Universidad Católica Andrés Bello
Facultad de Ingeniería
Escuela de Ingeniería Informática y Telecomunicaciones

Trabajo de Ascenso

Ensambladores, Enlazadores y Cargadores:

Una introducción basada en un computador hipotético de enseñanza

Presentado por:
Lic. Juan Carlos Núñez Castro
C.I. 5.539.048
ante la Universidad Católica Andrés Bello
para optar al escalafón de Asistente.

Caracas, Marzo 2006

INDICE

AGRADECIMIENTOS

A Dios , mi amigo, quien con su infinita sabiduría ha sabido poner en mi camino la experiencia necesaria (a pesar de mi terquedad) para aprender y mejorar. Perdóname Dios, si a veces fallo. Gracias por tu inspiración.

RESUMEN

En este trabajo se presentan los principios de funcionamiento de los ensambladores, enlazadores y cargadores. Para tal fin, se desarrolla el diseño de la arquitectura de un computador básico con propósitos de enseñanza, y un lenguaje ensamblador de gramática sencilla para tal computador, los cuales sirven de apoyo para el estudios de los procesadores de lenguaje mencionados. Como tal, sirve de complemento a cursos de Arquitectura del Computador, Estructura del Computador y Traductores.

INTRODUCCIÓN

Este documento es un apoyo para la enseñanza de un cierto tipo de *procesadores de lenguaje*: los ensambladores, enlazadores y cargadores. Procesador de lenguaje es el nombre genérico que recibe cualquier aplicación cuya entrada es un lenguaje Los ensambladores, enlazadores y cargadores también forman parte del *software del sistema.* Este tipo de software conforma una colección de programas que soportan la operación de un computador, y hacen posible que un programador se concentre en la resolución de un problema, dejando a un lado los detalles internos de operación de la máquina. Otros programas de sistemas lo constituyen: Sistemas Operativos, Editores, Manejadores de Base de Datos, Compiladores y Protocolos de Comunicación. Todos tienen en común un cierto nivel de dependencia del computador sobre el cual operan.

Esta dependencia ocasiona que en su estudio haya que incluir el análisis de computadores reales. Sin embargo, la mayoría de las máquinas reales tienen características inusuales e incluso únicas. Esto hace difícil el distinguir entre las características realmente fundamentales de un programa de sistemas, y las que dependen únicamente de las peculiaridades de una máquina determinada. Para evitar este problema, en este trabajo se diseña un *computador hipotético para la enseñanza de los principios de funcionamiento de un Ensamblador*, el cual ha sido bautizado como HALC. Este diseño constituye un computador software que incluye las características mas frecuentes de los computadores reales, evitando complejidades inusuales e irrelevantes. De esta forma, los conceptos centrales de un programa de sistema, y en particular de los ensambladores, enlazadores y cargadores se pueden distinguir de los detalles de implantación de una arquitectura determinada. Este enfoque proporciona al estudiante un punto de partida para iniciar el diseño de un software de sistemas en un computador nuevo o desconocido.

Una vez explicadas las bases del computador HALC, se desarrolla un lenguaje ensamblador de gramática sencilla para este computador. Sobre este lenguaje ensamblador, se diseña a continuación un ensamblador el cual permite ilustrar las características generales de este tipo de procesador de lenguaje.

El trabajo no estaría completo sin la inclusión de un análisis del funcionamiento general de los enlazadores y los cargadores. Para ello, apoyados en el código objeto que genera como salida el ensamblador desarrollado, se diseña un enlazador-cargador el cual permita estudiar el funcionamiento de "la amalgama" que permite el desarrollo de las técnicas de programación estructurada y de la programación orientada a objeto.

El estudio de estos temas constituye parte fundamental de la estructura curricular de los programas de ciencias de la computación e ingeniería de informática, así como del contenido de cursos de Arquitectura de Computadores o Estructura del Computador, los cuales se apoyan en la enseñanza del lenguaje ensamblador como herramienta de primera mano que permita un acercamiento a la organización de "la máquina".

En Venezuela, no obstante, es notoria la escasez de información en estos tópicos, y el mercado adolece de literatura que apoye al estudiante en el estudio de los mismos. De esta falencia, nace la intención de este trabajo: llenar ese vacio y apoyar la enseñanza de cursos donde se desarrollan estos temas, sentando al mismo tiempo las bases para el desarrollo de literatura mas especializada alrededor de este tipo de procesadores de lenguaje.

Un trabajo mas exhaustivo en estos tópicos debería desarrollar aspectos como el diseño de macroprocesadores, manejo de literales y expresiones en lenguajes ensambladores, técnicas de hashing para el acceso a tablas y enlace y carga dinámica. No deja de inquietarme el no desarrollarlos; son tópicos sobre los cuales espero continuar investigando, pero cuyo desarrollo a fondo escapan de lo que se espera impartir en los cursos mencionados.

Este trabajo recoge mi experiencia de mas de 14 años programando aplicaciones en lenguaje ensamblador (el único lenguaje en el cual he programado en "el mundo real"), así como en la enseñanza, tanto en la UCAB como en la UCV, de varios cursos en las cátedras de Estructura del Computador, Arquitectura del Computador y Sistemas de Computación.

Espero que, como tal, coadyuve en el proceso de formación de estudiantes de computación e informática.

CAPITULO I
EL COMPUTADOR HALC

En nuestra opinión, la definición mas acertada de un computador, lo define como *una colección de algoritmos y estructuras de datos capaces de almacenar y ejecutar programas*. Como tal, una vez definido este conjunto de algoritmos y estructuras y datos, un computador puede construirse como un dispositivo físico utilizando un conjunto de circuitos integrados, en cuyo caso es denominado *computador real*. No obstante, es posible también la implantación de tales algoritmos y estructura de datos a través de un conjunto de programas que se ejecuten en otra computadora, en cuyo caso el resultado final es un *computador simulado por software o máquina virtual*. Una máquina virtual es entonces un programa software que se comporta como un procesador real y que posee su propio lenguaje máquina. Se le llama "virtual" porque no existe realmente un chip hardware para la ejecución del lenguaje de la máquina virtual; por lo contrario, el funcionamiento de la máquina es simulado mediante software.

En función de la definición anterior, se desarrolla a continuación una especificación de un computador hipotético, el cual ha sido bautizado como HALC (rememorando al *cerebro* del clásico "2001 Odisea del espacio": **H**euristic **AL**gorithmic **C**omputer, aunque en este film el nombre real del computador era HAL 9000). El propósito de este diseño es presentar una estructura que apoye la enseñanza de aquellos procesadores de lenguaje que presenten una fuerte dependencia en relación a las características de un computador, como Ensambladores, Enlazadores y Cargadores.

HALC ilustra los conceptos y características de hardware mas frecuentes, tratando de evitar las particularidades que suelen encontrarse en máquinas reales. Su arquitectura no pretende ser novedosa; lo importante en ella es su carácter pedagógico, por lo que alguna de sus características pueden no justificarse en la práctica.

Estructura de HALC.

Una manera de familiarizarse con un nuevo computador, consiste en plantearse una serie de preguntas en relación a lo que se debería conocer para programarla:

- Memoria. ¿Cuál es la unidad básica de memoria, que capacidad posee y que esquema de direccionamiento emplea?.

- Registros. ¿Cuántos registros hay?. ¿Qué capacidad tienen?. ¿Cuáles son sus funciones y como se interrelacionan?. ¿Cuáles están a la disposición del programador y cuales no?.
- Datos. ¿Qué tipos de datos provee el computador?. ¿Cómo se almacenan los datos?.
- Instrucciones. ¿Cuál es el repertorio de instrucciones de máquina del computador?. ¿Cuál es el formato de las instrucciones?. ¿Cómo se almacenan en memoria?.
- Entrada/Salida: ¿Cómo es la comunicación la comunicación con los dispositivos periféricos?. ¿De qué tipos de dispositivos de entrada/salida se disponen?.

En función de estas interrogantes, se presenta a continuación la estructura del computador hipotético HALC

1. Memoria: El esquema de direccionamiento es de 32 bits. La unidad básica de direccionamiento es el byte cuyo ancho es de 8 bits. A diferencia de muchos computadores reales, en los cuales una palabra tiene una alineación determinada, en HALC una palabra esta formada por cuatro bytes (32 bits) consecutivos cualesquiera, sin importar su frontera. Los bytes de una palabra se enumeran ascendentemente de izquierda a derecha *(big endian)*; sin embargo, los bits dentro de un byte se enumeran ascendentemente de derecha a izquierda (*bit little endian*). La dirección de una palabra será igual a la dirección del byte de menor numeración (el byte mas a la izquierda).

Dirección de la palabra	Byte 0	Byte 1	Byte 2	Byte 3
0	7.....210	7.....210	7.....210	7.....210
4	7.....210	7.....210	7.....210	7.....210
8	7.....210	7.....210	7.....210	7.....210
12 (0x0C)	7.....210	7.....210	7.....210	7.....210

El paradigma de memoria de HALC es el de una sola memoria lineal, a diferencia de algunos procesadores que segmentan el espacio de direcciones en distintas.

2. HALC es una máquina de dos direcciones en la que el primer operando siempre es un registro o es el tope de una pila. Presenta seis tipos de direccionamiento:

    a. Relativo a registro base RB (TD = 000), el cual constituye el tipo de direccionamiento por defecto.
    b. De registros RG (TD = 001).
    c. Indirecto por registro IR (TD = 010).
    d. Relativo indexado RX (TD = 011).
    e. Inmediato IN (TD = 100).
    f. Directo D (TD = 101).

    El tipo de direccionamiento, dado que el primer operando siempre es un registro, solo aplica y tiene significado para el segundo operando, el cual puede ser una posición de memoria, un registro o un operando inmediato.

    Puede parecer extraño que todos estos tipos de direccionamiento coexistan en una misma arquitectura, y de hecho es así. En tal sentido, debe tomarse en cuenta que HALC fue diseñado para propósitos de enseñanza, y muchas de sus características se añaden con la finalidad de ilustrar conceptos de programación de sistemas.

3. Registros: Solo se presentan los registros visibles al programador; estos están conformados por:

a. Doce registros de propósito general con una longitud de 32 bits, e identificados por las direcciones de registros 0x00 - 0x0B.
b. Un registro base de 32 bits cuya dirección de registro es 0x0C.
c. Un registro índice de 32 bits cuya dirección de registro es 0x0D.
d. Un registro apuntador al tope de una pila de 32 bits, y cuya dirección de registro es 0x0E. Este registro es denominado simbólicamente como SP.
e. Un registro de intercambio de 32 bits. Todas las operaciones de entrada/salida involucran el byte mas a la derecha de este registro. Su dirección es 0x0F. Este registro es denominado simbólicamente como RIO.
f. Palabra de Estado SW: Registro de 32 bits que no necesita ser direccionado explícitamente, y almacena información de control como el código de condición CC. Este constituye un campo de dos bits que indica si el resultado de una operación aritmética fue igual, menor o mayor a cero. Igualmente, al comparar dos operandos A y B el CC indicará si ambos operandos son iguales, o si el primero es menor o mayor que el segundo.

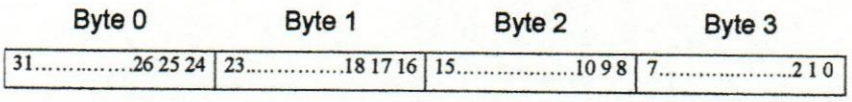
$$CC = 0 \text{; si } AC = 0 \text{ o si } A = B$$
$$CC = 1 \text{, si } AC < 0 \text{ o si } A < B$$
$$CC = 2 \text{, si } AC > 0 \text{ o si } A > B.$$

Las palabras de estado de la mayoría de los computadores reales almacenan mayor cantidad de información. Para este estudio sin embargo, es suficiente con el campo señalado.

g. Contador de Instrucciones PC: Registro de 32 bits que mantienen la dirección de la próxima instrucción a ser ejecutada. Es direccionado explícitamente por toda instrucción que involucre una transferencia de control.

4. Tipos de datos:

a. Enteros de 32 bits, representados en complemento a dos y almacenados en una palabra. Los bits de un entero se enumeran de 0 a 31 desde el bit 0 del byte 3 hasta el bit 7 del byte 0.

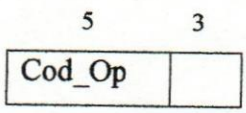


b. Carácter: Se usa la codificación ASCII para la representación de caracteres.
c. Booleanos: En teoría, un bit sencillo puede ser usado para representar un valor booleano; en la práctica se usa el byte para representar este tipo de datos ya que HALC no presenta instrucciones para la manipulación de bits individuales.
d. Dirección: Representan direcciones de memoria y permiten la implantación de apuntadores.

5. La pila: Cada entrada en la pila es de 4 bytes, y el tope de la misma se encuentra apuntado por el registro SP. La pila crece de las posiciones bajas de memoria hacia las más altas. El diseño no contempla condiciones de desbordamiento (overflow) sobre el área de la pila.

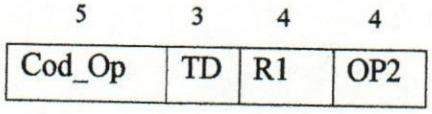
6. Formato de las instrucciones: A continuación se describe el formato de las instrucciones de HALC. En su descripción, se usa el símbolos *R1* para señalar al primer operando (que siempre es un registro), y el símbolo OP2 para hacer referencia al segundo operando, el cual puede ser, según la instrucción, un registro, una posición de memoria, el contenido de una posición de memoria o un dato inmediato. **Cod_Op** es un campo de 5 bits que contiene el código de operación de la instrucción; mientras que el campo **TD**, de 3 bits, indica el método o tipo de direccionamiento que se usará en la ejecución de la instrucción para referenciar al segundo operando.

- Formato F1: Se usa para aquellas instrucciones que no requieren de operando. Su longitud es de un byte.



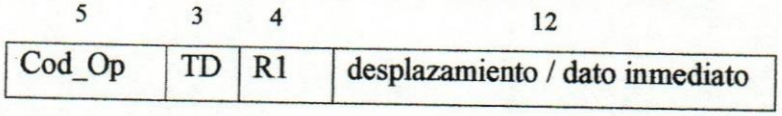
Los bits 0, 1 y 2 no se usan, por lo que su contenido se asumirá igual a cero.

- Formato F2: Se usa para aquellas instrucciones con direccionamiento de registro o indirecto con registro. Su longitud es de dos bytes.



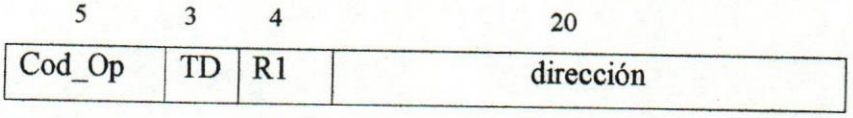
El segundo operando siempre es un registro. Si la instrucción hace referencia a un solo operando, este será siempre el segundo, y el valor del campo R1 se asume igual a cero.

- Formato F3: Se usa para aquellas instrucciones con direccionamiento relativo, relativo/indexado e inmediato. Su longitud es de tres bytes



Si la instrucción hace referencia explícita a un solo operando, este será siempre el segundo, y el valor del campo R1 se asume igual a cero.

- Formato F4: Se usa para instrucciones con direccionamiento directo o absoluto. Su longitud es de cuatro bytes.



Al igual que con formatos anteriores, si la instrucción hace referencia explícita a un solo operando, este será siempre el segundo, y el valor del campo R1 se asume igual a cero.

7. Subsistema de entrada/salida: HALC implanta un subsistema de entrada/salida aislado en memoria: el CPU tiene instrucciones distintas y especializadas en la comunicación con los periféricos, y cada una de estas instrucciones direccionan implícitamente al registro de propósito general 0x00 y al registro de intercambio 0x0F.

La arquitectura de entrada/salida de HALC es bastante sencilla. Se supone la existencia de tres periféricos: uno de entrada, uno de salida y otro de almacenamiento, todos orientados a carácter: cada uno transfiere un byte a la vez el cual es cargado en el byte 3 (el byte "mas a la derecha") del registro de intercambio RIO. Cada dispositivo esta identificado por un código de 8 bits; la identificación del dispositivo objeto se carga en el byte 3 del registro 0x00:

- Dispositivo de entrada: 0xF1.
- Dispositivo de salida: 0xF2.
- Dispositivo de almacenamiento: 0xF3.

Además de ser aislado en memoria, el subsistema de entrada/salida es orientado a programa; es decir, el programa, una vez iniciada una operación de E/S, debe chequear constantemente la culminación de la misma. Se dispone de un total de cuatro instrucciones de E/S: Test Device TD, Test I/O, Read Device RD, y Write Device WD. Las mismas son detalladas en la sección siguiente.

Instrucciones de Máquina.

El funcionamiento de un CPU está determinado por las instrucciones que ejecuta. Estas instrucciones se denominan *instrucciones de máquina*. Al conjunto de instrucciones distintas que puede ejecutar el CPU se le denomina *repertorio o conjunto de instrucciones de máquina*. HALC presenta un repertorio de instrucciones bastante simple, pero a su vez completo, que le permite al programador ejecutar operaciones lógicas, aritméticas y de comparación, transferir datos entre el CPU y la memoria, transferir el control del CPU a otras secciones de código, y ejecutar operaciones de entrada/salida.

HALC es una máquina de dos direcciones; sin embargo, el primer operando siempre es un registro, por lo que la instrucción no necesita indicar el tipo de direccionamiento que aplica para este operando. Una misma instrucción puede usar reglas distintas para hacer

referencia al segundo operando, por lo que se usa un campo (TD) para señalar el modo de direccionamiento a usar en este caso.

La tabla a continuación describe cada una de las instrucciones que constituyen el repertorio del computador HALC. A pesar de que se usan 5 bits para codificar el código de operación de una instrucción, el mismo es representado mediante dos dígitos hexadecimales. R1 representa al primer operando de la instrucción, y OP2 al segundo operando, el cual, como ya se explicó, puede ser un registro, un operando inmediato, o un operando en memoria. En el caso de las instrucciones de salto, DE indica la dirección efectiva una vez aplicado el modo de direccionamiento al segundo operando.

Mnemónico	código de Operación	Direccionamiento	Acción
ADD	00 (00000)	RB, RG, IR, RX, IN, D	R1 = R1 + OP2
AND	01 (00001)	RB, RG, IR, RX, D	R1 = R1 & OP2
CLEAR	02 (00010)	RG	OP2 = 0x00000000
CMP	15 (10101)	RB, RG, IR, RX, D	Compara R1 con OP2 e inicializa el CC
DEC	0C (01100)	RG	OP2 = OP2 - 1
HLT	1D (11101)	N/A	Detiene el funcionamiento del computador
INC	0B (01011)	N/A	OP2 = OP2 + 1
JE	04 (00100)	RB, D	Si CC = 00 entonces PC = DE
JGT	05 (00101)	RB, D	Si CC = 10 entonces PC = DE
JLT	06 (00110)	RB, D	Si CC = 01 entonces PC = DE
JNE	16 (10110)	RB, D	Si CC ≠ 00 entonces PC = DE
JSUB	07 (00111)	RB, D	MEM (SP) = PC, SP = SP + 4, PC = DE
JUMP	03 (00011)	RB, D	PC = DE
LA	0D (01101)	RB, IR, RX, D	R1 = DE
LB	13 (10011)	N/A	Reg. Base (0x0C) = dirección de esta instrucción
LDCH	17 (10111)	RB, IR, RX, D	$R1_{0-7}$ = OP2, $R1_{8-31}$ = 0s
LOAD	08 (01000)	RB, RG, IR, RX, D, IN	R1 = OP2
OR	09 (01001)	RB, R, IR, RX, D	R1 = R1 | OP2
POP	0F (01111)	RB, RG, RX, D	OP2 = MEM(SP), SP = SP - 4
PUSH	0E (01110)	RB, RG, RX, D	MEM (SP) = DE, SP = SP + 4
RD	1A (11010)	N/A	Lee un carácter desde el dispositivo indicado por Reg. 0x00
RSUB	0A (01010)	N/A	PC = MEM(SP), SP = SP - 4
STCH	18 (11000)	RB, IR; RX, D	OP2 = $R1_{0-7}$
STORE	14 (10100)	RB, RG, IR, RX, D	OP2 = R1
SUB	10 (10000)	RB, RG, RX, IN, D	R1 = R1 - OP2
SYSCALL	12 (10010)	IN	Reg. 0x00 = OP2, generar interrupción
TD	19 (11001)	N/A	Chequea el status del dispositivo direccionado por el Reg. 0x00

Mnemónico	código de Operación	Direccionamiento	Acción
TIO	1C (11100)	N/A	Indica si la operación de E/S en curso terminó
WD	1B (11011)	N/A	Escribe un carácter en el dispositivo indicado por Reg. =x00

Las instrucciones de entrada/salida, por interactuar con un subsistema al que normalmente se accede a través de un API, no son familiares para un programador común, por lo que merecen algo mas de explicación. En todas estas instrucciones, el registro 0x00 contiene la identificación del periférico involucrado en la operación de E/S.

La instrucción TD (Test Device) chequea el status del dispositivo señalado e inicializa el código de condición según el resultado de este chequeo:

CC = 00 ➔ dispositivo disponible.
CC = 01 ➔ dispositivo ocupado.
CC = 10 ➔ dispositivo fuera de servicio.

La instrucción RD (Read Device) lee un carácter desde el dispositivo indicado por el registro 0x00 y lo carga en los bits 0 - 7 del registro de intercambio RIO, colocando al mismo tiempo los bits 8 – 31 en cero. Por su parte, la instrucción WD (Write Device) escribe en el dispositivo indicado, el contenido del byte más a la derecha del registro RIO.

La instrucción TIO (Test I/O) permite cotejar si la operación de entrada/salida en curso finalizó, y el status de culminación de la misma, inicializando para tal fin el código de condición:

CC = 00 ➔ entrada /salida culminada exitosamente.
CC = 01 ➔ entrada/salida culminada con errores.
CC = 10 ➔ entrada/salida en desarrollo.

CAPITULO II
El LENGUAJE ENSAMBLADOR HAL.

Traductores y Ensambladores.

Un CPU ejecuta (o interpreta) instrucciones de máquina. Estas instrucciones son, básicamente, números binarios almacenados en la memoria del computador. Si un programador desea programar directamente en lenguaje de máquina, necesitará cargar sus programas como datos binarios. Así, un programa en lenguaje de máquina lucirá como

Dirección	Contenido
101	0010 0010 0000 0001
102	0001 0010 0000 0010
103	0001 0010 0000 0011
104	0011 0010 0000 0100
...	...
201	0000 0000 0000 0010
202	0000 0000 0000 0011
203	0000 0000 0000 0100
204	0000 0000 0000 0000

Este esquema de programación, si bien corresponde a lo que la máquina espera decodificar y ejecutar, no resulta apropiado para la codificación general de aplicaciones. Una mejora significativa implica hacer uso de nombres simbólicos o nemotécnicos para hacer referencia a los códigos de operación. De igual forma, cada referencia directa a una posición de memoria, pudiera ser sustituida por una dirección simbólica. Esto ultimo resulta conveniente porque, por lo general, mientras se escribe un programa no se conoce con exactitud la posición de los datos en memoria; así, se estaría independizando, en tiempo de programación, una estructura de datos, escalar o vectorial, de la posición de memoria de memoria que esta ocupa. En función de esta posibilidad, el programa anterior puede entonces lucir como

Etiqueta	Operación	Operando
FORMUL	LDA	I
	ADD	J
	ADD	K
	STORE	N
...	...	
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

Hecho esto, el programa anterior deberá ser traducido por un *traductor*, el cual no solo realice la traducción de operaciones simbólicas, sino además asigne direcciones de memoria a las direcciones simbólicas. El cómo se realiza esta traducción, constituye uno de los objetivos de este trabajo.

Un programa que convierte un programa escrito en algún lenguaje de programación a otro lenguaje, se denomina **traductor**. El lenguaje en el cual el programa estaba originalmente codificado recibe el nombre de *lenguaje fuente*, y el lenguaje al cual es traducido se denomina *lenguaje objeto*. La traducción es necesaria cuando no se dispone de un procesador capaz de ejecutar o interpretar instrucciones codificadas en algún lenguaje fuente. Si la traducción se realiza correctamente, tanto el programa fuente original como el programa objeto resultante, son funcionalmente equivalentes.

Se tienen dos tipos de traductores. Cuando el lenguaje fuente es, en esencia, una representación simbólica de instrucciones de máquina y direcciones de memoria, el traductor es denominado **ensamblador**, y al lenguaje fuente se le conoce como *lenguaje ensamblador*. Cuando el lenguaje fuente es un lenguaje de alto nivel como C o C++, y el lenguaje objeto en un lenguaje de máquina o una representación simbólica del mismo, el traductor es llamado **compilador**.

Resulta interesante entender que la frase *lenguaje ensamblador* difiere semánticamente de frases como *lenguaje C o lenguaje Java*, ya que el termino ensamblador no hace referencia a un solo lenguaje determinado: un lenguaje ensamblador se basa en las instrucciones de máquina de un procesador especifico, por lo que existen muchos lenguajes ensambladores, al menos uno para cada procesador (a diferencia de un solo lenguaje C, p.e).

Esto último trae como consecuencia para el programador que migrar de un procesador a otro, implique conocer un nuevo lenguaje de programación. Afortunadamente, la mayoría de estos lenguajes siguen el mismo patrón básico; por lo tanto, conocer los principios de programación en algún lenguaje ensamblador, facilita la comprensión de otros lenguajes de este tipo.

La programación en lenguaje ensamblador presenta ventajas obvias sobre la programación en lenguaje de máquina. El uso de nombres y direcciones simbólicas son de gran ayuda para el programador, el cual puede recordar mas fácilmente nemónicas como ADD, LOAD y STORE, por ejemplo, que los códigos de operación binarios de tales instrucciones. Igualmente, el uso de direcciones simbólicas evita que el programador asigne direcciones de memoria a sus datos y le permite hacer referencia a las mismas de una manera mas amigable.

Si bien es cierto que la programación en lenguajes de alto nivel resulta mas sencilla que en lenguaje ensamblador, esta ultima permite un acceso total a todas las características e instrucciones del computador, lo que no es posible alcanzar en lenguajes como C y Java. Esto permite el desarrollo de aplicaciones altamente sensibles a características de rendimiento que necesiten acceso a elementos de la arquitectura no disponibles en lenguajes de alto nivel.

Finalmente, la programación en lenguaje ensamblador le permite al estudiante de ciencias de la computación e ingeniería, tener un contacto directo y de primera mano con la organización interna de un computador, "tocar con las manos un computador en sus entrañas". Todas estas consideraciones explican el interés que todavía existe en los

ensambladores, y el porque su estudio debe ser incluido dentro de los currículo de estudios en computación y carreras afines.

El lenguaje ensamblador HAL.

Un lenguaje de programación se define mediante un conjunto de reglas. Para un lenguaje ensamblador, estas reglas especifican los símbolos que pueden utilizarse y como pueden combinarse para formar una *línea de código*. Esta ultima constituye la unidad básica de un programa en lenguaje ensamblador: un ensamblador procesa líneas de código.

HAL es un lenguaje ensamblador que hemos diseñado para el computador HALC. Como tal, presenta una estructura bastante simple para su línea de código, en la cual la amigabilidad para con el programador no es una meta, sino mas bien lo es, el intentar que su diseño permita ilustrar la estructura interna de un ensamblador. así, la codificación de líneas de código en este lenguaje obedece a reglas poco flexibles (la flexibilidad permite la amigabilidad pero a su vez genera complejidad).

Como lenguaje, HAL es lo que se conoce como un *lenguaje ensamblador puro*; es decir un lenguaje en el cual cada instrucción de un programa produce exactamente una instrucción de máquina y cuya traducción siempre genera código de máquina como salida. Esto lo diferencia de los llamados *ensambladores tradicionales*, los cuales proveen facilidades para la definición de subrutinas abiertas o macros; una macro se expande en varias instrucciones de máquina. Adicionalmente, a diferencia de los llamados *ensambladores de alto nivel* High Level Assembler HLA, HAL no provee estructuras de control propias de lenguajes de alto nivel.

A continuación, se describen las características de HAL como lenguaje ensamblador.

1. Instrucciones.

El lenguaje ensamblador HAL consta de dos tipos básicos de sentencias:

- Directrices o pseudo instrucciones; constituyen ordenes para el ensamblador mismo, por ejemplo, para que reserve memoria para la definición de una variable, o para indicar el fin del programa fuente en lenguaje ensamblador.
- Instrucciones; instrucciones de máquina codificadas en un lenguaje simbólico, y cuyos operandos también son simbólicos. Este conjunto de instrucciones fueron detalladas en el capitulo anterior.

2. Codificación de Instrucciones.

En HAL una instrucción esta compuesta por cinco campos, cada uno de ellos separados por uno o más blancos:

- Etiqueta: Permiten asignarle un nombre simbólico a posiciones de memoria que contengan tanto instrucciones como datos. Su longitud es de ocho caracteres y debe comenzar en la columna 1 y con un carácter alfabético. Este campo es opcional y solo se necesita si la instrucción o dato es referenciada luego por otra instrucción en este o en otro programa (en cuyo caso ambos programas deben estar enlazados).
- Operación: En este campo se especifica el mnemónico de una instrucción de máquina o el nombre de una pseudoinstrucción. El término mnemónico proviene del griego y significa *perteneciente a la memoria*, y eso es lo que es esencia: una ayuda a nuestra memoria. Este campo siempre es obligatorio, y le indica al ensamblador qué instrucción necesita ser ensamblada, o bien qué directriz debe ser procesada.
- Tipo de direccionamiento: Señala el tipo de direccionamiento deseado para el segundo operando de una instrucción. Este campo es necesario ya que el computador HALC, a diferencia de otros computadores, usa un mismo código de operación con diferentes tipos de direccionamiento.
- Operandos: En este campo se codifican los operandos simbólicos de la instrucción. La codificación de valores numéricos se realiza en el sistema decimal, a no ser que explícitamente se señale lo contrario. El número de

valores codificados en este campo depende del mnemónico: una instrucción puede tener cero, uno o dos operandos.

- Comentarios: Ocupan solo una línea, pero pueden continuarse en otra línea codificando % en la primera columna. Son solo de ayuda para el programador y el ensamblador los ignora.

Ejemplo:

etiqueta	operación	TD	operando	comentario
LAZO	LOAD	RX	R1,ARRAY1	cargar RX

3. Pseudo Instrucciones.

El lenguaje HAL posee las siguientes directrices para el ensamblador:

- START: Permite indicarle al ensamblador la dirección lógica de comienzo del programa en lenguaje ensamblador, así como el nombre del mismo.

  Ejemplo: La línea siguiente define a un programa cuyo nombre es PROGA, y su dirección lógica de comienzo es 100.

  PROGA START 100

- END: Indica el fin del programa fuente en lenguaje ensamblador y permite además señalar el punto de entrada al programa. Si no se especifica operando alguno, se asume que el punto de entrada será igual a la dirección de comienzo.
- BASE: Le indica al ensamblador cual es la dirección base que se usará en tiempo de ensamblaje para el cálculo de desplazamientos. Si se usa el símbolo "*" se asume como dirección base el contenido actual del contador de localizaciones.

- BYTE: Genera una constante de tipo caracter o tipo hexadecimal, que ocupa tantos bytes como sean necesarios para representarla.

  Ejemplo: La codificación de la proposición

  AUX1 BYTE C'ABC'

  genera una constante de tres bytes, y el contenido de cada bytes son los caracteres A, B y C.

  Ejemplo: La codificación de la proposición

  AUX2 BYTE X'0FC237DB'

  genera una constante hexadecimal de 4 bytes de longitud.

- WORD: Genera una constante entera de tamaño igual a una palabra. Si el operando tiene la forma A(símbolo), se genera una constante tipo dirección, es decir, una palabra cuyo contenido será la dirección del símbolo especificado.

  Ejemplo: La codificación de la proposición

  POINTER WORD A(DATO1)

  reserva una palabra y almacena en ella la dirección del símbolo DATO1.

- RESB: Reserva el número indicado de bytes para un área de datos.
- RESW: Reserva el número indicado de palabras para un área de datos.
- EQU: Permite la definición de un símbolo y la especificación de su valor. El valor siempre será una constante. Un uso común de esta directriz es el

establecimiento de nombres simbólicos que se puedan utilizar para mejorar la legibilidad de programas, en lugar de usar valores numéricos; esto incluye la definición de nombres mnemónicos para los registros.

- EXTDEF: Permiten señalar aquellos símbolos definidos en esta sección de control y que pueden ser usados en otra sección de control
- EXTREF: Permite señalar aquellos símbolos definidos en otras secciones de control y que serán importados para su uso local.

Ejemplo 1. Programa que calcula la suma de dos arreglos. Los números de línea se especifican solo a modo de referencia y no forman parte del programa. Se debe recordar que los valores numéricos de los operandos se expresan en el sistema de numeración decimal. .

Línea		**proposición fuente**			**Comentarios**
0	PROG1	START		0	identifica al programa
3	R0	EQU		0	
5	R1	EQU		1	
10	R2	EQU		2	
15	R3	EQU		3	
20	RX	EQU		13	
25		BASE		*	dirección base = lc
30	BEGIN	LB			carga registro base
35		CLEAR	RG	RX	
40		CLEAR	RG	R2	
45	LAZO	LOAD	RX	R1,ARRAY1	
50		ADD	RX	R1,ARRAY2	
55		STORE	RX	R1,ARRAY3	
60		INC		R2	
65		CMP		R2,SIZE	
70		JE		FIN	

```
75                ADD     IN    RX,4
80                JUMP            LAZO
85    FIN         SYSCALL IN    10        fin; llamar al SO
90    ARRAY1      RESW          10        reserva 10 palabras
95    ARRAY2      RESW          10
100   ARRAY3      RESW          10
105   SIZE        WORD          10        define constante entera
110               END                     fin de programa
```

Ejemplo 2. Programa que calcula el valor de la expresión m^n. Dado que HALC no incluye la multiplicación como operación aritmética, la misma se simula por un procedimiento de sumas sucesivas. Para ello, el programa define una subrutina la cual recibe como parámetros una tabla que incluye palabras para los operandos a multiplicar, así como para el resultado

Línea		**proposición fuente**			**Comentario**
0	EXPON	START		1000	dirección comienzo = 1000
3	R0	EQU		0	
5	R1	EQU		1	
10	R2	EQU		2	
15	R3	EQU		3	
20	RX	EQU		13	
25	M	WORD		5	
30	N	WORD		3	
35	TABLA	RESW		3	
40	RESULT	WORD		0	
45		BASE		*	
50	INICIO	LB			
55		CLEAR	RG	R1	
60		LOAD		R2,M	
65		CMP	RG	R1,R2	operando igual a 0?
70		JNE		GO1	

```
75                  STORE       R1,RESULT
80                  JUMP        FIN
85    GO1           LOAD        R2,N
90                  CMP     RG  R1,R2       exponente igual a 0?
95                  JNE         GO2
100                 INC         R1
105                 STORE       R1,RESULT
110                 JUMP        FIN
120   GO2           INC         R1
125                 CMP         R1,R2       exponente igual a 1?
130                 JNE         GO3
135                 LOAD        R2,M
140                 STORE       R2,RESULT
145                 JUMP        FIN
150   GO3           LOAD        R2,M
155                 STORE       R2,RESULT
160   LOOP          CLEAR       RX
165                 LOAD        R2,RESULT
170                 STORE   RX  R2,TABLA
175                 ADD     IN  RX,4
180                 LOAD        R2,M
185                 STORE   RX  R2,TABLA
190                 LA          R0,TABLA    pointer a tabla parametros
195                 JSUB        MULT
200   RET           ADD     IN  RX,4
205                 LOAD    RX  R2,TABLA
210                 STORE       R2,RESULT
215                 INC         R1
220                 CMP         R1,N
225                 JE          FIN
230                 JMP         LOOP
235   FIN           SYSCALL IN  10          fin de programa. ir al SO
240   %
245   %definición de una subrutina que calcula el producto entre dos números
```

```
250   %
255   MULT    PUSH    RG    R0      se salvan registros
260           PUSH    RG    R1
265           PUSH    RG    R2
270           PUSH    RG    RX
275           LOAD    IR    R1,R0   capturar primer operando
280           STORE         R1,OP1
285           ADD     IN    R0,4
290           LOAD    IR    R1,R0   capturar segundo operando
295           STORE         R1,OP2
300           LOAD    IN    R1,1
305           LOAD          R2,OP1
310   LOOP    ADD           R2,OP1
315           INC           R1
320           CMP           R1,OP2
325           JLT           LOOP
330           ADD     IN    R0,4
335           STORE   IR    R2,R0   almacena resultado en tabla
340           POP     RG    RX      se restauran registros
345           POP     RG    R2
350           POP     RG    R1
355           POP     RG    R0
360           RSUB                  retornar
365   OP1     RESW          1
370   OP2     RESW          1
375           END           INICIO  pto entrada = inicio
```

Ejemplo 3. Programa que lee un carácter desde el dispositivo de entrada y lo imprime en el dispositivo de salida.

```
Línea          Proposición fuente
0     INOUT    START    0
3     R0       EQU      0
```

```
5    REGIO      EQU        15
10              BASE       *
15              LDCH       R0,INDEV
20   INLAZO     TD                    dispositivo disponible?
23              JNE        INLAZO
25              RD
26   FINIO1     TIO                   fin e/s?
27              JE         GO
28              JL         FIN        errores de e/s?
30              JMP        FINIO
35   GO         LDCH       R0,OUTDEV
40   OUTLAZO    TD
45              JNE        OUTLAZO
50              WD
52   FINIO2     TIO
53              JG         FINIO2
55   FIN        SYSCALL    IN   10
60   DATO       RESB
65   INDEV      BYTE       X'F'1'
70   OUTDEV     BYTE       X'F2'
75              END
```

CAPITULO III
DISEÑO DEL ENSAMBLADOR HAL

Como ya se mencionó, un ensamblador es un programa que acepta un programa codificado en lenguaje ensamblador, y produce su programa binario equivalente. El programa simbólico de entrada se llama programa fuente, y el programa binario que resulta se llama programa objeto. El ensamblador opera sobre cadenas de caracteres y produce una interpretación binaria equivalente.

Los ensambladores son objetos interesantes, ya que tanto su operación como las características del lenguaje ensamblador dependen en gran medida de la arquitectura de un computador; características como tamaño de la palabra en memoria, formato de instrucciones, tipos de direccionamiento, entre otras, afectan la forma en que se codifican las instrucciones en lenguaje ensamblador, como son procesadas estas instrucciones por un ensamblador, y la definición general de muchas pseudoinstrucciones.

Para ejecutar la tarea básica de ensamblar instrucciones, un ensamblador debe manejar los símbolos que defina el programador. Un símbolo se define cuando este se escribe como etiqueta, y constituye generalmente una referencia simbólica a una posición de memoria. Un símbolo es usado cuando se le escribe como operando en una instrucción, y debe ser definido una sola vez, pero puede ser usado y referenciado múltiples veces. Por ejemplo, en el siguiente segmento

```
LAZO    ADD     R4,DATO1
        ...
        ...
        JUMP    LAZO
```

La instrucción ADD usa dos operandos, el primero de ellos es un registro el cual es referenciado simbólicamente por el símbolo R4; el segundo es el contenido de una posición de memoria, también referenciada simbólicamente con el símbolo DATO1. La misma

instrucción ADD es cargada en una posición de memoria cuyo dirección simbólica es LAZO. Esta instrucción es a su vez el objeto u operando de la instrucción JUMP. Una función clave de un ensamblador es la de reemplazar estas referencia simbólicas a memorias por direcciones de memoria, para lo cual debe mantener la traza de las posiciones donde, tanto instrucciones como datos, son cargadas. Para hacer esto, el ensamblador utiliza, como se explicará mas adelante, un *contador de localizaciones* y una *tabla de símbolos*.

Aunque cada máquina tiene un lenguaje ensamblador diferente, el proceso de ensamblado tiene bastantes similitudes entre los distintos computadores, lo que hace posible describir este proceso de manera general. Dada la generalidad de HALC, nos apoyaremos en él para ofrecer a continuación una introducción al diseño de este tipo de traductores.

La operación básica de un ensamblador.

Conceptualmente, todo ensamblador ejecuta su labor de ensamblaje sobre un espacio de direcciones lógico o hipotético, cuya dirección de comienzo se señala explícitamente en HAL a través de la pseudoinstrucción START. El ensamblador mantiene la traza de la memoria disponible dentro de este espacio de direcciones, y le va asignando a cada elemento del programa fuente (instrucciones y/o datos) posiciones libres dentro de esta región (tantas como se necesiten). Para tal fin, el ensamblador de HAL define una variable interna denominada ***Contador de Localizaciones LC*** (algunos mantienen mas de un LC), la cual siempre apunta a la siguiente posición disponible dentro del mencionado espacio de direcciones lógico. El valor inicial de esta variable es obviamente igual al operando de la pseudoinstrucción START ya comentada. Cada vez que se ensamble un elemento de un programa en lenguaje ensamblador, el LC verá incrementado su contenido según la longitud del elemento ensamblado. Cuando el elemento ensamblado tiene una etiqueta, es decir, define un símbolo, a este símbolo se le asignará como valor el contenido actual del LC. Esta dupla, <símbolo, valor del LC>, es colocada por el ensamblador en la *tabla de símbolos*. Por ejemplo, para procesar la siguiente línea de código

SUMAR ADD R3,A

el ensamblador almacena en la tabla de símbolos el símbolo SUMAR junto con el valor actual de LC, y además ensambla la instrucción. Ambas acciones constituyen actividades independientes, y son hechas por rutinas distintas del ensamblador, y, en muchos de ellos, inclusive en fases distintas del proceso de ensamblaje.

Como ya se comentó, la salida de un ensamblador es un programa binario equivalente al programa fuente de entrada. Este código objeto tiene cierta estructura, la cual es conocida por otros procesadores de lenguaje como enlazadores y cargadores, y aun otros traductores, que se ejecuten sobre el mismo Sistema Operativo. Así, bajo Linux, por ejemplo, el código objeto manejado por todos los procesadores de lenguajes que corren bajo este Sistema Operativo, tiene una estructura fija, que constituye el formato de todo archivo con extensión *.o*.

Para el ensamblador que diseñamos y presentamos, el código objeto generado estará constituido, por ahora, por tres tipos de registros con el siguiente formato (mas adelante se añadirán y modificarán otros registros).

- Registro de encabezamiento:

Byte 1	H, identificador del tipo de registro.
Bytes 2-9	Nombre del programa.
Bytes 10-13	Dirección de comienzo del programa objeto.
Bytes 14-17	Longitud en bytes del programa objeto.

- Registro de texto:

Byte 1	T, identificador del tipo de registro.
Bytes 2-5	Dirección de inicio del código objeto en ese registro.

Bytes 6	Longitud en bytes del código objeto en ese registro. así, el tamaño máximo del código binario en un registro de este tipo es de 256 bytes.
Bytes 7-69	Código objeto.

- Registro de fin

Byte 1	E, identificador del tipo de registro.
Bytes 2-5	Dirección de la primera instrucción ejecutable del programa (punto de entrada).

Ejemplo 4: Supongamos el siguiente programa en lenguaje HAL. Se ha incluido una columna etiquetada como LC, la cual muestra como se incrementaría el valor de esta variable según la longitud de cada instrucción en assembler; este campo se coloca para ilustración del ejemplo y obviamente no es codificado por el programador. Nótese que no todas las instrucciones en lenguaje ensamblador modifican el valor del LC, ¿Por qué?.

Línea	**LC**		**Proposición fuente**		
0		BEGIN	START		0
5	0	R0	EQU		0
10	0	R1	EQU		1
15	0	A	WORD		10
20	4	B	WORD		-18
25	8	C	RESW		1
30	C		BASE		*
35	C	INICIO	LB		
40	D		CLEAR	RG	R0
45	F		CLEAR	RG	R1
50	11		LOAD	D	R0,A
55	15		ADD	D	R0,B
60	19		STORE	D	R0,C
65	1D		STORE		R0,AUX
70	20		SYSCALL	IN	10

75	23	AUX	RESW	1
80	27	AUX1	BYTE	C'ABC'
85	2A		END	INICIO

El código objeto para este programa será el siguiente

```
48 424547494E202020 00000000 0000002A
54 00000000 08 0000000A FFFFFFEE
54 00000008 1D 98 1100 1101 45000000 05000004 A5000008 A00017 94000A
54 00000027 03 414243
45 0000000C
```

los espacios en blanco que se observan se colocan con fines de legibilidad, pero no forman parte del código objeto. Todos los caracteres alfabéticos se representan en ASCII, y debe tenerse en cuenta que la representación ASCII de un carácter ocupa dos dígitos hexadecimales (un byte). La interpretación del primer registro es la siguiente:

48 =	representación ASCII del caracter H.
424547494E202020 =	representación ACSII del nombre del programa BEGIN, el carácter ASCII 20 representa un blanco.
00000000 =	dirección de comienzo del programa objeto.
0000002A =	longitud en bytes del código objeto.

la interpretación de los otros registros es similar. Si el tamaño del código objeto es mayor a 256 bytes, se necesitarán varios registros de textos. Así mismo, se dejan "huecos" dentro del espacio de direcciones que corresponden a data no inicializada, es decir, aquellas posiciones de memoria que reservan y en las cuales no se definen valores iniciales (ejemplo, la pseudoinstrucción de la línea 25). Posteriormente se explicará como generar este resultado.

Estructura de un ensamblador básico.

Para efectuar la traducción del programa fuente a código objeto es necesario realizar las siguientes funciones (no necesariamente en el orden citado):

1. Conversión de los códigos de operaciones mnemónicos a sus equivalentes en lenguaje de máquina.
2. Conversión de operandos simbólicos a sus direcciones de máquina equivalentes.
3. Construcción de las instrucciones de máquina en el formato adecuado.
4. Conversión de las constantes de datos especificadas en el programa fuente a sus representaciones internas de máquina.
5. Escribir el programa objeto en un archivo de salida.

Todas estas funciones, excepto la número 2, son sencillas de realizar procesando el programa fuente línea por línea, instrucción por instrucción. No obstante, la traducción de referencias simbólicas a direcciones de memoria a veces presenta problemas. Si se considera la proposición de la línea 65 en el ejemplo 4, se observa que la instrucción no podrá ser procesada ya que se desconoce en ese momento la dirección que se le asignará al símbolo AUX: este símbolo no tiene una definición previa a la codificación de la instrucción que lo utiliza. Esta situación se conoce como *el problema del símbolo futuro o de la referencia no resuelta,* y tiene su origen en "referencias hacia delante" que realizan las instrucciones en su campo de operandos.

Para abordar la situación anterior y permitir la referencia a símbolos definidos "mas adelante", la mayoría de los ensambladores analizan dos veces el código fuente: en un *primer paso* se hace poco más que buscar en el programa fuente la definición de símbolos o etiquetas y asignarles direcciones (como las de la columna LC); no se ensamblan instrucciones, y al final de esta fase, la tabla de símbolos contiene todos los símbolos definidos dentro del programa. En el *segundo paso,* el programa fuente es leído de nuevo y se realiza la mayor parte del proceso de traducción de instrucciones, usando la tabla de símbolos que se generó anteriormente. Un ensamblador que trabaje de esta manera, recibe el nombre de *ensamblador de dos pasos*, en contraposición con aquellos ensambladores que

no permiten la codificación de referencias hacia delante, denominados *ensambladores de un paso*.

Además de traducir las instrucciones del programa fuente, el ensamblador debe procesar todas las pseudoinstrucciones. Estas proposiciones no se traducen por lo general en instrucciones de máquina, aunque influyen en la forma del código objeto resultante.

Finalmente, el ensamblador debe escribir el código objeto generado en algún dispositivo de almacenamiento secundario. Este programa objeto resultante será procesado luego por un enlazador y un cargador.

Podemos ahora dar una descripción general de las funciones realizadas por un ensamblador de dos pasos.

Primera Pasada (procesa las referencias simbólicas).

1. Asignar direcciones a todas las proposiciones del programa fuente que así lo requieran.
2. Guardar, en la tabla de símbolos, las direcciones asignada a todos los símbolos definidos dentro del programa.
3. Procesar, hasta donde sea posible, aquellas directrices del ensamblador que afectan la asignación de direcciones, es decir, que modifican el contenido del contador de localizaciones. Ejemplo de directrices de este tipo lo constituyen las proposiciones START y RESW.

Segunda Pasada (ensambla instrucciones y genera programa objeto).

1. Ensamblar las direcciones; es decir, traducir instrucciones y direcciones.
2. Generar los valores definidos por BYTE y WORD. Los elementos de memoria reservados a través de RESW y RESB, se inicializan en cero o en blanco según sea el caso.
3. Procesar las directrices no procesadas durante la primera pasada..
4. Escribir el código objeto resultante en memoria secundaria.

Estructuras de datos del ensamblador.

El ensamblador propuesto mantiene las siguientes estructuras de datos.

- Contador de localizaciones LC, usado para asignar direcciones a cada instrucción y símbolo definido dentro del programa. Su valor inicial se inicializa a partir de la dirección de comienzo especificada en la directriz START. Después de procesar cada proposición fuente, se suma al LC la longitud de la instrucción a ensamblar o del área de datos a generar. De esta forma, cada vez que se detecta una etiqueta en el programa fuente, el valor actual del LC proporciona la dirección que se le asignará a ese símbolo.

- Tabla de códigos de máquina TABOP, que contiene los símbolos mnemónicos de cada instrucción en lenguaje ensamblador, su correspondiente código de máquina y los tipos de direccionamiento permitidos con cada una. Esta información le permite al ensamblador determinar si el tipo de direccionamiento especificado en la línea de código es valido o no, y cual es formato de la instrucción. En ensambladores mas complejos, la información en esta tabla es mucho mas densa. Durante la primera pasada, TABOP se utiliza para validar si la operación especificada en una proposición es valida, y en caso de serlo poder deducir la longitud de la instrucción y así incrementa el LC apropiadamente. Durante la segunda pasada, esta tabla se usa para permitir la traducción de la proposición fuente.

- Tabla de formatos TABFORM, una pequeña estructura de datos usada por el ensamblador de HAL la cual señala, para cada tipo de direccionamiento, la longitud de una instrucción que use ese método de direccionamiento, y el formato de la instrucción resultante. Esta información es necesaria para poder conocer el incremento que se le aplicará al contador de localizaciones cuando se trate una instrucción de máquina. No es una estructura de datos que usen muchos ensambladores, en nuestro caso, dado que HALC usa un mismo código de operación con distintos tipos de direccionamiento, y además la longitud de una

instrucción depende del direccionamiento usado, esta estructura de datos resulta conveniente.

- Tabla de símbolos TABSIM, que incluye básicamente el nombre y valor (dirección) de cada etiqueta símbolo definido en el programa fuente, pudiendo adicionalmente mantener indicadores que señalen condiciones de error (por ejemplo, símbolo duplicado) y referencias cruzadas (en que línea se define un símbolo y en cuales se usa un símbolo ya definido). Esta tabla es creada durante la primera pasada; el ensamblador añade símbolos en la misma a medida que los encuentra en el programa fuente que va procesando proposición por proposición, especificando la dirección que el símbolo tendrá asignada (valor del LC). Durante la segunda pasada, se buscan en esta tabla los símbolos empleados como operandos, para así obtener las direcciones que se van a insertar en las instrucciones ensambladas.

Lógica general del ensamblador.

Los algoritmos que se muestran a continuación, muestran el flujo de ejecución de nuestro ensamblador de dos pasos. Tienen un carácter bastante general y se usa un lenguaje natural para describir, con una visión macro, muchas de las acciones que realiza.

Algoritmo general 1° paso:

```
comenzar
  nombre:= blancos, dirinicio:= 0, base:= 0, long:= 0
  leer la primera línea de código del programa
  si codop = 'START' entonces
        comenzar
              nombre:= etiqueta                    % le damos nombre al código objeto
              loc:= dirinicio:= operando instrucción START   % inicializamos contador de localizaciones
              grabar línea leída en archivo intermedio.
              leer siguiente línea de programa.
        fin
```

```
sino loc:=0          %no se especifico dirección de inicio; se asume contador de localizaciones = 0
mientras cod_op ≠ 'END' hacer
    comenzar
        si esta no es una línea de comentario entonces
          comenzar
            si la proposcion define un símbolo entonces
              comenzar
                buscar etiquete en TABSIM
                si se encuentra activar bandera de error          %simbolo duplicado
                sino
                  si codop = EQU entonces insertar en TABSIM (símbolo, operando)
                  sino insertar en TABSIM (símbolo, loc)
              fin
            buscar codop en TABOP
            si se encuentra entonces
              comenzar
                determinar tipo de direccionamiento usado (examinar campo TD)
                ubicar en TABFORM la longitud de la instrucción según el valor de TD
                incrementar LC en la longitud correspondiente
              fin                    % no es una instrucción de máquina ¿es una directriz?
            sino si codop = 'WORD' entonces loc:=loc + 4
              sino si codop = 'RESW entonces loc:=loc + 4 * operando
                sino si codop = 'RESB' entonces loc:=loc + operando
                  sino si codop = 'BYTE' entonces
                    comenzar
                      determinar la longitud de la constante
                      loc:=loc + longitud determinada
                    fin
                  sino si codop = BASE entonces
                    comenzar
                      si operando = * entonces base:= loc
                      sino base:= operando
                    fin
                  sino activar bandera de error          %código invalido
          fin  {si esta no es una línea de comentario}
        grabar línea leída en archivo intermedio
        leer siguiente línea
fin
```

```
  long:=loc – dirinicio                    % calculamos la longitud del programa
fin                                        %fin de la 1° pasada
```

Algoritmo general 2° paso; recibe como entrada el archivo intermedio, las variables *nombre, dirnicio, base y long*, así como la tabla de símbolos.

```
Comenzar
  leer la primera instrucción desde archivo intermedio
  si codop = 'START' entonces
      comenzar
          escribir instruccion leida en archivo de salida
          leer siguiente instrucción
      fin
  escribir (H,nombre,dirinicio,long)                    %se crea registro de encabezamiento
  escribir buffer en el archivo objeto
  longt := 0                                            % longitud del código objeto en el registro texto
  iniciot := dirinicio                    % dirección de inicio del código objeto en el registro texto
  mientras cod_op ≠ 'END' hacer
      comenzar
          buffer := blancos
          si esta no es una línea de comentario entonces
              comenzar
                  buscar codop en TABOP
                  si se encuentra entonces
                      comenzar
                          si hay símbolo en el campo operando entonces para cada operando
                            comenzar
                              buscar símbolo en TABSIM
                              si se encuentra entonces
                                comenzar
                                  si tipo de direccionamiento = relativo entonces
                                                  despz : = dirección en TABSIM - base
                                  guardar dirección del símbolo o despz como dirección del
                                  operando
                                fin
                              sino                          % no se halló el símbolo
```

```
                    comenzar
                      almacenar 0 como dirección del operando
                      activar bandera de error          %símbolo no definido
                    fin
                  fin                                   %si hay símbolo
                ensamblar instrucción
                si instrucción ensamblada no cabe en el registro texto actual entonces
                  comenzar                              % grabar nuevo registro de texto
                    escribir (T,iniciot,longt,codigo objeto generado)
                    iniciot := iniciot + longt
                    longt := 0
                  fin
                sino longt := longt + longitud de la instruccion ensamblada
              fin
            sino si codop = BYTE o codop = WORD entonces
                comenzar
                  ensamblar la constante en código objeto
                  si constante ensamblada no cabe en el registro texto actual entonces
                    comenzar
                      escribir (T,iniciot,longt,codigo objeto generado)
                      iniciot := iniciot + longt
                      longt := 0
                    fin
                  sino longt := longt + longitud de la constante generada
                fin
              sino si codop = RESW o codop = RESB entonces
                  comenzar
                    escribir (T,iniciot,longt,codigo objeto generado)
                    iniciot := iniciot + longitud área reservada
                    longt := 0
                  fin
            fin
          imprimir instrucción fuente leída en reporte de salida
          leer siguiente instrucción
      fin
grabar ultimo registro de texto
escribir (E,operando de la pseudoinstruccion END)
imprimir instrucción leída
```

fin fin de la 2° pasada

Traducción de instrucciones.

Los algoritmos mostrados muestran el flujo lógico de un ensamblador de dos pasos: Aunque la descripción se ha dado para un ensamblador simple, esta es también la lógica en que se basan ensambladores mas complejos. Es importante comprender totalmente estos algoritmos. Se recomienda seguir la lógica de los mismos, aplicándolos "a mano" al programa fuente del ejemplo 4, para producir el código objeto que se analizó con anterioridad.

En esta sección se explicará en detalle el proceso de traducción de instrucciones y la generación de código de máquina para las variables y constantes que se definan en un programa. Para tal fin, la explicación se basa en proposiciones del programa del ejemplo 4. Se usará una versión de la tabla de códigos de máquina similar a la tabla de la pagina 4.

La primera pasada para este programa produce los siguientes resultados.

Tabla de símbolos

Símbolo	Valor	Banderas	N° línea
R0	0		5
R1	1		10
A	0		15
B	4		20
C	8		25
INICIO	C		35
AUX	23		75
AUX1	27		80

nombre = BEGIN; dirinicio = 0; long = 2A

A continuación se mostrará el proceso de ensamblaje de algunas instrucciones

a) Proposición línea 15

A WORD 10

esta directriz le indica al ensamblador que debe reservar una palabra (cuatro bytes) dentro del espacio lógico donde opera, y le asigne a esa palabra el valor entero 10. Por tanto, el código objeto generado es

00000000 00000000 00000000 00001010

Este código binario es normalmente abreviado para su visualización en reportes, representándolo en hexadecimal como

00 00 00 0A

b) A continuación se ensamblará la proposición de la línea 20

B WORD -18

El ensamblador reserva una palabra y a la misma le asigna el valor entero -18; así el código generado es

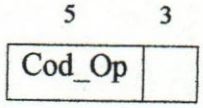
11111111 11111111 11111111 11101110

en hexadecimal se tiene FF FF FF EE

c) Proposición línea 35

LB

El ensamblador toma el mnemónico señalado como operación (LB) y lo busca en la tabla de códigos de máquina; de esta búsqueda deduce que el código binario de esta operación es 10011. La misma tabla señala que esta instrucción usa direccionamiento implícito. Con este dato se examina la tabla de formatos y deducimos que el formato para este tipo de instrucciones es



Los últimos tres bits de la instrucción no se usan, por lo que se colocará en cero, quedando entonces el código binario de la instrucción como

$$10011000 = 98_{16}$$

d) Proposición línea 45

CLEAR RG R1

El ensamblador toma el mnemónico codificado y lo busca en la tabla de códigos de máquina; esta búsqueda produce como resultado el código de operación deseado: 00010. La misma tabla señala que el tipo de direccionamiento valido para esta instrucción es el de registro. Examinando la proposición codificada, el ensamblador se da cuenta que este es el tipo de direccionamiento codificado (puede usarse este hecho para señalar un error en caso contrario). Con ayuda de la tabla de formatos, se deduce el formato de instrucciones con este tipo de direccionamiento

Número de bits	5	3	4	4
	Cod_Op	TD	R1	R2

Este tipo de direccionamiento se codifica como 001. Se continúa examinando la proposición codificada y se detecta que el operando especificado es R1; este símbolo se busca en la tabla de símbolos y se determina que su valor es 1; este valor se representa en cuatro bits (longitud de este campo) como 0001. Se asume un primer operando igual a cero (no se necesita para este tipo de instrucciones). Resumiendo, se tienen los siguientes valores para cada uno de los campos de esta instrucción

Cod-Op = 00010

TD = 001

R1 = 0000

R2 = 0001

así, el código binario de la instrucción es

$$00010001\ 00000001 = 1101_{16}$$

e) Proposición línea 55

ADD D R0,B

El código de máquina asociado a este mnemónico es 00000; el tipo de direccionamiento es directo D, cuyo código es 101. Las instrucciones con este tipo de direccionamiento tienen el siguiente formato

Número de bits	5	3	4	20
	Cod_Op	TD	R1	dirección

El primer operando codificado es R0, cuyo valor declarado en la tabla de símbolos es 0. El segundo operando es B; la tabla de símbolos nos da la dirección efectiva de este símbolo: 4. Se tiene entonces

Cod_Op = 00000

TD = 101

R1 = 0000

dirección = 00000000000000000100

así, el código binario de la instrucción es

$$00000101\ 00000000\ 00000000\ 00000100 = 05000004_{16}$$

f) Proposición línea 65

STORE R0,AUX

El código de máquina de una instrucción STORE es 10100; en esta instrucción, según la información en la tabla de operaciones, se pueden usar los tipos de direccionamiento relativo, de registro, relativo indexado y directo. El campo tipo de direccionamiento no señala ninguna en especial, por lo que se asume un método de direccionamiento relativo (es el tipo por defecto según se explico anteriormente). El código de este tipo de direccionamiento es 000, y el formato de este tipo de instrucciones es

Número de bits	5	3	4	12
	Cod_Op	TD	R1	desplazamiento

el primer operando codificado en la instrucciones el símbolo R0, cuyo valor según la tabla de símbolos es 0. El segundo operando es el símbolo AUX cuyo valor es 23_{16}; esta es la dirección efectiva. Sin embargo, la instrucción especifica un desplazamiento. Para calcular este desplazamiento se debe conocer cual es valor base usado; este valor se calculó en la primera pasada, y se obtuvo de la proposición BASE de la línea 30. Este valor es entrada para la segunda pasada. Por lo tanto, el desplazamiento es

dirección efectiva = base + desplazamiento $\quad \therefore$ desplazamiento = dirección efectiva - base

$$\text{desplazamiento} = 23_{16} - C_{16} = 17_{16}$$

así, se tienen los siguientes valores para cada campo de la instrucción

$$\text{Cod-Op} = 10100$$

$$\text{TD} = 000$$

$$\text{R1} = 000$$

$$\text{desplazamiento} = 000000010111$$

el código binario de la instrucción es

$$10100000\ 0000000\ 00010111 = A00017_{16}$$

Actividades.

1- Realizar la primera pasada de los ejemplos 1 y 2, y ensamblar las instrucciones resaltadas.

2- Suponga un ensamblador de dos pasos como el estudiado en clase. Se desea modificar dicho ensamblador de manera tal que pueda generar mensajes de advertencia (warning) para aquellas etiquetas que no sean referenciadas en el programa. Dicho mensaje debe aparecer a continuación de la línea donde se define la etiqueta no referenciada. Describa

los cambios que realizaría (al ensamblador en sus dos pasadas) a fin de soportar tal característica.

```
P3          START     1000
            LDA       DELTA
            ADD       BETA
LOOP        STA       DELTA
--- Warning: etiqueta jamás referenciada
            ...
            ...
```

3- Suponga un ensamblador como el estudiado en clase. En el caso de símbolos duplicados, dicho ensamblador debe generar un error. Se desea que proponga una modificación al diseño general del ensamblador de manera tal que este, no solo señale la ocurrencia de un símbolo duplicado, sino que también indique en cuales instrucciones se usa dicho símbolo duplicado.

Relocalización de programas.

La labor de un ensamblador se limita a la generación de un código objeto. Un código objeto no representa una unidad ejecutable; el mismo necesita ser procesado por otros procesadores de lenguajes para la obtención de un código ejecutable, y para la carga apropiada del mismo cuando se necesite su ejecución. La función de otros procesadores de lenguaje requiere generalmente de la ayuda del ensamblador. En esta sección, y en la siguiente, se explicará cómo el ensamblador genera este apoyo.

En un sistema con multiprogramación, existirán varios programas cargados en memoria ejecutándose concurrentemente. Si el ensamblador conoce de antemano en que región de memoria ha de ejecutarse un programa, podría asociarle a cada elemento del mismo, instrucción o dato, una dirección o posición de memoria. De ser así, esta asociación

o *binding* entre un elemento de un programa y una posición de memoria se llevaría a cabo en *tiempo de traducción*.

Sin embargo, este no es el caso: en tiempo de ensamblaje se desconoce generalmente la región de memoria donde se cargará un programa en *tiempo de ejecución*. Por esta razón, un ensamblador se vale del artificio de asignar memoria a cada elemento de un programa, dentro de un espacio de direcciones hipotético, ficticio o lógico, cuya dirección de comienzo, en el caso de HAL, la define el programador por medio de la directriz START. Esta asociación se establece durante la primera pasada del proceso de ensamblaje, y para ello el ensamblador mantiene en el contador de localizaciones, la traza de la siguiente posición disponible dentro de este espacio de direcciones lógico. Dentro de esta región, que por definición es continua, la asignación de memoria también es continua, lo cual comulga con el funcionamiento de una arquitectura Von Neumann, donde se asume que las instrucciones a ser ejecutadas ocupan posiciones consecutivas de memoria.

Por lo general, el espacio de direcciones que "ve" el ensamblador no coincide con el espacio de direcciones que el Sistema Operativo asigna, en *tiempo de carga*, al programa. Por ello, las instrucciones que hagan referencia directa a posiciones de memoria pueden presentar problemas en su ejecución. Esta situación es analizada a continuación, tomando como referencia el programa mostrado en el ejemplo 4.

Línea	**LC**		**Proposición fuente**		
0	0	BEGIN	START		0
5	0	R0	EQU		0
10	0	R1	EQU		1
15	0	A	WORD		10
20	4	B	WORD		-18
25	8	C	RESW		1
30	C		BASE		*
35	C	INICIO	LB		
40	D		CLEAR	RG	R0

45	F		CLEAR	RG	R1
50	11		LOAD	D	R0,A
55	15		ADD	D	R0,B
60	19		STORE	D	R0,C
65	1D		STORE		R0,AUX
70	20		SYSCALL	IN	10
75	23	AUX	RESW		1
80	27	AUX1	BYTE		C'ABC'
85	2A		END		INICIO

La instrucción de la línea 65 declara

STORE R0,AUX

el tipo de direccionamiento especificado en esta instrucción es relativo, y la primera pasada le asigna al símbolo AUX, una palabra de memoria cuya dirección de inicio es 0x23, dentro de un espacio de direcciones lógico que comienza en la posición 0x0000.

En la segunda pasada, la instrucción es ensamblada y, dado el tipo de direccionamiento, se calcula el desplazamiento a partir de la dirección base, la cual se define en la línea 30 como *base = 0x0C*, por lo tanto

dirección efectiva = dirección base + desplazamiento ∴

$$\text{desplazamiento} = 23_{16} - C_{16} = 17_{16}$$

así, la instrucción será ensamblada como

A00017

Si este programa se carga en tiempo de ejecución a partir de la posición 0x00001000, lucirá mas o menos de la siguiente manera

Posición de memoria	*Contenido*
00001000	0000000A
00001004	FFFFFFEE
00001008	00000000
0000100C	LB

Posición de memoria	*Contenido*		
0000100D	CLEAR	RG	R0
0000100F	CLEAR	RG	R1
00001011	LOAD	D	R0,A
00001015	ADD	D	R0,B
00001019	STORE	D	R0,C
0000101D	STORE		R0,AUX
00001020	SYSCALL	IN	10
00001023	00000000		
00001027	414243		

Por comodidad se han utilizados mnemónicos y símbolos para representar el contenido de aquellas posiciones de memoria que contienen instrucciones.

Cargado en memoria este programa, el Sistema Operativo iniciará la ejecución del mismo, colocando en el contador de instrucciones su punto de entrada: 0x0000100C (especificado en la línea 85 con la pseudoinstrucción END). La ejecución de la instrucción LB almacenada en esta dirección, carga en el registro base el valor 0x0000100C (la dirección de la posición de memoria a partir de la cual esta cargada la instrucción). Al ejecutar el computador la instrucción de la posición 0x0000101D, la unidad de control detecta que el tipo de direccionamiento es relativo, toma el desplazamiento y el contenido del registro base y calcula la dirección efectiva

$$DE = 0000100C_{16} + 17_{16} = 00001023_{16}$$

direccionándose por consiguiente la posición de memoria correcta. Se puede concluir entonces que aquellas instrucciones que usen direccionamiento relativo, no se ven afectadas cuando el programa es cargado en una región de memoria con origen distinto al que se usó para su ensamblaje. Esto resulta obvio dada las características de este tipo de direccionamiento.

Ahora bien, ¿qué sucede con aquellas instrucciones que no usan un direccionamiento relativo, en el cual las referencias a memoria son automáticamente relocalizadas en tiempo de ejecución, sino que basan su acceso a memoria en un

direccionamiento directo?. En un direccionamiento de este tipo, la instrucción siempre accesa la misma posición de memoria, y este vinculo entre una instrucción y la posición de memoria que ocupa su operando, se establece en tiempo de ensamblaje. ¿Qué sucede entonces si el espacio de direcciones en tiempo de traducción no coincide en su origen con el espacio de direcciones en tiempo de ejecución?. La instrucción de la línea 50, en el ejemplo 4, muestra esta situación

```
LOAD    D     R0,A
```

el tipo de direccionamiento usado en esta instrucción es directo, y en la primera pasada el ensamblador asocia al símbolo A una palabra cuya dirección de comienzo es 0x00000000. La segunda pasada ensambla entonces la instrucción como

```
45000000
```

Si en tiempo de ejecución el programa se carga a partir de la dirección 0x00001000, la instrucción anterior estará almacenada entonces en la posición 0x00001011. La misma intenta referenciar a un operando ubicado en la posición de memoria 0x00000000, fuera del espacio de memoria asignado, con un valor almacenado probablemente distinto al que se espera cargar, y que quizá sea parte de otro programa. Es necesario modificar esta instrucción al momento de su carga, para que haga referencia al elemento correcto, el cual, dentro del espacio de direcciones asignado a este programa, se encuentra en la posición 0x00001000.

Pareciera ser suficiente, en función de la problemática planteada, que en tiempo de carga se le sume a esta instrucción la dirección de comienzo de la región asignada. Igual situación presentan otras instrucciones en este programa. Sin embargo, si solo se tiene en cuenta el código objeto de este programa, el cual se muestra a continuación, es en general imposible reconocer que instrucciones necesitan ser modificadas y cuales no; mas aun, resulta difícil deducir qué valores binarios corresponden a instrucciones y cuales a datos.

```
48 424547494E202020 00000000 0000002A
54 00000000 08 0000000A FFFFFFEE
54 00000008 1D 98 1100 1101 45000000 05000004 A5000008 A00017 94000A
54 00000027 03 414243
45 0000000C
```

El ensamblador desconoce la localidad real donde se cargará un programa, por lo que no puede realizar los cambios necesarios en las instrucciones con direccionamiento directo. Sin embargo, el ensamblador puede identificar aquellas partes del programa objeto que necesitan ser modificadas, y podrá entonces generar estructuras de datos que las identifiquen. De esta manera, *el cargador (loader)*, una vez conocida la dirección de carga, podría realizar las modificaciones necesarias. Un programa objeto que contiene la información necesaria para que un procesador de lenguaje realice este tipo de modificación, se denomina *código objeto relocalizable* (a diferencia de uno que no lo tiene, el cual es denominado *código absoluto*).

Obsérvese que, independientemente de donde se cargue el programa, los símbolos A, B, y C, por ejemplo, siempre estarán 0, 4 y 8 bytes "mas adelante" de la dirección de inicio del programa. Por lo tanto, el problema de relocalización se puede resolver como sigue:

1. Cuando el ensamblador genere el código objeto de aquellas instrucciones que hagan referencias directas a memoria, insertará, en el campo de dirección de la instrucción, la dirección del operando *relativa al inicio del programa.*
2. El ensamblador también generará una notificación para el cargador, instruyéndolo a sumar la dirección de carga al campo de dirección de la instruccion indicada.

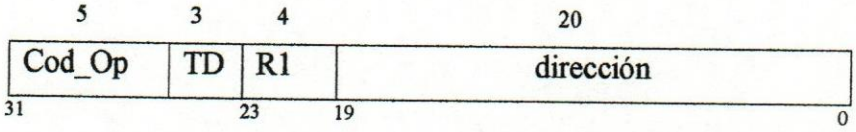
Como es lógico deducir, esta notificación para el cargador debe ser parte del programa objeto. Esto puede implantarse con un nuevo tipo de registro en el código objeto, denominado *Registro de Modificación*, con el siguiente formato

Registro de modificación

Byte 1	M
Bytes 2-5	dirección, relativa a un origen igual a cero, de la instrucción cuyo campo de dirección debe modificarse.

el conjunto de registros de modificación que conforman el código objeto de un programa, conforman una estructura que se conoce como *Diccionario de Relocalización RLD.*

Para ilustrar como el ensamblador HAL implanta este mecanismo, se debe recordar el formato de una instrucción con direccionamiento directo.



como se observa, los tres bytes mas a la derecha contienen la dirección tanto del primer operando, un registro, como del operando en memoria. En particular, los 20 bits menos significativos contienen la dirección de este operando. Así, el cargador deberá sumar un entero de 32 bits cuyos 8 bits mas significativos sean 0s. Esto obliga a que la relocalización de un programa esté limitada a los primeros 2^{20} bytes (1MB) de memoria, y es en general una desventaja de los programas que usen este tipo de direccionamiento. Luego se explicará como soslayar esta limitante.

Durante la segunda pasada, el ensamblador puede mantener un contador de localizaciones LC que se inicialice en cero al comienzo de la misma. El contenido del LC es incrementado en la longitud de cada elemento ensamblado (la directriz START no afecta su contenido ya que la misma no es procesada durante la segunda pasada). Así, antes de ensamblar cualquier elemento del programa, el LC le indicará al ensamblador cual es la posición del elemento a ensamblar relativa a un origen igual a cero.

Al ensamblar una instrucción, el ensamblador debe deducir el tipo de direccionamiento usado en la misma. Si este es directo, busca el símbolo definido como segundo operando en la tabla de símbolos y obtiene la dirección asignada al mismo. Seguidamente le resta a esta dirección, la dirección de inicio del programa, la cual se mantiene en la variable *dirinicio* calculada durante la primera pasada y recibida como entrada en esta fase de ensamblaje. El valor de esta resta es colocado en el campo de dirección de la instrucción ensamblada (constituye básicamente un desplazamiento). Finalmente, el ensamblador genera un registro de modificación, colocando en los bytes 2 -

5 el valor actual del LC. Por ultimo, el LC es incrementado en la longitud de la instrucción ensamblada.

Finalmente, debe tomarse en cuenta que todas las direcciones especificadas en los registros de textos y en el registro de fin, las cuales anteriormente eran direcciones directas, deben ser ahora relativas al origen del programa.

Ejemplo 5. Supóngase el siguiente programa fuente, el cual es básicamente el codificado como ejemplo 4 pero variando su dirección de inicio (la misma se especifica en decimal en la directriz START)

Línea	**LC**		**Proposición fuente**		
0	0	BEGIN	START		1000
5	3E8	R0	EQU		0
10	3E8	R1	EQU		1
15	3E8	A	WORD		10
20	3EC	B	WORD		-18
25	3F0	C	RESW		1
30	3F4		BASE		*
35	3F4	INICIO	LB		
40	3F5		CLEAR	R	R0
45	3F7		CLEAR	R	R1
50	3F9		LOAD	D	R0,A
55	3FD		ADD	D	R0,B
60	401		STORE	D	R0,C
65	405		STORE		R0,AUX
70	408		SYSCALL	IN	10
75	40B	AUX	RESW		1
80	40F	AUX1	BYTE		C'ABC'
85	412		END		INICIO

La primera pasada produce la siguiente salida

Símbolo	Valor	Banderas	N° línea
R0	0		5
R1	1		10
A	03E8		15
B	03EC		20
C	03F0		25
INICIO	03F4		35
AUX	0403		75
AUX1	040F		80

dirnicio = 0x03E8, nombre = BEGIN, long = 2A

Supongamos el procesamiento de la instrucción de la línea 55 durante la segunda pasada

ADD D R0,B

El ensamblador examina la tabla de símbolo para hallar la dirección asignada al segundo operando; esta valor es 0x03EC. Como el direccionamiento es directo, la instrucción será ensamblada usando como dirección el valor

$$03EC_{16} - 03E8_{16} = 0004_{16}$$

así, la instrucción ensamblada será

05000004

El contenido del LC al momento de ensamblar esta instrucción (segunda pasada) es 0x0015, por lo que el ensamblador generará el siguiente registro de modificación.

4D00000015

Aplicando este procedimiento al programa anterior, se obtiene el siguiente código objeto.

48 424547494E202020 00000000 0000002A
54 00000000 08 0000000A FFFFFFEE

```
54 00000008 1D 98 1100 1101 45000000 05000004 A5000008 A00017 94000A
54 00000027 03 414243
4D00000011
4D00000015
4D00000019
45 0000000C
```

Este código objeto tiene registros de texto similares a los obtenidos en el ejemplo 4, ¿por qué?.

Si en tiempo de carga este programa es cargado en una región de memoria cuya dirección de inicio es 0x00001000 (4096_{10}), lucirá como sigue (se resaltan las instrucciones con direccionamiento directo).

Posición de memoria	*Contenido*
00001000	0000000A
00001004	FFFFFFEE
00001008	00000000
0000100C	98
0000100D	1100
0000100F	1110
00001011	**45000000**
00001015	**05000004**
00001019	**A5000008**
0000101D	A00017
00001020	94000A
00001023	00000000
00001027	414243

Una vez cargado, y antes de que se inicie la ejecución del programa, el cargador examina los registros de modificación para realizar los cambios que señaló el ensamblador. El primer registro de modificación señala que 0x0011 bytes "mas adelante" de la dirección de inicio del programa en memoria, se tiene un valor al que hay sumarle la dirección de carga. El campo en cuestión comienza en la posición $1000_{16} + 11_{16} = 1011_{16}$. El

contenido de este campo es 0x45000000, y es al valor del campo de dirección de esta instrucción a quien se le suma la dirección de carga, quedando la instrucción una vez modificada como

45001000

el programa ya relocalizado tendrá la siguiente estructura (se resaltan las instrucciones cuyo campo de dirección fue modificado)

Posición de memoria	*Contenido*
00001000	0000000A
00001004	FFFFFFEE
00001008	00000000
0000100C	98
0000100D	1100
0000100F	1101
00001011	**45001000**
00001015	**05001004**
00001019	**A5001008**
0000101D	A00017
00001020	94000A
00001023	00000000
00001027	414243

Enlace de programas.

Generalmente, un programa está constituida por un conjunto de *secciones de control* enlazadas por un procesador de lenguaje denominado Enlazador (Linker). Una *sección de control* es una división lógica de un programa, es una rutina o un conjunto de estructuras de datos que ha sido codificada y traducida independientemente de otras, y su código objeto se mantiene en archivos especiales que usualmente se conocen como bibliotecas.

Una sección de control conserva su identidad después de su traducción y puede ser cargada y relocalizada independientemente de otras secciones de control. Es decir, cada sección de control tiene un nombre asignado, y la misma es reconocida siempre por ese nombre, sin importar el programa que la invoque; mas aun, es ese nombre el que se usa para su invocación: es a través de ese nombre que la sección de control es ubicada dentro de una cadena de bibliotecas Una sección de control, o múltiples copias de la misma, puede formar parte de distintos programas, y en cada uno de ellos puede estar cargada, y por tanto relocalizada, en regiones de memoria distintas. Así, las secciones de control facilitan el desarrollo de programas y constituyen la base de las técnicas de ingeniería de software y de la programación orientadas a objeto. Además, las interfaces de programación (API) de Sistemas Operativos como Unix y Linux, y de muchos Manejadores de Base de Datos, se desarrollan alrededor de este concepto.

EN HAL, la directriz START indica el comienzo de una sección de control, y, como se puede deducir, un programa fuente podrá estar constituido entonces por una sola sección de control. Esto no constituye una limitante ya que, como ya se mencionó, otras secciones de control que se necesiten, se pueden ensamblar independientemente.

Cuando múltiples secciones de control forman parte de un programa, se necesita proveer mecanismos que permitan su *enlace*. Por ejemplo, instrucciones en una sección control *A* pueden invocar instrucciones o usar datos definidos en otras secciones de control. Para tal invocación o tal uso, esas instrucciones deben conocer la dirección donde el elemento necesitado se encuentra almacenado. Dado que cada sección de control se carga y relocaliza independientemente de otras, el ensamblador desconoce, en tiempo de ensamblaje, la dirección que tendrán asignadas los elementos referenciados por la sección de control *A*. Si esto es así, ¿qué valor cargar, en el campo de dirección, de una instrucción de la sección de control de *A* que haga referencia a una palabra definida en otra sección de control?. El siguiente ejemplo aclara la problemática planteada.

Ejemplo 6: El siguiente programa copia un archivo desde el dispositivo de almacenamiento hacia el dispositivo de salida, y esta constituido por tres rutinas (secciones de control). Una

de ellas se encarga de las labores de coordinación del proceso de respaldo del archivo, e invoca a las otras dos rutinas, las cuales tiene como función la lectura y escritura de registros. Los registros del archivo de entrada tienen una longitud fija de 4096 bytes; el ultimo registro solo tiene almacenado el carácter nulo 0x00. Se supone además que los periféricos involucrados nunca están fuera de servicio, y no se cotejan condiciones de error.

Línea		Proposición fuente		
0	BACKUP	START		1024
5	CERO	EQU		0
10	R1	EQU		R1
20	LOOP	JSUB	D	LEEREG
25		LOAD	D	R1,LONG
30		CMP	I	R1,CERO
35		JE		ENDFILE
40		JSUB	D	WRITEREG
45		JUMP		LOOP
50	ENDFILE	SYSCALL	IN	10
55	BUFFER	RESB		4096
60	LONG	RESW		1
65		END		

Línea		Proposición fuente		
0	LEEREG	START		0
5	R0	EQU		0
7	R2	EQU		2
10	R3	EQU		3
12	RIO	EQU		15
15		CLEAR		R3
17		LA	D	R2,BUFFER

```
20                  LDCH        R0,INPUT
25      INLOOP      CMP         R3,SIZE
30                  JE          FINLEER
33      OCUPADO     TD
35                  JNE         OCUPADO
37                  RD
40      IORUN       TIO
45                  JNE         IORUN
50                  STCH    IR  RIO,R2
55                  CMP         RIO,EOF
60                  JE          FINLEER
70                  INC         R2
75                  INC         R3
80                  JMP         INLOOP
85      FINLEER     STORE   D   R3,LONG
87                  RSUB
90      INPUT       BYTE        X'F3'
95      SIZE        WORD        4096
100     EOF         BYTE        X'00000000'
105                 END
```

Línea	Proposición fuente		
0	WRITEREG	START	0
5	R0	EQU	0
7	R2	EQU	2
10	R3	EQU	3
15	RIO	EQU	15
20		CLEAR	R3
23		LDCH	R0,OUTPUT

```
25                LA      D    R2,BUFFER
30                LDCH    IR   RIO,R2
35                CMP           RIO,EOF
40                JE            FIN1
45   LAZO         CMP           R3,SIZE
50                JE            FIN2
55   OUTLOOP      TD
60                JNE           OUTLOOP
65                WD
70   IORUN        TIO
75                JNE           IORUN
80                INC           R2
85                INC           R3
90                JMP           LAZO
95   FIN1         TD
100               JNE           FIN1
105               WD
107  IORUN1       TIO
108               JNE           IORUN1
110  FIN2         RSUB
115  OUTPUT       BYTE          X'F2'
120  SIZE         WORD          4096
125  EOF          BYTE          X'00000000'
130               END
```

Nótese que en las tres secciones de control se repiten símbolos, lo cual no tiene nada de particular ya que estas se ensamblan independientemente, y los símbolos duplicados tienen un carácter local (no se exportan). Además, dos de las tres secciones de control tienen la misma dirección de comienzo.

Supóngase la proposición de la línea 20 de la sección de control BACKP

LOOP JSUB D LEEREG

en tiempo de ensamblaje, durante la segunda pasada, el ensamblador no reconoce la existencia del símbolo *LEEREG* y genera un error por símbolo no definido. Tal símbolo constituye una *referencia externa* y como tal, es el nombre de una rutina externa invocada en la sección de control *BACKUP*. Igual situación se da con las proposición de la línea 40 en la cual se referencia al símbolo externo *WRITEREG*. El ensamblador no sabe que tales símbolos constituyen *referencias externas*, es decir, referencias a símbolos definidos en otras secciones de control, y por ello falla el ensamblaje de instrucciones con estas referencias.

Una situación similar se presenta con las secciones de control LEEREG y WRITEREG, las cuales usan los símbolos externos LONG y BUFFER, definidos en la sección de control BACKUP.

Es labor del programador indicarle al ensamblador, cuáles símbolos usados dentro de una sección de control van a ser "importados" y como tal, están definidos en otras secciones de control. Así mismo, deberá especificar qué símbolos en cada sección de control, pueden ser exportados y usados en otras secciones de control. Para ello, se hace uso de la directrices EXTREF y EXTDEF.

Ejemplo 7. Este programa introduce el uso de las directrices EXTREF y EXTDEF para las tres secciones de control anteriores.

Línea	**LOC**	**Proposición fuente**		
0	0	BACKUP	START	1024
5	400	CERO	EQU	0
10	400	R1	EQU	R1
12	400	0	**EXTREF**	LEEREG,WRITEREG
15	400	0	**EXTDEF**	BUFFER,LONG

```
20    400   LOOP     JSUB     D    LEEREG
25    404            LOAD     D    R1,LONG
30    408            CMP      I    R1,CERO
35    40B            JE            ENDFILE
40    40E            JSUB     D    WRITEREG
45    412            JUMP          LOOP
50    415   ENDFILE  SYSCALL  IN   10
55    418   BUFFER   RESB         4096
60    1418  LONG     RESW         1
65    141C           END
```

El programa anterior le indica al ensamblador, a través de la directriz EXTREF de la línea 12, que los símbolos LEEREG y WRITEREG se encuentran definidos en otras secciones de control y van ser importados en esta sección. Así, el ensamblador, enterado de esto, no generará un error en la proposición de la línea 20, por ejemplo, por un intento de uso de un simbolo no definido. Sin embargo, dado que el ensamblador no conoce en tiempo de ensamblaje la dirección de memoria que tendrán asignados los símbolos que se importan, deberá generar la información necesaria para que el *enlazador* se encargue de resolver tales direcciones. Cuando un enlazador resuelve estas referencias externas se dice que esta *enlazando* las secciones de control involucradas. Un enlazador (linker) es un procesador de lenguaje encargado de enlazar las secciones de control que conformen un programa, y generar el código ejecutable del mismo.

La directriz EXTDEF de la línea 15 permite la especificación de símbolos definidos dentro de esta sección de control que van a ser exportados a otras secciones de control. Esta definición es importante, ya que, como se explicará luego, le permite al ensamblador guardar en estructuras de datos apropiadas la dirección, dentro de esta sección de control, de los símbolos a exportar, lo que a su vez le permite al enlazador realizar su función de enlace.

A continuación se muestran las especificaciones para las otras secciones de control.

Línea		Proposición fuente		
0	LEEREG	START		0
5	R0	EQU		0
7	R2	EQU		2
10	R3	EQU		3
12	RIO	EQU		15
13		EXTREF		BUFFER,LONG
15		CLEAR		R3
17		LA	D	R2,BUFFER
20		LDCH		R0,INPUT
25	INLOOP	CMP		R3,SIZE
30		JE		FINLEER
33	OCUPADO	TD		
35		JNE		OCUPADO
37		RD		
40	IORUN	TIO		
45		JNE		IORUN
50		STCH	IR	RIO,R2
55		CMP		RIO,EOF
60		JE		FINLEER
70		INC		R2
75		INC		R3
80		JMP		INLOOP
85	FINLEER	STORE	D	R3,LONG
87		RSUB		
90	INPUT	BYTE		X'F3'
95	SIZE	WORD		4096
100	EOF	BYTE		X'00000000'
105		END		

Línea	Proposición fuente			
0	WRITEREG	START		0
5	R0	EQU		0
7	R2	EQU		2
10	R3	EQU		3
15	RIO	EQU		15
17		EXTREF		BUFFER
20		CLEAR		R3
23		LDCH		R0,OUTPUT
25		LA	D	R2,BUFFER
30		LDCH	IR	RIO,R2
35		CMP		RIO,EOF
40		JE		FIN1
45	LAZO	CMP		R3,SIZE
50		JE		FIN2
55	OUTLOOP	TD		
60		JNE		OUTLOOP
65		WD		
70	IORUN	TIO		
75		JNE		IORUN
80		INC		R2
85		INC		R3
90		JMP		LAZO
95	FIN1	TD		
100		JNE		FIN1
105		WD		
107	IORUN1	TIO		
108		JNE		IORUN1
110	FIN2	RSUB		
115	OUTPUT	BYTE		X'F2'
120	SIZE	WORD		4096

```
125   EOF       BYTE        X'00000000'
130             END
```

Nótese como encaja la proposición EXTDEF de la sección de control BACKUP, con las proposiciones EXTREF de las secciones de control LEEREG y WRITEREG: todo símbolo que vaya a ser importado en una sección de control, tiene que ser exportado en otra. Se considera, por definición, que el nombre de una sección de control es un símbolo externo que puede ser usado por otras secciones de control, y no hace falta definirlo en una directriz EXTDEF.

Obsérvese también el tipo de direccionamiento usado a la hora de referenciar a un símbolo externo: directo. **No es posible en este tipo de referencias externas usar un direccionamiento relativo ya que el símbolo externo no guarda ninguna relación con ningún elemento dentro de la sección de control que lo importa**. En el ejemplo anterior, ¿a qué desplazamiento se encuentra en símbolo WRITEREG del origen de la sección de control BACKUP?; WRITEREG no guarda ninguna relación con la sección de control BACKUP y por lo tanto no es posible dar respuesta a esta pregunta.

Para poder enlazar dos secciones de control, el enlazador necesita conocer en que instrucciones de cada una se están usando referencias externas, y cuales son los símbolos que se van exportar. Para ello, el ensamblador genera una *tabla o diccionario de símbolos externos*. Esta tabla tiene una entrada por cada símbolo a importar o exportar dentro de una sección de control. El formato de cada entrada es el siguiente:

- Nombre del símbolo.
- Tipo de símbolo: Este campo toma el valor **E** si el símbolo se va a exportar (referencia externa), o **I** si se va importar I (símbolo definido dentro de esta sección de control y usado por otras)
- Desplazamiento, relativo al origen, del símbolo, en caso de que se vaya a exportar.

Esta tabla forma parte del código objeto generado por el ensamblador, por lo que en la práctica toma la forma de un conjunto de registros de un tipo especifico.

En la primera pasada se procesan las directrices EXTREF y EXTDEF; cada símbolo definido en las mismas es colocado en el diccionario de símbolos externos, con un indicador del tipo de símbolo E o I. En la segunda pasada, la directriz EXTDEF se vuelve a procesar. Esta vez, cada símbolo definido en ella se busca en la tabla de símbolos, para así determinar su desplazamiento relativo al origen e inicializar la entrada correspondiente en la tabla de símbolos externos.

En el ejemplo anterior, después de la primera pasada, se obtienen las siguientes estructuras de datos

Tabla de Símbolos

Símbolo	**Valor**	**Banderas**	**N° línea**
CERO	0		
R1	1		
LOOP	400		
ENDFILE	415		
BUFFER	418		
LONG	1418		

dirnicio = 0x0400, nombre = BACKUP, long = 101C

Tabla de Símbolos Externos

Simbolo	**Tipo**	**dirección Relativa**
LEEREG	I	
WRITEREG	I	
BUFFER	E	
LONG	E	

Durante la segunda pasada, el ensamblador completa la tabla de símbolos externos, examinado los símbolos en ella definidos y que sean del tipo E. Para cada símbolo a exportar, el ensamblador toma de la tabla de símbolos la dirección del mismo y le resta la dirección de inicio (0x0400 en este caso), resultando de esta manera la dirección del símbolo relativa a al comienzo de la sección de control; este resultado es entonces colocado en la entrada respectiva. La tabla así obtenida es

símbolo	**tipo**	**dirección Relativa**
LEEREG	I	
WRITEREG	I	
BUFFER	E	18
LONG	E	1018

Continuando con la segunda pasada, al intentar ensamblar la instrucción de la línea 20 de la sección de control BACKUP

JSUB D LEEREG

el ensamblador busca en la tabla de símbolos, el símbolo especificado en la instrucción; por tratarse de un símbolo que se importa, el mismo está ausente en esta tabla. Entonces , el ensamblador inicia la buscada en la tabla de símbolos externos, ubica el símbolo pero encuentra que desconoce la posición de memoria a partir de la cual, la sección de control en la que el símbolo LEEREG esta definido será cargada. así, no sabe que valor colocar en el campo de dirección de esta instrucción. En su lugar, el ensamblador inserta una dirección igual a cero

3D000000

y como el direccionamiento es directo, genera una entrada en el diccionario de relocalización.

4D00000000

Nótese lo siguiente, el ensamblaje de la instrucción de la línea 25 de la sección de control BACKUP

LOAD D R1,LONG

genera también un registro de modificación, ya que el direccionamiento es directo

Instrucción: 25001018

Registro de modificación: 4D0000004

Este registro de modificación le indica al cargador que debe relocalizar el campo de dirección de la instrucción que está en la posición 0x00000004 relativa al inicio de la sección de control BACKUP. Esta relocalización implica, como ya fue explicado, sumarle al campo de dirección de la instrucción señalada, la dirección de carga de la sección de control. Sin embargo, en el primer caso, el valor a almacenar en el campo de dirección es la dirección de carga de la sección de control en la que el símbolo LEEREG está definido.¿Cómo distingue el cargador entre ambas situaciones?. Para ello, es necesario modificar el registro de modificación, añadiendo otro campo que le indique al enlazador como manipular la dirección a la cual la instrucción señalada apunta. La estructura de este registro será entonces la siguiente.

Registro de modificación

Byte 1	M
Bytes 2-5	Dirección de la instrucción cuyo campo de dirección debe modificarse, relativa a cero.
Bytes 6-13	Simbolo externo cuyo valor se va a sumar al campo de dirección de la instrucción señalada.

Para la instrucción de la línea 25 debe tenerse en cuenta que el nombre de una sección de control es por definición un símbolo externo. En tiempo de carga, el valor de este símbolo es igual al la dirección de carga de la sección de control, y es precisamente esta dirección de carga la que debe sumarse para relocalizar la dirección. El registro de modificación es entonces

4D0000004BACKUP

lo que le indica al cargador que al campo de dirección de la instrucción cuya dirección relativa es 0x00000004, se le debe sumar el valor que tendrá el símbolo BACKUP en tiempo de carga.

Para la instrucción de la línea 20, el registro de modificación que genera el ensamblador es

4D0000000LEEREG

lo que le señala al cargador que el valor a sumar en el campo de dirección de la instrucción cuya dirección relativa es 0x00000000, es el valor que tome el símbolo LEEREG en tiempo de carga. De esta forma, el mismo mecanismo usado para la relocalización puede ser usado para el enlace. Ud. deberá intentar ensamblar cada una de las tres secciones de control mostradas y generar el código objeto completo para cada una de ellas.

Un comentario final, el uso del direccionamiento directo en HALC para invocar rutinas externas, limita la ubicación de las mismas, ya que en este tipo de direccionamiento, la dirección es un campo de 20 bits, por lo que el programa deberá ser cargado en los primeros 2^{20} bytes (1MB) de memoria. Una manera mas general de invocar rutinas externas y de soslayar la limitante anterior, es mediante el uso de constantes tipo dirección.

Línea		**Proposición fuente**		
0	BACKUP	START		0
5	CERO	EQU		0
10	R1	EQU		R1
12	R2	EQU		R2
13		EXTREF		LEEREG,WRITEREG
15		EXTDEF		BUFFER,LONG
17		LOAD		R2,RUTINA1
20	LOOP	JSUB	RG	R2
25		LOAD		R1,LONG
30		CMP	I	R1,CERO
35		JE		ENDFILE
37		LOAD		R2,RUTINA2
40		JSUB	RG	R2
45		JUMP		LOOP
50	ENDFILE	LOAD		R1,EOF

```
55                STORE          R1,BUFFER
57                LOAD     RG    R2,RUITNA2
60                JSUB     RG    R2
65                SYSCALL  IN    10
70   BUFFER       RESB           4096
75   EOF          BYTE           C'EOF '
80   LONG         RESW           1
85   RUTINA1      WORD           A(LEEREG)
90   RUTINA2      WORD           A(WRITEREG)
85                END
```

En este caso, se definen dos constantes tipo dirección RUTINA1 y RUTINA2 que contendrán en tiempo de ejecución, una vez que el enlazador haya resuelto las referencias externas señaladas y el cargador las haya relocalizado, las direcciones de las rutinas LEEREG y WRITEREG. El valor de estas constantes se carga en el registro R2, y la invocación a la rutina externa es hecha a través de un direccionamiento de registro.

Para programas como el anterior, el ensamblador deberá generar un registro de modificación por cada constante tipo dirección, y ensamblar una palabra con un valor igual a cero (la dirección del símbolo se desconoce) para cada constante especificada.

CAPITULO IV
ENLAZADORES Y CARGADORES

Tanto los enlazadores como los cargadores son parte integral de cualquier sistema de computación. Constituyen herramientas criticas que permiten la creación de programas modulares en lugar de bloques monolíticos de código. La función básica de cualquier cargador o enlazador es simple: *vincular nombres abstractos definidos dentro de un programa en lenguaje ensamblador o de alto nivel, con nombres concretos o físicos*, lo que le permite al programador diseñar sus programas usando abstracciones de mayor nivel que las ofrecidas por el computador subyacente. Así por ejemplo, pueden tomar un nombre como *getline* y vincularlo a *la posición de memoria que se encuentra 612 bytes mas allá del comienzo del código objeto iosys.*

Los enlazadores y cargadores ejecutan las siguientes relacionadas pero conceptualmente diferentes actividades:

- Carga de programas. Leen un programa desde un archivo en almacenamiento externo y lo almacenan en memoria para su posterior ejecución. Para ello, solicitan al Sistema Operativo la asignación de una región en memoria de tamaño apropiado. A fin de conocer el tamaño de memoria necesario, usan la información en el registro de encabezado del modulo objeto.
- Relocalización. Los traductores de lenguaje ejecutan su trabajo sobre un espacio de direcciones lógico que usualmente difiere del asignado en tiempo de carga; además, si un programa está constituido por diferentes secciones de control, todas ellas deben ser cargadas en regiones que no se solapen entre si. La relocalización es el proceso de asignar una dirección de carga, lógica o real, a cada elemento de un programa, y ajustar cada instrucción o dato para que reflejen esta dirección de carga. En muchos sistemas, la relocalización ocurre mas de una vez. Es muy común que un enlazador cree un código ejecutable (nombre que recibe la salida de un enlazador) a partir de múltiples subprogramas (secciones de control) sobre un espacio de direcciones lógico que comienza en la dirección cero, relocalizando

todas las secciones de control a partir de este origen. Cuando el módulo ejecutable es cargado, el programa es relocalizado de nuevo, para así ajustarlo a la dirección final de carga.

- Resolución de símbolos. Cuando un programa se construye a partir de múltiples subprogramas, las referencias entre ellas se realizan a través de símbolos. así, un programa principal puede usar una función *sqrt* almacenada en una biblioteca de procedimientos; el enlazador se encargará de resolver esta referencia externa tomando nota de la dirección asignada al símbolo *sqrt* y modificando el campo de de dirección de la instrucción que invoca a esta función en el programa principal.

Es posible la implantación de un programa que se encargue solo de la labor de enlace de secciones de control, y otro que realice la carga de programas en memoria. Igualmente, es posible el diseño de un Cargador-Enlazador que realice todas las funciones anteriormente descritas.

Sea cual fuese el diseño adoptado, todos se comportan como procesadores de lenguaje cuya entrada es un conjunto de uno o mas códigos objeto, cada uno con la siguiente estructura general:

Encabezado (nombre, tamaño, etc.)
Diccionario de texto (registros de texto)
Diccionario de relocalización
Diccionario de símbolos externos
Fin del modulo (punto de entrada)

Consideraremos a continuación el diseño general de un Cargador-Enlazador que procese la salida generada por el ensamblador HAL, y cuyo flujo general es similar al encontrado en muchos procesadores de lenguaje de esta naturaleza.

Diseño General de un Enlazador-Cargador.

Un programador tiene la inclinación natural de pensar en un programa como una entidad lógica que aglutina un conjunto de procedimientos relacionados. Sin embargo, desde el punto de vista de una cargador-enlazador no existe tal cosa como "un programa": solo hay secciones de control que necesitan ser enlazadas, relocalizadas y cargadas.

Considérese las siguientes tres secciones de control

Ejemplo 8

LC		**Proposición fuente**		
0	PROGA	START		0
		EXTREF		PROGB,B1,C1,C2
		EXTDEF		A1,A2
		.		
		.		
		.		
16	INIC1	LOAD		R3,A1
20		ADD	D	R3,C1
24		ADD	D	R3,B1
28		STORE	D	R3,C2
2C		JSUB	D	PROGB
		.		
		.		
		.		
40	A1	WORD		4
44	A2	RESW		10
48		END		INIC1

Registro de encabezado: nombre = PROGA

dirección de comienzo = 0

longitud = 48_{16}

Diccionario de Relocalización

dirección relativa	simbolo
20	C1
24	B1
28	C2
2C	PROGB

Diccionario de Símbolos Externos

símbolo	tipo	dirección relativa
PROGB	E	
B1	E	
C1	E	
C2	E	
A1	I	40
A2	I	44

LC		Proposición fuente		
0	PROGB	START		0
		EXTREF		PROGC,A1,C2
		EXTDEF		B1,B2
		.		
		.		
		.		
20	INIC1	LOAD	D	R3,A1
24		JSUB	D	PROCG
		.		
		.		
		.		
30	B1	RESW		1
34	B2	RESW		1
38		END		INIC1

Registro de encabezado: nombre = PROGB

dirección de comienzo = 0

longitud = 38_{16}

Diccionario de Relocalización

dirección relativa	simbolo
20	A1
24	PROGC

Diccionario de símbolos Externos

simbolo	tipo	dirección relativa
PROGC	E	
A1	E	
C2	E	
B1	I	30
B2	I	34

LC		**Proposición fuente**		
0	PROGC	START		4096
		EXTREF		A2,B2
		EXTDEF		C1
		.		
		.		
		.		
1000	INIC1	LOAD	D	R3,B2
1004		ADD	D	R3,A2
		.		
		.		
		.		
1020	C1	RESW		1
1024		END		INIC1

Registro de encabezado: nombre = PROGC

dirección de comienzo = 1000_{16}

longitud = 24_{16}

Diccionario de Relocalización

dirección relativa	simbolo
0	B2
4	A2

Diccionario de símbolos Externos

simbolo	tipo	dirección relativa
A2	E	
B2	E	
C1	I	20

Para cada una de las secciones de control anteriores se muestran los diccionarios de relocalización (en forma tabular) y de símbolos externos, así como su registro de encabezado. Cada una de ellas invoca símbolos externos definidos en las otras secciones de control. Nos interesa focalizarnos en el funcionamiento general de un enlazador-cargador, por lo que no es importante lo que cada una de las secciones de control haga; así, todas las porciones de código no relacionadas con el proceso de enlace han sido omitidas.

La entrada al enlazador-cargador es una cadena de secciones de control, una a continuación de la otra, como la mostrada anteriormente. Es usual que una sección de control haga referencia a un símbolo que esté definido en una sección de control "mas adelante" en la cadena de entrada, es decir, un símbolo cuya definición "aun no aparece". El vinculo o enlace no puede realizarse hasta que se le asigne una dirección al símbolo externo referenciado (es decir, hasta que esa sección de control sea leída). Nótese que esta es una situación similar a una referencia hacia adelante en un programa en lenguaje ensamblador. Por esta razón, el enlazador-cargador realiza dos pasadas sobre la cadena de secciones de control de entrada.

El corazón de nuestro enlazador-cargador es una estructura de datos llamada Tabla Global de Símbolos Externos GEST, la cual se usa para almacenar todos los símbolos encontrados en las tablas de símbolos externos de cada una de las secciones de control en el la cadena de entrada. Incluye además, el nombre de cada sección de control, el cual, como

ya fue explicado, se considera un símbolo externo. La GEST tiene esencialmente las mismas características y aplicaciones que la tabla de símbolos de un ensamblador.

Además de la GEST, el enlazador-cargador mantiene dos importantes variables. La primera de ella la llamaremos PROGADDR y mantiene la dirección de la región asignada por el Sistema Operativo para la carga de la aplicación enlazada. La segunda variable, llamada CSADDR, contiene la dirección de comienzo, dentro de la región asignada por el Sistema Operativo, de la sección de control actualmente manejada por el enlazador-cargador.

Durante la primera pasada el enlazador-cargador vincula cada símbolo externo en cada sección de control, con una dirección de memoria dentro de la región de carga que el Sistema Operativo haya asignado. Para tal fin, el enlazador-cargador obtiene dicha dirección de carga y con ella inicializa las variables PROGADDR y CSADDR. Luego, por cada sección de control, lee su registro de encabezado y extrae el nombre de la sección de control actual (la que esta siendo procesada) y su longitud. Seguidamente, añade el nombre de la sección de control junto con su dirección de comienzo (valor de la variable CSADDR) en la GEST. Luego, lee la tabla de símbolos externos de la sección de control, y por cada símbolo tipo I (símbolo definido en esta sección de control que va a ser exportado) se añade a la GEST la dupla <símbolo, dirección relativa + CSADDR>; esta ultima acción vincula efectivamente el símbolo con una posición de memoria.

Concluida la primera pasada, la GEST contiene la dirección asignada por el enlazador-cargador a cada símbolo externo a exportar en cada sección de control. Hecho esto, resta ajustar el campo de dirección de aquellas instrucciones que hagan referencia a símbolos externos. Como se observa, la salida de esta primera pasada es la tabla global de símbolos externos, la cual será entrada para la segunda pasada.

El algoritmo siguiente muestra, de manera general, el flujo de la primera pasada de nuestro enlazador-cargador.

```
comenzar
  obtener memoria del Sistema Operativo
  progaddr = csaddr = dirección de inicio de la región obtenida
  para cada sección de control hacer
    comenzar
      leer registro de encabezado
      cs_long = longitud de la sección de control
      cs_nombre = nombre de la sección de control
      buscar cs_nombre en GEST
      si se encuentra entonces
          inicializar flag_error (símbolo duplicada)
      sino
          añadir a la GEST la dupla  <cs_nombre, csaddr>
      mientras haya entradas por recorrer en la tabla de símbolos de la sección de control hacer
        comenzar
          si tipo = I entonces
            comenzar
              buscar símbolo en GEST
              si se encuentra entonces
                inicializar flag_error (símbolo duplicado)
              sino
                añadir a la GEST la dupla <símbolo, dirección relativa + csaddr>
            fin
          csaddr = csaddr + cs_long
        fin
    fin
fin
```

Si suponemos que el enlazador-cargador obtiene una región de memoria que comienza en la posición 10000_{16}, la tabla de global de símbolos externos que se obtiene luego de la primera pasada a la cadena de entrada del ejemplo 8 es la siguiente

Sección de control	Símbolo	dirección	Longitud
PROGA		10000	48
	A1	10040	
	A2	10044	
PROGB		10048	38
	B1	10078	
	B2	1007C	
PROGC		10080	24
	C1	100A0	

La segunda pasada del enlazador-cargador, una vez conocida la dirección de cada símbolo externo, ejecuta la carga de las secciones de control en memoria y el enlace de dichas secciones. Se usa la variable CSAADDR de la misma manera que en la primera pasada. El contenido de cada registro de texto es leído y cargado en la posición de memoria CSAADDR + *dirección de inicio especificada en el registro*. Luego se procesa el diccionario de relocalización de la sección de control actual. Por cada entrada encontrada, se toma la dirección relativa especificada y se le suma a CSADDR, para obtener la dirección de la instrucción cuyo campo de dirección necesita ser relocalizado y/o enlazado. Se ubica el símbolo especificado en dicha entrada en la GEST y se le suma su valor al campo de dirección de la instrucción, estableciéndose de esta manera el vinculo entre la instrucción y el símbolo al cual hace referencia.

Finalmente, el enlazador-cargador le transfiere el control al programa cargado y este inicia su ejecución. El registro de fin de cada sección de control indica cual es el punto de entrada. Si mas de una sección de control especifica un punto de entrada, se transferirá control al ultimo encontrado. Es responsabilidad del programador solo especificar un punto de entrada en la sección de control que define al programa principal.

El algoritmo siguiente muestra, de manera general, el flujo de la segunda pasada de nuestro enlazador-cargador.

```
comenzar
  csaddr = progaddr                    %dirección de carga
  para cada sección de control hacer
    comenzar
      leer registro de encabezado
      cs_long = longitud de la sección de control
      para cada entrada del diccionario de texto hacer
        comenzar
          leer registro de texto
          almacenar código objeto en la dirección csaddr + dirección relativa en el registro de texto
        fin          % fin de la carga de esta sección de control
      para cada entrada en el diccionario de relocalización hacer
      comenzar
        buscar símbolo en GEST y dir_link = dirección del símbolo según GEST
        dir_aux = dirección relativa en el diccionario de relocalización + csaddr
        al campo de dirección de la instrucción en la dirección dir_aux sumarle dir_link
      fin
      si se especifico dirección en el registro de fin entonces
        dir_inicio = dirección en el registro de fin + csaddr
      csaddr = csaddr + cs_long
    fin
  transferir control a la dirección dir_inicio
fin
```

Actividades: Ud. deberá ejecutar el enlace y la carga de la cadena de secciones de control del ejemplo 8, siguiendo paso a paso los algoritmos propuestos.

Refinamientos.

El enlazador-cargador presentado posee una estructura sencilla, que permite ilustrar el funcionamiento general de este tipo de procesadores de lenguaje. Sin embargo, un análisis más exhaustivo del mismo expone algunas debilidades, si bien es cierto que funciona correctamente.

El lector podrá haber observado que el enlazador-cargador propuesto genera un ejecutable en memoria principal, por lo que este proceso debe repetir cada vez que se desee ejecutar una misma aplicación formada por varias secciones de control.

Otro aspecto bastante sutil, pero de gran importancia es la obtención de memoria a través del Sistema Operativo. ¿Qué cantidad de memoria solicita el enlazador-cargador?. Debe ser una cantidad que permita, la carga en posiciones de memoria consecutivas de todas las secciones de control que conforman la aplicación. Sin embargo, ¿Cómo determina el enlazador-cargador ese tamaño?. Bien, para ello se deberá leer en una pasada previa los registros de encabezado de todas las secciones de control que conformen la cadena de entrada; con lo que ya se tienen tres pasadas sobre la cadena de entrada!.

Este ultimo aspecto tiene una fácil solución en los ambientes donde se implante el concepto de memoria virtual paginada bajo demanda. En dichos ambientes cada aplicación obtiene para su ejecución un espacio de direcciones virtuales continuo, cuyo tamaño máximo depende de la capacidad de direccionamiento del procesador. Este espacio de direcciones se divide en dos regiones, una para el Sistema Operativo, y otra región de usuario para la carga de la aplicación. El enlazador-cargador puede conocer de antemano la dirección de comienzo de esta ultima región, o en recibir esta información del mismo Sistema Operativo. Una vez conocido, el enlazador-cargador puede realizar su trabajo sobre memoria virtual; es decir, el enlace y la carga se realiza sobre una región virtual. Esta región se divide en páginas y el Sistema Operativo, dependiendo de su política de asignación de páginas, cargará algunas de ellas en marcos de página en memoria real, cargando a su vez todas en un archivo de páginas o de intercambio. Es interesante señalar que, dado que una página puede cargarse en cualquier marco de página disponible, en tiempo de ejecución se realiza otra función de relocalización de las direcciones virtuales generadas por el programa. En esta ocasión, este enlace lo lleva a cabo el mecanismo de traducción dinámica de direcciones del computador.

Muchos sistemas de computación presentan tanto un enlazador como un cargador. El primero realiza solo las funciones de enlace y se conoce a veces con el nombre de link-

editor. Este procesador de lenguaje realiza las funciones de enlace sobre un espacio de direcciones lógico o hipotético cuya dirección de inicio es 0. En este espacio de direcciones, asigna direcciones lógicas a cada uno de los símbolos tipo I en las tablas de símbolos externos de cada sección de control, y generará así la GEST. Luego, en una segunda pasada, resuelve las referencias externas en los campos de dirección de las instrucciones que así lo ameriten, para lo cual usa el diccionario de relocalización de cada sección de control. La salida de este enlazador es un módulo ejecutable con un solo registro de encabezado, el cual indica el nombre de la aplicación enlazada y su tamaño, los registros de texto de todas las secciones de control, la tabla global de símbolos externos, y un gran diccionario de relocalización, conformado por los diccionarios de este tipo de cada sección de control. De esta forma, la función de enlaces de secciones de control y generación de un ejecutable se realizaría una sola vez. La estructura general de un ejecutable se muestra a continuación.

Encabezado (nombre y tamaño del ejecutable, etc.)
Diccionario de texto (registros de texto)
Diccionario de Global de Símbolos Externos
Diccionario de Relocalización
Fin del modulo ejecutable(punto de entrada)

Otro programa, el cargador, se encargará de la carga en memoria del modulo ejecutable, y su relocalización. El cargador puede leer el registro de encabezado del modulo ejecutable y determinar el tamaño del mismo, y solicitar así una región apropiada al Sistema Operativo. Una vez asignada la región, puede cargar en ella el contenido de todos los registros de textos, para luego proceder a la relocalización según lo señale el diccionario de relocalización. Finalmente, transfiere control al ejecutable cargado y relocalizado

Actividades: Modifique el enlazador-cargador presentado, para que realice las actividades de enlace y carga independientemente.

EPILOGO

Durante este trabajo se ha desarrollados una introducción al estudios de varios tipos de procesadores de lenguaje. Este viaje se inició con la presentación de la máquina hipotética HALC, y culminó con el desarrollo de un enlazador-cargador para este tipo de sistemas; se plantearon adicionalmente algunas posibilidades de refinamiento. Espero haber apoyado al estudiante en el aprendizaje de estos temas. Y mejor me sentiría, si logro despertar en el estudiante la inquietud por profundizar en estos temas.

Como comenté en la introducción a este trabajo, el tema de ensambladores, enlazadores y cargadores es bastante amplio, y quedan por cubrir tópicos, algunos de ellos fascinantes, como enlace dinámico de programas, desensambladores y debuggers. Todos estos son tópicos que ameritan más tiempo del que se dispone en los cursos de Estructura y Organización del Computador. No deja de motivarme el investigar sobre ellos y, quien sabe, producir en el futuro otra versión de este documento.

Una versión mas exhaustiva de este trabajo debería cubrir, además de los temas mencionados, casos de estudio que expliquen como se realizan las funciones de ensamblaje, enlace y carga en sistemas reales como el Microsoft Assembler, o en Sistemas Operativos como Unix/Linux y Windows. Este es el reto, y gracias a Dios, las posibilidades de nuevos desarrollos no culminan aquí. Por lo contrario, se abre un abanico de nuevas posibilidades, no solo en plano individual, sino también en el colectivo, incorporando a estudiantes motivados que gusten de desarrollar trabajos de investigación en esta área. Espero que la semilla de la curiosidad siempre nos motive. El ser humano aprende en la medida en que participa en el descubrimiento y en la investigación.

He preferido estudiar los libros
antes que a los hombres.
Francis Bacon

APENDICE

TABLA ASCII

Caracteres no imprimibles				Caracteres imprimibles										
Nombre	Dec	Hex	Car.	Dec	Hex	Car.	Dec	Hex	Car.	Dec	Hex	Car.
Nulo	0	00	NUL	32	20	Espacio	64	40	@	96	60	`
Inicio de cabecera	1	01	SOH	33	21	!	65	41	A	97	61	a
Inicio de texto	2	02	STX	34	22	"	66	42	B	98	62	b
Fin de texto	3	03	ETX	35	23	#	67	43	C	99	63	c
Fin de transmisión	4	04	EOT	36	24	$	68	44	D	100	64	d
enquiry	5	05	ENQ	37	25	%	69	45	E	101	65	e
acknowledge	6	06	ACK	38	26	&	70	46	F	102	66	f
Campanilla (beep)	7	07	BEL	39	27	'	71	47	G	103	67	g
backspace	8	08	BS	40	28	(	72	48	H	104	68	h
Tabulador horizontal	9	09	HT	41	29	)	73	49	I	105	69	i
Salto de línea	10	0A	LF	42	2A	*	74	4A	J	106	6A	j
Tabulador vertical	11	0B	VT	43	2B	+	75	4B	K	107	6B	k
Salto de página	12	0C	FF	44	2C	,	76	4C	L	108	6C	l
Retorno de carro	13	0D	CR	45	2D	-	77	4D	M	109	6D	m
Shift fuera	14	0E	SO	46	2E	.	78	4E	N	110	6E	n
Shift dentro	15	0F	SI	47	2F	/	79	4F	O	111	6F	o
Escape línea de datos	16	10	DLE	48	30	0	80	50	P	112	70	p
Control dispositivo 1	17	11	DC1	49	31	1	81	51	Q	113	71	q
Control dispositivo 2	18	12	DC2	50	32	2	82	52	R	114	72	r
Control dispositivo 3	19	13	DC3	51	33	3	83	53	S	115	73	s
Control dispositivo 4	20	14	DC4	52	34	4	84	54	T	116	74	t
neg acknowledge	21	15	NAK	53	35	5	85	55	U	117	75	u
Sincronismo	22	16	SYN	54	36	6	86	56	V	118	76	v
Fin bloque transmitido	23	17	ETB	55	37	7	87	57	W	119	77	w
Cancelar	24	18	CAN	56	38	8	88	58	X	120	78	x
Fin medio	25	19	EM	57	39	9	89	59	Y	121	79	y
Sustituto	26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
Escape	27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
Separador archivos	28	1C	FS	60	3C	<	92	5C	\	124	7C	|
Separador grupos	29	1D	GS	61	3D	=	93	5D	]	125	7D	}
Separador registros	30	1E	RS	62	3E	>	94	5E	^	126	7E	~
Separador unidades	31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

BIBLIOGRAFIA

1. Beck L..(1997). **System Software: An introduction to System Programming**. Addison Wesley.

2. Franek F. (2004). **Memory as a Programming Concept in C and C^{++}**. Cambridge

3. Levine J. (2000). **Linkers and Loaders**. Morgan Kaufmann Publishers.

4. Saloman D. (1993). **Assemblers and Loaders**. Prentice Hall.

5. Tanenbaum A. (2006). **Structured Computer Organization**. Prentice Hall.