~~03061~~ 03063
2
14

Un Programa Auxiliar en el Diseño de Circuitos: Su Implantación Mediante Código Hilvanado

Tesis

que para obtener el título de

Maestro en Ciencias de la Computación

presenta

David Arturo Rosenblueth Laguette

U.A.C.P.P. del C.C.H., U.N.A.M., 1983

[illegible] C[illegible]N
FALLA DE ORIGEN

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

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

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

Contenido

1 Introducción 4

1.1 Diseño de circuitos auxiliado con computadora 4

1.2 Objetivos del programa 11

2 Diseño 13

2.1 Paquete de graficación 13

2.2 Paquete de conexiones 23

3 Selección del lenguaje de programación 33

3.1 Código hilvanado 33

3.2 Forth 45

4 Implantación 52

4.1 Paquete de graficación 52

4.2 Paquete de conexiones 60

5 Conclusiones 66

5.1 Resultados 66

5.2 Extensiones y trabajo futuro 67

Bibliografía 71

Comandos del sistema 78

Ejemplo 82

Capítulo 1

Introducción

1.1 Diseño de circuitos auxiliado con computadora

1.1.1 Niveles de abstracción en la descripción de circuitos

La idea de traspasar algunas etapas del diseño de circuitos a una computadora es bastante natural y apareció temprano en la historia de la computación [Computer 74]. El diseño auxiliado con computadora es una de las ramas más interdisciplinarias de la computación. Es costumbre que un sistema para diseño tenga una interfaz gráfica, así como un manejador de base de datos (DBMS). También es frecuente que posea un simulador y se comunique con un sistema orientado a la manufactura [Computervision 80]. Además, es importante conocer las necesidades de los usuarios y estar familiarizado con la aplicación en la que se va a emplear el sistema al elaborarlo.

Un sistema auxiliar en el diseño de circuitos recibe como entrada una descripción del circuito en algún lenguaje preferentemente de alto nivel y genera como salida otra descripción de más bajo nivel. La salida del sistema es por ejemplo, una lista de piezas acompañada por el dibujo de un circuito impreso o una lista de alambrado, que representan el mismo circuito descrito por el usuario, pero ahora está en un lenguaje "comprensible" por una máquina alambradora. Es decir, que la labor de un sistema auxiliar en el diseño de circuitos es básicamente un proceso de traducción en forma similar a la de un *compilador* de programas. De hecho, es posible encontrar paralelo en más de una fase entre el compilador de programas y el compilador de circuitos.

¿Cómo debe descibirse el circuito al sistema de diseño? Algunas de las ideas surgidas de la investigación de lenguajes de muy alto nivel y de programación automática son también

válidas en la descripción de circuitos digitales (por ejemplo [Parker 81]). La descripción del circuito puede consistir en expresar a un nivel muy abstracto un algritmo, y hasta probar que la compilación fue correcta y que el circuito generado sí cumple con el algoritmo dado (de hecho, actualmente es posible probar matemáticamente que un circuito sencillo, digamos un contador de 5 bits, funciona correctamente). Sin embargo, la mayoría de los sistemas de diseño prácticos emplean descripciones de más bajo nivel. Para determinados tipos de circuitos se conocen algoritmos que sintetizan sistemas digitales. El más difundido es probablemente el algoritmo de Quine-McCluskey [Hill 74] que obtiene un circuito combinacional construido en dos niveles de lógica a partir de la tabla de verdad. El método de Quine-McCluskey requiere, sin embargo, de un número grande de operaciones y una cantidad muy considerable de memoria.

Recientemente se han desarrollado variantes que requieren menos tiempo o que no garantizan encontrar el mínimo absoluto pero sí una solución bastante buena aun para funciones de varias salidas. Estos algoritmos se utilizan en la programación arreglos lógicos programables (PLAs) [Signetics 78], [MMI 79]. Se conocen también técnicas para sintetizar circuitos secuenciales con los que se puede hacer con cuidado la reducción del número de estados, y asignar estados de manera que se reduzca la parte combinacional del circuito o se eviten carreras [Hill 74]. Hay sistemas que reciben la descripción del circuito a un nivel más bajo todavía, en la que el usuario indica los componentes que desea emplear, por lo que no ocurre ningún proceso de síntesis por parte del programa. La actividad del sistema consiste más bien en una traducción de notación. Esto no implica que el diseñador esté impedido para abstaer por sí mismo y definir una estructura arborescente en su circuito, pero sí significa que debe sintetizar todo el circuito y escoger los componentes por él mismo.

Clasificaremos a los programas auxiliares en el diseño de circuitos de acuerdo con el nivel de abstracción del circuito a su entrada:

1. Algorítmico. El usuario especifica el algoritmo y el circuito se diseña y se prueba formalmente. Corresponde a los lenguajes de muy alto nivel en programación y al igual que estos, son pocos y se emplean solamente a un nivel experimental. En general la traducción no es totalmente automática y es necesario guiar al sistema [Bauer 81]. A pesar de lo atractivo de la metodología, es poco el trabajo que se está haciendo en esta dirección [Breuer 81].

2. Lenguajes Registro-Transferencia. Equivalen a los lenguajes de alto nivel en sistemas de programación pues en general sí hay un proceso de síntesis por parte del sistema de diseño, aunque no es de tan alto nivel como sería deseable. Bell y Newell introdujeron las notaciones ISP (instruction set processor) y PMS (processor-memory-switch) [Bell 71] [Bell 71a] que sirvieron para describir sistemas digitales complejos y posteriormente se desarrollaron como lenguajes [Director 81].

   Uno de los lenguajes de prgramación que más ha influído en el desarrollo de lenguajes descriptores de circuitos es APL [Iverson 62]. Aunque la intención original de Iverson era un sistema que sirviera tanto para describir sistemas de programación como sistemas digitales, APL evolucionó por dos caminos distintos. A pesar de ello, el lenguaje de Iverson es probablemente el lenguaje de computación con más rango de resolución (en el sentido de la facilidad que posee para definir algoritmos con operaciones muy primitivas por un lado, como programas con instrucciones de muy alto nivel por otro). Los principales lenguajes de circuitos basados en APL son AHPL [Hill 73], del cual existen varios compiladores y ALERT [Friedman 69], que fue programado en una IBM 7094. ALERT fue usado,

entre otras aplicaciones; para procesar una parte importante en el diseño de la IBM 1800, pero el resultado producido requería 100% más que en el sistema original, por lo que se propusieron modificaciones con las que ALERT produciría solamente un 30% de redundancia [Lewin 77].

Otro lenguaje conocido es CDL (Computer Design Language) propuesto por Chu [Chu 65], basado en Algol, del que existen varias implantaciones. Las extensiones principales hechas a Algol son: el tipo de dato *registro* y la instrucción de *transferencia*, que reemplaza al concepto de asignación. El lenguaje es adecuado para describir sistemas más bien sencillos, pues no aprvecha la estuctura de bloques de Algol para sintetizar circuitos especificados jerárquicamente.

DDL [Duley 68] es similar a los lenguajes mencionados. Permite especificar un circuito a nivel de registros o de compuertas, definiendo las conexiones (en ocasiones dando nombres a determinados alambres), las líneas terminales, relojes y elementos de retraso. Al igual que la mayoría de los lenguajes de circuitos, acepta también definiciones de microoperaciones.[1] Además, da la posibilidad de asignar un nombre a alguna microoperación o ecuación lógica a la manera de un macro (esto ya casi es jerárquico).

Existen más de 40 lenguajes para especificar circuitos [Computer 74] [Computer 77] [Shiva 79] inspirados en lenguajes de programación. La mayoría exhibe dos tipos de enunciados: los que definen la topología del circuito y los que establecen las microoperaciones. Cuando un lenguaje consiste únicamente en la parte declarativa, el orden lexicográfico de las instrucciones es irrelevante y recibe el nombre de lenguaje *no-procedural* [Lewin 77].

8. Componentes. Si el sistema para diseño de circuitos carece de métodos sintetizadores de circuitos, obliga al diseñador a describir los componentes de los circuitos. Los sistemas de este tipo pueden compararse con los ensambladores, que traducen los nombres de las instrucciones a códigos de operación y resuelven etiquetas, pero dejan al usuario la elección de las instrucciones de máquina. La ayuda proporcionada por los sistemas que requieren conocer los nombres de los componentes de los circuitos es de cualquier manera valiosa, quizá no tanto para describir sistemas complejos, pero sí en lo que respecta a la documentación del circuito. Estos sistemas son más bien empleados en el desarrollo de tarjetas impresas que en el diseño de circuitos integrados como se verá más adelante. Los programas para diseño de circuitos a nivel de componentes son no-procedurales.

1.1.2 Diseño de circuitos impresos y de circuitos integrados

A pesar de que tanto el diseño de tarjetas impresas como el de chips es bastante similar, y los lenguajes descriptores de circuitos sirven para definir sistemas digitales a ambos niveles de complejidad, es necesario establecer algunas diferencias fundamentales.

En los circuitos de alta integración (LSI), el objetivo principal del diseñador es ***minimizar el área*** del circuito [Hayes 80], por lo que se vuelve especialmente importante

[1] una microoperación es la transferencia de información que puede realizarse en paralelo durante un ciclo de reloj [Mano 79]

la geometría del sistema y es aceptable invertir un esfuerzo considerable en algoritmos que posicionen los componentes [Rupp 81].

Otra diferencia entre ambos tipos de sistemas digitales radica en el tipo de bloques elementales empleados: mientras en la fabricación de chips el diseñador utiliza transistores, las tarjetas impresas se elaboran con circuitos integrados. Pero los circuitos integrados de las tarjetas deben ser *estándar*; esto implica que ya no es válido aplicar algoritmos de sintetización que produzcan soluciones con un uso indiscriminado de compuertas lógicas. Es indispensable conocer los chips que hay en el mercado. El diseñador de tarjetas impresas (o alambradas) ya no es el mismo que en la década de los sesentas: ahora su objetivo principal es ***minimizar el número de chips*** [Blakeslee 79].

El costo del sistema es proporcional al número de chips, así como el costo de diseño y la probabilidad de falla. Ya dejan de ser tan necesarias las técnicas tradicionales de sintetización (de hecho, no se conoce ningún sistema práctico para diseñar tarjetas que sintetice circuitos con chips estándar), para dar lugar a mecanismos de acceso a información de chips existentes en el mercado. (Los bloques elementales debe existir en realidad y no solamente estar anunciados; pues no es raro encontrar diseños de circuitos hechos sobre el manual, con chips agotados, o basados en chips tan modernos que no existen.)

Un "compilador" de circuitos a nivel de tarjetas podría utilizar las técnicas de optimización similares de los compiladores de programas, pues debe escoger los circuitos integrados de manera que se minimice el número de ellos,[2] sin importar si desperdicia el 90% de la lógica. Sin embargo, ya que el tipo de circuitos integrados que debe usar varía con el tiempo (porque determinado circuito se agota o aparece alguno nuevo), equivale a un compilador cuyo lenguaje de máquina cambia con el tiempo.

En los circuitos integrados los sistemas para diseño de circuitos que carecen de técnicas de sintetización se emplean más bien en la especificación de tarjetas impresas.

La mayoría de las veces la verificaci'on se hace en forma *ad hoc* con simulación o construyendo prototippos del circuito [Breuer 81].

Inspirados en lenguajes de programación, aparecieron en las dos décadas pasadas más de 40 lenguajes descriptores de circuitos digitales [Shiva 79], con los que el diseñador del circuito escribía su sistema en forma similar a un algoritmo de programación. Estos lenguajes tuvieron menos éxito que el esperado debido a que a pesar de que no existe ninguna notación estándar para describir sistemas digitales, sí están muy extendidos los métodos gráficos.

1.1.3 Simulador

En forma análoga a la elaboración de programas, resulta de mucha utilidad un simulador, que hace las veces del depurador (debugger). La mayoría de los simuladores modelan el circuito con por lo menos 3 niveles de lógica en las señales: uno, cero e "inespecificado". Este último es especialmente usado para detectar carreras y salidas en falso (hazards) y se

[2] más bien lo que interesa es minimizar el número de patas, pero equivale a minimizar el número de chips dando más peso a los chips con 24 o 40 patas

asocia a un estado intermedio o desconocido. Lo más natural es emplear simuladores discretos. Este tipo de programas posee la característica de poder ejecutar varios procesos en forma quasiparalela. Es común que almacene los procesos activos en una cola, a cada uno de los cuales asocia una variable representando el instante de tiempo en que debe tomar el control del procesador. Este es un tiempo de simulación, y nada tiene que ver con el tiempo real.

Aunque hay simuladores basados solamente en las ecuaciones lógicas del circuito, cualquier simulador que valga la pena tendrá que considerar los tiempos de respuesta de los componentes. Los más sencillos operan con el tiempo de retraso promedio (que sobre todo en circuitos en los que los componentes están en distintos chips, la varianza del retraso es muy grande) pero los hay los que permiten repetir la simulación con retrasos mínimos o máximos según sea el peor de los casos [Breuer 81].

En el diseño de circuitos integrados el simulador cobra especial importancia y no es raro que asocien variables de punto flotante al estado eléctrico de las conexiones. Incluso es frecuente el empleo de simuladores analógicos con lo que se establece un modelo matemático del circuito (o parte de él) y la labor del programa consiste básicamente en la resolución de un sistema de ecuaciones diferenciales. Sin embargo, esta técnica solo es práctica a un nivel bajo de simulación (v. gr. a nivel de transistores) y se aplica únicamente a una fracción del circuito. El simulador más popular de este tipo es probablemente Spice [Breuer 81].

En la mayoría de las ocasiones el simulador es un elemento aislado del programa de diseño lógico y la traducción para comunicar ambos módulos se hace manualmente. No obstante, resulta clara la necesidad de automatizar este proceso. En otras, ni siquiera el simulador está programado en una computadora, pero de todas maneras existe, aunque sea en la mente de los diseñadores.

1.1.4 Diseño Físico

La salida de los sistemas para diseño de circuitos es siempre a nivel de componentes, es decir, una lista de conexiones acompañada de una lista de piezas. Sin embargo, hay sistemas que van más allá e involucran aspectos físicos del circuito; su salida está además constituida por la posición de cada componente y las trayectorias de las conexiones.

El programa encargado del diseño físico es más o menos independiente del que ayuda a diseñar el circuito lógicamente. De hecho, es raro que las características físicas afecten radicalmente el diseño lógico, por lo que puede considerarse un módulo separado y para fines prácticos la mayor parte de la información fluirá en un sentido: del programa de diseño lógico al de diseño físico.

En el caso del diseño de tarjetas, es probable que durante el diseño físico se decida trasferir una compuerta de un chip a otro más cercano, pero en todo caso se estará realizando un intercambio de compuertas y el número de chips permanecerá constante. No obstante, es conveniente considerar la posibilidad de que el programa de diseño físico agregue o elimine compuertas con objeto de lograr una mejor partición del circuito en tarjetas o reducir la longitud de una conexioón. En este caso habrá que informar al programa de diseño lógico y actualizar el modelo del circuito. Asimismo, es factible que si el programa de diseño físico tiene la función de agregar terminaciones, capacitores de acomplamiento y en general componentes

que están en función de aspectos físicos, exista cierta comunicación hacia el programa de diseño lógico.

Suelen incluirse en el diseño físico algoritmos para hacer la partición del circuito (en tarjetas o en chips según el nivel al que se realice el diseño), cálculo de las trayectorias del las conexiones (ruteo) y posicionamiento de los componentes.

El problema de la partición del circuito puede plantearse como sigue. Dada la descripción lógica del sistema, subdividirlo en una jerarquía de subsistemas de manera que se cumplan dos restricciones:

1. que cada subsistema quepa en un área dada
2. que el número de conexiones en la frontera de cada subsistema sea menor que un número dado

En un paquete de diseño a nivel de tarjetas, significa que sea factible acomodar cada subsistema en una tarjeta y que el número de conexiones externas no rebase el número de patas de los conectores que comunican a las tarjetas. La función que se pretende minimizar es el número de subsistemas [Kodres 72].

El problema puede plantearse como un problema de programación entera cero-uno, pero para casi cualquier caso práctico, el tiempo de solución del problema en estos términos es prohibitivo. Se ha desarrollado entonces, un número considerable de algoritmos heurísticos que en muchas ocasiones producen si-no la partición óptima sí una muy cercana.

Un tipo de estos algoritmos consisten en elegir el componente eléctrico con más conexiones externas. A continuación se agrupan gradualmente los componentes que tengan más conexiones al componente original, hasta que se sobrepase el tamaño de la tarjeta o el número de patas de los conectores. En este momento se retira del conjunto el elemento que hizo que se violaran las restricciones y se procede a aplicar los mismos pasos para una siguiente tarjeta.

Este tipo de algoritmos es sencillo, pero tiene la desventaja de que el proceso que determina las particiones agrega un elemento a la vez, por lo que frecuentemente produce lejanas a la óptima. Además, en las etapas finales de la partición, el método tiende a generar pequeñas "islas" que no están muy conectadas entre sí. Otra clase de algoritmos provee un crecimiento más uniforme en los subcircuitos, por lo que genera soluciones más satisfactorias. Se establece de antemano el número de tarjetas y se asocia a cada una los elementos que ocupen más área. El crecimiento es similar a los algoritmos anteriores, excepto que se realiza simultáneamente en todas las tarjetas.

La mayoría de estos algoritmos restringen el problema a que se satisfaga una condición de *tamaño* (además de la del número de conexiones externas), pero existe la posibilidad en el caso de emplear tarjetas impresas, de que el algoritmo obtenga subconjuntos de elementos que sí cumplan con ella, pero que no haya manera de *acomodar* los componentes en la tarjeta debido a que se están pasando por alto restricciones como la separación entre los alambres. El problema de la partición de circuitos está muy relacionado con el de colocación de los componentes dentro de las tarjetas, pero generalmente se tratan como problemas separados. Las técnicas actuales dejan todavía qué desear, por lo que es un problema que amerita estudio.

La colocación de los componentes es un problema que está relacionado con otros

tres problemas que han recibido mucha atención: el de asignación, el del agente viajero y el de asignación cuadrática. El problema de asignación es un caso particular del de asignación cuadrática que a su vez es un caso particular del de colocación de componentes [Hanan 72]. No existe ningún método exacto que sea computacionalmente práctico para resolver el problema del agente viajero (que puede verse como un problema de asignación restringido) para más de 20 ciudades; y mucho menos para el problema de asignación cuadrática.

El problema de colocación se define como encontrar la colocación óptima de los componentes (o módulos) con respecto a alguna norma definida en términos de las conexiones, como por ejemplo mínima longitud *pesada* de alambre. En realidad existen varios objetivos que deben satisfacerse como eliminar la interferencia entre las señales y preservar la disipación de calor dentro de determinados límites, pero para propósitos prácticos es imposible considerar todos ellos, y una buena función a minimizar que hasta cierto punto abarca a algunos de ellos, es la longitud pasada de alambre. El peso de las conexiones debe ser proporcionada por el usuario. Los algoritmos generalmente emplean una medida rectilineal de la distancia en lugar de la distancia euclidiana; es decir, la distancia entre dos puntos es la suma de las longitudes de los catetos del triángulo rectángulo que forman y no la hipotenusa.

Los algoritmos de colocación se dividen en dos clases: algoritmos de colocación inicial constructiva y colocación-mejoría iterativa. Hanan [Hanan 72] describe varios de ellos.

El ruteo de las conexiones en dadas la lista de conexiones de un circuito y la *colocación* de los módulos, encontrar las trayectorias de los alambres sujetas a determinadas restricciones [Akers 72]; por ejemplo que la separación mínima entre conexiones sea de 25 milésimas de pulgada (esto es estándar), que la longitud máxima de las pistas sea de 15 cm., y que el número de capas sea no mayor a 5.

La mayor parte de los algoritmos usados son variantes o extensiones del algoritmo de Lee [Lee 61]. El método estriba en establecer una cuadrícula con la separación mínima de las pistas sobre la tarjeta (en la que previamente han sido colocados los chips). Para cada pareja de puntos que deben conectarse se elige uno de ellos (que llamaremos A y se calcula incrementalmente la distancia rectilínea de los puntos sobre la cuadrícula más cercanos a A hasta llegar al punto destino, respetando conexiones anteriores y elementos eléctricos. A continuación se escoge la trayectoria que arrojó la distancia menor. El algoritmo puede generalizarse a tres dimensiones y se acostumbra modificarlo para que produzca "buenas" rutas y no ocurra que al decidir la ruta de una pista aparezcan posteriormente dificultades para generar las rutas del resto de las conexiones. Para un excelente resumen de los algoritmos para ruteo consúltese [Akers 72].

Existen algoritmos más eficientes que el de Lee, por ejemplo [Rosa 70]. En lugar de generar punto por punto de las trayectorias, se producen rectas ocasionando que se encuentre la ruta adecuada con menos cálculos.

1.1.5 Diagrama de bloques

Ahora que hemos descrito los programas principales de un sistema para diseño de circuitos podemos establecer otro flujo de información entre el modelo físico y el simulador. Un simulador sofisticado deberá considerar aspectos como la longitud de las conexiones, sobre todo si se trata de compuertas muy rápidas (STTL y ECL).

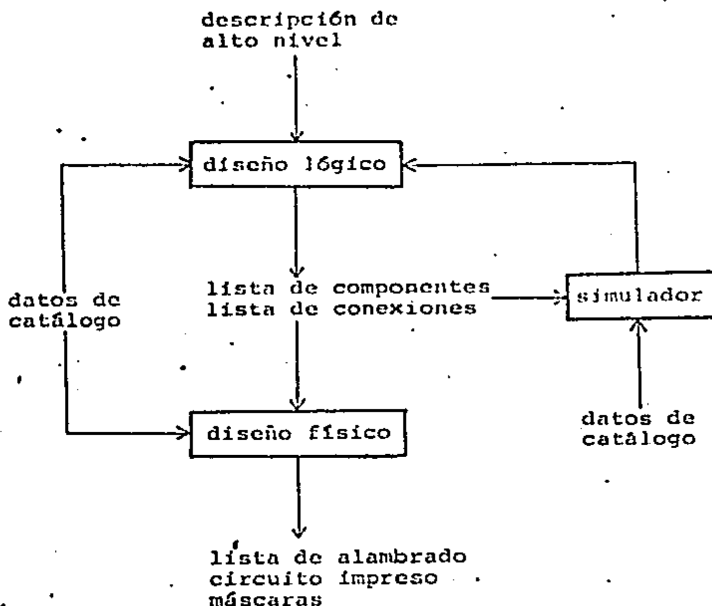


Figura 1.1. Diagrama de bloques de un sistema auxiliar en el diseño de circuitos

Además, es conveniente mencionar el hecho de que el sistema para diseño vertirá sus resultados a un sistema de manufactura. Sea éste automático o no, creará una realimentación más. A través de él, se formará un lazo entre la salida del paquete de diseño físico y el de diseño lógico. Una vez constuido el circuito, aparecerán cambios que, aunque es común que solo se efectúen (erróneamente) sobre el prototipo, deben actualizarse en el modelo lógico; de otra manera el peligro de tener información inconsistente en el diagrama lógico es muy grande.

Independientemente de que el sistema para diseño posea un programa para diseño físico, el programa para diseño lógico debe ser suficientemente flexible para albergar las modificaciones hechas ya sea durante el diseño físico o a causa de pruebas al circuito ya construido. Poca utilidad tiene, por ejemplo, un sistema que numera las patas de compuertas, si no presupone que el circuito puede estar a medio numerar debido a que desde la vez anterior que se numeraron las patas, algunas compuertas desaparecieron y otras se incluyeron. Como conclusión, el programa pra diseño lógico debe elaborarse tomando en cuenta su comunicación con varios sistemas, ya sean automáticos o manuales. La fig 1.1 muestra el diagrama de bloques del sistema para diseño físico y lógico.

1.2 Objetivos del programa

El sistema debe ser capaz de modelar circuitos, por lo que es factible aplicarlo al diseño de circuitos integrados, circuitos a nivel de tarjetas o incluso sistemas de potencia y en general sistemas que empleen una notación gráfica a los que se asocie una red entre sus componentes. No obstante, habrá funciones especializadas en el tipo de sistemas a los que esté orientado el programa. Se decidió emplear el programa en el diseño de circuitos a nivel de tarjetas sin que ello lo restrinja a modelar sistemas similares.

De los tres niveles de abstracción distinguibles en la entrada a los programas para diseño de circuitos, se estableció el nivel de componentes. Es decir, que el usuario debe especificar al sistema todos los elementos primitivos del circuito, que serán circuitos integrados, resistencias y capacitores. Aunque se haya elegido el nivel más bajo para la descripción de los circuitos, el sistema debe se capaz de estructurar la información de manera que el usuario pueda utilizar bloques (definidos por él mismo), como si fueran elementos primitivos. Es decir, que a pesar de que el usuario deba indicar todos los elementos del circuito, el sistema tendrá el poder expresivo para que el diseñador suba el nivel del lenguaje descriptivo.

El sistema no realizará ninguna labor sintética de circuitos debido a que la mayor parte de los componentes serán circuitos de mediana y alta integración. Cuando los elementos de una tarjeta eran únicamente circuitos de baja integración, sí se consideraba útil el empleo de algoritmos generadores de circuitos a partir de una descripción más abstracta que la buscada. Pero actualmente la decisión de qué circuitos integrados incluir en una tarjeta está regida por otras leyes que rara vez son la minimización del número de compuertas. Las razones en las que se basa un diseñador actual al elegir los chips se basan más bien en el catálogo del distribuidor. Toma en cuenta la disponibilidad de los elementos y su objetivo radica en la minimización del número de chips. Intenta identificar circuitos parecidos a los que necesita sin importar cuántos transistores desperdicia.

Por supuesto que sería deseable que estas decisiones fueran tomadas automáticamente por el sistema diseñador de circuitos, pero todavía no se conoce lo suficiente de inteligencia artificial como para lograr este nivel de sofisticación. Es difícil elaborar un programa que elija entre una memoria ROM y un multiplexor, o prefiera un microprocesador 68000 sobre un Z8000 por ejemplo. El problema es complejo porque implica la toma inteligente de decisiones basada en una gran cantidad de información. Se investiga en este sentido [Lafue 82]. Una alternativa prometedora realiza la simbiosis entre los manejadores de bases de datos y los sistemas llamados expertos (que contienen algoritmos de inteligencia artificial) con objeto de aprovechar las ventajas de ambos [Lafue 82]. Cabe esperar en un futuro próximo sistemas capaces de sintetizar circuitos a nivel de tarjetas de manera similar a como lo hacen los diseñadores humanos hoy en día.

Otra característica que debe tener el programa es poder funcionar en una computadora chica, digamos con un microprocesador Z80 y 64 Kbytes de memoria. Esta restricción se impuso más bien por razones de disponibilidad y tuvo consecuencias en lo que respecta al lenguaje escogido para su programación. Sin embargo, también repercutió en la complejidad que debería poseer el programa.

Expusimos tres módulos fundamentales para los programas diseñadores de circuitos, pero ninguno de ellos es indispensable. De hecho, existen sistemas formados exclusivamente por el simulador o por el subsistema que se encarga del diseño físico. Se consideró más útil

un programa que modelara el circuito desde el punto de vista lógico. Independientemente de la tecnología que se use para construir el circuito, ya sea alambre enrollado (wire-wrap) o circuito impreso o tabletas de prueba (bread board), el diseñador realiza el diagrama lógico. Un programa que ayuda a elaborar el diagrama lógico es útil en cualquiera de los casos anteriores.

En cuanto a la salida del programa, lo más natural es que consista en la lista de conexiones[3] pues efectivamente es una traducción de notación, que por costumbre el diseñador realiza directamente sobre el diagrama lógico, pero que por lo mismo es una fuente de errores. La lista de conexiones podrá en el futuro ser alimentada a otro programa que sea más o menos independiente del de diseño lógico que ayude al diseño físico.

La lista de conexiones consiste en un conjunto de puntos eléctricos. Cada punto, a su vez está formado por un conjunto de parejas que contienen el número de la pata del chip conectada a ese punto y el número del chip al que pertenece la pata. Dadas estas características es deseable que el programa numere las patas y los chips por sí mismo. Si los bloques de diseño efectivamente son chips completos, este proceso es trivial. Sin embargo, al usar chips de baja integración, y algunos de mediana integración, los elementos de diseño frecuentemente son *partes* de circuitos integrados (compuertas, por ejemplo), de manera que no todos los elementos del mismo tipo (número de catálogo) poseen la misma numeración en las patas ni números de chips distintos. Al utilizar compuertas, es necesario agruparlas en circuitos integrados, pues se fabrican varias dentro de un chip. Todas las que quepan en un circuito integrado tendrán asignado el mismo número de chip y diferentes numeración de patas.

Para que el sistema auxiliar agrupe compuertas en chips requiere cierta información de catálogo. Estrictamente hablando, al dotar al programa de la capacidad de numerar patas y chips, su labor deja de ser una mera traducción y sí efectúa determinado procesamiento de la información que describe el circuito. Al contener la numeración de las compuertas, es capaz de contestar a la pregunta relativa a cuáles compuertas existen dentro de chips ya incluidos en el circuito pero que todavía no han sido conectadas. Cuando un circuito ya construido sufre cambios, el diseñador se hace con frecuencia esta pregunta e invierte una cantidad considerable de tiempo buscando compuertas libres.

[3] La lista de conexiones es diferente a la lista de alambrado. La primera se refiere a la salida de un programa para diseño lógico mientras que la segunda implica coordenadas de las conexiones pues es el resultado de un sistema para diseño físico.

Capítulo 2

Diseño

2.1 Paquete de graficación

2.1.1 Justificación para un paquete interactivo

Este capítulo describe el diseño del sistema en base a los objetivos establecidos. El capítulo 4 explica cómo se implantó el sistema, qué objetivos se satisficieron y cuáles no.

Al plantear los objetivos del programa auxiliar en el diseño de circuitos se estableció el nivel de abstracción de los datos de entrada, pero no se predispuso al empleo de entrada gráfica o una notación similar a la de los lenguajes de programación. Sin embargo, si la descripción de los circuitos consiste solamente en los componentes que lo forman y en sus conexiones, se trata de una especificación no procedural (pues no depende del orden lexicográfico de las instrucciones). Debido a que las imágenes también son no procedurales y a la inercia que existe entre los diseñadores de circuitos en la utilización de diagramas lógicos se decidió desechar el tipo de lenguajes descriptores de circuitos mencionados en el capítulo anterior para utilizar una interfaz gráfica. En sistemas para diseño de circuitos más generales, que acepten secuencias de microoperaciones, sí será conveniente incluir una notación parecida a la usada actualmente para describir algoritmos, con extensiones en los tipos de datos como registros y en la instrucciones como transferencia de información.

Lo más común en el empleo de paquete de graficación es que se describan las imágenes desde un programa escrito en un lenguaje de propósito general. El nombre "paquete" indica un conjunto de subrutinas especializadas con las que se extiende el lenguaje original. Esta metodología, que es popular también en los manejadores de bases de datos, es adecuada para aplicaciones complejas que efectivamente ameriten las facilidades brindadas por un lenguaje de

programación como la evaluación de expresiones, estructuras de control de flujo y estructuras de datos. No obstante, es fácil desprestigiar a estos paquetes de graficación cuando se emplean en aplicaciones en las que la síntesis de la imagen no depende de un algoritmo complejo.

Si se emplea un paquete no interactivo para describir gráficamente un circuito, por ejemplo, la síntesis de las imágenes se vuelve viscosa pues el programador debe llamar desde su programa las subrutinas del paquete a la vez que construye mentalmente la imagen que desea crear. Por la misma razón es difícil leer un programa escrito de esta manera y si se desea depurar un programa o modificar la imagen, es necesario localizar el punto dentro del programa fuente escrito en el lenguaje de propósito general, que corresponde con la parte de la imagen a corregirse.

Aunado a lo poco explícito de las imágenes que emplean paquetes de graficación no interactivos, está la compilación de la imagen: casi cada cambio que sufre la imagen implica una nueva compilación, de manera que la evolución de la imagen conforme se construye degenera en lo que caen la mayoría de los lenguajes de programación; es decir, editar el programa, para después compilarlo, una vez concluido lo cual ejecutarlo para examinar su comportamiento con objeto de volver a editar, y así sucesivamente hasta depurarlo. Esto es en el mejor de los casos, ya que si aparecen errores de programación (bugs) hay que insertar instrucciones que impriman los valores intermedios de las variables o recurrir a un depurador (debugger). El empleo de lenguajes conversacionales como Basic y APL elimina este defecto, pero no salva al usuario de tener que buscar errores de programación en su programa ejecutándolo mentalmente.

Es importante liberar al usuario de los defectos que caracterizan a los lenguajes convencionales cuando son empleados en la generación de imágenes y ponerlo en contacto con la imagen misma en lugar de que interactúe con un listado. El usuario debe poder olvidarse de la manera en que el sistema gráfico representa internamente la información, si se desea un lenguaje gráfico de nivel suficientemente alto. Debe también recibir una realimentación inmediata de la imagen que está creando y estar habilitado para modificarla con facilidad.

Aunque existen algoritmos gráficos en los que la imagen sí depende del orden en que se sinteticen sus partes, generalmente se trata de algoritmos para eliminación de líneas y superficies, como el algoritmo del pintor [Newman 79]. Al hablar de un sistema para crear diagramas lógicos de circuitos, no solo estamos restringiendo las imágenes a que sean no procedurales sino que también a que tengan un número *constante* de elementos (no es importante poder variar el número de transistores, por ejemplo, con solo alterar un parámetro del dibujo). Aplicaciones en las que sí es conveniente un paquete no interactivo abarcan la generación de curvas recursivas como la curva "C" [Winston 81] o las curvas de Hilbert [Wirth 76] porque el número de elementos de la imagen depende de parámetros externos y forzosamente se requiere hacer uso de la instrucción condicional (IF) al generar la imagen.

El hecho de que las imágenes de nuestra aplicación sean constantes sugiere la elaboración de una máquina virtual (un intérprete) cuyas instrucciones consistan en los objetos que foman las imágenes. Los diagramas lógicos de circuitos se pueden describir en base a segmentos de rectas y cadenas de texto. En principio, el intérprete gráfico poseerá estas dos instucciones en su lenguaje. Las imágenes se representarán con "programas" escritos en este lenguaje. Debido a la sencillez del lenguaje, es factible que el usuario no interactúe con el listado del programa sino con el *resultado*. Esta es precisamente la diferencia entre un paquete de graficación interactivo y uno que no lo es.

Un paquete interactivo se comporta de manera similar a un editor de texto, con la

diferencia de que en lugar de que el usuario señale entidades sobre el listado fuente, se refiere a los efectos ocasionados por su ejecución, suprimiendo entonces que el usuario tenga que ejecutar mentalmente su programa. ¿Qué tan complejo puede ser el lenguaje del procesador gráfico para que todavía el usuario sea capaz de "editar" su programa refiriéndose a sus resultados? No mucho. Si se decide incluir instrucciones condicionales, por ejemplo, ya no se posee toda la información del algoritmo con solo examinar sus resultados. En general podemos afirmar que para que un paquete de graficación sea interactivo, debe existir una correspondencia uno a uno entre las instrucciones del programa y los objetos gráficos.

De acuerdo con la descripción que hemos dado del paquete de graficación, podría parecer que se está duplicando el funcionamiento de los dispositivos llamados *procesadores de despliegue* (display processor units). Estos dispositivos gráficos consisten en procesadores (construidos en circuitería) especializados en la generación de imágenes y su lenguaje de máquina está formado entre otras instrucciones por las que producen en un monitor CRT los elementos gráficos primitivos, digamos rectas y texto. La síntesis de las imágenes se efectúa barriendo en forma dirigida el haz del monitor, es decir, que para que la imagen se genere constantemente sobre la pantalla el programa debe terminar con un brinco al inicio del código. Además de brincos, es común que el procesador posea instrucciones para llamar subrutinas escritas en el mismo lenguaje. Si la descripción de un segmento de la imagen está hecha en base a coordenadas relativas y se forma una subrutina con él, se podrá repetir el segmento gráfico dentro de la imagen al llamar la subrutina adecuada.

Sin embargo, como hacen notar Foley y van Dam [Foley 82], lo más frecuente en los procesadores de despliegue, es que se salten pasos en la generación de la imagen. Al producir una imagen a partir de una descripción formada por instrucciones de la naturaleza descrita, se acostumbra realizar secuencialmente determinados procesos como la aplicación de transformaciones a la imagen y recorte de los elementos gráficos. El recorte sirve por ejemplo para asegurar que en la pantalla solo se genere la porción visible de la imagen. En un procesador de despliegue que no recorta, los puntos cuyas coordenadas los colocan fuera de la pantalla, ocasionarán un sobreflujo en los registros que determinan la posición del haz del monitor haciendo que sí aparezcan en la pantalla, pero como si ésta fuera un cilindro o una esfera.

Si el dispositivo de despliegue es capaz de almacenar una descripción de la imagen con instrucciones gráficas consistentes en rectas, por ejemplo, pero no realiza el recorte, de todas maneras es necesario almacenar otra descripción de la imagen y procesarla por programa, para generar el archivo gráfico destinado al procesador de despliegue.

En otras palabras, el empleo de un procesador de despliegue en un paquete de graficación interactivo, no implica el librarse de tener que programar otra máquina virtual (que sí realice todos los pasos necesarios para sintetizar la imagen) debido a que comúnmente la descripción que contiene el procesador de despliegue no es suficientemente abstracta. Recientemente han aparecido dispositivos que efectivamente almacenan el archivo gráfico antes de la aplicación de transformaciones y del recorte [Foley 82].

2.1.2 Modelo de datos

Se mencionó que para dibujar diagramas de circuitos basta con poder producir

rectas y cadenas de texto. Pero ¿debe haber alguna relación entre los elementos gráficos? Los diagramas lógicos contienen frecuentemente símbolos que se repiten, por lo que es en extremo útil que el usuario no tenga que dibujar cada símbolo que requiere, sino que baste con que haga referencia a él, dentro de una biblioteca de símbolos. Existen programas para diseño de circuitos que proveen esta facilidad, pero que al momento en que el usuario escoge un componente eléctrico, el símbolo asociado a él, se *funde* en el diagrama del circuito.

Un ejemplo de estos sistemas es SUDS [SUDS 80]. Evidentemente representa una ventaja sobre los paquetes que requieren la especificación de cada recta independientemente, pero todavía mejor sería que el sistema conservara el conjunto de rectas del símbolo como una entidad. Es decir, que mientras más alto sea el nivel de la descripción de la imagen, mejor. Al almacenar los símbolos como entidades, el usuario puede realizar modificaciones sobre la imagen más rápidamente: puede borrar o mover de lugar todo un símbolo con un solo comando, pues de otra manera tiene que referirse a todas las rectas que lo forman.

La relación que exista entre los elementos gráficos puede ser de varios tipos. Los modelos de datos más frecuentemente usados son el jerárquico, el reticular y el relacional [Kroenke 77]. El modelo jerárquico organiza la información en *registros* con los cuales forma conjuntos, que a su vez pueden constituir conjuntos de más alto nivel. Se restringe la estructura a que cada entidad (ya sea registro o conjunto) pertenezca solamente a un conjunto. El modelo reticular es una generalización del jerárquico, que sí permite que las entidades pertenezcan a más de un conjunto a la vez. El modelo relacional, por otro lado, representa la información con *relaciones* en sentido matemático, es decir, conjuntos de eneadas, en los que cada eneada está asociada a una entidad. Además, existen cinco operaciones básicas que son cerradas: toman relaciones como argumentos y producen relaciones también. Las operaciones sobre los datos de los modelos jerárquico y reticular no son cerradas pues permiten efectos laterales.

Al elegir el modelo de datos para el paquete de graficación, se decidió emplear el modelo jerárquico pues el reticular parece demasiado general para imágenes. Un paquete de graficación que estructure la información gráfica en una red, permitirá al usuario referirse a una entidad de varias maneras distintas, según el conjunto al que pertenezca. Sin embargo, será poco frecuente que se desee que determinada compuerta, por ejemplo, pertenezca a dos flip-flops al mismo tiempo. Si se pretende desplazar de lugar todas las compuertas de un flip-flop en un paquete que emplee el modelo jerárquico, se podrá hacer referencia al conjunto que forman.

El modelo relacional se excluyó debido por un lado, a que al crear una imagen, sí son deseables las operaciones con efectos secundarios que alteran la información y por otro a que los segmentos gráficos estarán formados por entidades de diferente naturaleza (digamos rectas y texto), que no es posible representar en una sola relación. No obstante, existen paquetes de graficación que utilizan el modelo relacional, ver por ejemplo [Foley 82].

Pareció más adecuado entonces, un modelo jerárquico, en el que las instrucciones de la máquina virtual constituyen los registros. Debido a que hemos organizado los datos en forma arbórea, la máquina virtual debe contener una tercera instrucción que indique que ciertos registros no se tratan de objetos gráficos primitivos sino de nodos intermedios. Cuando un sistema de graficación organiza las imágenes en un modelo jerárquico, los conjuntos de entidades reciben el nombre de *segmentos*. Si solo permite un nivel de jerarquía (los conjuntos no pueden a su vez pertenecer a otro conjunto), el sistema se denomina *paquete segmentado* y si almacena varios niveles jerárquicos, es llamado *paquete estructurado*. A la descripción de

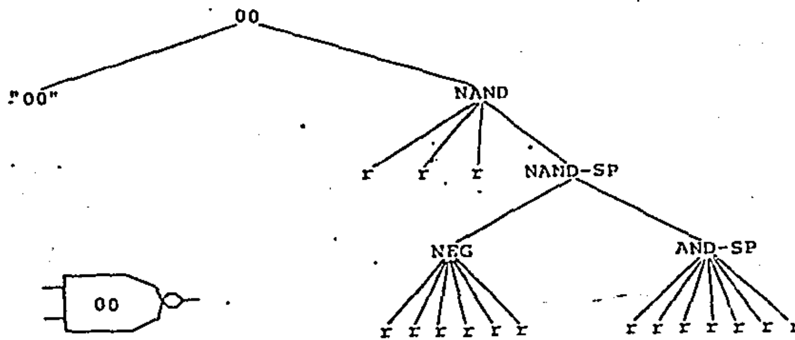


Figura 2. 1 Estructura del segmento 00

la imagen en términos de instrucciones del procesador gráfico, ya sea este virtual o no, se le conoce como *archivo gráfico* [Newman 70].

Se optó por un sistema con archivos gráficos estructurados pues si se posee un lenguaje de programación que permita definir subrutinas recursivas, el procesamiento de estructuras recursivas como lo son los árboles que forman las imágenes estructuradas no representa casi ninguna dificultad adicional sobre las imágenes segmentadas, obteniendo más generalidad. La fig 2.1 muestra la estructura de la compuerta asociada al nombre 00. Además, el estructurar una imagen jerárquicamente va de acuerdo con el objetivo del sistema planteado en el capítulo anterior de que el sistema permita al usuario construir bloques eléctricos a partir de componentes más primitivos y utilizarlos como elementos de diseño. Es conveniente recordar que de cualquier manera se requiere que el sistema contenga un modelo reticular para representar las conexiones del circuito. Sin embargo, las conexiones se almacenan en una estructura diferente al modelo jerárquico de las imágenes debido a que se trata de datos con distintas características. Quizás de haber elegido un solo modelo de datos tanto para la información gráfica como las conexiones habría significado menos trabajo de programación, pero definitivamente habría ocupado más memoria.

2.1.3 Unicidad de segmentos

Los dibujos técnicos contienen una redundancia considerable. Hay símbolos que aparecen muchas veces en un diagrama eléctrico, y vale la pena que el sistema almacene los datos en forma más compacta que como aparecen en el dibujo. Con este objetivo en mente se diseño la instrucción del procesador gráfico que indica que se refiere a un nodo intermedio del árbol, de manera similar a la llamada de subrutinas en un lenguaje de programación. Es

decir, que todas las veces que aparece un símbolo dado en el diagrama se almacenan con una referencia al segmento gráfico pertinente sin que se duplique su contenido. Se dice, entonces, que el archivo no contiene segmentos, sino *instancias* de segmentos. Esta característica, a la que llamaremos "unicidad de segmentos", tiene dos consecuencias importantes.

1. La cantidad de memoria requerida para almacenar el archivo gráfico se reduce notablemente. Dependiendo de la redundancia del diagrama lógico, ocupa entre 10 y 100 veces menos memoria. O sea, que la unicidad de segmentos puede significar el hecho de que el programa para diseño de circuitos quepa o no en una microcomputadora.

2. Los cambios efectuados en un segmento se realizarán automáticamente en *todas* las instancias de ese segmento.

Esta consecuencia puede ser deseable en ciertos casos y en otros no. A fin de que el usuario se encuentre con la libertad de alterar solamente una instancia de determinado segmento o todas sus instancias, es conveniente que el sistema tenga la capacidad de añadir al dibujo en proceso de creación, no solamente referencias a segmentos, sino copias completas, que ocupen otro lugar en la memoria, sin que esto implique que se aplane la estructura gráfica.

Cuando el usuario se encuentra dibujando un circuito, eventualmente deseará eliminar o mover de lugar un segmento gráfico. Para ello, señalará el segmento colocando el cursor o una pluma de luz sobre él. A continuación, el sistema debe mostrar al usuario el segmento señalado dibujándolo de otro color, por ejemplo, o con líneas punteadas si el dispositivo no es cromático. En realidad, el usuario se estará refiriendo a un subárbol de la imagen. ¿Qué nivel del árbol es el que desea señalar el usuario? En el paquete interactivo de graficación elaborado por Sutherland [Sutherland 63], el sistema muestra al usuario el segmento al nivel más abajo posible y contiene un comando para subir de nivel, que muestra el conjunto inmediatamente superior al que pertenece el segmento mostrado originalmente. En nuestro paquete de graficación se optó, por simplicidad, que el sistema mostrara únicamente el nivel superior del árbol, es decir, el descendiente inmediatamente inferior al segmento constituido por todo el dibujo, sin importar si existe más de una instancia del segmento. Si el usuario pretende alterar un subárbol de más bajo nivel, debe indicar al sistema su nombre, hacer los cambios como si se tratara del nivel superior, y regresar a la imagen principal, que al dibujarla, ya contendrá los cambios efectuados.

Las instancias de los segmentos se guardan como entidades que contienen la información necesaria para generar los segmentos con ciertas diferencias con respecto a la copia original. Las diferencias consisten en una transformación afín con la que se puede trasladar la instancia o girarla. En un paquete de graficación, las instancias poseen información para aplicar una tranformación arbitraria y representar otras diferencias como el color. Para la aplicación que nos concierne, rara vez será deseable tener instancias de segmentos que no sean trasladadas y giradas, y de hecho, los giros son siempre a ángulos múltiplos de 90 grados.

Al permitir archivos gráficos estructurados, rigurosamente hablando estamos violando el principio de que para que un paquete sea interactivo, el lenguaje del procesador gráfico debe consistir únicamente en instrucciones que permitan extraer toda la información acerca del archivo gráfico a partir de la imagen que genera al ejecutarse. La estructura de la imagen no se puede obtener de la imagen misma. A pesar de ello, no parece ser una desventaja grande si se incluye en los comandos del usuario uno que le permita saber el nombre de los segmentos que ha creado y a qué segmentos hacen referencia.

En resumen, el lenguaje del procesador gráfico está formado por las instrucciones siguientes:

- `recta(x1,y1,x2,y2)`
- `texto(x,y,''...'')`
- `call(segmento,x,y,a)`

2.1.4 Independencia del dispositivo

Una característica que debe buscar todo paquete de graficación es independencia del dispositivo gráfico. Los mismos dibujos que se generaron en un dispositivo de barrido secuencial en medio no permanente por ejemplo, deben poder duplicarse en otro que imprima las imágenes en papel, o aun en otro sistema de computación. El nuevo dispositivo probablemente posea pixels de diferentes dimensiones de manera que si no se realiza una corrección adecuada, la imagen aparecerá deformada.

El concepto más importante que debe tener un paquete para que sea lo más independiente del dispositivo posible es la separación entre el espacio de coordenadas de la aplicación y el del dispositivo. Todas las coordenadas que se almacenen en el archivo gráfico deben estar referidas a un espacio (llamado de aplicación) con pixeles cuadrados, de manera que

1. al generar una imagen, las coordenadas deben transformarse para que se corrija la deformación producida por el dispositivo.
2. cuando el usuario señala una posición sobre la pantalla, las coordenadas del cursor o pluma de luz deben recibir la transformación inversa al de la salida gráfica.

Otro concepto útil para independizar el paquete del dispositivo es el llamado *espacio normalizado*. Los paquetes con espacio normalizado suponen que las dimensiones del dispositivo valen $(1, \alpha)$, en donde $\alpha \leq 1$. De esta manera, no importa si las dimensiones de determinado dispositivo valen (4,3) como es el caso de la mayoría de los monitores CRT o (1,1), el sistema tratará de aprovechar la mayor superficie posible del dispositivo. Sin embargo, esta noción implica el uso de coordenadas de punto flotante. En el paquete de graficación se restringió firmemente el uso de números de punto flotante pues para un sistema que carece de procesador de punto flotante y realiza las operaciones aritméticas por programa, representa un tiempo en la generación de imágenes varios órdenes de magnitud mayor que si se emplean números enteros. Es decir, que el paquete elaborado no contiene el concepto de espacio normalizado. Aun así, si se actualizan los parámetros adecuados es factible transportar el sistema a otro dispositivo. Estos parámetros consisten en la matriz de transformación asociada al dispositivo y las coordenadas del llamado *puerto de visión*, que se explica a continuación.

2.1.5 Transformaciones y recorte

El *puerto de visión* es una región sobre el dispositivo que generalmente está delimitada por un rectángulo no inclinado. Se llama *ventana* a la región correspondiente en el espacio de

aplicación. Es común que el puerto de visión abarque toda la pantalla, pero en las interfaces tipo "menú" frecuentemente el sistema posee varios puertos de visión. De cualquier manera, el procesador gráfico debe generar únicamente la porción visible de la imagen.

Se habló de que el paquete de graficación debe también aplicar una transformación a las coordenadas de las entidades. Lo más natural al agregar el recorte es que se efectúe *después* de haber aplicado la transformación del dispositivo a todas las coordenadas de los objetos gráficos. Sin embargo, se sabe que no forzosamente esta secuencia es la mejor [Newman 79].

Si examinamos con cuidado, veremos que en realidad no existe únicamente la transformación del dispositivo. El paquete tiene que lidiar también con las transformaciones de las instancias; de hecho, cada nodo del árbol en el que está estructurada la imagen contiene una tranformación. Además, es adecuado incluir otra transformación más, llamada transformación *de visión*, que permite al usuario trasladar el puerto de visión sobre el dibujo en caso de que este no quepa en la pantalla, o generar el diagrama a diferentes escalas.

La transformación de visión puede concatenarse con la del dispositivo sin que esto represente ningún problema, al grado que siempre nos referiremos a la composición de ambas transformaciones como la transformación de visión. Las transformaciones de las instancias deben concatenarse también, conforme el sistema genera la imagen y recorre el árbol. Es decir, que al sintetizar una imagen a partir de su archivo gráfico, si la máquina virtual encuentra una instancia de segmento, debe concatenar la nueva transformación con la concatenación de las transformaciones de todos sus ancestros. Una vez que ha dibujado el segmento, el sistema debe restaurar la transformación al valor que tenía hasta antes de haber dibujado la última instancia.

El hecho de transformar las coordenadas del dibujo (por supuesto que no se aplica la transformación a todos los puntos de las rectas pero sí a las coordenadas de sus extremos) antes de recortar, parece adecuado pues es factible concatenar la transformación de la instancia con la tranformación de visión de manera que las coordenadas se transforman una sola vez.

En un paquete de graficación segmentado (un solo nivel), puede representar un ahorro considerable de operaciones el recortar antes de la transformación. En este caso, también se calcula la composición de ambas transformaciones, y además deben antitransformarse las coordenadas del puerto de visión. Así, solamente se transformarán los objetos gráficos que efectivamente sean visibles. No obstante, este método es adecuado cuando los giros son tan solo a ángulos múltiplos de 90 grados debido a que los algoritmos que recortan contra rectángulos inclinados son complejos y contrarrestan la disminución de operaciones que se busca.

La aplicación consistente en diagramas lógicos cae dentro de este caso particular, pero no hay que perder de vista que los archivos gráficos son estructurados. Ello implica que si el sistema pretende recortar antes de aplicar la transformación, debe almacenar, en el peor de los casos, las transformaciones a todos los niveles del árbol. Se puede argumentar que aunque se realice la secuencia inversa (primero transformar), también es necesario almacenar todas las transformaciones intermedias, pero el algoritmo es mucho más sencillo y directo en este caso. Como consecuencia, se seleccionó el método que aplica la transformación antes de recortar. Además, si el paquete se desea emplear en alguna aplicación que requiera giros a una mayor variedad de ángulos, su adaptaptación es en extremo sencilla.

2.1.6 Diagramas de bloques

En el diseño auxiliado por computadora en general, resulta de gran utilidad que el programa muestre el sistema en proceso de diseño a varios grados de resolución. Tal como está diseñado el paquete de graficación para circuitos, sí permite que el usuario examine un subárbol del circuito, pero si se le pide al programa que dibuje todo el sistema, generará el árbol completo, con resolución completa. Los diagramas de bloques consisten en representaciones de todas las partes del sistema, pero carecen de los detalles de cada parte. O sea, son modelos con menos resolución. Al tener estructurado el circuito jerárquicamente, es muy sencillo incluir diagramas de bloques: agreguemos una cuarta instrucción a la máquina virtual, similar a las instancias de segmentos, que no dibujan la instancia de su segmento a menos que el usuario lo indique con algún comando especial.

2.1.7 Coordenadas absolutas

Un número no despreciable de paquetes de graficación describe los segmentos en base a rectas definidas solamente en uno de sus extremos. Para ello, emplea coordenadas relativas al punto en donde terminó la recta anterior. Esta metodología es atractiva pues además de que la descripción de los segmentos requiere menos memoria, es independiente de su posición. De hecho, la mayor parte de los procesadores de despliegue funcionan de esta manera. Al dibujar un segmento, el usuario debe generar una recta "invisible" y posteriormente llamar a la subrutina que contiene el segmento.

Las coordenadas relativas pueden ser adecuadas para paquetes no interactivos, pero al aplicarlas a sistemas en los que el usuario percibe el resultado del programa gráfico en lugar de su listado fuente, se rompe con la regla básica para paquetes interactivos. Al permitir la existencia de rectas invisibles, se está omitiendo una parte importante del archivo gráfico que no es posible deducir de la imagen misma. Por otro lado, el archivo gráfico deja de ser no procedural debido a que sí es relevante el orden de las instrucciones del procesador gráfico. Consecuentemente, es necesario implantar comandos que estaban ausentes si solamente se utilizan coordenadas abrolutas: las rectas invisibles deben poder hacerse visibles en ciertas circunstancias a fin de que el usuario pueda decidir si las modifica o no, y el sistema deberá ser capaz de insertar instrucciones en determinados lugares de los segmentos ya que uno de los extremos de las rectas dependerá de la recta anterior.

Es decir, aunque aparentemente se simplifican las instrucciones gráficas, en realidad no es así, y definitivamente se hace más compleja la labor del usuario pues debe estar conciente del orden de sus instrucciones. El paquete de graficación de SUDS funciona de esta manera, pero las ventajas son muy criticables.

No se pierde nada al emplear coordenadas absolutas debido a que la transformación de la instancia, aunque requiera más espacio de memoria, puede especificar el desplazamiento del segmento al que hace referencia.

2.1.8 Espacio bidimensional

Otra característica del paquete de graficación es que almacena solamente dos coordenadas de los puntos. La generación de diagramas lógicos no requiere tres dimensiones. No obstante, las ideas expuestas en esta sección pueden generalizarse a un paquete cuya aplicación sí las amerite.

2.1.9 Procesador gráfico

El procesador gráfico es un conjunto de subrutinas que se encarga de interpretar las instrucciones del archivo gráfico, es decir, constituye una máquina (virtual en nuestro caso) que ejecuta el programa que genera la imagen. Esta descripción contiene tanto la información gráfica como las conexiones del circuito, pero nos concentraremos en el procesamiento gráfico.

En el paquete existen varios procesadores que recorren la misma imagen, dependiendo del tipo de procesamiento que requiera el archivo gráfico. Por ejemplo, el procesador que genera el dibujo sobre la pantalla, el que recorre la estructura para encontrar el segmento que señala el usuario y el conjunto de subrutinas que almacenan la imagen en el disco. Todos ellos recorren la estructura jerárquica de la imagen en "preorden", es decir, se visita primero la raíz y posteriormente todos los hijos. Dependiendo del procesador, se realizarán diferentes funciones en las hojas del árbol, pero el código que se ejecuta en los nodos intermedios es similar. Se presenta en una notación tipo Pascal ya que se facilita su expresión recursivamente.

```
procedure recorre(a : arbol);
  begin
  while a <> nil do begin
    p(a);
    if hoja(a)
      then q(a)
      else recorre(a.hijo);
    a := a.hermano;
    end
  end
```

a puede ser una referencia a un árbol y la subrutina lo visitará a él y a todos sus hermanos. p(a) significa la visita del nodo mismo, q(a) indica el procesamiento efectuado sobre los nodos de tipo hoja. a.hijo y a.hermano son funciones que regresan el nodo hijo y siguiente hermano de un nodo respectivamente.

Este algoritmo se presta especialmente para ser ejecutado simultáneamente por varios procesadores por lo que se puede programar con facilidad en máquinas paralelas.

Al describir el algoritmo se ha implicado de alguna manera que la estructura jerárquica de la imagen se está representando por medio de un árbol binario en el que cada nodo contiene dos apuntadores (direcciones de memoria); el apuntador izquierdo hace referencia al primer hijo y el apuntador derecho al siguiente hermano. A pesar de que al implantar la estructura

arborescente sí se utilizó esta representación, el diseño del paquete, por definición, es independiente de la implantación. La deficiencia proviene de haber usado Pascal en el algoritmo expuesto. Sin embargo, se pudo haber utilizado un lenguaje de más alto nivel, que no implicara ni siquiera el concepto de apuntador, como por ejemplo [Madhavji 81].

2.2 Paquete de conexiones

2.2.1 Modelo del circuito

Es evidente que además de poder representar circuitos gráficamente, el sistema debe ser capaz de almacenar la información relativa a las conexiones con objeto de generar la lista de conexiones del circuito. Para ello se necesita un modelo de datos reticular. Si se quiere ver en los términos empleados previamente, cada elemento eléctrico (circuito integrado o elemento discreto) pertenecerá a tantos conjuntos como patas tenga conectadas, y cada punto eléctrico representará un conjunto. Como no existe una diferencia básica entre los circuitos integrados y los elementos discretos como resistencias, capacitores y transistores, es decir, ambos contienen varias patas que pueden estar numeradas (en las resistencias y algunos capacitores las patas son eléctricamente idénticas, el sistema se simplifica al tratarlas como si fueran iguales), usaremos la palabra *chip* para denotarlos.

Para modelar las conexiones de un circuito, se emplea frecuentemente gráficas con dos tipos de nodos: uno asociado a los elementos eléctricos y otro con el que se representan los puntos eléctricos [Kodres 72]. Para el caso particular de circuitos con elementos que posean únicamente dos terminaciones, se puede usar una gráfica con un solo tipo de nodos asociado a los puntos eléctricos y los elementos se representan con las ramas de la gráfica. Pero las gráficas denotan siempre relaciones binarias, por lo que no es posible modelar un circuito con elementos de más de dos patas de la misma manera. Al distinguir dos tipos de nodos, el modelo permite representar circuitos más generales. A las gráficas de este tipo se les conoce como gráficas *bipartitas* [Harary 69].

La representación de ambos tipos de nodos es diferente. En los nodos consistentes en elementos eléctricos es muy importante poder distinguir entre sus terminaciones, mientras que en los nodos asociados a los puntos este reconocimiento es innecesario. Existen varias maneras de representar gráficas en una computadora; entre las más frecuentemente usadas están las matrices de adyacencia y de incidencia, y listas ligadas, pero hablaremos de ellas en el capítulo referente a la implantación del sistema.

Es conveniente incluir sentido a las ramas debido a que salvo ciertas excepciones, la información viaja siempre en la misma dirección en los alambres. Excluyendo a los alambres bidireccionales, en los que por medio de compuertas se controla el sentido del flujo de la información y dinámicamente se invierte este, conviene representar los circuitos con gráficas bipartitas bidireccionales. Se les denomina bi-digráficas [Kodres 72].

El hecho de que el modelo consista en una gráfica dirigida, es ventajoso al momento de diseñar el circuito, pues el sistema contiene suficiente información para prevenir al usuario en

caso de que pretenda conectar dos salidas al mismo punto. Por simplicidad en la implantación, se restringió el modelo a que los nodos de tipo punto eléctrico solamente pudieran contener una rama que esté entrando hacia ellos. Esto impide al sistema detectar cortos circuitos en compuertas alambradas, que se forman conectando varias salidas de colector abierto a un mismo punto eléctrico. Ellas se comportan como si todas las salidas de colector abierto entraran al punto eléctrico y todas las demás conexiones salieran de él. No obstante, en general las compuertas alambradas son la minoría.

2.2.2 Relación entre el diagrama lógico y las conexiones

A primera vista, y sobre todo estando familiarizado con el proceso de traducción seguido por los lenguajes descriptores de programas, la tendencia al diseñar un lenguaje con interfaz gráfica es la elaboración de algo equivalente al analizador sintáctico de los compiladores. Se ha hecho costumbre traducir los archivos fuente escritos en lenguajes de programación con compiladores que analizan texto que pertenece a una gramática dada que generalmente es de precedencia simple. Pero ¿por qué se lleva a cabo un *análisis* si en realidad lo que desea el usuario es *sintetizar* un algoritmo? Esto es necesario desde el momento en que el programa se elaboró con un editor de texto de propósito general, que no tiene información relativa a la gramática del lenguaje de programación.

Entre las características que hacen de Lisp el lenguaje de programación más importante que se haya inventado están los *editores de listas*. Estos sistemas hacen posible prescindir del listado fuente del programa debido a que tienen información acerca de la estructura de las listas que constituyen tanto los datos como el código de los programas de Lisp. En lugar de guardar una cadena informe de caracteres (que llamamos programa fuente), se almacena una representación *estructurada* del programa que otorga la capacidad de traducir incrementalmente el programa. Cada vez que el usuario desea un listado fuente, se realiza la traducción inversa de la representación interna. Recientemente se han desarrollado trabajos que extienden este concepto a lenguajes sintácticamente más complejos [Medina-Mora 81] como los lenguajes tipo Pascal.

La misma idea debe aplicarse a sistemas con interfaz gráfica, pues resultaría extraordinariamente complejo emplear un paquete de graficación para construir el diagrama lógico de un circuito y posteriormente analizar la imagen para deducir sus conexiones. En otras palabras, no es importante almacenar el diagrama de un circuito tal como desea percibirlo el usuario, sino que es más adecuado conservar una representación similar al archivo objeto, pero con la información suficiente para reconstruir el equivalente al listado fuente, que en este caso es el dibujo del circuito. Es de interés, en cambio, elaborar el modelo de conexiones incrementalmente conforme el usuario sintetiza el diagrama.

De hecho, lo ideal sería que el diseñador únicamente se preocupara por el diagrama, de manera que al dibujar su circuito, concurrentemente se modelaran sus conexiones; y cualquier cambio que sufra el diagrama quede automáticamente actualizado en el modelo eléctrico. Como se verá más adelante, esta propiedad representa dificultades serias y solo se logró parcialmente.

Para alcanzar este objetivo se estructuró el programa de diseño de circuitos con un modelo del circuito, que contiene tanto sus características gráficas como eléctricas (sus

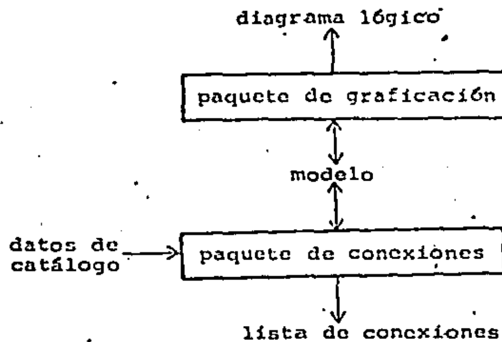


Figura 2. 2 Diagrama de bloques del sistema

conexiones) a partir del cual se genera el diagrama lógico y la lista de conexiones. La fig 2.2 es un diagrama de bloques del sistema. Note que la entrada de información por parte del usuario es a través de comandos del paquete de graficación y el diagrama lógico es una *salida* del sistema, lo que difiere ligeramente del diagrama de bloques mostrado en el capítulo anterior referente a sistemas para diseño de circuitos en general. Ahí se daba a entender que el usuario describía su circuito, probablemente en un lenguaje similar a los de programación, y no se generaba un diagrama lógico.

2.2.3 Extensiones al paquete de graficación

Tal como fue descrito el paquete de graficación, a pesar de que contiene algunas características favorables para nuestra aplicación, deja qué desear. Por ejemplo, dado que el programa supone que el usuario señala a los hijos del nodo formado por el dibujo sobre la pantalla, no hay manera de bajar en la jerarquía de la imagen para referirse a las patas de los chips.

Por otro lado, si se almacena el texto de un diagrama lógico con la instrucción del procesador gráfico descrita, la cantidad de memoria requerida es enorme: cada pata estará numerada, al igual que cada compuerta. Si se guardan como variables independientes las coordenadas de todas las entidades que sean cadenas de caracteres, una compuerta de dos entradas y una salida ocupa no menos de 50 bytes.

Asimismo, ya que el modelo debe poseer además de la información gráfica los datos necesarios para generar la lista de conexiones, requiere poder distinguir entre un texto cualquiera y los que tengan algún significado eléctrico como los referentes a los nombres de catálogo de los elementos y los números de patas y chips.

Pero eso no es todo, debido a la unicidad de segmentos del paquete de graficación, cuando el usuario señala la pata de un chip, por ejemplo, ¿cómo distinguir entre esa instancia y las demás, si todas ellas ocupan las mismas localidades de memoria?

El problema que concierne al señalamiento de las patas de los elementos eléctricos se resuelve incluyendo otro comando al paquete de graficación, similar al que sirve para localizar segmentos, con la diferencia de que en lugar de recordar el descendiente inmediato del dibujo, muestra al usuario la hoja del árbol sobre la cual está el cursor. Ello nos restringue a que las terminaciones de los elementos eléctricos sean elementos gráficos primitivos, pero no se pierde demasiada generalidad, y sí es una regla sencilla.

La compactación de la infomación textual de los diagramas lógicos se resuelve añadiendo instrucciones especiales al procesador gráfico. Ya hemos mencionado que el sistema no hará ninguna distinción explícita entre elemenos elécticos discretos como resistencias, y los circuitos integrados. Pero, debido a que es común que el diseñador utilice compuertas de baja integración, que son fragmentos de circuitos integrados, es importante definir entidades con las propiedades de las compuertas, por ejemplo el hecho de que varias compuertas tendrán asignado el mismo número de chip y que posean diferente numeración de patas.

Esta nueva entidad se puede representar como un registro en forma análoga a las instancias de segmentos, pero con infomación adicional: los números de las patas y el de la compuerta. Las *compuertas* consistirán en la numeración mencionada, además de los datos comunes a las instancias de segmentos. La numeración puede estar codificada, y las coordenadas de los caracteres sobreentendidas. Por ejemplo, el número de la compuerta será dibujado por el procesador gráfico siempre al centro de la compuerta (en el origen relativo al segmento) y los números de patas unos tres pixeles (o los que sean) sobre las rectas que las forman. (Esta regla se aplicará a las patas horizontales, y reglas similares para las verticales.)

No es necesario que el sistema represente en forma distinta los chips de las compuertas. Basta con que contenga la entidad compuerta para que sea capaz de representar chips también, que son casos particulares de compuertas con la característica de que solamente cabe uno en un circuito integrado. Cuando nos refiramos a las entidades del sistema, emplearemos el término compuerta para denotar igualmente compuertas, chips y elementos discretos.

Al haber agregado esta entidad, se ha resuelto otra dificultad señalada anteriormente: la carencia de información relativa a los números de patas que existía en el paquete original. Como estamos almacenando un modelo "intermedio" del circuito, que no es ni el diagrama lógico tal como lo va a percibir el usuario ni consiste únicamente en la información necesaria para que el sistema "entienda" el circuito, el modelo debe contener los datos suficientes para ambas salidas: el diagrama lógico y la lista de conexiones. Para ello, el programa de diseño debe poder distinguir entre los letreros comunes y corrientes y los números con significado eléctrico. Ya que estos últimos se han insertado en los registros asociados a las compuertas, el sistema posee la información faltante.

En lo que toca al texto, quedaría por resolver solamente el que el programa reconozca el nombre de catálogo de las compuertas (que es necesario para numerar las patas de las compuertas pues es la llave con la que el sistema consultará el catálogo). La solución es simple: los segmentos que representen compuertas estarán formados, además de por una serie de rectas y quizás instancias de segmentos, por una sola entidad de tipo texto. El sistema supondrá que este letrero (que será del tipo de texto que ya poseía el paquete de graficación) es el nombre de catálogo de la compuerta.

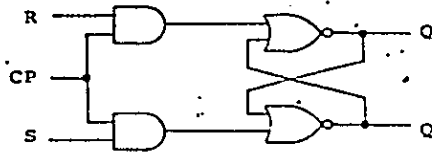
La descripción de las compuertas debe contener, a parte de los datos mencionados, la información que indica qué patas son salidas y cuales son entradas para que al elaborar incrementalmente las conexiones del circuito con una bi-digráfica, el usuario no tenga que comunicar al sistema el sentido de los alambres. Así, extenderemos los tipos de rectas a rectas comunes (que solo tienen significado gráfico), alambres (que son rectas con un sentido asociado) y patas de entrada y de salida. En el caso de que no se desee que el sistema verifique entradas y salidas por tratarse de compuertas de tres estados o de colector abierto, el diseñador podrá declarar sus salidas como si fueran entradas.

2.2.4 Circuitos con unicidad de segmentos

El problema más trascendente que apareció durante el diseño del programa radica en representar circuitos dentro de una estructura jerárquica conservando la unicidad de segmentos.

El paquete de graficación permite estructurar jerárquicamente las imágenes con la ventaja de que al modificarlas, el usuario hará referencia a pocas entidades del archivo gráfico. De hecho, si la imagen está bien estructurada, bastará con modificar un solo segmento (subárbol). Es decir, que la estructura que posea laimagen deberá ser escogida por el diseñador. Si este decide no aprovechar los segmentos y organiza el diagrama del circuito como una sola entidad gráfica, las alteraciones que sufra el dibujo durante la creación del archivo gráfico serán tardadas. Por ejemplo, si el usuario pretende desplazar una parte del circuito, se verá obligado a repetir el cambio en todos los elementos primitivos que constituyen el fragmento pertinente del diagrama. O si cambia el símbolo de un elemento eléctrico, tendrá que modificar una por una todas sus instancias. Consecuentemente, es el usuario el que debe construir una "buena" estructura agrupando en segmentos las entidades que sean semánticamente similares.

Al mencionar los objetivos del programa se subrayó la conveniencia de que también existiera una jerarquía al nivel de subcircuitos; no solo porque se presta para realizar modificaciones gráficas, sino porque el sistema permite "subir" de nivel el lenguaje descriptor del circuito. Debido a ser un sistema que requiere del usuario los componentes del circuito, si no se provee un mecanismo que defina bloques de diseño a partir de elementos más primitivos, el programa puede resultar muy primitivo. Parece adecuado traslapar la estructura gráfica con la eléctrica. El árbol gráfico tendrá como elementos primitivos rectas y texto, mientras que la jerarquía eléctrica tendrá compuertas en sus hojas, pero a partir del nivel de compuertas, hacia arriba, ambas estructuras pueden ser idénticas. La fig 2.3 es un ejemplo de la estructura del diagrama



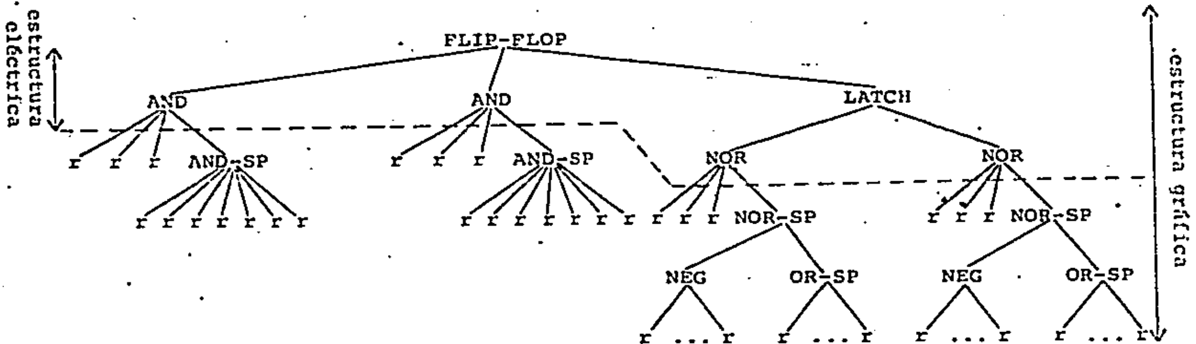


Figura 2.9 Traslape de las estructuras gráfica y eléctrica de un circuito.

En la estructura gráfica resulta especialmente útil almacenar en un solo lugar de la memoria las instancias de los segmentos, indicando en el registro de cada instancia las diferencias con respecto a la copia original. Las diferencias consisten en el desplazamiento y el giro, y numeración de patas en caso de ser instancia de compuertas. Las ventajas son ahorro de memoria y cambios automáticos a todas las instancias.

¿Será también conveniente la unicidad de segmentos en la estructura eléctrica? Si el circuito consiste en un registro de 64 bits, por ejemplo, es ventajoso conservar la unicidad de los 62 bits intermedios (suponiendo que los de los extremos difieran considerablemente), a fin de obtener las mismas ventajas logradas en el paquete de graficación. Y si el circuito posee digamos 16 de estos registros, además de que la cantidad de memoria necesaria para representar el circuito será varios órdenes de magnitud menor que si se conserva la redundancia, el tiempo en que se realizan las modificaciones por no tener que repetirlas en todos los bits de los registros disminuye notablemente.

El sistema, tal como ha sido descrito, es incapaz de estructurar el circuito eliminando redundancia con unicidad de los subcircuitos. Supongamos que ya existe algún mecanismo para representar bi-digráficas. El programa modelará circuitos a nivel de compuertas, e incluso circuitos estructurados. Podrá con ciertas extensiones sencillas establecer módulos eléctricos a partir de componentes y hacer que el usuario los emplee como si fueran elementos eléctricos primitivos. Las extensiones necesarias para ello consisten en:

1. que automáticamente cuando el diseñador incluya un nuevo módulo (no primitivo) en su

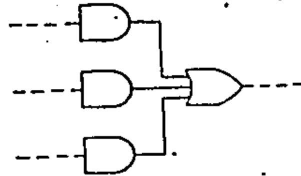


Figura 2.4 Módulo con conexiones internas y externas

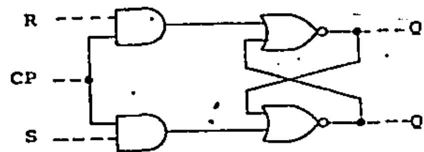


Figura 2.5 Módulo con conexiones internas-externas

circuito, se realicen determinadas conexiones que no varían de una instancia a otra.

2. que se copie la estructura del subcircuito original en las instancias

No obstante, ya que existen algunas conexiones que difieren entre las instancias del módulo, el sistema tendrá que duplicar el módulo en la memoria.

Es imperativo que el programa represente jerárquicamente los circuitos con baja redundancia debido a que, como se sugirió anteriormente, mientras mayor sea el contenido semántico del modelo que posee el sistema, más funciones se podrán automatizar. Si se permiten varias copias del mismo módulo eléctrico en la memoria, equivale a bajar el nivel de abstracción del modelo del circuito; pues se parecerá más a cómo percibe el usuario el diagrama lógico por un lado, y por otro el programa estará impedido para reconocer si dos módulos son parecidos y menos en qué varían.

La solución parte de identificar dos tipos de conexiones en los módulos eléctricos: las externas al módulo, que son las que difieren de una instancia a otra y las internas, que permanecen constantes. La fig 2.4 representa un módulo eléctrico del cual supuestamente existirán varias instancias en el diagrama del circuito. Las conexiones externas están dibujadas con líneas punteadas. Se pueden presentar puntos eléctricos que estén conectados a varias patas internas y que sin embargo difieran entre instancias, como por ejemplo en el módulo de la fig 2.5. Estos casos se modelarán como si fueran dos puntos eléctricos diferentes, uno externo y uno interno, y para propósitos de generación de la lista de conexiones se indica en algún lugar que en realidad constituyen el mismo punto.

Pero no basta con hacer esta distinción en las conexiones. Cuando se describió la

representación del circuito con una bi-digráfica se implicó que estaría formada por identificadores *globales*, ya fueran éstos apuntadores o números asociados a cada nodo de tipo punto eléctrico. Sin embargo, ahora ya no es válido emplear identificadores globales para las conexiones debido a que para las conexiones internas, por ejemplo, los puntos eléctricos serán distintos en cada instancia del módulo, a pesar de que ocupen la misma localidad de memoria en todas ellas.

La modificación que debe realizarse en el paquete de conexiones para conservar la unicidad de subcircuitos radica en la utilización de otro tipo de identificadores en las conexiones que no sean globales, por ejemplo, la *trayectoria* que se recorre en el árbol a partir de la raíz para localizar la conexión. Este identificador puede consistir, digamos, en la concatenación de los identificadores globales de todos los ancestros del nodo.

Las conexiones externas a un módulo deben permanecer indefinidas en la definición del módulo, y su valor debe establecerse en cada instancia de la misma manera en que se hace con los parámetros gráficos como la localización y el ángulo de los segmentos. Congruentemente con el paquete de graficación, las instancias contienen los valores de todos los parámetros que varían de una instancia a otra. Estos valores funcionan de una manera similar a los parámetros formales de las subrutinas de los lenguajes de programación. Y al igual que en ellos, se dificulta implantar un mecanismo que pase un número variable de parámetros [Kernighan 78]. En el contexto del paquete de conexiones, significa que una vez que se ha incluido una instancia de determinado módulo, ya no es posible agregar conexiones externas. Si el deseñador desea hacerlo, deberá generar un nuevo módulo (que contendrá al módulo original) y modificar la referencia de cada instancia.

Ya hemos delineado una forma de identificar las conexiones internas de diferentes instancias de módulos y establecido un mecanismo para establecer las conexiones externas semejante a los parámetros de las subrutinas. Lo más deseable sería que la técnica descrita permitiera al sistema hacer transparente el hecho de que las conexiones internas de los módulos, a pesar de ser únicas en el modelo del circuito, para fines del diagrama lógico se comportan como si fueran puntos eléctricos diferentes. Sin embargo, no es trivial lograr la transparencia total; sí es factible, pero ya que la simplicidad de implantación es una característica valiosa, cabe mencionar que no es necesario poder distinguir entre las conexiones internas de las instancias de los módulos en el diagrama lógico.

El sistema es igualmente útil si en lugar de que el diagrama del circuito muestre la estructura jerárquica a toda su resolución (o hasta el nivel que el usuario elija con el tipo de dato *bloque*), se dibuja el circuito solamente hasta el nivel del árbol en el que ya exista más de una instancia de alguna conexión interna. Como consecuencia, si el circuito contiene varias instancias de bloques con conexiones internas, éstas no serán visibles. Pero esto no es ninguna restricción relevante, ya que precisamente las conexiones internas, por definición, tienen la misma estructura en su bi-digráfica en todas las instancias, por lo que el usuario podrá examinarlas y modificarlas en el segmento gráfico asociado al diagrama del subcircuito. Es decir, que el usuario no interactúa con un solo diagrama *plano* del circuito, sino con uno por cada nodo que represente módulos eléctricos en la estructura arborescente.

La estructura se "aplana", entonces, hasta el momento de generar la lista de conexiones, que a su vez, puede consistir ya no de identificadores globales de chips, sino que ahora los chips se identificarán con la concatenación de los nombres de todos los ancestros del módulo en el que se localizan.

2.2.5 Limitaciones del paquete de conexiones

En resumen, lo que pretendemos que efectúe el programa de diseño de circuitos es que el usuario dibuje el diagrama lógico sin preocuparse por cómo es el modelo que contiene la representación intermedia del circuito. La labor principal del paquete de conexiones es garantizar una correspondencia entre lo que percibe el usuario y el modelo interno. Idealmente, todas las alteraciones que sufra el diagrama deberán actualizarse automáticamente en el modelo. Existen varios tipos de cambios que el usuario puede efectuar a través del paquete de graficación:

- generación de instancias de compuertas o módulos en el diagrama
- introducción de nuevas conexiones sobre el diagrama

Todas ellas, que son las que se presentan con más frecuencia, son relativamente fáciles de automatizar. Por ejemplo, si el usuario dibuja una línea a partir de una pata, el sistema concluirá que se trata de un alambre, y realizará las modificaciones necesarias en el modelo eléctrico (la bi-digráfica) para que corresponda con el diagrama.

Sin embargo, existen algunas modificaciones con las que no es fácil actualizar el modelo a partir de comandos gráficos únicamente. Por ejemplo:

- desplazamiento (sobre el diagrama) de un segmento
- cambio en la posición de las conexiones externas
- eliminación de conexiones

Cuando el usuario desplaza un segmento en el diagrama, digamos una compuerta, sus patas aparecerán desligadas de los alambres a los que en realidad indica la bi-digráfica que están conectadas. Lo más adecuado sería que el sistema generara las rectas necesarias para que en el diagrama correspondiera con el modelo eléctrico. Esto implica un algoritmo de ruteo, y aunque el sistema posee uno, debido a la rapidez que se necesita en la realimentación con el usuario, es muy sencillo y no respeta a los elementos gráficos en encuentra en el camino. Para sobreponerse a esta limitación, al igual que la reconstrucción del diagrama cuando las patas de un módulo cambian de posición, parece ser más conveniente emplear lenguajes orientados a restricciones, como ThingLab [Borning 81].

De igual manera, si el usuario ordena al paquete de graficación borrar una línea, a pesar de que el sistema tiene la información para saber si se trata de un alambre, hay datos de los que carece. La bi-digráfica asocia *todo* el punto eléctrico a un solo nodo. No obstante, en el diagrama del circuito la gráfica representa en muchos casos un punto con *varios* nodos.

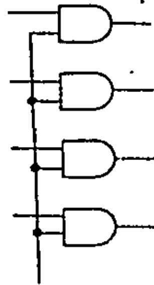


Figura 2. 6 Un mismo punto se asocia a varios nodos en el diagrama lógico

Un ejemplo se muestra en la fig 2.6. Esta dificultad se superaría con un modelo eléctrico más cercano al diagrama.

también la capacidad de crear nuevas subrutinas *en vivo* haciendo que el usuario personalice el sistema y lo adapte a su problema con facilidad.

No todas estas características son originadas en el código hilvanado per se; algunas son resultado de la combinación de Forth como lenguaje y el código hilvanado como su implantación. Es importante establecer la frontera entre un lenguaje y su implantación. De hecho, el código hilvanado empezó a usarse en el código generado por un compilador de Fortran. A continuación daremos una explicación de lo que es el código hilvanado en abstracto y posteriormente una descripción de Forth. Aunque tradicionalmente cuando se especifica un sistema primero se exponen sus características lógicas y luego su implantación, en este caso se justifica hacerlo en el orden inverso pues Forth está directamente inspirado en el código hilvanado.

3.1.2 Código hilvanado directo

El código hilvanado es una técnica de implantación de lenguajes que pretende ser tan rápida como el código compilado y tan compacta como el código interpretado. Se sabe que en general, un programa escrito en lenguaje de máquina (sobre todo si fue generado por un compilador) es rápido en su ejecución pero ocupa más espacio de memoria que el mismo programa escrito en un lenguaje interpretativo. Esta característica resulta del hecho de que el lenguaje de máquina está formado por instrucciones de más bajo nivel (casi siempre), por lo que se requieren más instrucciones para representar el mismo algoritmo. Sin embargo, debido a que las instrucciones son procesadas directamente por la circuitería, su ejecución es más rápida.

En el otro extremo, un programa interpretado contiene instrucciones de más alto nivel. Estrictamente hablando, cada instrucción representa una llamada a una subrutina que implica una trasferencia de control del procesador al código del intérprete que traduce la

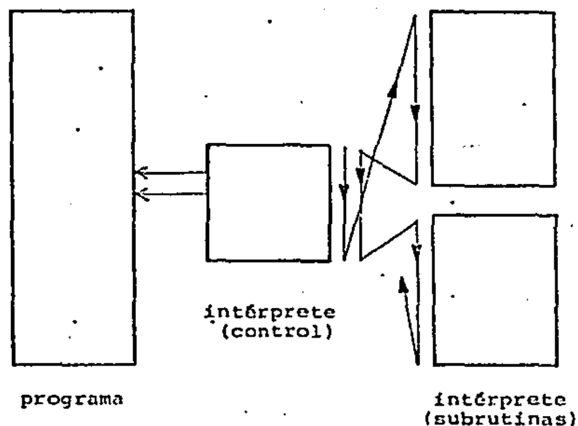


Figura 3.1 Flujo de control en el código interpretado (tomado de [Bell 73])

instrucción a la subrutina. Ver fig 3.1. Como resultado, se obtiene un programa más lento pero más compacto. Lo más sorprendente es que la afirmación anterior sigue siendo válida aun si sumamos la cantidad de memoria requerida por el intérprete a la del programa, para programas de regular tamaño, digamos del orden del tamaño del intérprete.

La comparación excluye la posibilidad de un lenguaje de máquina cuyas instrucciones sean las de un intérprete de alto nivel, así como un programa escrito de manera que todas sus instrucciones consisten en llamadas a subrutinas. Como se ve, la línea que separa a un intérprete de un compilador no está muy definida. Sin embargo, para propósitos prácticos podemos imaginar al código interpretado como uno formado exclusivamente por llamadas a subrutinas y el compilador como uno constituido únicamente por macros (expandidos).

¿Qué sucedería si a un programa escrito en código interpretativo en lugar de llamar al intérprete en cada instrucción *expandiéramos* el código pertinente del intérprete? Aparentemente obtendríamos algo que parece código compilado; pero ¿si el intérprete fuera muy compacto, digamos de una o dos instrucciones de máquina? ¿cómo es posible tener un intérprete tan sencillo? En realidad, un intérprete contiene código de dos tipos: el código de control, que ejecuta el ciclo principal de la máquina virtual (traer la instrucción, decodificarla y brincar a la subrutina asociada a ella) y el código de las instrucciones. La alternativa interesante es expandir el código de control, pues de otra manera caeríamos en un compilador común y corriente. Para que tenga alguna ventaja esta transformación, el intérprete (es decir el código de control del intérprete) debe ser realmente simple. Ello se logra eliminando el ciclo de decodificación de las instrucciones al reemplazar el código de operación por un apuntador a la subrutina asociada a la instrucción. Ahora sí, el control del intérprete consiste en

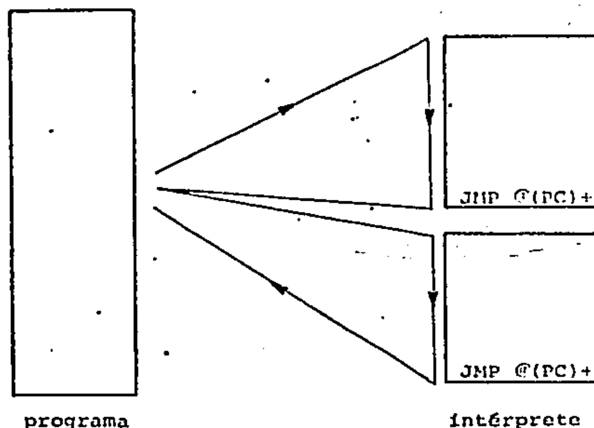


Figura 3.2 Estructura del código hilvanado (tomado de [Bell 73])

1. hacer un brinco indirecto a la dirección apuntada por el contador de programa de la máquina virtual
2. incrementar el contador de programa

Este código debe añadirse al final de cada subrutina del intérprete. En el lenguaje de máquina de la PDP-11 es una sola instrucción: JMP @(PC)+, en donde PC es un registro que contiene el contador de programa de la máquina virtual, los paréntesis significan el contenido del registro, la @ implica una indirección y el + un incremento a PC posterior al brinco. El código posee la estructura ilustrada en la fig 3.2. Los programas ejecutados de esta forma son más rápidos que los programas interpretados convencionalmente porque:

1. se ha eliminado la llamada al código de control del intérprete
2. ha desaparecido el ciclo de decodificación

Es de esperarse, sin embargo, que ocupe un poco más memoria que el código interpretado, pues ahora el equivalente al campo del código de operación de las instrucciones es una dirección de memoria; es decir, el espacio de códigos de operación es mayor. Casi sin querer hemos obtenido una ventaja más, que no salta a primera vista. El código de control del intérprete ya no necesita conocer la dirección de las subrutinas que forman el resto del intérprete. Podemos prescindir de la tabla que relaciona códigos de operación con direcciones de subrutinas. Se facilita entonces, la añadidura de nuevas subrutinas al intérprete. Esta representación de programas recibe el nombre de it código hilvanado directo y fue sugerida por Bell [Bell 73]. Se empleó como código objeto de un compilador de Fortran para la PDP-11, con resultados muy satisfactorios.

3.1.3 Código hilvanado indirecto

Si se está dispuesto a sacrificar cierta rapidez de ejecución a cambio de lograr una representación homogénea entre las subrutinas del intérprete, a las que llamaremos *funciones primitivas* y las subrutinas del programa, a las que denominaremos *funciones secundarias* es conveniente introducir una variación al código hilvanado directo. En lugar de ejectuar una indirección antes de saltar a la subrutina del intérprete, realicemos dos. Las funciones primitivas permanécen prácticamente iguales, a excepción de que el programa que las utilice deberá guardar ahora un apuntador a un apuntador a la subrutina. Las funciones secundarias, en cambio, ya no se llaman con un apuntador a la subrutina CALL y un parámetro como se hacía en el código hilvanado directo; ahora solamente se coloca el apuntador a la subrutina. Este apuntador debe a su vez apuntar al código que almacena el contador de programa en una pila y brinca a la primera instrucción de la función secundaria (que es el código normalmente realizado por la función CALL. El código de control del intérprete ahora es:

```
IR  := @(PC)+
JMP @(IR)+
```

en donde IR es el registro de instrucciones de la máquina virtual (supuestamente un registro del CPU) y el código que realiza la llamada a la subrutina:

```
PUSH (PC)
PC := IR
JMP (PC)
```

La fig 3.3 esquematiza la representación de funciones con esta modificación.

Lo que ha sucedido es que el proceso de llamado a subrutinas ya no es ejecutado por la función *que llama* sino por la que *es llamada*. En cierto sentido hemos *invertido* la manera tradicional de llamar subrutinas y ahora primero se cede el control a la subrutina y esta es la que "decide" guardar el contador de programa en la pila y ejecutar su código; pero podría hacer alguna otra cosa. Podría por ejemplo no ejecutar su código sino regresar determinado valor a la subrutina que la llama. Esto último equivale a tener una variable global en lugar de una subrutina. Es decir, que escogiendo de manera adecuada el código que se ejecuta al llamar una función, ésta puede comportarse como una subrutina o como una variable. Incluso es factible implantar corrutinas escribiendo prólogos que turnarán el control de flujo entre ellas. En conclusión, estamos usando el mismo código de control del intérprete para tratar tanto funciones primitivas como secundarias como variables. Recordemos que el intérprete es de dos instrucciones. Y podríamos continuar en forma similar. Diseñando apropiadamente nuevos códigos iniciales a funciones procesaremos en forma homogénea otros tipos de datos.

Figura 3. 3 Código hilvanado indirecto

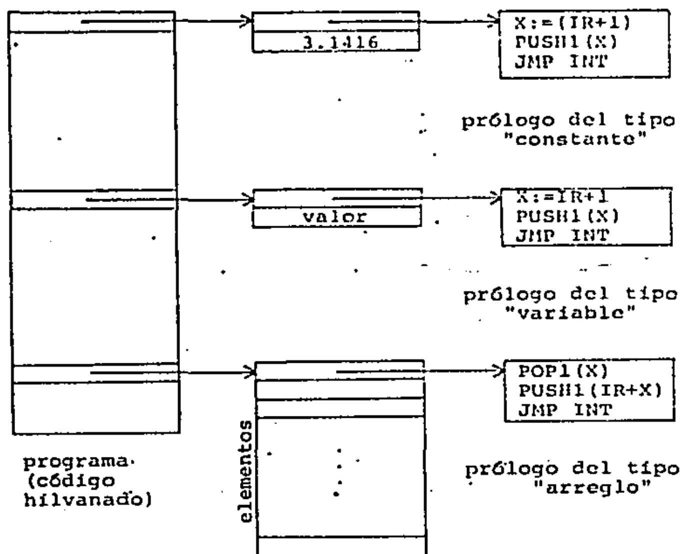


Figura 3. 4 Estructura de algunos tipos de datos

Al código inicial de cada función le llamaremos *prólogo* [Kogge 82]. La fig 3.4 muestra la estructura del tipo de dato *constante* que regresa determinado valor en la pila, así como el tipo de dato *variable* que regresa la dirección de la variable global en la pila (regresa la dirección en lugar del valor con objeto de poder modificarla). También ilustra el tipo de dato *arreglo* que recibe un parámetro: el índice. Su prólogo calcula la dirección del elemento accesado. De manera similar se establecen arreglos de más dimensiones o arreglos de registros (records). Esta máquina virtual fue propuesta por Dewar [Dewar 75] bajo el nombre de *código hilvanado indirecto* para implantar una versión rápida de Snobol4 llamada Spitbol.

Al implantar el código hilvanado indirecto en una microcomputadora de 8 bits, como el Z80 tal como ha sido descrito, aparece una dificultad inesperada. Debido por un lado a que no existen instrucciones de máquina con "post-incremento", es decir que tenga como efecto secundario un incremento al registro usado, y por otro a las limitaciones del lenguaje de máquina para direccionar palabras de 16 bits, el intérprete crece considerablemente. El intérprete presentado a continuación está en lenguaje de máquina del Z80 y fue tomado de [Loelinger 81]. Loelinger sugiere usar el registro BC como contador de programa y el registro DE como apuntador al prólogo.

```
        LD      A,(BC)
        LD      L,A
        INC     BC
        LD      A,(BC)
        LD      H,A
        INC     BC              ;IR := @(PC)+
        LD      E,(HL)
        INC     HL
        LD      D,(HL)
        INC     HL
        EX      DE,HL
        JP      (HL)            ;JMP @(IR)+
```

Un intérprete de esta magnitud ya es impráctico duplicarlo al final de cada función primitiva, por lo que parece más adecuado regresar a la estructura tradicional de guardar una sola copia del código de control del intérprete y hacer que al término de cada función primitiva el procesador brinque al intérprete. No obstante seguirán siendo válidas algunas ventajas logradas anteriormente como la facilidad para agregar funciones primitivas, ausencia de una tabla que traduzca códigos de operación a direcciones, homogeneidad de procesamiento para funciones primitivas, secundarias y tipos de datos, y un intérprete sencillo. Además, en el caso particular del Z80, si almacenamos la dirección del intérprete en el registro IY, el brinco es bastante rápido.

Si escogemos al registro IX como el apuntador a la pila, el prólogo de las funciones secundarias queda:

```
CALL:   DEC     IX
        LD      (IX),B
        DEC     IX
        LD      (IX),C          ;PUSH (PC)
        LD      C,E
        LD      B,D             ;PC := IR
        JP      (IY)            ;JMP (PC)
```

y el código de regreso de las funciones secundarias resulta:

```
RETURN: DEFW    $+2
        LD      C,(IX)
        INC     IX
        LD      B,(IX)
        INC     IX              ;POP PC
        JP      (IY)
```

Por supuesto que si colocamos en la memoria el código de retorno o el código de llamada inmediatamente antes del del intérprete desaparece el último brinco.

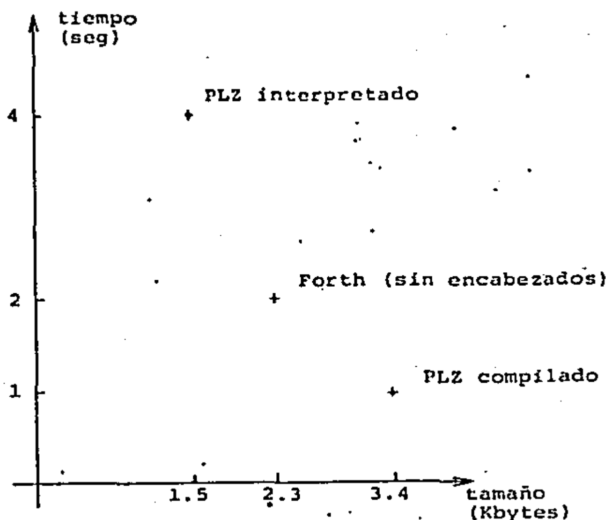


Figura 3.5 Comparación de la velocidad de ejecución y el tamaño del programa de tres lenguajes

Hasta antes de la última transformación que sufrió la máquina virtual y se centralizó de nuevo al intérprete, intuitivamente parecía ejecutar el código con más velocidad que un intérprete convencional sin haber sacrificado mucho espacio de memoria. Sin embargo, ahora ya no es claro si en realidad estamos en ventaja con respecto a un compilador o un intérprete tradicional. Con objeto de dilucidar el lugar que ocupa este intérprete hilvanado con respecto a ambos sistemas se elaboró un programa de prueba. La comparación fue hecha con el lenguaje PLZ [PLZ/SYS 78] que es similar a Pascal, aunque tiene influencia de Mesa y C entre otros lenguajes. La versión ofrecida para la microcomputadora MCZ 1/05 da la opción al usuario de traducir su código a lenguaje de máquina o al lenguaje de una máquina virtual que es interpretado. La fig 3.5 muestra los resultados. No se tiene mucha información de la estructura del código interpretado, pero es factible suponer que las instrucciones poseen un campo para el código de operación y otros para los argumentos. Incidentalmente, el programa de prueba es una parte del paquete de graficación del sistema para diseño de circuitos que nos concierne, por lo que sí es representativo para nuestra aplicación.

Vale la pena recordar otras comparaciones hechas con intérpretes de Basic en las que la relación de velocidades no es de dos a uno como en este caso, sino 10 a uno. Pero Basic está en desventaja con respecto a Forth pues la mayor parte de sus intérpretes hacen análisis sintáctico durante la ejecución, mientras que Forth, al igual que PLZ efectúa una traducción

previa. Es notable además, que la traducción aun para programas interpretados es mucho más lenta que la realizada por Forth: para el programa de prueba tomó al intérprete de PLZ 4.5 veces más tiempo (claro está que Forth es muy poco legible y su sintaxis es más sencilla). La cantidad de memoria ocupada por el programa en Forth mostrada en la fig anterior no incluye encabezados (que se explicarán en la sección siguiente) pues no son indispensables para ejecutar un programa; solamente se emplean durante la compilación y la depuración del programa. Aunque el inérprete que se usó para implantar el sistema para diseño de circuitos no genera código sin encabezados, existen versiones comerciales de Forth que sí son capaces de hacerlo. Los encabezados incrementan en un 25% el tamaño del programa de prueba. Además, los resultados de la figura excluyen la memoria ocupada por las variables globales para solamente abarcar el espacio requerido por el código.

3.1.4 Comparación del código hilvanado con PLZ

Establezcamos una función de costo con objeto de averiguar si las características exhibidas por Forth durante el programa de prueba son competitivas con el compilador o el intérprete:

$$C_i = M_i\alpha + T_i\beta$$

en donde:

M_i es la cantidad de memoria empleada por el programa de prueba traducido por el traductor i.

T_i es el tiempo requerido por el programa de prueba traducido por el traductor i

α es el costo en dinero de la memoria

β es el costo en dinero del tiempo de procesamiento

Un buen modelo haría a α y a β dependientes de la memoria y del tiempo respectivamente, pues es sabido que ambos costos se pueden aproximar con bastante precisión considerando funciones que aumentan exponencialmente con respecto tanto a la cantidad de memoria como al tiempo de procesamiento. Asimismo, una buena función de costo de la memoria debe ser discontinua debido a que en general los fabricantes ofrecen ampliación de memoria en bloques. No obstante, por simplicidad, consideraremos a α y β constantes. También compararemos funciones de costo normalizadas con respecto a $\alpha+\beta$ a fin de eliminar costos absolutos y tratar con el costo de la memoria con respecto al del tiempo. Nos queda entonces:

$$C'_i = M_i\alpha' + T_i(1 - \alpha')$$

en donde:

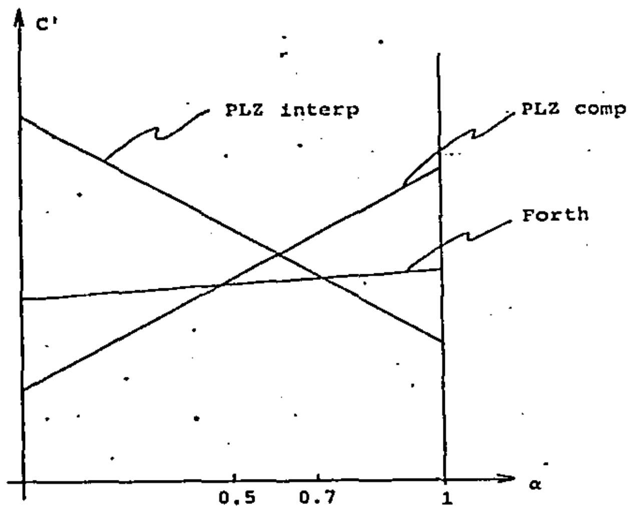


Figura 3.6. Costo relativo de los tres lenguajes para el programa de prueba

$$\alpha' = \frac{\alpha}{\alpha + \beta}$$

α' varía entre 0 y 1 y representa el costo relativo de memoria.

Tomando los datos obtenidos con el programa de prueba obtenemos:

$$C'_{interp} = -2.5\alpha' + 4;$$
$$C'_{forth} = 0.3\alpha' + 2;$$
$$C'_{comp} = 2.4\alpha' + 1.$$

que son las ecuaciones mostradas en la fig 3.6.

A pesar de ser este análisis muy simple, sí arroja algunos resultados interesantes. La cantidad de memoria está medida en Kbytes y el tiempo en segundos. Esto significa que si el costo de un Kbyte es cercano al de un segundo de procesamiento, Forth representará la alter-

nariva con costo mínimo. También se puede concluir de este análisis que Forth efectivamente es competitivo con respecto a los otros dos sistemas y aunque el costo relativo de un Kbyte de memoria no sea 0.6 del costo relativo de un segundo de procesamiento, la pérdida en costo por haber usado Forth no es muy grande.

3.1.5 Mezcla de lenguajes en un programa

Al tratar de hacer un estudio más cuidadoso, nos percatamos de que se están pasando por alto determinados parámetros importantes. Debe tomarse en cuenta, por ejemplo, que el programador que use el lenguaje PLZ, ya que tiene la opción de interpretar o compilar cada subrutina a su conveniencia, estará generando un conjunto grande de curvas, como consecuencia de las combinaciones entre subrutinas compiladas e interpretadas: 2^n, en donde n es el número de subrutinas del programa. El problema de cuál combinación escoger puede plantearse como un problema de programación entera cero-uno [Saaty 70] o resolverse por métodos heurísticos.

Al decidir la mejor mezcla de lenguajes, no se elaboró ningún modelo matemático, pero sabemos que se hizo una buena elección en base a las características tan constrastantes de los lenguajes disponibles. PLZ se eliminó en su totalidad debido a:

1. Es extremadamente lento para compilar en comparación con Forth. A pesar de haber construido el programa en un 80%, se consideró más práctico reprogramarlo en Forth, pues el ciclo de modificaciones (editar el programa, compilarlo, ligarlo y probarlo) era de unos 20 minutos. Psicológicamente es una desventaja debido a que el costo que asignamos al tiempo de respuesta de una computadora crece muy rápidamente, por lo menos a corto plazo. El lapso de tiempo empleado por PLZ para compilar un programa de esa magnitud es suficiente para distraerse e incluso olvidar cuáles fueron los cambios más recientemente incorporados al programa. Forth, en cambio, requere en el peor de los casos de 4 a 5 minutos y en la mayoría de ellos le bastan 30 segundos.

2. Aunque indiscutiblemente PLZ es un lenguaje de más alto nivel que Forth (Forth no permite nombrar los parámetros de las subrutinas), el emplear una mezcla de Forth y PLZ, le resta transportabilidad al sistema. Con excepción de las subrutinas que definitivamente resultan muy lentas en Forth y se escribieron en ensamblador, todo se hizo en código hilvanado.

3. El depurador de PLZ es enorme: ocupa 20 Kbytes. Una vez que el paquete de graficación elaborado en PLZ alcanzó determinado tamaño, fue imposible seguir auxiliándose del depurador y para fines prácticos era como si no existiera, a pesar de ser un programa excelente, con ocho puntos de ruptura (break points), asociados a ocho contadores y con información de tipos de datos. En comparación con el depurador de PLZ, el de Forth es muy primitivo pues solamente consiste en poder ejecutar cualquier subrutina con parámetros de valor arbitrario y hacer referencia a las variables por su nombre. Esta característica es algo "natural" en los programas escritos en Forth, ya que almacena los nombres de todas las subrutinas en un diccionario, que representa el 25% del tamaño total. Es decir, que hubo que desechar el depurador de PLZ por ser "demasiado bueno".

A estas ventajas de Forth sobre PLZ, que hicieron imposible seguir trabajando con PLZ, tanto por la cantidad de memoria como por el tiempo de compilación, hay que agregar

que como producto secundario, Forth ofrece extensibilidad en el repertorio de comandos. Todo sistema de programación "bien elaborado" ofrece al usuario la capacidad de construir nuevos comandos basados en las instrucciones originales y almacenarlos bajo algún nombre. Entre ellos contamos con los macros de editores de texto como Teco y Emacs y algunos paquetes de graficación como el de SUDS [SUDS 80] que es un sistema auxiliar en el diseño de circuitos. Al emplear Forth, inmediatamente proveemos al usuario con macros.

3.2 Forth

3.2.1 Niveles de extensibilidad

Forth ha sido comparado con un Meccano transparente. La analogía con un juguete desarmable efectivamente refleja la simplicidad y la flexibilidad del lenguaje. El usuario tiene acceso a todas las variables del intérprete de manera que es fácil extenderlo, definir nuevas estructuras de control y crear tipos de datos ausentes en el sistema original. Forth está basado en una máquina de código hilvanado. En la sección anterior se describió una característica del código hilvanado: la representación homogénea entre subrutinas (funciones) y tipos de datos. La diferencia entre los tipos de entidades en un ambiente hilvanado radica en sus prólogos, que contienen las instrucciones de cómo procesar cada entidad. Los prólogos indican si las entidades deben interpretarse como código o como variables o como arreglos o cualquier tipo de dato imaginable.

La máquina virtual que ejecuta código hilvanado, que llamaremos *intérprete interno* es en extremo sencilla y en algunos lenguajes de máquina ocupa dos instrucciones. No obstante, el intérprete interno presupone que el prólogo no está escrito en código hilvanado sino en lenguaje de máquina. A pesar de ello, Forth atinadamente provee un prólogo especial que hace que se ejecute en código hilvanado un segundo prólogo. Esto permite crear nuevas entidades cuyo prólogo está escrito en Forth. Como consecuencia, Forth habilita al usuario para que defina el código que se ejecuta al hacer referencia a algún tipo de dato.

Además, debido a que el usuario tiene acceso al diccionario de entidades (algunos textos llaman *palabras* a las entidades), Forth es extensible a dos niveles en los que la mayoría de los lenguajes se comportan como cajas negras. Se dice que en total, Forth es extensible a tres niveles [Harris 80].

- definición del código de nuevas funciones. Este nivel de extensibilidad es común a casi todos los lenguajes y no representa ninguna innovación. Probablemente el único aspecto interesante en este sentido es que Forth permite que una función regrese varios valores, es decir tenga diversos parámetros de salida.

- definición del código de creación de los tipos de datos. Al tener acceso al diccionario, el usuario está facultado para asignar un valor inicial a los tipos de datos. Ciertos lenguajes de programación contienen comandos para iniciar el valor de las variables, pero debido a que en la mayoría de ellos existe un conjunto predefinido de tipos de datos (digamos arreglos y estructuras), no hay manera de establecer el código de creación de los datos en el lenguaje

mismo. El programador de Forth puede no solo crear funciones, sino crear funciones que crean funciones.

- definición del código de ejecución de los tipos de datos. Equivale a que es factible programar los prólogos de las entidades. Este nivel de extensibilidad está muy relacionado con el anterior; de hecho, al definir una función que crea funciones, es necesario programar tanto el código de creación como el prólogo del tipo de dato, dentro de la misma entidad.

3.2.2 Lenguajes orientados a objetos

Vale la pena mencionar que existe un tipo de lenguajes que es más general que Forth en lo que respecta a los prólogos: los llamados lenguajes orientados a objetos [Rentsch 82]. Las entidades en los sistemas con objetos tienen asociada una respuesta distinta a cada *mensaje* que recibe el objeto. Desde el punto de vista de Forth, equivale a que las entidades posean varios prólogos.

El más conocido de ellos es Smalltalk [BYTE 81]. En este sistema se denomina *método* a la respuesta de los objetos a cada mensaje. En él, también es posible identificar un código asociado a la creación de objetos, al que llaman *métodos de clase* y un código relacionado con los objetos creados por los primeros que recibe el nombre de *métodos de instancia*. Al igual que en Forth, el sistema está compuesto por un conjunto homogéneo de entidades, en las que no hay una diferencia fundamental entre código y tipos de datos. Sin embargo, Smalltalk contiene dos tipos de objetos, las clases y las instancias. Es frecuente que los sistemas orientados a objetos organicen a las entidades en una estructura jerárquica de manera que si un objeto recibe un mensaje para el cual carece de método, el sistema busca el método pertinente en el ancestro inmediatamente superior al objeto y así sucesivamente hasta encontrarlo.

Otros sistemas orientados a objetos son CLU [Liskov 77], ALPHARD [Wulf 76] y en menor grado pero sí con los conceptos principales Simula [Dahl 66], Sketchpad [Sutherland 63] y los marcos (frames) en Lisp [Winston 81].

3.2.3 Parámetros

Forth emplea dos pilas (stacks): una para los parámetros de las subrutinas y otra para las direcciones a las que debe regresar al terminar de procesar las funciones que han sido llamadas. La mayoría de las implantaciones de los lenguajes derivados de Algol emplean un mecanismo para llamadas a subrutinas que utiliza una sola pila, en la que almacenan tanto los parámetros como las direcciones de retorno. Como concecuencia al llamar una subrutina cuyos parámetros han sido previamente colocados en la pila, se almacena todavía arriba de los parámetros el valor del contador de programa. La subrutina llamada, a su vez, tiene que "destapar" la pila extrayendo el contador de programa antiguo para sacar los parámetros, y volver a tapar la pila con objeto de que al regresar se tome el valor adecuado del contador de programa.

La existencia de dos pilas separadas simplifica el paso de parámetros e incrementa la velocidad de llamado de subrutinas.

Los parámetros en Forth no se pasan ni por valor ni por referencia. Al llamar a una función, esta extrae sus parámetros de la pila tal y como fueron dejados por la función que la llamó. La función que llama simplemente coloca valores en la pila, pero depende del prólogo del tipo de dato al que haga referencia para poder decir si en la pila queda la dirección o el valor de una variable.

En caso de que se trate del tipo de dato **variable**, que es una variable escalar global y tiene como prólogo regresar en la pila la dirección de la variable, parecería que se está haciendo una llamada por referencia. No obstante, existe la función "@" que permite calcular el valor de la variable antes de llamar a la subrutina, lo que equivale a una llamada por valor.

Además el tipo de dato **constant** automáticamente regresa el valor del dato en la pila sin necesidad de hacer una indirección. Este tipo de dato difiere de lo que la mayoría de los lenguajes entiende por "constante", es decir, un valor conocido durante compilación e imposible de modificar en el programa objeto sin compilarlo de nuevo. El tipo de dato **constant** se comporta más bien como una variable que regresa su valor en la pila y aunque no existe una forma automática de hacerlo, sí es factible modificar su valor durante la ejecución del programa, si se averigua su dirección en la memoria.

En resumen, el hecho de llamar a una subrutina con parámetros globales no implica que se se llamen por valor o por referencia, más bien depende del prólogo de los parámetros; para el tipo de dato **variable** se sobreentiende una llamada por referencia a menos que el usuario indique lo contrario, y para el tipo de dato **constant** es al revés. De manera similar, el usuario puede definir tipos de datos que regresen varios valores en la pila, simulando una llamada por valor a un arreglo, por ejemplo.

Las variables locales y los parámetros de una función no tienen nombre. El usuario debe hacer referencia a ellos por el lugar que ocupan en la pila con respecto al tope. Consecuentemente, dentro de una función, es posible que a la misma variable se haga referencia de más de una forma distinta, dependiendo del número de datos que exista sobre la variable en la pila. Este es probablemente el defecto más notorio en Forth y una de las características que lo hacen menos legible. Aunque Forth es suficientemente flexible como para construir un compilador con Forth mismo, que sí reconozca nombres de parámetros y variables locales no se sabe de ningún intento en este sentido.

Si vemos a la asignación como la llamada a una subrutina, podemos afirmar que la mayor parte de los lenguajes de programación que preveen la asignación de variables, presuponen que la variable que va a ser modificada es llamada por referencia y que todas las variables presentes en la expresión cuyo valor debe recibir la variable son llamadas por nombre. En Forth no existe esta presuposición; la asignación "!", al igual que todas las funciones, toma sus parámetros tal y como fueron colocados por la función que la llamó, de manera que es labor del usuario el cuidar de colocar el valor como primer parámetro de la asignación y la dirección de la variable como segundo.

3.2.4 Intérprete externo

El intérprete de comandos recibe el nombre de *intérprete externo*. Emplea notación polaca postfija y evalúa expresiones con una máquina virtual (de más alto nivel que el intérprete interno) que

- si encuentra un número lo convierte a binario y lo introduce a la pila
- si no es número busca en el diccionario suponiendo que se trata de una función; si la encuentra la ejecuta sin verificar que se haya pasado el número adecuado de parámetros, y si no, marca error.

Los únicos delimitadores son el espacio en blanco y el retorno de carro. Esto hace que los nombres de las funciones puedan contener cualquier carácter (diferente de los anteriores) como parte de su nombre.

Cada vez que termina de ejecutar las funciones que escribió el usuario en un renglón, el intérprete externo examina que no se hayan extraido más parámetros de los necesarios, en cuyo caso marca error.

El intérprete externo solamente introduce en la pila los números pero nunca las cadenas de caracteres. ¿Es posible entonces pasar cadenas de caracteres como parámetros? No en forma tan directa como los números. Forth emplea notación polaca prefija para parámetros alfanuméricos. De manera que las funciones deben buscar sus parámetros alfanuméricos en el arreglo (buffer) asociado a la terminal, al cual tienen acceso. Sin embargo, existen variantes de Forth, como Stoic, a cuyos parámetros alfanuméricos se les antepone un apóstrofe y conservan la notación polaca postija.

3.2.5 Compilador

El llamado "compilador" de Forth es muy primitivo como para que amerite tal nombre; más bien se parece a la función READ de Lisp, pues básicamente traduce identificadores externos de funciones (sus nombres) a identificadores internos (sus direcciones) con objeto de no tener que hacerlo durante la ejecución. En realidad existen varios compiladores dependiendo del prólogo que asignan a la función y de cómo traducen su *cuerpo*. El que se usa con más frecuencia es ":" que sirve para crear funciones secundarias. Este compilador, una vez que ha incluido a la nueva entidad en el diccionario y establecido su prólogo como el de una función secundaria (que se vio en la sección anterior), efectúa un proceso similar al del intérprete externo:

- si encuentra un número, incluye en el cuerpo de la función un apuntador a la función "literal" y a continuación el número en binario. El código de la función literal es, como es de esperarse, transferir a la pila el número al que apunta el contador de programa del intérprete interno y avanzar el contador de programa a fin de que el número no se interprete como una función.
- si encuentra una función, se da uno de dos casos: si se trata de una función "mediata", como son la mayoría de las funciones de Forth, la busca en el diccionario e incluye su dirección en el cuerpo de la función que está siendo definida. Si resulta ser una función "inmediata", la ejecuta. Un primer ejemplo de funciones inmediatas es la que termina la definición de las funciones ":". Si esta función no se ejecutara durante la compilación, no habría manera de finalizar el cuerpo de las funciones secundarias. Entre otras labores, esta función debe incluir un apuntador al intérprete interno como última instrucción de la nueva entidad y regresar al intérprete externo.

Otras funciones que se ejecutan durante la compilación son DO, LOOP, IF, ELSE y THEN. DO es una función que debe ir siempre acompañada por LOOP, que una vez compilada,

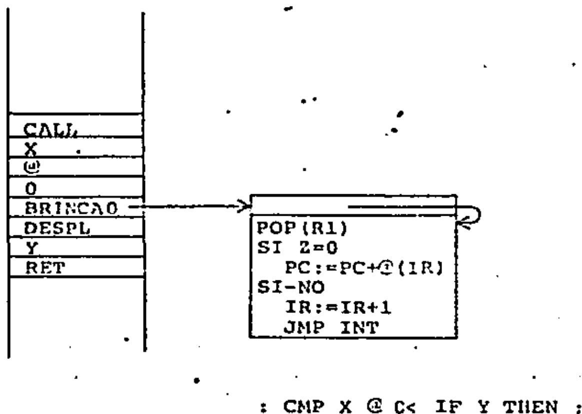


Figura 3.7 Compilación de la función IF ... THEN

toma de la pila dos valores entre los cuales hace variar de uno en uno un índice (equivale al FOR de Algol) y entre cada variación ejecuta el código localizado entre DO y LOOP. LOOP debe traducirse entonces en un apuntador a cierta función que decide si el código debe repetirse una vez más e incrementar el valor del índice o si el flujo de control debe seguir adelante. Sin embargo, la longitud de este brinco hacia atrás se calcula durante la compilación de la siguiente manera: DO mete a la pila de datos la dirección en la que apareció él mismo. LOOP, a su vez, toma de la pila la dirección dejada por DO y la resta de *su* dirección, traduciendo el LOOP por un brinco condicional relativo.

En forma similar se compilan IF y THEN. El IF, una vez compilado toma de la pila un valor. Si es cero, brinca a la instrucción inmediatamente después del THEN y si es distinto de cero, ejecuta el código entre el IF y el THEN. Al igual que en el caso anterior, la longitud del brinco se calcula durante la compilación. Cuando el compilador encuentra un IF, lo traduce a un apuntador a otra función, digamos 0-BRANCH, la cual tiene como código lo que se espera del IF una vez compilada la función, y mete a la pila la información adecuada para que el THEN calcule el tamaño del brinco y llene la palabra de memoria que está a continuación de 0-BRANCH. El THEN desaparece después de la compilación. La fig 3.7 ilustra la compilación del IF ... THEN. También existe la secuencia de instrucciones IF ... ELSE ... THEN.

Como puede apreciarse, Forth está diseñado para la programación estructurada, eliminando los GO TO's. Asimismo, es factible anidar estructuras de control, pues se comunican a través de una pila en la que se extraen primero los datos asociados a las estructuras más interiores.

Forth no verifica que todos los THEN casen con los IF, ni los DO con los LOOP, por lo

que es común que aparezcan errores de programación originados por falta de "terminadores" de estructuras de control. Aun así, si observamos que siempre entre la definición de una función y la siguiente la pila de datos debe estar vacía, es fácil hacer que la función ";" verifique esta condición y marque error en caso de que no se cumpla. Esta técnica no detectará exactamente en qué parte del programa está el error, pero sí en cuál función.

Existen otros compiladores como ;CODE que genera funciones primitivas, y VARIABLE y CONSTANT que crean variables y constantes. Para crear una variable se usa la forma <valor inicial> VARIABLE <nombre>. (Conserva la notación postfija para valores numéricos y la prefija para cadenas de caracteres.)

3.2.6 Diccionario

Las entidades de Forth se almacenan en un diccionario. Para propósitos de su uso, es irrelevante su estructura, pero vale la pena mencionar algunas características que tienen consecuencias importantes. La mayor parte de las implantaciones de Forth emplean una lista ligada de funciones por lo que tanto el compilador como el intérprete externo deben realizar una búsqueda secuencial. Aunque algunas variantes de Forth, como IPS [Meinzer 79], utilizan una función de dispersión (hashing), la lista ligada permite implantar con facilidad el concepto de "vocabularios".

Los vocabularios son conjuntos de funciones asociados a un nombre. Con ellos, es posible ocultar al compilador y al intérprete externo grupos de funciones permitiendo al usuario emplear el mismo nombre de funciones bajo contextos distintos. Los conjuntos no necesitan forzosamente ser disjuntos. En Forth es factible incluir digamos las funciones básicas en todos los conjuntos. Por ejemplo, en un sistema con un editor de texto y un ensamblador, el usuario puede utilizar en su programa de aplicación nombres de funciones que yazcan en los vocabularios del editor o del ensamblador. Los vocabularios fueron de especial utilidad en este trabajo debido a que se almacenaron las entidades gráficas en el diccionario de Forth y emplearon vocabularios distintos para el catálogo de circuitos integrados, el conjunto de símbolos eléctricos y el circuito del usuario.

Los nombres de las entidades no se guardan en su totalidad, pues se supone que el sistema reside en un sistema de computación chico. Generalmente Forth almacena los tres primeros caracteres y el número de caracteres del nombre. A estos cuatro bytes, junto con el apuntador a la función anterior y con el prólogo se le denomina *encabezado* de la función.

3.2.7 Conclusiones

Forth ha recibido y con razón el calificativo de un lenguaje de solo escritura (write-only). El lenguaje es tan críptico como APL, en parte debido a la notación polaca pero sobre todo porque los parámetros y las variables locales carecen de nombre.

La notación postfija no es una desventaja importante pues es bien sabido que mucho depende de la costumbre del programador. El usuario de Lisp (que emplea notación polaca prefija) rápidamente adquiere habilidad para leer sus programas. La falta de nombres en las

variables locales y en los parámetros sí resulta en un inconveniente grave y es de extrañar que no se haya corregido. Se mencionó que es relativamente fácil elaborar un compilador análogo a ":" en Forth mismo que sea capaz de reconocer nombres de parámetros, pero el programa para diseño de circuitos no contiene esta extensión. Una vez estimada la complejidad del nuevo compilador y dada la magnitud del programa producido, se ha llegado a la conclusión de que sí habría valido la pena haber hecho el nuevo compilador. Se elaborará, entonces, para otra aplicación, tomando en cuenta las experiencias obtenidas en este programa.

A pesar de que el compilador de Forth realiza una traducción muy sencilla, es costumbre que se almacene una copia fuente del programa además del código objeto. Existen "descompiladores" de Forth [Duncan 81], pero no son populares debido a que normalmente Forth guarda las funciones como arreglos. Para insertar instrucciones se necesitaría recorrer las entidades en la memoria o emplear otra zona de memoria generando basura.

La alternativa de insertar instrucciones no es conveniente porque los identificadores internos de las entidades de Forth son sus *direcciones*. Al mover de lugar una función, a pesar de que no afectaría los brincos ya que todos son brincos relativos, sí significa modificar todas las referencias a la función desplazada. Para que este mecanismo que traslada entidades en la memoria fuera práctico, habría que emplear otro tipo de identificadores internos que no dependiera de la dirección en la que se localiza la entidad. Un ejemplo de identificadores internos que cumple con la característica menciondada son los "surrogates" [Copeland 82], empleados en algunos manejadores de bases de datos (DBMS). Una vez asignados a una entidad, nunca cambian y la dirección física de las entidades se obtiene a través de una tabla que recibe como entrada el identificador interno y regresa la dirección de la entidad. ADABAS los llama números internos de secuencia [Kroenke 77].

Hay implantaciones de Forth que emplean un concepto similar y logran código relocalizable (place-independent-code) agregando una indirección más en el intérprete interno.

Al listar las desventajas de Forth, es inevitable recordar a Lisp. Lisp también es conversacional y contiene un diccionario en el que almacena entidades de varios tipos que se presta para guardar los objetos gráficos. Lisp carece de los inconvenientes de Forth: habilita al usuario en la alteración de las subrutinas *sobre* su representación objeto, almacena todos los caracteres de los nombres de las entidades y los argumentos de las funciones poseen nombre. No obstante Lisp estructura la información con listas ligadas y las celdas son extremadamente chicas: el apuntador a la siguiente celda ocupa la mitad de los nodos. A menos que se utilice algún mecanismo que compacte los apuntadores como "codificación del CDR" [Baker 77], cabe esperar que los programas ocupen más o menos el doble de memoria que los escritos en Forth.

En principio, una implantación de Lisp que codifique los apuntadores sería claramente deseable sobre Forth. Solamente prevalece la ventaja de Forth relativa a su facilidad de implantación como lenguaje. Lisp, aunque sencillo, es francamente más complejo de implantar que Forth.

El libro de Loelinger [Loelinger 81] es bueno para entender la implantación de Forth, pero para una introducción menos profunda, consúltense [BYTE 80] y [Kogge 82].

Capítulo 4

Implantación

4.1 Paquete de graficación

4.1.1 Segmentos

Los segmentos son conjuntos de entidades y se representan con una lista ligada de registros. La liga es en un solo sentido. Algunos paquetes de graficación emplean listas doblemente ligadas [Sutherland 63] que justifican con la operación que elimina una entidad del conjunto. Efectivamente, si el único parámetro que recibe la subrutina que extrae objetos de la lista ligada es un apuntador al registro, resulta ineficiente tener que recorrer el segmento desde el primer nodo hasta encontrar el nodo inmediatamente anterior al que se desea suprimir. La lista ligada en ambos sentidos permite localizar rápidamente los apuntadores que se alteran.

En nuestro paquete conviene preveer que para que el usuario comunique al sistema qué objeto pretende descartar debe previamente señalarlo sobre la pantalla. Al colocar el cursor sobre un objeto, el usuario llama a una subrutina que recorre la estructura (a las subrutinas que visitan secuencialmente los nodos de la jerarquía los hemos denominado máquinas virtuales ya que pueden verse como procesadores que ejecutan el archivo gráfico; por supuesto que la semántica de las instrucciones es diferente para cada procesador). Si se programa adecuadamente la máquina que encuentra el nodo sobre cuyo dibujo está el cursor, es factible que el sistema recuerde al campo del nodo anterior sin que sea necesario recorrer nuevamente la lista al momento en que el usuario ordena eliminar el registro.

Todos los nodos contienen un apuntador al siguiente nodo en la lista y el valor del último de ellos es nulo. Dentro de un segmento puede haber objetos primitivos, como rectas y

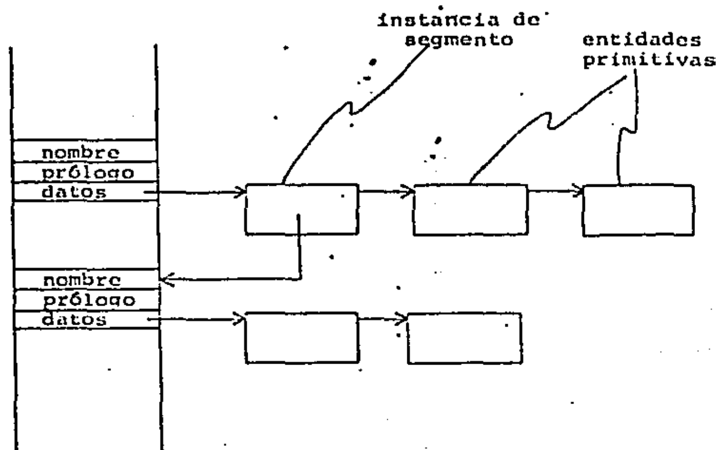


Figura 4.1 Estructura de los segmentos

cadenas de texto, o instancias de segmentos. Este tipo de registros posee un segundo apuntador al segmento del cual es instancia. La referencia al segmento de más bajo nivel es una dirección del diccionario de Forth, en lugar de apuntar al primer nodo del segmento con objeto de poder saber su nombre como indica la fig 4.1.

En otras palabras, se está modelando un árbol enario por medio de un árbol binario en el que el apuntador izquierdo de los registros se refiere al primer hijo y el derecho al siguiente hermano.

Además del apuntador al hermano, existe otro campo que es común a todos los registros, el tipo de dato. Este campo ocupa un byte.

Forth es un lenguaje extensible en el que el usuario puede crear subrutinas que se comportan como si fueran funciones primitivas definiendo los prólogos de las funciones. Los segmentos no solo están incluidos en el diccionario de Forth, sino que son funciones del lenguaje. Ahora bien, ya que las entidades de Forth solamente poseen solamente un prólogo (a diferencia de los lenguajes orientados a objetos), no es posible que sean funciones de todas las máquinas virtuales que realizan determinado proceso sobre la imagen. Sin embargo sí es viable que tengan como prólogo una llamada al procesador que se usa con más frecuencia, que es el que dibuja los segmentos en la pantalla. Consecuentemente, no existe un comando explícito para dibujar segmentos, simplemente se escribe el nombre del segmento en la terminal y el intérprete externo de Forth ejecuta la función como cualquier otra.

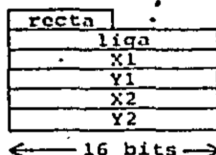


Figura 4. 2. Entidades de tipo recta

4.1.2 Segmentos de rectas

La estructura de las entidades de tipo segmento de recta se muestra en la fig 4.2. (Note que la palabra *segmento* se está empleando con dos significados: hasta ahora la habíamos utilizado para denotar conjuntos de entidades, es decir rectas, texto o instancias; pero ahora se refiere a porciones de rectas. Para evitar confusión, denominaremos *rectas* a los segmentos de rectas.) Poseen cuatro campos en los que se almacenan las coordenadas de sus extremos. Son números enteros y denotan coordenadas absolutas en el espacio del segmento, como se mencionó en el capítulo 2. Asimismo, están referidas al *espacio de aplicación* y no al del *dispositivo*. Es decir, cuando el usuario indica los extremos de la recta sobre la pantalla, se aplica la transformación inversa a la transformación de visión (pues el usuario pudo haber ordenado una amplificación o una reducción al dibujo).

La definición de las rectas se hace en forma interactiva, o sea, se señalan sobre la pantalla los extremos. El sistema calcula los valores de las coordenadas en el espacio de aplicación y dibuja la recta a fin de mostrar el resultado del programa gráfico que está editando el usuario. Es común que una vez generada una recta, la siguiente tenga un extremo coincidente con la anterior. El paquete presupone entonces que si solo se indica un punto de la recta, toma el extremo faltante de la recta anterior.

Como en cierta manera el paquete de graficación es un editor, deben existir comandos para modificar los programas. En el caso de rectas nos interesa alterar sus extremos. Para ello se proveyó al paquete de una función que "pega" el cursor a uno u otro extremo de la recta. En este modo de operación, cada vez que el usuario mueve el cursor, el sistema borra la recta y la dibuja nuevamente para actualizar el resultado del archivo gráfico, obteniendo un efecto conocido como banda de caucho [Newman 79].

4.1.3 Cadenas de texto

Las entidades que representan cadenas de texto contienen las coordenadas del *centro* de la cadena y un apuntador a la cadena misma. Lo más ortodoxo es incluir las coordenadas del extremo inferior izquierdo del letrero, pero cuando se utiliza el paquete de graficación para describir circuitos, esta convención resulta inadecuada si el sistema permite giros de segmentos. El texto no tiene que ser de gran longitud para que al girar la instancia de un segmento los letreros se traslapen con otras entidades haciendo que se pierda información en dispositivos con un solo nivel por pixel. En casos extremos, como por ejemplo el dibujo de un chip con

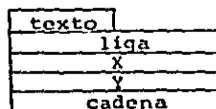


Figura 4.3 Entidades de tipo texto

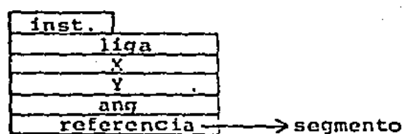


Figura 4.4 Entidades de tipo instancia

muchas patas en cada una de las cuales está el nombre de la pata, al girarlo es posible que los letreros queden más cerca de *otras* patas; efecto que es peligroso. SUDS, por ejemplo, soluciona este problema asociando dos vectores a los objetos de tipo texto. Una solución más sencilla consiste en escoger el centro, pues aunque también se presentan ocasiones en que se traslapa el letrero girado con otros objetos, ocurre con menos frecuencia. La fig 4.3 representa los nodos de este tipo.

4.1.4 Instancias de segmentos

Poseen la información que hace diferente a una instancia de otra. El sistema permite traslaciones y giros, por lo que aparte del apuntador al segmento del que son instancias, los nodos de este tipo tienen almacenadas las coordenadas absolutas de la instancia y un byte que denota el ángulo con respecto al eje horizontal.

También hay manera de modificar ambos parámetros interactivamente. Una vez señalado un segmento, el usuario ordena al sistema localizarlo, con lo que se dibuja con líneas punteadas el descendiente inmediato del segmento en edición. Si en realidad el diseñador desea alterar el desplazamiento de todos los objetos agrupados en el descendiente mostrado por el sistema, ordena que se pegue el cursor a la instancia y el paquete se encarga de actualizar el programa gráfico en la pantalla borrando y redibujando la instancia cada vez que el cursor se mueva. Asimismo, existe un comando que gira 90 grados la instancia. Para dibujar diagramas lógicos basta con este ángulo, pero como este parámetro ocupa un byte, es innecesario alterar el formato de los nodos en caso de emplear el paquete en otras aplicaciones que ameriten giros a más resolución. Probablemente sea adecuado un comando que gire gradualmente la instancia hasta que adquiera el ángulo deseado. En la fig 4.4 se aprecia un diagrama de las entidades tipo instancia.

Estas son las entidades principales del sistema. No contienen información eléctrica, pero existen otros tipos de datos que son variantes de las descritas, que se describirán en la sección del paquete de conexiones.

4.1.5 Subrutinas para dibujar segmentos

El procesador que genera las imágenes de los segmentos sobre la pantalla tiene la estructura común a casi todas las máquinas del sistema descrita en el capítulo 2; es decir, consiste en una subrutina recursiva que recorre el árbol en preorden.

En caso de visitar una instancia de segmento, procede de manera similar a la llamada a una subrutina de un lenguaje de programación almacenando en una pila el contador de programa (que en nuestro caso es un apuntador al nodo que está siendo interpretado) y realiza cierto proceso sobre el registro para luego llamarse a sí misma bajando un nivel en el árbol y procesar todos los hijos; concluido lo cual seguirá con el hermano del nodo que abandona. La subrutina recibe como parámetros de entrada los elementos de la transformación que debe aplicar a la instancia (que consiste solamente del desplazamiento y del ángulo) y un apuntador al segmento. Antes de bajar un nivel en la jerarquía, el proceso que realiza la subrutina es el cálculo de la composición de la transformación que recibió como parámetro con la de la instancia que está visitando. Esta nueva transformación es la que hereda a los hijos. Es decir, que cuando el procesador llega a una hoja del árbol, el objeto se verá afectado por la transformación que resulta de concatenar la transformación de todos sus ancestros. Las transformaciones del dispositivo y de visión pueden concebirse como transformaciones que están un nivel todavía más arriba que el segmento que se dibuja. Al regresar el control de flujo a la subrutina recursiva, el procesador debe restaurar la transformación que empleó en el primer hijo y aplicarla secuencialmente al resto de los hermanos. En lenguajes que permiten parámetros y valores locales en subrutinas, el almacenamiento de la transformación y su recuperación de la pila se hace automáticamente. Se muestra un listado en notación tipo Pascal.

```
procedure dibuja(a : byte; x, y integer; n : pointer);
  begin
  while n <> nil do begin
    if recta(n) then dibujarecta(aplica(a,x,y,n))
    else if texto(n) then dibujatexto(aplica(a,x,y,n))
    else begin
      concatena(a,x,y,n);
      dibuja(a,x,y,n);
      a := a.hermano;
      end
    end
  end
```

aplica regresa un nodo al que se le ha aplicado la transformación formada por **(a,x,y)** y **concatena** llama por referencia a **(a,x,y)** calcula la composición de la transformación

que contiene n y la regresa en (a,x,y). Pascal no permite que una subrutina regrese varios valores, como en el caso de aplica, pero Forth sí; por lo que habría que modificar un poco el listado para que fuera aceptado por un compilador de Pascal.

4.1.6 Localicación de segmentos

El paquete localiza objetos con dos comandos, ambos de ellos son similares al procesador que dibuja. El primero se usa para modificar instancias y también recorre el árbol, a diferencia de que debe recordar el nodo inmediatamente anterior al que señala el usuario. Se programaron con dos subrutinas; una para el primer nivel que almacena el nodo anterior a n en una variable global cuando encuentra la instancia y otra para los demás niveles del árbol. Además, la subrutina recursiva, que se llama localiza, al igual que localizarecta y localizatexto regresa un valor, dependiendo si el cursor se encuentra sobre el objeto o no. La condición de terminación ya no consiste únicamente en haber llegado al final de la lista ligada, sino que también se pregunta por la respuesta del descendiente. En caso de ser afirmativa, significa que se ha encontrado la entidad que señala el usuario y se suspende la ejecución de la subrutina. La función que recorre los niveles inferiores del árbol, simplemente regresa el mismo valor que recibió de su descendiente y la que visita el nivel superior de la estructura, si obtiene un valor afirmativo, procede a dibujar con líneas punteadas (o con algún color especial en un dispositivo cromático) el segmento encontrado.

El señalamiento de las entidades primitivas se realiza con un algoritmo de recorte parecido al que se usa para recortar la imagen (se utiliza el algoritmo de Sutherland y Cohen [Newman 79]), con la diferencia de que la ventana contra la que se recorta se establece unos cuantos pixeles alrededor del cursor. Si la región que rodea al cursor es demasiado grande, frecuentemente se localizarán segmentos no deseados, y si es demasiado chica, es posible que el usuario emplee demasiado tiempo colocando el cursor con suficiente precisión; es más, quizás ambas desventajas se traslapen y probablemente no exista ningún tamaño adecuado para la ventana.

SUDS, al igual que otros paquetes interactivos, resuelve el problema calculando la distancia de los segmentos al cursor. Así, basta con que el usuario acerque el dispositivo de señalamiento a la entidad gráfica. Sin embargo, el proceso puede resultar demasiado tardado. Es difícil resistir la tentación de calcular el cuadrado de la distancia para evitar la raíz cuadrada, pero en una máquina que almacena enteros en 16 bits, el programador no se puede dar el lujo de obtener números demasiado grandes. De hecho, debe evitar lo más posible algoritmos que requieran el producto de coordenadas. Si se utiliza el cuadrado de la distancia, es suficiente un dibujo con rectas con una longitud del tamaño de la pantalla para provocar un sobreflujo en los enteros.

El algoritmo de Sutherland y Cohen es más rápido que el que calcula el cuadrado de la distancia y puede lidiar con números bastante grandes sin necesidad de recurrir a la doble precisión (enteros de 32 bits); pero persiste el problema del tamaño de la ventana. La mayoría de los dispositivos de entrada permiten un movimiento casi continuo del cursor y el usuario puede, si tiene suficiente cuidado, posicionarlo sobre cualquier pixel. Pero esta es una característica heredada del dibujo sobre papel; en realidad, el usuario trabaja satisfactoriamente aunque solo pueda mover el cursor en una cuadrícula de baja resolución, digamos

10 o 15 pixeles. De esta manera se coloca el cursor con bastante rapidez pues las instancias de segmentos fueron colocadas dentro de la misma cuadrícula. (Claro está que la cuadrícula es invisible.) Conviene de cualquier manera que sea factible alterar la resolución del movimiento con un comando. En la máquina empleada para el programa de diseño se carecía de un bastón (joystick), por lo que el cursor se posiciona con comandos de teclado, pero la misma idea se puede extender a bastones y plumas de luz.

El otro procesador que sirve para localizar, es similar al primero y se emplea para señalar patas de compuertas. A diferencia de él, además de hacer que el sistema muestre una hoja en lugar de un descendiente inmediato del segmento, cuenta el número de patas que tuvo que visitar antes de encontrar la pata que busca. Las patas de las compuertas funcionan en forma parecida a los parámetros de las subrutinas y se distinguen dentro de los nodos de tipo *compuerta* por su posición. Es decir, la posición que ocupan dentro del nodo debe ser la misma que tienen en la lista ligada contando solamente los nodos de tipo *pata*. No es posible, entonces, agregar o eliminar patas de compuertas cuando ya han sido incluidas instancias de ellas en algún segmento.

4.1.7 Dispositivo de salida

El sistema es bastante independiente del dispositivo de salida. La interfaz con el dispositivo de graficación debe poder generar rectas y cadenas de texto en varios modos. El modo normal los pixeles que se prendan se pueden saturar. En dispositivos con un solo bit por pixel, se fuerzan a uno y en los de varios niveles por pixel se asignan a un valor determinado. El modo de borrado, similarmente, fija el valor de los pixeles de manera que se apaguen. El modo que llamaremos de realce se emplea durante el señalamiento. Si se carece de más de un nivel por pixel, basta con generar los objetos con rectas y caracteres punteados, aunque no está de más emplear un color especial. El modo de arrastre se utiliza cuando se modifican los parámetros de las entidades. El sistema de despliegue debe poder dibujar segmentos sin alterar el resto del dibujo. Este modo se implantó invirtiendo el valor de los pixeles que se alteran. En pantallas con más de un plano en la memoria de la imagen (frame buffer), los modos normal y de arrastre pueden ser el mismo.

La generación de rectas y texto se programó en ensamblador lográndose una velocidad satisfactoria. Las rectas se dibujan con el algoritmo de Bresenham [Newman 79] y los caracteres simplemente se copian (byte por byte) a la memoria de la pantalla.

4.1.8 Dispositivo de entrada

Las subrutinas que interactúan con el dispositivo de entrada gráfica son bastante independientes del resto del sistema. Básicamente consisten en una que lee la posición del cursor (en coordenadas del dispositivo de salida) y otra que permite al sistema colocar el cursor en un pixel determinado. El movimiento del cursor se debe efectuar discretamente a resolución variable como se indicó al describir la localización de segmentos.

4.1.9 Interacción con el disco

Debido a que el archivo gráfico emplea apuntadores, no es posible almacenarlo en el disco tal cual se encuentra en memoria. Si por alguna razón el sistema de diseño altera su tamaño (por ejemplo al agregar un nuevo comando), el archivo gráfico ya no quedará en la misma posición de memoria la siguiente vez que se cargue del disco. Se necesita entonces, un formato que sea independiente de la posición, aunque pierda algunas propiedades que sí son importantes cuando se contiene en la memoria principal; como son la facilidad de inserción de entidades y acceso rápido a los segmentos a partir de sus instancias.

Este nuevo formato carece de apuntadores físicos. Las listas ligadas se transforman en arreglos, que no es necesario guardar en la memoria; se transfieren al disco secuencialmente conforme se generan. Los apuntadores a los segmentos que poseen los nodos de tipo instancia, por otro lado, se reemplazan por el nombre del segmento al que apuntan (o más bien la parte del nombre que guarda Forth, que son los tres primeros caracteres y el número de caracteres); es decir, por apuntadores lógicos.

El procesador que genera el archivo gráfico en disco también recorre en preorden la estructura pero asocia otro significado a las instrucciones que la forman.

Al reconstruir el archivo gráfico en la memoria, se emplea una subrutina que realiza la traducción inversa y regenera la estructura al formato original.

Este método contiene un efecto indeseable si se emplea un diccionario incapaz de determinar la posición de una entidad antes de que aparezca su definición. Por ejemplo, si se utiliza una lista ligada de entidades para implantar el diccionario, como es el caso del intérprete de Forth, al aparecer en el archivo procedente del disco la instancia de un segmento que no ha sido definido todavía, el sistema se encuentra en un conflicto que no puede solucionar, porque no sabe de qué tamaño va a ser el segmento que falta. Si se emplea un diccionario con un mecanismo de acceso basado en una función de dispersión (hashing), el efecto no se produce porque, aunque sí es posible que la posición de las entidades dentro del diccionario dependa del orden en el que aparezcan al reconstruir el archivo gráfico, al momento en que surja la definición del segmento al que se hizo referencia anteriormente, la función de dispersión encontrará la posición asignada originalmente. El problema se resolvió estableciendo la regla de que todas las instancias de segmentos deben referirse a entidades que ya existan en el diccionario.

4.1.10 Listado fuente del archivo gráfico

Al diseñar el paquete de graficación se hizo notar que el hecho de que la imagen sea estructurada, viola en cierta medida el principio de los paquetes interactivos: que el usuario debe poder recibir toda la información pertinente del archivo gráfico a partir de la imagen que genera. Al examinar una imagen, no es posible deducir su estructura. Esta deficiencia se compensa incluyendo un comando que indique por lo menos los nombres de los segmentos que aparecen en el diccionario y los nombres de las instancias que contienen. Este procesador recorre no solo un segmento, sino todos los que existan. Es más, no le interesa visitar estructuras jerárquicas; solamente mostrar un resumen de cada segmento. Se trata de un resumen porque no es relevante el valor de las coordenadas de las rectas, basta con que indique el tipo de nodos que forman a un segmento y los nombres de las instancias.

4.1.11 Vocabularios

El comando que lista un resumen del diccionario, por simplicidad muestra el diccionario completo (solamente imprime las entidades gráficas a pesar de que comparten el diccionario con todas las entidades de Forth). En ocasiones el diseñador no desea examinar las características del conjunto completo de segmentos sino solamente determinado subconjunto, formado quizás por los segmentos más recientemente introducidos al sistema. Prueba de que conviene listar las entidades selectivamente son los comandos de los sistemas operativos concernientes al listado del directorio de archivos. Es común que exista alguna manera de especificar un subconjunto de archivos indicando parcialmente los nombres con una cadena que casa con varios de ellos.

Los vocabularios de Forth permiten realizar una función similar pero con subconjuntos previamente determinados. Se establecieron tres vocabularios de entidades gráficas: el de símbolos, el del catálogo y el del circuito. Cuando el usuario ordena listar el diccionario, solamente verá las entidades de su circuito, pero al dibujar segmentos, tendrá la capacidad de hacer referencia a los segmentos del vocabulario de símbolos. Durante la generación del dibujo, el sistema consulta dos vocabularios; el principal que es en el que se insertan los segmentos nuevos y el auxiliar, que se contiene segmentos usables desde el vocabulario principal. Cambiando de vocabulario, el diseñador puede examinar los otros dos e incluso modificarlos. También tiene la opción de crear nuevos vocabularios y escoger su estructura para que existan ciertos segmentos comunes a algunos de ellos.

4.2 Paquete de conexiones

4.2.1 Modelo del circuito

Al diseñar el paquete de conexiones se eligió emplear bi-digráficas para modelar los circuitos. En ellas, hay dos tipos de nodos: los que se asocian a los puntos eléctricos y los que representan elementos eléctricos (que hemos denominado compuertas). Existen tres maneras principales de representar gráficas dirigidas en la computadora [Pfaltz 77].

La representación con listas ligadas es la que requiere más tiempo para localizar las conexiones, pero a la vez es la más flexible para modificar la estructura. Los vértices de la gráfica se modelan con registros de tres campos: un apuntador a una lista ligada de arcos que salen del vértice, un apuntador a la lista de arcos que entran y otro más que contiene la información propia del vértice. El formato de los registros de los arcos posee cinco campos: dos de ellos apuntan al siguiente nodo y al anterior (las listas son doblemente ligadas), otros dos contienen las direcciones de los vértices que relacionan y el último la información referente al arco mismo.

En el caso de gráficas no dirigidas, la matriz de adyacencia es una tabla simétrica con los nombres de los vértices tanto en las columnas como en los renglones y ceros en la diagonal principal. Cada conexión presente entre un par de vértices implica un uno en el lugar correspondiente a ambos vértices y una ausente un cero. En gráficas dirigidas, el sentido de

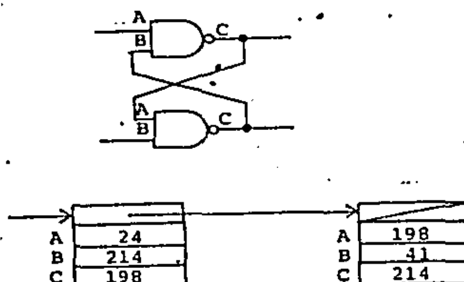


Figura 4.5 Representación de las conexiones de un circuito

los arcos se acostumbra almacenar en el signo de los unos dando como resultado una matriz antisimétrica.

La matriz de incidencia consiste en una tabla en la que cada renglón está asociado a un arco y cada columna a un vértice. Los elementos toman los valores $0, +1$ y -1. El elemento b_{ij} es diferente de cero si y solo si el vértice i es incidente al arco j. Cada renglón contendrá, por lo tanto, todos sus elementos iguales a cero con excepción de uno con valor -1 y otro con valor $+1$, dependiendo del vértice al que entra y del que sale el arco.

Aunque el acceso a la información es más rápido en las dos últimas representaciones que en la que emplea listas ligadas, no son útiles para nuestra aplicación si no se introducen modificaciones debido a que carecen de datos para poder distinguir entre las diferentes patas de una compuerta. La representación con listas ligadas se descartó por la dificultad que implican los algoritmos para alterar la estructura y porque no ser suficientemente compacta. Se eligió una variante de los modelos descritos que puede verse como una variante tanto de la representación ligada como de la matriz de incidencia.

Por un lado, se eliminaron de la tabla los vértices de tipo punto eléctrico haciendo permanecer solamente a los nodos de tipo compuerta. Por otro, en lugar de asociar un renglón a cada arco de la gráfica, los renglones de la tabla representan *patas* de compuertas. Además, los valores de los elementos de la matriz consisten en los *nombres* de los vértices que sean puntos eléctricos. Es decir, que no todas las columnas (compuertas) tienen el mismo número de renglones pues depende del número de patas. No es frecuente que durante el diseño de un circuito cambie el número de patas de una compuerta y consecuentemente las columnas de la tabla se implantaron con *registros* uno de cuyos campos es un arreglo. El conjunto de columnas, en cambio, sí estará constantemente variando, por lo que se consideró más adecuada una implantación con listas ligadas.

El modelo puede describirse también en términos de la representación con listas ligadas. En vez de tener una lista doblemente ligada por cada conjunto de arcos incidentes en los vértices, el sistema almacena arreglos con los nombres de los arcos. La fig 4.5 muestra un ejemplo de la representación de un circuito.

alam.
liga
X1
Y1
X2
Y2
conexión

Figura 4.6 Nodos de tipo alambre

La representación posee la información necesaria para distinguir una pata de otra y es muy compacta. También es rápido el acceso cuando el sistema pregunta por las conexiones de una compuerta dado el nombre de la compuerta. Sin embargo, si el programa requiere los nombres de las compuertas conectadas a un alambre dado un punto eléctrico, el acceso es muy lento ya que se ve forzado a recorrer *todo* el circuito y preguntar a cada compuerta si está conectada al punto pertinente.

Esta deficiencia puede corregirse calculando la *inversión* de las conexiones construyendo una tabla en la que la llave de entrada es el nombre de una conexión. La información se tendría duplicada y los cambios deben actualizarse en ambas tablas, a cambio de un acceso muchas veces más rápido. Dado que el programa para diseño de circuitos funciona en un sistema computacional pequeño, se consiredó innecesario almacenar la tabla invertida. De por sí el sistema no tendrá la capacidad de almacenar circuitos grandes y aunque recorra todo el circuito lo hará en poco tiempo, pero para la implantación de un programa que funcione en una computadora mayor sí vale la pena tener en cuenta la tabla invertida.

4.2.2 Alambres

Los alambres se modelan con entidades gráficas similares a las rectas, pero con un campo adicional que contiene el nombre de la conexión en caso de estar conectada. Si se quiere, con compuertas con una sola pata, aunque no intervienen para generar la lista de conexiones. Su uso reside únicamente en la determinación del nombre de la conexión cuando el diseñador señala un alambre. La fig 4.6 ilustra un nodo de tipo alambre.

Los nombres de las conexiones consisten en números enteros que el sistema genera cada vez que aparece un nuevo punto eléctrico. Si ya existe alguna pata que sea salida conectada a él, su signo será negativo y si no, positivo. Note que cuando el diseñador conecta o desconecta una salida a un punto previo, el sistema debe recorrer todo el circuito para cambiar de signo a todos los alambres y patas conectados a ese punto. Sin embargo, la verificación de no conectar dos salidas es sencilla, basta con examinar el signo de ambas conexiones.

4.2.3 Compuertas y subcircuitos

Las compuertas se modelan como segmentos; es decir como un conjunto de entidades, en una lista ligada. Las compuertas están restringidas a no tener conexiones internas

comp.
liga
X
Y
ang
referencia
conexión 1
conexión 2
conexión n

Figura 4.7 Estructura de las compuertas

y generalmente poseen algunas entidades de tipo *entrada* y *salida*, que son objetos idénticos a las rectas con excepción de que su campo de tipo es distinto.

Los subcircuitos son compuertas que sí contienen conexiones internas y también se almacenan como segmentos. Al diseñar el paquete de graficación se solucionó el problema de modelar subcircuitos respetando la unicidad en la memoria. Sin embargo, las funciones que capacitan al sistema para hacerlo no están implantadas pues una vez estimada su complejidad, se concluyó que ocuparían demasiada memoria y que sería de mayor utilidad práctica excluirlas a cambio de poder almacenar más variedad de segmentos, digamos símbolos y entidades de catálogo. Consecuentemente, el usuario sí dispone de subcircuitos como entidades, pero está sujeto a generar una sola instancia de ellos.

4.2.4 Instancias de compuertas

Las instancias de compuertas son extensiones de las instancias de segmentos gráficos. Se han agregado tantos campos como patas tenga la compuerta y por supuesto un indicador del número de patas. Este dato se almacena en el mismo byte que contiene el tipo de nodo; es decir, que además de los tipos de nodos ya mencionados, hay tantos tipos de entidades como compuertas con número distinto de patas existan en el circuito. Las instancias de compuertas corresponden a las columnas de la tabla descrita en la sección previa, y contienen los nombres de las conexiones de cada pata. También poseen campos para guardar el número de chip y el número de cada pata. La fig 4.7 muestra la estructura de una compuerta (se ha omitido la

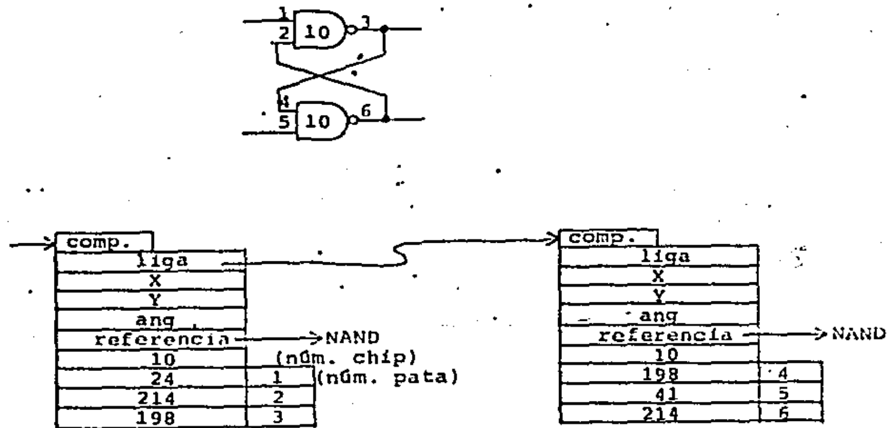


Figura 4.8 Modelo de un circuito

numeración de la compuerta y sus patas por claridad). La fig 4.8 muestra la representación completa de un circuito.

4.2.5 Numeración de patas y generación de la lista de conexiones

Antes de producir la lista de conexiones, el circuito debe ser numerado. Este proceso consiste en asignar un número a cada compuerta y a cada pata. La lista de conexiones estará en términos de los valores asignados. El sistema consulta el vocabulario del catálogo, que está estructurado de manera que cada chip es un segmento que contiene determinado número de entidades de tipo compuerta. Las compuertas del catálogo deben ser numeradas "a mano" de acuerdo con los datos del manual. La función que asigna los números de patas aprovechará las compuertas libres en caso de encontrar compuertas no numeradas y chips parcialmete usados; es decir, agrupará compuertas utilizando el mínimo número de chips. Además está programada de manera que supone que al ser llamada, ya existen algunas compuertas numeradas y faltan algunas por numerar, por lo que puede ser llamada aunque el circuito haya sufrido cambios.

La lista de conexiones se obtiene precisamente invirtiendo la tabla de conexiones pues se desea un conjunto de puntos eléctricos formados por listas de patas.

4.2.6 Macros

A pesar de ser un sistema compacto, por el hecho de haber empleado un lenguaje de programación interactivo, el usuario puede fácimente llamar funciones del sistema desde subrutinas que introduce a la vez que está diseñando su circuito. Otros paquetes de diseño denominan macros a estas subrutinas, pero en el ambiente de Forth no se procesan como macros pues no existe ninguna expansión ni se interpretan como si el usuario estuviera escribiendo los comandos en la terminal. Forth compila y almacena estas funciones como cualesquiera otras.

Mientras más "alto" sea el nivel de abstracción del modelo del circuito que contenga el sistema, mayor será su contenido semántico y existirá mayor facilidad para realizar cambios al modelo. Si el programa almacena un conjunto de puntos eléctricos como una entidad (un "bus"), el diseñador podrá conectar o desconectar varios alambres con un solo comando. El sistema sí realiza abstracción en subcircuitos pero no en puntos eléctricos; es decir, carece del concepto de bus. Debido a la interactividad de Forth, el diseñador tiene la opción de generar un comando que conecte o desconecte un bus particular. Sin embargo, el modelo no "sabrá" que se trata de un bus. Los macros son siempre parches a los lenguajes y es conveniente usarlos con reserva. Cuando se incluyen macros en un sistema, se está admitiendo, hasta cierto punto, que el lenguaje no es capaz de realizar determinadas abstracciones. Aun así, puede ser un mecanismo fácil de implantar que resuelve algunos problemas de abstracción, como es el caso de los buses.

Capítulo 5

Conclusiones

5.1 Resultados

El desempeño del sistema es satisfactorio. Efectivamente modela circuitos a partir de comandos de un paquete interactivo de graficación y resulta ser una ayuda para el diseño y la documentación de sistemas digitales.

El paquete de graficación ya es por sí mismo una técnica útil para almacenar los diagramas de los circuitos, pues equivale a un editor de dibujos que, debido a que las imágenes son estructuradas, permite realizar cambios a los diagramas fácilmente.

Además, el modelo contiene información de las conexiones que capacita al sistema en la extracción de datos sobre el circuito que normalmente requieren un tiempo considerable del diseñador si los obtiene directamente del diagrama lógico. La generación de la lista de conexiones junto con la lista de componentes consiste en una descripción de circuito suficientemente diluida para un paquete de diseño físico. También el programa de diseño verifica la consistencia de algunas características eléctricas como es el hecho de que no estén conectadas dos salidas. Un error común cuando se diseñan circuitos manualmente es la omisión de conexiones a las *entradas* de los componentes. Frecuentemente se incluyen chips deshabilitados por no haber conectado alguna pata, pero el sistema de diseño puede generar una lista con las entradas no conectadas impidiendo que ocurra este error. Asimismo, tiene la información necesaria para que a petición del usuario, produzca una lista con las compuertas libres existentes en los chips del circuito.

El programa funciona en una computadora con 64 Kbytes de memoria, por lo que fue necesario sacrificar determinadas características que si bien no son indispensables, sí conviene implantarlas en sistemas computacionales de mayor capacidad. El programa ocupa unos 10 Kbytes y el sistema operativo de la máquina empleada (MCZ 1/05 de Zilog) requiere otro tanto,

por lo que deja 32 Kbytes para el área de trabajo.[1] En ella se almacena el modelo del circuito, el diccionario de símbolos y la información del catálogo. Es suficiente para diseñar, por ejemplo, un circuito con 50 chips, con un conjunto de 30 símbolos y un catálogo de alrededor de 20 circuitos integrados distintos. Estas capacidades dependen, por supuesto del número de patas de los chips, del número de compuertas que contienen y de qué tan compleja sea la estructura jerárquica del circuito; pero en general bastará para diseñar una tarjeta de regular tamaño.

5.2 Extensiones y trabajo futuro

Dadas las dimensiones de la computadora empleada, no parece conveniente realizar extensiones al sistema si se pretende resolver problemas prácticos, pues al agregar más código, algunas características se verían gravemente afectadas. Disminuiría, por ejemplo, el tamaño del área de trabajo y por lo tanto el tamaño del circuito o el del catálogo. Se efectuó un experimento generando una versión de Forth con memoria virtual. Forth es un lenguaje que procesa durante la ejecución de la misma manera las funciones primitivas (escritas en ensamblador) y las secundarias (escritas en Forth); por lo que admite redefinir las funciones primitivas por código en Forth. Si se redefinen la asignación y la indirección (que son funciones de Forth), de manera que el nuevo código contenga una llamada a la subrutina de memoria virtual, se obtiene memoria virtual sin necesidad de reprogramar el paquete de diseño. Todos los apuntadores siguieron siendo de 16 bits, por lo que el espacio de direccionamiento no se vió incrementado, pero sí hacía que el sistema aprovechara los 16 Kbytes del sistema operativo intercambiando información con el disco. Sin embargo, el tiempo de respuesta del sistema resultó muy deteriorado debido a las llamadas tan frecuentes a la subrutina de memoria virtual: aumentó unas cuatro o cinco veces. Se decidió por lo tanto, restringir el tamaño del código al que ocupa actualmente, pues los 16 Kbytes que se ganan no amerita una reducción tan grande en la velocidad.

Comparando el sistema con SUDS, contiene algunas características que lo colocan en una posición ventajosa sobre todo en lo que respecta al paquete de graficación. Ellas radican en el hecho de almacenar imágenes estructuradas y el empleo de coordenadas absolutas (que eliminan las rectas invisibles). SUDS posee un paquete de conexiones más poderoso y es capaz de procesar macros, pero los macros del sistema elaborado no dejan nada qué desear. El usuario de SUDS tiene acceso a una biblioteca de circuitos integrados muy grande y puede generar el circuito impreso a través de un paquete de diseño físico. Sin embargo, nuesto sistema funciona en una microcomputadora pequeña y es mucho más transportable.

5.2.1 Entidades

Los tipos de entidades que reconoce el modelo de circuitos son adecuados para la mayoría de las aplicaciones, pero hay una entidad que no se planteó al diseñar el modelo que sí podría ser útil que es el conjunto de puntos eléctricos (buses). Si el sistema pudiera concebir

[1] el dispositivo de despliegue emplea su propia memoria de video

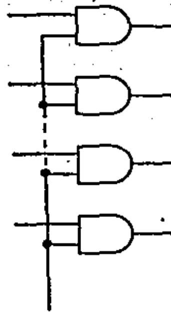


Figura 5.1 Operación para la cual se requiere un mejor modelo

este tipo de objeto, el usuario tendría un lenguaje más abstracto para describir y modidifar conexiones. A pesar de que los macros alivian en cierta medida esta carencia, no dejan de ser un parche al lenguaje descriptor de circuitos y una solución más limpia sería la inclusión de buses como entidades en el modelo. No obstante, esta extensión implica dificultades en el paquete de graficación que parece adecuado afrontarlas solamente en caso de que el programa residiera en un sistema computacional mayor.

Cuando el usuario ordena la conexión de dos alambres, el programa responde calculando la trayectoria de la conexión generando con un algoritmo sencillo hasta tres rectas horizontales o verticales para que el diagrama refleje la nueva conexión. De igual manera, si se modelaran conjuntos de alambres, el sistema debería reconstruir el diagrama cuando ocurrieran cambios en las conexiones de los buses para lo cual necesita un modelo de las conexiones más acorde con el diagrama lógico.

5.2.2 Gráfica de conexiones

El modelo de conexiones emplea una bi-digráfica con compuertas asociadas a un tipo de vértices y puntos eléctricos al segundo tipo. Este modelo posee toda la información para generar la lista de conexiones, pero resulta inadecuado para reconstruir el diagrama lógico a partir de la gráfica. Los diagramas generalmente permiten a un mismo punto eléctrico estar representado con varios nodos.

Es claro que el modelo debe extenderse en este sendido para que haya una mayor correspondencia con el diagrama que percibe el diseñador. Con un modelo así, resuelven problemas expuestos anteriormente como la incapacidad que tiene el sistema elaborado para desconectar dos subcircuitos. La fig 5.1 muestra este comando. Supongamos que el usuario decide eliminar el alambre que está dibujado en este ejemplo con una línea punteada; el sistema debe actualizar la lista de conexiones creando un nuevo punto eléctrico, pero no puede hacerlo

con el modelo que se utilizó. El sistema elaborado conserva la conexión lógica de ambos puntos mientras que la extensión propuesta otorga la capacidad de ejecutar este tipo de comandos.

5.2.3 Unicidad de subcircuitos

Es evidente también que el sistema debe poder aplicar la solución mencionada en la sección 2.1.3, en donde se sugiere almacenar los segmentos con conexiones en un solo lugar en la memoria. Se obtiene una representación compacta del circuito y se actualizan automáticamente los cambios hechos a un subcircuito en todas sus instancias. El sistema puede almacenar imágenes con unicidad de segmentos, pero siempre y cuando no contengan conexiones internas. Sin embargo, la solución es sencilla: consiste en generar la lista de conexiones asignando un nombre a los puntos eléctricos y a las compuertas formado por la concatenación de los identificadores de todos sus ancestros.

5.2.4 Lista invertida de conexiones

La respuesta del sistema mejoraría considerablemente si además de almacenar el modelo de conexiones en una tabla a la que se accesa por medio de los identificadores de las computeras, se guardara otra tabla accesible por medio de identificadores de puntos eléctricos. Se tendría entonces la información duplicada en una lista invertida. Para las dimensiones de la computadora empleada esto podría significar demasiada memoria pero sí es aconsejable en sistemas mayores.

5.2.5 Extensiones a Forth

Habría sido conveniente extender Forth para que pudiera compilar funciones cuyos parámetros tuvieran nombre. Este lenguaje hace referencia a los parámetros y a las variables locales según la posición que ocupan en la pila, pero no es demasiado complicado eleborar un compilador en Forth mismo que reconozca sus nombres obteniendo un código legible.

5.2.6 Lenguaje orientado a restricciones

Aun con todas las extensiones propuestas, habrá determinadas características que no estarán presentes en el sistema y que sí son deseables en un paquete interactivo para diseño auxiliado por computadora. Por ejemplo, si se altera la separación entre las patas de una compuerta, aparecerán desconectados los alambres en todas sus instancias del diagrama. Asimismo, si el usuario traslada un subcircuito en el diagrama, a menos que el sistema esté programado para resolver este problema, las conexiones se desprenderán del subcircuito y se perderá la correspondencia entre el modelo de conexiones y el diagrama lógico. (Estamos suponiendo que el usuario solo pretende desplazar el subcircuito.)

Resolover este tipo de problemas representa una dificultad seria; es aconsejable establecer una metodología que abarque a todos ellos. Los lenguajes orientados a restricciones, como ThingLab ofrecen esta posibilidad.

Bibliografía

[Baker 77]
Baker H. G. Jr.
Lisp processing in real time on a serial computer,
MIT Artificial Intelligence Laboratory
A. I. working paper 139, abril 1977.

[Bauer 81]
Bauer F. L. et al,
"Programming in a wide spectrum language: a collection of examples",
Science of Computer Programming,
North-Holland 1 (1981) pp 73-114

[Bell 71]
Bell C. G. y Newell A.,
Computer structures: readings and examples,
McGraw-Hill, New York, 1971

[Bell 71a]
Bell C. G. y Grason J.,
"The register transfer module design concept",
Computer Design,
mayo 1971, pp 87-94

[Bell 73]
Bell J. R.
Threaded code,
Communications ACM, Vol 16, N 6, junio 1973, pp 370-372

[Blakeslee 79]
Blakeslee T. R.

Digital design with standard MSI and LSI,
John Wiley, 2ª ed, U. S. A., 1979

[Borning 81]
Borning A.
The programming language aspects of ThingLab, a constraint-oriented simulation laboratory,
ACM Trans. on Programming Languages and Systems, vol 3, n 4, octubre 1981, pp 353-387

[Breuer 72]
Breuer M. A. (ed.),
Design Automation of Digital Systems—Theory and Techiques,
Prentice-Hall, Englewood Cliffs, N. J., 1972

[Breuer 81]
Breuer M. A., Friedman A. D., y Iosupovicz A.,
"A survey of the state of the art of design automation",
Computer,
Vol 14, No 10, oct 1981, pp 58-75

[BYTE 80]
BYTE vol 5, n 8, agosto 1980
(número dedicado a Forth)

[BYTE 81]
BYTE vol 6, n 8, agosto 1981
(número dedicado a Smalltalk)

[Chu 65]
Chu Y.
"An Algol-like computer design language",
Comm. ACM,
Vol 8, No 10, pp 607-615, 1965

[Computer 74]
Computer,
Vol 7, No 12,
(número especial sobre lenguajes descriptores de circuitos),
diciembre 1974

[Computer 77]
Computer,
Vol 10, No 6,
(número especial sobre aplicaciones de lenguajes de circuitos),
junio 1977

[Computervision 80]

The CAD/CAM Handbook,
Computervision,
Machover C., Blauth R. E., ed.
Bedford, Massachusetts, 1980

[Copeland 82]
Copeland G.
What if mass storage were free?,
Computer vol 15, n 7, julio 1982, pp 27-35

[Dahl 66]
Dahl O. J. y Nygaard K.
SIMULA—An Algol-based simulation language,
Comm ACM vol 9, n 9, septiembre 1966, pp 671-678

[Dewar 75]
Dewar R. B. K.
Indirect threaded code,
Communications ACM, Vol 18, N 6, junio 1975, pp 330-331

[Director 81]
Director S. W., Parker A. C., Siewiorek D. P. y Thomas D. E.,
"A design methodology and computer aids for digital VLSI systems",
1980/81 CMU Computer Science Research Review,
CMU, 1981

[Duley 68]
Duley J. R. y Dietmeyer D. L.
"A digital system design language (DDL)",
IEEE Trans. Computers,
C17, pp 850-861, 1968

[Duncan 81]
Duncan R.
Forth decompiler,
Dr. Dobb's Journal, n 59, septiembre 1981, pp 49-53

[Friedman 69]
Friedman T. D. y Yang S. C.
"Methods used in an automatic logic design generator (ALERT)",
IEEE Trans. Computers,
C18, 593-614, 1969

[Foley 82]
Foley J. D. y van Dam A.
Fundamentals of interactive computer graphics,
Addison Wesley, 1982

[Greenberg 70]
Greenberg H. y Hegerich R. L.
A branch search algorithm for the knapsack problem,
Management Science, vol 16, n 5, enero 1970

[Hanan 72]
Hanan M. y Kurtzberg J. M.
Placement techniques,
capítulo 5 de [Breuer 72]

[Harary 69]
Harary F.
Graph theory,
Addison Wesley, 1969

[Harris 80]
Harris K.
Forth extensibility,
BYTE vol 5, n 8, agosto 1980, pp 164–184

[Hayes 80]
Hayes J.
MOS scaling,
Computer vol 13, n 1, enero 1980, pp 8–13

[Hill 73]
Hill F. J. y Peterson G. R.
Digital Systems: Hardware Organization and Design,
John Wiley, New York, 1973

[Hill 74]
Hill F. J. y Peterson G. R.
Introduction to Switching Theory and Logical Design,
2ª ed., John Wiley, New York, 1974

[Iverson 62]
Iverson K. E.
A Programming Language,
John Wiley, New York, 1962

[Kernighan 78]
Kernighan B. W. y Ritchie D. M.
The C programming language,
Prentice-Hall, New Jersey, 1978

[Kodres 72]
Kodres U. R.

Partitioning and card selection,
capítulo 4 de [Breuer 72]

[Kogge 82]
Kogge P. M.
An architectural trail to threaded-code systems,
Computer, Vol 15, N 3, marzo 1982, pp 22–32

[Kroenke 77]
Kroenke D.
Database Processing,
Science Research Associates, 1977

[Lafue 82]
Lafue G. M. E. y Mitchell T. M.
Data base management systems and expert systems for CAD,
Rutgers University, New Jersey, Laboratory for computer science research,
Technical Report LCSR-TR-28, mayo 1982

[Lee 61]
Lee C. Y.
An algorithm for path conections and its applications,
IRE Trans. on Electronic Computers, vol EC-10, n 3, pp 346–365,
septiembre 1961

[Lewin 77]
Lewin D.
Computer-aided design of digital systems,
Crane Russak, New York, 1977

[Liskov 77]
Liskov B., Snyder A., Atikson R. y Shaffert C.
Abstraction mechanisms in CLU,
Comm ACM vol 20, n 8, agosto 1977, pp 564–576

[Loelinger 81]
Loelinger R. G.
Threaded Interpretive Languages,
Byte Books, 1981

[Madhavji 81]
Madhavji N. H. y Wilson I. R.
Dinamically structured data,
Software—Paractice and Experience, vol 11, pp 1235–1260, 1981

[Mano 79]
Mano M. M.
Digital logic and computer design,

Prentice-Hall, N. J., 1979

[Medina-Mora 81]
Medina-Mora R. y Notkin D. S.
ALOE users' and implementors' guide,
Department of Computer Science, Carnegie-Mellon University,
CMU-CS-81-145, noviembre 1981

[Meinzer 79]
Meinzer K.
IPS, an unorthodox high level language,
BYTE vol 4, n 1, enero 1979, pp 146–159

[Newman 79]
Newman W. M. y Sproull R. F.
Principles of interactive computer graphics,
Mc Graw-Hill, 2ª ed., 1979

[Parker 81]
Parker A. C. y Hafer L. J.,
"Automating the design of testable hardware",
en *VLSI 81,*
Gray J. P. (ed.),
Adademic Press, Londres, 1981, pp 357–363

[Pfaltz 77]
Pfaltz J. L.
Computer Data Structures,
Mc Graw-Hill 1977

[PLZ/SYS 78]
Zilog, Inc.
Report on the programming language PLZ/SYS,
Zilog, California, 1978

[Prince 71]
Prince M. D.
Interactive Graphics for Computer-Aided Design,
Addison-Wesley, 1971

[Rentsch 82]
Rentsch T.
Object oriented programming,
SIGPLAN Notices vol 17, n 9, septiembre 1982, pp 51–57

[Rosa 79]
Rosa R. C.
Programs for two-sided printed circuit board design,

Computer-Aided Design,
vol 11, n 5, septiembre 1979, pp 297-303

[Rupp 81]
Rupp C. R.
"Components of a silicon system",
en *VLSI 81,*
Gray J. P. (ed.),
Adademic Press, Londres, 1981, pp 227-236

[Saaty 70]
Saaty T. L.
Optimization in integers and related extremal problems,
McGraw-Hill, 1970

[Shiva 79]
Shiva S. G.
"Computer hardware description languages—a tutorial",
Proc. IEEE,
Vol 67, No 12, diciembre 1979

[Sutherland 63]
Sutherland I. E.
Sketchpad. A man-machine graphical communication system,
Spring Joint Computer Conference, 1963, pp 329-346

[SUDS 80]
Newcomer J. M. (ed.),
SUDS Users' Manual,
Computer Science Department, Carnegie-Mellon University,
Pittsburg, Pa, 1980

[Thanh 82]
Thanh N. T. y Raschner E. W.
Indirect threaded code used to emulate a virtual machine,
SIGPLAN Notices vol 17, n 5, mayo 1982, pp 80-89

[Winston 81]
Winston P. H. y Horn B. K. P.
LISP,
Addison-Wesley, 1981

[Wirth 76]
Algorithms + data structures = programs,
Prentice-Hall, 1976

[Wulf 76]
Wulf W. A., London R. y Shaw M.
An introduction to the construction and verification of ALPHARD programs,
IEEE Trans. on Software Engineering, vol SE-2, n 4, diciembre 1976, pp
253-264

Comandos del sistema

ESTA TESIS NO DEBE SALIR DE LA BIBLIOTECA

1 glosario

vocabulario	conjunto de segmentos
segmento	conjunto de nodos asociado a un nombre (una entrada al diccionario)
nodo	línea, texto o instancia de segmento
línea	recta, pata o alambre
compuerta	instancia de segmento con patas conectables
módulo	segmento con compuertas. no puede haber más de una instancia de cada módulo
bloque	instancia de segmento que no se dibuja si la cvariable "blo" vale 1

2 funciones de usuario

2.1 funciones y variables de graficación

2.1.1 variables

nombre	tipo	explicación
icur	variable	incremento de las coordenadas del cursor en pixels
tex	cvariable	cuando vale 1 permite la impresión de texto. Es útil apagar esta bandera al usar escala menor a 1 puesto que los caracteres no cambian de escala (inicialmente vale 1)
bor	cvariable	cuando vale 1 borra la pantalla antes de invocar un segmento (inicialmente vale 1)
blo	cvariable	cuando vale 1 inhibe el dibujo de las instancias de tipo bloque (inicialmente vale 1)

Las "cvariables" son banderas que se prenden con la función "si" y se apagan con la función "no". Por ejemplo "tex si".

2.1.2 funciones

funciones del cursor y de la pantalla

arr	mueve el cursor "icur" pixels hacia arriba
abj	mueve el cursor "icur" pixels hacia abajo
der	mueve el cursor "icur" pixels hacia la derecha
izo	mueve el cursor "icur" pixels hacia la izquierda

las flechas oprimidas cuando el cursor de la terminal esté en la primera columna equivalen a las funciones arr, abj, der e izo.

<cvariable> si	asigna 1 a la cvariable <cvariable>
<cvariable> no	asigna 0 a la cvariable <cvariable>
<n> i!	asigna "icur" al <n>
tc	mueve el cursor al centro de la pantalla
ta	apaga la pantalla
tp	prende la pantalla
tz	borra la pantalla
tr	asigna 18 a icur
td	asigna 6 a icur
tu	asigna 3 a icur

funciones para crear segmentos

s: <nombre>	crea un segmento con el nombre que se dé a continuación. El segmento es una función cuyo código es dibujar la función y establecer el valor de la variable "seg-ed"
copia! <nombre>	copia el segmento en edición bajo el nombre <nombre>

funciones para insertar y quitar nodos

- **r** almacena las coordenadas del cursor con objeto de que sean el punto inicial de una línea (inicialmente está en el centro)
- **ar** agrega recta con punto final igual a la posición del cursor
- **ap-** agrega pata tipo entrada
- **ap+** agrega pata tipo salida
- **at <texto>** agrega un nodo de tipo texto con la cadena <texto> que termina con retorno de carro
- **ai <nombre>** agrega instancia del segmento cuyo nombre esté a continuación
- **ab <nombre>** agrega bloque del segmento cuyo nombre esté a continuación
- **ac <nombre>** agrega compuerta cuyo nombre esté a continuación
- **loc** establece el valor de la variable nodo-actual
- **e** elimina el nodo actual
- **sd** dibuja el segmento en edición después de borrar la pantalla (actualiza la pantalla)
- **s?** indica el nombre y el ángulo del nodo actual en caso de ser una instancia

funciones para modificar el nodo actual

- **gira** gira el nodo actual 90 grados
- **referencia <nombre>** cambia la referencia del nodo actual en caso de que sea instancia o compuerta por <nombre>
- **ir** invierte recta en caso de que el nodo actual sea línea
- **<a** amarra el nodo actual pegándolo al cursor. Si el nodo es una línea, el cursor se pega a uno de los extremos. Para arrastrar el otro extremo, usar la función "ir". Tiene una llamada implícita a loc
- **a>** desamarra el cursor del nodo actual

funciones para modificar todos los nodos del segmento en edición

- **<n> <d> x-escala** escala en x el segmento en edición. El numerador de la escala es <n> y el denominador <d>
- **<n> <d> y-escala** escala en y. <n> es el numerador y <d> el denominador de la nueva escala
- **xy** mueve el origen del segmento en edición a la posición del cursor

funciones que alteran la transformación de visión

- **<n> <d> esc** cambia la escala. <n> es el numerador y <d> el denominador de la nueva escala
- **despl** cambia el desplazamiento de manera que el nuevo centro de la pantalla sea la posición actual del cursor
- **0despl** elimina el desplazamiento

funciones relacionadas con el diccionario

- **l** lista los segmentos del vocabulario "current". Las rectas se indican con "|", las entradas con "-", las salidas con "+", los alambres con "/", los textos entre comillas, las instancias con el nombre del segmento del que son instancias, las compuertas con el nombre del segmento precedido de "*" y los bloques con el nombre del segmento precedido de "]". el listado se suspende al oprimir alguna tecla y se aborta al oprimir la tecla <esc>
- **rem <nombre1> <nombre2> ... <nombren>** remueve los segmentos cuyos nombres se den a continuación
- **-rem <nombre1> <nombre2> ... <nombren>**

remueve todos los segmentos excepto los que se den a continuación

renom <nombre actual> <nombre nuevo>
cambia el nombre del segmento <nombre actual> por <nombre nuevo>

funciones relacionadas con el disco

g <archivo> — guarda el vocabulario actual en un archivo cuyo nombre sea <archivo>
t <archivo> — trae el archivo <archivo> y lo agrega al vocabulario actual

función de impresión

imp — imprime los 248 bits izquierdos de la pantalla.
La impresión se suspende al oprimir alguna tecla y se aborta al oprimir la tecla <esc>

funciones relacionadas con conexiones

sim — asigna el vocabulario actual al vocabulario de símbolos
cat — asigna el vocabulario actual al vocabulario del catálogo
cir — asigna el vocabulario actual al vocabulario del circuito
c — localiza una pata o alambre y muestra sus conexiones
x — localiza una pata o alambre, muestra sus conexiones y las conecta a la pata señalada con el comando "c"
o — igual a "x", pero además calcula la trayectoria (conexión gráfica)
fg — crea un alambre que se pega al cursor (conexión guiada)
fq — despega el alambre pegado al cursor (termina conexión guiada)
tierra — conecta a tierra la pata actual
vcc — conecta a vcc la pata actual
d — desconecta la pata o alambre localizada con "c"
<n> pa! — asigna el número de la pata señalada con "c" a <n>
Solo la numera internamente; para que se numere en la pantalla, se puede usar la función "sd". numerar con 0 = desnumerar
ch! — numera el chip alguna de cuyas patas se señaló con "c". Solo se numera internamente. numerar con 0 = desnumerar
numera — numera todo el segmento en edición
desnumera — desnumera todo el segmento en edición
conexiones — imprime lista de conexiones
conexiones-archivo <archivo>
manda la lista de conexiones al archivo <archivo>
<num de patas> chip! <nombre>
crea un segmento con la forma de un chip cuyo número de patas (que debe ser par) sea <num de patas> y cuyo nombre sea <nombre>
>+ — si la pata actual es una entrada, la vuelve salida
>- — si la pata actual es una salida, la vuelve entrada

Ejemplo

Ejemplo de una sesión

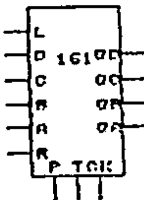
El usuario cuenta con tres vocabularios (es decir, conjuntos de segmentos) para diseñar circuitos. Al iniciar la sesión conviene bajar a la memoria la información que ya exista referente a símbolos, datos de catálogo o el circuito que esté en proceso de diseño.

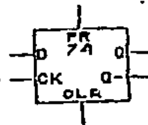
El vocabulario "sim" contiene los símbolos eléctricos y existe un archivo que se llama "sim" también, que posee algunos de los más comunes. Para bajar un archivo se emplea el comando "t" (ver la lista de comandos), pero antes de llamarlo es necesario asignar el vocabulario actual al de símbolos.

```
sim            ( invoca al vocabulario de símbolos)
t sim          ( trae el archivo sim)
```

El usuario puede ahora examinar el vocabulario con el comando "l" que producirá una lista de segmentos y un resumen de las entidades que los forman. Verá, por ejemplo, los segmentos "04" y "00" que corresponden a las compuertas TTL 7404 y 7400. Es factible ver los símbolos en la pantalla al imprimir el nombre del segmento tal como si fuera un comando. Si oprime "04", verá en la pantalla lo siguiente:

De manera similar se graficarán otros símbolos.





El sistema posee información que no muestra al dibujar los segmentos aunque sí lo hace al listar el vocabulario. Se trata de la estructura arborescente de los segmentos y de qué pata es entrada y cuál es salida.

El diseñador tiene la opción de crear más símbolos o editar su circuito. En caso de optar por agregar entidades al diccionario debe emplear el comando "s:". Supongamos, por ejemplo que se desea crear el símbolo correspondiente al circuito 7408. Esta es una compuerta AND de dos entradas. Se inserta una entrada al diccionario así:

```
s: 08
```

Este comando borra la pantalla y coloca el cursor al centro. Posteriormente puede añadirse el "cuerpo" de la compuerta creando recta por recta, pero si se examina el diccionario, se encontrará que ya existe el dibujo que estamos buscando: forma parte de la compuerta 7400 (NAND) y se llama "and-sp". Lo más sencillo es entonces agregar una instancia de dicho segmento:

ai and-sp

Posteriormente se agregan las patas moviendo el cursor a uno de sus extremos. En ese momento se da el comando "r" para indicar al sistema que en ese punto se va a iniciar una recta. Una vez colocado el cursor en el otro extremo, se agrega la pata con el comando "ap-" si es entrada y "ap+" si es salida. Para dibujar una recta que no tenga significado eléctrico se emplea el comando "ar". El texto, que en este caso es "08" se genera con el comando

at 08

que asignará la posición del cursor al centro de la cadena de texto. Se eligió el centro para que al dibujar instancias giradas, los letreros alteraran lo menos posible el resto del dibujo. El nuevo símbolo queda así:

Las modificaciones en los dibujos se realizan colocando el cursor sobre la entidad que se desea alterar y emitiendo el comando "loc". El sistema entonces efectúa una búsqueda y si encuentra una entidad sobre la que esté el cursor, la convierte en el "nodo actual" y la dibuja con líneas punteadas.

Ahora el usuario puede eliminar el nodo actual con el comando "e" o alterar alguno de sus parámetros dependiendo del tipo del nodo. Si el nodo actual es de tipo recta, se puede "amarrar" el cursor a cualquiera de sus extemos, produciendo el efecto de "banda de caucho". Si es de tipo texto, es factible moverlo de lugar y si se trata de una instancia hay comandos para trasladarlo (se amarra el cursor con el comando "<a"), girarlo ("gira") o cambiar el segmento del cual es instancia (comando "referencia"). En cualquier caso el cursor se desamarra con "a>".

Cuando el usuario haya generado los dibujos pertinentes, debe actualizar el archivo en disco con el comando

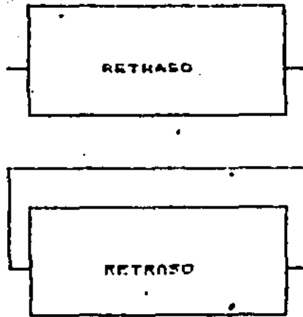
g sim (guarda el vocabulario actual en el archivo "sim")

En seguida se baja a la memoria el circuito en proceso de diseño.

cir (invoca al vocabulario "cir")
t cir (trae el archivo "cir" al vocabulario actual)

Normalmente el circuito tendrá una estructura jerárquica en la que el diseñador podrá editar partes del circuito en forma independiente para después juntarlas y así sucesivamente hasta formar todo el circuito. La edición de cada segmento se hace en forma muy similar a la creación de símbolos que se describió anteriormente, con las siguientes diferencias:

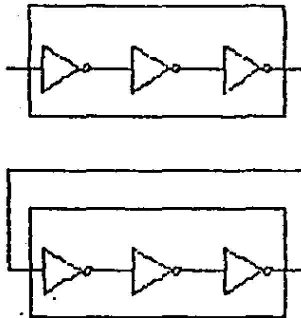
1) Hay un tipo de instancias que pueden ocultarse o no, dependiendo de una bandera. Estas instancias se denominan bloques y se agregan con el comando "ab". La bandera se denomina "blo". A continuación se muestra un circuito formado por dos bloques.



Al apagar la banderas "blo" y "tex" con los comandos

```
blo no
tex no
```

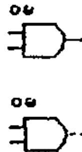
los bloques se hacen transparentes y además se inhibe la impresión de texto.



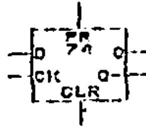
2) Las compuertas se agregan con el comando "ac" en lugar de "ai". Por ejemplo:

```
ac 08            ( agrega una instancia de la compuerta 08 )
                 ( mueve el cursor )
ac 08            ( agrega otra instancia )
```

y la pantalla se verá así:



Al agregar una instancia de la compuerta "74", la pantalla queda:



3) Las conexiones se inician con el comando "c" (en lugar del comando "r" que se emplea para rectas). Este comando provoca que el sistema dibuje con líneas punteadas todas las conexiones asociadas a la pata (o alambre) sobre la que está el cursor.

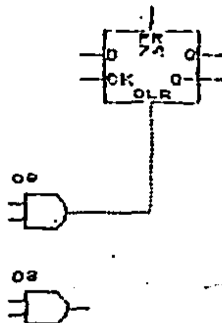
4) Las conexiones se terminan de una de tres maneras dependiendo del efecto gráfico que desee el usuario

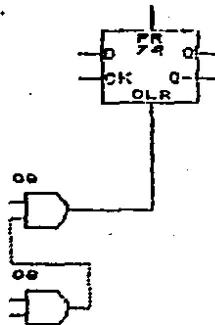
a) El comando "x" realiza una conexión lógica pero no gráfica. Este comando debe emplearse con cuidado porque no hay manera de concluir, a partir del dibujo, que dos puntos están conectados. Conviene acompañar este comando de alguna nota o símbolo que señale la conexión. Existen los macros "vcc" y "tierra", por ejemplo, que llaman a "c" y a "x", pero tienen el efecto lateral de agregar la instancia de símbolos determinados.

b) El comando "o" conecta dos puntos eléctricos dejando que el sistema calcule la trayectoria entre ellos

c) Los comandos "ts" (<ctrl> s) y "to" (<ctrl> o) se utilizan para conexiones guiadas. Cada vez que se desea una nueva recta se oprime "ts" y al llegar al punto final se emplea "to"

A continuación se muestra un par de conexiones generadas con el comando "o".

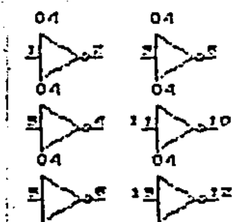


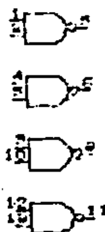


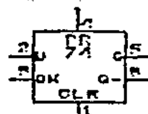
Una vez terminado el circuito, se procede a agrupar compuertas en chips y a numerar los chips. Para ello se requiere información de catálogo que debe traerse del disco de la misma que los archivos anteriores:

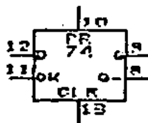
```
cat            ( invoca el vocabulario "cat")
t cat          ( baja el archivo "cat")
```

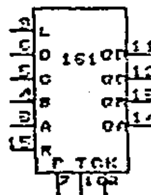
Este vocabulario contiene información de cuántas compuertas caben en cada chip y cómo deben numerarse. Los datos se almacenan gráficamente y consisten en varios segmentos independientes entre sí (que hacen referencia a símbolos del vocabulario "sim") cada uno de los cuales representa un chip. Por ejemplo, el chip 7404 contiene seis compuertas inversoras, el 7400 posee cuatro NANDS, el 7474 dos flip-flops y el 74161 un contador.



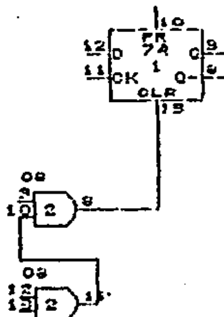








El comando "numera" agrupa y numera las compuertas del circuito que se encuentre en edición. Para ello es necesario invicar el vocabulario "cir" y dibujar el segmento principal del circuito.



El producto final del sistema es la lista de conexiones que se obtiene con el comando "conexiones" en la pantalla o "conexiones-archivo" en el disco. Para el ejemplo anterior la lista es:

```
>   2 .8  1 .13
>   2 .11  2 .10
```

Cada vez que aparece el carácter ">" indica un nuevo punto eléctrico, que está formado por una lista de parejas <chip,pata>. El primer punto eléctrico, por ejemplo, consta de la pata 8 del chip 2 y de la pata 13 del chip 1.