UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

COMPILADOR PARA SISTEMAS DIGITALES

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A N :

MARIA DEL ROSARIO GONZALEZ AGUIRRE

OMAR ARTURO MARTINEZ HERNANDEZ

DIRECTOR DE TESIS: ING. MARTIN PEREZ MONDRAGON

MEXICO, D. F. 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE

INTRODUCCION

La tecnología **VLSI** (*Very Large Scale Integration*) ha vencido la extensión que pueden ocupar cientos de miles o aun millones de transistores integrados en un sólo circuito integrado de silicio, siendo esto hoy en día la clave para el diseño eficiente de sistemas electrónicos. Por consiguiente, los problemas de manufactura de circuitos integrados y diseño de sistemas han incrementando convenientemente su complejidad y como consecuencia una amplia variedad de tópicos en el **Diseño Asistido por Computadora** (*CAD*) han emergido. Este es un período donde nadie puede ser un especialista en todos los tópicos en **CAD** orientados a **VLSI**.

El diseño de un circuito integrado como el de un sistema puede ser especificando diferentes niveles de espacio jerárquico, desde una representación de alto nivel hasta una de bajo nivel de geometría. En las estrategias de diseño tradicional el diseñador especifica la función de cada circuito integrado en un nivel más alto. Siguiendo una serie de refinamientos sucesivos el diseño es preparado para transformarlo en un formato de bajo nivel. Si los pasos de transformación sucesiva son realizados a mano, el procedimiento de diseño es tedioso, propenso a error y requiere demasiado tiempo. Los **"Compiladores para Sistemas Digitales"** serán en el futuro capaces de automatizar el proceso de diseño, de tal modo que el costo y tiempo disminuirán dramáticamente. En su última forma los Compiladores para Sistemas Digitales aceptarán un funcionamiento específico y generarán la representación geométrica de la organización de un circuito integrado de alto desarrollo. Estas herramientas permiten el más alto grado de automatización en el proceso de diseño.

Uno de los primeros acercamientos al área de los Compiladores para Sistemas Digitales es el **"Bristle Block System"** (*Sistema de Bloque Encrespado*), este se originó basandose en el trabajo de tesis de doctorado de el *Dr. Johannsen en Caltech*. Johannsen introduce un concepto de bloque donde las celdas son tratadas en una primera fase como cajas negras de estructura jerárquica y donde sólo el tamaño y posición de las conexiones externas (pines) son conocidas. Una arquitectura es seleccionada y el usuario tiene que describir su sistema en términos de los componentes de dicha arquitectura.

Un enfoque más poderoso lo encontramos en el sistema **MACPITTS**, que es una herramienta de generación de **"layout"** desarrollada por Siskind, Southard y Crouch en **"MIT Lincoln Laboratory"**. **MACPITTS** y su sucesor **METASYN** emplean un lenguaje de diseño funcional en base a **LISP**, el cual es interpretado por un programa **LISP**. El sistema consiste de diversas herramientas de generación de componentes y es capaz de generar el layout en una arquitectura destino flexible relativa a una plataforma.

ARSENIC, es un sistema experimental, que se estudia en la Universidad de Illinois, basado en un método de jerarquía recursiva para derivar la organización completa del sistema mediante refinadas decisiones. *P. ej.*, la falta de información en un sistema más abajo del nivel de bloque y - **si esta no está especificada en el nivel siguiente hacia abajo** - el mecanismo de referencias descenderá en la estructura de árbol hasta que la información derivada sea encontrada. Las decisiones entre refinamiento alternativo son tomadas por medio de una estrategia de **"look-ahead"**.

El primer Compilador para Sistemas Digitales ha sido desarrollado por Denyer, Renshaw y Bergman en la Universidad de Edinburgh[11], es una herramienta restringida al campo de las arquitecturas 'bit-serial' en el procesamiento digital de señales. El lenguaje de entrada consiste de un conjunto fijo de operadores y procedimientos organizados en una plantilla.

En la actualidad se encuentran en la fase de desarrollo algunos de estos sistemas que han cambiado de investigación de laboratorios en aplicaciones comerciales. No obstante la aceptación de la industria es aun baja.

En la presente trabajo de investigación el **Capítulo I** versa sobre la definición de los conceptos generales para **modelado de Sistemas Digitales**, e introduce el concepto de **Lenguajes de Descripción de Hardware.**

El **Capítulo II** tratará del proceso de **Diseño Conceptual** orientado al Diseño de Sistemas Digitales con **Lenguajes de Transferencia de Registro (RT)**, mostrando un panorama general de características.

Dentro del **Capítulo III** se presenta una introducción a la **Teoría de Compiladores** para Lenguajes de Programación de alto nivel, siendo este el punto de referencia que permite establecer el concepto de **Compilador para Sistemas Digitales** e introducir un esquema de construcción.

El **Capítulo IV** muestra la implementación de esta herramienta a través de un ejemplo de diseño. Este capítulo también incluye la **gramática del Lenguaje RT** y el **código fuente del compilador** implementado.

"La Ciencia es la Estética de la Inteligencia"

Gaston Bachelard
Filósofo Francés
(1884-1962)

MODELADO Y LENGUAJES DE DESCRIPCION DE HARDWARE

El objetivo principal de este trabajo es proporcionar una herramienta que permita diseñar sistemas digitales como una disciplina sistemática basada en ciertos conceptos fundamentales. La tarea de dicha herramienta, a la cual hemos intitulado **Compilador para Sistemas Digitales**, es recolectar toda la información referente al sistema y ejecutar un rango de verificación de errores de diseño, tales como dimensiones de los registros así como también las conexiones de cada pin en cada circuito integrado utilizado y señalar cualquier pin que no le haya sido asignada conexión.

Para poder llegar a realizar esta tarea es necesario discutir los medios a través de los cuales el diseñador deberá proporcionar una descripción funcional de el sistema, así como las técnicas de verificación del funcionamiento y representación de dicha descripción a través de una simulación lógica.

I.1 ESPECIFICACION DE MODELADO

El diseño digital envuelve un número de pasos. Desde las especificaciones funcionales del circuito, hasta los documentos verificados de la fabricación (*"layout" e interconexión de módulos primitivos, si CIs en una tarjeta o transistores en chip*), el circuito experimenta varias descripciones en diferentes niveles de abstracción. Investigaciones en **"Compiladores para Circuitos Integrados"** intentan proporcionar herramientas las cuales directa y automáticamente producen la elaboración de documentos desde las especificaciones funcionales. Por algunos años los diseñadores de circuitos han tenido que vivir con la necesidad de refinar sus diseños en una forma **"stepwise"** y así manipular diferentes modelos, posiblemente envolviendo varios modelos principales.

Entre las especificaciones funcionales de un circuito digital y la máscara para estos, el seguir niveles de diseño (Figura 1.1) ha sido ampliamente aceptado y reconocido, aunque algunos de ellos pueden ser escalonados sobre una tarea de diseño particular, dependiendo de su complejidad y la disponibilidad para construcción de bloques:

- nivel de sistema
- nivel arquitectural
- nivel de transferencia de registro
- *nivel* ***"gate"***
- *nivel* ***"switch"***
- *nivel* ***"circuit"***

Durante el proceso de diseño, usualmente el circuito es descrito bajo las consideraciones de los diferentes niveles de abstracción. Estos niveles permiten la manipulación de información sintáctica, mientras que los niveles más cercanos a la implementación tecnológica permiten verificaciones precisas y detalladas. A fin de evitar una excesiva cantidad de datos, cuando solo una pequeña parte de el circuito es referenciada,

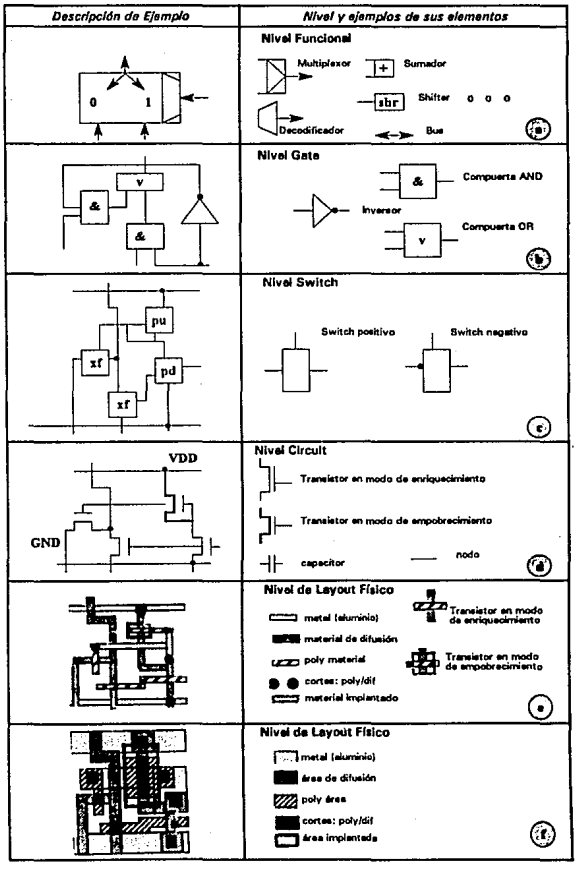


Figura 1.1 Niveles de Diseño

es altamente recomendable combinar diferentes niveles de descripción dentro de un solo modelo. Un progreso considerable en este aspecto es llevado por el surgimiento de lenguajes multinivel, desarrollados de modo que puedan cubrir diferentes niveles de modelado, con un "**kernel**" común de primitivas para todos los niveles.

I.2 NIVELES DE DISEÑO

Definición y Características

- **Nivel de sistema**

En un nivel de sistema se especifica sincronización y comportamiento funcional de sistemas digitales, además de la interconección de componentes, sin especificar una implementación detallada. El ruteo de mensajes en una red, o sincronización de procesos comunicados paralelamente son ejemplos típicos.

La parte operativa es expresada con asignaciones a variables abstractas (*asignándole el tipo VARIABLE)*. Las operaciones pueden ser permanentemente válidas, o su ejecución depende de la parte de control en el modelo.

La parte de control es descrita usando primitivas que frecuentemente son equivalentes a un modelo de control gráfico.

- **Nivel de Arquitectura de Hardware y Nivel de Transferencia de Registro (RT)**

Los módulos escritos en estos niveles especifican los cambios de estado de los elementos de hardware, pero no los detalles de su trayectoría de datos. Una descripción puede ser:

Síncrona
El tiempo es expresado en términos de ciclos de reloj, uno o varios relojes son provistos desde el exterior a través de una interfase. Como el estado del sistema puede cambiar en solo un pulso de reloj se introdecen circuitos combinacionales a fin de estabilizar con respecto al ciclo del reloj más rápido.

Asíncrona
Los retrasos son expresados en términos de unidades de tiempo, y los valores pasados de "**carriers**" pueden ser referenciados en expresiones. Un pulso de sincronización puede ser el flanco de subida o de bajada de un carrier Booleano local y no solo un pulso externo de reloj.

La parte funcional de una descripción puede representar:

Circuitos combinacionales, los cuales no tienen elementos de memoria. Todas las declaraciones son conexiones de expresiones posiblemente complejas a pines o cables de conexión llamados **TERMINALES.** Todas las declaraciones están permanentemente activas.

Circuitos secuenciales, que son aquellos que tienen elementos de memoria. Los carriers del tipo **LATCH** o **REGISTRO** son inicializados bajo las condiciones de un nivel booleano o flanco de subida/bajada.

Típicamente el control es modelado a través de un autómata. Un conjunto de acciones concurrentes que dependen de un estado dado son etiquetadas con el estado del identificador. El secuenciamiento es modelado por el cambio de estado en el autómata, lo cual toma un ciclo de reloj.

- **Nivel Gate**

En este nivel una descripción es una red de compuertas y buses interconectados.

Los componentes primitivos corresponden a compuertas lógicas usuales (***NOT, NAND, AND, NOR, OR, XOR, XNOR***), la compuerta de transferencia (***TG***), compuertas tres estado (***TNOT, TAND, TOR, TXOR, TXNOR, TNAND, TNOR***) y modelos simples de bus.

Sus elementos de interfase son direccionales. Son usualmente carriers que toman valores en 3 o 4 estados lógicos.

Los diferentes tipos de atributos de sincronización (*valores típico, mínimo y máximo de retrasos de subida y bajada*) son parámetros para el cual los valores por default pueden ser definidos.

- **Nivel de Switch**

En este, la descripción es una red de transistores y nodos interconectados; los flancos no son orientados. El nivel de switch está especialmente adaptado para tecnología **MOS**. Los tres tipos de transistores **MOS** (*canal N, P y "depletion"* o empobrecimiento) son dados como componentes primitivos, con un parámetro de graduación: **NTRANS, DTRANS, PTRANS.**

Sus tres elementos de interfase no son direccionables (***GATE, DRAIN, SOURCE***), pero su valor es extraído para lógica discreta *(usualmente tres o más estados)*.

Otros componentes primitivos son el **NODO**, el cual modela una interconexión punto, con un número de variable de interfase no direccional **"PINS"**, y la **"INPUT"**, un caso especial de nodo el cual es conectado a entradas primarias o tierra (**GND**), o potencia (V_{DD}).

- **Nivel Circuit**

En el nivel eléctrico el usuario se encuentra interesado en determinar la corriente y el voltaje en cables específicos del circuito como una función del tiempo. El modelo es escrito en términos de las leyes de la física, usando el sistema de unidades **MKS**. La principal característica del nivel circuit, comparada con la de los otros niveles, es que todos los carriers toman valores de tipo **REAL**, y que una descripción representa un modelo continuo del circuito.

El funcionamiento es introducido por la interconexión de primitivas y componentes definidos por el usuario. Los componentes primitivos son, por ejemplo, los modelos de varios transistores y de los dipolos elementales los cuales pueden ser encontrados en circuitos eléctricos: resistencias *(R)*, conductancias *(G)*, fuentes de voltaje *(E)*, fuentes de corriente *(J)*, capacitancia *(C)* e inductancia *(L)*. Sus interfases son hechas de pines no direccionables.

Dos funciones "i"y "v" son ligadas a cada pin P, para regresar la intensidad de la corriente en P, así como sus voltajes. Para cada componente primitiva cuyos pines de interfase $P_1, P_2, ..., P_n$ definen una ecuación en $i(P_j)$, y $v(P_j)$, acordada por las leyes eléctricas para el componente. Además de que el compilador genera automáticamnte (*Leyes de Kirchhofk*):

$$i(P_1) + i(P_2) + ... + i(P_n) = 0$$

La notación, computacionalmente hablando, para describir sistemas de hardware en cualquier nivel de modelado o diseño es llamada ***Lenguajes de Descripción de Hardware (HDL)***, los cuales son muchos y muy variados siendo clasificados en numerosos grupos basados en el alcance e intensión de su uso.

I.3 LENGUAJES DE DESCRIPCION DE HARDWARE

I.3.1 Características de Lenguajes de Descripción de Hardware

Los **HDL** son usados en simulaciones de alto nivel, generación de silicon compilers, generación automática de patrones de prueba, especificación de hardware, y documentación, auxiliar en la enseñanza de los principios de hardware digital. Algunos **HDLs** tienen mnemónicos similares a los del lenguaje de programación **PASCAL**. En la clasificación de estos lenguajes podemos distinguir dos grandes clases: **lenguajes no procedurales y lenguajes algorítmicos**.

Existe una fuerte tendencia hacia **HDLs** multinivel. Un gran número de estos han sido propuestos, dado que en ocaciones pequeñas piezas de una descripción no pueden ser expresadas en el nivel **RT**, así que este

"resto" solo puede ser representado usando primitivas de otro nivel como el gate o cualquier otro. Otra razón es el bus, que es un importante recurso arquitectural en niveles altos de descripción del sistema. La característica de multinivel nos permite emplear el mismo lenguaje para diseño **"top down "** mediante una especificación que puede ser sucesivamente refinada hasta un concepto más detallado del hardware.

Dentro de las características principales de estos lenguajes se encuentran las siguientes:

Comprensibilidad. Cuando existe multinivel, los lenguajes **HDL** son sustancialmente mas fáciles de aprender que lenguajes de programación de alto nivel, tales como Pascal. Muchos lenguajes utilizan palabras reservadas con mnemónicos diseñados hábilmente, de modo que las descripciones en **HDLs** son por si mismas documentables y alcanzan un alto grado de legibilidad y comprensibilidad. La comprensibilidad es aun más marcada cuando una gráfica interactiva se encuentra disponible al usuario. Esta interfase gráfica es la implementación de un **HDL** gráfico.

Primitivas del lenguaje. La potencia y flexibilidad de un **HDL** están determinadas por el repertorio de sus primitivas. Podemos distinguir desde primitivas estructurales hasta funcionales. Las primitivas estructurales dentro de un **HDL** normalmente son uniformes a lo largo de todos los niveles de abstracción, todos ellas es lo mismo, no importa si son utilizadas en el nivel **RT**, nivel gate o en un nivel switch, y aun en nivel de circuito y layout simbólicos. Las primitivas estructurales pueden ser subdivididas en módulos de definición de características *(tal como funciones, módulos y celdas)*, y, en notaciones para especificar interconexión. La variedad de primitivas funcionles depende de los niveles metodológicos cubiertos por el **HDL**. Dentro de estos niveles los **HDL** pueden extenderse hasta el nivel de switch. En este nivel, por ejemplo, se puede desarrollar un método que modele el sistemas de bus de modo que implemente sistemáticamente los principios generales del circuito de layout simbólico.

Módulos de celda con capacidad de planta. Por lo menos un **HDL** soporta eficientemente el diseño **VLSI** estructurado y la integración de **HDLs** basados en herramientas **CAD** dentro de el diseño físico a través de la definición de celdas y características similares a las de planta. Para este propósito un **HDL** necesita la declaración topológica de los atributos de cada celda para soportar la generación automática de la interconexión como confinamiento o agrupamiento de celdas. También requiere de operadores que expresen el tipo de confinamiento, vertical u horizontal; además de transformaciones elementales para rotar y generación de espejo, un muy poderoso sublenguaje topológico para confinamiento de expresiones debe ser capaz de describir consistentemente la estructura de grupos de celdas un poco complejas incluyendo algunos niveles anidados.

Descripción concisa de patrones de ruteo. Normalmente los modelos alambrados son sujetos de otras disciplinas dentro de la ciencias de diseño: entrada esquemática de sistemas y algoritmos de ruteo. Por supuesto, algún alambrado arbitrario puede ser expresado en un **HDL** por medio de descripciones y ruteo de celdas definidas por el usuario. Sin embargo, tipos específicos de alambrado son consisamente

predefinidas mediante primitivas de algunos **HDLs**. Una amplia extensión de ejemplos en **HDLs** son las descripciones alambradas para: cambiar el ancho de la trayectoria *(juxtaposición de trayaectorias para crear una más amplia trayectoria además de suscribir para dividir una trayectoria)*, y para patrones de ruteo los cuales preservan el ancho de la trayectoria. Esto puede ser dividido en subgrupos: **conexiones directas** las cuales no afectan la secuencia de los bits, cajas de ruteo definidas por el usuario, y **alambrado de operadores,** los cuales readaptan la secuencia de bits algoritmicamente. Dentro de un **HDL** tales operadores son muy poderosos. Con solo el nombre del operador, especificación del ancho de la trayectoria, y algunas veces un solo parámetro entero adicional, el modelo completo de alambrado complejo puede ser expresado y utilizado dentro de un simulador.

Refinamiento "Step wise" usando un HDL. La capacidad de refinamiento es una importante propiedad del lenguaje para lograr sus usos como lenguaje de diseño. Por tanto no senecesita cambiar el lenguaje en planeación top down desde una especificación puramente funcional para una más detallada descripción, la cual entonces puede ser empleada como entrada al diseño lógico y diseño físico. Se puede utilizar el mismo lenguaje y las mismas herramientas para analizar y sintetizar una descripción conceptual a través de una notación estructural/topológica/funcional que lleva hábilmente consigo todas las ideas arquitecturales; para capacidad de prueba, diseño estructurado, etc. más allá del equipo de implementación de silicio.

HDL usado como un Cálculo del Diseñ. El refinamiento step-wise permite que el diseño top-down pueda ser ejecutado sin cambiar el lenguaje. Esto es un requerimiento si se desea emplear el lenguaje como diseño de cálculo a fin de alcanzar las metas de diseño por medio de una secuencia de manipulaciones algebraicas. Este tipo de lenguaje solo puede ser un medio de expresar tales reglas algebraicas, sin embargo esto no puede ser álgebra en si misma. Este medio, sin embargo, es poderoso y adecuado para la exploración efectiva de muchas áreas de aplicación experimentando con arquitecturas alternativas y estructuras.

I.3.2 CLASIFICACION DE LENGUAJES DE DESCRIPCION DE HARDWARE

Existe una necesidad de auxiliares de navegación para entrar en el área de documentación y diseño de sistemas de hardware y hardware/software sin perdernos. Pretendemos dar una presentación general de la terminología y proporcionar esquemas transparentes de clasificación. Esto intenta ser una guía en movimiento a través de más de una decena de niveles de abstracción y varias áreas de aplicación de tales lenguajes.

Sin embargo, antes de hablar acerca de clasificación de lenguajes tenemos que decir que no existe un uso uniforme del término **"Computer Description Hardware Lenguage" (CHDL)**, además de que la terminología en el campo no es empleada de una manera uniforme.

Los CHDL son organizados en grupos, de los cuales a continuación se presentan los más comunes:

PL Lenguaje Procedural *(lenguaje de programación)*. Estos son notaciones para expresar procedimientos o algoritmos a través de estructuras de control y estructuras de datos con secuencias deseadas o acciones reales o instrucciones.

HL Lenguajes de Hardware. Son notaciones gráficas y/o textuales, las cuales pueden ser empleadas para la descripción de estructuras, funcionamiento y morfología *(geometría y/o topología)* de hardware en uno o más niveles de abstracción. Puesto que existen lenguajes y aplicaciones *(diseño de procedimientos)* los cuales requieren la representación de procedimientos y su hardware subyacente en el mismo tiempo, quisieramos introducir el término **HSL** para lenguajes de SoftwareHardware:

HSL Lenguajes de Hardware/Software *(fusión de PL y HL)* y **DHSL Lenguaje de Hardware digital/Software**. En muchos casos los DHSL son usados en el campo de estructuras de cómputo, la cual es una subárea de la Ingeniería de Hardware. La razón de esto es que, la fusión de PL y HL dentro de la misma descripción es un requerimiento típico de una aplicación para procesador basado en sistemas digitales. Si tal lenguaje especializado para una aplicación y tal procesador son basado en un solo sistema, podemos dar un nombre diferente para esta clase de lenguajes:

CHL Lenguaje de Hardware de Cómputo y CHDL Lenguaje de Diseño de Hardware de Cómputo *(en vez de CHL también el término CHDL Lenguaje de Descripción de Hardware de Cómputo es frecuentemente usado en literatura)*. Una computadora es un tipo especial de hardware, normalmente particionado en procesador, memoria y/o interfase, donde la estructura de el procesador es caracterizada por su dicotomía dentro de una parte de control y una operativa. Los CHLs proporcionan características especiales de lenguaje para modelar esta dicotomía. Este grupo se encuentra conformado por los siguientes subgrupos:

DL Lenguajes de Diseño *(lenguajes multinivel con capacidad de refinamiento)*
SL Lenguajes de Especificación *(lenguajes descriptivo)*
HDL Lenguaje de Diseño de Hardware
DHDL Lenguajes de Diseño de Hardware Digital

El término '**Lenguaje de Hardware de Cómputo**' *(CHDL)* es muy empleado frecuentemente en acercamientos a niveles metodológicos más altos, como el nivel funcional o nivel conductual.

1.3.3 CRITERIO DE CLASIFICACION

Para la clasificación de los lenguajes enfocados a la especificación y manipulación de descripciones de sistemas de hardware y de software/hardware presentada, se tomaron en cuenta cuatro criterios principales:

el **nivel** *(nivel de abstracción)*, el **área** *(área de aplicación)*, la **dimensión de la notación** y el medio fuente utilizado.

- **Niveles de abstracción**

En la ciencia de la computación e ingeniería muchos más niveles de abstracción son necesarios para una descripción exhaustiva de sistemas y subsistemas que en cualquier otra disciplina científica. Cuando usamos una estrategia top-down para caminar a través de los niveles, y cuando empezamos en el mundo de las aplicaciones computacionales, encontramos redes abiertas, redes de área local, sistemas multiproceso, partes de control de procesos y parte de datos de un proceso, etc. hasta que finalmente llegamos a un nivel, donde transistores individuales son descritos en detalle. Más de una docena de niveles de abstracción pueden ser identificados en esta forma. Este hecho nos conduce a elegir un esquema de clasificación jerárquica que nos permite distinguir las siguientes clases de niveles de abstracción:

- niveles transaccionales
- niveles procedurales
- niveles semiprocedurales
- niveles no procedurales

En **niveles de abstracción no procedurales** solo se describe el hardware. El término 'no procedural' indica que el software no está incluido en tal descripción, por consiguiente la especificación de secuenciamento (*estructuras de control*) no es sujeto de estos niveles. Las descripciones no procedurales son principalmente combinacionales. También la asignación de los valores a los registros puede ser sujeto de descripciones no procedurales. Las trayectorias de enlace de datos pueden ser descritas en estos niveles mientras que no se especifiquen estructuras de control. La semántica de objetos no procedurales es diferente a la de los procedurales. En la "ejecución" (*simulación*) de una descripción no procedural, el orden de esta sigue un modelo fundamentalmente diferente: Durante cada ciclo de ejecución cada sentencia siempre es explorada para averiguar si esta debe ser ejecutada en ese momento, en otras palabras, cada instrucción especificada tiene una etiqueta y cada expresión de la etiqueta (*referencias a reloj, señales de habilitación, etc*) es verificada por su valor verdadero durante cada ciclo de la etiqueta. Siempre que la etiqueta es verdadera, la instrucción debe ser ejecutada. Todas estas descripciones son llamadas no procedurales, las cuales están en niveles más bajos, que descripciones procedurales.

Los **niveles semiprocedurales** son niveles intermedios entre niveles procedurales y no procedurales. Para una descripción la cual es primariamente no procedural, el concepto de máquina de estado finito y características para especificar su secuencia de estados son adicionados. Sin embargo, programas con notación procedural no son usados en este nivel.

Los **niveles procedurales** son esos niveles (Figura 1 renglones 2 al 6), donde el hardware y software, o hardware y estructuras de control, respectivamente, son descritas simultáneamente. Durante un diseño top down de un sistema multiproceso encontramos los siguientes niveles de abstracción: el **nivel de concurrencia de procesos** (*muestra su sincronización y coordinación, p. ej., entre un programa concurrente corriendo en paralelo dentro de distintos procesadores*), el nivel donde un solo proceso secuencial es descrito (*p. ej., para un programa secuencial corriendo en un procesador*). El **nivel de microprograma**, donde la implementación de un procesador es descrita algoritmicamente *(mediante un microprograma)*, y finalmente el **nivel dicotómico**, donde la parte de datos/parte de control de cooperación dentro de el procesador es descrito en detalle. El 'nivel de guarda de microprograma' puede ser utilizado como un nivel de representación de un resultado intermedio en el diseño de un procesador desde un nivel de especificación de microprograma. Estos niveles describen paso a paso el funcionamiento, de modo que el tiempo de ejecución de una secuencia de instrucciones depende de la secuencia espacial de esas instrucciones en el texto del procedimiento.

Los **niveles transaccionales** (*niveles de enlace*) describen las relaciones de comunicación y comunicaciones eslabonadas entre computadoras dentro de una red. Los sujetos de este alto nivel de abstracción son: la descripción de la estructura de enlace, la descripción de canales de comunicación, los protocolos de comunicación utilizados, y la estructura de datos de mensajes a ser transmitidos. La descripción de un módulo en estos niveles deben tener características procedurales.

- **Dimensión de la notación**

Además de los niveles de abstracción, existen otras características importantes de descripciones, que son muy independientes de los niveles de asbtracción y sus aplicaciones. La primera de estas importantes características es expresar la dimensión de la información. Las tres posibles formas de expresar la dimensión de la información son las siguientes:

- información estructural
- información de comportamiento
- información morfológica

Las tres dimensiones son ilustradas a través de la Figura 1.3. En donde un programa es una notación de comportamiento y un sistema de entrada esquemática solo maneja información estructural, en tanto que un layout fisico es solo geométrico *(que es morfológico)*. Con respecto a dimensiones, una notación o descripción puede ser unidimensional (*p. ej. los formatos de sistemas de entrada esquemática*), bidimensional (*como estructural y funcional en el mismo tiempo*), o tridimensional.

Comportamiento

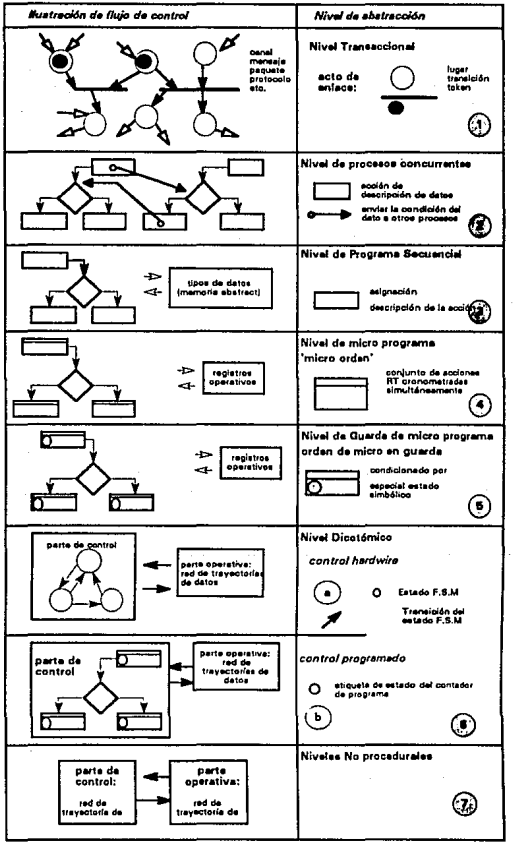


Figura 1.2 Niveles de abstracción

El comportamiento de un sistema, subsistema o componente es caracterizado por sus relaciones entrada/salida, o por sus relaciones estímulo/respuesta, La manera de como el comportamiento es expresado es el ingrediente más importante de una descripción para determinar a cual nivel de abstracción pertenece. La forma de la información acerca del comportamiento es muy dependiente del nivel de abastracción, mientras que estructura y morfología son relativamente niveles independientes. De este modo podemos distinguir las siguientes formas de comportamiento:

- **Comportamiento procedural**
  - Comportamiento paralelo
    - síncrono (*acciones síncronamente paralelas*)
    - concurrente (*procedimientos asíncronamente paralelos*)
  - Comportamiento secuencial
- **Comportamiento semi procedural**
  - Comportamiento dicotómico (*Funcionamiento RT incluyendo secuencias de estados*)
- **Comportamiento no procedural**
  - Comportamiento combinacional
  - Comportamiento RT (*redes de trayectorias de datos incluyendo registros*)

Dado que empleamos estas formas de comportamiento como criterio de clasificación subclases de niveles de abstracción han sido formadas. Las notaciones para *comportamiento concurrente* y *secuencial* son las que encontramos en los lenguajes de programación . Las notaciones para el **comportamiento síncronamente paralelo** son escontradas en lenguajes de microprogramación. El término **comportamiento dicotómico** es, el que se propone usar en lugar de el tradicional, pero engañoso término 'nivel funcional'. Algunos autores emplean el término **CHDL** (*Lenguajes de Descripción de Hardware de Cómputo*), o funcionamiento **CHDL** para estos lenguajes que cubren este nivel.

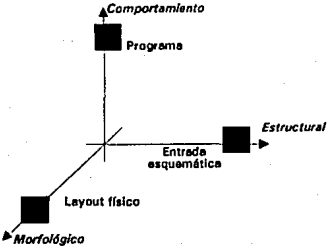


Figura 1.3 Dimensiones de la información

Estructural

La descripción estructural es una de las tres dimensiones de información en una descripción general de sistemas complejos. La forma del ingrediente estructural de una descripción dada en un nivel de abstracción particular es casi independiente de ese nivel. La descripción estructural pueden ser divididas en descripción

- Particionada
- Interfaseada
- Interconectada

La **descripción particionada** define cómo un sistema, o módulos son divididos en de submódulos a través de una estructura jerárquica. En la **descripción interfaseada** se especifica la vista externa de un módulo, el cual define el arreglo de los puertos y prepara a sus conexiones para los módulos externos; este especifica nombre y/o número de pin, formato y localización de puertos de un módulo dado, también la topología del conjunto de todos los puertos es una parte importante de información en interfaseada. En la **descripción interconectada** se especifican las conexiones entre los diferentes objetos de una descripción estructural a través de un **"net"**, que es un modelo de alambrado simple; que conecta dos o más puertos o pines que lleven la misma señal. Otro concepto que distinguimos en estas descripciones es un **"netlist"**, el cual es una lista que especifica todos los net dentro de un área de ruteo específica o modelo alambrado similar. El alambrado de una interconexión puede ser especificado por netlist o por notaciones gráficas entre módulos y componentes de el mismo nivel de anidación, entre un puerto periferial de un módulo y sus módulos internos y componentes (*para evitar confusión un conjunto de distintos net debe ser llamado "network"*).

Morfológica

Si utilizamos una notación gráfica para la descripción, o si la geometría o topología física tienen que ser transportada por la descripción, necesitamos además de la información estructural y de comportamiento un tercer tipo de ingrediente: la **información morfológica** que especifique las siguientes propiedades:

- forma y color
- colocación (geométrica) exacta
- colocación relativa (topología)

Emplearemos el término morfología para hablar de geometría y topología. Las descripciones topológicas son algunas veces abstracciones de información geométrica. *p. ej.*, el layout simbólico puede ser solo la abstracción de un layout físico. Tenemos que distinguir la morfología simbólica de la morfología física. Si en una presentación gráfica la forma y la posición de los objetos solo tiene el propósito de presentar un buen dibujo, la geometría será **geometría simbólica**, mientras que si la especificación geométrica transporta información acerca de objetos físicos, es llamada **geometría física**.

- **Medio fuente de un lenguaje**

Una importante característica de una descripción de un lenguaje es el medio fuente de presentación, forma textual o una forma gráfica. Si un lenguaje es empleado para la interacción hombre-máquina, de igual manera como usamos un lenguaje funcional para diseñar una pieza de hardware, una versión gráfica del lenguaje debe sustancialmente incrementar la productividad de el diseño comparada con su versión textual.

Una razón para este hecho es porque el canal de entrada de datos visuales es mucho más rápido, mucho más poderoso y mucho más inteligente que su canal de entrada de cadena de texto. Pero en el campo del diseño **VLSI**, esta no es la única razón. Porque de la falta de, los problemas de geometría y topología son mucho más importantes que en métodos clásicos de diseño de hardware. Por supuesto, el layout físico en una forma textual es completamente incomprensible para el humano, de modo que verificar uno es imposible. Sin embargo, también en niveles de abstracción mucho más altos representaciones gráficas son inevitables.

- **Area de aplicación y alcance de el lenguaje**

Muchos lenguajes son en sumo grado la orientación de metas y la meta para esta clase de lenguajes nunca ha sido cuestionada. Estos lenguajes frecuentemente son implementaciones de una filosofía o metodología de síntesis, análisis, extracción, simulación, verificación deductiva, especificación, captura del diseño, captura de esquemáticos y otras aplicaciones. De los requisitos de las aplicaciones podemos distinguir niveles simples de aplicación (*especificación, reglas de verificación, optimización, nivel simple de simulación, verificación de temporizado, activación de modelos de descripción, etc*) y aplicaciones multinivel. Estos son requeridos para las herramientas que traducen una descripción de un nivel a otro (*p. ej., síntesis, extracción, verificación deductiva, modelado*) si un lenguaje no separado es proporcionado como entrada (*lenguaje fuente*) y salida (*lenguaje objeto*). De este modo tenemos la alternativa de elegir entre un lenguaje multinivel y una familia de lenguajes separados. La elección óptima algunas veces depende de la intensión de la aplicación.

Como podemos concluir el hecho de utilizar un **HDL** reduce en gran medida la complejidad del diseño, ya que utilizan paquetes de vectores de bits en palabras, como en lenguajes de programación de alto nivel. Describen una trayectoria de un dato de 32 bits guardando sus valores de datos en una sola palabra que es procesada en seguida dentro de un simulador y otras herramientas. Además de que en el nivel **RT** se dispone de operadores potentes como el de la multiplicación, un equivalente a la altura de cientos de compuertas y muchas otras. Además de la flexibilidad de emplear un lenguaje **RT**.

Los **HDLs** deben de ser capaces de describir un hardware particular en diferentes niveles de abstracción. Es por esta razón que, utilizar un mismo lenguaje en el proceso de diseño puede empezar con un nivel de especificación muy alto de muy baja complejidad y después de varios pasos de refinamiento este puede ser finalizado con una compleja y detallada descripción de una solución concepto.

La Figura 1.4 ilustra la localización de un número de lenguajes y notaciones descriptivas similares dentro de nuestro esquema de clasificación, especialmente dentro del alcance de los niveles de abstracción descritos. Existen lenguajes de nivel simple como también lenguajes multinivel, los cuales cubren más de un nivel. *P. ej.*, la mayoría de los lenguajes **RT** también incluyen las primitivas booleanas de nivel gate. Sin embargo, también lenguajes barrocos son posibles, los cuales cubren diferentes niveles. **Pascal** es un lenguaje de nivel simple, solo envuelve procedimientos secuenciales. **Modula-2** es un lenguaje de nivel dos, que cubre procesos secuenciales y concurrentes. **Ada** es un lenguaje de nivel tres, ya que también cubre el nivel transaccional.

Ahora veamos que en Lenguajes de Descripción de Hardware **Sticks** y **CIF**, que son más formatos de datos que lenguajes, son notaciones de nivel simple, o topológico. El **SPICE NDL** es un lenguaje de nivel cuatro, ya que este expresa información estructural en los niveles de circuit y switch. **KARL** es un lenguaje de

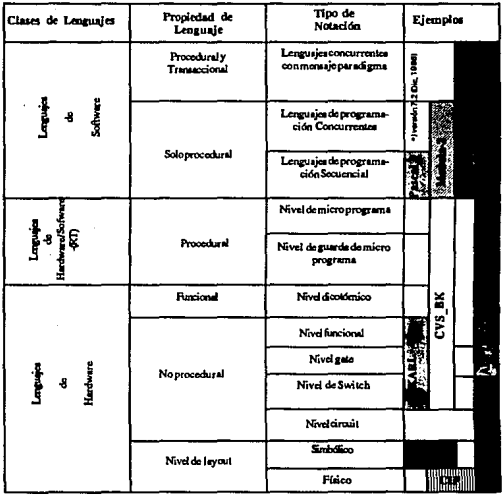


Figura 1.4 Clasificación de Lenguajes de Descripción de Hardware

nivel tres, cubre los niveles de switch, gate y funcional. **CVS_BK** es implementado para el proyecto **ESPIRIT**[1] que es una extensión de **KARL** dentro de los niveles semiprocedurales y procedurales. Ha sido una sorpresa, que **CVS_BK** no sea mucho más complejo que **KARL-3**, aunque cubre muchos más niveles de abstracción; su gramática tiene apriximadamente solo 30% más regrals de producción. Es mucho más simple que Ada.

Esto es una prueba que indica, que los lenguajes multinivel no son lenguajes barrocos necesariamente. **ISP** es un procesador típico orientado a lenguajes de hardware dirigido hacia niveles procedurales. Los controladores de especificaciones **DDL** y **AHPL** son primariamente restringidos a notaciones de estado finito, excluyendo definitivamente los niveles procedurales. A partir de la versión 7.2 de **VHDL** este no solo puede ser considerado como un lenguajes de hardware sino que también es un lenguaje procedural/ transaccional

" El Arte es la
Filosofía que Refleja
el Pensamiento"

Antoni Tàpies
Pintor español
(1923-)

DISEÑO CONCEPTUAL

En el capítulo anterior hablamos de los niveles de diseño y su notación para la descripción de hardware en diferentes grados de abstracción. Ahora, ya que contamos con una visión general en esta área podemos decir que, la implementación de nuestro trabajo se encuentra enmarcado dentro de la definición de Diseño Conceptual en el **Nivel de Transferencia de Registro**, ya que este nos provee de los conceptos para descripciones de flujo de datos particulares, necesarios para describir la estructura y topología de hardware, además de examinar muy de cerca los patrones de alambrado.

Los lenguajes generados para este nivel de diseño son llamados **Lenguajes de Transferencia de Registro**, los cuales se han desarrollado, desde mediados de los 70, como formatos de entrada para Diseño Conceptual de sistemas. Este tipo de lenguajes son capaces de expresar el funcionamiento así como también la conexión de los elementos y su colocación estructural y topológica de atributos del hardware en una forma concisa.

II.1 DISEÑO CONCEPTUAL

- **Características Generales**

El proceso de diseño de harware digital, como ya hemos mencionado, consiste de un sucesivo refinamiento de la especificación funcional de un circuito en su estructura de silicio. Generalmente, este refinamiento es ejecutado a través de diferentes etapas, en las cuales el diseñador genera las descripciones del hardware a diseñar en los diferentes niveles de diseño definidos para circuitos integrados.

Típicamente dichas descripciones son desarrolladas con diferente grado de detalle en cinco de los niveles de diseño. En el **Nivel de Transferencia de Registro** (*RT*) el hardware es especificado en una forma mezclada funcional/estructural, donde los componentes de descripción puramente funcional son variables algorítmicas, mientras que una estructural utiliza elementos de hardware como multiplexores, registros, nodos, etc. Para el **Nivel Gate** la función del hardware es especificada en términos de sistemas de lógica combinacional o secuencial. En tanto que en el **Nivel Switch** se especifica a través de modelos de tecnología independiente de conectores, transistores, resistencias con cargas de pull-up y pull-down. En el **Nivel Circuit** los modelos de tecnología independiente del nivel switch son sustituidos por los correspondientes diodos, transistores, resistencias, etc, y, finalmente en el **Nivel de Layout** se realiza la definición del hardware en estructuras de silicio.

El proceso de diseño conceptual puede ser caracterizado por los siguientes aspectos:

- Esta filosofía inicia la descripción de hardware digital en el Nivel de Transferencia de Registro (*RT*), auxiliado por lenguajes **RT**, ya que una descripción en un nivel lógico o circuit podría resultar mucho más extensa y compleja dada la estructura de los elementos.

- Soporta el diseño jerárquico de circuitos integrados, conduciéndonos a la concepción de celdas reservadas, con capacidad de arreglos, y su colocación dentro del circuito integrado.

- Guía al diseñador no solo a conceptualizar el diseño por desarrollos algorítmicos adecuados para ser implementados en estructuras de silicio, sino también a pensar acerca de la capacidad de prueba del diseño.

- En cada nivel de abstracción el diseño es simulado a fin de poder verificar si este satisface los requierimientos deseados.

- Los pasos de refinamiento son apoyados computacionalmente y los resultados son automáticamente verificados.

- Automáticamente se generan patrones de prueba válidos para el nivel **RT**, como para otros niveles y equipo de prueba de circuitos.

Durante los últimos años la complejidad tecnológica de un circuito tiende a crecer mucho más rápido que la potencia de las herramientas para diseño automático de circuitos (*Compiladores de circuitos integrados*) las correcciones para su construcción es hoy sólo posible para algunos tipos especiales de arquitectura. Esto implica la necesidad de diseñar herramientas de verificación potentes y sofisticadas para validar las correcciones de cada uno de los pasos del diseño. El esfuerzo de la estandarización hace valer el diseño electrónico intercambiando formatos, los cuales son de algún tipo de Lenguaje de Descripción de Hardware, y debe aceptar una amplia gama de herramientas de otros vendedores, dado que el diseñador ya no es restringido a emplear las herramientas de un solo vendedor.

- **Proceso del Diseño Conceptual**

El proceso de diseño conceptual es dividido en dos partes principales, el diseño **"top down"** y la extracción **"bottom up"**. El objetivo principal de la primera parte consiste en la transformación de una descripción de hardware en un diseño de alto nivel (*RT nivel*) en un diseño de bajo nivel con más y más detalles de implementación hasta que el circuito es descrito en estructuras de silicio. La segunda parte es considerada como algún tipo de verificación de hardware. De este modo el diseñador recorre los niveles de diseño en un orden reversivo. En la última parte se aplican sistemas de extracción de hardware para librar de un nivel más bajo de descripción a la especificación correspondiente en el nivel más alto siguiente hasta que nuevamente se alcanza un nivel de descripción **RT**. En cada etapa de diseño se simula la descripción generada por el extractor y comparan los resultados de la simulación con la correspondiente durante la fase de diseño. En los siguiente párrafos el lector encontrará una descripción de los dos subprocesos.

- **Proceso de diseño Top Down**

Esta fase del diseño conceptual toma lugar durante la especificación en el nivel **RT**. Ya en el nivel **RT** el diseñador especifica el funcionamiento; así como algunos detalles de la implementación del circuito, la topología de la planta del sistema y de los puertos de entrada, salida y bidereccionales de las subceldas. En esta etapa se requiere de un lenguaje de nivel **RT**, el cual debe proveer las características para expresar la información funcional y estructural. Empleando un **HDL** de este tipo, el diseño conceptual en esta fase abarca los siguientes pasos:

PASO 1

Divide la función del sistema de hardware completo a ser diseñado en diversas subfunciones y buscan algoritmos apropiados para ser implementado en silicio.

PASO 2

Se genera una descripción funcional de cada subfunción la cual se simulan para validar su funcionamiento con respecto a los requerimientos de diseño.

PASO 3

Seleccionar y especificar una planta flexible para el diseño de las topologías en la interfase de cada subunidad y generar la descripción estructural y topológica correspondiente. Con esto se persigue que la mayor parte de las interconexiones de las celdas puedan ser alambradas por confinamiento (*agrupamiento*) sin alambrado de celdas adicionales.

PASO 4

Diseñar para cada subunidad (*celda*) la correspondiente celda reservada con capacidad de arreglo de modo que la trayectoria de datos completa pueda ser formada por confinamiento de las celdas reservadas, sí es posible.

PASO 5

Considerar nuevamente la diversas celdas reservadas y describir elementos **RT** complejos para elementos de bajo nivel con la misma estructura I/O para mantener la consistencia de el patrón de prueba. *p. ej.*:

- Remplazar la construcción del multiplexor por la expresión combinacional correspondiente, o en nivel switch por modelos de redes de transistores.
- Remplazar todos los tipos de operadores, aritméticos, lógicos y relacionales, del **HDL**

por la correspondiente división de operadores definida por el usuario.

- Describir los registros por medio de concatenación de almacenamiento dinámico con una trayectoria de 1 bit de ancho y más distribución diferencial y control de señales.

PASO 6

Ahora, se realiza la partición de arreglos de las celdas en subarreglos separados mediante una simple celda clave, la cual puede ser descrita en un diseño de nivel bajo (**PASO 3**). Seguir esta estrategia de describir el circuito completo y simularlo, de modo que la celda reservada puede ser válida en un contexto de nivel **RT**, el cual es mucho más eficiente que una simulación de una descripción en un nivel switch puro.

El **PASO 1** y el **PASO 2** son las fases de captura del problema de diseño y validiación de la especificación. Durante la ejecución de los pasos siguientes la fase de diseño conceptual **top down** toma lugar. De tal modo que el diseñador genera una combinación de descripción funcional y topológica del circuito. Durante el **PASO 5** el diseñador describe algunos elementos **RT** como multiplexores, registros, etc. en diferentes niveles de abstracción, ya que a veces puede ser necesario para el diseñador sumergirse por debajo del nivel switch y comprobar si su concepto de diseño puede ser realizable en los niveles más bajos. Al final de la fase de diseño conceptual en el nivel **RT** el diseñador tendrá una combinación válida de la descripción estructural y funcional.

El diseño en los niveles circuit y layout, puede ocurrir en dos diferentes formas. Una primera posibilidad puede ser, que el diseñador haya usado en la especificación de hardware en el nivel **RT** un **HDL**; siendo este el formato de entrada a diseños de automatización de sistemas, como **ISP** y **APL** del sistema **ALERT**[11], sistema desarrollado por **IBM**. En este caso la descripción en el nivel **RT** es traducida automáticamente a un código intermedio que es el formato de entrada a otras etapas en el diseño del sistema generando los niveles correspondientes en los niveles más bajos. El nuevo proceso de diseño puede ser influenciado por el diseñador, dando éste los parámetros de entrada específicos al circuito. Sí, no obstante, el diseñador ha seleccionado un sistema de descripción de hardware en el nivel **RT**, el cual no está integrado para diseñar herramientas en los más bajos niveles, entoncés él debe hacer manualmente la transformación entre los diferentes niveles y generar los formatos de entrada para las herramientas de simulación de hardware en los niveles de diseño más bajos como **DOMOS** y **SPICE2**[11]. La desventaja de este método de diseño es la ocurrencia de errores durante la transformación de las descripciones.

- **Refinamiento en el nivel lógico de diseño conceptual**

Antes de este punto el diseño ha sido estructuralmente refinado dentro del nivel **RT**.

Ya se ha mencionado que durante el diseño lógico dos estrategias son usadas conjuntamente. La

primera de ellas, se dijo, es la **estrategia top down**, en la cual los elementos estructurales de la descripción en el nivel **RT** son sustituidos por ecuaciones lógicas, definiendo la relación entrada/salida de los elementos. Estas ecuaciones pueden ser implementadas en computadora por técnicas de minimización clásicas a través de las tablas de verdad de los elementos en el nivel **RT**. Este refinamiento, es casi independiente de la tecnología con que el circuito será físicamente realizado.

Sin embargo, durante el refinamiento en el nivel lógico, la tecnología fuente tendrá ya alguna influencia en la descripción lógica del diseño. Esto es causado por la segunda estrategia de refinamiento, el **diseño bottom up**. En este enfoque el diseñador ya tiene algún conocimiento de la tecnología y además puede seleccionar la más apropiada. *P. ej.*, si empleamos tecnología **NMOS** o **CMOS**, solo una etapa compuesta de compuertas será utilizada extensivamente. Estas son relaciones lógicas **AND** y **OR** con una inversión a la salida.

De esto observamos que la descripción en el nivel lógico de un diseño está influenciado por la tecnología utilizada, pero el lenguaje en el cual el diseño lógico es descrito puede ser tecnológicamente independiente, ya que las primitivas del lenguaje son operadores Booleanos y funciones.

Usualmente la descripción completa de un diseño no es transferida al nivel lógico en un sólo paso, sino por un conjunto de ellos. En cada uno de estos pasos solo una pequeña parte del diseño es modificada, *p. ej.* una sola celda. Así, a fin de simplificar el proceso de diseño, el **HDL** empleado debe proporcionar primitivas del nivel **RT** y nivel lógico. En este caso el diseñador deberá aprender un lenguaje, y simular una descripción mezclada del nivel **RT** y lógico con un sólo simulador. Los resultados de la simulación pueden ser comparados con los resultados de simulación del nivel **RT** a fin de obtener una primera verificación de cada uno de los pasos de refinamiento.

- **Refinamiento físico de un diseño conceptual**

El siguiente paso en el ciclo de diseño es el desarrollo de una descripción del circuito del diseño. La transición del nivel lógico al circuit es caracterizada por dos aspectos principales:

- Dependencia de la tecnología en el diseño del circuito.
- La diferencia en modelar la lógica y el funcionamiento eléctrico de los elementos de la descripción lógica y circuit.

Estos dos puntos son la razón por lo que la mayoría de los **HDL** para el nivel **RT** no cubren descripciones en nivel circuit. La dependencia de la tecnología debe requerir un gran conjunto de primitivas en el **HDL** y los diferentes modelados requieren de diferentes simuladores. Así, en la mayoria de los casos el diseñador empleará una descripción diferente tanto para el nivel circuit como para el nivel lógico o nivel **RT**. El lenguaje comúnmente más empleado es el formato de entrada **SPICE2**. Las descripciones del nivel circuit generalmente son los formatos de netlist simples que son o dispositivos (***SPICE o DOMOS***), o net orientados

(*EDIF*). Algunos de ellos soportan diseños jerárquicos (*SPICE, EDIF*)

A fin de forzar una estandarización en el área de la descripción de circuitos el **Comité EDIF** ha publicado el **"Electronic Desing Interchange Format" EDIF**. Se espera que este se convierta en un formato estándar para las descripciones de niveles bajos. El formato de intercambio **EDIF** cubre un amplio rango de niveles de representación. Fue desarrollado por un comité de representantes de las más grandes compañías americanas de procesamiento de datos.

El desarrollo de la descripción en el nivel circuit para una descripción lógica de un circuito dada es casi simple, ya que las compuertas lógicas, generalmente, pueden ser sustituidas directamente por circuitos transistores equivalentes. Más difícil es la correcta dimensión de los elementos del circuito, especialmente si el temporizado o el disipador de potencia son dados. Después de que todos los elementos han sido clasificados correctamente, una primera simulación transitoria puede ser ejecutada a fin de verificar el desarrollo del circuito.

Una detallada simulación puede llevarse a cabo después de la generación del layout del diseño, lo cual no es sujeto de este texto.

- **Verificación funcional del diseño**

A fin de verificar la función del diseño existe una serie de pasos que pueden ser realizados. Estos pasos proporcionan la corrección de un paso de refinamiento metodológico, *p. ej.*, proporcionar la transición del diseño lógico al circuit. Este paso de verificación será ejecutado después de cada paso de refinamiento, *p. ej.*, para cerrar el ciclo de diseño entre los niveles de diseño **RT** y layout (Figura 2.1). Esta figura no contiene la fabricación y prueba del circuito integrado. Dado que la fabricación es costosa, todos los errores de diseño deben ser detectados y corregidos antes de la fabricación del prototipo, a fin de evitar un rediseño. Para detectar este tipo de errores en la etapa inicial del diseño se requiere alto desarrollo y herramientas de verificación sofisticadas.

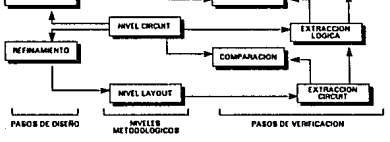


Figura 2.1. Lazos de diseño y verificación

II.2 LENGUAJES DE TRANSFERENCIA DE REGISTRO

- **Características generales**

Los **Lenguajes de Tranferencia de Registro** (*RTL*) incorporan construcciones para describir el funcionamiento de máquinas de estado síncronas en términos de operaciones lógicas y/o aritméticas y estados de transición. La simulación del modelo del sistema es producto de la evaluación de los enlaces de funcionamiento en ciclos sucesivos de reloj. *P.ej.*, la descripción de una instrucción de Adición a Memoria en **DDL** es dada en la Figura 2.2. El sistema consiste de señales globales (*y almacenamiento*), así como de una máquina de estado finito local (*autómata*). Las señales globales son disponibles a cada autómata, los cuales están comprendendidos de control independiente y funciones de transferencia de información (Figura 2.3).

```
<SISTEMA>  SISTEM_NAME:
<REGISTER> INST_REG(16), ACC(16), MAR(10).  /* Define el Registro de Instrucción, el Acumulador y el Registro de Dirección de Memoria */
<TI> CLK1(200E-09).                         /* Define la señal de reloj para que una máquina de estado pueda ser sincronizada */
<MEMORY> M0:1023,16).                       /* define 1024 palabras de memoria */
<AUTOMATA> CPU: CLK1:                       /* La señal CLK1 es la señal de sincronización para el autómata CPU */
<ESTADOS>
        •
        •
        •
MEMADD: ACC ←ACC + M(MAR), →IFETCH
        •
        •
        •
```

Figura 2.2 Descripción de una máquina de estado síncrono en un Lenguaje de Transferencia de Registro

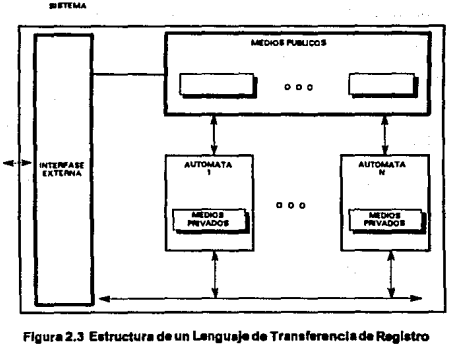


Figura 2.3 Estructura de un Lenguaje de Transferencia de Registro

Las descripciones en un Lenguaje de Transferencia de Registro son útiles en describir y simular el funcionamiento de un sistema en un **nivel arquitectural**, que se opone a un nivel gate (*oprimitivo*). Teniendo desarrollada la descripción arquitectural, las herramientas de síntesis lógica pueden ser ejecutadas para desarrollar una tecnología independiente del nivel de descripción gate, un algoritmo de transformación lógica puede entonces mapear el enlace de tecnología independiente dentro de los registros de una biblioteca. El resultado de la tecnología de descripción dependiente es la base para verificación de temporizado adicional, y ultimar el diseño físico.

Los **RTL** aun son deficientes en definir máquinas de estado asíncronas, y sistemas específicos además de limitantes en la representación de medidas del circuito integrado, y la capacidad para fusionar jerárquicamente la descripción de un circuito integrado particular dentro de un modelo más amplio a nivel sistema, esto es debido a su enfoque síncrono de modelar el comportamiento de sistemas. En resumen, la metodología por medio de la cual una descripción de tecnología dependiente es producida, síntesis y transformación lógica, es revestida con diversas tareas:

1. La minimización de la función lógica para cada autómata es difícil de realizar. Las condiciones que especifican la información que es transferida a cada registro son empleadas para generar la descripción Booleana del enlace de control requerido, minimizar el conjunto colectivo de enlaces de control en el flujo de datos para el total de autómatas es un esfuerzo formidable. Para facilitar esta tarea, el diseñador puede definir señales Booleanas intermedias en la descripción del autómata y entonces describir las condiciones para el flujo de datos en términos de esas variables intermedias.

2. La descripción de un autómata se constiyuye de un conjunto de operaciones de flujo de datos en cada estado que deben ser diseñados para ejecutarse simultáneamente (Figura 2.4). Este **conjunto compatible de operaciones** requiere de la síntesis de los enlaces combinacionales discretos para producir las manipulaciones requeridas. Sin embargo, ¿A qué grado puede una herramienta de síntesis reconocer que la función de un enlace de control puede ser incorporado dentro de una operación similar en otros estados? ¿Cuáles son las consecuencias de desarrollo al compartir semejante network?. Para dirigir el programa de síntesis, el diseñador generalmente define una operación lógica simple de enlace, la cual entonces puede ser referenciada en diferentes estados de el autómata con conjuntos diferentes de señales de entrada.

```
<    STATE    >
          /* CONJUNTO COMPATIBLE DE OPERACIONES */
EXEC1: M[MAR]  INPUT, <— ACC  <—  ACC+1, IN  <—  0
                  —> EXEC2.
          /* TRANSICIONA UN ESTADO INCONDICIONAL */
```

Figura 2.4 Conjunto compatible de operaciones

3. El diseño de la tecnología dependiente resultante en la síntesis y transformación lógica no pueden utilizar efectivamente las macros disponibles en la biblioteca de la tecnología. La tarea de reconocer en donde funciones más altas o macros pueden o deben ser incorporadas es muy difícil; la descripción final del circuito integrado probablemente contendrá una gran concentración de funciones primitivas. El diseñador podrá necesitar dirigir el algoritmo de síntesis para una especificación de macro para algunos subconjuntos del diseño de descripción: **registros, operaciones de enlace o un autómata completo.**

4. La flexibilidad disponible al describir una operación en un simulador de nivel de transferencia de registro puede exceder las capacidades de la herramienta de síntesis a generar las manipulaciones resultantes, tal puede ser el caso de operadores aritméticos, como de adición binaria y sustracción.

5. El número y secuencia de la transición del estado en cada autómata puede ser empleada como tentativa para sintetizar la función del secuenciador de estado apropiado; sin embargo, es problable que el diseñador eliga describir el secuenciador explícitamente, incluyendo la asignación de vectores Booleanos específicos a máquinas de estado.

6. Quiza el aspecto más frustrante de esta metodología de diseño yace no con la tarea de síntesis en sí misma, sino en la extrema dificultad presente al intentar el proceso inverso; *i.e.*, si en el diseño se incorporan cambios dentro de la descripción sintetizada en el nivel gate, ¿A qué grado puede la descripción a nivel arquitectural estar al día para reflejar la función concurrente?, ¿Cómo pueden las dos descripciones ser mantenidas a fin de siempre estar en concurrencia?.

Este último punto es indudablemente el más difícil para soportar. La descripción de los Lenguajes de Transferencia de Registro ofrecen una muy productiva trayectoria para la descripción del nivel sistema y verificación, pero frecuentemente son de mucho menos uso después de la síntesis y simulación a nivel del circuito integrado.

II.3 ESTRUCTURA DE LENGUAJES DE TRANSFERENCIA DE REGISTRO

La estructura del sistema de descripción de lenguajes **RT** es discutido en esta sección análogamente a un lenjuage de programación estructurado. Los programas estructurados y las descripciones lógicas construidas jerárquicamente comparten la necesidad para la declaración de identificadores empleados en la descripción, la **simbolización** (*"typing"*) de variables y nombres de señales, en particular, la implementación de un conjunto de reglas para determinar **el alcance** de un identificador (***determinar la extensión de aplicabilidad cuando el mismo nombre es utilizado en multiples declaraciones***). Reglas de aplicabilidad son requeridas en el diseño de sistemas, dado que no solo un diseñador desarrolla esta tarea. Cada diseñador está en libertad para definir y usar identificadores dentro del alcance de particiones individuales, ambiguedades no son introducidas si el mismo identificador es declarado y empleado en cualquier otra parte. Dos casos aparecen: un nombre puede ser reusado en alcances desempalmados no translapados, o un nombre puede

ser redefinido dentro del alcance de una declaración previa. Como en el caso de los lenguajes de programación estructurada, la definición más interna pertenecerá al alcance de su rango específico de validez.

La estructura básica para dividir la función del sistema en network será denotada como un **"block"**; cuyo exterior puede ser considerado una **caja negra** con pines. Para continuar la analogía con un lenguaje de programación estructurada, un block debe ser la contraparte de un procedimiento (*"procedure"*). Las restricciones nos son colocadas en el número de pin o la complejidad de la función contenida dentro del

alcance de una descripción de block, claro, el sistema de entrada está contenido dentro del alcance de la declaración del nivel top.

El bosquejo de la descripción de un sistema está representado en la Figura 2.5. Antes de la colección de las descripciones de block, una sección de declaración es proporcionada. Cada descripción de block contiene interfase de señales y modelado de información. La descripción de block del nivel top circunda el diseño de sistemas completos, las particiones de esta función que define la jerárquia del diseño están contenidas dentro del alcance de la descripción del nivel top. La anidación de blocks (estos referenciados por un nombre global) están fuera de el alcance de la descripción del nivel top y son incluidas separadamente.

Figura 2.5 Estructura completa de una descripción de sistema jerárquico. El bloque de descripción en el nivel top contiene las particiones del sistema completo, exclusivo de las funciones externas anidadas.

La sección de declaración es empleada para definir identificadores **globales** que serán encontrados por el parser (*analizador de sintaxis en un compilador*) en la descripción subsecuente del block, estos identificadores incluyen nombres de blocks, nombres de variables y constantes. La sección de declarativa presume la forma esquematizada en la Figura 2.6. Típicamente se asume que

1.- Los blancos no pueden estar incluidos en la cadena de un identificador
2.- La coma sirve como separador
3.- El punto y coma sirve como terminación de sentencia

los comentarios pueden ser incluidos en cualquier lugar en una descripción del diseño y debe ser empleado frecuentemente para adicionar claridad y legibilidad. Un medio para incluir comentarios es identificar un solo caracter que empieze el comentario; éste entonces terminará al final de línea. Otra forma es usar un par de delimitadores complementarios, uno que inicie y otro de terminación de comentario. De esta forma, una sección de comentarios puede ser localizada en cualquier lugar del texto; potencialmente puede extenderse en varias líneas. Una nota particular en esta sección inicial es el grupo **UNITS**, el cual proporciona posibles tipos de variables adicionales. La asignación de un valor a una variable definida para contener una unidad de medida requiere normalización hacia el valor por default. Para el cálculo de valores significativos de la variable, cuando compile la descripción del sistema, un conjunto de unidades por default debe ser definida en una manera consistentente.

Otro atributo único de este formato de la descripción es la designación de algunas variables como **parámetros.** La intensión de esta definición es permitir a cada block tener un conjunto asociado de parámetros que pueden ser asignados únicamente a ese block. Los parámetros no siguen las reglas de alcance normalmente aplicadas a nombres de variables, por el contrario, son identificadores globales que pueden ser definidos y asignados a nodos individuales en la jerarquía del diseño. La mayoría de los parámetros de block son de tipo **Booleano** y pueden ser empleados para describir las propiedades del **árbol** de una descripción estructurada jerárquicamente. El parámetro puede definir al block como un nodo o una hoja de la jerarquía del diseño para diseñar una herramienta de aplicación particular. *P. ej.*, si un nodo de la descripción del diseño es definido incompletamente cuando un modelo simulador es requerido, el parámetro del block "**simulatable := FALSE;**" puede ser asignado. Durante la construcción del modelo, la expansión de este block (*y toda partición*) puede ser inhabilitada; las salidas del block entonces darán **valores indefinidos** durante la simulación.

La sección de **TYPE** es similar a la de un lenguaje de programación estructurada en ese subrango, los arreglos, registros, y una lista enumerada pueden ser descritos y dar definiciones de variables.

```
SYSTEM:
      DECLARATION_BLOCK block1, block2, ..., blockn;
/* Define todos los nombres de bloque de alcance global – i.e., nivel top y nombre de bloques anidados*/
      UNITS
      time:  nsec, /* default*/
            psec = 1.03E-3 nsec,
            usec = 1000 nsec,
            msec = 1.6E6 nsec,
            sec = 1000 msec;
      voltaje:    ...;
      corriente:  ...;
      potencia:   ...;
      frecuencia: ...;
      capacitancia: ...;
      energía;  ...;
      CONSTANTS
      identificador = integer/Boolean/string;
      TYPE
      sim_model = (LOGIC, TABLE, PARTS,BEHAVIORAL)
      designer = RECORD
            nombre : packed array[1..20] of char;
            dato:  ...;
            revision : integer;
            status:  ...;
      end;
      duty_cycle = 0..100; /*subrrango entero denotando un porcentaje*/
      rango = (nom, min, max);
      propagación = RECORD
            rising: array[rango] of time;
            falling: array[rango] of time;
      end;
      tecnología = (NMOS_2UM, CMOS_2UM, CMOS_1UM, ECL);
      VAR
      sisclk: frecuencia;
      supply: voltage;
      parámetro simulatable: Boolean;
      parámetro placed: Boolean;
      parámetro delay: propagación;
      parámetro default_block_delays: Boolean;
/* retardos de bloque por default son suficientes para networks combinacionales*/
      parámetro combinacional : Boolean;
            /*Declaración de funciones*/
      function f1(var1:type1, var2: type2, ...): result _type;
begin
var
end;
```

Figura 2.6 Sección de declaración de una descripción del sistema

Puede ser evidente que una fracción significativa de las construcciones de futuros lenguajes no pertenece a la interconectividad de networks lógicos en todo. En realidad, la mayoría de estas construcciones adicionales o características no son requeridas. Sin embargo, el énfasis actual en el diseño de manejadores de datos puede contener tanto del usuario que nuevos tipos de información son requeridos o producidos por la herramienta de aplicación del sistema de diseño, los medios para la descripción del diseño deben ser capaces de expander y absorber estos datos. La complejidad de los compiladores para seleccionar, calcular y manipular los datos del diseño de tan diversas descripciones es considerable. Esto plantea las siguientes preguntas: ¿de qué debe consistir el diseño del sistema de base datos? Exactamente ¿cómo debe ser el lenguaje de descripción del diseño?.

La estructura de una descripción de bloque es desglosada en la Figura 2.7. El encabezado del block define los pines que son usados para conectar al block a un ambiente exterior (Figura 2.8.). Un **bus** es una colección de pines relacionados; el cual es especificado por un identificador seguido de un rango de enteros; los pines individuales en el bus son denotados por concatenaciones de enteros a el nombre del bus. El encabezado de block también incluye una indicación de la señal direccionable a cada pin o bus, muy similar a la asignación de **tipo** de cada señal. El conjunto de tipos de señales direccionables pueden incluir: **Entrada, Salida, Bidereccional, alta impedancia, y drain/collector abierto** (*Dottable*). La compilación de la descripción del diseño puede por lo tanto desarrollar una verificación inicial de validación de interconexiones (*p. ej.* dos salidas no manejan el mismo net). El simulador puede usar estas designaciones para determinar valores de señales cuando resuelva el caso de fuentes múltiples en un net, el cual aparece cuando circuitos bidereccionales, alta impedancia y dottable son empleados.

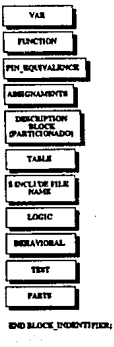


Figura 2.7 Estructura de la descripción de un bloque

Después del encabezado de block sigue el volumen de descripción del block, el cual asemeja la estructura de un procedimiento programado, pudiendo así contener diferentes secciones únicas.

```
BUS DEFINITION
BLOCK 256K DRAM ( A(8:0):I, DIN:I, CAS-:I, RAS-:I, R_W:I, DOUT:
```

Figura 2.8 Encabezado de bloque

La sección de **PIN_EQUIVALENCE** define relaciones de equivalencia lógica entre pines, esta información puede ser de utilidad durante el diseño físico, si una interconexión difícil para el alambrado puede ser simplificada a través de un "swapping" de pines[1]. La Figura 2.9 ilustra una posible sintaxis para especificar la equivalencia de los pines, las expresiones pueden ser anidadas para describir mejor las relaciones de enlace. Se emplean dos delimitadores: paréntesis cuadrados, que significa que la lista de identificadores encerrada es un conjunto desordenado, mientras que los paréntesis redondos indican que los nombres de los pines encerrados constituyen un vector o conjunto ordenado.

```
BLOCK 2-2ADI (A1:I, A2:I, B1:I, B2:I, OUT:0) ;
      PIN_EQUIVALENCE [[A1, A2] , [B1, B2]] ;
BLOCK 4-BITREG ( D (0:3) : I, CLK:I , Q(0:3) : O ) ;
      PIN_EQUIVALENCE [ D0,Q0) , (D1,Q1) , (D2,Q2) , (D3,Q3)]
```

Figura 2.9 Sección PIN_EQUIVALENCE de la descripción del block.

La sección de **ASSIGNMENTS** es la contraparte para la sección ejecutable de un procedimiento; esta sección contiene los enunciados que asigna valores a las variables y parámetros del block. Como aludimos al principio, estos parámetros de block pueden ser definidos para contener una gran cantidad de información, desde la representación de datos hasta condiciones de diseño. Como el diseño evoluciona e información adicional del bloque es disponible, esta sección estará cambiando; la descripción del block que ha cambiado puede ser recompilada para reflejar esos datos.

[1] Si pines lógicamente equivalentes son verdaderamente intercambiados (swapped) por un programa de alambrado, esa información debe ser escrita posterior a un bloque de parámetros en la descripción de diseño de la base de datos

La función del bloque para propósitos de simulación y modelado de prueba puede ser preparada en un número de representaciones diferentes, usualmente, más de una será codificada para el diseño. Algunas de estas posibles representaciones son descritas en la Figura 2.7. La representación funcional más simple, adecuada para network combinacionales, presume proveer una **TABLA de verdad**. Esta técnica es apropiada, particularmente, para describir contenidos de **ROM** y **PLA** funcionalmente. El valor del parámetro de "**delay**" debe ser asignado a todas las transiciones de salida.

Una sección **LOGICA** debe describir la función del network en términos de ecuaciones lógicas y primitivas de simulación. Un ejemplo es dado en la Figura 2.10. Una ecuación lógica asigna el resultado de evaluación de una expresión Booleana a un nombre de señal. Una primitiva de simulación está instantaneamente asignando un identificador a la primitiva y nombres de señales para las primitivas de los pines. Un netlist como registro es mostrado; un formato de asignación explícita es requerido para primitivas con entradas no simétricas. Cada pin de salida del bloque debe ser referenciado tanto por una ecuación como por una salida de primitiva de simulación. Los delays internos pueden ser asignados a ecuaciones y primitivas dentro de la sección de modelo **LOGICO**, estos pueden ser empleados en lugar del parámetro de delay del bloque por default para proporcionar una única trayectoria de delays para las salidas del bloque.

Una lista es proporcionada de nombres de señales internas presentes en la definición lógica pero que nos son pines del block.

```
LOGIC
    INTERNALS     NOT1C, NOTA, NOTB SEL1, SEL2 ;
    G1:  (NOTA) = INV (A) [3  7] ;
    G2:  (NOTB) = INV (B) [3  7] ;
    G3:  (NOT1C) = INV (C) [8  4] ;
    G4:  (SEL1) = NOR (1G ,  NOT1C] ;
    G5:  (SEL2) = NOR (2C ,  2G);
    1Y0 = NOT( NOTB AND NOTA AND SEL1 ) [10  18] ;
    1Y1 = NOT( NOTB AND A  AND SEL1 ) [10  18] ;
    1Y2 = NOT( B   AND NOTA AND SEL1 ) [10  18] ;
    1Y3 = NOT( B   AND  A    AND SEL1 ) [10  18] ;
    2Y0 = NOT( NOTB AND NOTA AND SEL2 ) [10  18] ;
    2Y1 = NOT( NOTB AND A  AND SEL2 ) [10  18] ;
    2Y2 = NOT( B   AND NOTA AND SEL2 ) [10  18] ;
    2Y3 = NOT( B   AND  A    AND SEL2 ) [10  18] ;
```

Figura 2.10 Modelo de simulación de una Network.

La sección de **TEST** describe el modelo de network como la interconexión de primitivas de prueba, muy parecida a la menera de la sección **LOGICA** (*solo sin la opción de ecuaciones Booleanas*).

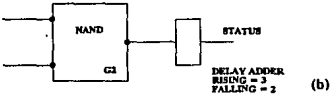
La sección **BEHAVIORAL** es a diferencia de otra; un modelo conductual de un network es un procedimiento escrito para simulación. La intensión de codificar conductas para enlaces complejos es proporcionar una simulación eficiente y potente, en el orden de descripciones de Lenguajes de Transferencia de Registro. Un amplio completo rango de construcciones programadas son disponibles (*p. ej. , variables locales, expresiones condicionales y la capacidad de mensajes de salida a un archivo de error*). Unico para un enfoque de modelo conductual sobre otras técnicas son las siguientes capacidades:

1.- Lectura y almacenamiento local al tiempo de simulación (*una variable global*) cuando el procedimiento es llamado.
2.- Representación de valores de señal, bus y arreglos en una variedad de formas (*binario, octal, decimal, hexadecimal*) a demás de ejecutar operaciones lógicas y/o aritméticas.
3.- Catalogar un cambio en el valor de una señal, bus o arreglo en un tiempo de simulación futura desde dentro del procedimiento (*en lugar de usar un delay de block*).

El procedimiento conductual es llamado por el simulador siempre que una transición en un pin de entrada a el block ha ocurrido.

La sección de **PARTS** es el medio principal de describir como la función contenida dentro de un nodo de la jerarquía del diseño ha sido particionado. Esta sección lista todos los subblocks y la naturaleza de sus interconexiones. Un ejemplo de una sección de **PARTS** es ilustrada en la Figura 2.11a. La sección consiste de una lista de declaración de **NETS**, seguida de una o más declaraciones de partes. La subsección **NETS** es simplemente una lista de identificadores que definen los nombres del net empleados dentro de la descripción de block, esto es, los nombres para las interconexiones en este nodo de la jerarquia del diseño jerárquico que no son pines del block. (*Si nets de señales adicionales no están presentes en la partición, además de los pines del block, la sección* **NETS** *no es requerida*). Siguiendo directamente cada identificador de net, delays adicionales pueden ser especificados; estos valores específicos de net deben ser adicionados a los parámetros de delays asociados con el o los blocks manejadores de ese net Figura 2.11b. Estos datos son destinados a alojar delays de net más precisos después de que el diseño físico del circuito integrado ha sido completado por adición de bloques de reatrdo para blocks de delay adicionales para salidas del circuito individual. Cada declaración de subblock empieza con un "**short name**", seguido del nombre de block de un subblock en la partición. Una lista de nombres de señales sigue al nombre del block; estos nets representan las conexiones a los pines del subblock. Un formato posicional es mostrado, el nombre del pin y net son listados para corresponder con la lista de pines en la definición del subblock. Un formato de asignación específico del pin podría igualmente ser empleado. Los identificadores reservados 0, 1, y U pueden ser empleados en esta lista de conexiones para indicar un pin de entrada ligado a un valor fijo 0 o 1, o un pin de salida a ser dejado sin conexión. Siguiendo la lista de conexión, el campo opcional **BOOK=identificador**;

puede ser incluido. Esto es una indicación de que el nombre del block en el principio de este registro no es usado para modelado, ya que algo está siendo suplantado con la descripción de la tecnología específica del circuito o macro diseño. Esto es el significado para seleccionar una descripción particular desde el manual de tecnología del sistema de diseño, ultimado para ser colocado y alambrado durante el diseño físico del circuito integrado Figura 2.11c.

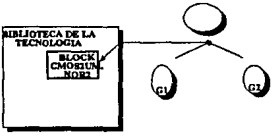


(b)

```
PARTS
NETS STATUS [ 3   2] ;
```

```
G1: NOR (OUT, CMD, STATUS) BOOK = CMOS2UM, NOR2, (X, Y) ;
G2: NAND (SATUS, S0, S1) ;
```

(a)



(c)

Figura 2.11 Sección PARTS.

"Lo que Importa Verdaderamente en la Vida
no son los Objetivos que nos Marcamos,
sino los Caminos que Seguimos para Lograrlo"

Peter Bamm
Escritor Alemán
[1897-1975]

COMPILADOR PARA CIRCUITOS INTEGRADOS

Hasta este momento hemos estudiado el ambiente a través del cual el proceso de diseño de hardware es definido, pero no hemos mencionado la forma en como esta descripción será traducida, de modo que a partir de esta podamos obtener la información necesaria para después realizar una simulación de este diseño y posteriormente su simulación. En este capítulo introduciremos el concepto de **Compilador** y partiremos de este para definir el proceso de traducción de la descripción del hardware; a fin de poder implementar el Compilador para Sistemas Digitales que cumplira con el objetivo de este trabajo.

En los primeros años de la década de los 50 el término compilador fue concebido por **Grace Murray Hooper**[6]. **La traducción** se vió entonces como **compilación** de una secuencia de subprogramas seleccionados desde una libreria, llamándose le **programación automática**, este hecho produjó un esceptisimo universal en relación a su éxito. Hoy en día la traducción automática de lenguajes de programación es un hecho consumado, así en la actualidad los traductores de lenguajes de programación son llamados **compiladores**.

En sus primeros días el concepto de tradución se estructuraba, desarrollaba e implementaba a través de diferentes componentes y técnicas a la medida de cada necesidad. Este camino resulto ser, como es de esperarse, demasiado complejo y costoso. Hoy en día, el proceso de compilación es bien entendido e implementar compiladores es rutina. Sin embargo, desarrollar un compilador inteligente, eficiente y confiable sigue siendo una tarea compleja.

Entre los primeros desarrollos de compiladores, como hoy conocemos, se encuentran los de **FORTRAN** que aparecieron a fines de la década de los 50. Ellos presentaron al usuario con un problema, el **programa fuente**, desarrollando algo un poco ambisioso, para su momento, la optimización del código de máquina ya que se deseaba competir con el entonces dominante lenguaje ensamblador. Los sistemas Fortran comprobaron la viabilidad de traducir (*compilar*) lenguajes de alto nivel, abrindo así el camino para el florecimiento de nuevos lenguajes de programación y sus compiladores tanto en áreas comerciales, científicas y experimentales.

III.1 TRADUCTORES

La entrada de un traductor es un programa fuente que es convertido en un **programa objeto** (*lenguaje de máquina*). El primero escrito en un **lenguaje fuente** y el objeto en un **lenguaje objeto**.

Si el lenguaje fuente traducido es un lenguaje ensamblador y el programa objeto es lenguaje de máquina, el traductor es llamado **ensamblador**. Un lenguaje ensamblador es muy parecido a un lenguaje de máquina. En un lenguaje ensamblador básico la mayoría de sus instrucciones son representaciones simbólicas de instrucciones de lenguaje de máquina.

Un traductor que transforma un lenguaje de alto nivel como **FORTRAN, PASCAL o COBOL** en un lenguaje de máquina particular o ensamblador es llamado **compilador** (Figura 3.1). El tiempo en el cual la conversión de el programa fuente a un objeto ocurre es llamado **tiempo de compilación**. El programa objeto es ejecutado en un **tiempo de corrida**.

Otro tipo de traductores son los llamados **intérpretes**, que procesan simultáneamente el código fuente y sus datos. Esto es, la interpretación de la forma interna fuente ocurre en un tiempo de corrida y no se genera

un programa objeto. La Figura 3.2 nos muestra este proceso. Algunos interpretes analizan una a una las sentencias fuente cada vez que son ejecutadas, como esto consume mucho tiempo es usado rara vez. Un enfoque más eficiente involucra técnicas de compilación dirigidas a la traducción del programa fuente a una forma intermedia que será procesada por el interprete del programa. De entre los lenguajes más populares implementados bajo un ambiente interpretativo se encuentra **BASIC, APL, LISP y SMALLTALK-80.**

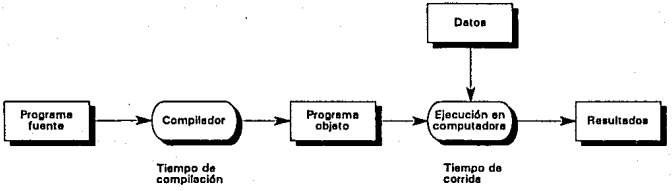


Figura 3.1 Proceso de compilación

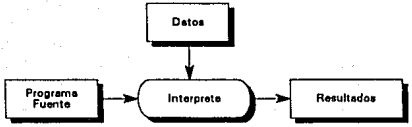


Figura 3.2 Proceso interpretativo

III.2 RUTINAS SEMANTICAS

El corazón de un compilador está situado en sus **rutinas semánticas**, ya que estas definen los detalles de cómo cada construcción es verificada y traducida. Este aspecto de compilación es formalizado vía **atributos gramaticales** (*generalmente gramática de contexto libre*) y **semánticos** como *tipo, valor.*

Las rutinas semánticas suplen el significado (*semántico*) del programa, basándose en la estructura sintáctica del lenguaje, a través de dos funciones. La primera, es la verificación **semántica estática** de cada construcción, es decir, comprueban que la construcción sea correcta y significativa (*que las variables involucradas estén definidas, los tipos sean correctos, etc*). Si la construcción es semánticamente correcta, las rutinas semánticas también hacen efectiva la traducción.

Las rutinas semánticas en algunos casos pueden generar alguna representación intermedia del programa o directamente generar código objeto. Si se genera dicha representación, esta servirá como entrada a un **generador de código**, componente que en realidad produce el programa en el lenguaje de máquina deseado. La representación intermedia puede ser procesada, opcionalmente, por un **optimizador**, de modo que un programa más eficiente en lenguaje de máquina se genera.

Es recomendable mantener los componentes de verificación y traducción en una rutina semántica distinta. La verificación de semántica se realiza primero y es gobernada únicamente por las reglas semánticas de un lenguaje fuente. La generación de una representación intermedia está influenciada por la semántica de el lenguaje fuente (*ya que la representación intermedia de código generada debe implementar correctamente las construcciones de lenguaje*) y la máquina destino (*ya que la elección del código de representación intermedia puede reflejar las capacidades esperadas de la máquina destino*).

III.3 ESTRUCTURA DE UN COMPILADOR

Al pensar en un compilador vemos una caja negra que traduce lenguajes de programación de alto nivel en instrucciones de lenguajes de máquina (Figura 3.3), pero ¿qué acciones son desarrolladas dentro de esa caja?, ¿cómo se realiza la traducción del lenguaje de alto nivel hacia el lenguaje de máquina?, la situación no consiste en solo elegir uno u otro lenguaje fuente y buscarle un equivalente en un lenguaje destino, esta relación es un poco más complicada, ya que mientras la sentencia procesada en el lenguaje fuente es reconocida y aceptada fácilmente por una gramática, existe una variedad de alternativas para describir la salida del compilador con tan solo nombrar un tipo particular de computadora cuya salida es determinada por su tecnología y área de aplicación.

Figura 3.3. Compilador visto por un usuario

La tarea de construir un compilador para un programa fuente en particular es compleja. La naturaleza de la complejidad de este proceso depende en gran medida del lenguaje fuente, sin embargo, esta se puede salvar frecuentemente si el diseñador de lenguajes de programación toma en cuenta varios factores de diseño

tales como el tipo de gramática, el número de palabras reservadas, el área de aplicación, por solo nombrar algunas. Después de tratar con lenguajes fuente de alto nivel como **ALGOL** y **PASCAL** un modelo básico de un compilador similar al mostrado en la Figura 3.4 puede ser formulado. Aunque este puede variar para la compilación de diferentes lenguajes de alto nivel, este es representativo del proceso.

Un compilador debe ejecutar dos tareas principales: el **análisis** de un programa fuente y la **síntesis** de su correspondiente programa objeto, estas dos tareas se desarrollan y completan a través de distintas fases (Figura 3.4) para transformar el programa fuente en una representación de lenguaje de máquina, que cuando sea ejecutada desarrolle correctamente las acciones descritas por el fuente. La tarea de análisis interviene en la descomposición del programa fuente en sus partes básicas creando una representación intermedia de este. Usando esta representación, la tarea de síntesis construye los módulos equivalentes de programa objeto para la representación intermedia generada. El funcionamiento de estas tareas se realiza más fácilmente por medio de la construcción y mantenimiento de diversas tablas.

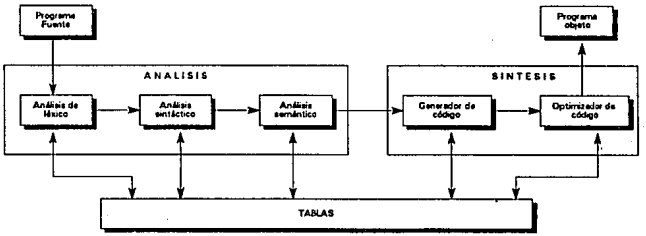


Figura 3.4. Fases de un compilador

Como sabemos un programa fuente es una cadena de símbolos como letras, dígitos, o ciertos símbolos especiales como +, -, y (,), además de construcciones de lenguaje elementales como nombres de variables, etiquetas, constantes, palabras reservadas, y operadores. Por consiguiente es deseable que el compilador identifique los diferentes tipos como clases. Estas construcciones de lenguaje están dadas en su misma definición.

El programa fuente (Figura 3.4.) es la entrada a un **analizador de léxico** o **"scanner"**, cuyo propósito es separar el texto de entrada en piezas o **"tokens"** como constantes, nombres de variables, palabras reservadas (***DO, IF, THEN, etc***), y operadores. En escencia, el analizador de léxico representa el nivel bajo del análisis de sintaxis. Por razones de eficiencia, cada clase de token es dada por una representación interna numérica única. Por ejemplo, un nombre de variable puede estar dado por un número de representación de 1, una constante un valor de 2, una etiqueta el número 3, el operador de adición (+) un valor de 4, etc. **P. ej.,** la sentencia en **PL/I**

TEST: IF A > B THEN X = Y;

puede ser traducida por el analizador de léxico en la siguiente secuencia de tokens y los números de representación asociados:

TEST	**3**
:	**26**
IF	**20**
A	**1**
>	**15**
B	**1**
THEN	**21**
X	**1**
=	**10**
Y	**1**
;	**27**

Notemos que al explorar y generar ar el número de representación para cada token los espacios (*o blancos*) en la sentencia han sido ignorados ya que no representan partes ejecutables.

El scanner coloca las constantes, etiquetas, y nombres de variable en una **tabla de símbolos**, que contiene nombre, tipo (*REAL, INTEGER, o BOOLEAN*), dirección en el programa objeto, valor, y línea en la cual está declarada.

El analizador de léxico suministra estos tokens al **analizador de sintaxis**, los cuales están formados por dos partes. La primera es la dirección o localización del token en la tabla de símbolos. La segunda es el número único de representación del token. Esto ofrece una ventaja distinta para el analizador de sintaxis, todos los tokens son representados por una información de longitud fija: una dirección (*o apuntador*) y un entero.

El análisis de sintaxis es mucho más complejo que el analizador de léxico. Su función es tomar al programa fuente (*en la forma de tokens*) desde el analizador de léxico y determinar la manera en la cuál este es descompuesto en partes constitutivas. Es decir, determina la estructura sintactica completa del programa fuente. Este proceso es análogo a determinar la estructura de una oración en un lenguaje como el Español o el Inglés. En semejante instancia, estamos interesados en identificar ciertas clases como "**sujeto**", "**predicado**", "**verbo**", "**sustantivo**" y "**adjetivo**".

En esta fase nos concierne el agrupamiento de tokens en grandes clases sintácticas como **expresiones**, **sentencias y procedimientos**. La salida del analizador de sintaxis es una estructura jerárquica llamada **árbol sintáctico**; en el cual las operaciones implicadas por el progrma fuente están determinadas y registradas, cada nodo representa una operación y los nodos derivados de este representan los argumentos, los niveles son los tokens y cada nodo no hoja representa un tipo de clase sintáctica. **P. ej.**, un análisis de la sentencia fuente

(A + B)*(C + D)

puede producir las clases sintácticas **<factor>**, **<término>**, **<expresión>** como se puede apreciar en el árbol sintáctico dado en la Figura 3.5.

El análisis de sintaxis se basa en una gramática para determinar la estructura de un lenguaje fuente. El proceso de reconocimiento es llamado **"parsing"**, y consecuentemente a este análisis se le refiere como **"parser"**.

El árbol sintáctico producido por el analizador de sintaxis es utilizado por el **analizador semántico**, cuya función es determinar el **significado o semántica** del programa fuente. Aunque conceptualmente es deseable separar la sintaxis de un programa fuente desde su semántica, los analizadores de sintaxis y semántica trabajan en operación cerrada. El análisis semámtico es un proceso diferente y único en un compilador. Para una expresión como **(A + B)*(C + D)**, el analizador semántico debe determinar que acciones son especificadas por los operadores aritméticos de adición y multiplicación. Cuando el parser reconoce un operador como '+' o '*', invoca una *rutina* o *regla semántica* que especifica la acción ha ser ejecutada. Esta rutina puede verificar que los operandos a ser sumados han sido declarados, que son del mismo tipo (*sino, la rutina debe, probablemente, hacer de ellos el mismo*) y que tengan valores. El analizador semántico, frecuentemente, interactúa con las diferentes tablas del compilador en la ejecución de su tarea.

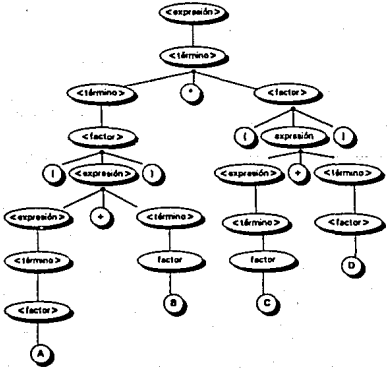


Figura 3.5 Arbol sintáctico para la expresión (A+B)*(D+D)

La acción de el analizador semántico puede involucrar la generación de una forma intermedia del código fuente. Para la expresión **(A + B)*(C + D)**, el código fuente intermedio pudo ser el siguiente conjunto de cuádruplas:

(+, A, B, T_1)
(+, C, D, T_2)
(*, T_1, T_2, T_3)

donde **(+, A, B, T_1)** está interpretado por el significado "suma **A** y **B** y coloca el resultado en **T_1** temporalmente", **(+, C, D, T_2)** significa "suma **C** y **D** y coloca el resultado temporalmete en **T_2**", y la interpretación del significado para **(*, T_1, T_2, T_3)** es "multiplica **T_1** y **T_2** y coloca el resultado en **T_3**". La forma exacta de el lenguaje fuente intermedio empleado depende, en gran medida, de como este se procesó durante la tarea de análisis. Una expresión infija puede ser convertida a una forma intermedia llamada notación polaca. Otro tipo de forma intermedia de lenguaje fuente puede ser un árbol de sintaxis el cual representa el ánalisis de la gramática del programa fuente.

La salida del analizador semántico es transferida al **generador de código**. En este punto la representación intermedia del programa en lenguaje fuente es traducido a cualquier lenguaje ensamblador o lenguaje de máquina. Como un ejemplo, la traducción de las tres cuádruplas para la expresión anterior puede producir la siguiente secuencia de direcciones específicas, acumulador específico e instrucciones de lenguaje ensamblador:

```
LDA A  Carga el contenido de A en el acumulador
ADD B  Suma el contenido de B a el acumulador
STO T1 Almacena el contenido de acumulador en el almacenamiento temporal T1
LDA C  Carga el contenido de C en el acumulador
ADD D  Adiciona el contenido de D a el acumulador
STO T2 Almacena el contenido del acumulador en el almacenamiento temporal T2
LDA T1 Carga el contenido de T1 en el acumulador
MUL T2 Multiplica el contenido de del acumulador por T2
STO T3 Almacena el contenido del acumulador en el almacenamiento temporal T3
```

La salida del generador de código es pasada a un **optimizador de código**. Esta fase puede ser muy compleja y lenta, ya que involucra numerosas subfases, algunas de las cuales puede ser necesario aplicar más de una vez. Muchos compiladores permiten que la optimizacion esté desactivada para adelantar la traducción, e incluso otros no tienen optimizador. En este último caso las rutinas semánticas generan las llamadas al generador de código para producir el código objeto. Su propósito es obtener un programa objeto más eficiente. Ciertas optimizaciones que son posible en un nivel local incluyen la evaluación de expresiones constantes, el empleo de ciertas propiedades de operadores como la asociatividad, conmutatividad y distributividad, además de la detección de subexpresiones comunes. Por la conmutatividad del operador de multiplicación, el código ensamblador anterior puede ser reducido de la manera siguiente:

```
LDA A
ADD B
STO T1
LDA C
```

```
ADD D
MULT1
STO T2
```

Note que la expresión evaluada por este código es **(C + D)*(A + B).**

Se pueden desarrollar optimizaciones más generales que determinen la evaluación única de ocurrencias múltiples de subexpresiones idénticas además de eliminar las sentencias que son invariantes dentro de un ciclo y colocarlas fuera de este. Estos tipos de optimizaciones son máquinas independientes. Existen, sin embargo, ciertas optimizaciones de máquinas dependientes las cuales pueden tambien ser ejecutadas. Un buen optimizador de código puede producir igual o mejor código que un experimentado programador de lenguaje ensamblador.

Dentro de la explicación del modelo de un compilador se ha mencionado el empleo de tablas para el mejor funcionamiento de este, estas tablas son conocidas como de **símbolos y manejadora de errores**, estas dos actividades interactúan con todas las fases de l modelo como se puede apreciar en la descripción de estas.

- **Tabla de símbolos**

Su función escencial es el registrar los identificadores en el programa fuente y colectar información acerca de sus diferentes atributos. Estos atributos pueden proporcionar información de asignación de almacenamiento para un identificador, su tipo, alcance (*donde este es válido en el programa*), y en el caso de nombres de procedimientos, cosas como el número y tipo de sus argumentos, el metodo de transferencia para cada argumento (*por valor o referencia*), el tipo regresado, entre otras.

En general esta es una estructura de datos que contiene registros para cada identificador, con campos para los atributos. La estructura de datos nos permite encontrar el registro para cada identificador y almacenar y/o reconstruir datos desde ese registro rápidamente.

Cuando un identificador es detectado en el programa fuente por el analizador de léxico, el identificador entra a la tabla de símbolos. Sin embargo, los atributos de un identificador no pueden ser determinados normalmente durante el análisis de léxico. En cada tiempo un identificador es utilizado, la tabla de símbolos proporciona acceso a la información colectada acerca de este cuando su declaración fue procesada. Asi, esta tabla es empleada por cualesquiera de las fases de el compilador.

- **Reporte y detección de errores**

En cada fase se pueden encontrar errores, de modo que después de detectar un error, una fase debe de alguna forma tratar con ese error, así esa compilación puede proceder, permitiendo detectar errores adicionales en el programa fuente. Un compilador que se detiene cuando encuentra el primer error no es tan útil como este podría parecer.

Las fases de análisis sintáctico y semántico usualmente manejan una gran fracción de los errores detectados por el compilador. La fase de léxico puede detectar errores donde los caracteres restantes en la entrada no forman ningún token perteneciente al lenguaje. Los errores cuando el flujo de tokens viola la estructura de las reglas sintácticas son determinados por la fase de análisis de sintaxis. Durante el análisis semántico, el compilador intenta detectar construcciones que tengan la correcta estructura sintantica pero no asi el significado de la operación involucrada.

El modelo de un compilador dado en la Figura 3.4. hace distinciones entre sus fase. En ciertos compiladores, sin embargo, algunas de estas fases están combinadas en un módulo llamado '**pasada**'. Una pasada leé el programa fuente o la salida de una pasada previa y desarrolla las actividades especificadas por su fase, registrando los resultados en un archivo intermedio, el cual puede ser leído por una pasada subsecuente. La estructura de un lenguaje fuente tiene un fuerte efecto respecto al número de pasadas. Ciertos lenguajes requieren al menos dos pasadas para generar código. El ambiente en el cual el compilador debe operar es otro factor que interviene en el número de pasadas. Un compilador de pasadas múltiples es estructurado para usar menos espacio que uno de una sola pasada, puesto que el espacio ocupado por una pasada puede ser reusado por la siguiente. Un compilador de pasadas múltiples es, por supuesto, más lento que uno de una sola pasada ya que en cada pasada se lee y escribe de y para un archivo intermdio; otros de los factores que inciden en el número de pasadas para ser usadas en un compilador particular pueden ser la memoria disponible, velocidad y tamaño de el compilador, velocidad y tamaño del programa objeto, etc.

Como ya se ha mencionado el concepto de compilador se ha extendido en diversas áreas de la ciencia y tecnologia, bien pues una de estas muchas áreas es la Ingeniería en Electrónica en el campo de diseño de circuitos **VLSI.** Este proceso de diseño se puede modelar como un traductor de un lenguaje fuente de alto nivel a un lenguaje destino de bajo nivel. En este caso un llamado **Compilador para Sistemas Digitales** especifica el ambiente y composición de un circuito implementado con circuitos **VLSI** colectando toda la información tocante al diseño y llevando acabo un rango de inspección para errores de diseño y finalmente una simulación. Justamente como un compilador ordinario debe entender y hacer valer las reglas de un lenguaje de máquina en particular, un compilador para sistemas digitales debe entender y hacer cumplir las reglas de diseño que dicte la flexibilidad de un circuito dado.

A la afirmación antes expuesta podrían surgir una diversidad de preguntas como: ¿es posible construir un compilador de esta clase? ¿se puede definir y modelar un compilador para sistemas digitales con la misma estructura y filosifía de un compilador de lenguajes de programación de alto nivel?. A estas preguntas y tantas otras podemos decir que el objetivo de este trabajo es proporcionar una herramienta que nos permita analizar y sintetizar el proceso de diseño de un circuito, que es la escencia de un compilador. En el resto del capítulo se establecerá el concepto y estructura de este tipo de harramientas.

III.4 COMPILADOR PARA SISTEMAS DIGITALES

- **Arquitectura global del sistema**

Una descripción escrita en un **HDL** multinivel puede ser la primera entrada a un número de herramientas de software que incluyen manipulación gráfica, simulación, síntesis, verificación formal,

generación de patrones de prueba, etc. En la actualidad la tendencia es integrar todas estas herramientas en un ambiente dentro de un solo sistema **CAD**. Una jerarquía con estas características, mostrada en la Figura 3.6, despliega la estructura parcial del sistema **CASCADE** .

Sea cual sea la aplicación, una descripción sufrirá dos distintas fases de proceso, como es mostrado en la Figura 3.7.

Fase 1

En la primera fase es la orientación del lenguaje. En un **HDL** multinivel, este es parametrizado con el nivel del lenguaje a ser usado. En esta fase se construye y verifica el modelo, independientemente de la herramienta **CAD** que se aplicará al modelo.

En esta se desarrollan el análisis de léxico, sintaxis y semántica, a fin de verificar que la descripción es correcta con respecto a la definición del lenguaje y aun más importante es, satisfacer las restrcciones impuestas en el diseño de hardware para ese nivel (*estas restricciones pueden ser parte del nivel semántico HDL*). Esto corresponde al ambiente modelado en la Figura 3.6.

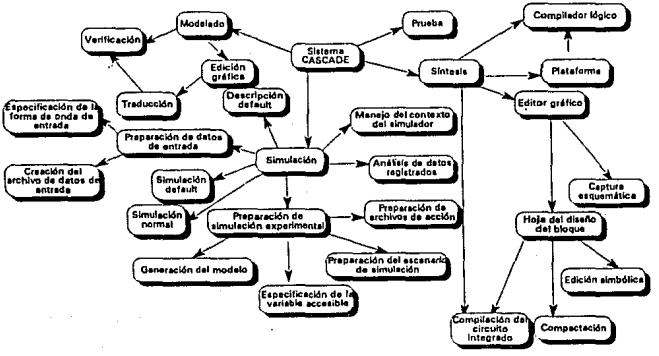


Figura 3.6 Ambiente de CASCADE

- **Edición del modelo**

Las descripciones definidas por el usuario deben primero ser editadas dentro de un archivo de computadora. Un editor de textos de propósito general puede ser usado; pero tales editores no tienen

información a cerca del significado de la cadena de caracteres que es manipulada, y no brinda ayuda al usuario para la construcción de su modelo.

Algunos editores gráficos de propósito especial han sido utilizados por muchos años y, muchos de estos están comercialmente disponibles para descripciones de nivel gate o eléctrico. En este caso, el usuario no ve igual una descripción textual. El modelo es construido como la interconexión de primitivas o módulos definidos previamente. Sin embargo, estas herramientas son ligadas a un solo nivel de descripción, donde un solo tipo de carrier y, de valor es disponible. Otra fuerte limitación se basa en el hecho de que solo las descripciones estructurales son soportadas; el usuario no tiene medios para definir el funcionamiento directo de un módulo: el diseñador puede sólo interconectar componentes existentes.

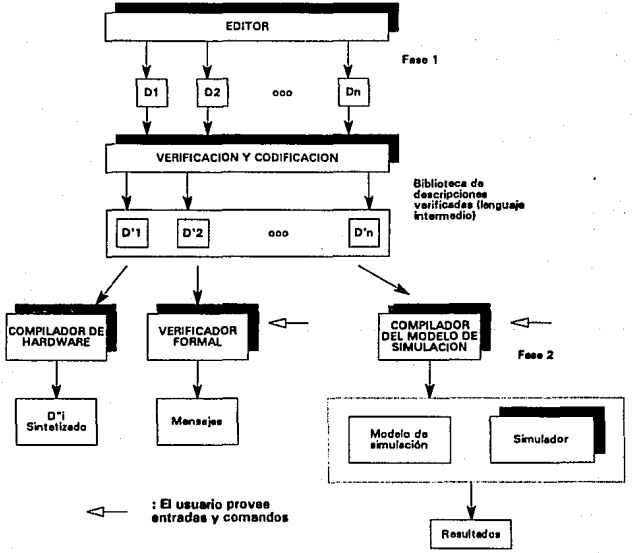


Figura 3.7 Procesando una descripción

Los **HDLs** multinivel requieren de editores más elaborados, parametrizados con el nivel del lenguaje con capacidades textuales y gráficas mezcladas. Plantear la referencia al nivel de lenguaje hace disponible inmediatamente todas las componentes primitivas para ese nivel, en cuanto a los editores de nivel simple discutidos previamente. En conclusión, para el nivel **RT** y más abstractos, el usuario debe ser capaz de especificar información adicional como dimensiones, etc. Un switch a una sintaxis dirigida a un editor textual debe, como ya es realizado en ambientes de programación, ayudar al usuario a crear descripciones funcionales sintácticamente correctas.

Aun cuando un editor que cubra todos estos requerimientos no se ha producido, recientes progresos en la Ingeniería de Software ha fomentado, para equipos de **CAD**, la implementación de ambientes amigables al usuario.

- **Verificación y Codificación**

El empaquetamiento de la verificación y codificación corresponde a un compilador "**front-end**", y debe ser común para todas las aplicaciones. Naturalmente, la verificación de cada módulo de descripción debe ser realizado independientemente, y debe producir una salida de forma interna para cada uno. Cuando un módulo se describe como una network de unidades, solo las interfases de unidades anidadas son proporcionadas al verificador. Los principios del verificador son mostrados en la Figura 3.8

Además del usual análisis de léxico y sintáctico, el número máximo de verificaciones de semántica estática debe ser ejecutada en este punto; incluyendo las siguientes lista de verificaciones:

- tipo y dimensión compatibles en expresiones
- interconexión real de unidades anidadas con respecto a las interfases formales declaradas
- contexto en el cual carriers son modificados, de acuerdo a su tipo declarado
- limitaciones de fanout, etc.

La forma interna verificada puede ser dividida en una codificación del encabezado de la descripción (*nivel del lenguaje de referencia, identificador, interfase*), y una codificación del cuerpo de la descripción (*descripción estructural/funcional*). Dependiendo de la subsecuente aplicación, solo un aspecto puede ser empleado. En la unión de los dos:

- no se debe perder información con respecto a la descripción fuente **HDL**;
- se debe permitir una traducción de regreso al programa fuente;
- se requiere una generalidad para una variedad de aplicaciones;
- se necesita una fácil manipulación, transformación y vizualización.

En general, la forma interna verificada es una combinación de tablas y registro de estructuras. Se han propuesto formas internas estándares, a fin de permitir la transferencias entre diferentes sistemas de CAD.

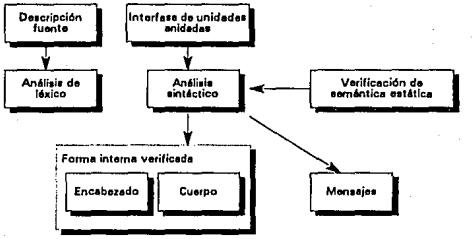


Figura 3.8 Verificación de un programa fuente

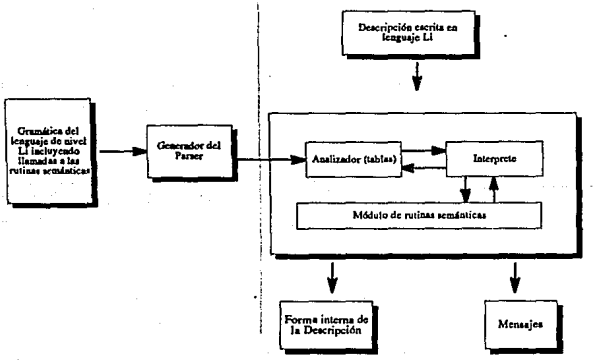


Figura 3.9 Construcción del verificador empleando un Compilador de compilador

- **Herramientas para la escritura del compilador**

Una de las especificaciones semánticas para un **HDL** han sido claramente identificadas, la implementación de la fase 1 es muy similar en naturaleza a la escritura de un ambiente de programación y de un compilador. Para un **HDL** multinivel, la complejidad es del mismo orden de magnitud que en un compilador **ADA.**

La utilización de los mejores paquetes de Ingeniería de Software tecnológicamente es la llave para producir ambientes de modelado eficientes, confiables y potentes. *P. ej.*, sería muy ventajoso utilizar un meta editor, parametrizado con la sintaxis del lenguaje fuente, para construir editores de propósito especial para cada nivel **HDL**.

El empleo de un compilador de compilador es hoy en día ampliamente aceptado. La Figura 3.9 muestra los principios de la construcción de un verificador de sintaxis directa, utilizando un parser típico.

En el lado derecho de la Figura 3.9, la sintaxis se proporcion bajo un formato apropiado, que debe cumplir las restricciones impuestas por el generador de parser, (*los europeos prefieren parser predicativos basados en gramáticas LL(1), mientras que los diseñadores americanos se inclinan más por LR(1) o LALR basadas en técnicas "bottom-up"*). Las llamadas a las rutinas semánticas son incluidas en la sintaxis.

El generador de parser produce un conjunto de tablas las cuales son ligadas a un intérprete estándar, y a los módulos de verificación y codificación de las rutinas semánticas proporcionadas por el escritor del compilador. El resultado es un verificador de sintaxis directa que acepta como entrada una descripción de usuario escrita en un lenguaje de referencia Li, y como salida la forma de descripción interna y mensajes de error posibles (*lado derecho de la Figura 3.9*)

Fase 2

La fase 2 esta orientada a la aplicación. Esta toma las salidas de la fase 1 en la forma interna estándar, y ejecuta una o varias de las siguientes tareas:

- Edita una liga entre todas las formas internas de las diferentes unidades, cuando un modelo es descrito como una red interconectada de sub-módulos.

- Transformaciones directas en la forma interna para producir un modelo modificado de el mismo circuito (*optimizaciones, síntesis*).

- Una traducción en otro código, que sea requerido como entrada para una aplicación existente, o que pueda ser procesada más eficientemente por algún algoritmo (*prueba, pruebas formales, valor verdadero y simulación por default*). Esto es muy similar a generación de código en compiladores optimizados.

Este trabajo no cubre todos los aspectos relacionados con la fase 2 para los diferentes ambientes de aplicaciones. Nos limitamos al planteamiento del problema para la producción de un modelo de simulación multinivel, e insistimos en la cohabitación de problemas que deben ser tomados en cuenta

- entre los niveles de abstracción
- entre los algoritmos de simulación y los modos de programar, que serán por lo tanto serán revisados primero.

"La Ciencia se Compone de Errores,
que a su Vez son los Pasos
Hacia la Verdad"

Julio Verne
Novelista Francés
[1828-1905]

IMPLEMENTACION

La implementación del **Compilador para Sistemas Digitales** es desglosada en este capítulo a través del diseño de un multiplicador binario que ilustra de forma adecuada una metodología de diseño basada en las características de los **Lenguajes de Transferencia de Registro** y el **diseño top down**, que esperamos motive al lector para adoptarla en sus propios diseños. Dicho compilador recolecta la información respecto al diseño y ejecuta un rango de verificaciones para los errores de diseño. *P. ej.*, se verificará que las dimensiones de los registros no estén excedidas. También reportará al usuario sobre las conexiones hechas para cada pin en cada paquete de circuito integrado empleado e indicará que pines no han sido asignados.

En esta metodología, aplicada al Diseño de Sistemas, el diseño top down empieza con una especificación de los requerimientos del sistema. En este nivel de descripción del sistema, al que llamaremos nivel top, solo los detalles de funcionamiento de las entradas y salida requeridos son detallados, en tanto que los detalles de la implementación real serán especificados hasta más tarde.

Tanto el código fuente del compilador como la gramática del lenguaje **RTL** implementados para cumplir el objetivo de presente trabajo son proporcionados como anexos a este capítulo.

IV.1 DISEÑO DE UN MULTIPLICADOR BINARIO

El proceso de multiplición binaria puede ser ejecutado exactamente de la misma manera como se realiza la multiplicación en el sistema decimal. Supongamos que queremos multiplicar el número (***multiplicando***) binario 1100 por el número (***multiplicador***) binario 1011. El proceso de multiplicación inicia multiplicando el multiplicando por el dígito menos significativo del multiplicador (*1 en este caso*) y escribir el resultado (***primer producto parcial***) inmediatamente después de la línea horizontal dibujada abajo del multiplicador. Después multiplicamos el multiplicando por el segundo dígito menos significativo, nuevamente 1, y escribir el resultado inmediatamente después del primer producto parcial desplazado un dígito a la izquierda. Este procedimiento continua en forma óbvia hasta obtener el resultado final que es el producto de la suma de todos los productos parciales.

```
    1100
    1011
  ------
    1100
   1100
  0000
 1100
------
10000100
```

Figura 4.1. Ejemplo de Shift - and - add

Como es posible apreciar, se puede diseñar un sistema que desarrolle este procedimiento para realizar la multiplicación de dos números cualesquiera de cuatro bits. En la Figura 4.1 se puede apreciar que el

procedimiento puede ser implementado en la forma de un algoritmo "**shift-and- add**". En términos de la Figura 4.1 el procedimiento propuesto para realizar esta implementación es como sigue. El primer producto parcial, que es igual al multiplicando, es almacenado en un registro al cual, por conveniencia, se referirá como "acumulador". El segundo producto parcial, igual al multiplicando dezplazado un dígito a la izquierda; es adicionado al contenido del acumulador. El tercer producto parcial es igual a cero por tanto no nececitamos adicionarlo al acumulador. El cuarto producto parcial, igual al multiplicando dezplazado tres dígitos a la izquirda, es sumado al contenido del acumulador, de modo que el acumulador contendrá el producto final completo.

Para la parte de datos del problema, la elección de dispositivos de hardware se presenta muy clara. Necesitamos dos registros de 4 bits, uno para almacenar al multiplicador y otro para el multiplicando. Un registro adicional también es requerido para el acumulador. Dado que el producto de dos números binarios consiste de hasta 8 dígitos binarios, podría pensarse que se requiere de un registro de 8 bits para esta función, sin embargo, dada la forma de trabajar del algoritmo es posible implementar el sistema empleando solo tres registros de 4 bits y un flip-flop.

Una vez que cada dígito del multiplicador ha sido empleado para formar el producto parcial correspondiente no es requerido por más tiempo. De esta forma, después de formar cada producto parcial, el espacio de almacenamiento ocupado por este dígito puede ser sobreescrito, eleminando así la necesidad de un registro de 8 bits.

El algoritmo sugerido por nosotros puede ser declarado como a continuación se muestra. La notación **REG<0:m>** denota un registro llamado **REG** con **m** + 1 espacios de almacenamiento numerados desde 0 a m.

Los dispositivos de almacenamiento son los siguientes:

A<0:3> Contendrá al multiplicando;
B<0:3> Contendrá (inicialmente) al multiplicador;
C<0:3> Será empleado en la acumulación del producto;
OVR Es un filp-flop que almacena una bandera de overflow existente para el sumador.

Ademas, elegimos emplear un contador para obtener una pista de los pasos del algoritmo y otro flip-flop, llamado **DONE**, que fungirá como bandera para señalar el término de la tarea.

Algoritmo

Paso1: Inicializar el contador a 3; inicializar **DONE**=0; limpiar **C** y **OVR**, asignar al multiplicando y el multiplicador a los registros **A** y **B** respectivamente.

Paso2: if **B<3>** = 1 then cargar **C** con la adición de **C** y **A** y cargar **OVR** con el carry del sumador.

Paso3: Desplazar el contenido de **B** y **C** un bit a la derecha, colocando **C<3>** dentro de **B<0>**; igualar **C<0>** a **OVR**; decrementar el contador en 1.

Paso4: Limpiar **OVR**; if counter = 0 then **DONE** = 1 else ir al paso 2.

En el **Paso1** simplemente se inicializan los contenidos de los dispositivos de almacenamiento y el contador.

El **Paso2** indica que la adición de un producto parcial al acumulador es solo necesaria para los dígitos no cero en el multiplicador. Note también que la adición de dos números de 4 bits puede llevar en su resultado un bit 5 y, a fin de cuidar este hecho, este carry de salida estará alimentando al flip-flop **OVR**.

En el **Paso3**, las operaciones de corrimiento requeridas son ejecutadas. En términos de la multiplicación detallada en la Figura 4.1, después de que el primer producto parcial es adicionado al registro **C**, este contiene 1100. El registro **B** contiene 1011, y el dígito final de la palabra de 4 bits no es por más tiempo necesario. El contenido de **B** es corrido un bit a la derecha y el dígito a la derecha de **C** es colocado dentro de **B<0>**. Simultáneamente, el contenido restante de **C** es corrido un bit a la derecha y la salida del flip-flop **OVR** (que es igual a 0 en estos momentos) es colocada en **C<0>**.

En el **Paso4** es necesario limpiar **OVR** porque no tenemos especificado alguna acción en el **Paso2** en ocaciones cuando **B<3>** = 0. En tales ocaciones, la adición no es requerida y por tanto el dígito de carry no puede ocurrir, es importante limpiar **OVR** en caso de que este retenido un dígito de carry de una operación de adición previa. La secuencia completa de operaciones mostrando cómo el contenido de los registros **B** y **C** cambian está dado en la Figura 4.2: cuando el contador alcance 0, el contenido de **C** y **B** darán el producto requerido.

IV.1.1 UN PRIMER NIVEL DE DESCRIPCION RTL

En este momento estamos seguros de que el algoritmo propuesto trabajará y por tanto podemos expresarlo formalmente en un **Lenguaje de Transferencia de Registro**. El algoritmo se ha definido como un módulo (Figura 4.3) con la clara implicación que el sistema de hardware diseñado tiene el potencial de emplearse en un sistema grande. Conviene aclarar que el **RTL** construido para este propósito experimente algunas similitudes con el Lenguaje de Programación Pascal. En la Figura 4.3 el módulo en cuestión es nombrado "**multiplier**" y tiene una lista de parámetros asociados que describen el conjunto completo de terminales (*entradas y salidas*) que el sistema requiere tener. Así el módulo tiene cuatro terminales para leer los 4 bits del multiplicador y otras cuatro para el multiplicando. También tiene ocho terminales que permiten la lectura de los 8 bits del producto y tres terminales adicionales, una para recibir el comando de inicio, otra para la recepción de los pulsos de reloj y la tercera como bandera que indica cuando la operación de multiplicación ha sido concluida. Dentro del multiplicador un número de registros son requeridos; este

requerimiento ya ha sido identificado en nuestro ejemplo preeliminar y el registro listado en la Figura 4.3 muestra las dimensiones necesarias en cada caso. Note que en los casos en que la dimensión no es declarada (*como con OVR*) el compilador asumirá un registro de 1 bit. También hay que notar que, en esta etapa, no nos atañen los tipos específicos de dispositivos y representamos todos los elementos solo en términos de registros y terminales. Así, el término "**registro**" es una descripción general y de aquí que **contador** aparezca en la lista de registros. El punto importante aquí, entonces, es que la descripción **RTL** en la Figura 4.3, esta intenta ser independiente de la tecnología y tipo de dispositivo. La descripción representa una etapa en un diseño top down para el diseño del multiplicador. Desde la especificación de detalles se ha evitado tanto como es posible algún diseño ingenieril, dando esta descripción **RTL**, debemos ser capaces de realizar el multiplicador empleando cualquier dispositivo disponible apropiado para la tarea.

Paso	counter	C	B	OVR	DONE
1	4	0000	1011	0	0
2	4	1101	1011	0	0
3	3	0101	0101	0	0
4	3	0101	0101	0	0
2	3	0010	0101	1	0
3	2	1001	0010	1	0
4	2	1001	0010	0	0
2	2	1001	0010	0	0
3	1	0100	1001	0	0
4	1	0100	1001	0	0
2	1	0000	1001	1	0
3	0	1000	0100	1	0
4	0	1000	0100	0	1

Figura 4.2 Explicación del algoritmo shift - and - add

El sistema tendrá cuatro etapas que se encuentran listadas en la línea 3.

Ahora veremos el cuerpo de la descripción **RTL**. El primer enunciado (línea 5) debe leerse "las terminales producto están conectadas a la concatenación de los registros **C** y **B**". Básicamente este es un enunciado que dice que el producto estará disponible desde los registros **B** y **C**, donde los dígitos más significativos son proporcionados por **C**.

En la línea 6 la descripción **RTL** contiene la etiqueta **Paso1** indica la entrada al primer estado del algoritmo de control. En este estado, el sistema espera por el pulso de inicio aplicado externamente. El enunciado **on** *Start* **do** indica acciones que deben ser ejecutadas cuando Start toma el valor lógico de 1.

Claramente, la sentencia **on...do** juega un papel similar a una sentencia **if...then** en Pascal. Una gran diferencia entre el **RTL y Pascal** es también evidente en este punto. Cuando el pulso Start llega, se requiere ejecutar un número de operaciones simultáneamente (*estas deben ser ejecutadas al menos dentro de un solo periodo de reloj*). Las sentencias listadas de la línea 8 a la 13 de la descripción **RTL** representan operaciones de hardware que son ejecutadas simultánamente; esto contrasta con el Pascal ordinario, donde una sentencia es ejecutada después de otra, en un estilo estrictamente secuencial. La línea 8 indica que la entrada al *counter* debe estar permanentemente conectada a la constante 4, y que la constante debe de ser un valor de 4 bits (*la construcción empleada tiene la representación general valor%tamaño*).

```
1   modulo multiplier(Multiplier<0:3>, Multiplicand<0:3>, Producto<0:7>, Start, DONE, clock);
2   register A<0:3>, B<0:3>, C<0:3>, OVR, counter<0:2>;
3   edo Paso1, Paso2, Paso3, Paso4;
4   begin
5         Producto ← C @ B;
6         Paso1:
7          on Start do begin
8               counter ← 4%4;
9               A ← Multiplicand;
10              B ← Multiplier;
11              OVR ← 0 ;
12              C ← 0
13              goto Paso2
14         end else
15              goto Paso1;
16        Paso2
17         begin
18              on B<3> do begin
19                    add(C, A, OVR);
20              end;
21              goto Paso3
22         end;
23        Paso3
24         begin
25              shiftright(B, C<3>);
26              shiftright(C, OVR);
27               counter ← counter -1;
28              goto Paso4
29         end;
30        Paso4
31         begin
32              OVR ← 0;
33              on counter = 0 do begin
34                   DONE ← 1;
35                   goto Paso1
36              end else
37              goto Paso2
38         end
39  end {multiplier}
40  $
```

Figura 4.3 Primer nivel de descripción RTL del multiplicador

Antes de pasar al **Paso2** en la descripción **RTL**, mencionarames algunos puntos de interés. Hay que notar que las líneas de la 8a la 12 especifican la transferencia de datos y que la instrucción 13 es una sentencia de control especificando un cambio de estado. También es importante señalar que este **RTL** permite una sentencia **else** para facilitar la escritura del código. En este caso la sentecia **else** proporciona un significado simple de declaración que cuando el sistema está en el primer estado (**Paso1**) requiere esperar el pulso de inicio. Finalmente note que hay flexibilidad considerable respecto de la duración permitida del pulso de inicio. Este debe ser, por supuesto, lo suficientemente largo para establecer al multiplicador, la otra única restricción es que este no debe exceder más alla del tiempo tomado para completar la operación de multriplicación - si esto sucedio, el sistema debe reinicializarse tan pronto como haya completado la tarea.

Ya una vez en el **Paso2**, declaramos que, si **B<3>** es igual a 1, tenemos que ejecutar una operación de adición. No adentraremos en detalles acerca de como la adición será ejecutada dado que hemos empleado

una forma de diseño top down y detalles como este pueden quedar hasta más tarde. El bosquejo del **RTL** no incluye ninguna notación estándar para las diferentes operaciones de adición posibles, simplemente empleamos el operado no estándar "add(**C, A, OVR**)" que indica la adición de **C** a **A** con el resultado almacenado en **C**, y el bit de carry en **OVR**. Finalmente hay que notar que a pesar del valor de **B<3>**, el sistema requiere ir al **Paso3**; Cuando **B<3>** tiene el valor de 1, la sentencia **goto** es ejecutada simultáneamente con la operación de adición.

Entrando al **Paso3**, otra operación no estándar es encontrada. El lenguaje básico no proporciona operaciones para corrimiento a la derecha. La línea 25 de la Figura 4.3 declara que el registro **B** está recorriendo un bit a la derecha, y coloca el valor del bit menos significativo del registro **C** (*p. ej.*, **C<3>**) dentro de la posición vacante en **B** (*p. ej.*, dentro del bit **B<0>**). Similarmente, la línea 26 en la Figura 4.3 establece que el registro **C** está corriendo un bit a la derecha, y coloca el valor retenido en **OVR** dentro de la posición del bit vacante (*p. ej.*, *dentro de* **C<0>**).

El resto del listado es muy claro y el lector no debe tener dificultad para establecer que en este se describe fielmente el procedimiento shift-and-add ilustrado en la Figura 4.2.

Con la descripción **RTL** concluida, un **simulador** puede ser utilizado para verificar que el sistema verdaderamente ejecuta las funciones requeridas.

Hata este momento el sistema es totalmete independiente de la tecnología y del tipo de dispositivos. Pero es el momento en el que tenemos que trasladarnos a un nivel más bajo de descripción de comportamiento del sistema tomando desiciones acerca de los tipos de dispositivos que utilizaremos en la realización final del sistema. Como primer paso, para adentrarnos en esta descripción de nivel más bajo recolectaremos toda la información referente a los diferentes registros en el sistema en la forma de una tabla resultará muy util. Para ejemplo del multiplicador esta información ha sido recolectada, directamente de la descripción **RTL**, en la Tabla 4.1.

Nombre	Número de línea	Operación	Fuente/Destino
A	9	Carga	Controlador
	9	Dato de entrada	Multiplicand<0:3>
	19	Dato de salida	adiciona entrada #2
B	5	Dato de salida	Producto<4:7>
	10	Carga	Controlador
	10	Dato de entrada	Multiplier<0:3>
	18	Dato de salida	Controlador
	25	Shiftright	Controlador
	25	Correr dato	C<3>
C	5	Dato de salida	Producto<0:3>
	12	Inicializar	Controlador
	19	Dato de salida	adiciona entrada #1
	19	Dato de entrada	adiciona salida
	19	Carga	Controlador
	26	Shiftright	Controlador
	26	Correr dato	OVR
OVR	11	Inicializar	Controlador
	19	Dato de entrada	Carry de salida del sumador
	19	Carga	Controlador
	26	Dato de salida	C_shiftin
	32	Inicializar	Controlador
counter	8	Carga	Controlador
	8	Dato de entrada	4 bits de constante 4
	27	Decrementar	Controlador
	33	= 0 ?	Controlador
Multiplicand	9	Dato de salida	A<0:3>
Multiplier	10	Dato de salida	B<0:3>
Producto	14	Dato de entrada	C<0:3> y B<0:3>
Start	7	Dato de salida	Controlador
DONE	34	Inicializa implícitamente a 0	Controlador
add	19	Dato de entrada 1	C<0:3>
	19	Dato de entrada 2	A<0:3>
	19	Dato de salida	C<0:3>
	19	Carry de salida	OVR

Tabla 4.1. Especificación de las características de los registros y dispositivos

La Tabla 4.1 lista los tipos de operacione que debemos ejecutar para cada registro, e indica también las conexiones que debemos hacer entre los registros (*y para el controlador*), de modo que los datos y señales de control puedan ser transferidos en la manera requerida. Con las conexiones requeridas dentro del sistema, el diagrama de bloques de la parte de datos y parte de control pueden ser dibujados (Figura 4.4 y Figura 4.5 respectivamente).

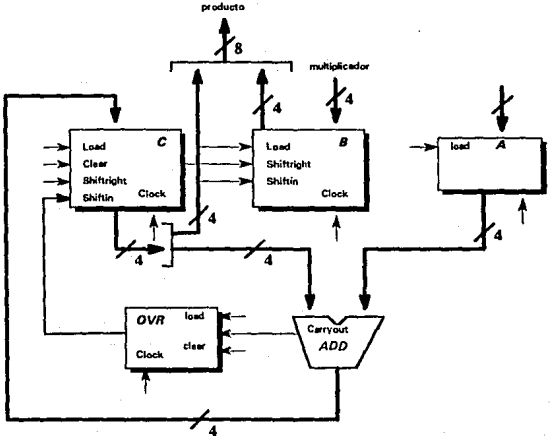


Figura 4.4. Parte de datos del multiplicador

- **Temporizado del Sistema**

Antes de discutir la selección de los dispositivos, debemos dedicar un poco de atención a la manera en la cuál la señal de reloj gobernará el comportamiento del sistema. A fin de ver por qué, asumiremos que dispusimos el sistema de modo que el controlador y todos los registros responden al flanco de subida del reloj.

Ahora. en la descripción **RTL** en la Figura 4.3, los cuatro pasos del algoritmo de control son claramente definidos. Cada uno de estos pasos representa un estado diferente del controlador y, como el controlador cambia de un estado a otro, este requiere de señales de control, que iniciaran las diferentes acciones requeridas en cada paso. Asi, cuando el controlador percibe la señal de inicio este producirá las señales

que activaran la tranferencia de los datos listadas de la línea 8a la 12. El controlador responderá a la señal de inicio en un flanco de subida del reloj (*llamaremos a este flanco de subida número 1*), pero habrá un delay antes de que las señales de control aparezcan en las salidas del controlador. Como consecuencia, los registros recibiran las señales de control algún tiempo después de que el flanco de subida ha pasado. Esto sugiere que los registros no responderan a la señal de transferencia de datos sino hasta el siguiente flanco de subida del reloj (*flanco de subida número 2*).

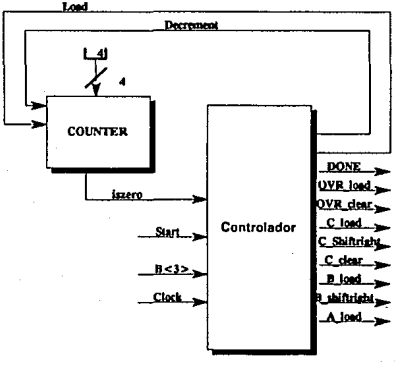


Figura 4.5 Parte de control del multiplicador

Cuando el flanco de subida 2 ocurre, no solo la transferencia de datos del **Paso1** tomará lugar, sino que también el controlador responderá al flanco para pasar al **Paso2** y producir (*si B<3> es un 1*) las señales de control que ejecutarán las acciones descritas en la línea 19. Notar que, una vez más, estas señales de control aparecerán en las salidas del controlador después de un delay y que los registros solo responderán a las señales de control cuando el flanco de subida del reloj 3 ocurra.

La manera en la cual el sistema opera puede resumirse como sigue: El controlador genera señales de control en el inicio de un ciclo de reloj y los registros responden a estas señales en el inicio del siguiente ciclo de reloj.

Esto no dificulta ver que, este modo de operación permitirá la correcta realización del sistema descrito en la Figura 4.3. *P.ej.*, supongamos que tuvieramosmos la sección de la descripción **RTL**, con **A** y **B**

representando registros de 4 bits.

```
Paso10
    begin
        B <- A;
        A <- B;
        on B<3>   do begin
                     .
                     .
                     .
        end
    end
```

La sentencia **B <- A** implica que los contenidos de los registros **A** y **B** están siendo intercambiados. La sentencia **on B<3> do begin** será seguida por un conjunto de acciones que son ejecutadas si **B<3>** es un 1. Hay que notar que el sistema pasa al **Paso10** en un flanco de subida (*llamado i*) y que este será un flanco de sebida **i + 1** que causa que **A** y **B** intercambién sus contenidos. La prueba en el valor de **B<3>** toma lugar

al tiempo de llegada del flanco de subida del reloj **i + 1** y, por tanto ocurrirá antes de que los registros **A** y **B** hayan tenido tiempo de intercambiar sus contenidos. Así, la prueba será realizada en el valor de **B<3>** antes de que el **Paso10** haya entrado.

En tanto que consideraciones, tales como esta, estén producidas claramente en mente, el **RTL** puede ser libremente utilizado en situaciones donde todos los dispositivos responden al flanco de subida.

Una solución alternativa es utilizar la idea de un reloj de dos fases, donde el controlador responde al flanco de subida del pulso de reloj y los registros al flanco de bajada del pulso. Se pueden utilizar inversores en la linea de reloj, para ocacionar que los registros respondieran al flanco de bajada *(recordando como este es realizado en el flip-flop maestro-esclavo)*. En el esquema de reloj de dos fases, la generación de señales de control y la respueta de los registros **A** estas toma lugar en el mismo ciclo de reloj. Esto puede, sin embargo, conducirnos a complicaciones y es difícil implementar en el **RTL**. Así en este trabajo nos restringiremos al caso en donde todos los dispositivos responden al flanco de subida del pulso de reloj.

Ahora que hemos establecido el modo de operación que queremos que nuestro sistema siga, debemos verificar si esta impone algunos requerimientos específicos en la entrada de la señal Start y también como afectará el valor de la bandera **DONE** cuando las multiplicación es concluida.

Claramente, la señal Start debe de satisfacer el "**set up**" y soportar los tiempos del controlador. Asumiremos que la señal de Start es proporcionada al controlador por un flip-flop que es disparado por el reloj del sistema (*este es un método típico de sincronizar entradas externa con el reloj del sistema*). Con

esta suposición, se sigue que la señal Start llegará al controlador con un delay en en flip-flop después del flanco de subida del reloj del sistema. Este, por lo tanto, tiene que ser retardado por un periodo de reloj completo a fin de satisfacer el set-up y mantener los tiempos en el controlador. El flip-flop que suministra la señal Start puede ser inicializado en el siguiente flanco de reloj porque la propagación del delay del flip-flop será suficiente para empalmar al alacanzar el tiempo requerido. La señal Start puede ser mantenida por un largo periodo, pero debe ser inicializada antes de que la operación de multiplicación sea completada.

La bandera **DONE** debe ser establecida en un ciclo de reloj después de que los valores finales son disponibles en la salida del producto y, permanecer fijo solo por un ciclo de reloj. Esto es porque no tenemos asociado ningún dispositivo de almacenamiento con esta señal. Nuevamente se asume que la lógica externa interpretará correctamente esta señal para probar esta en un flanco de subida de reloj.

Ahora estamos en posición de identificar los tipos de dispositivos necesario para realizar el sistema final.

- **La selección de los Dispositivos**

El proceso de selección de dispositivos depende de los recursos disponibles al diseñador. En una gran organización, el diseñador puede tener acceso a una base de datos actualizada, la cual contiene los detalles de todos los dispositivos disponibles en la organización. En otro caso el diseñador tendrá que confiar en sus hojas de especificaciones y catálogos. En cualquiera de los dos casos, el diseñador ordenará los recursos disponibles a fin de empalmar las caracteristicas requeridas por los registros con las características de los dispositivos disponibles, *p. ej.*, la Tabla 4.1 nos dice que el registro A requiere tener cuatro pines de entrada, cuatro pines de salida y un pin de carga para controlar la operación de carga del registro. Consultando las hojas de especificaciones observamos que el flip-flop D 74379 es un primer candidato para la implementación de ésta función.

El 74379 tiene los cuatro pines de entrada requeridos junto con los cuatro pines de salida para lectura de los 4 bits almacenados en la palabra. También tiene un pin de reloj y un pin de carga, el cual habilita la carga en el siguiente flanco de subida del reloj. Además, existen 4 pines que permiten leer el complemento de cada uno de los bits almacenados los cuales no son necesarios para nuestro diseño. Es muy frecuente que en el diseño de sistemas digitales algunos de los pines en el dispositivo seleccionado no sean requeridas para la implementación de la función que está siendo implementada. En tales circunstancias, los pines no usados deben ser cuidados de manera apropiada. Sin embargo, el problema de asignar una conexión apropiada a pines no utilizados es un por menor cuya consideración es más apropiado para un nivel bajo en el proceso de diseño (*top down*).

No experimentamos dificultad en localizar, dentro de los manuales de especificaciones, un dispositivo

que pueda ser utilizado para proporcionar exactamente el comportamiento requerido por el registro **A**. En realidad, el lector debe tener muy poca dificultad para encontrar un dispositivo apropiado para cada uno de los registros listados en la Tabla 4.1. Esto se debe a que el multiplicador que hemos elegido como ejemplo es realmente un sistema muy simple. En el diseño de sistemas más grandes y complejos, frecuentemente pasa que los dispositivos no pueden ser encontrados (*en manuales de especificaciones o catálogos*), en tales circunstancias, es necesario introducir (*dentro del proceso de diseño*) un nivel adicional, en el cual el comportamiento del registro requerido es construido con los componentes que son disponibles. Esto es una característica muy importante del proceso de diseño top down y deseamos ilustrar su implementación aquí. a fin de realizar esto, ignoraremos el hecho de que las características requeridas por los registros **B** y **C** pueden ser proporcionadas por el registro de corrimiento 74195 y procederemos como si los dispositivos apropiados no estuvieran disponibles para estos registros. Así, la implementación real de los registros **B** y **C** no serán consideradas en este nivel del proceso de diseño y simplemente continuaremos con el proceso de seleccionar los dispositivos para los registros restantes en la Tabla 4.1.

El registro **OVR** puede ser implementado por medio de un flip-flop D dual 7474. Solo uno de los dos flip-flops en el paquete realmente será requerido para realizar **OVR**, pero el diseño del multiplicador no es concluido todavía y algún posible uso podría ser encontrado en una etapa más adelante para el otro flip-flop. Para el contador podemos utilizar un 74169.

Ninguno de los componentes restantes en la columna 1 de la Tabla 4.1 es un registro, esto es, ninguno es un dispositivo de almacenamiento de información. El **Multiplicador** y el **Multiplicando** son datos de entrada; el **Producto** es un dato de salida; **Start** es una entrada de control y **DONE** es una salida de control.

El elemento final en la columna 1, nombrado "Add" es simplemente un sumador y esta disponible como un 7483.

IV.1.2 UN SEGUNDO NIVEL DE DESCRIPCION RTL

Para propósitos de ilustración, hemos procedido hasta este punto suponiendo que el dispositivo apropiado para la implementación de los registros **A** y **B** no es disponible. En este punto del proceso de diseño entonces, hemos seleccionado dispositivos para todos los registros cuyas características están disponibles en manuales de especificación, pero no así para los registros **B** y **C** cuya selección es aplazada hasta el siguiente nivel de diseño.

Antes de que el siguiente nivel de diseño sea iniciado, una nueva descripción **RTL** debe ser producida, dando detalles de cómo los dispositivos seleccionados están ejecutando las operaciones requeridas. Esta descripción incorporará detalles del esquema de interconexión para los diferentes dispositivos. Existen algunos esquemas para lograr esto automáticamente, *p. ej.*, **DAA**, pero se requiere de vastos recursos computacionales. Con el bosquejo del esquema **RTL** este trabajo ha sido realizado por el usuario. Aunque

el esquema **RTL** no proporciona un gran negocio de automatización en el proceso de diseño, restringe al diseñador a seguir las reglas básicas del diseño top down.

Para obtener el segundo nivel de descripción **RTL**, regresaremos al primer nivel de descripción y completaremos las sentencias en la Figura 4.3 para acomodar los dispositivos seleccionados para los diferentes registros.

Los registros en la Figura 4,3, en el nuevo nivel de descripción, son definidos ahora como módulos, justo como el multiplicador en si mismo fue definido como un módulo en el previo nivel de descripción.

El nuevo nivel de descripción se muestra en la Figura 4.6. Note que la línea 2 en la Figura 4.3 ha sido remplazada por las líneas de la 2 a la 11 en la Figura 4.6. De la línea 2 a la 6 se especifican los tipos de dispositivos que proponemos utilizar, incluyendo que aun tengan que ser diseñados (*en este caso todavía tenemos que diseñar un dispositivo con las características de los registros B y C y ha sido llamado módulo Shifter en la línea 3*). De las líneas 7 a la 11 se muestra cuantos de cada tipo de dispositivo son requeridos, asociando el nombre de los registros con cada tipo. *P. ej.*, requerimos un 74379 correspondiente al registro A y dos Shifter correspondientes a los registros B y C y así sucesivamente.

La lista de parámetros asociada con cada dispositivo en la líneas de la 2 a la 6 de la Figura 4.6 detallan las terminales que deseamos utilizar en cada uno de los dispositivos seleccionados. *P. ej.*, con el dispositivo 74379 deseamos 4 pines de entrada, 4 pines de salida, un pin de carga y un pin de reloj.

La palabra reservada **external** sigue al módulo de declaración e indica que una descripción detallada del módulo es encontrada en otro lugar. Puede ser encontrada en un manual (*o base de datos*) o venir disponible en el siguiente nivel de descripción de diseño (*como es el caso con el módulo Shifter*).

Las líneas 3 y 4 de la Figura 4.3 aparecen sin cambio como las lines 12 y 13 en la Figura 4.6. La línea 5 de la Figura 4.3 aparece en la línea 14 de la Figura 4.6 y ha experimentado un pequeño cambio. En esta descripción de nivel más bajo, especificamos que los pines de salida de los registros B y C son usados para proporcionar el producto y este es realizado por "appending_out" para cada uno de los nombres de los registros. Empleamos el guión para separar los componentes del nombre del conjunto de terminales referenciadas en el componente. Note que, puesto que el **Producto** es un conjunto de terminales y no un componente, entonces no utilizaremos el guión de referencia para este elemento.

Las líneas 6 y 7 en la Figura 4.3 aparecen sin cambio en las líneas 15 y 16 en la Figura 4.6. Las líneas 8 a la 12 en la Figura 4.3 cambian y aparecen como las líneas 17 a la 21 en la Figura 4.6. En las líneas de la 17 a la 19 se declara que el contador y el registro A y B están siendo cargados (*via su respectivo pin de control de carga*) con el dato que se asumio está disponible en sus entradas cuando las señal de inicio llega.

```
1   modulo multiplier(Multiplier<0:3>, Multiplicand<0:3>, Producto<0:7>, Start, DONE, clock);
2       modulo M74379(in<0:3>, out<0:3>, load, clock) external;
3       modulo Shifter(in<0:3>, out<0:3>, load, shiftright, shiftin, clear, clock); external;
4       modulo M7474(in, out, load, clear, clock); external;
5       modulo M74169(in<0:3>, , load, decrement, iszero, clock) external;
6       modulo M7483(in1<0:3>, in2<0:3>, out<0:3>, carryout); external;
7       component M74379  A;
8       component Shifter   B, C;
9       component M7474   OVR;
10      component M74169  counter;
11      component M7483 Adder;
12      edo Paso1, Paso2, Paso3, Paso4;
13      begin
14          Producto ←  C_out  @ B_out;
15          Paso1:
16              on Start do begin
17                  activate counter _load;
18                  activate A _load;
19                  activate B _load;
20                  activate OVR_clear ;
21                  activate C_clear;
22                  goto Paso2
23              end else
24                  goto Paso1;
25          Paso2
26              begin
27                  on B_out<3> do begin
28                      activate C_load;
29                      activate OVR_load
30                  end;
31                  goto Paso3
32              end;
33          Paso3
34              begin
35                  activate B_shiftright;
36                  activate C_shiftright;
37                  activate couter_decrement;
38                  goto Paso4
39              end;
40          Paso4
41              begin
42                  activate OVR_clear;
43                  on counter = 0 do begin;
44                      activate DONE;
45                      goto Paso1
46                  end else
47                      goto Paso2
48              end
49          A_in ←  Multiplicand;
50          B_in ← Multiplier;
51          C_in ←  Adder_out;
52          Adder_in1 ←  C_out;
53          Adder_in2 ←  A_out;
```

```
54              OVR_in ←    Adder_carryout;
55              B_shiftin ←   C_out<3>;
56              C_shiftin ←   OVR_out;
57              counter_in ←   4 % 4;
58              B_clear ← 0;
59              A_clock ←    clock;
60              B_clock ←    clock;
61              C_clock←    clock;
62              OVR_clock ←   clock;
63              counter_clock ←    clock;
64
65  end {multiplier}
66  $
```

Figura7.6 Segundo nivel de descripción RTL del multiplicador

Similarmente las líneas 20 y 21 indican la necesidad de inicializar los registros **OVR** y **C**.

En el **Paso2**, la línea 28 de la Figura 4.6, que declara "**activate** C_load", implica que la operación de adición **C + A** está siendo ejecutada porque es sobrentendido (*ver* Figura 7.4) que existe una conexión permanente entre las salidas de los registros **C** y **A** y las entradas del **Adder**. Similarmente, la línea 29 de la Figura 4.6 indica que la salida del carry en el Adder es cargada en el registro **OVR**.

En el **Paso3** de la Figura 4.6 se sigue directamente al **Paso3** en la Figura 4.3. La única diferencia significativa es que en el segundo nivel de descripción **RTL** las operaciones requeridas son expresadas en términos de la aplicación real de señales de control para las terminales del dispositivo.

En el **Paso4** el único punto interesante a notar es que la variable **DONE** que representa una terminal de salida del multiplicador, toma el valor 1 en la conclusión del proceso de multiplicación. **DONE** regresará 0 en el regreso al **Paso1**.

Las sentencias restantes en la descripción, líneas 49 a la 63, simplemente definen las conexiones permanentes dentro de la parte de datos del sistema. La línea 57 indica que la entrada al contador debe estar permanetemente conectada **A** la constante 4. Las linea 59 a la 63 indican la conexión del reloj a los dispositivos que lo requieren.

IV.1.3 UN TERCER NIVEL DE DESCRIPCION RTL

En la terminación del segundo nivel de descripción, hemos identificado el tipo de dispositivos que proponemos utilizar para los cuatro registro. Para cada uno de estos dispositivos, hay aun que especificar conexiones para las entradas hasta hora no empleadas. En resumen, por supuesto, aun tenemos que ejecutar el diseño de los registros **B** y **C**.

Comenzaremos el tercer nivel del proceso del diseño completando los detalles de interconexión para los cuatro registros cuyo tipo de dispositivo ya ha sido seleccionado.

- **Terminación de la descripción de Registro A**

En la Figura 4.7a se muestra la especificación final para el registro A que es realizado a través de un circuito integrado 74379. La línea 1 dá el nombre del módulo como este aparecio en el segundo nivel de descripción (*línea 2 de la Figura 4.6*). En el segundo nivel de descripción, la lista de parámetros contiene solo las terminales del 74379 que fueron pertinentes para el problema en cuestión. La línea 2 en la Figura 4.7a lista el conjunto completo de pines (*pines de suministro de potencia*) para el 74379. La palabra reservada **external** indica que la información completa del 74379 no está contenida en nuestra descripción, pero es encontrada en cualquier lugar. Solo nos interesa asegurar que las terminales en el 74379 que no fueron incluidas en el segundo nivel de descripción son ahora convenientemente tomadas en cuenta.

En la línea 3, el hecho que estemos concentrados con el dispositivo 74379 es indicado y al dispositivo le ha sido dado el nombre de **temp**.

Para apreciar el significado de las líneas 5 a la 8 referiremos A la Figura 4.7b. La caja externa en la figura representa el segundo nivel de descripción (*p. ej.*, M74379). La caja interna representa al dispositivo que usamos para crear el M74379. Este dispositivo (el D74379) tiene las conexiones **in, out, clock y load** en común con el M74379, la única diferencia es que la operación de carga será ejecutada usando el pin de habilitación en el dispositivo real 74379. El hecho de que la habilitación sea activo bajo será tomado en cuenta cuando diseñemos el controlador. La Figura 4.7b indica también que las salidas complementarias el D74379 no son requeridas por el M74379.

```
1     modulo M74379(in<0:3>), out<0:3>, load, clock);
2           modulo D74379(in<0:3>, out<0:3>, outcomp<0:3>, load, clock); external;
3           component D74379 temp;
4           begin
5                 temp_in ← in;
6                 out ← temp_out;
7                 temp_clock ← clock;
8                 temp_load ← load;
9           end {M74379}
10    $
```

Figura 4.7a Descripción RTL del dispositivo 74379

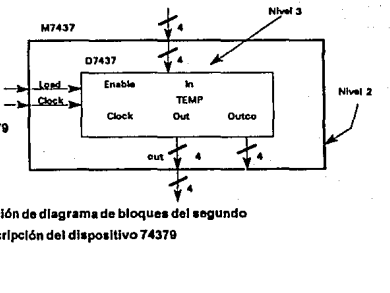


Figura 4.7b Representación de diagrama de bloques del segundo nivel de descripción del dispositivo 74379

- **Finalización de la especificación para el Registro contador**

La especificación final para el registro **contador** es mostrada en la Figura 7.8a. Por razones que son explicadas más, un inversor es requerido en la realización del M74379. Hemos elegido obtener este inversor de un circuito integrado 7404, cuyo conjunto completo de pines de salida son listados en la figura 4.8a. El significado de la línea 1 a la 6 no debe requerir explicación adicional. De las líneas 7 a la 8 los detalles de las conexiones que son comunes a M74169 y D74169 (*ver* Figura 4.8b). Note que la línea 9 indica que una señal desde el controlador decrementa al contador que está a el pin enableP. La hoja de datos del 74169 muestra que esta señal debe ser un bajo y que tendrá el efecto deseado mientras que el enableT y los pines Up/Down del dispositivo están también fijados en bajo; esto es tomado en cuenta en las líneas 13 y 14.

```
1     modulo M74169(in<0:3>, out<0:3>, load, decrement, iszero, clock);
2           modulo D74169(in<0:3>, out<0:3>, load, decrement, clock, updown, enableT, enableP, ripplecarryout);
            external;
3           modulo D7404(in1, out1, in2, out2, in3, out3, in4, out4, in5, out5, in6, out6); external;
4           component D74169 temp;
5           component D7404 inverter;
6           begin
7                 temp_in ← in;
8                 temp_load ← load;
9                 temp_enableP ← decrement;
10                temp_clock ← clock;
11                inverter_in1 ← temp_ripplecarryout;
12                iszero ← inverter_out1;
13                temp_enableT ← 0;
14                temp_updouwn ← 0
15          end {M74169}
16    $
```

Figura 4.8a Descripción RTL en el dispositivo 74169

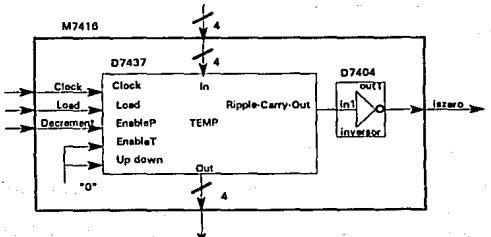


Figura 4.8b Representación de diagrama de bloques del segundo nivel de descripción del dispositivo 74169

Un inversor es requerido entre la terminal **"ripplecarryout"** del D74169 y el **iszero** en el M74169 (*referimos a la Figura 4.8b*). La razón para esto es que el 74169 produce un 0 en el ripplecarryout cuando el contenido del contador alcanza cero. Volviendo al primer nivel de descripción **RTL** (Figura 4.3) vemos en la línea 33 de la descripción

on *counter* = 0 **do begin**

que las acciones a ser ejecutadas, una vez que el contador alcanza 0, dependen de que la señal counter = 0 esté en **"on"**. a lo largo de nuestra descripción **RTL** hemos asumido que on corresponde a positivo, de modo que se requiere un 1 desde M74169 para indicar que el contador ha alcanzado cero.

En las líneas 7 a la 12 de la descripción se muestra la interconexión de las diferentes señales de control y el inversor. Las líneas 14 y 14 especifican dos puntos en el D74169 que deben ser permanentemente conectadas a 0.

- **Finalización de la especificación para el dispositivo Add**

La Figura 7.9a detalla la especificación final para el dispositivo Add. Debe ser claro que la medida extra tomada para este dispositivo en este nivel es para inicializar la terminal **carryin** a 0 (*ver* Figura 4.9b).

```
1    modulo M7483 (IN1<0:3>, IN2<0:3>, OUT<0:3>, carryout);
2         modulo D7483 (in1<0:3>, in2<0:3>, out<0:3>, carryin, carryout) external;
3         component D7483    plus;
4         begin
5              plus_in1 ←  in1;
6              plus_in2 ←  in2;
7              out ←  plus_out;
8              caryout ←  plus_carryout;
9              plus_carryin ←  0
10        end {M7483}
11   S
```

Figura 4.9a. Descripción del RTL para el dispositivo 7483

- **Finalización de la especificación para el registro OVR**

La especificación final para el registro **OVR** es mostrada en la Figura 4. 10a. Esta especificación parece muy compleja, especialmente cuando recordamos que la intensión original fue utilizar un simple flip-flop de delay (*el 74-f*) para el registro **OVR**. La razón para la complejidad radica en el hecho de que los flip-flops simples no tienen pines de carga y sin la carga exacta, esto puede llevarnos a problemas de temporizado.

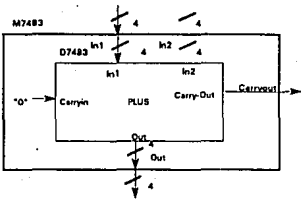


Figura 4.9b. Diagrama de bloques para la representación del segundo nivel de descripción del dispositivo 7483

```
1   modulo M7474(in, out, clear, load, clock);
2       modulo D7474(in1, out1, outcomp1, clock1, clear1, preset1, in2, out2, outcomp2, clock2, clear2, preset2);
            external;
3       modulo D7402(in1a, in1b, out1, in2a, in2b, out2, in3a, in3b, out3, in4a, in4b, out4); external;
4       modulo D7404(in1, out1, in2, out2, in3, out3, in4, out4, in5, out5, in6, out6); external;
5       modulo D7410(in1a, in1b, in1c, out1, in2a, in2b, in2c, out2, in3a, in3b, in3c, out3); external;
6       component D7474 flipflop;
7       component D7402 orgate;
8       component D7404 inverter;
9       component D7410 andgate;
10      begin
11          inverter_in1 ← clear;
12          andgate_in1a ← inverter_out1;
13          andgate_in1b ← load;
14          andgate_in1c ← in;
15          orgate_in1a ← clear;
16          orgate_in1b ← load;
17          flipflop_in1 ← orgate_out1;
18          inverter_in2 ← clock;
19          flipflop_clock1 ← inverter_out2;
20          andgate_in2a ← flipflop_out1;
21          andgate_in2b ← clock;
22          andgate_in2c ← 1;
23          flipflop_in2 ← flipflop_out1;
24          flipflop_clock2 ← andgate_out2;
25          out ← flipflop_out2;
26          clear1 ← 1;
27          clear2 ← 1;
28          preset1 ← 1;
29          preset2 ←1
30      end {M7474}
31  $
```

Figura 4.10a. Descripción del RTL para el dispositivo 7474

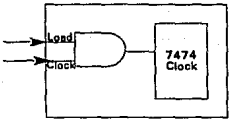


Figura 4.10b Posible implementación para la carga del 7474.

Recordando que el diseño está basado en la idea de que en un flanco de subida del reloj (*llamado flanco de subida i*) el controlador producirá señales de control para la implementación de transferencia de datos entre los registros en el tiempo de flanco de subida **i + 1**. Estas señales de control llegan a los registros después de la llegada del flanco de subida **i + 1**, pero ellas no tienen efecto en los contenidos de los registros hasta que el flanco del reloj llega para sincronizar las entradas como carga o habilitación.

Ya que un flip-flop ordinario no tiene ni carga ni entradas de habilitación, requerimos algún medio para instruir el 7474 para "carga" en el siguiente flanco de subida del reloj. Se puede pensar que el simple arreglo mostrado en la Figura 4.10b debe proporcionar una solución (*para activar en alto las señales de carga*).

La señal de carga llegará a la compuerta **AND** después de algún delay siguiendo el flanco de subida del reloj y cuando este llega, la señal de reloj aun estará en alto, causando un flanco de subida en la entrada del reloj del 7474. Sin embargo, en este tiempo, el 7474 puede no tener el dato correcto establecido en su entrada y así una configuración alternativa (*inevitablemente más compleja*) debe ser buscada.

Un arreglo del circuito apropiado es mostrado en la Figura 7.10c. Estas configuraciones similares son frecuentemente usadas en circuitos integrados complejos para prevenir errores en el registro de los registros cuando una señal de habilitación cambia.

Note que con esta configuración estaremos realizando el registro **OVR** empleando ambos flip-flops D en el paquete 7474.

Considerando primero el circuito asociado con el flip-flop numero2 (*llamado FF_2*) en la Figura 4.10c. este es el flip-flop que genera la salida para el M7474 y así necesitaremos verificar que este cargará y limpiará correctamente. Hay que notar que si la entrada clear es 1, la entrada a la compuerta **AND**, de tres entradas, tendrá un 0 a la salida y esto realmente limpiará FF_2 en el siguiente flanco de subida del reloj. Sí, por otro lado, la entrada **load** es un 1 (*y asumimos que en esta circunstancia clear será un 0*) entoncés la entrada a FF_2 será el valor de entrada **in** y este será transferido a la salida en el siguiente flanco de subida del reloj.

Para verificar que el circuito operará correctamente, todo lo que requerimos es garantizar que el FF_2 está registrando en el tiempo correcto. En el circuito de la Figura 4.10c esto se ha logrado empleando la idea de un reloj de dos fases. El FF_1 está registrado con la señal inversa del reloj y por consiguiente responde en el flanco de bajada del pulso. Para el tiempo cuando el flanco de bajada ocurre, el controlador tendrá establecidos los estados requeridos de modo que, la señal **clear** y **load** estarán en sus niveles correctos. Si ninguno está en nivel 1, la entrada al FF_1 será 1 y, después del flanco de bajada de reloj, la salida irá a 1. Esto asegura que el FF_2 estará registrando en el siguiente flanco de subida, por causa del sentido la salida de FF_1 es "**gated**" con la señal de reloj.

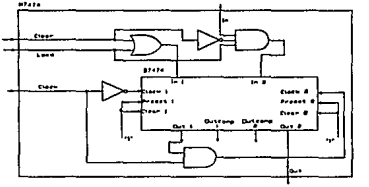


Figura 4.10c. Representación en diagrama de bloque en el segundo nivel de descripció del dispositivo 7474

El lector ahora ya no deberá tener problema en interpretar la descripción **RTL** en la Figura 4.10a. Para las compuertas **AND** y **OR** se han elegido circuitos integrados estándares, con los inversores disponibles en el paquete 7404 ya empleados en la realizacaión de el registro **counter**.

- **Diseño del Shifter (para los registros B y C)**

La Tabla 4.1 lista las funciones requeridas de los registros **B** y **C**. Ambos registros requieren ser capaces de cargar 4 bits, sus contenidos deben ser accesibles para lectura en paralelo, y tiene que ser posible correr sus contenidos un bit a la derecha. En adición, también se debe poder limpiar el contenido del registro **C**. Obviamente como los dos registros son muy similares en función y por economía, diseñaremos un dispositivo simple que puede ser usado por ambos.

Existe un registro de corrimiento que puede ser utilizado para implementar las funciones de los registros **B** y **C** (*el 74195*), pero como intentamos diseñar bajo los principios de los **RTL** basados en el diseño top down y como el diseño de los dispositivos capaces de implementar las funciones requeridas de los registros **B** y **C** proporcionan una conveniente ilustración no utilizremos el circuito integrado antes mencionado.

Como un punto principal, notemos que la operación de corrimiento a al derecha puede ser especificada empleando nuestro **RTL** (*en términos de un registro* **X** *de 4 bits*) como

X<1:3><- X<0:2>
X<0><- shifting

La primera sentecia indica que las entradas 1 a la 3 en **X** son remplazadas por las entradas 0 a 2 y la segunda sentencia indica que la entrada 0 en **X** esta recibiendo el **bit de corrimiento.**

Las dos sentencias indican que la operación de corrimiento a la derecha puede ser ejecutada dentro del registro **X** para lectura de los tres bits X<0:2> y estos retroalimentando a las terminales de entrada X<0:3>. Sí, en adición, el bit de corrimiento es presentado a la terminal de entrada X<0>, entonces la implementación de la operación de carga deberá completar la tarea. Esto se encuentra ilustrado en la Figura 4.11.

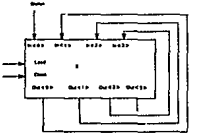


Figura 4.11 Método para implementación del corrimiento a la derecha

Notemos que en el **Paso1** de la Figura 4.3, el registro **B** requiere ser cargado con el multiplicador y en el 2 el registro **C** es cargado con la salida de el sumador. De acuerdo al primero y segundo nivel de descripción **RTL** estas son solo operaciones de carga ejecutados por dos registros. En el **Paso3** los registros **B** y **C** requieren realizar una operación de corrimiento; hemos visto que este puede ser implementado a través de retroalimentación y una operación de carga. Con el empleo de esta implementación, el conjunto completo de operaciones a ser ejecutado por los registros **B** y **C** puede ser completado por la estructura mostrada en la Figura 4.12.

La idea detrás de la Figura 4.12 nace del hecho que los registros **B** y **C** requieren ejecutar operaciones de carga desde dos diferentes fuentes en pasos distintos en el proceso de multiplicación. Uno es la operación de carga especificada en el primero y segundo nivel de descripción **RTL** y la otra es la operación de carga que hemos introducido para ejecurar la operación de corrimiento a la derecha. Un multiplexor puede ser utilizado para seleccionar la fuente correcta para las dos operaciones. La señal **"shiftright"** puede ser empleada como la señal selectora para el multiplexor, porque, cuando este es declarado, el multiplexor seleccionara la entrada 1 y, con la señal **shiftright** también conectada a la terminal **load** del registro **X** (*a través de la compuerta* **OR**), el registro cargara desde la fuente apropiada.

Hemos asumido que el controlador nunca declarará las señales **shiftright** y **load** en el mismo tiempo así que, cuando la señal **load "external"** es declarada en la Figura 4.12, la señal **shiftright** es bajo y el dato

externo estará alimentando al registro **X** a través de la entrada 0 de el multiplexor. El requerimiento para limpiar la entrada (*recordando que el registro C tiene que ser limpiado en el* **Paso 1**) puede también ser manejado empleando el multiplexor. Esto puede ser logrado ya que la mayoría de los multiplexores tienen una entrada de habilitación que, cuando es alimentada con un 1 lógico, causa que todas las salidas del multiplexor tomen un valor de 0 independientemente de la señal selectora o datos de entrada. Para la conexión de la señal **clear** a la entrada de habilitación en el multiplexor, el registro **X** será inicializado cuando la señal clear tome el valor de 1. El controlador está, por supuesto, asumiendo declarar una y solo una de las entradas a la compuerta **OR** de 3 entradas en algún tiempo dado. El problema de garantizar que las señales de control apropiadas son disponibles constituye la parte de control del problema y será discutido después de finalizar la parte de datos.

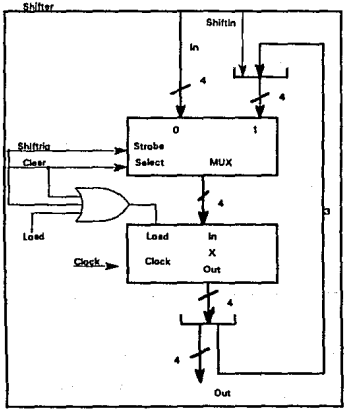


Figura 4.12 Diagrama de bloque para el dispositivo del corrimiento

- **Selección de dispositivos**

La Figura 4.12 indica que los dispositivos que hemos seleccionado para el registro **Shifter** son un multiplexor y el registro **X**.

Para el registro **X**, el 74379 (*seleccionado primero para el registro* **A**) tiene las características requeridas. Como al principio, no requerimos de las salidas complementadas del dispositivo, podremos

emplearlas como en la Figura 4.7. Para el multiplexor, se requiere un **MUX** de 2a1 líneas y el 74157 es un dispositivo apropiado. En esta aplicación, todos los pines de entrada y salida del 74157 son requeridas de modo que hemos declarado este como un D74157 (*en la línea 3 de la* Figura 4.13a).

La selección del 74379 para el registro **X** nos forza a modificar levemente el diseño del shifter descrito en la Figura 4.12. Esta modificación es necesaria porque el pin de carga en el 74379 es activado por un 0 lógico y el arreglo en la Figura 4.12 asume que un 1 lógico es requerido para iniciar la operación de carga. Esta situación puede ser modificada simplemente remplazando la compuerta **OR** en la Figura 4.12 por una conpuerta **NOR**. El arreglo resultante es mostrado en la Figura 4.13b la cual indica que un circuito integrado 7427 proporciona la compuerta **NOR**. Este es mostrado más explícitamente en la descripción **RTL** de la Figura 4.13a, donde la compuerta **NOR** es declarada en la línea 4 como el dispositivo D7427. Este circuito integrado contiene tres compuertas **NOR** de tres entradas y solo una de ellas será utilizada aquí.

En la línea 10 de la FIgura 4.13a el símbolo @, representa la concatenación (*y utilizada primero en la* Figura 4.3) ha sido empleada.

Empero, el registro Shifter ha sido designado para implementar los registros **B** y **C**. Para estos dos registros, no hemos especificado todavía la conexión requerida para el **pin clear**. Antes de ira un nivel adicional en nuestra descripción **RTL** a fin de especificar esta conexión (*como se debe hacer estrictamente*) simplemente señalaremos que el **pin clear** en el registro **C** estará conectado al controlador (*de modo que* **C** *pueda ser limpiado en el* **Paso1** *del algoritmo*) y el **pin clear** en el registro **B** será conectado a 0 lógico (*porque no es requerimiento limpiar* **B**).

La parte de datos de las especificaciones ha sido concluida. Todos los dispositivos necesarios han sido seleccionados y el esquema de interconexión ha sido especificado. Ahora es el turno de diseñar la parte de control.

```
1     modulo Shifter(in<0:3>, out<0:3>, load, shiftright, shiftin, clear, clock);
2           modulo D74379(in<0:3>,out<0:3>,outcomp<0:3>, load, clock); external;
3           modulo D74157(in0<0:3>,in1<0:3>,out<0:3>,select, strobe); external;
4           modulo D7427(in1a, in1b, in1c, out1, in2a, in2b, in2c, out2, in3a, in3b, in3c, out3); external;
5           component D74379 X;
6           component D74157 MUX;
7           component D7427 gate1;
8           begin
9                 MUX_in0 ← in;
10                MUX_in1 ← shiftin @ X<0:2>;
11                X_in ← MUX_out;
12                MUX_select ← shiftright;
13                MUX_strobe ← clear;
14                out ← X_ out;
15                X_load ← gate_out;
16                gate_in1a ← shiftright;
17                gate_in1b ← clear;
```

```
18              gate_inlc ← load;
19              X_clock ← clock;
20          end (Shifter)
21      S
```

Figura 4.13a Descripción del dispositivo Shifter

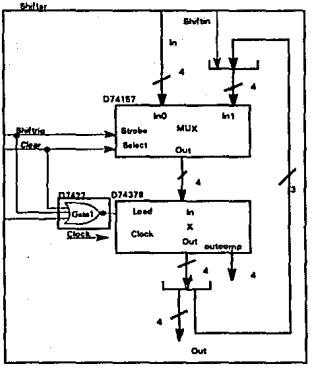


Figura 4.13b Diagrama de bloques del segundo nivel de descripción del dispositivo shifter

IV.1.4 DISEÑO DEL CONTROLADOR

El primer nivel de descripción del multiplicador (Figura 4.3) fue esbozada en la suposición que el algoritmo de multiplicación debería envolver cuatro distintos pasos. Los pasos adicionales no fueron introducidos en niveles más bajos de la descripción, así inferimos que el controlador debe tener cuatro distintos estados. Comenzaremos el diseño del controlador esbozándolo a través de la descripción de una tabla de estados.

El segundo nivel de descripción **RTL** (Figura 4.6) contiene toda la información que necesitamos para declarar la tabla de estados para el controlador (*los niveles más bajos de descripción son concernientes con el desarrollo interno de dispositivos individuales*). Como un primer paso, identificaremos todas las entradas al controlador (a parte de la del reloj). Estas entradas están dadas en la Figura 4.6 por las expresiones que aparecen entre las palabras reservadas **on** y **do**. Así, la Figura 4.6 nos dice que el controlador requiere las entradas **Start**, **B_out** y **counter_iszero** (*en las líneas 16, 27 y 43 respectivamente*). Esta información

nos permite llenar las cuatro primeras columnas de la tabla de estados mostrada en la Tabla 4.2.

Las salidas del controlador son determinadas para identificar las señales de control requeridas desde el controlador hacia los diferentes dispositivos en la parte de datos del multiplicador. Todas estas señales de control aparecen en la Figura 4.6 siguiendo a la palabra reservada **activate.** De este modo el controlador requiere tener 11 distintas salidas y estas están representadas en la tabla de estados por las 11 columnas (*lado derecho*) etiquetadas desde **counter_load** hasta **DONE** en la Tabla 4.2. Las entradas en estas columnas son determinadas por el hecho que en tres casos, el controlador requiere salir a 0 a fin de inicializar una acción particular, estos casos son **counter_load, counter_decrement** y **A_load.** En los otros ocho casos, la señal requerida es un 1.

Estado presente	Start	B_out<3>	counter_hzero	Estado siguiente	counter_load	A_load	B_load	C_load	OVR_load	C_clear	OVR_clear	B_shiftright	C_shiftright	counter_decrement	DONE
Paso1	0	*	*	Paso1	1	1	0	0	0	0	0	0	0	1	0
Paso1	1	*	*	Paso2	0	0	1	0	0	1	1	0	0	1	0
Paso2	*	0	*	Paso3	1	1	0	0	0	0	0	0	0	1	0
Paso2	*	1	*	Paso3	1	1	0	1	1	0	0	0	0	1	0
Paso3	*	*	*	Paso4	1	1	0	0	0	0	0	1	1	0	0
Paso4	*	*	0	Paso2	1	1	0	0	0	0	1	0	0	1	0
Paso4	*	*	1	Paso1	1	1	0	0	0	0	1	0	0	1	1

Tabla 4.2 Tabla de Estados del controlador

La única tarea restante es fraguar la tabla de estados para especificar el estado siguiente. Esto se logra examinando de las líneas 15 a la 48 en la Figura 4.6 y para cada estado en turno, notando la etiqueta que aparece después de la palabra reservada **goto** para cada una de las etiquetas de estado está entrando en la columna de estado siguiente de la tabla de estados en el renglón correspondiente para las condiciones de estrada apropiadas. Así, en la Tabla 4.2, el primer renglón indica que cuando el controlador está en el estado **Paso1** y cuando **Start** es 0, el estado siguiente será **Paso1** (*note que las otras dos entradas no tienen que afectar en esto*). Esta información es obtenida de las líneas 16, 23 y 24 de la Figura 4.6. Similarmente las líneas 16 a 22 contienen la información que necesitamos para completar el renglón 2 de la Tabla 4.2; estas líneas nos dicen que cuando el controlador está en **Paso1** y **Start** es 1, el stado siguiente es **Paso2.** Los renglones restantes de la tabla de estados son completados en una forma similar.

Varias de las columnas de salida en la Tabla 4.2 son idénticas, esto implica que el controlador proporciona señales idénticas a diferentes dispositivos en la parte de datos del sistema. *P. ej.*, el controlador

requiere proporcionar señales de salida idénticas a **B_load** y **C_clear**. Obviamente no hay necesidad de proporcionar una salida separada del controlador para cada uno de estas y la tabla de estado puede ser simplificada fusionando las dos columnas **B_load** y **C_clear** para formar una solo columna que llamaremos **Z<0>** (*ver* Tabla 4.3). Ahora hay que notar que las columnas **counter_load** y **A_load** tienen entradas idénticas y que estas son simplemente los complementos de las entradas requeridas para **B_load** y **C_clear**. Consecuentemente, las señales de control requeridas para **counter_load** y **A_load** pueden ser obtenidas de la salida del controlador **Z<0>** para uso de un inversor.

De tal modo que vemos una sola salida del controlador, que hemos llamado **Z<0>**, puede ser utilizada para proporcionar las señales de control de los cuatro dispositivos en la parte de datos del problema. La entrada del dispositivo **C_load y OVR_load** puede ser alimentada de la misma salida del controlador que llamaremos **Z<1>**. Y similarmente la entrada al dispositivo **B_shiftright** y **C_shiftright** pueden ser alimentadas de una sola salida del controlador que hemos llamado **Z<3>**. El **Counter_decrement** puede ser obtenida invirtiendo **Z<3>**.

Tomando todos estos factores dentro del informe, obtenemos una tabla de estados simplificada que tiene solo cinco columnas de salida, como se muestra en la tabla 4.3. Esta tabla también indica una asignación de estados (*elegida arbitrariamente*) que ha sido realizada para los cuatro estados del controlador.

	Estado presente	Start	B_out<3>	counter_lszero	Estado siguiente	Z<0>	Z<1>	Z<2>	Z<3>	Z<4>
A	00	0	*	*	00	0	0	0	0	0
B	00	1	*	*	01	1	0	1	0	0
C	01	*	0	*	11	0	0	0	0	0
D	01	*	1	*	11	0	1	0	0	0
E	11	*	*	*	10	0	0	0	1	0
F	10	*	*	0	01	0	0	1	0	0
G	10	*	*	1	00	0	0	1	0	1

Tabla 4.3 Tabla de estados reducida para el controlador

Donde la correspondencia entre las nuevas salidas y las salidas previas está dada por:

Z<0> B_load, C_clear
Z<0>' counter_load, A_load
Z<1> C_load, OVR_load
Z<2> OVR_clear
Z<3> B_shiftright, C_shiftright
Z<3>' counter_decrement
Z<4> DONE

y los estados codificados están dados por:

Paso1 00
Paso2 01
Paso3 11
Paso4 10

- **Implementación del controlador**

Dentro del Diseño Digital existen diferentes formas de implementar un controlador, de entre estas muchas posibilidades hemos elegido implementarlo con una **ROM** (*Memoria de solo lectrura*), dado que con esta elección no tenemos que preocuparnos de la asignación de estados óptima porque las diferentes asignaciones no afectaran el tamaño de nuetra memoria. Es por esta razón que en la Tabla 4.3 la asignación de estados fue realizada arbitrariamente.

La Figura 4.14 muestra un diagrama de bloques para el controlador realizándolo con una **ROM**. El hecho de que el controlador tenga cuatro estados implica que requeriremos dos flip-flops para almacenar la información. Estos flip-flops conforman lo comunmente llamado **registro de secuencia de estado** (*SSR*) y es indicado en la Figura 4.14 junto con la **ROM** que requiere de 5 entradas y 7 salidas. Es importante mencionar que las salidas Z<0> a la Z<4> de la Tabla 4.3 son asignadas a los pines de salida desde **out**<*0*> hasta **out**<*4*> del controlador en la Figura 4.14, con *Z<0>* 'y *Z<3>* 'asignados a los pines de salida del controlador **out**<*5*> y **out**<*6*> respectivamente. Hay que notar también que el SSR es controlado por una señal externa de reloj, especificada en la línea 1 de la Figura 4.3.

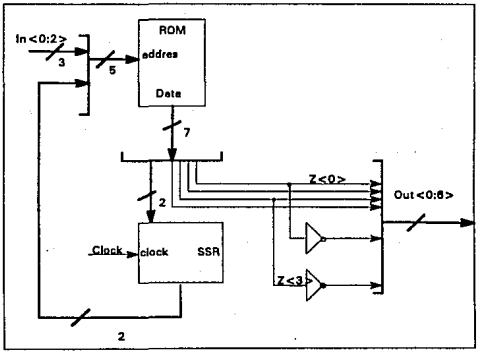


Figura 4.14 Diagrama de bloques del controlador

```
1   module controler(in<0:2>,out<0:6>,clock);
2       module ssr(in<0:1>,out<0:1>,clock);
3           module D7474(in1,out1,outcomp1 clock1,clear1,preset1,in2,out2,outcomp2,clock2,clear2,preset2);
                external;
4           component D7474 temp;
5           begin
6               temp_in1 @ temp_in2 ← in;
7               out ← temp_out1 @temp_out2;
8               temp_clock1 ← clock;
9               temp_clock2 ← clock;
10              temp_clear1 ← 1;
11              temp_clear2 ← 1;
12              temp_preset1 ← 1;
13              temp_preset2 ← 1;
14          end; {ssr}
15      module M74S188(in<0:4>,out<0:6>);
16          module D74S188(in<0:4>,out<0:7>,enable);external;
17          component D74S188 fsm;
18          begin
19              fsm_in ← in;
20              out ← fsm_out<0:6>;
21              fsm_enable ← 0
22          end; {M74S188}
23      module D7404(in1,out1,in2,out2,in3,out3,in4,out4,in5,out5,in6,out6);external;
24      component ssr SSR;
25      component M74S188 PROM;
26      component D7404 inverter;
27      begin
28          PROM_in<0:2> ← in
29          PROM_in<3:4> ← SSR_out;
30          SSR_in ← PROM_out<5:6>;
31          SSR_clock ← clock;
32          out<0:4> ← PROM_out<0:4>;
33          inverter_in1 ← out<0>;
34          inverter_in2 ← out<3>;
35          out<5> ← inverter_out1;
36          out<6> ← inverter_out2;
37      end {controlador}
38  $
```

Figura 4.15a Descripción RTL del controlador

La siguiente tarea es seleccionar los dispositivos que conforman al controlador, **ROM** y **SSR**. La **ROM** que requerimos debe tener 5 entradas y 7 salidas (**ROM** *de 32 x 7*). Por inspección de las hojas de especificaciones determinamos que una **ROM TTL** 74S188 (*32 x 8*) de 256 bits será apropiada para la tarea. La salida extra en esta **ROM** no será utilizada. El **SSR** puede ser realizado empleando un flip-flop D dual 7474. También requerimos un 7404 para proporcionar los dos inversores necesarios para realizar **Z**<0>' y **Z**<3>'. La descripción **RTL** de estos tres dispositivos son dados en la Figura 4.15a.

	Estado presente	Start	B_out<3>	counter_isZero	Estado siguiente	Z<0>	Z<1>	Z<2>	Z<3>	Z<4>
A	00	0	*	*	00	0	0	0	0	0
B	00	1	*	*	01	1	0	1	0	0
C	01	*	0	*	11	0	0	0	0	0
D	01	*	1	*	11	0	1	0	0	0
E	11	*	*	*	10	0	0	0	1	0
F	10	*	*	0	01	0	0	1	0	0
G	10	*	*	1	00	0	0	1	0	1

Figura 4.15b. Contenidos de la ROM para la implementación del controlador

De las líneas 2 a la 14 se describe la implementación de **SSR**. En la línea 2 las entradas y las salidas del módulo **SSR** son descritas en términos de un vector de 2 bits, antes que en términos de dos bits separados de información. La relación entre las entradas del módulo **SSR** y las dos terminales de entrada del paquete 7474 es entoncés expresada en la línea 6 en una sola sentencia de transferencia

temp_in1 @ temp_in2 ← in;

La variable temp es el nombre dado al componente D7474 en la línea 4. El símbolo @ es el operador de concatenación y la sentencia de transferencia sobre los estados que **temp_in1** y **temp_in2** están siendo tratados como una terminal de 2 bits con las siguientes conexiones individuales:

temp_in1 ← in<0>;
temp_in2 ← in<1>;

La línea 7 de la Figura 4.15a tiene un significado similar, indica que **temp_out1** y **temp_out2** son conectadas, respectivamente, a **out<0>** y **out<1>** del módulo **SSR**. La línea 8 y 9 especifican que la entrada del reloj a el **SSR** está conectada a cada una de las entradas de reloj del paquete 7474, los dos flip-flops comparten la misma fuente de reloj. De la línea 10 a la 13 se especifican las conexiones necesarias para los **pines de clear** y **preset** del 7474. Estos pines no juegan ningún papel en la operación del controlador, y estableciendo estos en alto (*de modo que salgan como sin conexión o "floating"*) evitaremos posibles problemas a causa de la captación de ruido.

Las conexiones de la **ROM** son especificadas de la línea 15 a la 22 de la Figura 4.15a. La línea 20 expresa el hecho que solo siete de la salidas de la **ROM** serán usada la octava estará sin conexión.

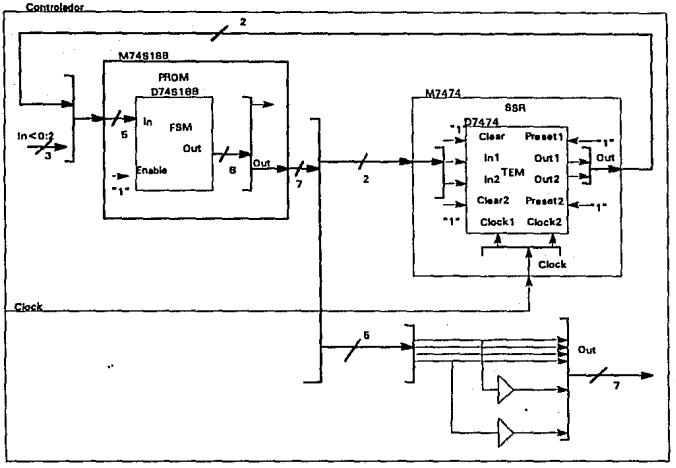


Figura 4.15c Diagrama de bloques del segundo nivel de descripción para el controlador

En las líneas 28 a la 36 se detallan las conexiones requeridas entre las entradas y salidas del controlador y la **ROM** y entre la **ROM** y el **SSR**. Las líneas 33 a la 36 detallan las conexiones de out<0> y out<3> (*correspondientes a* Z<0> *y* Z<3> *en la* Tabla 4.3) a los inversores para controlar las señales out<5> y out<6> respectivamente (*correspondientes a* Z<0>' *y* Z<3>' *en la* Tabla 4.3)

La Figura 4.15b. muestra los contenidos de la **ROM** que son derivados directamente de la tabla de estados dada en la Tabla 4.3. La columna encabezada como "**ROW**" ha sido incluida por facilidad de comparación- esta indica el renglón en la tabla de estados para cada dirección y datos de entrada correspondiente.

El diseño del controlador está concluido ahora. La Figura 4.15c es un diagrama de bloques del controlador. La Figura 4.16 proporciona la descripción completa **RTL** del multiplicador, detallando todos los datos y conexiones de control entre los módulos que hemos definido en párrafos anteriores.

```
1   module Multiplier(Multiplier<0:3>,Multiplicand<0:3>,
2                     Product<0:7>, Start, Done, clock);
3       module M74175(in<0:3>,out<0:3>,load,clock);
4           external;
5       module Shifter(in<0:3>,out<0:3>,load,shiftright,
6                      shiftin,clear,clock);
7           external;
8       module M7474(in, out, load, clear, clock);
9           external;
10      module M74169(in<0:3>, load, decrement, iszero, clock);
11          external;
12      module M7483(in1<0:3>, in2<0:3>,out<0:3>, carryout);
13          external;
14      module controller(in<0:2>, out<0:6>, clock);
15          external;
16      component M74175  A;
17      component Shifter  B, C;
18      component M7474  OVR;
19      component M74169  counter;
20      component M7483  Adder;
21      component controller Control;
22      begin
23          Product ← C_out @ B_out;
24           A_in ← Multiplicand;
25          B_in ← Multiplier;
26          C_in ← Adder_out;
27          Adder_in1 ← C_out;
28          Adder_in2 ← A_out;
29          OVR_in ← Adder_carryout;
30          B_shiftin ← C_out<3>;
31          C_shiftin ← OVR_out;
32          counter_in ←  4 % 4;
33          Control_in<0> ← Start;
34          Control_in<1> ← B_out<3>;
35          Control_in<2>  ←  counter_iszero;
36          A_load ← Control_out<5>;
37          B_load ← Control_out<0>;
38          C_load ← Control_out<1>;
39          counter_load ← Control_out<5>;
40          C_clear ← Control_out<0>;
41          OVR_load ← Control_out<1>;
42          OVR_clear ← Control_out<2>;
43          B_shiftright ← Control_out<3>;
44          C_shiftright ← Control_out<3>;
45          counter_decrement ← Control_out<6>;
46          Done ← Control_out<4>;
47          B_clear ← 0;
48          A_clock ← clock;
49          B_clock ← clock;
50          C_clock ← clock;
51          OVR_clock ← clock;
52          counter_clock ← clock;
53          Control_clock ← clock
54end {multiplicador}
55$
```

Figura 4.16. Descripción final RTL del multiplicador

CONCLUSIONES

El arte de diseñar no solo se sustenta en la imaginación y habilidad para generar ideas, sino también en las herramientas de diseño con las cuales poder realizar esta idea. En el mundo de hoy, el diseñador no puede, ni debe, rechazar la idea del concepto de **Diseño Asistido por Computadora** si él desea generar un producto de alta calidad en todos los aspectos.

En el campo que nos compete, el **Diseño Digital**, existe una gran variedad de dichas herramientas orientadas a la simulación de circuitos, al diseño mismo del sistema y aun a la construcción del prototipo. La escencia del **Diseño Conceptual para Sistemas Digitales** nace de la fusión de estas áreas y se ve materializada en el surgimiento de los **Lenguajes de Descripción de Hardware**.

Al inicio de esta investigación sólo sabiamos que estos conceptos existian, pero no teniamos la idea de lo que esto representa. Para poder comprender y asimilar dichos conceptos, que son fundamentales en el desarrollo de este trabajo, tuvimos que transportarnos a la escencia misma de las herramientas antes mencionadas: objetivo, presentación y sobretodo entender la filosofía bajo la cual han sido implementadas. esto nos llevó a las entrañas del **Diseño Digital**, aprendimos que el diseñar es una tarea compleja de diferentes fases.

El objetivo principal fue implementar una herramienta que nos permitiera ver el hecho de diseño como una disciplina sistemática, esto en un principio podría parecer una idea descabellada si pensamos en el hecho de que una idea no es materializada solo por un diseñador sino por un conjunto de ellos, pero no es así, ya que no estamos planteando una metodología de diseño sino una técnica de verificación de mismo. Para cumplir con este objetivo partimos de la definición de lo que son los **Lenguajes de Transferencia de Registro** pero no era suficiente con esto, dado que estos están ligados a un concepto más abstracto, el **Nivel de Transferencia de Registro**, que es solo uno de los diferentes niveles de diseño dentro de los muchos que existen en esta área. La asimilación de lo que representan los niveles de diseño surgió de la comparación del software existente ya que la gran mayoría de estos sistemas o herramientas combinan dichos niveles; estos compilan la información de un nivel de diseño para obtener un nivel más bajo.

El software con el que comenzamos el análisis, para entender lo que son en general los niveles de diseño fue **OrCAD**, que es un sistema que apartir de un diseño esquemático nos permite crear el layout a través de una interface hacia otro software llamado **Tango**. Otro sistema que trabaja bajo una filosofía de compilación de niveles es **EMCI** (*Editor de Mascarillas de Circuitos Integrados*), este trabaja con gráficas interactivas que permiten la modificación de layout, tomando en cuenta las reglas geométricas más comunes. Un sistema que nos permite clarificar el concepto del uso de los **HDLs** como cálculo en el diseño es el llamado **MATLAB**, que es un software interactivo de alto rendimiento para análisis numérico, procesamiento de señales y gráficas en un ambiente sencillo.

Con toda esta información delimitamos los objetivos y líneas de dirección para la determinar la estructura del lenguaje y su compilador. Las características del compilador encudran todas las características del **Nivel de Transferencia de Registro** y obtiene como salida un código que nos permitirá realizar las descripciones y simulación del circuito en cualquier otro nivel de diseño.

La entrada al compilador es la información referente a un diseño digital codificada con la estructura del **Lenguaje de Transferencia de Registros (RTL)**, después de compilar dicha descripción obtendremos dos archivos que nos dan la información necesaria para hacer las correcciones pertinentes al diseño, en el caso de detectar errores y mensajes de advertencia, así como listar todas las conexiones declaradas en la descripción, marcando las longitudes asignadas y los usos definidos en el módulo para cada terminal. También podremos producir la tabla de descripción de estados del controlador para estos módulos, utilizando los reportes generados.

Una aplicación inmediata de éste compilador es el uso y estudio del mismo en alguna de las materias relacionadas con **Compiladores** o con **Diseño de Sistemas Digitales** con la finalidad de realizar pruebas constantes y optimizar su funcionamiento.

Una ventaja del Lenguaje de Transferencia de Registros es su gran parecido al Lenguaje Pascal con el que los estudiantes de la **Facultad de Ingeniería** estamos familiarizados.

Cabe mencionar que el compilador puede ser complementado con un **Sistema de Simulación** el cual bien podría ser tema para otra tesis, o bien general la interfase apropiada para los simuladores disponibles en el mercado como **Workbrench, TopSpice, PSpice.**

"Invertir en Conocimientos
Produce Siempre los Mejores
Intereses"

Benjamin Franklin
Estadista y Científico
[1706-1790]

El desarrollo del **RTL** fue motivado por dos razones principales. La primera fue hacer disponible un lenguaje apropiado para enseñar el diseño de hardware como una disciplina sistemática basada en ciertos conceptos fundamentales, como se mencionó al principio de este trabajo, clara y naturalmente reflejados en el lenguaje, La segunda fue desarrollar un lenguaje que no requiriera el uso constante de un manual, aun si el lenguaje no es empleado frecuentemente.

- **Notación, Terminología y Vocabulario**

De acuerdo a la forma tradicional *Backs-Naur*, las construcciones sintácticas se encuentran denotadas por palabras encerradas entre paréntesis de ángulo <>. Estas palabras describen el significado de la construcción. La posible repetición de una construcción es indicada por llaves {}. El símbolo <*vacio*> denota la secuencia nula de símbolos.

El vocabulario básico del **RTL** consiste de símbolos clasificados en **letras, dígitos, y símbolos especiales.**

<letras> : := A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digitos> : := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<simbolos especiales> : := @ | & | <- | : | ; | , | $ | (|) | % | < | > | _ | **modulo component | terminal | edo | begin | end | goto | activate | on | do**

La construcción

/* <cualquier secuencia de símbolos que no contengan "*/" >*/

es un comentario y puede ser cambiado en el texto sin alterar su significado.

- **Identificadores y Números**

Los **identificadores** sirven para identificar tipos y variables. Su asociación debe ser única dentro de su alcance de validez, ***i.e.***, dentro del módulo en el que ellos son declarados.

<identificador> : := *<letra>* { *<letra>* | *<digito>* }
<<letra> o *<digito>>* : := *<letra>* | *<digito>*

Los **números** pueden ser notación **hexadecimal, octal, binaria o decimal.**
<número> : := *<número* hexa*decimal>* | *<número decimal>* | *<número octal>* | *<número binario>*

<número hexadecimal> : := **0H** *<cadena* hexadecimal> | **0h** *<cadena* hexadecimal>

<cadena hexadecimal> : = *<dígito hexadecimal>* { *<dígito hexadecimal>* }

<dígito hexadecimal> : := A | B | C | D | E | F | a | b | c | d | e | f *<dígito>*

<número decimal> : := *<dígito>* { *<dígito>* }>

<número octal> : := **0O** *<cadena octal>* | **0o** *<cadena octal>*

<cadena octal> : := *<dígito octal>* { *<dígito octal>* }

<dígito octal> : := 0 | 1 | 2 | 3 | 4 | 5 | 6 |7

<número binario> : := **0B** *<cadena binaria>* | **ob** *<cadena binaria>*

<cadena *binaria>* : := *<dígito binario>* { *<dígito binario>* }

<dígito binario> : := 0 | 1

- **Definiciones de Tipo**

Existen dos tipos de variables simples en la descripción del **RTL: terminal y estado.** Una declaración de tipo determina las operaciones que pueden ser ejecutadas en una variable de ese tipo, y le asocia un identificador.

- **Tipo terminal**

El tipo terminal define un conjunto de puntos de conexión locales a un módulo.

<terminal> : := **terminal** *<lista de puertos>*;

<lista de puertos> : := *<variable de puerto>*, { *<variable puerto>* }

<variable puerto>: := *<identificador>* { **<RANGO>** }

<rango> : := *<<número>* : *<número>>*

- **Tipo estado**

Un tipo estado define un **conjunto de identificadores** asociados con el **estado** de una máquina secuencial. Estas variables deben, en una etapa más adelante de diseño, ser convertidas a valor binarios que pueden ser cargados en un registro de secuencia de estados en el controlador, o implementado vía algún otro esquema (*este esta determinado por el diseñador*).

<declaración de estado> : := **edo** <lista de *identificador>*

<lista de identificador> : := *<identificador>* { , *<identificador>* }

- **Tipos estructurados**

Un tipo estructurado es caracterizado por el tipo(s) de sus partes elementales y por su método estructurado. Este **RTL** solo soporta un conjunto limitado de primitivas estructuradas: **arreglos** de terminales, y módulos.

- **Tipos arreglos**

Un tipo **arreglo** es aplicable solo a tipos terminales, y consiste de un **número fijo de terminales**. Los elementos del arreglo son designados por índices (*que deben ser números constantes*). Solo los arreglos de una dimensión son soportados en el **RTL**. Ver *<lista de puertos>* en la declaración de tipo terminal para la sintaxis de la declaración.

Ej.

a<0:7>, ros<15:0>

- **Tipo Módulo**

Una definición de módulo define el conjunto de terminales (*o puertos de conexión*) a través del cuál un tipo particular de componente puede comunicar, así como definir la operación interna de un componente de ese tipo. Asocia un identificador con el tipo del módulo.

<módulo> : := **modulo** *<identificador>*(*<lista de puerto>*) *<cuerpo del módulo>*

<cuerpo del módulo> : := **external** | *<block>*

- **Declaración de terminales y componentes**

Las declaraciones de terminales consisten de una **lista de identificadores**, algunos de los cuales pueden tener dimensiones de arreglo.

<declaración de terminal> : := **terminal***<lista de puerto>*;

La declaración de componentes consiste de un identificador de tipo módulo seguido de una lista de identificadores denotando los nuevos componentes.

<declaración de componente> : := **component** *<cuerpo del componente>*;

<cuerpo del componente> : := *<componente>* { *<componente>* }

<componente> : := *<identificador de módulo>* *<identificador>* { *<identificador>* }

<identificador de módulo> : := *<identificador>*

- **Denotaciones de variable**

Las denotaciones de variables como designar una variable terminal o una variable terminal de puerto.

<variable> : := *<variable terminal>* | *<variable terminal de puerto>*

- **Variable terminal**

Una variable terminal es especificada por su identificador o por su identificador y un rango, especificando un subconjunto de las terminales del arreglo. Los índices del subrango deben estar dentro de los límites declarados en la definición del identificador terminal.

<variable terminal> : := *<identificador>* { *<especificación del subrango>* }

<especificación de subrango> : := *<<número>* { *<número>*} >

- **Variable terminal de Puerto**

Una variable terminal de puerto es especificada por una variable módulo seguida de una variable terminal. El identificador del terminal debe corresponder con los puertos definidos por el tipo módulo.

<variable terminal de puerto> : := *<identificador de módulo>_<variable terminal>*

- **Expresiones**

Las expresiones son construcciones que denotan reglas de combinación para terminales. Las expresiones consisten de operandos (*i.e.*, constantes y variables) y operadores. Las reglas de composición especifican la precedencia de los operadores de acuerdo a los tres operadores empleados. El **operador not** tiene la más alta precedencia, le sigue el **operador and** y finalmente el **operador or.**

<factor> : := *<variable>* { *<op. de concatenación>* } | *<número>* | (*<expresión>*) | ~ *<factor>*

<op. de concatenación> : := @

<término> : := *<factor>* | *<término>* *<op. and>* *<factor>*

<op. and> : := &

<expresión> : := *<término>* | *<expresión>* *<op. or>* *<término>*

<po. or> : := |

Ej.

factor x, 0Hffe9, ~x

término x & y, x<1:3> & y<5:7>

expresiones in1 | in3, subtract_out<3:5> | out<0:2>

- **Operadores**

Los operandos de los operadores and y or deben referirse a variables de la misma longitud, *i.e.*, si cualquiera de los dos operandos referencia un arreglo terminal unidimensional, entonces el número de bits especificados en cada caso será el mismo. Si los dos operandos son de cantidades de n bits, entonces el resultado producido por el operador es también una de n bits.

Ej.

terminal a, b, x<0:15>, y<1:16>;
modulo check(in<7:0>, out<7:0>); **external**;

- **Instrucciones**

Las instrucciones denotan las interconexiones del hardware. Todas las instrucciones son "ejecutadas" en paralelo. Las instrucciones pueden ser prefijadas por una atiqueta que puede ser referenciada por una instrucción goto.

<instrucción> : := *<instrucción no etiquetada>* | *<identificador de estado>*

<identificador de estado> : := *<identificador>*

<etiqueta no etiquetada> : := *<instrucción de transferencia>* | *<instrucción goto>* |
<instrucción activate> | *<instrucción on>* | *<instrucción compuesta>* |
<vacia>

- **Instrucción de transferencia**

Una instrucción de transferencia denota la conexión del hardware especificado en el lado derecho de la esxpresión a las terminales especificadas en el lado izquierdo. El número de bits especificado por el lado derecho e izquierdo debe ser el mismo, o el del lado derecho debe ser una expresión de un solo bit.

<instrucción de trasnferencia> : := *<variable>* { @ *<variable>* } <- *<expresión>*

Ej.

terminal a<0:7>, b<0:7>, x, y, z;
x <- y|z
c <- a &b
a<0:1>@c<5:6><-b<3:6>

- **Instrucción goto**

La instrucción goto indica que el controlador está cambiando de estado, el nuevo estado está definido por la etiqueta del estado.

<instrucción goto> : := **goto** *<etiqueta del estado>*

<etiqueta del estado> : := *<identificador>*

Las siguientes restricciones conciernen a la aplicabilidad de las etiquetas de estado:

1. El alcance de la etiqueta de estado es el módulo dentro del cual ste es definido. No es posible transferir el control directamente dentro de otro módulo vía una instrucción **goto**.

2. Cada etiqueta debe ser especificada en una declaración de estado en el encabezado del módulo en el cual la etiqueta es empleada.

3. Si las etiquetas de estado son definidas en un módulo habrá más de una etiqueta definida, esto es, los controladores deben tener más de un estado. Si un controlador tiene solo un estado, este se puede realizar con lógica combinacional.

- **Instrucción Activate**

La instrucción **activate** especifica que el controlador está declarando la señal especificada por la variable cada vez que las condiciones del estado son satisfechas (*i.e* cuando el controlado está en el estado especificado y las condiciones del estado son satisfechas).

<instrucción activate> : := **activate** *<variable>*

La señal debe referir a una terminal de un solo bit, y debe ser habilitado por un *<identificador de estado>*.

- **Instrucción On**

La instrucción **on** especifica que una instrucción debe ser ejecutada solo en la condición de que la expresión sea verdadera. Si es falsa, la instrucción no será ejecutada, o la instrucción seguira al símbolo **else** para ser ejecutada.

<instrucción on> : := **on** *<expresión>* **do** *<instrucción>* | **on** *<expresión>* **do** *<instrucción>* **else** *,instrucción>*

Las expresiones entre los símbolos **on** y **do** deben ser una expresión de un solo bit.

- **Instrucción compuesta**

La **instrucción compuesta** especifica que las instrucciónes de las cuales esta constituida son tratadas como si ellas fueran una instrucción simple, con cada instrucción ejecutada concurrentemente.

<instrucción compuesta> : := **begin** *<instrucción>* {; *<instrucción>* } **end**

- **Declaración de módulo**

La declaración de módulo sirve para definir nuevos tipos de dispositivos de hardware y asociarles identificadores de modo que puedan ser empleados como instancia a componentes.

<declaración de módulo> : := *<encabezado del módulo>* *<cuerpo del módulo>*

<cuerpo del módulo> : := **external** | *<block>*

<block> : := *<parte de declaración del módulo>* *<parte de declaración de componente>*
<parte de declaración de terminal> *<parte de declaración de estado>*
<parte de instrucción>

El **encabezado de módulo** puede contener una descripción de la implementación del módulo, o puede contener la palabra reservada external. La aparición de la palabra reservada **external** especifica que los detalles de la implementación del módulo (o especificación) aparecen en cualquier lugar.

La parte de **declaración de módulo** contiene todos las definiciones de tipo módulo local al módulo.

<parte de declaración de módulo> : := *<vacia>* | *<declaración de módulo>*

La parte de **declaración de componente** es una instancia a uno (o más) dispositivos de hardware del tipo dado por el identificador de módulo, y asocia un nombre con cada instancia. Estos componentes son locales al módulo.

<parte de declaración de componente> : := *<vacio>* | *<declaración de componentes>*

La parte de **descripción de terminal** contiene todas las declaraciones del terminal local al módulo de declaración.

<parte de declaración de terminal> : := *<vacio>* | *<declaración del terminal>*

La parte de **declaración de estado** contiene todos las declaraciones del estado local a la declaración módulo.

<parte de declaración de estado> : := *<vacio>* | *<declaración de estado>*

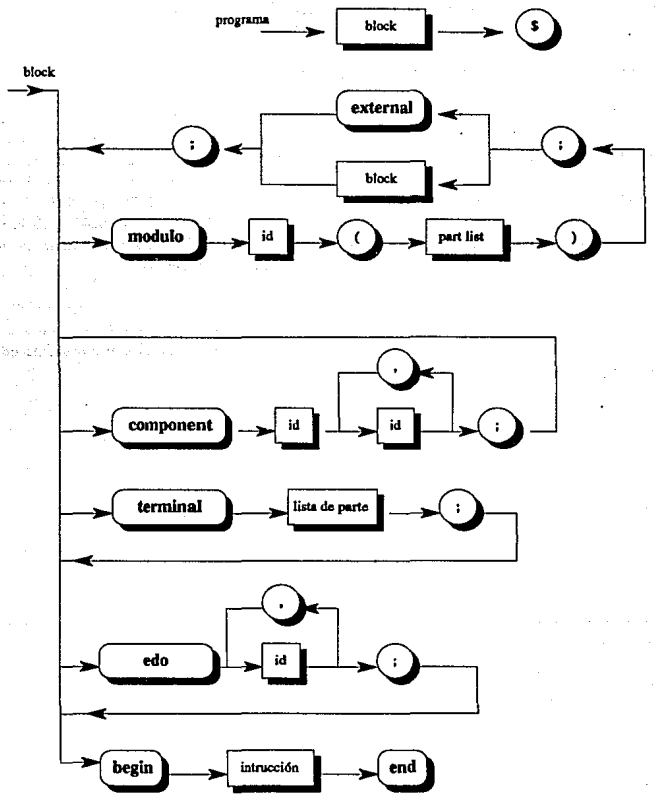
La parte de instrucción especifica las conexiones que son realizadas para implementar el algoritmo de hardware deseado.

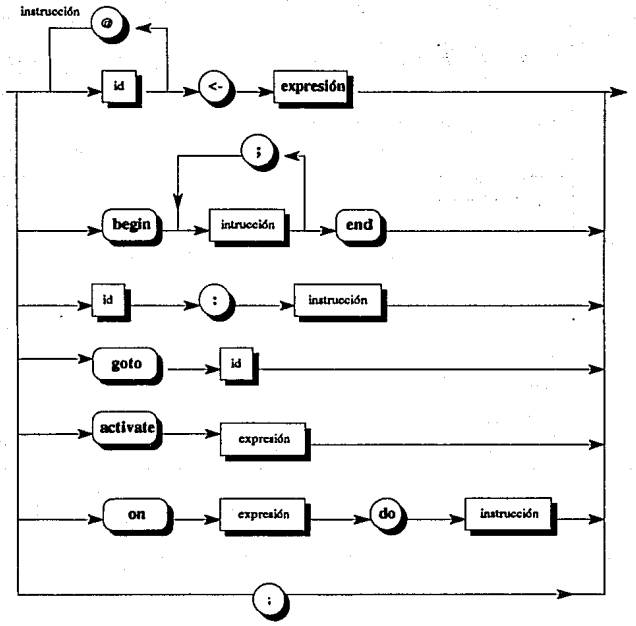
<parte de instrucción> : := <instrucción compuesta>

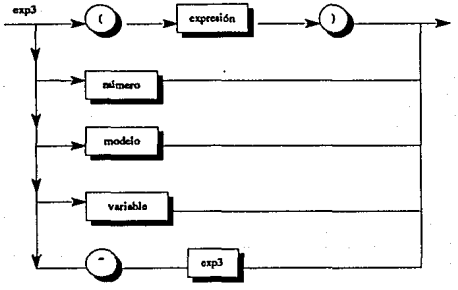
Todos los identificadores introducidos en la parte de declaración del módulo, en la parte de declaración de componente, en la parte de declaración de terminal y en la parte de declaración de estado son locales a la declaración del módulo que define el alcance de estos identificadores. No pueden salir fuera de su alcance. El identificador introducido en el encabezado del módulo es globalmente definido. La lista de puertos introducidos en el encabezado del módulo introducen identificadores que son definidos global y localmente, ellos pueden ser referenciados localmente por su solo nombre, o pueden ser referenciados globalmente cuando precede al nombre del componente seguido por un símbolo. Los identificadores que aparecen en la definición del módulo anidado más profundo no son visibles fuera de su definición.

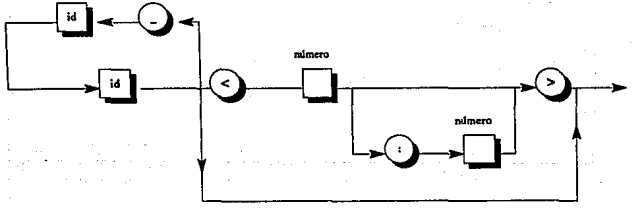
- **Representación gráfica de la Sintaxis**

Una descripción alternativa para el RTL es dado en el diagramas de sintaxis desglosado en la Figura 4.A.1.


programa
block
$
block
external
;
;
block
modulo
id
(
part list
)
component
id
,
id
;
terminal
lista de parte
;
edo
,
id
;
begin
intrucción
end


instrucción
@
id
<-
expresión
;
begin
intrucción
end
id
:
instrucción
goto
id
activate
expresión
on
expresión
do
instrucción
;


exp3
(
expresión
)
número
modelo
variable
-
exp3


id
_
id
<
número
>
:
número

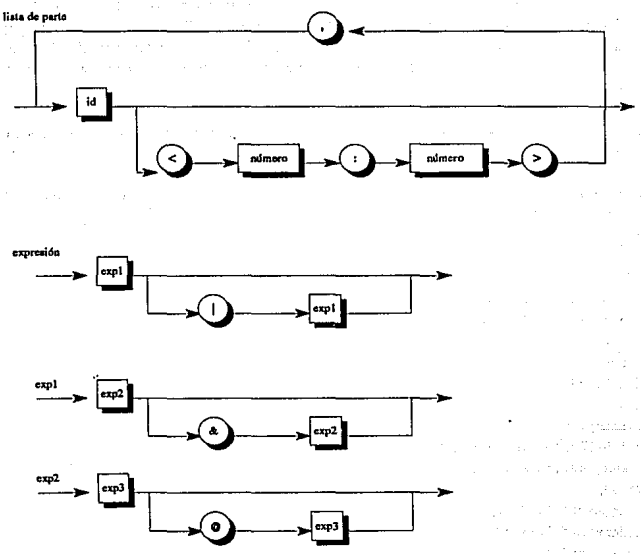


Figura A.1 Diagrama de sintaxis RTL

En este apendice se demuestra el uso de nuestro compilador, introduciendo las opciones que ofrece. Asumimos que el archivo de entrada está asociado a un texto almacenado con el nombre de **PROGRAMA.TXT**, un listado resultante se obtendrá en el archivo **LISTADO** en el que se despliegan los mensajes de error si es que los hay y en el lugar donde los identifica el compilador, el # de errores detectados, el # de advertencias, el # de nodos sin recuperar, el total de nodos asignados, y el # de nodos utilizados.

Por otra parte se genera un archivo llamado **CONEXION** en el cual se describe el uso de cada pin conectado en el diseño.

Para ejemplificar la utilización del compilador emplearemos la descripción de un mecanismo Shifter el cual ha sido diseñado y codificado de acuerdo a las gramáticas especificas del **RTL** y que se muestra a continuación:

```
modulo Shifter(in<0:3>, out<0:3>, load, shiftright,
          shiftin, clear, clock);
modulo D74379(in<0:3>, out<0:3>, outcomp<0:3>,
load, clock);
     external;
 modulo D74157(in0<0:3>, in1<0:3>, out<0:3>,
 select, strobe);
     external;
 modulo D7427(in1a, in1b, in1c, out1, in2a, in2b,
        in2c, out2, in3a, in3b, in3c, out3);
    external;
     component D74379 X;
     component D74157 MUX;
     component D7427  gate1;
     begin
          MUX_in0 <- in;
          MUX_in1 <- shiftin @ X_out<0:2>;
          X_in <- MUX_out;
          MUX_select <- shiftright;
          MUX_strobe <- clear;
          out <- X_out;
          X_load <- gate1_out1;
          gate1_in1a <- shiftright;
          gate1_in1b <- clear;
          gate1_in1c <- load;
          X_clock <- clock;
     end {Shifter}
$
```

Si generamos una corrida utilizando el archivo anterior como entrada, obtenemos la siguiente salida en el archivo **LISTADO**:

```
1  {$I+}
2  modulo Shifter(in<0:3>, out<0:3>, load, shiftright,
3              shiftin, clear, clock);
4  modulo D74379(in<0:3>, out<0:3>, outcomp<0:3>,
5  load, clock);
6        external;
7  modulo D74157(in0<0:3>, in1<0:3>, out<0:3>,
8  select, strobe);
9        external;
10 modulo D7427(in1a, in1b, in1c, out1, in2a, in2b,
11           in2c, out2, in3a, in3b, in3c, out3);
12
13       external;
14
15       component D74379 X;
16       component D74157 MUX;
17       component D7427  gate1;
18
19       begin
20            MUX_in0 <- in;
21            MUX_in1 <- shiftin @ X_out<0:2>;
22            X_in <- MUX_out;
23            MUX_select <- shiftright;
24            MUX_strobe <- clear;
25            out <- X_out;
26            X_load <- gate1_out1;
27            gate1_in1a <- shiftright;
28            gate1_in1b <- clear;
29            gate1_in1c <- load;
30            X_clock <- clock;
31       end {Shifter}
32  $

w- X_outcomp <0:3>No es usado ni por Input ni por Output
w- gate1_in2a     No es usado ni por Input ni por Output
w- gate1_in2b     No es usado ni por Input ni por Output
w- gate1_in2c     No es usado ni por Input ni por Output
w- gate1_out2     No es usado ni por Input ni por Output
```

w- gatel_in3a No es usado ni por Input ni por Output
w- gatel_in3b No es usado ni por Input ni por Output
w- gatel_in3c No es usado ni por Input ni por Output
w- gatel_out3 No es usado ni por Input ni por Output

Número de errores detectados : 0
Número de advertencias : 9
Número de nodos sin recuperar : 2
Número de nodos asignados : 48
Número de nodos usados : 82

Al final del **LISTADO** generado en ésta corrida se escriben nueve líneas con mensajes de advertencia (*indicados por "w" al inicio de la línea, los mensajes de error se inican con "E"*).

La primera línea indica que la salida complementada del 74379 no tiene que ser utilizada en la descripción.

El compilador advierte al usuario de todas las terminales que estén sin conectar, en este caso la intención del usuario es dejar la salida desconectada. Las siguientes ocho líneas de la misma salida advierten al usuario que dos de las compuertas **NOR** del integrado 7427 nó son usadas como se pretendía.

A la hora de una corrida podemos activar ó desactivar la desplegado de mensajes de error y advertencias empleando los comentarios de cada opción del compilador.

El archivo **CONEXION** de la corrida anterior se presenta a continuación:

```
{$Modulo      :Shifter      }
{$Modulo      :D74379       }
{$Fin del Modulo:D74379     }
{$Modulo      :D74157       }
{$Fin del Modulo:D74157     }
{$Modulo      :D7427        }
{$Fin del Modulo:D7427      }
{terminalX_in          <0:3>Uso = [Salida]}
X_in          <0:3><-MUX_out      <0:3>
{terminalX_out         <0:3>Uso = [Entrada]}
{terminalX_outcomp     <0:3>Uso = [**SIN USO**]}
{terminalX_load        Uso = [Salida]}
X_load        <-gatel_out1
{terminalX_clock       Uso = [Salida]}
```

```
X_clock      <-clock
{terminalMUX_in0      <0:3>Uso = [Salida]}
MUX_in0      <0:3><-in          <0:3>
{terminalMUX_in1      <0:3>Uso = [Salida]}
MUX_in1      <0:3><-TEMP000
{terminalMUX_out      <0:3>Uso = [Entrada]}
{terminalMUX_select  Uso = [Salida]}
MUX_select    <-shiftright
{terminalMUX_strobe  Uso = [Salida]}
MUX_strobe    <-clear
{terminalgatel_in1a    Uso = [Salida]}
gatel_in1a    <-shiftright
{terminalgatel_in1b    Uso = [Salida]}
gatel_in1b    <-clear
{terminalgatel_in1c    Uso = [Salida]}
gatel_in1c    <-load
{terminalgatel_out1    Uso = [Entrada]}
{terminalgatel_in2a    Uso=[**SIN USO**]}
{terminalgatel_in2b    Uso=[**SIN USO**]}
{terminalgatel_in2c    Uso=[**SIN USO**]}
{terminalgatel_out2    Uso=[**SIN USO**]}
{terminalgatel_in3a    Uso=[**SIN USO**]}
{terminalgatel_in3b    Uso=[**SIN USO**]}
{terminalgatel_in3c    Uso=[**SIN USO**]}
{terminalgatel_out3    Uso=[**SIN USO**]}
{terminalTEMP000    Uso = [EntradaSalida]}
TEMP000      <-shiftin
{$Tabla de Estados          }
{$Fin de la Tabla de Estados    }
{terminalin              <0:3>Uso = [Entrada]}
{terminalout             <0:3>Uso = [Salida]}
out          <0:3><-X_out        <0:3>
{terminalload            Uso = [Entrada]}
{terminalshiftright      Uso = [Entrada]}
{terminalshiftin         Uso = [Entrada]}
{terminalclear           Uso = [Entrada]}
{terminalclock           Uso = [Entrada]}
{$Fin del Modulo:Shifter       }
```

En este archivo se debe observar el gran número de líneas contenidas entre los delimitadores "{}". Estas pueden ser tratadas como comentarios y sirven para :

1.- Listar todas las terminales en la descripción, denotando los tamaños declarados y los usos con los que se definió en el módulo tales como: Entrada, Salida, Ambas ó ninguna. **[**SIN USO**]**.

2.- Generar la tabla de descripción de estados del controlador para estos módulos, los cuales son definidos por medio del uso de declaraciones de estado, etiquetas de estado y declaraciones **"goto"**.

3.- Preservar algunas de las estructuras de la descripción original presentando el inicio y final de cada declaración de módulo en la descripción.

La 1a. y la última líneas de la descripción de estados, son la definición del módulo Shifter, y son descritos de la manera siguiente **{$Modulo: Shifter}** y **{$Fin del módulo: Shifter}**. Las líneas 2 - 7 indican la presencia de los módulos D74379, D74157, y D7427. Estos módulos han sido declarados como **EXTERNOS** y además no vienen detallados. Las líneas siguientes de texto relatan las conexiones terminales para realizar el módulo Shifter.

Las declaraciones Terminal aparecen como comentarios de la manera siguiente:

{Terminal X_in<0:3> Uso = [Salida]}

Esta línea nos dice que existe una terminal X_in que es de cuatro bits de tamaño, y que éstos son numerados del 0 al 3 (MSB a LSB) y que esto aparece en el lado izq. de la declaración e implica que los datos han sido transferidos a este y que esto puede ser utilizado como una salida.

X_in<0:3><-mux_out<0:3>

Esta línea es una colección de todas las declaraciones de conexión en las cuales **X_in** ocurre en el lado izquierdo. En este caso solo hay una declaracion y ésta expresa la acción de conectar la **MUX_out <0:3>** al **X_in <0:3>**. La acción del **MUX_out <0:3>** ocurre en el lado derecho de una declaración de conexión e implica que el acto es una entrada *(los datos tienen que ser transferidos desde el MUX_out <0:3> y después regresar al listado)*. Esta nó es una declaración de conexión siguiendo un fin de comentario debido a que la conexión del **MUX_out<0:3>** al **X_in <0:3>** ha sido indicada. Sin embargo para evitar la duplicación de éste ordenamiento, las declaraciones de conexión son listadas solo para terminales de salida. Algunas terminales no son ni entradas ni salidas y son marcadas como **[**SIN USO**]**.

Las componentes terminales y las conexiones a terminales son listadas hasta la línea de comentario **{$Tabla de Estados}**.

En el caso general la línea de comentario {*$Tabla de Estados*} puede estar seguida de información concerniente a los estados de transición pero el módulo Shifter solo tiene un estado tal que los estados de transición nó se aplican en este caso de aquí que la línea {*$Tabla de Estados*} este seguida por {*$Fin de la Tabla de Estados*}. Estas dos líneas pueden parecer redundantes pero ellas pueden facilitar el trabajo de interpretación de algunos programas creados para acarrear hacia afuera los procesos siguientes en la información presentada en el **LISTADO**.

Todas las líneas de comentarios terminales concluyen con el comentario {*$Tabla de Estados*} referida a terminales asosiadas con las conexiones a terminales en los componentes del módulo.

OPCIONES DEL COMPILADOR:

Opción de Listado:

Permite o nó generar un listado completo del archivo de entrada, se puede activar o desactivar con los comentarios {$l+} ó {$l-} respectivamente en la primera línea del archivo de entrada.

Opción de Advertencias:

Permite o nó generar mensajes de advertencia en el **LISTADO**, activando ó desactivando respectivamente con los comentarios {$w+} ó {$w-} en la primera línea del archivo de entrada.

Opción de Estado:

Cuando un diseño es completado; es decir, el nivel más bajo de descripción **RTL** de un sistema ha sido obtenido, la descripción puede no contener referencias explícitas de los estados del sistema. Esto es, no deben referenciarse tipos de estados variables por ejemplo declaraciones de estado o declaraciones goto en la descripción final. En los niveles más altos de descripción, el uso de tipos de estado es importante pero su presencia en una descripción indica que los pasos siguientes son necesarios antes de completar un diseño. Los mensajes de error son impresos si los tipos de estado están bajo la especificación del usuario. Los mensajes de estado pueden ser activados ó desactivados con los comentarios {$s+} ó {$s-} respectivamente en la primera línea del archivo de entrada.

Opción de Expresión:

El compilador podrá escoger tipos correctos de expresiones que nó deben estar presentes en la descripcion final. Los tipos de expresiones involucradas son aquellas que incluyen los operadores lógicos

AND (&), OR (|), y **NOT (^)**. Estas operaciones son usadas algunas veces en las descripciones **RTL**. Los mensajes de error son generados si cualquier operador lógico está presente. Estos mensajes pueden evitarse o nó con los comentarios {$e+} ó {$e-} respectivamente, en la primera línea del archivo de entrada.

Opción de Explicación:

La opción de explicación permite al usuario expander declaraciones para clarificar el significado de las mismas. El usuario puede obtener o nó una explicación para un grupo de aclaraciones con los comentarios {$x+} y {$x-} al inicio del archivo de entrada.

Dentro del archivo **LISTADO** las líneas que comienzan con un (*) son producidas por el uso de la opción {$x+} y constituyen la explicación producida desde la línea anterior de la descripción de entrada por el compilador, *p. ej.*, la línea siguiente:

```
a @ b <- c<1:2>;
```

es expandida en dos declaraciones de transferencia una para **a** y otra para **b**:

```
on # s0 do b <- c<2>;
on # s0 d0 a <- c<1>;
```

Las dos declaraciones de transferencia son ahora condicionadas por declaraciones para mostrar que las transferencias son ejecutadas con la condición de que el sistema esté en el estado **s0**. Note también que la terminal **a** (*el bit más significativo de la terminal es concatenada a @ b*) es concatenada a la terminal **c<1>** (*el bit más significativo del set terminal c<:2>*); y b es conectado a la terminal **c<2>**.

Si entre dos declaraciones no existiera un (;) el compilador reconoce que esto es un error común y asume que el (;) esta presente y esto es mostrado con la advertencia:

```
w- ; insertado antes de esta línea.
```

El compilador colecciona condiciones y las junta para formar una sola declaración como la siguiente:

```
on # s1&c<3> do c<0:2> <-(a&x) @ c<3;2>
```

El compilador también genera datos para una tabla de descripción de estados de la operación del controlador.

El compilador reconoce que las declaraciones **goto** nó son condicionadas por ningún valor de entrada terminal, y genera declaraciones separadas como las siguientes:

```
on *AnyInput* do goto #s0
on *AnyInput* do goto #s1
```

Se debe notar que la tabla de descripción de estados aparece en el archivo **CONEXION** entre los comentarios: {*$Tabla de Estados*} y {*$Fin de la tabla de Estados*}.

De las dos siguientes líneas de una tabla de descripción:

```
s0:
on *AnyInput* do goto # s1
```

observamos que el controlador se encuentra en el estado **s0**, el pulso siguiente de reloj puede provocar que el controlador pase a un estado **s1** considerando el valor de cualquiera de las terminales. Similarmente, las siguientes dos líneas de la tabla de descripción de estados, las cuales si el controlador está en **s1** el pulso siguiente de reloj puede provocar que el controlador pase a un estado **s0** considerando el valor de cualquier terminal.

```
program compilad(programa, salida, listado, conexion);

const
        nopres   = 12; {Número de palabras reservadas}
        tabmax   = 50; {Longitud de la tabla de identificadores}
        bitmax   = 32; {Número de bits en un modelo - máquina dependiente}
        longid   = 16; {Longitud de los identificadores}
        errormax = 20; {Número máximo de mensajes de error permitido}

type
        simbolos = (nulo, ident, numero, flechizq, flechder,parenizq, parender,
                                porcent, modelo, onsimb, andsimb, concat, notsimb, gotosimb,
                                actsimb, onsimb, dosimb, puntoycoma, Idef, pesos, coma,
                                dospuntos,Inisimb, xfer, edef, cdef, finsimb, modulosimb,
                                iniconment,fincomment, edarnsimb, elsesimb);

        modulos  = (termdef, edodef, modulo);
        acarreos  = (tipopunto, tipovector);
        setsimb  = set of simbolos;
        palfa    = packed array [1..bitmax] of char;
        nalfa    = string[longid];
        nodop    = ^nodo;
        nodo     = record
                                len  : integer; {Longitud de esta expresión}
                                rptr : nodop  ; {Rama derecha}
                                case ntipo: simbolos of notsimb, andsimb, onsimb, concat,
                                                        xfer, onsimb,puntoycoma, gotosimb,
                                                        actsimb : (lptr:nodop);
                                                        modelo  : (pattn: palfa);
                                                        ident   : (tablap, rega, regf : integer)
                                end;
        pair     = record
                                  izq, der: integer
                                end;
        pairlist  = array[1..bitmax] of pair;
        longlist  = array[1..bitmax] of integer;
        useset    = set of(input,output);
        tblentrada= record
                                nombre  : nalfa;
                                typ     : acarreos;
                                nomlong : integer; {longitud del nombre}
                                reune   : nodop;
                                M       : integer;
                                uso     : useset;
                                case clase : modulos of
                                        modulo  : (nargs, argindex: integer);
                                        termdef, edodef : (regien, freg, treg: integer)
                                end;

var
   caracter   : char;  {Ultimo caracter leido}
   simbolo    : simbolos; {Ultimo símbolo leido}
   identif    : nalfa;  {Ultimo identificador leido}
   num        : integer;{Ultimo número leido}
   patron     : palfa;  {Ultimo patrón leido}
   plen       : integer;{Longitud del último patron leido}
   cc         : integer;{Cuenta los caracteres}
   ll         : integer;{Longitud de la línea}
   kk         : integer;
   linea      : array[1..82] of char;
   palabra    : array[1..nopres] of nalfa;
   wsimb      : array[1..nopres] of simbolos;
   ssimb      : array[char] of simbolos;
   nivel      : integer;{Nivel de anidamiento estático}
   errores    : integer;{Contador de errores}
   lineanum   : integer;{Número de linea fuente}
   blankid    : nalfa;
   expbandera : boolean;{Se activa si las expresiones son permitidas}
   advertencia: integer;{Contador de advertensias}
```

```
  lbandera  : boolean;{Se activa si se ha leido "listado"}
  pbandera  : boolean;{Se activa cuando las advertencias han sido mostradas}
  nodos     : integer;{Número de nodos usados}
  banderaexp : boolean;{Se activa si requiere explicación en listado }
  edobandera : boolean;{Se activa para la descripción de estado de máquina}
  nodosdesoc : integer;{Contador de nodos desocupados}
  nodoscont  : integer;{Contador de nodos en uso}
 termcont   : 0..tabmax; {Número actual de terminal temporal}
 nlibre, plibre, llibre : nodop; {lista de nodos libres}
 listado, conexion, salida, programa  : text;
 arglabla, tabla      : array[0..tabmax] of tblentrada;
 simbolos1, simbolos2, simbolos3 : setsimb;


procedure comperror(n:integer);
{Reporte de error para salida. Imprime en el archivo "listado" si se requiere}

   procedure mensajes(var f:text; err: integer);

   begin {mensajes}
   if not (err in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
         20,21,22,23,24,25,26,27,30,31,32,33,34,35,36,37,39,40,42,44,
         47,48,50]) then
          writeln(f,'error numero:', err:3)
   else
         case err of
                1: writeln(f, 'Esperando símbolo de identificador');
                2: writeln(f, 'Identificador no decalarado');
                3: writeln(f, 'Esperando identificado de tipo módulo');
                4: writeln(f, 'Esperando número');
                5: writeln(f, 'Esperando ; o .');
                6: writeln(f, 'Error en expresión');
                7: writeln(f, 'Esperando declaración');
                8: writeln(f, 'Identificador multidefinido');
                9: writeln(f, 'Identificador no inicializado');
                10: writeln(f, 'Expresión no permitida en descripción estructural');
                11: writeln(f, 'Esperando "begin"');
                12: writeln(f, 'Esperando "end"');
                13: writeln(f, 'Esperando <-');
                14: writeln(f, 'Caracter ilegal');
                15: writeln(f, 'Declaración fuera de orden');
                16: writeln(f, 'Modelo demasiado largo');
                17: writeln(f, 'Indice fuera de rango');
                18: writeln(f, 'Símbolo ilegal');
                19: writeln(f, 'Esperando "$"');
                20: writeln(f, 'Error en parte de la declaración');
                21: writeln(f, 'Sólo el bit terminalseguirá ACTIVATE');
                22: writeln(f, 'Esperando )');
                23: writeln(f, 'Variables usadas con ACTIVATE deben ser de un bit de longitud');
                24: writeln(f, 'Esperando >');
                25: writeln(f, 'Esperando "do"');
                26: writeln(f, 'Error en la terminal de acceso');
                27: writeln(f, 'Declaraciones ACTIVATE no pueden ser incondicionales');
                30: writeln(f, 'Número demasiado largo');
                31: writeln(f, 'Será especificada  una longitud más grande que 1');
                33: writeln(f, 'No es permitido declarar etiquetas en descripción estructural');
                34: writeln(f, 'Lado izquierdo de declaración transfiere incorrectamente');
                35: writeln(f, 'Longitud incompatible de las expresiones');
                36: writeln(f, 'No puedo accesar el módulo de identificadores');
                37: writeln(f, 'No se permite "goto" en descripción estructural ');
                39: writeln(f, 'Expresión condicional debe ser de 1 bit de longitud');
                40: writeln(f, 'Mensaje expresión condicional');
                42: writeln(f, 'Esperando :');
                44: writeln(f, 'Modelo mal formado');
                47: writeln(f, 'Esperando número o modelo único ');
                48: writeln(f, 'Múltiple definición de estado');
                50: writeln(f, 'Declaración "goto" aparece con estado no corriente');
         end;
         end; {mensajes}
```

```
begin {comperror}
  if bandera then
        begin
          writeln(listado, ''; cc+7,'^');
          write(listado, 'E-');
          mensajes(listado,n);
        end;
  write(lineanum: 5, ':');
  mensajes(salida,n);
  errores:= errores + 1;
  if errores > errormax then
        begin
          if bandera then
                    writeln(listado, 'Demasiados errores. Ejecución terminada');
          writeln('Demasiados errores. Ejecución terminada');
          halt;
        end;
end; {comperror}


function dec_bin(n,f,l:integer): boolean;

{Esta función convierte un número decimal n a un modelo bit binario}
{sólo salva los bits f para l en el modelo, regresa verdadero si esto sucede}

var
  i, k: integer;

begin {dec_bin}
  dec_bin := true;
  for i := 1 to bitmax do
        patron[i] := '0';
  if n = 0 then
        plen := 1
  else
        if n = 1 then
              begin
                    plen := 1;
                    patron[bitmax] := '1';
              end
        else
              begin
                if n < 0 then
                    begin
                          if f = 1 then
                                patron[1] := '1';
                          n := n + maxint + 1;
                    end;
                    i := bitmax;
                    k := bitmax;
                    while n  <> 0 do
                          begin
                                if (k >= f) and (k <= l) then
                                      begin
                                            if odd(n) then
                                                  patron[i]:= '1'
                                            else
                                                  patron[i] := '0';
                                            i := i - 1;
                                      end;
                                k := k -1;
                                n := n div 2;
                          end;
                    plen := i - f + 1;
              end;
end; {dec_bin}
```

```
function adecimal(var n: integer; f, l: integer): boolean;

(Convierte un modelo en variable "patron" a un número decimal.)
(Regresa falso si no importa aparece)
(Convierte solo bits 'f' a 'l' de el modelo)

var
        i : integer;
acierto : boolean;

begin (adecimal)
  acierto := true;
 n := 0;
 for i := f to l do
                if patron[i] = '?' then
                  acierto := false;
                if acierto then     (Modelo único - no no importa)
                  for i := l downto f do
                                if n >= 0 then
                                        n := n*2 + (ord(patron[i]) - ord('0'))
                                else
                                        if acierto then
                                          begin
                                                acierto := false;
                                                comperror(30);
                                          end;
                                adecimal := acierto;
end; (adecimal)

procedure traecar;

(Trae un caracter de la entrada. Los caracteres están almacenados
en línea para velocidad de acceso)

begin (traecar)
  reset(programa);
  read(programa,caracter );
 if cc = ll then
        begin
         if Eof(programa) then
                begin
                        if lbandera then
                                writeln(listado, '*** Programa incompleto');
                        writeln('*** Programa incompleto');
                        halt;
                       end;
          ll := 0;
          cc := 0;
          lineanum := lineanum + 1;
          if lbandera then
                write(listado, '', lineanum: 5, '', 3);
          while not Eoln(programa) do
          begin
                ll := ll + 1;
                 read(programa,caracter );
                 if lbandera then
                        write(listado, caracter );
                 if ll <= 81 then
                        linea[ll] := caracter ;
                {writeln(linea[ll]);}
          end;
          if lbandera then
                 writeln(listado);
        end;
       ll := ll + 1;
        read(programa,linea[ll]);
        {write  (linea[ll]);}
       cc := cc + 1;
        caracter:= linea[cc];
```

```
        writeln (caracter );
end; (trasecar)


procedure traesimb;
(Convierte una secuencia de caracteres a simbolos compilados)

var
  i, j, k: integer;
  cero: boolean;

  procedure modela(c : char);

  (Convierte una secuencia de caracteres a un modelo bit. Este manejará)
  (constantes binarias, octales, o hexadecimales. No más que "bitmax"bit )
  (pueden ser almacenados. Esto es MAQUINA DEPENDIENTE)

  var
              range: set of char;
              base, i, inc, bits: integer;
              overflow:  boolean;

  begin (modela)
  simbolo  := modelo;
  plen     := 0;
  cero := false;
  range   :=['0', '1', '?'];
  base    := 2;
  bits    := 1;
  if(c = 'H') or (c = 'h') then
        begin
              range := ['0'..'9','A'..'F', 'a'..'f', '?'];
               bits  := 4;
               base  := 16;
        end
  else
        if(c = 'O') or (c = 'o') then
              begin
                    range := ['0'..'7', '?'];
               bits  := 3;
                    base  := 8;
               end;
  trasecar;
  overflow := false;
  while caracter in range do
  begin
   if not overflow then
        begin
              if (plen + bits -1) > bitmax then (Demasiados bits en el modelo)
                 overflow := true
              else
                begin (Cree espacio para el siguiente modelo)
                if plen > 0 then
                      for i  := (bitmax - plen - bits - 1) to bitmax - bits do
                                  patron[i] := patron[i + bits];
                                  case base of
                                  2:  patron[k] := caracter ;
                                  8:  begin
                                        if caracter  <> '?' then
                                              inc := ord(caracter ) - ord('0');
                                        for i := bitmax downto bitmax-2 do
                                        if caracter  = 'ch' then
                                              patron[i] := caracter
                                        else
                                          begin
                                              patron[i] := chr(inc mod 2 + ord('0'));
                                              inc := inc div 2;
                                          end;
                                  end;
```

```
                        16: begin
                                If caracter <> '?' then
                                    If caracter in ['A'..'F'] then
                                        inc := ord (caracter ) - ord ('A') + 10
                                    else
                                        If caracter in ['a'..'f'] then
                                            inc := ord (caracter ) - ord ('a') + 10
                                        else
                                            inc := ord(caracter ) - ord('0');
                                for i:= bitmax downto bitmax-3 do
                                    If caracter = '?' then
                                        patron[i] := '?'
                                    else
                                     begin
                                        patron[i] := chr(inc mod 2 + ord('0'));
                                        inc := inc div 2;
                                     end;
                                end;
                    end; {case}
                    plen := plen + bits;
                end;
        end;
        end;
        If overflow then
        begin
            comperror(30);
            plen := 0;
        end;
end; {modela}

  procedure opciones;

  {Conjunto de opciones especificadas dentro del archivo de descripción}

  var
        i: integer;
        oldbandera: boolean;

        procedure switch(var bandera: boolean);

        begin {switch}
         trasecar;
         If caracter in ['+', '-'] then
                begin
                    bandera := caracter = '+';
                     trasecar;
                end;
        end; {switch}

  begin {opciones}
        oldbandera := ibandera;
        trasecar;
        while caracter in ['L', 'l', 'X', 'x', 'W', 'w', 'S', 's', 'E', 'e'] do
         begin
                case caracter of
                 'L', 'l' :  switch(ibandera);
                 'X', 'x' :  switch(banderaexp);
                 'W', 'w' :  switch(pbandera);
                 'S', 's' :  switch(edobandera);
                 'E', 'e' :  switch(expbandera);
         end;
        If caracter = ',' then
                trasecar;
         end;
        If banderaexp then
                ibandera := true;
         If ibandera and not oldbandera then
         begin
                write(listado, '', lineanum: 5,':3);
```

```
                for i:= 1 to ll do
                        write(listado,linea[i]);
                    writeln(listado);
            end;
    end; {opciones}

begin {traesimb}
  repeat
  while (caracter  =' ') or (ord(caracter ) = 9) do
  {toma TAB de una manera curiosa}
                traecar;
  If caracter  in ['A'..'Z', 'a'..'z'] then
  begin                 {palabra reservada o identificador}
          k  := 0;
          identif := blankid;
           repeat
          If k< longid then
           begin
                k:= k + 1;
                identif[k] := caracter ;
           writeln  (identif[k]);
          end;
          traecar;
        until not(caracter  in ['A'..'Z', 'a'..'z', '0'..'9','_']);
                plen := k;
                i := 1;
                j := nopres;
        repeat
        k := (i+j) div 2;
          If identif <= palabra[k] then
                j := k - 1;
          If identif >= palabra[k] then
                i := k + 1;
         until i > j;
                If i - 1 > j then
                  simbolo := wsimb[k]
                else
                  simbolo := ident;
        end
        else
          If caracter  in ['0'..'9']  then   {Número o modelo}
           begin
                k := bitmax;
                plen := 0;
                num := 0;
                simbolo := numero;
                ceros := false;
                for i := 1 to bitmax do
                        patron[i] := '0';
                If caracter  = '0' then
                begin
                        traecar;
                        ceros := true;
                  end;
                If caracter  in ['H', 'h', 'O','o', 'B', 'b'] then
                        modelo(caracter )
                else
                        If caracter  in ['0'..'9','D', 'd'] then
                        begin
                                If caracter  in ['D', 'd'] then   {Tipo de modelo}
                                begin
                                        traecar;
                                        ceros := false;
                                  end;
                                If caracter  in ['0'..'9'] then {Número decimal}
                                begin
                                        repeat
                                          If num >= 0 then
                                                num := num * 10 + (ord(caracter ) - ord('0'))
```

```
                                else
                                        comperror(30);
                                traecar;
                              until not (caracter in ['0'..'9']);
                              if cero and (num = 0) then
                                plen := 1
                              else
                                if (num = 1) and not cero then
                                {Unico modelo de todos los 1's asumidos}
                                begin
                                        for i := 1 to bitmax do
                                                patron[i] := '1';
                                        plen := 1;
                                end;
                        end
                else
                        comperror(44);
        end
        else
                if cero then
                        plen := 1
                else
                        comperror(44);
        end
        else
         if caracter = '<' then
                begin
                        traecar;
                        if caracter = '-' then
                                begin
                                        simbolo := xfer;
                                        traecar;
                                end
                        else
                                simbolo := inicoment;
                        end
                      else
                      begin
                        simbolo := ssimb[caracter ];
                                if simbolo = inicoment then
                                begin
                                        traecar;
                                        if caracter = '$' then
                                                opciones;
                                        while ssimb[caracter ] <> fincoment do
                                                traecar;
                                end
                                else
                                if simbolo = nulo then
                                        comperror(14);
                                traecar;
                        end;
  until not(simbolo in [inicoment, nulo]);
end; {traesimb}


procedure sigif(fsy: simbolos; err: integer);

{Salta para el simbolo siguiente si el actual simbolo es aceptado;}
{si no reportará un error}

begin {sigif}
if fsy = simbolo then
  traesimb
else
        comperror(err);
end; {sigif}
```

```
procedure saltos(fsys: setsimb);     (Salta)

begin (saltos)
while not(simbolo in fsys) do
        traesimb;
end; (saltos)

procedure busca(fsys: setsimb; err: integer);
(Espera simbolo de fsys; si no hay error y remplaza en fsys)

begin (busca)
if not(simbolo in fsys) then
  begin
   comperror(err);
   repeat
                traesimb;
  until simbolo in fsys
  end;
end; (busca)

procedure buscasalta(s1, s2: setsimb; n: integer);
(Si el simbolo corriente no está en el conjunto S1 entonces reporta un error)
(entonces salta a algún simbolo dentro de los conjuntos S1 o S2)

begin (buscasalta)
if not(simbolo in s1) then
  begin
  comperror(n);
  saltos(s1 + s2);
  end;
end; (buscasalta)

function parar(fsy: simbolos): boolean;
(Usada para parar sentencia de repetición;
para al simbolo <> fsy sino traesimb y repite )

begin (parar)
if simbolo <> fsy then
  parar := true
else
        begin
        parar := false;
        traesimb;
        end;
end; (parar)

function nuenodo(typ: simbolos; lp, rp: nodop; i: integer): nodop;
var
 x: nodop;

begin (nuenodo)
        if nlibre <> nil then
        begin
                x := nlibre;
                nlibre := x^.rptr;
        end
        else
        begin
        case typ of
                        andsimb:
                                        begin
                                         new(x);
                                        x^.ntipo := andsimb;
                                        end;
                orsimb:
                                        begin
                                                new(x);
                                                x^.ntipo := orsimb ;
                                        end;
```

```
                notsimb:
                        begin
                                new(x);
                                x^.ntipo := notsimb;
                        end;
                xfer:
                        begin
                        new(x);
                                x^.ntipo := xfer;
                        end;
                actsimb:
                        begin
                         new(x);
                                x^.ntipo := actsimb;
                         end;
                gotosimb:
                                begin
                                 new(x);
                                x^.ntipo := gotosimb;
                                end;
                onsimb:
                        begin
                                new(x);
                                x^.ntipo := onsimb;
                                end;
                concat:
                                begin
                                 new(x);
                                x^.ntipo := concat;
                                end;
                puntoycoma:
                                        begin
                                        new(x);
                                        x^.ntipo := puntoycoma;
                                        end;
        end;
        nodosdesoc := nodosdesoc + 1;
  end;
  nodos := nodos + 1;
  nodosconti := nodosconti + 1;
  with x^ do
   begin
  len := 1;
  {ntipo := typ;}
  lptr := lp;
  rptr := rp;
   end;
  nuevnodo := x;
end; {nuevnodo}


function nuesident(tp, sl, fin: integer): nodop;

var
   x: nodop;

begin {nuesident}
if libre <> nil then
   begin
   x := libre;
   libre := x^.rptr;
   end
else
        begin
          new(x);
         x^.ntipo := ident;
          nodosdesoc := nodosdesoc + 1;
          end;
nodosconti := nodosconti + 1;
```

```
nodos  := nodos + 1;
with x^ do
        begin
        len := abs(st - fin) + 1;
        {ntipo := ident;}
        rptr := nil;
        tablap := tp;
        regs := st;
        regf := fin;
        end;
nuesident := x;
end; {fnuesident}


function nuemodelo(l: integer; p : palfa): nodop;
{Asigna el contenido de un nodo a un modelo bit especificado }

var
  x : nodop;

begin {fnuemodelo}
if plibre <> nil then
   begin
        x := plibre;
        plibre := x^.rptr;
        end
else
        begin
          new(x);
          nodosdesoc := nodosdesoc + 1;
          end;
nodos := nodos + 1;
with x^ do
        begin
        ntipo := modelo;
        len := 1;
        pattn := p;
        rptr := nil
        end;
nuemodelo := x
end; {fnuemodelo}


procedure devnodo(var p: nodop);
{Regresa un nodo en la lista libre apropiada para reusarlo despues}

begin {fdevnodo}
if p <> nil then
   begin
        with p^ do
                case ntipo of
                nosimb, andsimb, onsimb, concat, xfer, onsimb, puntoycoma,
                actsimb, gotosimb:
                begin
                rptr := nlibre;
                        nlibre := p
                end;
        modelo:
                        begin
                         rptr := plibre;
                         plibre:= p;
                        end;
        ident:
                begin
                rptr := llibre;
                        llibre := p
                end;
        end;
        p := nil;
```

```
        nodosconf := nodosconf - 1
  end;
end; {dispsenodo}


procedure dapsarbol(var p : nodop);
{Retoma los nodos de una estructura árbol a su lista libre}
begin {dapsarbol}
if p <> nil then
  with p^ do
        if ntipo in [oraimb, concat, xfer, andsimb, onsimb, puntoycoma,
                notsimb, gotosimb, actsimb] then
                begin
                dapsarbol(lptr);
                dapsarbol(rptr);
                end;
        devnodo(p);
end; {dapsarbol}


function copiarbol(p: nodop): nodop;
{Regresa una copia completa de una estructura de árbol}

begin {copiarbol}
copiarbol := nil;
if p <> nil then
        with p^ do
                begin
                case ntipo of
                xfer, andsimb, orsimb, concat, onsimb, puntoycoma, notsimb,
                actsimb, gotosimb :
                        copiarbol := nuenodo(ntipo, copiarbol(lptr), copiarbol(rptr), len);
                ident:  copiarbol := nuesident(tablap, regs, regf);
                modelo: copiarbol := nuemodelo(len, pattn)
                end; {case}
        end;
end; {copiarbol}


procedure impnum(var f: text; n: integer);
{imprime un número en un archivo 'f'...}

var
  i : integer;

begin {impnum}
if n < 10 then
  i := 1
else
        if n < 100 then
          i := 2
        else
                if n < 1000 then
                        i := 3
                else
                        if n < 10000 then
                                i := 4
                        else
                                i := 5;
write(f, n: i);
end; {impnum}


procedure parbol(var f : text; p : nodop);
{Imprime una linea... en una estructura árbol}
{en el archivo 'f'}

var
  i : integer;
```

```
  brackets : boolean;
  range : setaimb;

begin  {parbol}
if p <> nil then
  with p^ do
  case ntipo of
              andsimb, orsimb, concat:
                begin
                range := [ident, modelo, notsimb, concat];
                      if ntipo = orsimb then
                        range := range + [orsimb, andsimb];
                      brackets := not (lptr^.ntipo in range);
                      if brackets then
                        write(f, '(');
                      parbol(f, lptr);
                      if brackets then
                        write(f, ')');
                      if ntipo = concat then
                              write(f, '@')
                      else
                              if ntipo = orsimb then
                                      write(f, '|')
                              else
                                      write(f, '&');
                      if ntipo = orsimb then
                              range := range + [orsimb];
                      brackets := not(rptr^.ntipo in range + [andsimb]);
                      if brackets then
                              write(f, '(');
                      parbol(f, rptr);
                      if brackets then
                              write(f, ')');
                    end;

        xfer:
                begin
                parbol(f, lptr);
                      write(f, '<-');
                      parbol(f, rptr);;
                end;

        notaimb:
                 begin
                      write(f, '^');
                      brackets := not(lptr^.ntipo in [ident, modelo]);
                      if brackets then
                              write(f, '(');
                      parbol(f, lptr);
                      if brackets then
                              write(f, ')');
                end;

        actaimb:
                 begin
                 write(f, 'activate');
                      parbol(f, lptr);
                end;

        gotoaimb:
                 begin
                 write(f, 'goto');
                      parbol(f, lptr)
                end;

        onsimb:
                begin
                 write(f, 'on');
                 if lptr = nil then
```

```
                    begin
                    if (rptr^.ntipo <> actsimb) and (rptr^.ntipo <> gotosimb) then
                        begin
                        writeln('ERROR DE COMPILACION:parbol ES NULO EN ESTA CONDICION');
                          write('rptr: ');
                          parbol(f, lptr);
                          writeln;
                         halt
                        end
                    else
                        write(f, '"AnyInput"');
                    end
                else
                    parbol(f, lptr);
                write(f, 'do');
                if rptr^.ntipo = dospuntos then
                    begin
                    writeln(f, '(');
                    parbol(f, rptr);
                     writeln(f);
                    writeln(f, ')');
                    end
                else
                    parbol(f, rptr);
                end;

puntoycoma:
                    begin
                        parbol(f, lptr);
                        if rptr <> nil then
                          begin
                          writeln(f, ';');
                              parbol(f, rptr);
                          end;
                        end;
        modelo:
                    begin
                     write(f, 'OB');
                     for i := bitmax - len to bitmax do
                         write(f, pattn[i]);
                     write(f, '%');
                     impnum(f, len);
                     end;
        ident:
                    with tabla[tablap] do
                    begin
                if clase = edodef then
                    write(f, '#');
                 write(f, nombre:nomlong);
                if clase = termdef then
                begin
                if typ <> tipopunto then
                    begin
                    write(f, '<');
                    impnum(f, rege);
                    if len > 1 then
                        begin
                        write(f, ':');
                        impnum(f, regf);
                    end;
                 write(f, '>');
                end;
      end;
end;
end; {case}
end; {parbol}
```

```
procedure marcas(p: nodop);

{Usa el registro de cada variable en estructura de arbol}
{Este es registrado como: un input para el módulo analizar}
{Para salida en este modulo}

 procedure marcalo(p: nodop; use: useset);
 begin
        If p <> nil then
              begin
              If p^.ntipo in [orsimb, andsimb, concat, xfer, onsimb, puntoycoma,
                              notsimb, gotosimb, actsimb, ident] then
                    with p^ do
                     case ntipo of
                     xfer:
                                 begin
                                  marcalo(iptr,[Output]);
                                  marcalo(rptr,[Input]);
                                 end;
                     ident:  tabla[tablap].uso := tabla[tablap].uso + use;
                     notsimb:  marcalo(iptr, use);
                     gotosimb, actsimb:   marcalo(iptr,[Output]);
                     onsimb:
                                 begin
                                 marcalo(iptr,[Input]);
                                 marcalo(rptr,[]);
                                 end;
                     puntoycoma:
                                       begin
                                       marcalo(iptr,[]);
                                       marcalo(rptr,[]);
                                       end;
                    orsimb, andsimb, concat:
                                                          begin
                                                          marcalo(iptr,use);
                                                          marcalo(rptr,use);
                                                          end;
                     end;
        end;
 end; {marcalo}


begin {marcas}
If p <> nil then
   marcalo(p,[]);
end; {marcas}


procedure desunir  (var y:longlist; var ny:integer;var xcpairlist;
                              npairs,  max:integer);
{Dado un vector con rango 1..bitmax y subrangos npairs, produce}
{ la lista ordenada de tamaños de subrangos formados }

var
   I, nyl, nyr : integer;
   yl, yr: set of 1..bitmax;

begin
yl := [1];
yr := [max];
for I := 1 to npairs do
        with x[i] do
              begin
             yl := yl +[izq];
              yr := yr +[der];
              If izq > 1 then
                    yr := yr + [izq - 1];
                    If der < max then
                          yl := yl + [der + 1];
```

```
                end;
        ny := 0;
        nyl := 1;
        nyr := 1;
        repeat
          while not(nyl in yl) do
                        nyl := nyl + 1;
                        while not(nyr in yr) do
                                nyr := nyr + 1;
                                ny := ny + 1;
                                y[ny] := nyr - nyl + 1;
                                yl := yl - [nyl];
                                yr := yr - [nyr];
        until nyr >= max;
end; (desunir)


function disecar(var x:nodop; size: integer):nodop;

(Pasa por una sentencia transferida, produciendo una nueva sentencia)
(abarcando solo el primer "tamaño" en bits. La sentencia es modificada)
(de modo que subsecuetemente sera disecada regresando una nueva)
(sentencia de transferencia. Usado para remover operadores de concatenación)
var
        p: nodop;
        f, s, i : integer;

begin(disecar)
p := nil;
if x <> nil then
  with x^ do
  begin
        if size > len then
                size := len;
                case ntipo of
                  xfer, andsimb, orsimb, notsimb:
                                p := nuenodo(ntipo, disecar(lptr, size),
                                                        disecar(rptr,size),size);
                  concat:
                                if lptr^.len > 0 then
                                  begin
                                        i := size - lptr^.len;
                                        p := disecar(lptr, size);
                                        if i > 0 then
                                                p := nuenodo(concat, p, disecar(rptr,i), size);
                                  end
                                else
                                  p := disecar(rptr, size);
                  ident:
                                begin
                                  s := regs;
                                  if regs > regf then
                                        begin
                                          f := s - size + 1;
                                          regs := f -1;
                                        end
                                  else
                                        begin
                                          f := regs + size - 1;
                                          regs := f + 1;
                                        end;
                                  p := nueident(tablap, s, f);
                                end;
                        modelo:
                                begin
                                  p := nuemodelo(size, pattn);
                                  if len > size then
                                        with p^ do
                                        for i := 1 to size do
```

```
                                        patin[bitmax - size + 1] := x^.patin[bitmax - len + l];
                                    end;
                            end;
        len := len - size;
        end;
    disecar := p;
end; {disecar}


function rmconcat(p : nodop): nodop;
{Remueve el operador de concatenación hacia mano izquierda del tamaño}
{de una sentencia transferida}

var
    ny, npairs, max, i : integer;
    y : longlist;
    x: pairlist;
    r, s: nodop;

        procedure rangos(p : nodop; offset : integer; var max: integer);
        begin
        if p <> nil then
            with p^ do
                case ntipo of
                notsimb, xfer, andsimb, orsimb, concat:
                                begin
                                 rangos(lptr, offset, max);
                                 if ntipo = concat then
                                        rangos(lptr, offset + lptr^.len, max)
                                 else
                                        rangos(lptr, offset, max);
                                end;
                modelo, ident:
                                begin
                                        npairs := npairs + 1;
                                        with x[npairs] do
                                                begin
                                                izq := offset + 1;
                                                        der := offset + len;
                                                        if max < der then
                                                                max := der;
                                                end;
                                end;
            end;
    end; {rangos}


    begin {rmconcat}
        npairs := 0;
        max := 0;
        rangos(p^.lptr, 0, max);
        desunir(y, ny, x, npairs, max);
        s := nil;
        for i := 1 to ny do
                begin
                 r :=disecar(p, y[i]);
                 if s = nil then
                        s := r
                 else
                                s := nuenodo(puntoycoma,r,s,0);
                end;
    dspsarbol(p);
        rmconcat := s;
    end;{rmconcat}


procedure coleccion(p, cond : nodop);
{Coloca una sentencia transferida p en la lista colección}
{de el identificador.....}
```

```
function esreverso(p: nodop): boolean;
    {Regresa verdadero si el bit ordenado de lhs de una sentencia }
    {transferida....}

     begin {esreverso}
     esreverso := false;
     if p <> nil then
           with p^ do
           with tabla[tablap] do
        esreverso:=(freg>lreg) and (regs<regf) or (freg<lreg) and (rege>regf);
     end;


 procedure reverso(var p: nodop);
{invierte el bit ordering de una sentencia a una expresión}

    var
          i: integer;
          temp : char;

    begin {reverso}
    if p <> nil then
          with p^ do
          case ntipo of
          xfer, andsimb, orsimb, concat:
                begin
                reverso(lptr);
                      reverso(rptr);
                end;
          notsimb:
                reverso(lptr);
          ident:
                if len > 1 then
                  begin
                i := regs;
                regs := regf;
                regf := i;
                end;
          modelo:
                 for i := 1 to len div 2 do
                        begin
                        temp := pattn[bitmax - i + 1];
                        pattn[bitmax - i + 1] := pattn[bitmax - len + i];
                        pattn[bitmax - len + i] := temp;
                        end;
    end;
    end; {reverso}

begin {coleccion}
    {p es siempre una sentencia xfer}
    if p <> nil then
      with p^ do
      begin
            if esreverso(lptr) then
                reverso(p);
            if cond <> nil then
                  p := nuenodo(orsimb, copiarbol(cond), p, p^.len);
            with tabla[lptr^.tablap] do
            begin
              if reune = nil then
                    reune := p
              else
                reune := nuenodo(puntoycoma, p, reune, 0);
              marcas(reune);
            end;
          if banderaexp then
            begin
              write(listado, ",':' 5, ': 3);
```

```
                        parbol(listado, p);
                        writeln(listado);
                      end;
        end;
end; {coleccion}

procedure separa(tp : integer);
{Restres la tabla de símbolos y produce la transferencia de sentencias}
{para cada terminal. Estos envuelven determinando}
{el conjunto completo de conecciones para cada subcampo de cada terminal}

var
  p, q: nodop;
  i, npairs, ny: integer;
  xc pairlist;
  y : longlist;

      procedure rangos(p :nodop );
      {Determina el rango de índices del subcampo y el número de subcampos}
      begin {rangos}
      If p <> nil then
            with p^ do
            If not(ntipo in [puntoycoma, onsimb, xfer]) then
              begin
                    writeln('COMPILACION ERROR: arg ilegal para rangos');
                    halt
              end
            else
              case ntipo of
                        puntoycoma:
                                begin
                                  rangos(lptr);
                                  rangos(rptr);
                                end;
                        onsimb:
                                rangos(rptr);
                        xfer:
                                with lptr^ do
                                begin
                                  npairs := npairs + 1;
                                  with x[npairs], tabla[tp] do
                                  begin
                                        If freg > lreg then
                                          izq := freg - rega + 1
                                        else
                                          izq := rega - freg + 1;
                                          der := izq + len - 1;
                                  end;
                                end;
            end;
      end; {rangos}

      function edolhs(p: nodop): boolean;
      {Regresa verdadero si el lhs es un solo terminal o este empieza para la}
      {primera posición bit de la declaración index terminal}

      begin {edolhs}
      with p^, tabla[tp] do
            If typ = tipopunto then
                        edolhs := true
                else
                        edolhs := freg = regs;
      end; {edolhs}

      procedure impnomb(var f: text; tp: integer);
      {Imprime el nombre de la declaración terminal y sus dimensiones}

      begin {impnomb}
      with tabla[tp] do
```

```
            begin
              write(f, nombre : nomlong);
              if reglen > 1 then
                    begin
                     write(f, '<');
                     impnum(f, freg);
                     write(f, ':');
                     impnum(f, lreg);
                     write(f, '>')
                    end;
            end;
end; {impnomb}

procedure impnombre(var f: text; tp: integer);
{imprime un comentario en el archivo conexion dando la}
{declaración terminal y su uso}

begin {impnombre}
with tabla[tp] do
            begin
             write(f, 'terminal');
             impnomb(f, tp);
             write(f,'uso = [');
             if uso = [] then
                    begin
                     write(f, '***SINUSAR***');
                     advertencia := advertencia + 1;
                     if pbandera then
                            begin
                                   write('w-');
                                   impnomb(salida, tp);
                                   writeln('No es usado ni por Input ni por Output');
                            end;
                     if lbandera then
                            begin
                                   write(listado, 'w-');
                                   impnomb(listado, tp);
                                   writa(listado, 'No es usado ni por Input ni por Output');
                            end;
             end
             else
                    begin
                     if Input in uso then
                            write(f, 'Input');
                     if Output in uso then
                            write(f, 'Output');
                    end;
             write(f, ']');
      end;
end; {impnombre}

function generar(var p: nodop; when: nodop; size: integer):nodop;
{Genera ecuaciones para cada uno de los subcampos de la}
{declaración terminal}
var
 q, cond, lhs, rhs: nodop;
      lhsize : integer;

      procedure walk(var p:nodop; when: nodop);
      {Un procedimiento recursivo para recorrer el árbol}
      {y producir la transferencia de sentencia}

      var
            x :nodop;

      begin {walk}
            if p <> nil then
              with p^ do
              case ntipo of
```

```
                                        puntoycoma:
                                              begin
                                                    walk(lptr, when);
                                                    walk(rptr, when);
                                                    if lptr = nil then
                                                          begin
                                                                x := rptr;
                                                                davnodo(p);
                                                                p := x;
                                                          end;
                                              end;
                                        onsimb:
                                          begin
                                              walk(rptr, lptr);
                                              if rptr = nil then
                                                dspsarbol(p);
                                          end;
                                        xfer:
                                          if edolhs(lptr) then
                                                begin
                                                 lhsize := lptr^.len;
                                                 x := disecar(rptr, size);
                                                 if(lhs <> nil) and (when = nil) then
                                                       begin
                                                       {Ocurre una sentencia incondicional cuando}
                                                       {ocurrio una asignación previa}
                                                        if bandera then
                                                              begin
                                                               write(listado, 'E - en conflicto con uso' );
                                                               parbol(salida, lhs);
                                                               writeln;
                                                               errores := errores + 1;
                                                               dspsarbol(x);
                                                              end
                                                             else
                                                              begin
                                                                    if when <> nil then
                                                                      x:= nuenodo(andsimb, copiarbol(when), x, size);
                                                                    if cond = nil then
                                                                      cond := copiarbol(when)
                                                                     else
                                                                      cond := nuenodo(orsimb, copiarbol(when), cond, 1);
                                                                    if rhs = nil then
                                                                                rhs := x
                                                                     else
                                                                            rhs := nuenodo(orsimb, x, rhs, size);
                                                                    if lhs = nil then
                                                                      lhs := disecar(lptr, size)
                                                                     else
                                                                      begin
                                                                            x:= disecar(lptr, size);
                                                                            dspsarbol(x);
                                                                      end;
                                                              end;
                                                        if lhsize = size then
                                                                      dspsarbol(p);
                                                 end;
                                                                            end;
                                    end; {case}
                              end; {walk}
                        begin {generar}
                         cond := nil;
                         lhs := nil;
                         rhs := nil;
                          walk(p, when);
                         q := nil;
                         if lhs <> nil then
                               begin
                                 q := nuenodo(xfer, lhs, rhs, 0);
```

```
                    dspsarbol(cond);
                  end;
            generar := q;
          end; {generar}

begin {separa}
 p := tabla[tp].reune;
 if errores = 0 then
       begin
          writeln(conexdon);
          write(conexdon, '(');
          impnombre(conexdon, tp);
          writeln(conexdon, ')');
          if p <> nil then
                with tabla[tp] do
                      begin
                       npairs := 0;
                       rangos(p);
                       desunir(y, ny, x, npairs reglen);
                       for i := 1 to ny do
                             begin
                              q := generar(p, nil y[i]);
                              if q <> nil then
                                    begin
                                     parbol(conexdon, q);
                                     writeln(conexdon);
                                     dspstree(q);
                                    end
                               else
                                    begin
                                     if ibandera then
                                           begin
                                            write(listado, 'W - Sin usar campos de ');
                                            impnomb(listado, tp);
                                            writeln(listado);
                                           end;
                                     advertencia := advertencia + 1;
                                     if pbandera then
                                           begin
                                            write ('W - Sin usar subcampos de);
                                            impnomb(output, tp);
                                            writeln;
                                           end;
                                    end;{}
                              if freg < ireg then
                                    freg := freg + y[i]
                               else
                                    freg := freg - y[i];
                             end;
       end;
 if p <> nil then
       dspsarbol(p);
       tabla[tp].reune := nil;
end; {separa}

function posicion(id: nelfa; tc: integer): integer;
{Busca en la tabla de simbolos .....}
var
  i: integer;
begin {posicion}
 tabla[0].nombre := id;
 i := tc;
 while tabla[i].nombre <> identif do
       i := i - 1;
       posicion := i;
end; {posicion}

function rdnum(var xc: integer): boolean;
{Lee un simbolo y lo convierte a número}
```

```
{Regresa verdadero si sucede}
begin {rdnum}
 x := 0;
 rdnum := simbolo in [numero, modelo];
 if not(simbolo in [numero, modelo]) then
        comperror(4)
  else
        begin
         if simbolo = numero then
                begin
                x := num;
                traesimb;
         end
          else
                if simbolo = modelo then
                        begin
                  if not adecimal(x, 1, pien) then
                        comperror(47);
                  traesimb;
                end;
        end;
end; {rdnum}

function expresion(fsys: setsimb; tc: integer):nodop;
{Analiza una expresión y regresa una estructura de árbol}
{esta rutina maneja la expresión OR}
{regresa 'nil' si ocurre un error}
var
 L : nodop;

  function variable(fsys: setsimb): nodop;
  {Analiza una variable}
  {Regresa nil si ocurre un error}
  var
        p       : nodop;
        ercont  : integer;
        i       : integer;

         function enrango(x, y, l: integer): boolean;
         {Verifica si las dimensiones de la variable ...}
         begin {enrango}
                enrango := (l >= x) and (l >= y) or (l >= y) and (l <= x)
         end; {enrango}

  begin {variable}
        ercont := errores;
        variable := nil;
        p := nil;
        i := posicion(identif, tx);
        if (i = 0) or (tabla[i].lvl <> nivel) then
          buscaosalta([], fsys, 2)
        else
          with tabla[i] do
          if clase in [modulo, edodef] then
                buscaosalta([], fsys, 36)
           else
                begin
                traesimb;
                p := nuevident(i, freg, ireg);
                if simbolo = flechizq then
                  with p^ do
                  begin
                        if typ = tipopunto then
                        buscaosalta([], fsys + [flechder], 26)
                  else
                        begin
                          traesimb;
                          if rdnum(i) then
                                begin
```

```
                If not enrango(freg, lreg, i) then
                        comperror(17);
                regs := i;
                regf := i;
                len  := 1;
                If simbolo = dospuntos then
                        begin
                        traesimb;
                        If rdnum(i) then
                                begin
                                        If not enrango(freg, lreg, i) then
                                        comperror(17);
                                regf := i;
                                len := abs(regs - i) + 1;
                                If len > bitmax then
                                        comperror(16);
                                end;
                        end;
                end;
                end;
                sigif(fechder, 24);
        end;
        If (errores > errcont) and (p <> nil) then
                devnodo(p);
        end;
        variable := p
end; {variable}

function exp(s : simbolos; L, R : nodop) : nodop;
{Regresa un nodo de tipo 's'dando (posiblemente) dos operandos.}
{Regresa nil si ocurre un error}
        var
        x : nodop;
        begin {exp}
         x := nil;
         If (L <> nil) and (R <> nil) then
                If L^.len <> R^.len then
                        comperror(35)
                else
                        x := nuenodo(s, L, R, L^.len);
                exp := x;
                If x = nil then
                  begin
                        dspsarbol(L);
                        dspsarbol(R);
                  end;
        end; {exp}

 function exp1(fsys : setsimb):nodop;
        {Analiza expresiones 'AND'}
        {Regresa nil si ocurre un error}
        var
         L : nodop;

                function exp2(fsys: setsimb): nodop;
                 {Analiza expresiones CONCATENADAS}
                 {Regresa nil si ocurre un error}
                 var
                        L, R, x : nodop;

                         function exp3(fsys: setsimb): nodop;
                                {Analiza ...}
                                label 1;
                                 var
                                        x: nodop;
                                        notfound : boolean;
```

```
                procedure setsize(var x: nodop);
                (Lee el tamaño de una constante numérica o modelo)
                begin (setsize)
                        traesimb;
                                if simbolo <> numero then
                                        busca(fsys, 4)
                                else
                                        begin
                                                x^.len := num;
                                                if num > bitmax then
                                                        comperror(16);
                                                traesimb;
                                        end;
                                end; (setsize)

        begin (exp3)
                        x := nil;
                        notfound := false;
                        exp3 := nil;
                        while simbolo in simbolos1 do
                        begin
                                1:
                        case simbolo of
                                notsimb:
                                        begin
                                          if not expbandera then
                                                comperror(10);
                                          traesimb;
                                          notfound := not notfound;
                                          goto 1
                                        end;
                                ident:
                                        x := variable(fsys);
                                modelo:
                                        begin
                                         x := nuemodelo(plen, patron);
                                         traesimb;
                                         if simbolo = porcent then
                                                setsize(x);
                                        end;
                                numero:
                                        if dec_bin(num, 1, 32) then
                                         begin
                                                x := nuemodelo(plen, patron);
                                                traesimb;
                                                if simbolo = porcent then
                                                        setsize(x);
                                         end;
                                parenizq:
                                         begin
                                                if not expbandera then
                                                        comperror(10);
                                                traesimb;
                                                x := expresion(fsys + [parender], tx);
                                                sigif(parender, 22)
                                         end
                        end;
                        if not (simbolo in fsys) then
                                begin
                                  depsarbol(x);
                                  buscaosalta(fsys, [parenizq], 6);
                                end
                  end;
                If notfound and(x <> nil) then
                        x := nuenodo(notsimb, x, nil, X^.len);
                exp3 := x;
        end; (exp3)
begin (exp2)
        x := nil;
```

```
                        R := nil;
                        L := exp3(fsys + [concat]);
                        if simbolo <> concat then
                              exp2 := L
                        else
                              begin
                                traesimb;
                                R := exp2([concat] + fsys);
                                if(L <> nil) and (R <> nil) then
                                        x := nuenodo(concat, L, R, L^.len + R^.len);
                                exp2 := x;
                                if x = nil then
                                      begin
                                        dispearbol(L);
                                        dispearbol(R);
                                      end;
                              end;
                  end; {exp2}

            begin {exp1}
              L := exp2(fsys + [andsimb]);
              if simbolo <> andsimb then
                    exp1 := L
              else
                    begin
                      if not expbandera then
                            comperror(10);
                      traesimb;
                      exp1 := exp(andsimb, L, exp1([andsimb] + fsys));
                    end;
            end; {exp1}
begin {expresion}
  L := exp1(fsys + [orsimb]);
  if simbolo <> orsimb then
        expresion := L
  else
        begin
          if not expbandera then
                comperror(10);
          traesimb;
          expresion := exp(orsimb, L, expresion([orsimb] + fsys, b))
        end;
end; {expresion}

procedure edodescripcion(indx : integer);
{imprime detalles de la máquina de estado finito controlada }
{por el archivo conexion}
begin {edodescripcion}
  with tabla[indx] do
        if errores = 0 then
          begin
              writeln(conexion);
              write(conexion, nombre:nomlong);
              writeln(conexion, ':');
              if uso = [] then
                begin
                    advertencia := advertencia + 1;
                    if pbandera then
                          writeln('w - edo', nombre:nomlong, 'No es usado');
                    if lbandera then
                          writeln(letado , 'w - edo', nombre:nomlong, 'No es usado');
                end
                else
                 if reune = nil then
                      begin
                        if lbandera then
                              writeln(letado, 'E- edo', nombre:nomlong, 'estado siguiente no');
                        writeln('E- edo', nombre:nomlong, 'siguiente estado no');
                        errores := errores + 1;
```

```
                    end
                   else
                    begin
                          parbol(conexion, reune);
                          writeln(conexion);
                    end;
        end;
end; {edodescripcion}

procedure bloque(tx, argptr:integer;fsys, stopsy:setsimb);
{Examina el cuerpo de un bloque}

var
  i, ntabla: integer;

  procedure nuesentrada(var entrada: tbientrada; nme: nalfa; idlen: integer; k: modulos);

  begin {nuesentrada}
        with entrada do
        begin
         nombre := nme;
         typ    := tipopunto;
         nomlong := idlen;
         reune  := nil;
         lvl    := nivel;
         clase  := k;
         uso    := [];
         if k = modulo then
                 begin
                 nargs  := 0;
                 argindex := 0;
         end
          else
                  begin
                 reglen := 1;
                 freg   := 0;
                 lreg   := 0;
         end;
        end;
  end; {nuesentrada}

   function entra(var entrada: tbientrada):integer;

  {Adiciona un nombre a la tabla de simbolos.
        Bandera de definición múltiple de nombres}

  var
         i: integer;

  begin {Entran objetos dentro de la tabla}
  {      i := posicion(entrada, nombre, tx);}
         with tabla[i] do
         if(i > 0) and (nivel = lvl) then
           begin
                   entrar := 0;
         comperror(8);
          end
         else
          begin
                 tx      := tx + 1;
         entrada.lvl := nivel;
         tabla[tx] := entrada;
         uso     := [];
         entrar    := tx;
          end;
  end;
```

```
procedure modifarg(var entrada: tbientrada; m, a : nalfa; nm, na : integer);
{Concatena el string del nombre del componente y el accesado}
{nombre del string terminal, separadas por el caracter '_'}
var
i : integer;

begin {modifarg}
    entrada.nomlong := nm + na + 1;
    if entrada.nomlong > longid then
      entrada.nombre[nm + 1]:= '_';
      na := longid- (nm + 1);
      for i := 1 to na do
            entrada.nombre[nm + 1 + i] := a[i];
end; {modifarg}

 procedure ptosycomas;
{Ayuda ********************}
 begin {ptosycomas}
    if simbolo <> puntoycoma then
      begin
          if lbanders then
              begin
                    writeln(listado, '': cc + 7, '^');
                    writeln(listado, 'w -; ******');
              end;
          advertencia := advertencia + 1;
          if pbanders then
                writeln(lineanum: 5, ':w -; insertado');
      end
    else
      traesimb;
 end; {ptosycomas}

 function declcuerpo(fsys: setsimb; fob: modulos): integer;
{Procesa el cuerpo de una declaración terminal o un argumento
     con declaración módulo}

var
entrada     : tbientrada;
baseofarga  : integer;

 function pair : boolean;
{Analiza las dimensiones de un terminal}

 begin {pair}
     pair := false;
     with entrada do
     if rdnum(freg) then
       if simbolo = dospuntos then
            begin
                traesimb;
                if rdnum(freg) then
                  begin
                        pair := true;
                        if freg = ireg then
                              comperror(31);
                  end;
            end
       else
                comperror(42);
 end; {pair}

 procedure traeresto;
{Analiza la información despues de el nombre de la variable}
 begin {traeresto}
     traesimb;
     if(simbolo = flechizq) and (fob <> edodef) then
          with entrada do
          if simbolo = flechizq then
```

```
            begin
                  traesimb;
                  typ := tipovector;
                  if pair then
                    begin
                          reglen := abs(freg - lreg) + 1;
                          if reglen > bitmax then
                                comperror(16);
                    end;
                  sigif(flechder, 24);
            end;
      end; {traersato}

      procedure declaration;
      {Adiciona la declaración a la tabla de simbolos}
      var
      i : integer;

      begin {declaration}
            i := entrar(entrada);
            {***************}
            traeresto;
            tabla[i] := entrada
      end; {declaration}

      begin {dclbody}
            baseofargs := br;
            traesimb;
            repeat
              if simbolo <> ident then
                    comperror(1)
              else
                    begin
                          if fob = modulo then
                            nuesntrada(entrada, identif, plen, tarmdef)
                          else
                                nuesentrada(entrada, identif, plen, fob);
                          declaration;
                    end;
              busca([coma, puntoycoma, parender] + fsys, 5);
            until parar(coma);
            if fob = modulo then
                  sigif(parender, 22);
            ptosycomas;
            busca(fsys, 20);
            declcuerpo := br - baseofarge;
      end; {declcuerpo}

      procedure sentencia(fsys: setsimb; cond, curedo: nodop);
      {Analiza una sentencia}
      var
            i : integer;
            x: nodop;

            function mkterm(p : nodop): nodop;
            {Realiza una asignación a un terminal temporalmente}
              var
                    lha: nodop;
                    i,j: integer;
                    n: nalfa;
                    entrada: tbientrada;
              begin {mkterm}
                    mkterm := nil;
                    if termcont > tabmax then
                      begin
                            writeln(listado, 'FATAL ERROR:  Muchos terminales temporales usados (', tabmax:4,')');
                            if banders then
                              begin
                              writeln(listado, 'FATAL ERROR:');
```

```
                        writeln(listado, ' Muchos terminales temporales usados (', tabmaxc4,')');
                    end;
                  halt;
                end;
              n:='TEMP';
              j := termcont;
              termcont := termcont + 1;
              for i := 7 downto 5 do
                begin
                  n[i] := chr(j mod 10 + ord ('0'));
                  j := j div 10;
                end;
              nuesentrada(entrada, n,8, termdef);
              i := entrar(entrada);
              if i > 0 then
                begin
                  the := nuesident(i, 0,0);
                  with tabla[i] do
                    begin
                      reune := nuenodo(xfer, the, copiarbol(p), 1);
                      marcea(reune);
                      if banderaexp then
                        begin
                          write(listado, ","": 5,":3);
                          perbol(listado, reune);
                        end;
                    end;
                  if banderaexp then
                    writeln(listado);
                  mkterm := the;
                end;
            end; {mkterm}

          procedure transfer(fsys: setsimb; cond: nodop);
          {Analiza una sentencia transferida}
var
  L,R, x, y, z, oncond: nodop;
  mANDnodo, leconcatenated, xferOK: boolean;
  i : integer;

              function checkths(p: nodop; var iscat, xok: boolean):boolean;
                var
                  x : boolean;

                  procedure walk(p: nodop; var x: boolean);
                  {Un procedimiento recursivo a verificar por concatenación}
                    begin {walk}
                      if(p <> nil) and x then
                    with p^ do
                  if ntipo in [ident, concat] then
                    begin
                      if ntipo = concat then
                        begin
                          iscat := true;
                          walk(lptr, x);
                          walk(rptr, x);
                        end;
                    end
                  else
                    x := false;
                end; {walk}

                begin {checkths}
              x := p <> nil;
              xok := x;
              iscat := false;
              walk(p,x);
              checkths := x;
                end; {checkths}
```

```
begin {transfer}
 x := nil;
 R := nil;
 L := expresion(fsys + [xfer], tx);
 if simbolo <> xfer then
      comperror(13)
 else
      begin
       if L <> nil then
       if not checklha(L, iconcatenated, xferOK) then
        begin
             comperror(34);
             deparbol(L);
        end
        else
             if not xferOK then
               begin
             comperror(40);
             deparbol(L);
               end;
             traesimb;
             R :=  expresion(fsys, tx);
             if(L <> nil) and (R <> nil) then
              if L^.len <> R^.len then
                          comperror(35)
                           else
                                begin
                                 if L^.len <> R^.len then
                                      if R^.ntipo = modelo then
                                       begin
                                            for i := bitmax - L^.len + 1 to bitmax - 1 do
                                            R^.patrn[i] := R^.patrn[bitmax];
                                            R^.len := L^.len;
                                       end
                                      else
                                       begin
                                            y := nil;
                                            if R^.ntipo <> ident then
                                                 R := mkterm(R);
                                            for i := 1 to L^.len do
                                                 if y = nil then
                                                      y := R
                                                 else
                                                  y := nuenodo(concat, copiarbol(R), y, i);
                                             R := y;
                                       end;
                                      x := nuenodo(xfer, L, R, L^.len);
                                       marcas(x);
                                      if iconcatenated then
                                        x:= rmconcat(x);
                     rmANDnodo := false;
                     {Genera la condición en la transferencia}
                     if(curedo <> nil) and (cond <> nil) then
                      begin
                           oncond := nuenodo(andsimb, curedo, cond, 1);
                           rmANDnodo := true;
                      end
                      else
                           if cond = nil then
                                oncond := curedo
                           else
                                oncond := cond;
                          z := oncond;
                          if xferOK and (x <> nil) then
                           begin
                                if x^.ntipo = puntoycoma then
                                  begin
                                       if oncond <> nil then
```

```
                                        If oncond^.ntipo in [andsimb, orsimb] then
                                            z:= mkterm(z);
                                        while x^.ntipo = puntoycoma do
                                            begin
                                              coleccion(x^.lptr, z);
                                              y := x^.rptr;
                                              devnodo(x);
                                              x := y;
                                            end;
                                    end;
                                coleccion(x,z);
                            end;
                        If rmANDnodo then
                                devnodo(oncond);
                    end;
                end;
                If x = nil then
                  begin
                        dispearbol(L);
                        dispearbol(R);
                  end;
        end; {transfer}

procedure califica(fsys: setsimb; cond, curedo: nodop; identrada: Integer);
{Analiza el cuerpo de la sentencia despues de una estado de etiqueta }
var
        x : nodop;
      begin {califica}
        traesimb;
        If simbolo <> dospuntos then
                buscaoseita([dospuntos], fsys, 42)
        else
                begin
                If curedo <> nil then
                        comperror(48);
                  traesimb;
                  x := nuesident(identrada,0,0);
                  sentencia(fsys, cond, x);
                  devnodo(x);
                end;
  end; {califica}
 procedure ensentencia(fsys: setsimb);
{Analiza la sentencia "on ... do" }
var
        x, y: nodop;
      begin {ensentencia}
        traesimb;
        y := nil;
        x := expresion(fsys +[dosimb], tx);
        If x <> nil then
                If x^.tan <> 1 then
                        begin
                        comperror(39);
                        dispearbol(x);
                end;
        cigif(dosimb, 25);
        If cond <> nil then
                begin
                y := nuenodo(andsimb, x, cond, 1);
                sentencia(fsys + [elsesimb], y, curedo);
                devnodo(y)
                end
              else
                        sentencia(fsys + [elsesimb], x, curedo);
        If simbolo = elsesimb then
                begin
                        traesimb;
                        x := nuenodo(notsimb, x, nil, 1);
                        If cond <> nil then
```

```
                                begin
                                y := nuenodo(endsimb, x, cond, 1);
                                sentencia(falya, y, curado);
                                devnodo(y)
                                end
                        else
                                sentencia(falya, x, curado);
                end;
        dspsarbol(x);
end; {ensentencia}

procedure veasentencia;
{Analiza la sentencia goto, salva el estado siguiente de información}
{en la tabla de simbolos de entrada para el estado corriente}
var
        i : integer;

        procedure storsedo(var p: nodop; cond: nodop; identrada: integer);
        {Almacena el estado siguiente de información el la tabla de simbolos ***}

        var
          hit: boolean;
          x: nodop;

          procedure match(p: nodop);
          {Verifica si ya tenemos una entrada para el mismo siguiente estado. }
          {Si es asi, entonces las condiciones OR*****}

          begin {match}
                        if (p <> nil) and not hit then
                                with p^ do
                                case ntipo of
                                        puntoycoma:
                                                                begin
                                                                        match(lptr);
                                                                        match(rptr);
                                                                        end;
                                                onsimb:
                                                        begin
                                                                match(rptr);
                                                                if hit then
                                                                  lptr := nuenodo(onsimb, cond, lptr, 1);
                                                                end;
                                                gotosimb:  hit := lptr^.tablap = identrada
                                end; {case}
          end; {match}

          begin {storsedo}
          hit := false;
          match(p);
          if not hit then
                  begin
                        x := nuesident(identrada, 0,0);
                        x := nuenodo(gotosimb, x, nil, 0);
                        x := nuenodo(onsimb, cond, x, 0);
                        marcas(x);
                        if banderaexp then
                          begin
                          write(listado, ' ', '**', 5, ' ':3);
                                parbol(listado, x);
                                writeln(listado)
                          end;
                        if p = nil then
                          p := x
                        else
                                p := nuenodo(puntoycoma, x, p, 0);
                        end;
          end; {storsedo}
```

```
begin (*esesentencia*)
            traesimb;
            if simbolo <> ident then
                  comperror(1)
            else
                  begin
                  i := posicion(identif, tx);
                  if (i > 0) and (tabla[i].lvl = nivel) then
                        if curedo <> nil then
                              storedo(tabla[curedo^.tablap].reune, copiarbol(cond), i)
                        else
                              comperror(50)
                  else
                        comperror(2);
                  traesimb
                  end;
end; (*gotostament*)

procedure actsentencia(fsys: setsimb);
(*Analiza la sentencia " activate .. do ")

var
      i : integer;
      y : nodop;

      procedure guardact(var p: nodop; cond, actid: nodop);
      (*Salva la sentencia activa en la salida especificada)
      (de el estado corriente de una maquina controladora de estado finito)

      var
        x: nodop;

      begin (*guardact)
      x := nuenodo(actsimb, actid, nil, 0);
      x := nuenodo(onsimb, cond, x, 0);
      marcas(x);
      if banderasxp then
        begin
              writa(listado, '', ':', 5,':3);
              parbol(listado, x);
              writeln(listado)
              end;
      if p = nil then
        p := x
      else
            p := nuenodo(puntoycoma, x, p,0);
      end; (*guardact)

begin (*actsentencia)
traesimb;
if simbolo <> ident then
      comperror(1)
else
      begin
            y := expresion(fsys, tx);
            if y^.ntipo <> ident then
                  comperror(21)
            else
                  if y^.len <> 1 then
                        comperror(23)
                  else
                        if (curedo = nil) and (cond = nil) then
                              comperror(27)
                        else
                              begin
                              i := y^.tablap;
                              if (i > 0) and (tabla[i].lvl = nivel) then
                                    if curedo <> nil then
                                          guardact(tabla[curedo^.tablap].reune, copiarbol(cond), y)
```

```
                                                    else
                                                        comperror(50)
                                                else
                                                    comperror(2);
                                                end;
                                    end;
                            end; {actsentencia}

begin {sentencia}
    while simbolo = puntoycoma do
            traesimb;
    if simbolo in simbolos2 + [ident] then
        case simbolo of
                ident:
                                        begin
                                         i := posicion(identif, b);
                                         if tabla[i].clase = edodef then
                                                if tabla[i].M = nivel then
                                                    begin
                                                    califica(fsys, cond, curedo, i);
                                                    if not edobandera then
                                                            begin
                                                            comperror(33);
                                                            saltos(fsys)
                                                            end;
                                                    end
                                                else
                                                    comperror(9);
                                        end;
                        inisimb:
                                        begin
                                            traesimb;
                                            x := cond;
                                            if cond <> nil then
                                                if cond^.ntipo in [andsimb, orsimb] then
                                                        x :=mkterm(cond);
                                                sentencia([puntoycoma, finsimb] + fsys, x, curedo);
                                            while simbolo in [puntoycoma] + simbolos2 do
                                              begin
                                                        if simbolo <> puntoycoma then
                                                            ptosycomas;
                                                        sentencia(fsys + [puntoycoma, finsimb], x, curedo)
                                              end;
                                              while simbolo = puntoycoma do
                                                        traesimb;
                                              sigif(finsimb, 12)
                                        end;
                        gotosimb:
                                        if not edobandera then
                                            begin
                                             comperror(37);
                                             traesimb;
                                             saltos(fsys)
                                            end
                                        else
                                            vasasentencia;
                         actsimb:
                                            actsentencia(fsys);
                                onsimb:
                                            onsentencia(fsys);
        end;
        buscasalta(fsys, [], 7);
  end; {sentencia}

procedure modulodecl(fsys:setsimb);
{Analiza un módulo de descripción}
var
  i, idindex : integer;
  entrada    : tblentrada;
```

```
  isexternal : boolean;

begin {modulodecl}
 traesimb;
 idindex := 0;
 if simbolo <> ident then
        comperror(1)
 else
        begin
          writeln(conexion);
          writeln(conexion, '{$modulo:', identif, '}');
          nuseentrada(entrada, identif, plen, modulo);
          idindex := entrar(entrada);
          traesimb
        end;
 nivel := nivel + 1;
 if simbolo = parenizq then
        tabla[idindex].nargs := declcuerpo(fsys + [parender], modulo)
 else
        bloque(tx, argptr, fsys, [puntoycoma, pesos]);
        nivel := nivel - 1;
        if not isexternal then
        {Solo imprime los terminales de módulos que tienen una descripción local}
        for i := idindex +1 to tx do
                with tabla[i] do
                if clase = termdef then
                        begin
                 if errores = 0 then
                        separa(i);
                  dapsarbol(reune);
                 reune := nil
                end;
                writeln(conexion, '{$Endmodulo:', tabla[idindex].nombre, '}');
                if tabla[idindex].nargs > 0 then
                        begin    {mueve args como arguemanto de stack}
                 tabla[idindex].argindex := argptr;
                 for i := idindex + 1 to idindex + tabla[idindex].nargs do
                        begin
                                argtabla[argptr] := tabla[i];
                        argptr := argptr +1;
                        if argptr > tabmax then
                          begin
                                writeln('argumento de pila overflow (', tabmax:3, 'items');
                                halt;
                          end;
                          end;
                end;
                 tx := idindex;
                 if simbolo <> pesos then
                        begin
                  ploycomas;
                  busca(fsys, 20);
                 end;
end; {modulodecl}

procedure compdecl(fsys:setsimb);
{Analiza la declaración de un componente}
var
  i,k, obc: integer;
  entrada: tblentrada;

begin {compdecl}
 traesimb;
 if simbolo <> ident then
        buscasalta([], fsys+[puntoycoma], 1)
 else
        begin
          i := posicion(identif, tx);
          with tabla[i] do
```

```
                    if i = 0 then
                busca(fsys + [puntoycoma], 2)
              else
                if clase <> modulo then
                    busca(fsys + [puntoycoma], 3)
                else
                    begin
                  traesimb;
                  repeat;
                  if simbolo <> ident then
                    buscaosalta([], fsys + [coma, puntoycoma], 1)
                  else
                    begin
                      if narga > 0 then
                    {Declara cada argumento como <nombrecomponente><nombreargumento>}
                        for k := argindex to argindex + narga - 1 do
                            begin
                          entrada := argtabla[k];
                          entrada.clase := termdef;
                          modifarg(entrada, identif, argtabla[k].nombre, plen, argtabla[k].nomlong);
                          obc := entrar(entrada);
                        end;
                        traesimb;
                    busca([coma, puntoycoma] + fsys, 5);
                    end;
                  until parar(coma);
              end;
          end;
        ptosycomas;
        busca(fsys, 20)
end; {compdecl}


begin {bloque}
ntabla := tx + 1;
repeat;
              while simbolo = modulosimb do
                    modulodecl(fsys + [externsimb, pesos]);
              while simbolo = cdef do
                        compdecl(fsys);
              if simbolo = tdef then
                i := declcuerpo(fsys, termdef);
                if simbolo = sdef then
                    begin
                    if not edobandera then
                        comperror(33);
                    i := declcuerpo(fsys, edodef);
                    end;
                if simbolo in simbolos3 then
                    comperror(15)
                else
                    buscaosalta([inisimb, pesos], fsys, 11)
until simbolo in simbolos2 + [pesos];
if simbolo <> pesos then
  begin
  sigif(inisimb, 11);
  repeat
  repeat
  repeat
  sentencia(fsys + [puntoycoma, finsimb], nil, nil)
  until not (simbolo in simbolos2)
  until parar(puntoycoma);
  sigif(finsimb, 12);
  if not(simbolo in stopsy) then
        buscaosalta([], fsys, 18)
      until simbolo in simbolos3 + stopsy + [inisimb];
      for i := ntabla to tx do
          with tabla[i] do
              if clase = termdef then
```

```
                                    begin
                                     if errores = 0 then
                                            separa();
                                            depsarbol(reune);
                                           reune := nil
                                    end;
            writeln(conexion);
            writeln(conexion, '{$edotable}');
           for i := ntabla to tx do
                  with tabla[i] do
                  if clase = edodef then
                           begin
                            if errores = 0 then
                                   edodescripcion();
                                   depsarbol(reune);
                                  reune := nil;
                           end;
                  writeln(conexion, '{$Fin de la tabla}');
          end;
end; {bloque}
procedure inicio;          {inicializa todos los valores iniciales}
  procedure initabsimb;    {inicializa las tablas de simbolos    }
   begin {initabsimb}
  for caracter := 'A' to 'Y' do
          asimb[caracter] := nulo;

          {Debe haber un Diccionario de palabras en orden lexicográfico}

          palabra[1]  := 'activate' ;
          palabra[2]  := 'begin';
          palabra[3]  := 'component';
          palabra[4]  := 'do';
          palabra[5]  := 'else' ;
          palabra[6]  := 'end';
          palabra[7]  := 'external';
          palabra[8]  := 'goto';
          palabra[9]  := 'module';
          palabra[10] := 'on';
          palabra[11] := 'edo';
          palabra[12] := 'terminal' ;

          waimb[1]  :=  actsimb;
          waimb[2]  :=  inisimb;
          waimb[3]  :=  odef;
          waimb[4]  :=  dosimb;
          waimb[5]  :=  elsesimb;
          waimb[6]  :=  finsimb;
          waimb[7]  :=  externsimb ;
          waimb[8]  :=  gotosimb;
          waimb[9]  :=  modulesimb;
          waimb[10] :=  onsimb;
          waimb[11] :=  sdef;
          waimb[12] :=  tdef;

          asimb['|'] := orsimb;
          asimb['^'] := notsimb;
          asimb['$'] := pesos;
          asimb['('] := parentiza;
          asimb[')'] := parender;
          asimb['<'] := flechizq;
          asimb['>'] := flechder;
          asimb['@'] := concat;
          asimb[';'] := puntoycoma;
          asimb[','] := coma;
          asimb['{'] := inicoment;
          asimb['}'] := fincoment;
          asimb[':'] := dospuntos;
          asimb['%'] := porcent;
          asimb['&'] := andsimb;
```

```
                {simbolos = (nulo, ident, numero, modelo, xfer);}

 end; {initabsimb}


 begin {inicio}
                initabsimb;
                simbolos1   := [ident, modelo, numero, parenizq, notsimb];
                simbolos2  := [inisimb, onsimb, gotosimb, actsimb];
                simbolos3  := [tdef, modulosimb, sdef, cdef];
                cc          := 0;
                ll          := 0;
                caracter    := ' ';
                nivel       := 0;
                errores     := 0;
                advertencia := 0;
                lineanum    := 0;
                for kk      := 1 to longid do
                        blankid[kk]:=' ';
                nodoscont   := 0;
                nodos       := 0;
                nodosdesoc  := 0;
                nlibre      := nil;
                plibre      := nil;
                llibre      := nil;
                termcont    := 0;
                lbandera    := false;
                pbandera    := true;
                banderaexp  := false;
                edobandera  := false;
                expbandera  := false;
                assign(programa,'programa.txt');
               REWRITE(PROGRAMA);
                assign(listado, 'listado');
                rewrite(listado);
                assign(conexion, 'conexion');
                rewrite(conexion);
                {rewrite(listado, 'listado');
                rewrite(conexion, 'conexion');}
               assign (salida,'RESULTADO.txt');
                rewrite(salida);
 end; {inicio}


begin {principal}
 inicio;
 traesimb;
 repeat
                bloque(0,0, simbolos3 + simbolos2, [pesos]);
                if simbolo <> pesos then
                   comperror(19)
 until simbolo = pesos;
 if errores > 0 then   {Borra el archivo si hay errores en fuente}
        rewrite(conexion);
 if bandera then
        begin
        writeln(listado);
        writeln(listado, 'Número de errores detectados: ', errores);
        writeln(listado, 'Número de advertencia:  ', advertencia);
        writeln(listado, 'Número de nodos sin recuperar', nodoscont);
        writeln(listado, 'Número de nodos asignados', nodosdesoc);
        writeln(listado, 'Número de nodos usados', nodos );
        end;
 if errores > 0 then
        begin
        writeln;
        writeln('Número de errores detectados : ' , errores);
        end;
end.{principal}
```

BIBLIOGRAFIA

1 T. Ohtsuki.
Advances in CAD for VLSI
Vol. 7 R. W. Hartenstein. *Hardware Description Languages*
New York, North - Holland, 1987, 495 pp.

2 Thomas E. Dillinger.
VLSI Engineering
New Jersey, Prentice Hall, 1988, 863 pp.

3 Thomas Downs, Mark F. Schulz.
***Logic Desing with Pascal* (Computer Aided Design Techniques)**
New York, Van Nostrand Reinhold, 1988, 513 pp.

4 Jean Paul Tremblay, Paul G. Sorenson.
Theory and Practice of Compiler Writing
Singapure, Singapure National Printers, 1985, 796 pp.

5 Alfred V. Aho, Jeffrey D. Ullman.
Principles of Compiler Desing
Unites States of America
Addison - Wesley Publishing Company, 1979, 603 pp.

6 Charles N. Fischer, Richard J. LeBlanc, Jr.
Crafting a Compiler
Menlo Park, California, The Benjamin/Cumming Publishing Company, Inc

7 Tom Swan.
Mastering Turbo Pascal 5.5
Unites States of America
Hayden Books, 1991, 877 pp.

8 Ibraihim Zeid.
CAD/CAM Theory and Practice
Ed. McGraw-Hill Inc.
U.S.A, 1991.