

-CONTADOR DE PROGRAMA (16 bits , 3 , CP)
contiene la dirección de la localidad de memoria de la siguiente instrucción que será ejecutada.

-ACUMULADOR (24 bits , 4 , A)
contiene el dato para ser procesado.

-REGISTRO DE INDICE (24 bits , 5 , X)
contiene una dirección o un dato para ser usado por un programa. Se usa para el direccionamiento indirecto.

-APUNTADOR DE PILA (16 bits , 6 , AP)
contiene la dirección del tope de la pila que contiene las direcciones de regreso de subrutinas.

-REGISTRO DE CONDICION (3 bits , 7 , RC)
registro cuyo valor el procesador actualiza al final de la ejecución de cada una de las instrucciones que manipulan datos. También es conocido como registro de control.

-UNIDAD ARITMETICA Y LOGICA (UAL)
es el conjunto de circuitos que realizan las funciones aritméticas y lógicas sobre operandos que se encuentran almacenados en el acumulador y en la memoria.

-UNIDAD DE CONTROL (UC)
decodifica las instrucciones y controla otros bloques que hacen venir y ejecutar las instrucciones.

El diagrama del procesador de la màquina COSI con los componentes descritos se muestra en la fig. 9.

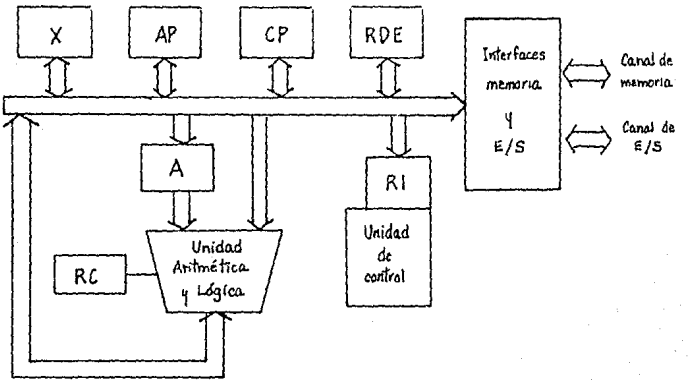


Fig. 9 Estructura del procesador de COSI.

1.2.3 FUNCIONAMIENTO DEL PROCESADOR

La pregunta sería, ¿ cómo trabajan juntos todos los componentes descritos para ejecutar las instrucciones ?. La operación consiste en ejecutar repetidamente tres pasos que dan origen al llamado ciclo de control : primero, leer la siguiente instrucción de la memoria; segundo, decodificarla, es decir, determinar exactamente lo que la computadora debe hacer; y tercero, ejecutar las acciones que se requieran (fig. 10). El ciclo de control es realizado por la unidad de control y la unidad aritmética y lógica.

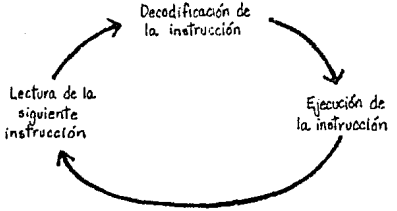


Fig. 10 El ciclo de control.

Los registros CP, A, X y AP son manipulados de manera explícita por las instrucciones, y contienen valores que son significativos para la ejecución de cada instrucción.

El proceso comienza cuando se activa la unidad de control; consulta el contador de programa, encuentra la dirección de la instrucción a ejecutar y la coloca en el RDIM; la instrucción es traída y colocada en el RDAM, para después almacenarla en el registro de instrucción. El traer una instrucción desde la memoria toma tiempo, dando la oportunidad de incrementar el contador de programa para señalar la siguiente instrucción.

A continuación la unidad de control calcula la dirección real del operando que tomará parte en la instrucción y al mismo tiempo determina la operación que se debe llevar a cabo.

Finalmente se activa la UAL y ejecuta la instrucción : obtener y copiar un dato de la memoria al acumulador o viceversa, del acumulador a la memoria; realizar alguna comparación; llevar a cabo una operación aritmética; etc. Por último, el control se transfiere a la unidad de control para comenzar un nuevo ciclo, y así sucesivamente.

Normalmente, el contador de programa se incrementaría de acuerdo al tamaño de la instrucción para contener la localidad de la siguiente instrucción. Como una excepción, algunas instrucciones son de transferencia, por ejemplo, saltos condicionados y no condicionados y llamadas a subrutina. Estas instrucciones son ejecutadas cargando el contador de programa con la dirección indicada en la instrucción.

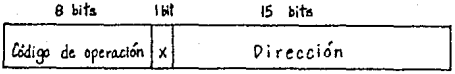
En caso de una subrutina, primero, la dirección almacenada en el contador de programa se guarda en la pila (la cual es parte de la memoria) y el AP se actualiza para indicar el nuevo tope de la pila, y segundo, el contador de programa se carga con la dirección de la subrutina. De esta manera, una vez ejecutada la subrutina, se puede regresar al punto desde donde se llamó. Esto nos permite ejecutar las mismas subrutinas varias veces, tantas como se quiera.

1.2.4 INSTRUCCIONES DE MAQUINA

Las instrucciones especifican las operaciones a realizar por la computadora y los operandos involucrados.

Una instrucción está compuesta de un código de operación y uno o más operandos (fig. 11). El código de operación especifica la función que será ejecutada y los operandos identifican las localidades de memoria de los datos que participan en la operación.

El formato de instrucción en COSI consta de un código de operación y un operando que hace referencia a una localidad de memoria; tiene una longitud de 24 bits (3 bytes). Se ilustra a continuación :



El bit x es una bandera que indica el modo de direccionamiento, directo o indirecto. (Ver 1.2.6 Modos de direccionamiento)

Código de operación	Operando (s)

Fig. 11 Una instrucción está compuesta de un código de operación y uno o más operandos.

Las instrucciones se pueden clasificar de diferentes maneras. La clasificación que seguiremos es de acuerdo a la función que realizan. Así, las instrucciones de una computadora pueden, con algunas excepciones, ser agrupadas de la siguiente manera :

1. movimiento de datos,
2. operaciones aritmèticas y lògicas,
3. transferencia de control y,
4. Entrada/Salida

1.2.4.1 Movimiento de datos

Causa que el contenido de una localidad de memoria sea colocado en el acumulador o en algún registro y viceversa. Al hacer la copia, el dato permanecerá sin ser borrado en el lugar de origen, así existirán dos copias del dato.

Como ejemplos se tienen CARGAA (carga el acumulador), CARGAX (carga el registro de índice), GUARDA (almacena el contenido del acumulador en alguna localidad de memoria), etc.

1.2.4.2 Operaciones aritmèticas y lògicas

Son aquellas que manipulan los datos. La operaciòn se lleva a cabo sobre el dato contenido en alguna localidad de memoria y el dato almacenado en el acumulador (A), dejando el resultado en este último.

Como ejemplo, estàn las operaciones aritmèticas SUMA, RESTA, etc., y las comparaciones COMP, COMPF.

1.2.4.3 Transferencia de control

Las instrucciones de transferencia pasan por encima de la secuencia normal de instrucciones, alterando el contador de programa.

El efecto de una transferencia incondicional es cargar una direcciòn especifica en el contador de programa. La transferencia condicional prueba cierta condiciòn y carga la direcciòn en el CP si la condiciòn es satisfecha. Si la condiciòn no se cumple, el CP no sufre modificaciòn siguiendo el flujo normal. Se tienen como ejemplos SALTA y SALTIG (salta si resulta igual a cero) para la transferencia incondicional y condicional, respectivamente.

Como caso especial se encuentra la llamada a subrutina. Una subrutina es una secuencia de instrucciones, definida y almacenada una sola vez, que puede ser llamada (o invocada) desde cualquier lugar.

Una ventaja de usar subrutinas es obvia : se reduce el tamaño del programa por almacenar sòlo una vez una secuencia de instrucciones, en lugar de repetirla cada vez que se necesita; sòlo es necesaria una instrucciòn para invocarla.

Otra ventaja crucial en el desarrollo de programas largos es que èstos pueden ser divididos en tareas individuales, que son definidas y procesadas por subrutinas con interfaces e interacciones bien definidas con el resto del programa. De esta manera, diferentes programadores pueden trabajar sobre subrutinas diferentes en forma independiente del resto del programa.

Se necesita guardar el valor actual del CP cada vez que la subrutina es llamada y, al terminar ésta, reemplazar con ese valor el valor del CP.

1.2.4.4 Entrada/Salida

Las computadoras tienen canales para comunicar la memoria principal con algún medio de almacenamiento secundario o con el mundo exterior (dispositivos periféricos). Por lo que se necesitan instrucciones para la lectura o escritura de datos, ejemplo LEEENT (lectura de un entero).

Todas estas instrucciones de los 4 grupos forman el lenguaje de máquina de una computadora.

Un programa entendible por la computadora debe ser expresado en lenguaje de máquina, es decir, debe ser codificado en una forma numérica binaria (ceros y unos). A este programa se le conoce como programa en lenguaje de máquina.

Las instrucciones soportadas por COSI son las instrucciones básicas que pueden ser encontradas en cualquier computadora simple.

En el Apéndice A se encuentra definido el código de operación (en hexadecimal) de cada instrucción disponible en COSI, así como una explicación de la función que realizan. Se encuentran agrupadas de acuerdo a la clasificación antes mencionada.

1.2.5 REGISTRO DE CONDICION

En los procesadores basados en un acumulador, una instrucción de salto condicional prueba el valor del acumulador y salta de acuerdo a una condición específica (es decir, cero / no cero, positivo / negativo, etc.).

En un programa, un porcentaje alto de todas las instrucciones son saltos condicionados y es por ésto que es muy importante optimizarlos y hacer su ejecución lo más eficaz.

El estado del acumulador para este tipo de instrucciones se consulta en el registro de condición que consta de varios bits de condición.

1.2.5.1 Bits de condición

Los bits de condición (o códigos de condición) son una colección de bits individuales cuyos valores el procesador actualiza automáticamente de acuerdo al resultado de algunas instrucciones. Los bits de condición son afectados por instrucciones de movimiento y manipulación de datos.

El nombre y significado de los bits de condición varían en los diferentes procesadores, pero el conjunto más popular encontrado en la mayoría de las computadoras es el siguiente :

N (negativo). Se iguala a 1 si el valor del acumulador es negativo y 0 en otro caso.

C (cero). Se iguala a 1 si todos los bits de un resultado son cero y 0 en otro caso.

S (sobreflujo). Se iguala a 1 en operaciones aritméticas que causan el sobreflujo en complemento a dos y 0 si ningún sobreflujo sucede.

1.2.5.2 Control del programa.

Las instrucciones para el control del programa tienen una importancia fundamental, ya que son las primitivas que permiten la acción y repetición condicional.

Las instrucciones de salto condicionado prueban una condición específica y saltan sólo si la condición resulta ser verdadera. En procesadores sin códigos de condición, las condiciones típicas son "Acumulador cero", "Acumulador negativo". En procesadores con códigos de condición, los valores de uno o más bits de condición son probados.

Las condiciones de salto proporcionadas por los bits N, C y S son :

Mnemònico	Salta si	Condiciòn
SALTIG	Igual (a cero)	C = 1
SALTDIF	Diferente (a cero)	C = 0
SALTME	Menor que	N xor S = 1
SALTMA	Mayor que	(N xor S) or C = 0
SALTMEIG	Menor o igual a	(N xor S) or C = 1
SALTMAIG	Mayor o igual a	N xor S = 0

Las instrucciones de salto condicionado se utilizan casi siempre despuès de la instrucciòn de comparaciòn (COMP o COMPF), la cual coloca los bits de condiciòn de acuerdo al resultado de (A) - V, donde (A) indica el contenido del acumulador y V el valor con el que se compara.

1.2.6 MODOS DE DIRECCIONAMIENTO

El propòsito de los modos de direccionamiento es el de proporcionar la direcciòn real de un operando que va a ser manipulado. En una instrucciòn que manipula datos, la direcciòn real es la direcciòn en donde realmente se encuentra el dato; en una instrucciòn de salto serìa la direcciòn a donde debe transferirse el control.

Los modos de direccionamiento pueden ser clasificados como directos o indirectos.

1.2.6.1 Direccionamiento directo

En el modo de direccionamiento directo, la direcciòn real se toma de la instrucciòn, no ofrece mayor dificultad.

1.2.6.2 Direccionamiento indirecto

Es una técnica utilizada para el acceso a la memoria y es muy útil para calcular la dirección al momento de la ejecución de un programa. El direccionamiento indirecto consiste de hacer referencia a las localidades de memoria en forma indirecta.

En este modo de direccionamiento indirecto la dirección real se calcula utilizando el contenido del registro X (registro de índice) y el valor que se encuentra en la instrucción.

En COSI están definidos los dos modos de direccionamiento y se indican por el valor del bit x de la instrucción.

La siguiente tabla describe cómo se calcula la dirección real a partir de la dirección dada en la instrucción y del valor del bit x.

Modo	Indicación	Cálculo de la dirección real (DR = Dirección Real)
Directo	x = 0	DR = dirección
Indirecto	x = 1	DR = dirección + (X)

En donde (X) se refiere al contenido del registro X.

Por ejemplo, supongamos que en la localidad 31 de la memoria se encuentra almacenado el número 75 y en la 61 el 1346, ambos como enteros :

```
31    00000000 00000000 01001011
 .
 .
61    00000000 00000101 01000010
```

Además, el registro X igualado al valor 30 (00000000 00000000 00011110) y el acumulador al valor 67 (00000000 00000000 01000011). La instrucción a ejecutar es la siguiente :

00010000 10000000 00011111

como el bit x está prendido (igual a uno), la dirección real del operando que se suma (ya que el código de operación es 00010000, que corresponde a la instrucción SUMA) al contenido del acumulador es la 61 (31 + 30).

Es decir, el resultado que se obtiene de la suma es 1413, en lugar del 142 que resultaría de sumar el contenido del acumulador con el contenido de la localidad 31, ésto en caso de que el bit x estuviera apagado (igual a cero).

1.2.7 DATOS

Los datos son la unidad de información codificados en grupos de bits. Son los elementos básicos de la información, procesados o producidos por la computadora.

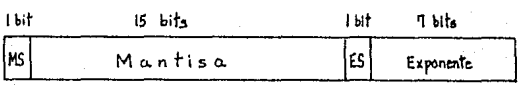
Muchas computadoras almacenan y manipulan enteros (punto-fijo), reales (punto-flotante) y caracteres, como elementos básicos y primitivos.

En el Apéndice B se pueden consultar los distintos sistemas numéricos (binario, octal y hexadecimal) y las notaciones utilizadas para los enteros, reales y caracteres.

Los tipos de datos que se pueden encontrar en COSI son enteros y reales.

Los enteros se almacenan como números binarios de 24 bits. La representación complemento a dos se usa para representar los números negativos en punto-fijo.

Los reales tienen una longitud de 24 bits, cuyo formato es el siguiente :



MS es el signo de la mantisa (0 = positivo, 1 = negativo)

La mantisa se interpreta por un valor entre 0 y 1. La representación utilizada es la de signo-magnitud y es normalizada. Para normalizar números en notación punto flotante, el bit de mayor orden de la mantisa debe ser 1.

ES es el signo del exponente (0 = positivo, 1 = negativo).

El exponente es un número binario positivo, cuyo rango oscila de 0 a 127 con una representación de complemento a dos.

Ejemplos :

Número	MS	Mantisa	ES	Exponente
0	0	0000000000000000	0	0000000
3	0	1100000000000000	0	0000010
20	0	1010000000000000	0	0000101
-20	1	1010000000000000	0	0000101
65.32	0	1000001010100001	0	0000111
-13492	1	1101001011010000	0	0001110

1.3 ENTRADA/SALIDA (E/S)

De los tres grandes subsistemas de una computadora, èste es el que màs evoluciòn ha sufrido.

El subsistema de Entrada/Salida es el medio de comunicaciòn entre la representaciòn interna usada por la computadora y la representaciòn externa. Es el medio de acceso de la gente a las computadoras.

El subsistema de E/S està constituido de dispositivos e interfaces (ver fig. 12). Un dispositivo periférico ejecuta funciones para observar, controlar y comunicarse con el mundo exterior de la computadora. Una interfaz de E/S controla la operaciòn del dispositivo periférico de acuerdo a los comandos del procesador.

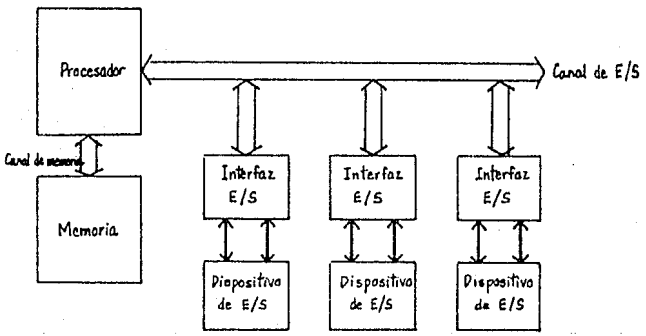


Fig. 12 Subsistema de E/S.

1.3.1 DISPOSITIVOS PERIFERICOS

Un sistema contemporàneo incluye dispositivos de E/S, formalmente llamados dispositivos periféricos.

Los dispositivos periféricos pueden clasificarse en dos grandes gupos : dispositivos de E/S y dispositivos de almacenamiento.

Un dispositivo de entrada es el medio por donde la computadora siente el mundo exterior (convierte los datos del mundo exterior a formas usables por la computadora), como una lectora de tarjetas, un teclado, una palanca de juegos.

Un dispositivo de salida es el medio por el cual la computadora afecta o controla el mundo exterior (convierte los datos de la computadora a una forma que sea útil en el mundo exterior de ésta), como una perforadora de tarjetas, una pantalla de video, una impresora, una graficadora.

Un dispositivo de almacenamiento es un mecanismo en el cual la computadora puede almacenar información mediante un procedimiento llamado escritura, y tal información puede ser recuperada màs tarde (lectura). Como ejemplos tenemos las cintas magnéticas, los discos (flexibles y duros), etc. Con estos dispositivos se puede hacer tanto entrada como salida.

A continuación se dará una descripción de algunos de los dispositivos mencionados.

1.3.1.1 Terminal en línea

Uno de los dispositivos màs común de E/S es la terminal en línea. Esta consiste de un teclado por el cual el usuario indica los datos y un dispositivo de salida, que puede ser una pantalla de video y/o una impresora.

La pantalla de video despliega comùnmente, de 20 a 24 líneas de 60 a 80 caracteres por línea.

La entrada por el teclado está limitada por la velocidad de tecleo de las personas. La velocidad de salida está determinada por la capacidad de la línea que conecta la terminal a la computadora.

1.3.1.2 Disco flexible (diskette)

El medio de almacenamiento secundario más común en una microcomputadora es el disco flexible o diskette; es una pieza circular delgada flexible de poliéster revestida con un material magnético (ver fig. 13).

Los datos son grabados en una o en ambas superficies (caras). La unidad de disco trabaja en una forma muy parecida a un plato giratorio. El agujero en el centro permite empotrarse para hacerlo girar; el mecanismo de acceso consiste de una cabeza de lectura/escritura que se encuentra montada sobre un brazo movible que le permite posicionarse sobre cualquier pista del disco, como se ve en la fig. 14.

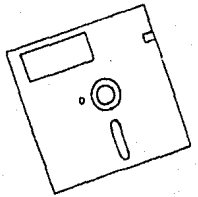


Fig. 13 El medio más popular de almacenamiento secundario en microcomputadoras es el disco flexible (diskette).

Las pistas son una serie de círculos concéntricos sobre los cuales se graban los datos. El mecanismo de acceso camina de pista a pista, leyendo o escribiendo. Las pistas son subdivididas en sectores que son numerados secuencialmente - 0, 1, 2 y así sucesivamente. Esto se ilustra en la fig. 15.

El contenido de un sector es lo que se mueve entre el disco y la memoria principal.

Debido a que la capacidad de almacenamiento de un disco puede contener cientos de programas y datos de diferentes aplicaciones, ¿ cómo puede la computadora encontrar el programa o dato correcto ?. Cuando un programa se almacena en el disco, normalmente se registra en un conjunto consecutivo de sectores o en sectores no consecutivos que son ligados. Así, si la computadora puede encontrar el comienzo del programa, puede hallar el programa completo.

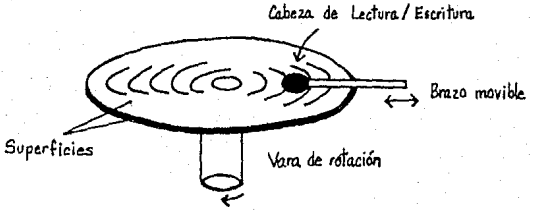


Fig. 14 Acceso al disco.

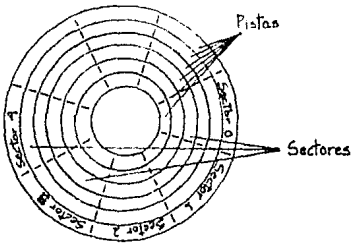


Fig. 15 Los datos se registran en una serie de círculos concéntricos llamados pistas. Las pistas son subdivididas en sectores.

Ahora la pregunta sería, ¿ cómo se puede determinar donde comienza un programa dado ?. Una porción de la primera pista se reserva para contener un directorio. Cuando un programa es escrito en el disco se le asigna un nombre. El nombre es registrado entonces en el directorio junto con la dirección de la pista y sector donde comienza.

Para recuperar el programa, basta dar el nombre del programa. Los datos son accesados de la misma manera; los datos de una aplicación se agrupan en un archivo al cual también se le asigna un nombre.

Por lo tanto, se resume que, el dispositivo básico de entrada en casi todas las microcomputadoras es el teclado y el dispositivo de salida es la pantalla de video. Y el de almacenamiento secundario, el disco flexible o diskette.

Los dispositivos periféricos con que cuenta COSI son :

- teclado y,
- pantalla de video.

1.3.2 FUNCIONAMIENTO DE LOS DISPOSITIVOS DE E/S

Muchas de las terminales están conectadas a la computadora por dos circuitos independientes (canales): uno conecta el teclado a la computadora para la entrada y otro conecta la computadora al dispositivo de salida.

El uso de dos canales o líneas de transmisión independientes permite transmitir en ambas direcciones simultàneamente. Aunque en varios casos cada caracter tecleado por el usuario es inmediatamente impreso o desplegado en pantalla.

Algunas computadoras tienen una sola línea de transmisión. En tales sistemas el usuario debe tener cuidado de no teclear mientras la computadora està transmitiendo información a la terminal o viceversa, ya que si ocurre que al mismo tiempo hay dos señales en la línea de transmisión se produciría un conflicto entre ellas y la información se perdería.

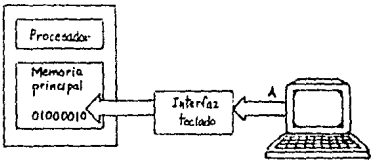
Pero en la actualidad, dicha situación ya no es problema para el usuario, ya que existen mecanismos para solucionarlo : Fullduplex (para computadoras con dos líneas de transmisión) y Halfduplex (para las de una sola línea).

Para llevar a cabo la transmisión de la información existe la interfaz de E/S. Una interfaz de E/S controla la operación del dispositivo periférico de acuerdo a los comandos del procesador, ésta además convierte los datos de la computadora en el formato que es requerido por el dispositivo o por la memoria (fig. 16) y se encarga de sincronizar la transmisión.

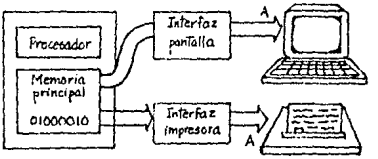
El subsistema de E/S no será tratado ampliamente en COSI, ya que el nivel en el que se trabajan los temas no lo comprende.

1.3.3 INSTRUCCIONES DE E/S

La mayoría de las computadoras tienen un conjunto especial de instrucciones de E/S. Como se mencionó, las computadoras tienen canales de datos para comunicar la memoria principal con el mundo exterior o con su medio de almacenamiento secundario.



a) La entrada desde el teclado pasa por la interfaz y se convierte a la representación interna de la computadora.



b) Los datos almacenados en la memoria principal se mandan a la pantalla o impresora, convertidos a su representación externa

Fig. 16 Las funciones de la interfaz de Entrada / Salida

Ya que la computadora debe esperar cada vez que los datos se mueven, el tiempo de ejecución de un programa está determinado en gran parte por la E/S. Por lo tanto, la necesidad de canales es evidente.

Los canales operan mientras que la unidad de control realiza otros trabajos. Un canal de dato moverá, èl mismo, un byte o una palabra a la memoria principal o hacia fuera de ella.

Si una computadora tiene un canal de datos (o canales) debe haber en el repertorio de instrucciones del procesador aquellas que dan control al canal o canales; pruebas para la transferencia condicional si el canal ha terminado con su trabajo, si el dato necesitado ha sido colocado en memoria o si el contenido de ciertas localidades de memoria ha sido copiado, etc.

Por la complejidad que encierra este tema, se omitirá de este trabajo. Se asume que el procesador permite todo movimiento de datos sin ejecutar operaciones simultáneamente.

Así, las instrucciones básicas necesarias de una pequeña computadora simple como lo es COSI deben ser: lectura de dato y escritura de dato.

El conjunto de instrucciones de E/S de COSI consta de cuatro, que leen del teclado o escriben en la pantalla (ver Apéndice A).

CAPITULO 2

SOFTWARE

SOFTWARE

En el capítulo anterior se discutió el HARDWARE. Ahora se centrará la atención en el SOFTWARE.

El SOFTWARE de la computadora consiste de las instrucciones que el HARDWARE ejecuta para llevar a cabo una tarea útil.

Los componentes básicos del SOFTWARE de una computadora son :

- Traductores de los lenguajes de programación (realizan la traducción de un lenguaje a otro de más bajo nivel)
- Sistema operativo. (maneja los recursos de una computadora)

2.1 LENGUAJES DE PROGRAMACION

El lenguaje permite la expresiòn de pensamientos e ideas; sin èste, la comunicaciòn como se conoce seria muy difícil.

En la programaciòn de computadoras, un lenguaje de programaciòn sirve como medio de comunicaciòn entre la persona con un problema y la computadora usada para ayudar a resolverlo. Un lenguaje de programaciòn efectivo favorece tanto el desarrollo como la expresiòn de programas.

Un programa para la solución de un problema dado serà màs fàcil y màs natural de obtener si el lenguaje de programaciòn usado està enfocado al problema. Esto es, el lenguaje debe contener construcciones que reflejen la terminología y los elementos utilizados en la descripciòn del problema, independientemente de la computadora que se use.

Una jerarquía de los lenguajes de programaciòn basada en el incremento de la independencia a la màquina es la siguiente:

1. Lenguaje a nivel màquina
2. Lenguaje ensamblador
3. Lenguaje de alto nivel

2.2 LENGUAJE DE MAQUINA

Es el de nivel màs bajo entre los lenguajes de programaciòn. Cada instrucciòn del programa se representa por un còdigo numèrico binario y las direcciones numèricas son usadas a través del programa para referirse a las localidades de memoria; las acciones definidas por cada instrucciòn de màquina son llevadas a cabo por el HARDWARE de la computadora. Este lenguaje se describiò en el capítulo uno, para mayor detalle favor de consultarlo.

2.3 LENGUAJE ENSAMBLADOR Y EL ENSAMBLADOR

EL lenguaje ensamblador y el ensamblador se describirán de acuerdo a los diseñados para COSI, la cual es, como se dijo al principio, una máquina simple con los elementos básicos de una computadora.

2.3.1 LENGUAJE ENSAMBLADOR

Al principio, el programador de computadoras tuvo a su disposición una máquina básica que interpretaba, a través del HARDWARE, ciertas instrucciones fundamentales. El programaba esta computadora escribiendo una serie de ceros y unos (lenguaje de máquina), que colocaba en la memoria de la máquina.

Debido a que los programadores encontraron difícil el escribir y leer programas en lenguaje de máquina, los programas fueron escritos en un lenguaje que tiene una forma más simbólica y estilizada. El más simple de éstos es el lenguaje ensamblador.

El lenguaje ensamblador es una representación simbólica del lenguaje de máquina. El programador escribe una instrucción mnemónica para cada instrucción a nivel máquina y los operandos utilizan símbolos en lugar de números para representar las direcciones de la memoria principal.

Desafortunadamente, no existen computadoras que puedan ejecutar directamente las instrucciones en lenguaje ensamblador. Escribir códigos mnemónicos puede simplificar el trabajo del programador, pero las computadoras son aún máquinas binarias y requieren instrucciones binarias. Así, la traducción se hace necesaria y es como surgen los ensambladores (fig. 17).

Debido a la relación uno a uno entre el lenguaje y la máquina, los ensambladores son dependientes de esta última y un programa escrito para un tipo de máquina no siempre correrá en otra.

Fig. 17 El ensamblador se encarga de traducir el programa en lenguaje ensamblador a código binario.

Aún cuando todos los fabricantes de computadoras definan un lenguaje ensamblador estàndar para una nueva màquina cuando èsta es introducida al mercado, otros usuarios pueden definir diferentes lenguajes ensamblador para la misma màquina.

Mientras que el efecto de cada instrucción en lenguaje de màquina es fijo en el HARDWARE, la persona que define un lenguaje ensamblador es libre de especificar el formato de la línea, el largo màximo de los identificadores, el formato para las constantes y las directivas del ensamblador. Conceptualmente puede haber docenas de diferentes lenguajes ensamblador para una sola màquina pero en la prâctica solo hay pocos -el del fabricante y a lo mejor una o dos versiones màs.

Existen algunas ventajas al usar lenguajes ensamblador :

1. Es mnemònico, es decir, SUMA es mucho màs fâcil de recordar que su equivalente en còdigo binario, 00010000.
2. Las direcciones son simbòlicas, es decir, DATO1 en lugar de un número binario que indica la localidad de memoria.
3. Es de fâcil lectura.

Las desventajas del lenguaje ensamblador son que requiere del uso de un ensamblador para su traducciòn y que sigue ligado a una màquina particular.

2.3.1.1 Mnemònicos

Los mnemònicos son símbolos que se utilizan para denotar las instrucciones de màquina. Cada mnemònico corresponde a una sola instrucciòn a nivel màquina.

Los mnemònicos utilizados en el lenguaje ensamblador de COSI se encuentran en el apèndice A junto con su correspondiente instrucciòn de màquina.

2.3.1.2 Pseudo-instrucciones

Para crear un lenguaje ensamblador conveniente, se debe componer de ciertas caracteristicas adicionales. Estas generalmente son llamadas pseudo-instrucciones o directivas del ensamblador y dan al ensamblador instrucciones sobre còmo ensamblar el programa, pudiendo o no generar instrucciones.

El primer tipo de pseudo-instrucciones en nuestro lenguaje ensamblador se utiliza para la definiciòn de datos. Permite definir direcciones simbòlicas como variables del programa y reservar las localidades de memoria respectivas. Es posible indicar el valor inicial de la variable.

PAL : reserva una palabra en memoria para contener un dato con un valor inicial, ya sea en notaciòn punto-fijo o punto-flotante, si es un entero o un real, respectivamente.

RPAL : reserva de una a varias palabras consecutivas de acuerdo al valor que especifique, con èsto se crea espacio para ser usado por las variables del programa.

El segundo tipo de pseudo-instrucciones no involucra datos.

INICIO : el operando especifica una direcciòn a partir de la cual se comenzarà a almacenar el programa en memoria y contendrà la primera instrucciòn ejecutable.

FIN : esta instrucción indica que se ha alcanzado el final del programa.

Las pseudo-instrucciones que almacenan constantes o reservan memoria no deben usarse a la mitad de una secuencia de instrucciones ejecutables. Por ejemplo, considere el siguiente fragmento de un programa :

```
     CARGAA  SUM
SUM  RPAL    1
     SUMA    SIG
```

Después de ejecutar la instrucción CARGAA, la computadora tratará de ejecutar la siguiente instrucción, la cual no es una instrucción sino es el dato SUM, creando una operación impredecible.

Las pseudo-instrucciones del lenguaje ensamblador de COSI se pueden cosultar en el apêndice A.

2.3.1.3 Formato del lenguaje

Los programas en lenguaje ensamblador están usualmente orientados por línea, es decir, que cada declaración en lenguaje ensamblador está contenida en una sola línea con un formato preestablecido.

En este lenguaje ensamblador cada línea contiene cuatro campos ordenados de la siguiente manera :

ETIQUETA CODIGO DE OPERACION OPERANDO COMENTARIO

El campo etiqueta es opcional. Una etiqueta es simplemente un identificador (o símbolo); èsto es, una secuencia de letras y dígitos comenzando con una letra.

El largo máximo de un símbolo varía con los diferentes ensambladores, a veces permiten símbolos de un largo arbitrario, pero sólo reconocen los primeros seis u ocho caracteres. En nuestro caso el largo máximo será de cinco caracteres.

A todo símbolo en un programa en lenguaje ensamblador se le asigna un valor en el momento de su definición, que es la localidad de memoria donde será almacenado. Un símbolo sólo se debe definir una vez, pero puede ser referido las veces que se desee.

El programa ensamblador no pierde de vista las etiquetas y sus valores gracias a una tabla de símbolos interna.

El campo código de operación contiene el mnemónico, ya sea de una instrucción de máquina o de una pseudo-instrucción. Dependiendo del contenido de este campo, el campo operando especifica cero o un operando.

Un operando es una expresión que consiste de un símbolo o de una constante.

El campo comentario es ignorado por el ensamblador, pero es esencial para una buena programación. Este campo debe contener una descripción del algoritmo y de las estructuras de datos usadas en el programa.

Muchos de los primeros ensambladores usaron un formato fijo en la línea de entrada, en el cual cada campo ocupaba una posición fija. Debido a que no es muy conveniente, la mayoría de los ensambladores actuales permiten un formato variable.

Nuestro lenguaje ensamblador permite un formato variable sencillo con las siguientes reestricciones :

- los campos se separan por uno o más espacios,
- una etiqueta, si está presente, comienza en la primera columna de la línea,
- la ausencia de una etiqueta se indica por la presencia de un caracter blanco en la primera columna de la línea,
- se asume que el campo comentario comienza con el siguiente espacio después del operando, si es que existe, o el de la instrucción, y
- la línea que comience con un símbolo de pesos ($) se considera como línea de comentario.

2.3.1.4 Subrutinas

Las subrutinas que se manejan no contemplan el uso de parámetros (no se cuenta con las instrucciones necesarias), se comunican con el programa principal vía las variables globales declaradas al principio del programa.

Las subrutinas se deben declarar al final del programa principal, en caso contrario, podrían haber errores durante la ejecución del programa.

Aún cuando en nuestro caso no se contempló el uso de subrutinas con parámetros, mencionaremos una técnica para el paso de parámetros. La cual puede ser implementada, en caso de que se desee su uso.

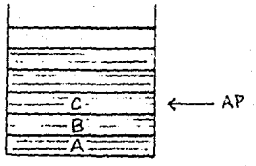
En esta técnica los parámetros son almacenados en la pila del sistema. Por ejemplo, si se tiene la declaración de procedimiento

```
SUB (dat1, dat2, dat3)
```

donde dat1, dat2 y dat3 son los parámetros, entonces, la llamada a la subrutina con parámetros A, B y C, se verá de la siguiente manera :

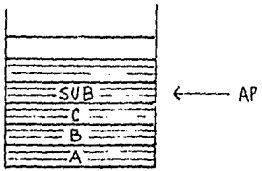
```
push  A          }
push  B          }  se almacenan en la pila
push  C          }
saltsub  SUB
pop              }
pop              }  se eliminan de la pila
pop              }
```

Push X se encarga de meter el parámetro X en la pila, entonces la pila se verá como sigue :



Se debe recordar que en COSI una palabra está formada por 3 bytes, por lo tanto, cada parámetro ocupará 3 bytes.

Cuando se llega a la instrucción saltsub SUB, el AP apuntará al nuevo tope de la pila, donde se guardan los parámetros y el valor del CP y el CP señalará a la primera instrucción de la subrutina :



Al mismo tiempo el valor de AP se almacena en algún lugar, llamémoslo base (puede ser el registro de índice), el cual nos servirá para hacer referencia a los parámetros, es decir, las localidades de éstos serán :

(A) - base + (3 * t)
(B) - base + (2 * t)
(C) - base + (1 * t)

donde t es el tamaño del dato que en este caso es de 3 bytes.

Al regreso de la subrutina se deben eliminar los parámetros de la pila, para ésto se utiliza Pop. En síntesis, para hacer uso de subrutinas con parámetros se deben incluir las instrucciones PUSH X y POP.

2.3.2 EL ENSAMBLADOR

Los ensambladores fueron escritos para automatizar la traducción del lenguaje ensamblador al lenguaje de máquina.

Un ensamblador es un programa que acepta como entrada un programa en lenguaje ensamblador (programa fuente) y cuya salida es una cadena de bits formando un programa en lenguaje de máquina (programa objeto) equivalente al programa en lenguaje ensamblador (fig. 18).

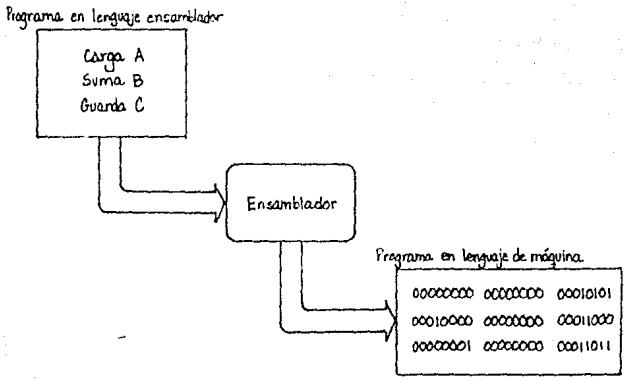


Fig. 18 El ensamblador es un programa que acepta como entrada un programa en lenguaje ensamblador y cuya salida es una cadena de bits.

En el programa objeto las direcciones son relativas con respecto al comienzo de èste. Esto es, el programa se comienza a almacenar en la localidad cero o en la localidad indicada en la pseudo-instrucciòn INICIO, continuando en las siguientes localidades.

El ensamblador puede colocar directamente el programa generado en las localidades de memoria indicadas y transferir el control al procesador para comenzar su ejecuciòn, ensamblador de carga y ejecuta (fig. 19), o almacenarlo en un archivo para que posteriormente un programa llamado cargador lo coloque en la memoria de la computadora para su ejecuciòn (fig. 20).

El ensamblador diseñado para ser soportado por COSI es del primer tipo, o sea, de carga y ejecuta.

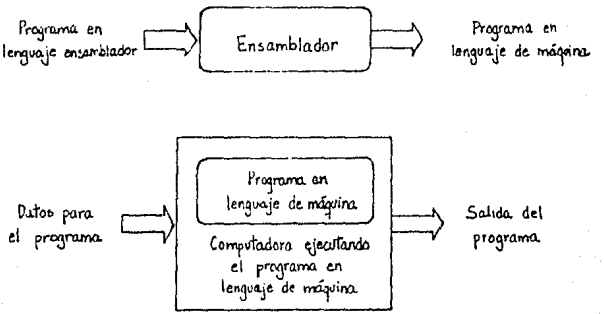


Fig. 19 Ensamblador de carga y ejecuta.

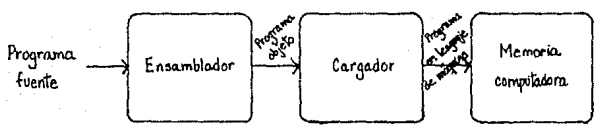


Fig. 20 El cargador se encarga de almacenar el programa en la memoria de la computadora para su ejecución.

Existen varios tipos de ensambladores: el ensamblador de un paso, de un paso y medio, de dos pasos y de múltiples pasos. Nuestro estudio se enfocará al ensamblador de dos pasos.

2.3.2.1 Ensamblador de dos pasos

Se llama ensamblador de dos pasos porque lee el archivo fuente dos veces para resolver las referencias adelantadas (símbolos referenciados antes de ser definidos).

En el primer paso, cada línea se recorre y las definiciones de símbolos que se encuentran se utilizan para construir la tabla de símbolos.

En el segundo paso, las líneas se recorren nuevamente y esta vez, las instrucciones en lenguaje de máquina y los valores de los datos son generados y colocados en el programa objeto, ya que al final del primer paso, todos los símbolos se encuentran definidos en la tabla de símbolos.

Antes de continuar el estudio de cómo trabaja, veremos dos estructuras internas fundamentales para el funcionamiento de un ensamblador, el contador de localidad y la tabla de símbolos.

Contador de localidad (CL)

El ensamblador automáticamente numera las líneas del archivo fuente para facilitar las referencias y correcciones. Además, para no olvidar dónde serán cargadas las instrucciones y datos ensamblados, el ensamblador utiliza una variable interna llamada el contador de localidad de memoria (CLM) o simplemente, contador de localidad (CL).

El CL se inicializa con cero o por medio de la pseudo-instrucción INICIO y se actualiza después de que cada línea del código es procesada, normalmente sumando el largo de la instrucción o del dato que acaba de ser ensamblado.

Durante el ensamble de las instrucciones, los valores que toma el CL corresponden a los que el contador de programa (CP) irá teniendo cuando el programa se esté ejecutando.

Sin embargo, el CF no siempre toma los mismos valores que el CL. Por ejemplo, durante el ensamble de datos, el CL toma valores que el CF nunca tomaría a menos que el programa trate de ejecutar sus datos como instrucciones !!.

Tabla de símbolos

El corazón de un ensamblador es su **tabla de símbolos**, ya que por medio de esta estructura es como el ensamblador recuerda los símbolos definidos por el usuario, junto con otra información, como su tipo y valor.

Para construir la tabla de símbolos, el ensamblador debe poder actualizar apropiadamente el CL después de leer cada línea del programa fuente, puesto que el valor del símbolo que está siendo definido, está dado por el valor que tiene en ese momento el CL.

Ahora consideremos cómo el ensamblador hace su trabajo. El ensamblador trabaja en dos fases o pasos. Cada paso consiste en leer el programa, instrucción por instrucción. En la primera pasada sólo tiene que definir los símbolos; en la segunda, debe generar las instrucciones y direcciones en código binario.

Si una línea fuente contiene una instrucción de máquina, el ensamblador debe determinar el largo de la instrucción, aunque el valor de sus operandos no sea aún conocido.

Afortunadamente, en la mayoría de los lenguajes ensambladores, la asignación de localidades a instrucciones es muy fácil, ya que casi todas las instrucciones usan un número determinado de palabras. En nuestro lenguaje, una palabra por instrucción es la regla.

Sin embargo, si una línea fuente contiene una pseudo-instrucción que afecta el CL, el ensamblador debe determinar el efecto que causa sobre éste desde la primera vez que se encuentra con ella, ya que puede representar más de una palabra.

Por ejemplo, la instrucción RFAL i representa i palabras de memoria. Si a RFAL i se le asigna la localidad j, entonces la siguiente instrucción se le asigna la localidad j+(i*3), ya que cada palabra ocupa tres localidades. Algunas pseudo-instrucciones como FIN no generan instrucciones.

La tarea del ensamblador se puede agrupar en dos pasos o recorridos de la siguiente manera :

Paso 1. Propósito - definir símbolos.

1. Determinar el largo de las instrucciones de màquina.
2. No perder de vista el contador de localidad de programa.
3. Recordar los valores de los símbolos hasta el paso 2
4. Procesar algunas pseudo-instrucciones.

Paso 2. Propósito - generar el programa objeto.

1. Buscar los valores de los símbolos.
2. Generar las instrucciones.
3. Generar los datos.
4. Procesar pseudo-instrucciones.

En el apèndice C se encuentra el diagrama de flujo del ensamblador de dos pasos.

2.3.2.2 Errores detectados por un ensamblador

Los errores màs comúnes encontrados en los programas en lenguaje ensamblador son los siguientes :

- Múltiple definición de símbolo.
  En el paso 1, un símbolo ya definido en la tabla de símbolos se vuelve a definir.

- Código de operación ilegal.
  En el paso 1 ó 2, se encontró un mnemónico no reconocible.

- Símbolo no definido.
  En el paso 2, el campo operando contiene un símbolo que no está definido.

- Error de dirección.
  En el paso 2, una instrucción especifica una dirección inaccesible.

- Error de sintaxis.
  En el paso 1 ó 2, se encontró un caracter o formato ilegal.

Por supuesto, el ensamblador sólo detecta errores en el formato del lenguaje; el mensaje "NINGUN ERROR DETECTADO" no garantiza que el programa se ejecutará apropiadamente !!.

2.3.2.3 Literales

Sea un operando i donde i es una constante que se refiere a una localidad en memoria, la cual será creada por el ensamblador e inicializada con el valor i. Esto es :

```
      CARGAA  i
```

es lo mismo que:

```
      CARGAA  X
   X  PAL     i
```

La única diferencia es que no se necesita escribir la instrucción PAL ni mencionar su nombre.

Consideremos las modificaciones necesarias para que el ensamblador simple maneje literales. En el paso 1, conocemos cuántas localidades de memoria requiere el programa, con exclusión de las literales. Las localidades inmediatamente después de la última usada por el programa y las variables se utilizarán para contener los valores de las literales.

Se necesita otra estructura asociada llamada la tabla de literales, la cual asocia la literal con la localidad donde será almacenada.

Durante el paso 2, si la literal i se encuentra, i se convierte a una palabra representada por i (es decir, si i es un entero, entonces se convierte a su equivalente en punto-fijo).

Además se verifica que la palabra esté en la tabla de literales. Si es así, se reemplaza i con la localidad asociada. Si no, se agrega a la tabla de literales y se le asigna la localidad disponible. El contador se incrementa para indicar la siguiente localidad disponible.

Por ejemplo, considere el siguiente programa en lenguaje ensamblador :

```
        CARGAA  1
        SUMA    2
        MULTI   2
        GUARDA  TOTAL
        REGSIST
TOTAL   PAL     0
        FIN
```

En el paso 1, se encuentra que TOTAL representa la localidad 15 (se almacenará el programa a partir de la localidad cero) y que la 18 es la primera localidad disponible después del programa.

En el paso 2, cuando se encuentra la instrucción CARGAA 1 que contiene una literal cuyo valor es 00000000 00000000 00000001 y, como aún no se encuentra en la tabla de literales, se guarda, asignándole la localidad 18 y la instrucción se traduce a 00000000 00000000 00010010.

La segunda instrucción, SUMA 2, también contiene una literal 00000000 00000000 00000010, la cual como no se encuentra en la tabla de literales, se guarda junto con el valor 21, que es la localidad que le corresponde, y la traducción queda 00010000 00000000 00010101.

Para la tercera instrucción la literal que está presente es la misma que de la segunda 00000000 00000000 00000010. Como ésta ya se encuentra en la tabla de literales y cuyo valor es 21, la instrucción que resulta es 00010010 00000000 00010101. La traducción del resto del programa se lleva a cabo sin mayor problema.

El programa objeto es entonces:

```
 0        00000000 00000000 00010010
 3        00010000 00000000 00010101
 6        00010010 00000000 00010101
 9        00000100 00000000 00000111
12        01000000 00000000 00000000
15        00000000 00000000 00000000
18        00000000 00000000 00000001
21        00000000 00000000 00000010
```

En el apèndice C se encuentra el diagrama de flujo del ensamblador de dos pasos modificado para utilizar literales.

2.3.2.4 Tiempo de ensamble vs. Tiempo de ejecución

Es de suma importancia el enfatizar la diferencia entre las acciones llevadas a cabo en tiempo de ensamble y las realizadas en tiempo de ejecución. Si ésto se puede distinguir claramente desde el principio, entonces, muchos de los problemas conceptuales se pueden evitar.

Las operaciones en tiempo de ensamble son todas aquellas ejecutadas por el ensamblador, sòlo una vez, cuando el programa en lenguaje ensamblador es traducido. Esto es, la traducción se hace en tiempo de ensamble, que es cuando el ensamblador tiene el control.

Las operaciones en tiempo de ejecución son aquellas que las lleva a cabo el procesador al ejecutar las instrucciones de màquina, y ésto sucede en el llamado tiempo de ejecución.

2.4 LENGUAJE DE ALTO NIVEL Y EL COMPILADOR

Como se vió, la dificultad para los humanos de leer y reconocer cadenas de bits (lenguaje de máquina), fue la causa del surgimiento del lenguaje ensamblador y por lo tanto, del ensamblador para la traducción de éste a las cadenas de bits.

Pero aún cuando el lenguaje ensamblador es un vehículo útil para explicar los detalles del HARDWARE de una computadora (es a menudo utilizado para escribir sistemas operativos y otros sistemas de SOFTWARE), no es muy conveniente para la escritura de programas de aplicación entendibles y de fácil mantenimiento debido a su dependencia a la máquina.

2.4.1 LENGUAJE DE ALTO NIVEL

Una computadora necesita tres instrucciones a nivel máquina para sumar dos números, es la forma en como la computadora trabaja. Los seres humanos no tienen porque pensar como las máquinas.

¿ Por què no permitir al programador indicar la suma y asumir las otras instrucciones ?. Por ejemplo, una manera de ver la suma es como una expresión algebraica :

C = A + B

¿ Por què no permitir al programador escribir proposiciones en una forma similar a las expresiones algebraicas, leer estas proposiciones de un programa fuente y generar el código necesario a nivel máquina? (fig. 21). Como respuesta a esta pregunta surgen los lenguajes de alto nivel.

Entre los lenguajes de alto nivel se encuentran FORTRAN, COBOL, Pascal, etc.

Dentro de las ventajas de los lenguajes de alto nivel sobre los lenguajes de bajo nivel, como son el de máquina y el ensamblador, se incluyen las siguientes :

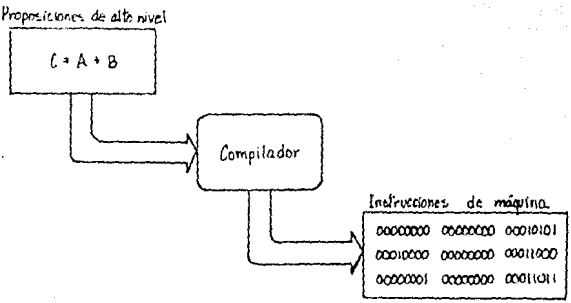


Fig. 21 El compilador es un programa que traduce las proposiciones de alto nivel a instrucciones de bajo nivel (ensamblador o binario).

1. Los lenguajes de alto nivel son más fàciles de aprender, ya que requieren o muy poco o nada de conocimientos sobre el HARDWARE de la computadora.
2. El programador no tiene que preocuparse por tareas que involucran referencias numèricas o simbòlicas a instrucciones, localidades de memoria, constantes, etc.
3. Un programador no necesita conocer la forma de convertir los datos de la representaciòn externa a la interna y viceversa.
4. Los lenguajes de alto nivel ofrecen una variedad de estructuras de control que no estàn disponibles en los lenguajes de bajo nivel. Varias de las construcciones son :
   Proposiciones de condiciòn (IF-THEN-ELSE, CASE)
   Proposiciones de repeticiòn (WHILE, FOR)
   Anidaciòn de proposiciones
   Estas estructuras de control proporcionan un estilo de programaciòn y facilitan la programaciòn estructurada, resultando programas màs fàciles de leer, de entender y de modificar, con un costo reducido de programaciòn.
5. Los programas escritos en un lenguaje de alto nivel son usualmente màs fàciles de depurar que sus equivalentes en lenguaje de màquina y ensamblador.

6. Debido a la presencia de ciertas caracterìsticas en los lenguajes, tales como los procedimientos, los lenguajes de alto nivel permiten una descripciòn modular y jeràrquica de las tareas.
7. Finalmente, los lenguajes de alto nivel son relativamente independientes de la màquina donde se ejecutan. Consecuentemente, los programas son portables.

2.4.1.1 Definiciòn del lenguaje

Los lenguajes de programaciòn deben ser definidos con precisiòn. La especificaciòn propia de un lenguaje de programaciòn involucra la definiciòn de lo siguiente :

1. El conjunto de simbolos (o alfabeto) que puede ser usado para construir programas correctos.
2. El conjunto de reglas o fòrmulas, las cuales definen el conjunto de proposiciones para formar los programas sintàcticamente correctos (sintaxis).
3. El significado de cada proposiciòn (semàntica).

Dado un alfabeto A cualquiera, se pueden intercambiar sus simbolos para construir proposiciones. Es obvio que el nùmero de proposiciones que pueden ser construido es infinito.

Un lenguaje L sobre un alfabeto A es un conjunto de cadenas del alfabeto A que a su vez es un subconjunto del conjunto de todas las posibles cadenas que pueden construirse con A.

Una gramàtica G de un lenguaje L dado es un conjunto finito de reglas o producciones que describe todas las proposiciones del lenguaje L.

Una de las notaciones que se utiliza para describir lenguajes asì como sus propiedades es la notaciòn BNF (Backus Naur Form); la cual es una especie de taquigrafia usada en las producciones.

Esta notaciòn tiene un nùmero significativo de ventajas como mètodo para especificar la sintaxis de un lenguaje :

Una gramàtica brinda una descripciòn sintàctica precisa y fàcil de entender para los programas de un lenguaje particular.
- De una gramàtica diseñada apropiadamente es posible construir automàticamente un analizador sintàctico eficiente. Ciertos procesos de construcciòn de estos analizadores pueden revelar ambigüedades sintàcticas y otras dificultades que podrían permanecer ocultas en la etapa inicial de diseño de un lenguaje y su compilador.
- Una gramàtica brinda una estructura a los programas, que es ùtil para su traducciòn a còdigo objeto y para la detecciòn de errores.

Una gramàtica contiene cuatro tipos de elementos: terminales, no-terminales, símbolo inicial y producciones o reglas.

Los símbolos de los cuales se componen las proposiciones de un lenguaje son llamados terminales. La palabra "token" es sinònimo de "terminal" cuando hablamos de lenguajes de programaciòn.

Los no-terminales son símbolos especiales que denotan parte de la oraciòn. Los tèrminos "variable sintàctica" y "categoría sintàctica" son sinònimos de "no-terminal".

Un no-terminal se selecciona como símbolo inicial y de èste se desprende el lenguaje que se està construyendo, en nuestro caso el no-terminal "programa" es el símbolo inicial, ya que de èl se inicia el reconocimiento.

Las producciones o reglas definen el orden en el cual las categorías sintàcticas se construyen hasta formar las cadenas de terminales. Cada producciòn consiste de un no-terminal seguido del signo ::= que a su vez es seguido de una serie de no-terminales y/o terminales.

Para nuestro objetivo se ha diseñado un pequeño lenguaje de programaciòn a partir de Pascal; consta de lo màs esencial. Se escogiò Pascal por ser un lenguaje utilizado en la Facultad de Ciencias con fines didàcticos, evitando al usuario problemas por tratar con un lenguaje desconocido. Ademàs, Pascal es un lenguaje estructurado y fàcil de comprender, lo cual tratamos de heredar.

Con objeto de facilitar la representaciòn de nuestro lenguaje se utiliza la gràfica de sintaxis que es una derivaciòn de la notaciòn BNF.

En el apèndice D se define la gramàtica, con el simbolo inicial <programa>.

Este lenguaje reducido de Pascal tiene dos instrucciones que permiten a un programa leer y escribir datos, READ m y WRITE m, respectivamente; donde m es el nombre de la variable, cuyo valor serà el que se va a leer o escribir segùn sea el caso y, el cual debe haber sido declarado con su tipo, entero o real, al principio del programa.

2.4.2 EL COMPILADOR

Un traductor es un programa que acepta como entrada un programa escrito en un lenguaje de programaciòn (programa fuente) y produce como salida un programa en otro lenguaje (programa objeto).

Si el lenguaje fuente es un lenguaje de alto nivel como FORTRAN, Pascal o el definido anteriormente, y el programa objeto està en un lenguaje de bajo nivel como un lenguaje ensamblador o de màquina, entonces tal traductor recibe tradicionalmente el nombre de compilador.

El tiempo en el cual un programa fuente es traducido a un programa objeto es llamado tiempo de compilaciòn. La fig. 22 ilustra el proceso de compilaciòn. Se debe notar que el programa fuente y los datos son procesados en diferentes tiempos.

Alguna vez se consideró que los compiladores eran programas casi imposibles de desarrollar. Sin embargo, hoy en dìa èstos son diseñados y realizados con un esfuerzo mucho menor.

Los desarrollos màs importantes que han permitido la realizaciòn de compiladores relativamente sencillos son :

- El entender cómo organizar y separar por módulos el proceso de compilación.

- El descubrimiento de técnicas sistemáticas para el manejo de muchas tareas importantes que ocurren durante la compilación.

- El desarrollo de herramientas de SOFTWARE que facilitan la realización de los compiladores.

Lo anterior es lo que se debe considerar al momento de diseñar un compilador para un lenguaje determinado.

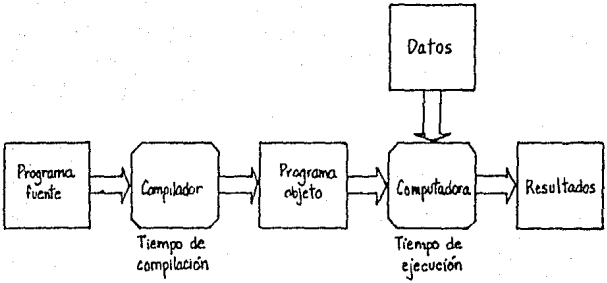


Fig. 22 El proceso de compilación.

2.4.2.1 Estructura de un compilador

La tarea de construir un compilador para un lenguaje particular es compleja. La complejidad o naturalidad de un proceso de compilación depende, en gran medida, del lenguaje fuente.

El proceso de compilación es tan complejo que no es razonable desde el punto de vista lógico ni de implementación, el considerar que este proceso se pudiera realizar mediante un solo paso.

Por esta razón, es necesario dividir el proceso de compilación en series de subprocesos llamados etapas de compilación. Actualmente estas etapas se encuentran bien definidas :

1. Análisis léxico
2. Análisis sintáctico
3. Análisis semántico
4. Generación de código
5. Optimización

El modelo básico de un compilador se puede ver en la fig. 23.

Las dos primeras y la última de estas etapas se llevan a cabo utilizando técnicas bien conocidas. El análisis semántico y la generación de código se realizan mediante técnicas fusionadas en una sola.

El analizador léxico tiene por objeto examinar el texto fuente y llevar a cabo la reconstrucción del mismo en una terminología de representación interna.

Las dos funciones principales del analizador léxico pueden agruparse en : reconstrucción de líneas y representación interna de los átomos del lenguaje.

La reconstrucción de líneas ataca problemas relativamente simples como : eliminar comentarios, identificar los separadores de instrucciones, eliminar espacios superfluos, etc.

Los átomos ("tokens") de un lenguaje son los símbolos cuyo significado o función están definidos en las reglas sintácticas del lenguaje. La segunda tarea del analizador léxico es reconocer tales átomos y traducirlos a una representación interna que facilite su manejo en el resto de las etapas.

El analizador léxico a pesar de ser un proceso simple suele consumir un gran porcentaje del tiempo total de compilación. La razón de este fenómeno es que durante el proceso se examinan cada uno de los símbolos del texto fuente.

La tarea del analizador sintáctico es mucho más compleja, su función es la de verificar que cada una de las proposiciones del programa obedezca las reglas gramaticales del lenguaje fuente, es decir, asegura que las proposiciones del programa pertenezcan al lenguaje.

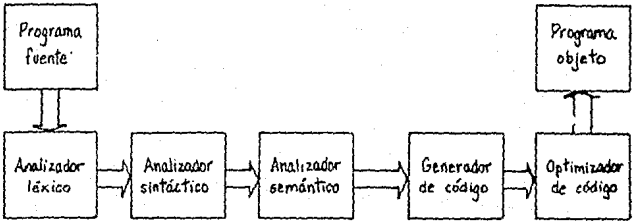


Fig. 23 Modelo básico de un compilador.

La función del analizador semántico es determinar el significado del programa fuente. Se asegura que el significado de las proposiciones del programa fuente sea congruente con las reglas semánticas del lenguaje.

En la etapa de generación de código, como su nombre lo indica, se produce el código objeto que realizará en tiempo de ejecución, las acciones establecidas en el texto fuente.

El código generado en esta etapa puede ser directamente el lenguaje de máquina o un código intermedio que será posteriormente traducido al lenguaje de máquina.

En muchos casos, el compilador efectúa una revisión del código generado con objeto de optimizarlo, para una ejecución más eficiente y menos uso de espacio; ésto se realiza en la etapa de optimización, la cual no será tratada en este trabajo, ya que los programas serán sencillos y pequeños.

2.4.2.2 Análisis léxico

El analizador léxico representa una interface entre el programa fuente y el analizador sintáctico. El analizador léxico examina caracter por caracter del texto fuente y lo separa en átomos, los cuales representan los nombres de las variables, etiquetas, etc.

El analizador léxico usualmente interactúa con el analizador sintáctico en una de dos formas. El analizador léxico puede procesar el programa fuente en un paso separado, antes del análisis sintáctico o en la segunda forma, que involucra una interacción entre los dos analizadores (el analizador léxico es llamado por el analizador sintáctico si se requiere el siguiente átomo del programa fuente).

Este último enfoque es el método preferido, ya que no se necesita construir y almacenar en memoria una forma interna de todo el programa fuente antes de comenzar el análisis sintáctico.

2.4.2.3 Anàlisis sintàctico

El reconocimiento sintàctico es el proceso para determinar si una cadena de simbolos terminales corresponde a una proposiciòn del lenguaje.

El objetivo principal de un analizador sintàctico es el de verificar que cada una de las proposiciones del programa fuente obedezca las reglas gramaticales del lenguaje fuente.

Existen dos mètodos para realizar este anàlisis :

- Desde arriba (Top-Down)
- Desde abajo (Bottom-Up)

El problema a resolver es determinar si una cadena de simbolos terminales puede generarse con las producciones dadas.

Los mètodos desde abajo hacen uso de relaciones lògicas entre simbolos de la gramàtica y consisten bàsicamente en encontrar la raiz de la forma oracional y reducir dicha raiz al no-terminal A usando una regla de la forma :

$$A \Rightarrow a$$

Por lo tanto la labor serà encontrar la raiz y despuès el simbolo no-terminal al que debe reducirse.

Generalmente los mètodos desde abajo consisten en lo siguiente :

1. Asignar la cadena dato a la cadena de comparaciòn x.
2. Escoger las producciones en algùn orden y comparar los consecuentes de cada producciòn con subcadenas de x.
3. Si no existe ninguna subcadena de x igual a alguno de los consecuentes, es necesario repetir el proceso a partir del paso 1.

4. Si existe una subcadena de x igual a algún consecuente, substituir la subcadena por el antecedente de la producción encontrada.
   Si la cadena resultante de la substitución consiste únicamente del símbolo distinguido, el proceso ha terminado con èxito, en caso contrario se repiten los pasos 2,3 y 4.

En los mètodos de reconocimiento sintàctico desde arriba se intenta producir la proposición que se desea reconocer partiendo del símbolo inicial de la gramàtica. Así pues, el punto de partida es el conjunto de reglas cuyo antecedente común es el símbolo inicial.

En tèrminos generales, la aplicación de un análisis desde arriba procede de la siguiente forma :

1. Partiendo del símbolo inicial se aplican las reglas en algún orden, hasta obtener una cadena generada consistente sólo de símbolos terminales.

2. Se compara la cadena generada con la cadena dato, si son iguales se ha terminado el proceso con èxito.

3. Si la comparación falla, se procede a repetir el primer paso con objeto de generar otra de las proposiciones del lenguaje y compararla con la cadena dato.

Existe un problema con los analizadores sintàcticos desde arriba. Este es la llamada recursividad por la izquierda. Una gramàtica G se dice que es recursiva por la izquierda si tiene un no-terminal A tal que hay una derivación A => Aa para alguna a.

Con una gramàtica con recursividad por la izquierda, el analizador sintàctico puede entrar a un ciclo infinito. Esto es, cuando se trata de expander A, se puede eventualmente encontrarse tratando otra vez de expander A sin haber consumido alguna entrada.

Por lo tanto, para hacer uso de analizadores sintàcticos desde arriba se debe eliminar la recursividad por la izquierda de la gramàtica.

Dentro de esta técnica existe el método llamado descenso recursivo. Como su nombre lo indica es altamente recursivo.

En el método descenso recursivo, una secuencia de aplicaciones de producciones se lleva a cabo por medio de una serie de llamadas a funciones. En particular, las funciones son escritas para cada no-terminal. cada función regresa un valor de verdadero o falso dependiendo de si reconoce o no una subcadena.

Un analizador recursivo descendente contiene una función recursiva por cada símbolo no-terminal. Cada función especifica donde comienza la búsqueda de la frase para su no-terminal asociado; se compara la cadena de entrada comenzando en un punto específico con las alternativas de su no-terminal asociado e invocando otras funciones para reconocer otras submetas cuando se requieran.

Sea un lenguaje definido con la siguiente gramática :

```
A  ::=  a B
B  ::=  b
```

el lenguaje generado de A consiste de una sola expresión :

```
a b
```

Aplicando el método descenso recursivo, donde se tiene una función por cada símbolo no-terminal, se obtiene el siguiente analizador :

```
Program analizador (input,output);
 Var  c : char;
  Procedure A;
   Procedure B;
    begin
     if c <> 'b' then
      error;
    end;
   begin
    if c = 'a' then
     begin
      read (c);  B;
     end
    else error;
   end;
  Begin
   read (c);  A;
  End.
```

Nuestro compilador realizarà un anàlisis sintàctico del tipo descenso recursivo de la tècnica desde arriba dada la gran flexibilidad que brinda con respecto a modificaciones en la gramàtica sin alterar el algoritmo y esencia del programa.

Otra de las ventajas es que no requiere de extensas tablas para su funcionamiento, permitiendo de esta forma ahorrar un espacio considerable de memoria. Tambièn se considerò esta tècnica por la sencillez de nuestro lenguaje, ademàs de que la gramàtica de èste no presenta recursividad por la izquierda.

2.4.2.4 Anàlisis semàntico y generaciòn de còdigo

El anàlisis semàntico tiene a su cargo la validaciòn del significado de las proposiciones del programa fuente y en muchos compiladores tambièn se maneja la generaciòn del còdigo objeto.

No se han desarrollado hasta la fecha analizadores semànticos automàticos salvo para lenguajes restringidos que ademàs generan còdigo objeto para màquinas con caracterìsticas muy similares.

La causa de esta carencia de mètodos formales de anàlisis semàntico reside en la dificultad para especificar rigurosamente las reglas semànticas de los lenguajes de programaciòn.

La especificaciòn rigurosa de la semàntica de un lenguaje debe abarcar una gran variedad de aspectos : los tipos de objetos (variables, subrutinas) definidos en el lenguaje; las operaciones vàlidas sobre los diferentes tipos; la ubicaciòn de las instrucciones en la memoria; etc., de ahì la dificultad que existe en la definiciòn de notaciones formales para describir reglas semànticas.

Asì el anàlisis semàntico ataca dos aspectos principales : la verificaciòn del cumplimiento de las reglas semànticas del lenguaje y la generaciòn del còdigo objeto.

El primero abarca aspectos como :

- los objetos del lenguaje y sus tipos (enteros, reales, arreglos),
- las operaciones vàlidas aplicables a los objetos,
- las estructuras de control del lenguaje y,
- el alcance de las declaraciones de los objetos.

El segundo se refiere a :

- la ubicaciòn del còdigo objeto y las àreas de datos,
- los algoritmos de traducciòn de expresiones aritmèticas y,
- la utilizaciòn de los registros de la màquina.

Aùn cuando son de naturaleza distinta los problemas contemplados en el anàlisis semàntico y la generaciòn de còdigo, estàn intimamente ligados; esta es la razòn por la que se tratan en conjunto.

El mètodo que emplearemos asume una estrategia comùn para el procesamiento de las proposiciones del lenguaje fuente. Esta tècnica està basada en la idea de que el tipo de anàlisis semàntico ejecutado y la naturaleza del còdigo producido està especificado (es decir, determinado) por cada una de las producciones de la gramàtica.

Asi como las reglas de producciòn de un lenguaje son aplicadas, el analizador sintàctico puede simplemente invocar la rutina de anàlisis semàntico y generaciòn de còdigo apropiada y el còdigo objeto correcto serà generado en una forma sistemàtica.

Este tipo de compilaciòn se llama **dirigido por la sintaxis**, ya que las reglas de producciòn de la gramàtica son usadas para dirigir el tipo de procesamiento que se ejecuta sobre las proposiciones del lenguaje fuente.

El còdigo generado por el compilador es un lenguaje ensamblador que se adapta al ensamblador construido al principio, para su posterior traducciòn al lenguaje de màquina.

El listado del programa compilador para nuestro lenguaje se puede consultar en el apèndice E.

2.5 SISTEMA OPERATIVO

Aún cuando el objetivo de esta tesis no es el de tratar el Sistema Operativo, se dará una rápida visión sobre el tema ya que sin èste no serviría de nada todo lo anterior. Esto debido a que el Sistema Operativo representa la interfaz entre el HARDWARE con los usuarios y el SOFTWARE.

¿ QUE ES UN SISTEMA OPERATIVO ?

El Sistema Operativo es un programa a través del cual, los usuarios y los programadores utilizan el HARDWARE (fig. 24). Pero el proporcionar una interfaz entre el usuario con el HARDWARE no es su única función. Un Sistema Operativo moderno cataloga los programas para su ejecución, controla el uso de los recursos del sistema de cómputo y proporciona funciones útiles para todos los programas que corren bajo el sistema.

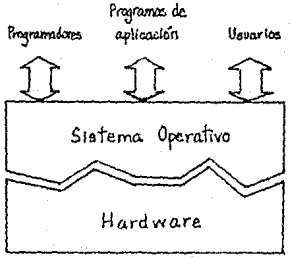


Fig. 24 El Sistema Operativo es una interfaz entre el SOFTWARE y el HARDWARE.

Para que el Sistema Operativo ejecute una de sus funciones, el usuario debe indicarle cuàl y como respuesta, el Sistema Operativo reunirà los recursos necesarios para llevarla a cabo. El mòdulo encargado de aceptar, interpretar y llevar a cabo los comandos es el llamado procesador de comandos.

El procesador de comandos consiste de submòdulos, cada uno de los cuales ejecuta una tarea (fig. 25). Por ejemplo, un submòdulo contiene las instrucciones que guian a la computadora a travès del proceso de copiado de un programa del disco a la memoria principal.

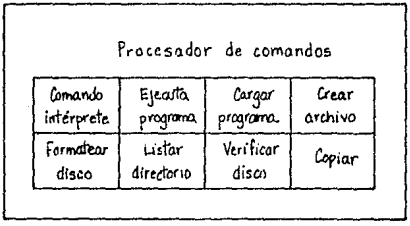


Fig. 25 El procesador de comandos está compuesto de módulos, cada uno de los cuales ejecuta una función específica.

Otro de los mòdulos de que consta el Sistema Operativo es el sistema de control de E/S, cuya función es la de comunicar el equipo periférico; acepta los requerimientos de E/S de los programas de aplicación y genera los comandos primitivos necesarios para controlar fisicamente un dispositivo periférico.

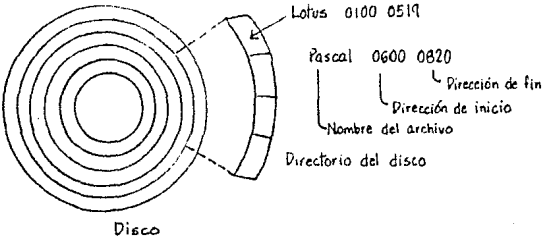


FIG. 26 El directorio del disco es la llave de acceso a los programas y archivos.

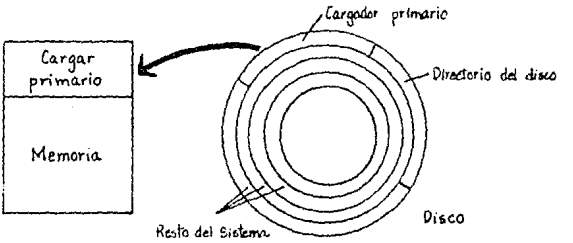
El sistema de archivos es la parte encargada de encontrar la localidad de un archivo a través de búsquedas en el directorio del disco. Usualmente, el directorio del disco se encuentra almacenado en la pista 0, sector 1 y/o 2, (fig. 26).

Cargar y ejecutar un programa comienza con un comando que el Sistema Operativo lee e interpreta. Claramente se ve que el Sistema Operativo debe estar en memoria antes que el comando sea emitido. ¿ Cómo puede el sistema ser cargado antes ?.

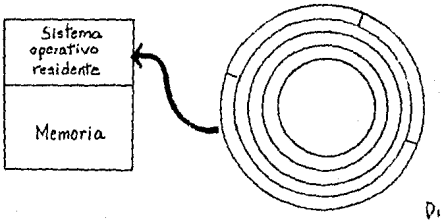
Desafortunadamente, no se puede simplemente teclear un comando y dejar que el Sistema Operativo se cargue a èl mismo, èsto debido a que cuando se prende la computadora, la memoria principal se encuentra vacía y, por lo tanto, no es posible leer, interpretar ni llevar a cabo comandos.

Tipicamente, el Sistema Operativo està almacenado en disco. La idea es copiar èste a la memoria. Esto lo realiza un programa especial llamado cargador primario (boot). Generalmente, el cargador primario està almacenado en el primer sector de un disco y el HARDWARE està diseñado para leerlo automàticamente cada vez que la computadora es prendida.

El cargador primario consiste de unas pocas instrucciones, pero que son suficientes para leer el resto del Sistema Operativo a la memoria (fig. 27).



a) Cuando la computadora es prendida, el HARDWARE automáticamente lee el cargador primario que se encuentra en los primeros sectores del disco.



b) El cargador primario contiene las instrucciones que leen el resto del Sistema Operativo, del disco a la memoria

Fig. 27 Carga del Sistema Operativo.

Aún cuando en este trabajo no se contemplò el desarrollo de un Sistema Operativo, el sistema "SEC" podria representar a uno, aunque incipiente y bastante sencillo.

Esto debido a que representa la interfaz entre COSI, que es la computadora, y el usuario; nos permite hacer uso de la màquina (el HARDWARE) de una forma parecida a la de un Sistema Operativo real. Ademàs se puede considerar que se tiene un procesador de comandos, cuyos submòdulos son nuestros procedimientos que llevan a cabo cada una de las opciones que ofrece el menù principal del sistema "SEC".

Es importante señalar que no es un Sistema Operativo sino que se està haciendo una similitud a èste por las funciones que realiza.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

2.6 INTEGRACION DEL SISTEMA

Hasta el momento se han tratado en forma individual las partes y recursos de que consta la màquina COSI. En esta parte, se harà una recopilaciòn para tener una visiòn general y no perder detalle alguno.

El HARDWARE de la màquina COSI està formado por una memoria, la cual consiste de bytes de 8 bits y donde cada tres bytes consecutivos forman una palabra, la cual se direcciona por medio de la localidad de su primer byte; tiene un total de 2^{9} bytes, es decir, 512 localidades y puede crecer hasta un total de 2^{16} bytes. Se incluyen los registros RDAM y RDIM, y cuyo acceso (lectura y escritura) es en bytes.

El procesador es de un solo acumulador y està formado por los siguientes registros y unidades (se especifican junto con su longitud, nùmero de registro y mnemònico) :

Registro de instrucciòn	(24 bits	, 1	, RI	)
Registro de direcciòn efectiva	(16 bits	, 2	, RDE	)
Contador de programa	(16 bits	, 3	, CP	)
Acumulador	(24 bits	, 4	, A	)
Registro de Indice	(24 bits	, 5	, X	)
Apuntador de pila	(16 bits	, 6	, AP	)
Registro de condiciòn	(3 bits	, 7	, RC	)
Unidad Aritmètica y Lògica	(		UAL	)
Unidad de Control	(		UC	)

El formato de las instrucciones consta de un còdigo de operaciòn y un operando y cuya longitud es de 3 bytes. El bit x indica el modo de direccionamiento, directo o indirecto.

Las instrucciones disponibles se clasifican en 4 grupos

- movimiento de datos
- operaciones aritmèticas y lògicas
- transferencia de control
- Entrada/Salida

Estas se pueden consultar en el apèndice A y forman el lenguaje de màquina.

Los tipos de datos que maneja COSI son enteros y reales, que se almacenan en palabras (3 bytes).

El subsistema de E/S no se trata ampliamente en este trabajo, sin embargo se tienen dos dispositivos periféricos, el teclado (entrada) y la pantalla de video (salida).

Todo este HARDWARE se complementa con el SOFTWARE.

Se tiene un lenguaje ensamblador formado por las instrucciones mnemònicas y las pseudo-instrucciones (apèndice A) cuyo formato consta de 4 campos ordenados de la siguiente forma :

ETIQUETA CODIGO DE OPERACION OPERANDO COMENTARIO

con las siguientes reglas :

- los campos se separan por uno o màs espacios,
- una etiqueta si està presente, comienza en la primera columna de la linea,
- la ausencia de una etiqueta se indica por la presencia de un caracter blanco en la primera columna de la linea,
- se asume que el campo comentario comienza con el siguiente espacio despuès del operando, si es que existe, o el de la instrucción, y
- la linea que comienza con el simbolo de pesos se considera como linea de comentario.

Para la traducción del lenguaje ensamblador al lenguaje de màquina se cuenta con un ensamblador de dos pasos (apèndice C) que es del tipo carga y ejecuta.

Como lenguaje de alto nivel se maneja una versión reducida de Pascal, cuya gramàtica se define en el apèndice D.

El mètodo utilizado para construir el compilador que traduce, en nuestro caso, el lenguaje de alto nivel a nivel ensamblador, es el descenso recursivo, el cual tiene una función por cada simbolo no-terminal. En el apèndice E se lista el programa compilador para el lenguaje utilizado.

Cada una de las partes mencionadas se probaron por separado tomando en cuenta el diseño original. En màs de una ocasiòn se tuvieron que modificar debido a los problemas de implementaciòn que presentaban cada una de ellas. Tambièn se hicieron correcciones durante la unificaciòn de las partes, principalmente para su apropiada presentaciòn al usuario.

Se tratò que la presentaciòn del sistema sea en una forma sencilla pero descriptiva y fàcil de comprender por parte del usuario.

Alto Nivel	Ensamblador
var	
tc : real;	
suma : real;	
limite : real;	
begin	
tc := 0.;	CARGAF 0.
	GUARDA TC
suma := 0.;	CARGAF 0.
	GUARDA SUMA
read limite;	LEEREAL LIMIT
while suma<=limite do	1
	CARGAF SUMA
	COMPF LIMIT
	SALTMA 2
begin	
tc := tc+1.;	CARGAF TC
	SUMAF 1.
	GUARDA TC

oprima cualquier tecla para continuar

Fig. 28

El sistema consta de dos opciones para comenzar : dar el nombre de un archivo en donde el programa està escrito en nuestro lenguaje o el nombre de aquel en lenguaje ensamblador. Para este efecto, el usuario deberà haber creado previamente el programa y guardado en un archivo.

En la primera opción el programa será compilado, ensamblado y ejecutado; en la segunda sólo será ensamblado y ejecutado.

La visualización de la etapa de compilación consta del programa en lenguaje de alto nivel con su correspondiente en lenguaje ensamblador (ver fig. 28).

En la etapa de ensamblado, se observa el código en lenguaje ensamblador y su equivalente a nivel máquina (código binario) con su respectivo contador de localidad (ver fig. 29).

```
     Ensamblador                    Maquina
     CARGAF 0.              0   00000001 00000000 00111111
     GUARDA TC              3   00000011 00000000 00110110
     CARGAF 0.              6   00000001 00000000 00111111
     GUARDA SUMA            9   00000011 00000000 00111001
     LEEREAL LIMIT         12   00110001 00000000 00111100
1
     CARGAF SUMA           15   00000001 00000000 00111001
     COMPF LIMIT           18   00011011 00000000 00111100
     SALTMA 2              21   00100100 00000000 00110000
     CARGAF TC             24   00000001 00000000 00110110
     SUMAF 1.              27   00010100 00000000 01000010
     GUARDA TC             30   00000011 00000000 00110110
     CARGAF 1.             33   00000001 00000000 01000010
     DIVIDEF TC            36   00010111 00000000 00110110
     SUMAF SUMA            39   00010100 00000000 00111001
     GUARDA SUMA           42   00000011 00000000 00111001
     SALTA 1               45   00100000 00000000 00001111
2
```

oprima cualquier tecla para continuar

Fig. 29

Durante la ejecución se ve el funcionamiento interno de la computadora (procesador y memoria) y la instrucción a nivel máquina que está siendo ejecutada en ese momento con su equivalente en lenguaje ensamblador y en su caso, con su equivalente en lenguaje de alto nivel (ver fig. 30).

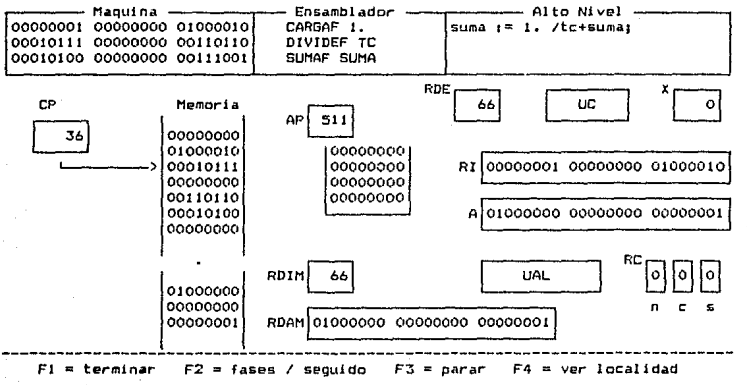


Fig. 30

2.6.1 IMPLEMENTACION TECNICA

Tanto el còdigo en lenguaje de alto nivel como el de ensamblador permanecen en la memoria principal durante toda la ejecuciòn del sistema, guardados en arreglos.

Estos arreglos tienen la caracteristica de tener en cada registro un campo numèrico que guarda la liga hacia el còdigo de màs bajo nivel. Esto para presentar las instrucciones equivalentes de los distintos niveles.

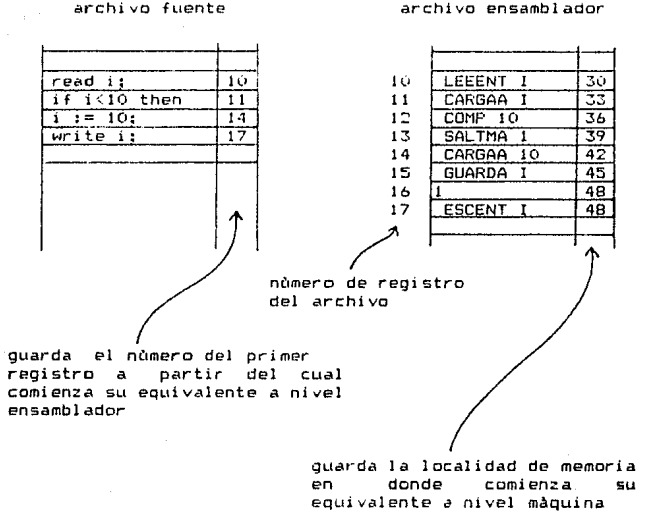
Por ejemplo, sea una parte de un programa en lenguaje de alto nivel :

```
read i;
if i<10 then
 i := 10;
write i;
```

su equivalente en lenguaje ensamblador serà, considerando a i como variable entera :

```
 LEEENT I
 CARGAA I
 COMP 10
 SALTMA 1
 CARGAA 10
 GUARDA I
1
 ESCENT I
```

Suponiendo que esta parte del còdigo se encuentra almacenada a partir del registro 10 del archivo que guarda el programa en ensamblador y que el programa en lenguaje de màquina se encuentra almacenado a partir de la localidad 30 de la memoria y, sabiendo que es una instrucciòn a nivel màquina por cada instrucciòn a nivel ensamblador, los arreglos quedan de la siguiente manera :



Con lo anterior podemos establecer las ligas entre los códigos a diferentes niveles.

CAPITULO 3

IMPACTO Y USO DEL SISTEMA "SEC" EN LA ENSEÑANZA DE COMPUTACION

IMPACTO Y USO DEL SISTEMA "SEC" EN LA ENSEÑANZA DE COMPUTACION

Descrita la teoría y las características de nuestra màquina COSI y del ensamblador y compilador soportados en ella, nuestro propòsito ahora es el de mostrar còmo aplicar el sistema "SEC" en la enseñanza de computaciòn y el impacto que causa en èsto.

Se mostrarán los conceptos que abarca el sistema y la forma en que èstos se ejemplifican para una mejor comprensiòn por parte del estudiante.

Un estudiante, muchas veces crea un programa en lenguaje de alto nivel y lo ejecuta, sin saber en realidad todo el proceso que encierra èsto. En ocasiones se le explica pero muchos de los conceptos no son comprendidos del todo ya que èl no los puede ver (además de que el objetivo del curso de Computaciòn I no es el de profundizar en la estructura y funcionamiento interno de las computadoras, sino el de un conocimiento general de èstas), o a veces sòlo sabe que debe compilar el programa antes de ejecutarlo.

Con este sistema se espera que el estudiante comprenda mejor estos conceptos al poderlos ver gràficamente.

Para el proceso de traducciòn se tienen varios conceptos. En nuestro caso, se divide en dos etapas : la compilaciòn y el ensamblado. La compilaciòn es la traducciòn del programa en lenguaje de alto nivel a un programa en lenguaje ensamblador y, el ensamblado es del programa en lenguaje ensamblador a uno en lenguaje de màquina (còdigo binario), llevados a cabo por un compilador y un ensamblador, respectivamente.

En muchas ocasiones, la compilaciòn se considera como la traducciòn del programa en lenguaje de alto nivel a nivel de màquina, sin pasar por el lenguaje ensamblador, pero se prefiriò hacerlo de este modo para presentar lo referente a este tema, además de que es màs conveniente, ya que el lenguaje de màquina no es accesible a primera vista. El porquè de èsto se explicarà màs adelante.

Supòngase que se tiene el siguiente programa que calcula la suma de los primeros n nûmeros :

```
Var
 limite : integer;
 suma   : integer;
 i      : integer;
Begin
 read limite;
 while limite < 0 do
  read limite;
 if limite = 0 then
  suma := 0
 else
  begin
   suma := 0;
   for i := 1 to limite do
    suma := suma + i;
  end;
 write suma;
End.
```

Se debe notar que el programa se pudo escribir en una forma màs conveniente si la variable suma se inicializa al principio, pero se diseñò de esta manera para utilizar las tres instrucciones de control : IF-THEN-ELSE, WHILE-DO y FOR-TO-DO.

Al compilar el programa en "SEC", obtenemos el siguiente listado :

```
Var
limite : integer;
suma   : integer;
i      : integer;
Begin
read limite                          LEEENT LIMIT
while limite < 0 do            1
                                     CARGAA LIMIT
                                     COMP 0
                                     SALTMAIG 2
read limite;                         LEEENT LIMIT
                                     SALTA 1
                               2
if limite = 0 then                   CARGAA LIMIT
                                     COMP 0
                                     SALTDIF 3
suma := 0                            CARGAA 0
                                     GUARDA SUMA
else                                 SALTA 4
```

```
                                    3
begin
suma := 0;                            CARGAA 0
                                      GUARDA SUMA
for i:=1 to limite do                 CARGAA 1
                                    5
                                      GUARDA I
                                      COMP LIMIT
                                      SALTMA 6
suma := suma + i;                     CARGAA SUMA
                                      SUMA I
                                      GUARDA SUMA
                                      CARGAA I
                                      SUMA 1
                                      SALTA 5
                                    6
end;                                4
write suma;                           ESCENT SUMA
End.                                  REGSIST
                                    LIMIT RPAL 1
                                    SUMA RPAL 1
                                    I RPAL 1
                                      FIN
```

el cual presenta el programa fuente con su equivalente en lenguaje ensamblador.

Los nombres de las variables a nivel ensamblador son de hasta cinco caracteres, es por èsto que la variable limite se encuentra truncada a LIMIT, y no se hace distinciòn entre minùsculas y mayùsculas; todo es convertido a mayùsculas.

La estructura de una instrucciòn a nivel ensamblador consta de tres partes : una etiqueta (opcional), un mnemònico o pseudo-instrucciòn (indica la acciòn a ejecutar) y un operando (indica la direcciòn de una localidad de memoria). Por ejemplo, la instrucciòn "LEEENT LIMIT", la cual no presenta etiqueta, indica que se va a leer un entero (LEEENT) llamado LIMIT.

A cada instrucciòn de alto nivel le corresponde una o varias instrucciones a nivel ensamblador. Por ejemplo, a la instrucciòn "read limite" le corresponde una sola, "LEEENT LIMIT", en cambio, la instrucciòn "suma := 0" se traduce en "CARGAA 0" y "GUARDA SUMA".

Lo anterior explica el porque un programa en lenguaje ensamblador requiere de màs instrucciones que el mismo en lenguaje de alto nivel.

Al ensamblar el programa obtenemos lo siguiente :

```
   LEEENT LIMIT          0 00110000 00000000 01001110
1
   CARGAA LIMIT          3 00000000 00000000 01001110
   COMP 0                6 00011010 00000000 01010111
   SALTMAIG 2            9 00100110 00000000 00010010
   LEEENT LIMIT         12 00110000 00000000 01001110
   SALTA 1              15 00100000 00000000 00000011
2
   CARGAA LIMIT         18 00000000 00000000 01001110
   COMP 0               21 00011010 00000000 01010111
   SALTDIF 3            24 00100010 00000000 00100100
   CARGAA 0             27 00000000 00000000 01010111
   GUARDA SUMA          30 00000011 00000000 01010001
   SALTA 4              33 00100000 00000000 01001000
3
   CARGAA 0             36 00000000 00000000 01010111
   GUARDA SUMA          39 00000011 00000000 01010001
   CARGAA 1             42 00000000 00000000 01011010
5
   GUARDA I             45 00000011 00000000 01010100
   COMP LIMIT           48 00011010 00000000 01001110
   SALTMA 6             51 00100100 00000000 01001000
   CARGAA SUMA          54 00000000 00000000 01010001
   SUMA I               57 00010000 00000000 01010100
   GUARDA SUMA          60 00000011 00000000 01010001
   CARGAA I             63 00000000 00000000 01010100
   SUMA 1               66 00010000 00000000 01011010
   SALTA 5              69 00100000 00000000 00101101
6
4
   ESCENT SUMA          72 00110010 00000000 01010001
   REGSIST              75 01000000 00000000 00000000
LIMIT RPAL 1            78 00000000 00000000 00000000
SUMA RPAL 1             81 00000000 00000000 00000000
I RPAL 1                84 00000000 00000000 00000000
   FIN
0                       87 00000000 00000000 00000000
1                       90 00000000 00000000 00000001
```

Contiene el programa en lenguaje ensamblador y su equivalente en binario. El número que precede a las instrucciones a nivel máquina indica la localidad donde se encuentra almacenada (como el tamaño de una palabra en COSI es de tres bytes, el contador se incrementa de tres en tres).

Con èsto se explica el porquè al principio se dijo que un programa en lenguaje de màquina no es fàcil de entender, ùnicamente se observan ceros y unos. Por esta razòn se consideró el estudiar el lenguaje ensamblador debido a que es una representaciòn simbòlica del lenguaje de màquina.

A cada instrucciòn a nivel ensamblador le corresponde una a nivel màquina, excepto a las pseudo-instrucciones, que pueden o no generar instrucciones. Por ejemplo, "RPAL" reserva una palabra de memoria, "FIN" indica el final del programa en ensamblador, pero ninguna de las dos genera instrucciòn alguna.

Por lo tanto, los programas traducidos a còdigo binario, los llamados ejecutables, son mucho màs grandes que los fuente en lenguaje de alto nivel (en cuanto al nùmero de instrucciones, no en cuanto al tamaño del archivo).

La estructura de una instrucciòn a nivel màquina, en nuestro caso, està formada por un còdigo de operaciòn que ocupa un byte, y una direcciòn de memoria dada en dos bytes; ocupando en total una palabra de memoria.

El que la màquina sòlo entienda ceros y unos, hace que la programaciòn a este nivel sea difícil y muy rara en estos tiempos que se tiene gran cantidad de herramientas que facilitan la tarea a los programadores.

Es màs fàcil aprender el mnemònico "RESTA" que la cadena "00010001", o indicar una variable con un nombre como "SUMA" que recordar que corresponde a la direcciòn "00000000 01010001". Y por supuesto es mucho mejor y màs fàcil escribir "suma := 0" que "CARGAA 0" y "GUARDA SUMA", ademàs que hay instrucciones de alto nivel que dan lugar a 5, 6 ó màs instrucciones a nivel ensamblador.

Se muestra el desarrollo que han tenido y los cambios que han sufrido los lenguajes de programaciòn y sus traductores a través del tiempo, siempre con la finalidad de simplificar el trabajo de las personas.

Como ventaja que ofrece un compilador, por ejemplo, el compilador de Pascal, es que ademàs de hacer la traducciòn se encarga de cargarlo en memoria, es decir, el usuario no tiene que preocuparse por las localidades en donde serán alojados tanto el programa como las variables, la direcciòn donde inicia la ejecuciòn, etc.

Sin embargo, si se programa a nivel de màquina, el programador se enfrentaria con el problema de hacer todo èl mismo y, por lo tanto deberà tener cuidado con muchisimas cosas, como el no ocupar àrea ya utilizada o indicar una direcciòn de inicio errònea o equivocar la direcciòn de alguna variable, ya que estos errores causarian un gran problema.

Por ejemplo, si el programa anterior se hubiera programado directamente en lenguaje de màquina y almacenado a partir de la direcciòn 31 de la memoria y las variables a partir de la direcciòn 112, el programa seria :

```
 31  00110000 00000000 01110000
 34  00000000 00000000 01110000
 37  00011010 00000000 01111001
 40  00100110 00000000 00110001
 43  00110000 00000000 01110000
 46  00100000 00000000 00100010
 49  00000000 00000000 01110000
 52  00011010 00000000 01111001
 55  00100010 00000000 01000011
 58  00000000 00000000 01111001
 61  00000011 00000000 01110011
 64  00100000 00000000 01100111
 67  00000000 00000000 01111001
 70  00000011 00000000 01110011
 73  00000000 00000000 01111100
 76  00000011 00000000 01110110
 79  00011010 00000000 01110000
 82  00100100 00000000 01100111
 85  00000000 00000000 01110011
 88  00010000 00000000 01110110
 91  00000011 00000000 01110011
 94  00000000 00000000 01110110
 97  00010000 00000000 01111100
100  00100000 00000000 01001100
103  00110010 00000000 01110011
106  01000000 00000000 00000000
      .
      .
112  00000000 00000000 00000000
115  00000000 00000000 00000000
118  00000000 00000000 00000000
121  00000000 00000000 00000000
124  00000000 00000000 00000001
```

El programador tendria que indicar la direcciòn de la primera instrucciòn a ejecutar, que en este caso es la 31, pero si por error indicara la 32, entonces, la primera instrucciòn que se tratarìa de ejecutar es :

```
00000000 01110000 00000000
```

la cual se traduce como carga el acumulador con el valor almacenado en la dirección 28,672, con lo cual ya se tiene el primer error, quizá ejecutable, dependiendo del tamaño de la memoria (en nuestro caso, la memoria tiene 512 localidades, por lo tanto se marcaría un error), pero sin ser lo que se deseaba.

Si continuamos, todas las instrucciones indicarían la misma operación a ejecutar, cargar el acumulador (00000000) pero con los contenidos de distintas localidades :

```
32  00000000 01110000 00000000
35  00000000 01110000 00011010
38  00000000 01111001 00100110
41  00000000 00110001 00110000
44  00000000 01110000 00100000
    .
    .
68  00000000 01111001 00000011
71  00000000 01110011 00000000
74  00000000 01111100 00000011
    .
    .
```

Otros errores podrían ser que el código de instrucción no correspondiera a ninguna ejecutable, que la dirección del operando no fuera la correcta, que la dirección que se indica se sale del rango de la memoria, etc., y encontrar los errores a este nivel es un trabajo bastante difícil y tedioso.

Es por ésto, que es una ventaja que el usuario no tenga que preocuparse por éstas cosas o programar a este nivel; disminuyendo así, los errores que se puedan cometer.

En nuestro caso, si se programa con lenguaje de alto nivel, el sistema cargará el programa en lenguaje de máquina a partir de la dirección cero de la memoria y las variables después de la última ocupada por éste.

Sin embargo, si se programa a nivel ensamblador, con la pseudo-instrucción "INICIO m" existe la posibilidad de indicar que a partir de la dirección m se almacene el programa en memoria. Además, esta dirección se toma como la localidad de la primera instrucción a ejecutar.

Otra de las cosas que se muestra son las instrucciones de control de alto nivel, como son el IF-THEN-ELSE, el WHILE-DO y el FOR-TO-DO. Cada una de èstas tiene su estructura bien definida, la cual se presenta a continuaciòn :

- IF condiciòn THEN proposiciòn1 ELSE proposiciòn2

  donde la clàusula ELSE puede omitirse.

Implementaciòn :

```
        Prueba condiciòn
        Si no se cumple la condiciòn => salta E1
        Proposiciòn1
        Salta E2
    E1  Proposiciòn2
    E2  ...
                              donde E1 y E2 son etiquetas
```

Omitiendo la clàusula ELSE :

```
        Prueba condiciòn
        Si no se cumple la condiciòn => salta E1
        Proposiciòn1
    E1  ...
```

Ejemplo :

```
 if limite=0 then          CARGAA LIMIT  }   prueba
                           COMF 0        }   condiciòn
                           SALTDIF 3     }   si no se cumple
  suma := 0                CARGAA 0      }   proposiciòn1
                           GUARDA SUMA   }
 else                      SALTA 4       }   salta
  begin                3                }
   suma := 0;              CARGAA 0      }   proposiciòn2
   .                         .           }
   .                         .           }
  end;                  4                }   ...
```

- WHILE condición DO proposición

Implementación :

```
E1 Prueba condición
   Si no se cumple la condición => salta E2
   Proposición
   Salta E1
E2 ...
                          donde E1 y E2 son etiquetas
```

Ejemplo :

```
 while limite<0 do     1                    } prueba
                         CARGAA LIMIT       } condición
                         COMP 0             }
                         SALTMAIG 2         } si no se cumple
  read limite;           LEEENT LIMIT       } proposición
                         SALTA 1            } salta
                       2                    } ...
```

- FOR variable:=expresión1 TO expresión2 DO proposición

Implementación :

```
      Establece limite
   E1 Prueba condición
      Si es mayor => salta E2
      Proposición
      Incrementa la variable en uno
      Salta E1
   E2 ...
                          donde E1 y E2 son etiquetas
```

Ejemplo :

```
 for i:=1 to limite do    CARGAA 1          } establece limite
                         5                  }
                          GUARDA I          } prueba condición
                          COMP LIMIT        }
                          SALTMA 6          } si es mayor
  suma := suma + i;       CARGAA SUMA       }
                          SUMA I            } proposición
                          GUARDA SUMA       }
                          CARGAA I          } incrementa en uno
                          SUMA 1            }
                          SALTA 5           } salta
                         6                  }  ...
```

Con lo ya expuesto, se ejemplifican diferentes conceptos: què son los lenguajes; què es un lenguaje de alto nivel, un lenguaje ensamblador y un lenguaje de màquina, además de sus estructuras, sus diferencias, la importancia de uno y de otro, sus ventajas y desventajas; para què sirven los compiladores y ensambladores; el proceso de traducciòn; los conceptos de bit, byte y palabra; etc.

Una vez traducido el programa a lenguaje de màquina, el siguiente paso es su ejecuciòn. Para èsto se deben conocer las partes de que consta el HARDWARE de la computadora : la memoria, el procesador y la E/S, y su funcionamiento e interacciòn.

El procesador consta de registros y unidades; la memoria està formada por un àrea de almacenamiento y registros, y la E/S por los dispositivos periféricos (teclado para la entrada y pantalla de video para la salida), èsta es la estructura de COSI.

Muchas veces al àrea de almacenamiento se le conoce como memoria, sin incluir los registros. Para comodidad, se maneja la misma convenciòn; los registros se indicarán por medio de sus nombres.

La ejecuciòn de un programa se desglosa como la realizaciòn de cada una de las instrucciones de que consta èste, en el llamado ciclo de control.

El sistema "SEC" muestra el funcionamiento del ciclo de control con cada una de sus etapas. La primera etapa consiste en traer de la memoria la instrucciòn a ejecutar, cuya localidad es indicada por el CP, para èsto la instrucciòn pasa primero por el RDAM para después llegar al RI.

La segunda fase se encarga de determinar la funciòn a ejecutar y la direcciòn real del operando que interviene (RDE), al mismo tiempo se incrementa el CP para señalar la siguiente instrucciòn que se ejecutarà.

Estas dos primeras etapas siempre se llevan a cabo de la misma manera, es decir, su funcionamiento es el mismo, sòlo cambian las direcciones de memoria y los valores de los registros que intervienen; por ejemplo, el valor del CP se va modificando para indicar la direcciòn de la siguiente instrucciòn a ejecutar.

En la tercera y última etapa es donde se lleva a cabo la ejecución de la instrucción : cargar el acumulador, guardar el acumulador en memoria, sumar, restar, comparar, saltar, leer una variable, etc. El funcionamiento varía de acuerdo a la instrucción.

En cada ciclo de control, además de mostrar la instrucción a nivel máquina que está siendo ejecutada, se exhibe su equivalente en ensamblador y en lenguaje de alto nivel.

El acceso a la memoria, como se dijo anteriormente, se realiza por medio de bytes, y no de bit en bit como se podría pensar al ver el sistema funcionando. Esto sólo es para efecto de presentación, dar la noción de transferencia.

Por último se ilustrará la representación interna de los números enteros (punto-fijo) y reales (punto-flotante).

Los números enteros se representan fácilmente a nivel máquina, consiste en obtener su equivalente en binario. Por ejemplo, el 3 es 11 en base dos, el 52 es 110100, pero como se debe de representar en una palabra de computadora (tres bytes), la representación interna del tres será "00000000 00000000 00000011" y del 52, "00000000 00000000 00110100".

En cambio, la representación del número real no es tan sencilla, está formada por el signo de la mantisa, la mantisa, el signo del exponente y el exponente, que en nuestro caso ocupan 1, 15, 1 y 7 bits, respectivamente.

La representación del tres real es "01100000 00000000 00000010" y del 52 real es "01101000 00000000 00000110" (consultar APENDICE B, Punto-flotante), las cuales son muy distintas a aquellas en representación punto-fijo.

Es por esto que en los programas se deben de declarar, de alguna manera, el tipo de las variables, ya que internamente se manejan de una forma muy distinta, una de otra.

Así como la máquina no hace distinción entre instrucción y dato, lo mismo ocurre con el tipo de las variables, se le debe decir de alguna manera, en este caso, en la parte de declaración de variables, si se trata de un número entero o de un número real, para hacer la conversión interna correcta y llevar a cabo las operaciones en forma adecuada.

En el programa utilizado como ejemplo, se declararon las variables de la siguiente manera :

```
Var
 limite : integer;
 suma   : integer;
 i      : integer;
```

en ésta parte se le está indicando al compilador que nuestras tres variables son de tipo entero.

La pregunta que surge es, cuál es la forma adecuada de las operaciones ?. Lo que sucede es que las operaciones aritméticas de números enteros se llevan a cabo internamente en una forma muy distinta a las de los números reales.

Para nosotros, la suma 3 + 3.00 no presenta dificultad, el resultado sería el mismo que de la suma 3 + 3 o 3.00 + 3.0, pero para las computadoras ésto sería causa de conflictos.

Por ejemplo, si se suma un entero, el 3 cuya representación vimos que era "00000000 00000000 00000011", con un real, el mismo 3 pero con representación punto-flotante, que es "01100000 00000000 00000010"; si el compilador la trata como una suma de enteros, se produciría un error de ejecución debido a que el resultado que se obtendría es 6 291 461, "01100000 00000000 00000101" el cual no se encuentra en el rango de enteros de la mayoría de las computadoras, y además difiere del 6 esperado, "00000000 00000000 00000110".

A nivel de máquina y de ensamblador se debe indicar el tipo de operación aritmética, de comparación y de lectura y escritura para evitar éstos errores, es decir, si se trata de una suma, resta, comparación, lectura, etc. de enteros o de reales.

Por ejemplo, "SUMA VALOR" para sumar el entero almacenado en el acumulador con la variable entera VALOR, "SUMAF TOTAL" para sumar el real almacenado en el acumulador y el real TOTAL, "LEEENT LIMIT" para leer el valor de la variable entera LIMIT.

A nivel de compilación, al conocer los tipos de las variables que intervienen, se construyen las operaciones a nivel ensamblador de acuerdo a éstos.

Por ejemplo, la instrucción "read limite", se traduce como "LEEENT LIMIT"; Si la variable limite se hubiera declarado como de tipo real, entonces la traducción sería "LEEREAL LIMIT".

Las operaciones aritméticas y de comparación se realizan entre variables del mismo tipo, es decir, enteros con enteros y reales con reales.

Algunos compiladores hacen la conversión automática, es decir, en forma implícita, ésto es, si se trata de hacer alguna operación entre un entero y un real, el entero siempre es convertido a su equivalente en representación punto-flotante.

Por ejemplo, si se tiene la suma de 3 + 3.00, el entero 3, "00000000 00000000 00000011" se convierte al real 3.00, "01100000 00000000 00000010" y así obtener un resultado de tipo real, 6.00, "01100000 00000000 00000011". Es importante señalar que la variable donde se guarde el valor debe ser de tipo real, de lo contrario habrá un error.

Pero así como hay compiladores que efectúan la conversión automática, hay otros que no la hacen (nuestro caso) y no aceptan, por lo tanto, operaciones entre variables de tipo diferente. Esto causa muchos errores a los programadores inexpertos, aunque los expertos pueden llegar a caer en ellos. La solución a ésto es hacer la conversión en forma explícita.

En el apéndice F se encuentra el Manual de Uso del Sistema "SEC".

CONCLUSIONES

CONCLUSIONES

Actualmente se vive la era de la revoluciòn de las computadoras cuyo impacto serà profundo. Tendrà como beneficio el incremento en el potencial mental del hombre y, se ha llegado a un momento en que con sòlo oprimir un botòn se pueden realizar càlculos muy complicados, llevar a cabo decisiones, almacenar y recuperar grandes cantidades de informaciòn, jugar ajedrez, analizar electrocardiogramas, etc. Se ha llegado a considerar a la computadora como el sistema intelectual de los robots.

Con esta tesis se desea dar una introducciòn sencilla al mundo de las computadoras y, despertar el interès hacia esta rama, ya que se considera que en un futuro no muy lejano, si no es que ya, no serà suficiente con encender una computadora y echar a andar programas (ejecutar los tantos paquetes que existen hoy en dia).

El proyecto presentado en esta tesis, pretende ser un prototipo didàctico. Los componentes principales de que consta un sistema de computadora (HARDWARE y SOFTWARE) estàn sujetos a cambios constantes, ya que el avance tecnològico en estas àreas es impresionante, por lo que el diseñar una computadora con tecnologia de vanguardia resultaria casi imposible, ademàs de que no cumpliria con el objetivo trazado, que es el de exponer los conceptos fundamentales del tema sin entrar en particularidades.

Primero, se da una visiòn de la estructura del HARDWARE. Se muestran sus componentes bàsicos (memoria, procesador y entrada/salida) y las funciones que realiza cada uno de ellos.

Se muestra la estructura de la memoria, la cual està formada por un àrea de almacenamiento y dos registros (RDAM, RDIM), el del procesador, que consta de registros (RI, RDE, X, A, AP, CP, RC) y unidades (unidad de control, UAL) y el equipo periférico, formado por un monitor (para la salida) y un teclado (para la entrada).

Esta es la estructura de la màquina COSI, la cual contiene los elementos bàsicos de que consta cualquier computadora simple, por lo que cumple con nuestro propòsito.

Todos estos elementos intervienen durante la ejecuciòn de un programa, lo cual el sistema lo muestra de una manera accesible.

El contenido del procesador y de la memoria se ilustra en forma binaria para dar realismo al hecho, salvo los contenidos de los registros que guardan direcciones de memoria, los cuales se muestran en decimal, èsto se debe a que se consideró que de esta manera se ilustraría y se entendería mejor.

Pero todo èsto no sería útil sin el SOFTWARE, que en nuestro caso consta de un ensamblador y un compilador.

Una de las cosas que el sistema trata a grandes rasgos, es el proceso de traducciòn de un programa, primero la compilaciòn, que en este caso es la traducciòn del programa en lenguaje de alto nivel al lenguaje ensamblador, el cual lo lleva a cabo el compilador y posteriormente, el ensamblado, que es la traducciòn del lenguaje ensamblador a su equivalente en lenguaje de màquina y que es realizado por el ensamblador.

Con el programa a nivel màquina, se ejemplifica el uso del còdigo binario, lo cual es lo ùnico que entiende la computadora.

En el caso del lenguaje ensamblador, se muestra que es una versiòn mnemònica (simbòlica) del lenguaje de màquina, ademàs de que existen las pseudo-instrucciones que pueden o no generar instrucciones a nivel màquina.

Cabe mencionar que la compilaciòn se pudo hacer de manera que tradujera directamente del lenguaje de alto nivel al lenguaje a nivel màquina, pero se consideró que era conveniente el ilustrar que existe el lenguaje ensamblador y un proceso llamado ensamblado que es llevado a cabo por un ensamblador.

Con èsto se ilustra la evoluciòn que han tenido los lenguajes, cuyo objetivo es la comodidad del ser humano y el mejor aprovechamiento y uso de las computadoras.

Con lo anterior se ilustra el proceso que se realiza antes de que un programa sea ejecutado, es decir, la traducción de éste a una forma entendible por la máquina.

Otra de las cosas que se ilustra es la representación interna de los números, la diferencia que existe entre la representación de los números enteros (punto-fijo) y la de los números reales (punto-flotante).

Se observó que las representaciones son muy distintas una con otra y se entiende el porqué en un programa se debe hacer la distinción entre el tipo de número, entero o real. También el porqué muchos compiladores no aceptan operaciones entre diferentes tipos, a menos que éste haga una conversión automática de tipos en caso de diferir.

Con lo desarrollado creo que se logra el objetivo de la tesis, que es el de ayudar en forma gráfica, en la enseñanza de los conceptos fundamentales de un sistema de cómputo simple, sin entrar en detalles, sino en forma general, presentando lo esencial.

El trabajo está dirigido a estudiantes que incursionan en el área de computación (Computación I), es por ésto que se hizo con un esquema general y sencillo para facilitar su comprensión. Pero también, puede ser útil en otros cursos como Computación II y Programación de Sistemas.

Como todo trabajo, éste también tiene sus limitaciones. El uso de subrutinas no es posible en el lenguaje de alto nivel; el simulador sí las maneja, pero el compilador no. En el capítulo 2 se explica de manera breve una técnica para el paso de parámetros.

También hace falta el uso de arreglos. El simulador contempla la instrucción que reserva el espacio en memoria, pero carece de las necesarias para su manejo.

Estas limitaciones se pueden solucionar sin que ésto represente un cambio extremo en el diseño de COSI.

APENDICE A

CONJUNTO DE INSTRUCCIONES

CONJUNTO DE INSTRUCCIONES

COSI proporciona un conjunto bàsico de instrucciones para realizar un gran nûmero de tareas simples.

La siguiente tabla contiene una lista de todas las instrucciones, con sus respectivos mnemònicos, còdigos de operaciòn (hexadecimal) y una descripciòn de la funciòn que realizan.

	Mnemònico	Còdigo	Funciòn	
MOVIMIENTO DE DATOS				
CARGA DE REGISTRO	CARGAA m	00	A <- (m .. m+2)	
	CARGAF m	01	A <- (m .. m+2)	*
	CARGAX m	02	X <- (m .. m+2)	
GUARDA DE REGISTRO	GUARDA m	03	(m .. m+2) <- A	
	GUARDAX m	04	(m .. m+2) <- X	
OPERACIONES ARITMETICAS Y LOGICAS				
OPERACION ARITMETICA	SUMA m	10	A <- (A) + (m .. m+2)	
	RESTA m	11	A <- (A) - (m .. m+2)	
	MULTI m	12	A <- (A) * (m .. m+2)	
	DIVIDE m	13	A <- (A) / (m .. m+2)	
	SUMAF m	14	A <- (A) + (m .. m+2)	*
	RESTAF m	15	A <- (A) - (m .. m+2)	*
	MULTIF m	16	A <- (A) * (m .. m+2)	*
	DIVIDEF m	17	A <- (A) / (m .. m+2)	*
	NEG	18	A <- -(A)	
	NEGF	19	A <- -(A)	*
COMPARACION	COMP m	1A	(A) : (m .. m+2)	
	COMPF m	1B	(A) : (m .. m+2)	*

	Mnemónico	Código	Función	
	TRANSFERENCIA DE CONTROL			
TRANSFERENCIA	SALTA m	20	CP <- m	
	SALTIG m	21	CP <- m si RC es =	
	SALTDIF m	22	CP <- m si RC es <>	
	SALTME m	23	CP <- m si RC es <	
	SALTMA m	24	CP <- m si RC es >	
	SALTMEIG m	25	CP <- m si RC es <=	
	SALTMAIG m	26	CP <- m si RC es >=	
SUBRUTINA	SALTSUB m	27	AP <- (CP) ; CP <- m	
	REGSUB	28	CP <- (AP)	
	ENTRADA / SALIDA			
ENTRADA/SALIDA	LEEENT m	30	(m .. m+2) <- entero	
	LEEREAL m	31	(m .. m+2) <- real	*
	ESCENT m	32	entero <- (m .. m+2)	
	ESCREAL m	33	real <- (m .. m+2)	*
	OTROS			
	REGSIST	40	regresa el control al sistema operativo	

* Instrucciones utilizadas para números en punto-flotante.

Las pseudo-instrucciones disponibles en el lenguaje ensamblador para COSI se enuncian a continuación, con su respectivo formato y una descripción de su función.

Pseudo-instrucción	Formato	Función
INICIO	INICIO m	CL <- m
FIN	FIN	termina el ensamble
MANEJO DE DATOS		
PAL	etiq PAL m	metetabla(etiq,CL); (CL .. CL+3) <- m; CL <- CL + 3
RPAL	etiq RPAL m	metetabla(etiq,CL); CL <- CL + (3 * m)

NOTA : metetabla(etiq,CL) es la función que agrega etiq en la tabla de símbolos con el valor del CL (contador de localidad).

APENDICE B

REPRESENTACION DE LOS DATOS

REPRESENTACION DE LOS DATOS

SISTEMAS NUMERICOS

En un sistema numèrico, un nûmero se representa por una cadena de digitos, en donde cada posiciòn de los digitos tiene un peso asociado.

En el sistema numèrico decimal, el valor D de un nûmero de 4 digitos $d_3d_2d_1d_0$ es :

$$D = d_3 \cdot 10^3 + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

Cada digito d_i tiene un peso de 10^i . Asi, el nûmero 7892 se calcula como :

$$\begin{aligned} 7892 &= 7 \cdot 10^3 + 8 \cdot 10^2 + 9 \cdot 10^1 + 2 \cdot 10^0 \\ &= 7 \cdot 1000 + 8 \cdot 100 + 9 \cdot 10 + 2 \cdot 1 \end{aligned}$$

El punto decimal se usa para representar tanto potencias negativas como positivas. Asi, $d_1d_0.d_{-1}d_{-2}$ tiene el valor :

$$D = d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$

Por ejemplo, el valor de 46.41 se calcula como :

$$\begin{aligned} 46.41 &= 4 \cdot 10^1 + 6 \cdot 10^0 + 4 \cdot 10^{-1} + 1 \cdot 10^{-2} \\ &= 4 \cdot 10 + 6 \cdot 1 + 4 \cdot .1 + 1 \cdot .01 \end{aligned}$$

En general, cada posiciòn tiene un peso asociado de b, donde b es la base o la raiz de un sistema numèrico. La forma general de un nûmero es :

$$d_{p-1}\ d_{p-2} \cdots d_1\ d_0\ .\ d_{-1}\ d_{-2} \cdots d_{-n}$$

donde hay p digitos a la izquierda del punto y n digitos a la derecha del punto. El valor D de un nûmero se calcula como :

$$D = \sum_{p-1>i>-n} d_i \cdot b^i$$

La representación de un número es única. El dígito a la izquierda en un número se conoce como el dígito de mayor orden o de mayor significancia; el de la derecha es el dígito de menor orden o de menor significancia.

SISTEMA BINARIO

La base binaria se utiliza en casi todas las computadoras. Los dígitos permitidos, 0 y 1, son llamados bits; cada bit d tiene un peso de 2 .

$$10001_2 = 1\cdot2^4 + 0\cdot2^3 + 0\cdot2^2 + 0\cdot2^1 + 1\cdot2^0$$
$$= 1\cdot16 + 0\cdot8 + 0\cdot4 + 0\cdot2 + 1\cdot1 = 17_{10}$$

Los números subscritos se utilizan para indicar la base del número.

SISTEMA OCTAL Y HEXADECIMAL

Las bases 8 (octal) y 16 (hexadecimal) proporcionan una representación conveniente para los números en una computadora.

El sistema octal requiere 8 dígitos, del 0 al 7. El sistema hexadecimal requiere 16 dígitos, aquí las letras A a la F se usan además de los dígitos 0 al 9.

Los sistemas octal y hexadecimal son útiles para representar números binarios, ya que sus raíces son potencias de dos. Como una cadena de 3 bits puede tomar 8 diferentes combinaciones, se sigue que cada 3 bits se representan por un dígito en octal.

De igual manera, una cadena de 4 bits se representa por un número hexadecimal. Se agrupa del punto hacia la izquierda. Como ejemplos tenemos :

$$1010110001100_2 = (101\ 011\ 000\ 110)_2 = 5306_8$$
$$= (1010\ 1100\ 0110)_2 = AC6_{16}$$

$$1101100111010100 1_2 = (011\ 011001\ 110\ 101\ 001)_2 = 33165_8$$
$$= (0001\ 1011\ 0011\ 1010\ 1001)_2 = 1133A9_{16}$$

Se puede agregar ceros para acompletar a 3 o 4 dígitos, según sea octal o hexadecimal, respectivamente.

En caso de contener dígitos a la derecha del punto decimal se procede de igual forma, comenzando del punto a la derecha.

$$11.1010011011_2 = (011\ .\ 101\ 001\ 101\ 100)_2 = 3.5154_8$$
$$= (0011\ .\ 1010\ 0110\ 1100)_2 = 3.A6C_{16}$$

Convertir de octal o hexadecimal a binario es fàcil. Simplemente se reemplaza cada dígito octal o hexadecimal con su correspondiente cadena de 3 o 4 bits.

$$1573_8 = (001\ 101\ 111\ 011)_2 = (0011\ 0111\ 1011)_2 = 37B_{16}$$

REPRESENTACION PUNTO-FIJO Y PUNTO-FLOTANTE

PUNTO-FIJO

Es posible interpretar una cadena de ceros y unos en varias formas. Comencemos considerando la interpretaciòn de enteros. Una cadena de bits es un entero binario y existen varios mètodos para indicar el signo del nùmero.

Se explicaràn brevemente 3 mètodos de tratar el signo. Cada uno de èstos es una variedad de lo que se conoce como representaciòn de nùmeros en punto-fijo.

El tèrmino punto-fijo se refiere al hecho de que el punto decimal tiene una posiciòn fija en la palabra de la computadora; si se considera esta posiciòn al lado derecho, todos los nùmeros son enteros, negativos o positivos.

- COMPLEMENTO A UNO

Un número negativo x se representa por :

$$x = 2^n - 1 - |x|.$$

Si n = 4, entonces

$$-5_{10} = 2^4 - 1 - |-5_{10}|$$
$$= 1111_2 - 0101_2 = 1010_2$$

Se tiene un doble cero pero su ventaja reside en la operación sencilla de complemento. El complemento a uno se obtiene fácilmente con sólo cambiar los bits.

17_{10} es 00010001_2
invirtiendo 11101110_2 el cual representa -17_{10}

-127_{10} es 10000000_2
invirtiendo 01111111_2 el cual representa 127_{10}

PUNTO-FLOTANTE

Esta es la segunda forma común de interpretar números. Es similar a la notación científica. Está formada por el signo de la mantisa, la mantisa, el signo del exponente y el exponente.

Supóngase que los bits del exponente representan el entero "E" y los bits de la mantisa representan la fracción "M", donde 0 <= M < 1. El valor del número será :

$$\pm 2^{\pm E} * M$$

dependiendo de los bits de signo (0=positivo. 1=negativo).

Los m bits de la mantisa representan una fracción con el punto a la izquierda. Entonces M, el valor de la mantisa, es siempre un número entre 0 y 1, excluyendo el 1. El exponente de e bits representa enteros de 0 a $2^e - 1$.

Supóngase e = 6 y m = 10, entonces

sm	mantisa	se	exponente		
0	0000000000	0	000000	$= 0 \cdot 2^{0}$	= 0
1	1100101100	0	001000	$= -.5429687 \cdot 2^{8}$	= -138.99
0	1000000000	1	000010	$= .5 \cdot 2^{-2}$	= .125

Existen varias representaciones diferentes para un mismo número, por ejemplo,

+1 = 0 1000000000 0 000001
= 0 0100000000 0 000010
= 0 0010000000 0 000100

Sin embargo, muchas computadoras normalizan los números. Un número diferente de cero en punto-flotante es normalizado si el bit de la izquierda de la mantisa es diferente de cero. Con ésto, todo número tiene una representación única.

La representación normalizada del cero es aquella con todos los bits igualados a cero. Si dos números X y Y están normalizados, se garantiza que X es mayor que Y si el exponente de X es numéricamente mayor que el exponente de Y. Sólo si los exponentes son iguales, la relación entre X y Y depende de los valores de la mantisa.

REPRESENTACION DE DATOS ALFANUMERICOS

Mucha de la información procesada en computadoras no es representada en forma numérica sino como cadenas de caracteres.

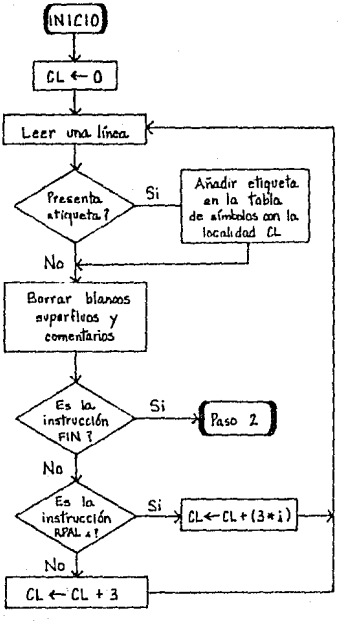
Los registros de los empleados de una compañía, por ejemplo, contienen información tal como nombre, dirección, puesto. Cada uno de éstos puede ser almacenado como una secuencia de caracteres.

Cada caracter es asociado con un único número binario. Existe un conjunto estándar de códigos de 7 bits llamado ASCII (American Standard Code for Information Interchange) que con frecuencia se almacenan en bytes (8 bits), en donde el octavo bit es un cero.

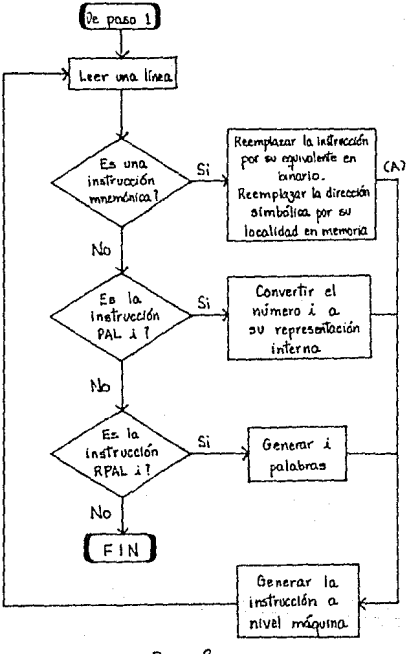
Se debe notar que los caracteres numéricos son muy distintos de los números representados en un sistema como el complemento a dos. Por ejemplo, el número binario 5_{10} se representa por 00000101_2 y el caracter 5 en código ASCII por 00110101.

APENDICE C

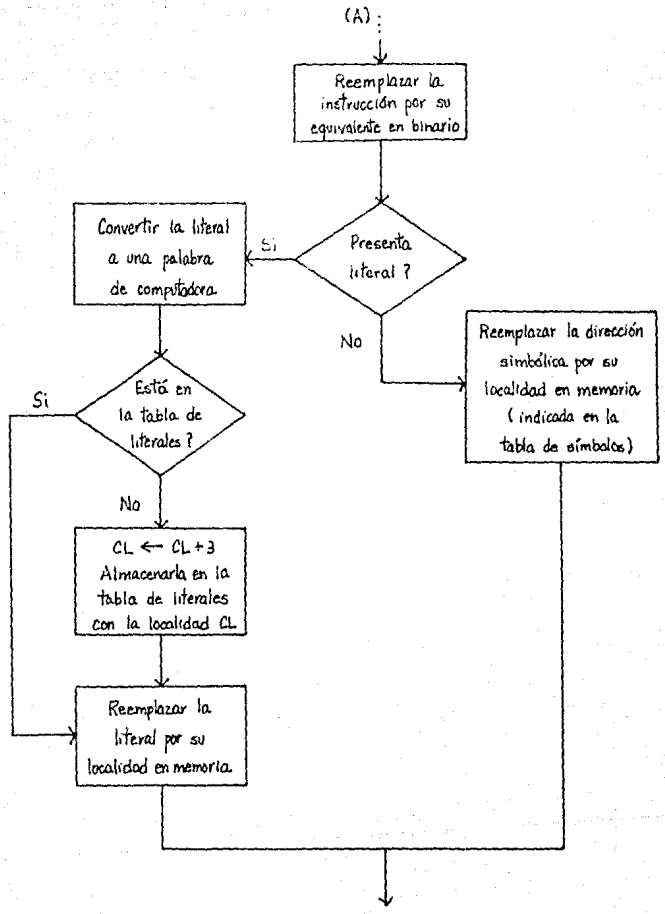
DIAGRAMA DE FLUJO DEL ENSAMBLADOR DE DOS PASOS


INICIO
CL ← 0
Leer una línea
Presenta etiqueta?
Si
Añadir etiqueta en la tabla de símbolos con la localidad CL
No
Borrar blancos superfluos y comentarios
Es la instrucción FIN ?
Si
Paso 2
No
Es la instrucción RPAL x ?
Si
CL ← CL + (3 * i)
No
CL ← CL + 3

Paso 1


De paso 1
Leer una línea
Es una instrucción mnemónica?
Si
Reemplazar la instrucción por su equivalente en binario. Reemplazar la dirección simbólica por su localidad en memoria
(A)
No
Es la instrucción PAL i ?
Si
Convertir el número i a su representación interna
No
Es la instrucción RPAL i ?
Si
Generar i palabras
No
FIN
Generar la instrucción a nivel máquina

Paso 2



Parte del Paso 2 para manejo de literales

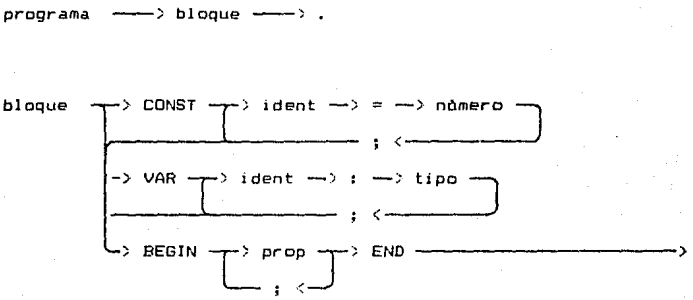
APENDICE D

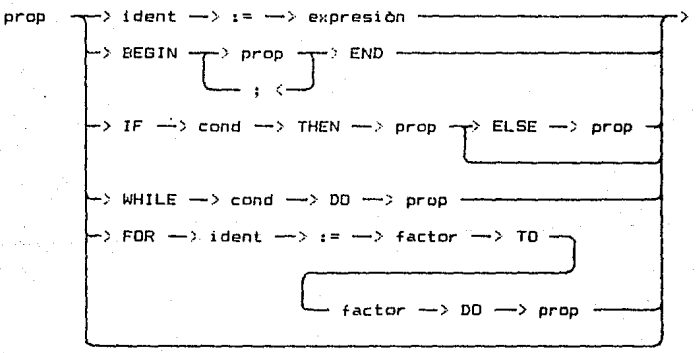
==========

SINTAXIS DEL LENGUAJE

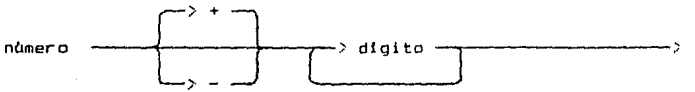
==========

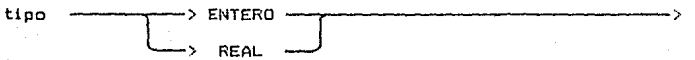
SINTAXIS DEL LENGUAJE

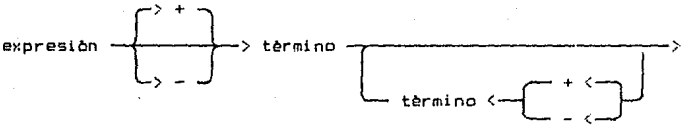


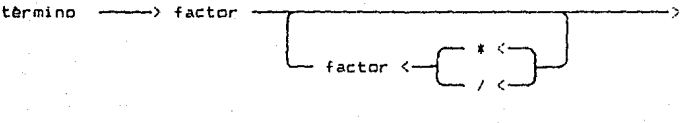


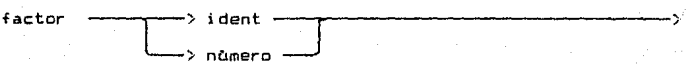

ident
letra
dígito
letra


número
+
-
dígito

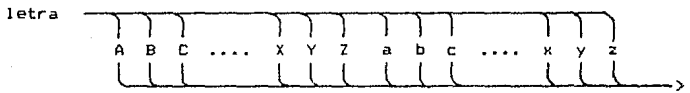

tipo
ENTERO
REAL

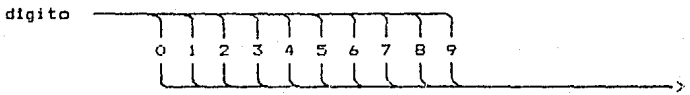

expresión
+
-
término
término
+
-


término
factor
factor
*
/


factor
ident
número


cond
factor
=
<>
<
>
<=
>=
factor


letra
A
B
C
....
X
Y
Z
a
b
c
....
x
y
z


digito
0
1
2
3
4
5
6
7
8
9

APENDICE E

LISTADO DEL PROGRAMA COMPILADOR

```
Procedure COMPILER;      (* compila un programa y obtiene su equivalente *)
                         (* en lenguaje ensamblador                      *)

const  numpalres = 15;   numident = 50;   longident = 5;

type   simbolo = (nul,ident,numero,mas,menos,por,entre,igual,dif,menor,
         mayor,punto,menig,mayig,coma,pcoma,dpuntos,viene,intsim,realsim,
         beginsim,endsim,constsim,varsim,ifsim,thensim,elsesim,whilesim,
         dosim,forsim,tosim,readsim,writesim);
       objeto  = (constante,variable);
       tipo    = (numentero,numreal);

var    cc,ll,itid,ietiq  :  byte;
       ch                : char;
       id                : string[8];
       error             : boolean;
       tipnum            : tipo;
       sim               : simbolo;
       palres            : array[1..numpalres] of st8;
       palressim         : array[1..numpalres] of simbolo;
       tiposim           : array[char] of simbolo;
       tablaident        : array[0..numident] of record
                                nombre : st8;
                                clase : objeto;
                                val : st8;
                                tip : tipo;
                                icod : integer;
                              end;


 procedure inicializa;    (* inicializa las tablas y las variables *)
                          (* que intervienen en la compilación y lee *)
                          (* el programa fuente al arreglo          *)
  begin
   palres[ 1] := 'BEGIN';          palres[ 2] := 'CONST';
   palres[ 3] := 'DO';             palres[ 4] := 'ELSE';
   palres[ 5] := 'END';            palres[ 6] := 'FOR';
   palres[ 7] := 'IF';             palres[ 8] := 'INTEGER';
   palres[ 9] := 'READ';           palres[10] := 'REAL';
   palres[11] := 'THEN';           palres[12] := 'TO';
   palres[13] := 'VAR';            palres[14] := 'WHILE';
   palres[15] := 'WRITE';
   palressim[ 1] := BEGINSIM;      palressim[ 2] := CONSTSIM;
   palressim[ 3] := DOSIM;         palressim[ 4] := ELSESIM;
   palressim[ 5] := ENDSIM;        palressim[ 6] := FORSIM;
   palressim[ 7] := IFSIM;         palressim[ 8] := INTSIM;
   palressim[ 9] := READSIM;       palressim[10] := REALSIM;
   palressim[11] := THENSIM;       palressim[12] := TOSIM;
   palressim[13] := VARSIM;        palressim[14] := WHILESIM;
   palressim[15] := WRITESIM;
   tiposim['+'] := mas;            tiposim['-'] := menos;
   tiposim['*'] := por;            tiposim['/'] := entre;
   tiposim['='] := igual;          tiposim[','] := coma;
   tiposim['.'] := punto;          tiposim['<'] := menor;
```

```
  tiposim[')'] := mayor;              tiposim[';'] := pcoma;
  tiposim[':'] := dpuntos;
  ifuente := 0;  cc := 0;  ll := 0;  ch := ' ';  icodens := 0;
  icodfuente := 0;  ietiq := 0;  sim := nul;  itid := 0;
  error := false;  entrofor := false;
  while (not eof(prog_fuente)) do
   begin
    readln (prog_fuente,linea);
    if (length(linea)<>0) then
     begin
      while (linea[1]=' ') do linea := copy(linea,2,length(linea));
      codfuente[ifuente].inst := linea+' ';  ifuente := ifuente + 1;
     end;
   end;
end;

procedure traesimbolo;     (* lee el  siguiente  atomo del  programa y *)
                           (* determina de que clase es el simbolo     *)
 var  i,j,k : integer;  esreal : boolean;

 procedure traecar;     (* obtiene el siguiente caracter               *)
  begin  { traecar }
   if cc=ll then
    if  ((icodfuente>=ifuente) and (sim<>punto)) then error := true
    else
     begin
      cc := 0;  ll := 0;  linea := '';
      linea := codfuente[icodfuente].inst;
      icodfuente := icodfuente + 1;  ll := length(linea);
     end;
   cc := cc + 1;  ch := UPCASE (linea[cc]);
  end;  { traecar }

 begin  { traesimbolo }
  while (ch=' ') do traecar;
  case ch of
   'A'..'Z' : begin
              id := ''; k := 0;
              while (ch in ['A'..'Z','0'..'9']) do
               begin
                id := concat (id,ch);  traecar;
               end;
              i := 1;  j := 15;
              repeat
               k := (i+j) div 2;
               if (id <= palres[k]) then j := k-1;
               if (id >= palres[k]) then i := k+1;
              until i>j;
              if (i-1>j) then sim := palressim[k]
              else
               begin
                id := copy(id,1,5);  sim := ident;
               end;
             end;
   '0'..'9' : begin
```

```
            id := '';  esreal := false;
            k := 0;  sim := numero;
            while (ch in ['0'..'9','.']) do
             begin
              id := concat(id,ch);
              if (ch='.') then esreal := true;
              k := k + 1;  traecar;
             end;
            if (esreal) then tipnum := numreal
            else tipnum := numentero;
           end;
  ':'     : begin
            traecar;
            if (ch='=') then
             begin
              sim := viene;  traecar;
             end
            else sim := dpuntos;
           end;
  '<'     : begin
            traecar;
            if (ch='>') then sim := dif
            else
            if (ch='=') then sim := menig
            else sim := menor;
            if (ch in ['>','=']) then traecar;
           end;
  '>'     : begin
            traecar;
            if (ch='=') then
             begin
              sim := mayig;  traecar;
             end
            else sim := mayor;
           end;
  '+','-','*','/','=','.',',',';'
          : begin
            sim := tiposim[ch];
            if (sim<>punto) then traecar;
           end;
  '$'     : begin
            cc := 11;  ch := ' ';  traesimbolo;
           end;
  else    : error := true;
 end;
end;  { traesimbolo }

procedure generacod (et,mne,op : st8);     (* genera las proposiciones *)
                                           (* a nivel ensamblador      *)
begin  { generacod }
 codens[icodens].inst := et+' '+mne+' '+op;  icodens := icodens + 1;
end;  { generacod }

procedure vaciaident;     (* agrega las variables al final del *)
                          (* programa a nivel ensamblador      *)
```

```
 begin  { vaciaident }
  for ll:=1 to itid do
   if (tablaident[ll].clase=variable) then
    begin
     codfuente[tablaident[ll].icod].coneccion := icodens;
     generacod (tablaident[ll].nombre,'RPAL','1');
    end;
 end;  { vaciaident }

procedure bloque;      (* examina el programa fuente en su estructura y *)
                       (* verifica la correcta escritura de constantes, *)
                       (* variables y proposiciones                     *)

 function posicion (auxid : st8) : integer;
                       (* regresa el  valor 0 si el identificador no se *)
                       (* encuentra en la tabla o el  número del  lugar *)
                       (* que ocupa si si esta                          *)
  var  ii : integer;
  begin  { position }
   tablaident[0].nombre := auxid;  ii := itid;
   while (tablaident[ii].nombre<>auxid) do ii := ii - 1;
   posicion := ii;
  end;  { position }

 procedure meteident (k : objeto; auxid : st8);
                       (* agrega un nuevo  identificador a la  tabla de *)
                       (* identificadores con su tipo, valor, etc.      *)
  var  ii : integer;
  begin  { meteident }
   ii := posicion(auxid);
   if (ii=0) then
    begin
     itid := itid + 1;
     if (k=variable) then
      begin
       if (sim=intsim) then tipnum := numentero
       else tipnum := numreal;
       id := '';
      end;
     with tablaident[itid] do
      begin
       nombre := auxid;  clase := k;  val := id;
       tip := tipnum;  icod := icodfuente - 1;
      end;
    end
   else error := true;
  end;  { meteident }

procedure declaracionconst;     (* examina  la  parte  de  la *)
                                (* declaración de constantes  *)
 var  neg : boolean;  auxid : st8;
 begin  { declaracionconst }
  if ((sim=ident) and (not error))then
   begin
    auxid := id;  traesimbolo;
```

```
    if ((sim=igual) and (not error)) then
     begin
      neg := false;  traesimbolo;
      if ((sim=menos) and (not error))then
       begin
        neg := true;  traesimbolo;
       end;
      if ((sim=numero) and (not error)) then
       begin
        if (neg) then id := concat('-',id);
        meteident (constante,auxid);  traesimbolo;
       end
      else error := true;
     end
    else error := true;
   end
  else error := true;
 end;  { declaracionconst }

procedure declaracionvar;     (* examina la  parte que  corresponde a *)
                              (* la declaración de variables          *)
 var  auxid : st8;
 begin  { declaracionvar }
  if ((sim=ident) and (not error)) then
   begin
    auxid := id;  traesimbolo;
    if ((sim=dpuntos) and (not error)) then
     begin
      traesimbolo;
      if (((sim=intsim) or (sim=realsim)) and (not error)) then
       meteident (variable,auxid)
      else error := true;
      if (not error) then traesimbolo;
     end
    else error := true;
   end
  else error := true;
 end;  { declaracionvar }

procedure  proposicion;       (* examina  el cuerpo  principal de todo *)
                              (* programa, las proposiciones           *)
 var  ii,etiq,etiq1 : integer;  auxid : st8;
      auxcad : string[3];

 procedure expresion (t : tipo);     (* examina las expresiones        *)
  var  sumop : simbolo;

  procedure termino (func : st8);     (* examina  algunas  de  las *)
                                      (* operaciones aritméticas   *)
   var  mulop : simbolo;

   procedure factor (func : st8);     (* determina  el  tipo  del *)
                                      (* identificador  y  genera las *)
                                      (* expresiones              *)
    var  ii : integer;
```

```
  begin  { factor }
   if ((sim=ident) and (not error)) then
    begin
     ii := posicion(id);
     if (ii=0) then error := true
     else
      if (tablaident[ii].tip=t) then
       if (tablaident[ii].clase=constante) then
        generacod ('  ',func,tablaident[ii].val)
       else generacod ('  ',func,id)
      else error := true;
    end
   else
    if ((sim=numero) and (not error)) then
     if (tipnum=t) then generacod ('  ',func,id)
     else error := true
    else error := true;
   if (not error) then traesimbolo;
  end;  { factor }

 begin  { termino }
  factor (func);
  while ((sim in [por,entre]) and (not error)) do
   begin
    mulop := sim;  traesimbolo;
    if (not error) then
     if (mulop=por) then
      if (t=numentero) then factor ('MULTI')
      else factor ('MULTIF')
     else
      if (t=numentero) then factor ('DIVIDE')
      else factor ('DIVIDEF');
   end;
 end;  { termino }

begin  { expresion }
 if ((sim in [mas,menos]) and (not error)) then
  begin
   sumop := sim;  traesimbolo;
   if (not error) then
    if (t=numentero) then termino('CARGAA')
    else termino ('CARGAF');
   if (not error) then
    if (sumop=menos) then
     if (t=numentero) then generacod ('  ','NEG',id)
     else generacod ('  ','NEGF',id);
  end
 else
  if (t=numentero) then termino ('CARGAA')
  else termino ('CARGAF');
 if (not error) then
  while ((sim in [mas,menos]) and (not error)) do
   begin
    sumop := sim;  traesimbolo;
    if (not error) then
```

```
      if (sumop=mas) then
       if (t=numentero) then termino ('SUMA')
       else termino ('SUMAF')
      else
       if (t=numentero) then termino ('RESTA')
       else termino ('RESTAF');
    end;
 end;  { expresion }

procedure condicion;     (* examina la condición de las  estructuras *)
                         (* IF y WHILE y determina las instrucciones *)
                         (* necesarias para su traducción            *)
 var  ii : integer;  t : tipo;  auxsim : simbolo;
 begin  { condicion }
  if (not error) then
   begin
    if (sim=ident) then
     begin
      ii := posicion(id);
      if (ii=0) then error := true
      else
       begin
        t := tablaident[ii].tip;
        if (t=numentero) then
         if (tablaident[ii].clase=variable) then
          generacod ('  ','CARGAA',id)
         else generacod ('  ','CARGAA',tablaident[ii].val)
        else
         if (tablaident[ii].clase=variable) then
          generacod ('  ','CARGAF',id)
         else generacod ('  ','CARGAF',tablaident[ii].val);
       end;
     end
    else
     if (sim=numero) then
      begin
       t := tipnum;
       if (t=numentero) then generacod ('  ','CARGAA',id)
       else generacod ('  ','CARGAF',id);
      end
     else error := true;
    if (not error) then traesimbolo;
    if (not error) then
     begin
      auxsim := sim;  traesimbolo;
      if (not error) then
       if (sim=ident) then
        begin
         ii := posicion (id);
         if (ii=0) then error := true
         else
          if (tablaident[ii].tip=t) then
           if (t=numentero) then
            if (tablaident[ii].clase=variable) then
             generacod ('  ','COMP',id)
```

```
          else generacod ('  ','COMP',tablaident[ii].val)
         else
          if (tablaident[ii].clase=variable) then
           generacod ('  ','COMPF',id)
          else generacod ('  ','COMPF',tablaident[ii].val)
         else error := true;
        end
       else
        if (sim=numero) then
         if (tipnum=t) then
          if (t=numentero) then generacod ('  ','COMP',id)
          else generacod ('  ','COMPF',id)
         else error := true
        else error := true;
      if (not error) then
       begin
       ietiq := ietiq + 1;  etiq := etiq + 1;  STR (etiq,auxcad);
       if (auxsim=igual) then generacod ('  ','SALTDIF',auxcad)
       else
        if (auxsim=dif) then generacod ('  ','SALTIG',auxcad)
        else
         if (auxsim=menor) then generacod ('  ','SALTMAIG',auxcad)
         else
          if (auxsim=mayor) then generacod ('  ','SALTMEIG',auxcad)
          else
           if (auxsim=menig) then generacod ('  ','SALTMA',auxcad)
           else
            if (auxsim=mayig) then generacod ('  ','SALTME',auxcad)
            else error := true;
       if (not error) then traesimbolo;
      end;
     end;
   end;
end;  { condicion }

procedure establecelimite;     (* establece los limites a nivel *)
                               (* ensamblador de la estructura FOR  *)
 var  ii : integer;
 begin  { establecelimite }
  if (sim=viene) then
   begin
    traesimbolo;
    if ((sim=ident) and (not error)) then
     begin
      ii := posicion (id);
      if (ii=0) then error := true
      else
       if (tablaident[ii].tip=numentero) then
        if (tablaident[ii].clase=variable) then
         generacod ('  ','CARGAA',id)
        else generacod ('  ','CARGAA',tablaident[ii].val)
       else error := true;
     end
    else
     if ((sim=numero) and (not error)) then
```

```
      if (tipnum=numentero) then generacod ('  ','CARGAA',id)
      else error := true;
    if (not error) then
     begin
      traesimbolo;
      if ((sim=tosim) and (not error)) then
       begin
        ietiq := ietiq + 1;  etiq := etiq + 1;
        STR (etiq,auxcad);  generacod (auxcad,'  ','  ');
        generacod ('  ','GUARDA',auxid);  traesimbolo;
        if (not error) then
         begin
          if (sim=ident) then
           begin
            ii := posicion(id);
            if (ii=0) then error := true
            else
             if (tablaident[ii].tip=numentero) then
              if (tablaident[ii].clase=variable) then
               generacod ('  ','COMP',id)
              else generacod ('  ','COMP',tablaident[ii].val)
             else error := true;
           end
          else
           if (sim=numero) then
            if (tipnum=numentero) then generacod ('  ','COMP',id)
            else error := true
           else error := true;
          if (not error) then traesimbolo;
         end;
       end
      else error := true;
     end;
   end
  else error := true;
 end;  { establecelimite }

procedure incrementa;    (* genera las instrucciones que incrementan *)
                         (* en uno  a la  variable de  control de la *)
                         (* de la estructura FOR                     *)
 begin  { incrementa }
  generacod ('  ','CARGAA',auxid);  generacod ('  ','SUMA','1');
  STR ((etiq-1),auxcad);  generacod ('  ','SALTA',auxcad);
  STR (etiq,auxcad);  generacod (auxcad,'  ','  ');
 end;  { incrementa }

begin  { proposicion }
 if (not error) then
  begin
   if (entrofor) then
    begin
     codfuente[icodfuente-1].coneccion := icodens - 1;
     entrofor := false;
    end
   else codfuente[icodfuente-1].coneccion := icodens;
```

```
etiq := ietiq;
if ((sim=beginsim) and (not error)) then
 begin
  traesimbolo;  proposicion;
  while ((sim=pcoma) and (not error)) do
   begin
    traesimbolo;  proposicion;
   end;
  if ((sim=endsim) and (not error)) then traesimbolo
  else error := true;
 end
else
if ((sim=ident) and (not error)) then
 begin
  ii := posicion(id);
  if (ii=0) then error := true
  else
   if (tablaident[ii].clase<>variable) then error := true
   else
    begin
     traesimbolo;
     if ((sim=viene) and (not error)) then traesimbolo
     else error := true;
     if (not error) then expresion (tablaident[ii].tip);
     if (not error) then
      generacod ('  ','GUARDA',tablaident[ii].nombre);
    end;
 end
else
if ((sim=ifsim) and (not error)) then
 begin
  traesimbolo; condicion;
  if ((sim=thensim) and (not error)) then
   begin
    traesimbolo;
    if (not error) then
     begin
      proposicion;
      if ((sim=elsesim) and (not error)) then
       begin
        ietiq := ietiq + 1;  etiq1 := ietiq;
        STR (etiq1,auxcad);  generacod ('  ','SALTA',auxcad);
        STR (etiq,auxcad);  generacod (auxcad,'  ','  ');
        traesimbolo;
        if (not error) then proposicion;
        if (not error) then
         begin
          STR (etiq1,auxcad); generacod(auxcad,'  ','  ');
         end;
       end
      else
       begin
        STR (etiq,auxcad);  generacod (auxcad,'  ','  ');
       end;
     end;
```

```
    end
   else error := true;
  end
 else
 if ((sim=whilesim) and (not error)) then
  begin
   ietiq := ietiq + 1;  etiq := etiq + 1;  STR (etiq,auxcad);
   generacod (auxcad,' ',' ');  traesimbolo;  condicion;
   if ((sim=dosim) and (not error)) then traesimbolo
   else error := true;
   if (not error) then proposicion;
   if (not error) then
    begin
     STR ((etiq-1),auxcad);  generacod ('  ','SALTA',auxcad);
     STR (etiq,auxcad);  generacod (auxcad,'  ','  ');
    end;
  end
 else
 if ((sim=forsim) and (not error)) then
  begin
   traesimbolo;
   if (not error) then
    begin
     ii := posicion(id);
     if (ii=0) then error := true
     else
      if ((tablaident[ii].tip=numentero) and
          (tablaident[ii].clase=variable)) then
       begin
        auxid := id;  traesimbolo;
        if (not error) then
         begin
          establecelimite;
          if ((sim=dosim) and (not error)) then
           begin
            ietiq := ietiq + 1;  etiq := etiq + 1;
            STR (etiq,auxcad);
            generacod ('  ','SALTMA',auxcad); traesimbolo;
            if (not error) then proposicion;
            if (not error) then incrementa;
            entrofor := true;
           end
          else error := true;
         end;
       end
      else error := true;
    end;
  end
 else
 if ((sim=readsim) and (not error)) then
  begin
   traesimbolo;
   if ((sim=ident) and (not error)) then
    begin
     ii := posicion(id);
```

```
        if (ii=0) then error := true
        else
         if (tablaident[ii].clase=variable) then
          if (tablaident[ii].tip=numentero) then
           generacod ('  ','LEEENT',id)
          else generacod ('  ','LEEREAL',id)
         else error := true;
        if (not error) then traesimbolo;
       end
      else error := true;
     end
    else
    if ((sim=writesim) and (not error)) then
     begin
      traesimbolo;
      if ((sim=ident) and (not error)) then
       begin
        ii := posicion(id);
        if (ii=0) then error := true
        else
         if (tablaident[ii].clase=variable) then
          if (tablaident[ii].tip=numentero) then
           generacod ('  ','ESCENT',id)
          else generacod ('  ','ESCREAL',id)
         else
          if (tablaident[ii].tip=numentero) then
           generacod ('  ','ESCENT',tablaident[ii].val)
          else generacod ('  ','ESCREAL',tablaident[ii].val);
       end
      else
       if ((sim=numero) and (not error)) then
        if (tipnum=numentero) then generacod ('  ','ESCENT',id)
        else generacod ('  ','ESCREAL',id)
       else error := true;
      if (not error) then traesimbolo;
     end;
   end;
 end;  { proposicion }

begin   { bloque }
 if ((sim=constsim) and (not error)) then
  begin
   traesimbolo;  declaracionconst;
   while ((sim=pcoma) and (not error)) do
    begin
     traesimbolo;
     if (sim<>varsim and sim<>beginsim and not error) then
      declaracionconst;
    end;
  end;
 if ((sim=varsim) and (not error)) then
  begin
   traesimbolo;  declaracionvar;
   while ((sim=pcoma) and (not error)) do
    begin
```

```
      traesimbolo;
      if (sim<>beginsim and (not error)) then declaracionvar;
     end;
   end;
  if ((sim=beginsim) and (not error)) then
   begin
    ibegin := icodfuente;  proposicion;
   end
  else error := true;
 end;  { bloque }

BEGIN  { compiler }
 inicializa;   traesimbolo;
 if (not error) then bloque;
 if (not error) then
  begin
   if (sim<>punto) then error := true;
   else
    begin
     generacod ('  ','REGSIST',' ');  vaciaident;
     generacod ('  ','FIN','  ');
    end;
  end;
END;
```

APENDICE F

==========

MANUAL DE USO DEL SISTEMA "SEC"

==========

MANUAL DE USO DEL SISTEMA "SEC"

Se explicarà el funcionamiento y uso del sistema "SEC : un Sistema para la Enseñanza de Computaciòn" para su mejor aprovechamiento.

Para iniciar el uso del sistema se debe ejecutar el archivo con el nombre SEC.COM. Aparecerà un menù principal con cuatro opciones :

Estado del Procesador-Memoria	...	(1)
Compilacion -> Ensamblado -> Ejecucion	...	(2)
Ensamblado -> Ejecucion	...	(3)
Regresar a DOS	...	(4)

1. ESTADO DEL PROCESADOR-MEMORIA

En esta opciòn se observa el estado del procesador, es decir, la informaciòn almacenada en cada uno de los registros que lo componen, asì como el contenido especìfico de una localidad de memoria, la cual serà indicada por el usuario y deberà estar dentro del rango de la memoria.

Si esta opciòn se elige al inicio, es decir, antes de que las opciones 2 y 3 se utilizen, entonces, se observarà que los registros y la memoria se encuentran inicializados (igualados a cero).

En otro caso, el contenido tanto del procesador como el de la memoria serà aquel que haya quedado al final de la ejecuciòn del ùltimo programa.

2. COMPILACION -> ENSAMBLADO -> EJECUCION

En esta opciòn se ejecuta un programa escrito en lenguaje de alto nivel, es decir, en la versiòn reducida de Pascal. El programa debe de haber sido creado con anterioridad y guardado en un archivo.

Supongamos que tenemos el siguiente programa guardado en un archivo cuyo nombre es SUMA.PAS :

```
Var
 i      : integer;
 limite : integer;
 suma   : integer;
Begin
 read limite;
 suma := 0;
 for i := 1 to limite do
  suma := suma + i;
 write suma;
End.
```

Al pedir la opciòn dos, el sistema preguntarà por el nombre del archivo que guarda el programa. Teclearemos SUMA.PAS.

Si el archivo no existe, aparecerà un mensaje de error; si el archivo existe, se procederà a su traducciòn y despuès a su ejecuciòn.

Durante la traducciòn, se muestra su equivalente en lenguaje ensamblador (resultado de la compilaciòn)

```
Var
i      : integer;
limite : integer;
suma   : integer;
Begin
read limite;                          LEEENT LIMIT
suma := 0;                            CARGAA 0
                                      GUARDA SUMA
for i := 1 to limite do               CARGAA 1
                                    1
                                      GUARDA I
                                      COMP LIMIT
                                      SALTMA 2
```

```
suma := suma + i;                    CARGAA SUMA
                                     SUMA I
                                     GUARDA SUMA
                                     CARGAA I
                                     SUMA 1
                                     SALTA 1
                                   2
write suma;                          ESCENT SUMA
End.                                 REGSIST
                                   I RPAL 1
                                   LIMIT RPAL 1
                                   SUMA RPAL 1
                                     FIN
```

y el equivalente de èste en lenguaje de màquina (resultado del ensamblado)

```
     LEEENT LIMIT          0 00110000 00000000 00110000
     CARGAA 0              3 00000000 00000000 00110110
     GUARDA SUMA           6 00000011 00000000 00110011
     CARGAA 1              9 00000000 00000000 00111001
   1
     GUARDA I             12 00000011 00000000 00101101
     COMP LIMIT           15 00011010 00000000 00110000
     SALTMA 2             18 00100100 00000000 00100111
     CARGAA SUMA          21 00000000 00000000 00110011
     SUMA I               24 00010000 00000000 00101101
     GUARDA SUMA          27 00000011 00000000 00110011
     CARGAA I             30 00000000 00000000 00101101
     SUMA 1               33 00010000 00000000 00111001
     SALTA 1              36 00100000 00000000 00001100
   2
     ESCENT SUMA          39 00110010 00000000 00110011
     REGSIST              42 01000000 00000000 00000000
   I RPAL 1               45 00000000 00000000 00000000
   LIMIT RPAL 1           48 00000000 00000000 00000000
   SUMA RPAL 1            51 00000000 00000000 00000000
     FIN
   0                      54 00000000 00000000 00000000
   1                      57 00000000 00000000 00000001
```

Se debe notar que en el programa en lenguaje ensamblador que la variable limite está truncada a cinco caracteres, quedando LIMIT. Todos los nombres de variables, constantes, etc. se conservan con un màximo de cinco caracteres.

Hay que tener cuidado, ya que si se tiene, por ejemplo, dos variables : limitearriba y limiteabajo, al ser compilado el programa que las contiene, ambas quedaràn como LIMIT, lo cual producirà errores inesperados.

Otra de las cosas que se observa es que no se hace distinciòn entre minùsculas y mayùsculas, ya que todo se convierte a mayùsculas. Por ejemplo, las siguientes referencias suma, SUMA, sUma, SuMA, etc. se considerarán a la misma variable.

Cabe señalar que en la compilaciòn de un programa, las variables de èste son automàticamente inicializadas a cero.

El listado del programa en binario (lenguaje de màquina) se encuentra precedido por un nùmero, el cual es la localidad de memoria en donde se encuentra el inicio de la palabra (instrucciòn o dato). Por ejemplo,

```
GUARDA SUMA              6 00000011 00000000 00110011
```

indica que en la palabra de memoria cuya direcciòn es 6 se encuentra almacenada la instrucciòn.

No se debe olvidar que el tamaño de una palabra en COSI es de tres bytes, por lo que el contador aumenta de tres en tres.

Al final del proceso de ensamblado, se pregunta si se desea que durante la ejecuciòn del programa se pare al comienzo de alguna instrucciòn; si la respuesta es afirmativa, se pedirà la localidad donde comienza la instrucciòn.

Si no se recuerda la localidad de inicio de la instrucciòn deseada, hay la opciòn de pedir que se vuelva a listar el resultado del proceso de ensamblado.

Para la ejecuciòn del programa una vez traducido, se presenta el procesador y parte de la memoria. Durante esta etapa, se puede observar el funcionamiento interno que lleva a cabo el procesador para la ejecuciòn de cada una de las instrucciones, su interacciòn con la memoria y el equipo perifèrico.

El acceso a la memoria y el movimiento entre los registros es por medio de bytes, y no de bit en bit como se observa durante la ejecuciòn, èsto ùnicamente es para efecto de presentaciòn.

Durante la ejecuciòn se tienen varias opciones ha escoger, las cuales aparecen en la parte baja de la pantalla.

F1 - Terminar

Finaliza la ejecución aún cuando ésta no haya terminado por sí sola.

F2 - Seguido/Parado

Da la opción de que la ejecución pare después de cada etapa del ciclo de control y continúe al presionar cualquier tecla, o de seguir sin pausa alguna; ésto es, al oprimir la tecla F2 si se encuentra en el modo seguido pasará al modo parado o viceversa. El modo inicial es el seguido.

F3 - Parar

Para la ejecución y continua al oprimir cualquier tecla.

F4 - Ver localidad

Detiene la ejecución para observar el contenido de una localidad de memoria que desee el usuario. Como ayuda se presenta una tabla conteniendo las variables del programa que está siendo ejecutado, junto con sus respectivas localidades de memoria en caso de que el usuario desee observar el valor de una varible.

Estas opciones sólo se pueden invocar al presionar la tecla de función deseada después de la señal auditiva; ésta suena al final de cada ciclo de control.

Durante la ejecución, en la parte superior de la pantalla se observa la instrucción a nivel de máquina que está siendo ejecutada, así como su equivalente en lenguaje ensamblador y en su caso en lenguaje de alto nivel.

Si en el programa existe una instrucción de lectura, por ejemplo : LEEENT LIMIT, entonces, del lado izquierdo de la pantalla aparecerá la imágen de una computadora, la cual simula ser COSI que va a leer el valor de la variable, el cual será tecleado por el usuario. Si en la entrada existe algún error, ya sea de tipo, de longitud, etc., aparecerá un mensaje de error y se dará por terminada la ejecución.

Si en lugar de una instrucción de lectura es una de escritura (ESCENT SUMA), se dibujará la computadora con el valor escrito en su pantalla.

Si durante alguna de las etapas : compilación, ensamblado o ejecución se llegase a encontrar un error, se desplegará un mensaje y se dará por terminado el proceso.

3. ENSAMBLADO -> EJECUCION

Realiza lo mismo que la opción anterior, a diferencia de que ésta ejecuta un programa escrito en lenguaje ensamblador y, por lo tanto, sólo realiza el ensamblado durante el proceso de traducción.

Por lo demás es similar.

4. REGRESAR AL SISTEMA OPERATIVO

Da por finalizado el uso del sistema "SEC" y regresa el control al Sistema Operativo.

Como se observa el manejo del sistema "SEC" no presenta mayor dificultad, por lo que con lo expuesto se puede hacer uso de él sin problema alguno.

BIBLIOGRAFIA

BIBLIOGRAFIA

1. INTRODUCCION MODERNA A LA CIENCIA DE LA COMPUTACION
   Les Goldschlager, Andrew Lister
   Prentice Hall, 1986, 1a. Edicibn

2. A VISIBLE ASSEMBLER FOR A COURSE IN INTRODUCTORY SYSTEM SOFTWARE
   Cloy Ezell
   SIGCSE BULLETIN, Vol. 17, No. 4, Diciembre 1985

3. A MODERN APPROACH TO TEACHING COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE PROGRAMMING
   William F. Decker
   SIGCSE BULLETIN, Vol. 17, No. 4, Diciembre 1985

4. MONITORING PROGRAM EXECUTION : A SURVEY
   Bernhard Plattner, Jurg Nievergelt
   COMPUTER, Vol. 14, Noviembre 1981

5. HYPERTEXT
   Janet Fiderio, Mark F., Michael B., Jeff C.
   BYTE, Vol. 13, No. 12, Octubre 1988

6. A HUMAN-FACTORS STYLE GUIDE FOR PROGRAM DESIGN
   Henry Simpson
   BYTE, Vol. 7, No. 4, Abril 1982

7. FUNDAMENTAL CONCEPTS OF PROGRAMMING SYSTEMS
   Jeffrey D. Ullman
   Addison-Wesley, 1976

8. MICROCOMPUTER ARCHITECTURE AND PROGRAMMING
   John F. Wakerly
   Wiley, 1981

9. INTRODUCTION TO COMPUTER SYSTEMS USING THE PDP-11 AND PASCAL
   Glenn H. MacEwen
   McGrawn Hill, 1980

10. SYSTEM SOFTWARE
    Leland L. Beck
    Addison-Wesley, 1985

11. FROM HARDWARE TO SOFTWARE
Graham Lee
McMillan Press, 1982

12. COMPUTER ORGANIZATION AND PROGRAMMING
C. William Gear
McGraw Hill, 1980, 3a. Edición

13. COMPUTACION
Larry Gonick
Haria, 1985

14. INTRODUCTION TO COMPUTER ORGANIZATION AND DATA STRUCTURES
Harold S. Stone, Daniel P. Siewiorek
McGraw Hill, 1975

15. COMPUTERS AND COMPUTATIONS
David J. Kuck
Wiley, 1978

16. GLOSARIO DE COMPUTACION
Alan Freedman
McGraw Hill, 1983, 1a. Edición

17. PRINCIPLES OF COMPILER DESIGN
Alfred V. Aho, Jeffrey D. Ullman
Addison Wesley, 1975

18. ALGORITHMS + DATA STRUCTURES = PROGRAMS
Niklaus Wirth
Prentice Hall, 1976

19. THE THEORY AND PRACTICE OF COMPILER WRITING
Jean-Paul Tremblay, Paul G. Sorenson
McGraw Hill, 1985

20. COMPILADORES
Luis Legarreta Garciadiego
Fundación Arturo Rosenblueth

21. OPERATING SYSTEMS, A SYSTEMATIC VIEW
William S. Davis
Addison-Wesley, 1987, 3a. Edición

GLOSARIO DE TERMINOS

GLOSARIO

- Acumulador (A) : Acumulator (A)
- Anàlisis lèxico : Lexical analysis
- Anàlisis semàntico : Semantic analysis
- Analizador lèxico : Scanner
- Analizador sintàctico : Parser
- Analizador sintàctico Desde Abajo : Bottom-Up Parser
- Analizador sintàctico Desde Arriba : Top-Down Parser
- Apuntador de Pila (AP) : Stack Pointer (SP)
- Cargador primario : Boot
- Compilador : Compiler
- Contador de Localidad (CL) : Location Counter (LC)
- Contador de Programa (CP) : Program Counter (PC)
- Descenso recursivo : Recursive-descent
- Dirección Real (DR) : Target Address (TA)
- Directivas del ensamblador : Assembler directives
- Disco duro : Hard disk
- Disco flexible : Floppy disk
- Ensamblador : Assembler
- Ensamblador de Carga y Ejecuta : Load and Go Assembler
- Ensamblador de Dos Pasos : Two-pass Assembler
- Entrada/Salida (E/S) : Input/Output (I/O)
- Generación de código : Code generation
- Gramàtica : Grammar
- Lenguaje de Alto Nivel : High-Level Language
- Lenguaje de Máquina : Machine Language
- Lenguaje de Programación : Programming Language
- Lenguaje Ensamblador : Assembly Language
- Memoria de Lectura-Escritura (MLE)
  : Read-Write Memory (RWM o RAM)
- Memoria de Lectura Exclusiva (MLEX)
  : Read-Only Memory (ROM)
- Memoria de Lectura Exclusiva Programable (MLEX/P)
  : Programmable Read-Only Memory (PROM)
- Memoria de Lectura Exclusiva Reprogramable (MLEX/REP)
  : Erasable Programmable Read-Only Memory (EPROM)
- Pista : Track
- Programa Fuente : Source Program
- Programa Objeto : Object Program
- Referencia adelantada : Forward reference

- Registro de Condición (o de Control) (RC)
  : Status Word (SW)
- Registro de Dato de Memoria (RDAM)
  : Memory Data Register (MDR)
- Registro de Dirección de Memoria (RDIM)
  : Memory Address Register (MAR)
- Registro de Dirección Efectiva (RDE)
  : Effective Address Register (EAR)
- Registro de Indice (X) : Index Register (X)
- Registro de Instrucción (RI)
  : Instruction Register (IR)
- Simbolo terminal : Terminal symbol
- Simbolo no-terminal : Non-terminal symbol
- Tabla de Literales : Literal Table
- Tabla de Simbolos : Symbol Table
- Tiempo de Compilación : Compile Time
- Tiempo de Ejecución : Run Time
- Tiempo de Ensamblado : Assembly Time
- Traductor : Translator
- Unidad Aritmética y Lógica (UAL)
  : Arithmetic Logic Unit (ALU)
- Unidad Central de Procesamiento (UCP)
  : Central Processing Unit (CPU)
- Unidad de Control : Control Unit
- Unidad de Disco : Drive Disk