UNIVERSIDAD DE CANTABRIA
DEPARTAMENTO DE ELECTRONICA

PSAL : ESTUDIO, ANALISIS E IMPLEMENTACION DE ALGORITMOS DE SINTESIS DE ALTO NIVEL

MEMORIA
Presentada para optar al Grado de
DOCTOR EN CIENCIAS FISICAS
por el Licenciado en Ciencias,

Pablo Pedro Sánchez Espeso

EL DIRECTOR

D. Eugenio Villar Bonet
Profesor Titular de Electrónica
Departamento de Electrónica
Universidad de Cantabria

Santander, Marzo de 1991

Mi más sincero agradecimiento a D. Eugenio Villar Bonet, sin cuya ayuda, dirección y apoyo no hubiera sido posible la realización de este trabajo.

A mi familia y amigos, ya que sin su enorme paciencia y comprensión esta Tesis no habría visto la luz.

A mis compañeros del Grupo de Microelectrónica de la Universidad de Cantabria por su total apoyo y ayuda.

INDICE

CAPITULO 3: PSAL1: Sistema de síntesis algorítmica.

CAPITULO 4: PSAL2: Sistema de síntesis de alto nivel.

CAPITULO 5: Ejemplos de síntesis

INTRODUCCION

En los ultimos años se ha producido un gran avance en el desarrollo de herramientas de diseño asistido por computador (CAD). Sin embargo, este progreso ha incidido principalmente en la automatización del diseño a nivel lógico, eléctrico y layout [Ba86] [Go87][ML88][Ro88][Co89], mientras que las etapas iniciales del diseño (especificación del algoritmo hardware y determinación de la arquitectura de transferencias de registros que lo implemente), siguen dependiendo principalmente de la habilidad del diseñador. Por ello, el desarrollo de herramientas de síntesis de alto nivel ha cobrado un creciente interes en los últimos años [Ro88] [BoDe88][IDT89a][Ts89][DA89][RuTh89][Mi90][Al90][MFPa90] [ML90]. Una muestra de este interes viene dada por el hecho de que en la EDAC, celebrada en febrero de 1991 en Amsterdam, cuatro sesiones estuvieron dedicadas a la síntesis de alto nivel.

Las principales ventajas que se obtienen con los sistemas de síntesis de alto nivel, pueden ser resumidas en los siguientes puntos [MFPa88][Ca90]:

- Una reducción del tiempo de diseño lo cual permite aumentar el tiempo de vida del nuevo producto en el mercado. Uno de los problemas a los que se enfrenta la industria es la constante reducción del tiempo de vida de un producto en el mercado. Las herramientas de síntesis de alto nivel, permiten afrontar esta dinámica, gracias a una reducción sustancial del tiempo de diseño.

- Una reducción del número de errores producido durante el proceso de diseño, gracias a la formalización y automatización de este.

- Una rápida evaluación de multiples alternativas de diseño.

- Una mejor documentación del sistema.

- El uso de estas herramientas permite el acceso a la tecnología de circuitos integrados a personas no expertas en este campo.

El objetivo de la síntesis de alto nivel es obtener la estructura del sistema digital que mejor implementa el comportamiento deseado satisfaciendo el conjunto de restricciones impuesto por el diseñador. De una especificación de entrada a nivel algorítmico, el sistema debe producir una descripción a nivel de transferencia de registros.

En la síntesis de alto nivel, se distinguen habitualmente cuatro tareas [MFPa88]:

1º) Compilación.
2º) Optimización de la representación interna.
3º) Localización de las operaciones en pasos de control.
4º) Síntesis del "data path".

El comportamiento de un sistema digital es representado mediante lenguajes procesales (HLL, High Level Languages), similares a los utilizados en programación. Lenguajes como VHDL [LiMa86][Sh85][NuMe89][Co88], ISPS [SiBe82][Ba81], V [Ben89], C-Hardware [DMKu88] o Pascal [SeSo88] han sido utilizados con esta finalidad. Esta gran diversidad de lenguajes de entrada ha sido uno de los obstaculos en el desarrollo de la síntesis de alto, dado que dificulta no solo la introducción de estos sistemas en entornos ya existentes, sino tambien la comparación entre resultados de distintas técnicas asi como la definición de benchmarks.

El primer paso de todo sistema de síntesis de alto nivel (compilación), consiste en transformar la descripción del comportamiento del sistema a diseñar, realizada mediante alguno de los lenguajes antes reseñados, en una representación intermedia (del tipo árbol o grafo) sobre la que se aplicarán los algoritmos de síntesis. La representación intermedia contiene información sobre el flujo de control (precedencia en la ejecución de las operaciones) y de datos (flujo de información entre operaciones) y esta especialmente preparada para poder realizar sobre ella todo el proceso de síntesis de alto nivel. Entre los multiples representaciones intermedias propuestas destacan VT ("Value Trace")[SnSi78], YIF ("Yorktown Intermediate Format") [Ca89], CDFG ("Control-Data Flow Graph") [BoDe88] y el formato, especialmente adaptado a VHDL, propuesto por Camposano [CaTa89].

Una vez la representación intermedia ha sido generada, se aplican diversas transformaciones con objeto de obtener una descripción optimizada equivalente a la inicial. Para ello, pueden emplearse optimizaciones típicas de compiladores de lenguajes de programación tales como [AhSe86]:

- Propagación de constantes.

- Eliminación de código "muerto", es decir, de sentencias que nunca se ejecutan o cuyo resultado nunca se emplea.

- Eliminación de subexpresiones comunes. Esta optimización sustituye operaciones por simples asignaciones, lo que reduce la complejidad de la descripción.

- Movimiento de código ("code motion"). Esta modificación de la descripción permite, por ejemplo, cambiar las operaciones que se realizan en una construcción de control, dando lugar a una descripción equivalente a la inicial, pero en donde pueden ser aplicadas otras optimizaciones [WaTh89].

- Sustitución "in-line" de subrutinas. Esta transformación optimiza el flujo de control, eliminando las llamadas a las subrutinas. Para ello, reemplaza la llamada a la subrutina por el cuerpo de esta.

- Desenrollado de lazos, lo que permite una mayor optimización de la descripción [BaSh79].

Existen, por otro lado, una serie de optimizaciones específicas de los sístemas de síntesis de alto nivel [PaKu80][Po88][MoFo86][GeMo86][WaTh89], como por ejemplo:

- Modificación de operaciones, como por ejemplo, la sustitución por desplazamientos de bits de multiplicaciones por potencias de dos.
- Incremento del paralelismo [Po88][MoFo86].
- Reducción del número de niveles en el grafo de flujo de control o de datos.
- Creación de procesos concurrentes [PaKu80].

Una vez la representación intermedia ha sido creada y optimizada se inicia la ejecución de las tareas de síntesis [Ca90]: localización de operaciones en estados y síntesis del "data-path".

La descripción a nivel de transferencia de registros aporta información explícita, tanto de la estructura del circuito (registros, ALUs, buses, etc.), como de la secuencia de estados y las

acciones que deben realizarse en cada uno. En la descripcion algorítmica, por el contrario, el orden de ejecución de las operaciones esta limitado unicamente por las dependencias entre los datos (una variable no puede ser usada antes de haber sido definida). Esta descripción no aporta información explicita de la estructura del circuito. En la primera tarea, asignación de estados ("scheduling"), se asocia a cada operación el estado en el cual va a ser ejecutada. En la segunda o localización del "data path", se determinan los elementos hardware que forman el sistema y su interconexión.

Estas dos tareas no son independientes: una asignación de estados dada, determina los elementos hardware que el sistema precisa y, por otra parte, la disponibilidad de ciertos recursos determina la asignación de estados. Los sistemas de síntesis algorítmica abordan el problema de dos modos diferentes [MFPa88]:

1º) Sistemas que limitan inicialmente los recursos hardware, para después realizar la localización de operaciones en pasos de control, y por último la localización de los recursos hardware. Ejemplos de este tipo de sistemas lo constituyen las herramientas de diseño automático Facet[TsSi86] y CATREE [GeEl87](que serán comentadas mas tarde).

2º) Sistemas que realizan, al mismo tiempo, la localización de operaciones en pasos de control y la asignación de recursos hardware. Un ejemplo lo constituye el sistema YSC [BrCa88].

El método mas simple de asignación de operaciones en estados, consiste en delegar en el usuario esta tarea. Esta es la aproximación utilizada en el sistema Silc [Ba85], en el cual se supone que el usuario debe definir explicitamente el paralelismo del diseño. El resto de las técnicas aplicadas suelen ser clasificadas en 4 grupos:

1º) Localización del "data-path" y asignación de estados independientes.
La primera y más simple técnica de asignación de estados se conoce con el nombre de asignación ASAP ("as soon as possible") en la que las operaciones se asignan en el primer estado en que ello es posible. Esta técnica ha sido utilizada en los sistemas Emerald/Facet [TsSi86][Th86][TsSi84](desarrollado en la Universidad de Carnegie Mellon, CMU) y CATREE [GeEl87](desarrollado en la Universidad de Waterloo).

Una mejora de este algoritmo, combina la asignación ASAP, con una asignación posterior condicional. En el sistema MIMOLA [Zi80][BaMa89][BhLe90], la "asignación posterior" (es decir, la asignación de una operación a un estado posterior al que le corresponde en la asignación ASAP) es utilizada cuando el número de

operaciones que se ejecutan en paralelo en un estado es mayor que el número de unidades operacionales de las que dispone el sistema para realizar la síntesis. Esta aproximación es utilizada igualmente en el sistema Flamel [Tr87] (desarrollado en Stanford y AT&T).

Otra técnica muy utilizada, es la asignación ALAP (" as late as possible") en la que las operaciones se asignan en el último estado en que ello es posible.

Algoritmos mas complejos hacen uso de la técnica conocida como "list scheduling" [HaEl89]. Sistemas como EMUCS [Th86](desarrollado en CMU por Hitchcock y Thomas), SLICER [PaGa87a] (desarrollado en la Universidad de Illinois por Pangrle y Gajski) y CATHEDRAL-II [DM86][DM88][DM89][DM90](desarrollado en el IMEC por H. De Man y sus colaboradores) constituyen ejemplos de sistemas que utilizan esta técnica. En "list scheduling", las operaciones son ordenadas en orden topológico (de mayor a menor) usando las precedencias dictadas por las dependencias de datos y control contenidas en el CDFG ("Control-Data Flow Graph"). A continuación, las operaciones son asignadas a estados de forma iterativa, respetando el orden que tenian en la lista. Cuando se produce un conflicto, es decir cuando no se dispone de recursos hardware para poder realizar una operación en un determinado estado, una o varias operaciones son reasignadas en el estado siguiente. La selección de dichas operaciones es realizada mediante una función de prioridad local, aplicada a todas las operaciones que pueden ser asignadas en el estado que estamos considerando.

En el sistema SLICER, dicha función de prioridad esta basada en la "movilidad" de las operaciones. Se define la movilidad de una operación como el número de estados existente entre aquél en el cual dicha operación es asignada con la técnica ASAP y el que le corresponde al aplicar la asignación ALAP. El sistema CATHEDRAL-II utiliza una función de prioridad similar. Dicha función incluye un método heurístico para asignar lazos en un CDFG. El concepto de movilidad es igualmente empleado en el sistema FIDIAS [SoSe89] (desarrollado en la Universidad Complutense de Madrid por F. Tirado y sus colaboradores).

2º) Interdependencia entre asignación de estados y localización del "data-path"
En los dos siguientes sistemas la asignación de operaciones se realiza de forma simultanea con la localización de unidades operacionales. El sistema Elf [Gi87] utiliza una variación del algoritmo "list scheduling" (anteriormente presentado), que permite utilizar restricciones temporales. Su función de prioridad esta basada en la urgencia y el peso de la operación. El peso de una operación es el número de estados necesarios para ejecutar la instrucción, más la suma de los pesos de todas las operaciones sucesoras (

operaciones que utilizan su resultado), a lo largo del camino "más pesado". La urgencia de una operación es la relación existente entre su peso y el número de pasos de control existentes después de su última ejecución. Cuando una operación es retrasada un estado, su urgencia aumenta, lo cual incrementa la prioridad de localización de dicha operación en el siguiente paso.

El sistema MAHA [Pa86][PaPa88b][Pa89a][HaPa89] (desarrollado por Park y Parker en la Universidad de Southern California) utiliza la determinación del camino crítico y el concepto de libertad de una operación ("freedom") para guiar el proceso de asignación de estados. La libertad de una operación es lo mismo que la movilidad calculada en el sistema SLICER; en este caso, sin embargo, la técnica "list scheduling" no es usada.

El sistema MAHA determina el camino crítico y lo divide en n pasos, uno por cada ciclo de reloj (utiliza para ello el programa CSSP o " Clocking Scheme Synthesis Package" [PaPa88b]). A continuación localiza unidades funcionales para las operaciones del camino crítico, utilizando la técnica "first-come first-served". El concepto de "libertad" es utilizado para localizar las operaciones que no pertenecen al camino crítico. Los nudos con menor "libertad" son localizados en primer lugar.

3º) Asignacion de estados y localización del "data-path" simultanea[BePa90][PaKo90] [SaZa90][ClTh90].
El sistema DAA [KoTh85a][KoTh84][KoGe85][Ko85](desarrollado en CMU y AT&T por Kowalski) utililiza una aproximación con refinamiento gradual para asignar las operaciones a estados. En primer lugar, las operaciones son divididas en grupos, utilizando una métrica que toma en consideración la reutilización de unidades funcionales, las interconexiones y el paralelismo. A continuación, se localizan las operaciones de un grupo y se realiza la asignación de estados. DAA utiliza un algoritmo basado en la técnica "list scheduling" con una función de prioridad similar a la utilizada en el sistema SLICER.

El Yorktown Silicon Compiler, YSC [BrCa88][CaRo89][CaBe89], desarrollado por Camposano en IBM, divide la asignación de estados en dos pasos. En el primero, se realiza una asignación en el menor número posible de estados. Estados simples son creados para lazos, llamadas a módulos y para impedir conflictos en el uso de memorias o registros. A partir de estos datos, se produce un diseño inicial utilizando herramientas de síntesis lógica. Dichas herramientas proporcionan medidas de la velocidad y del area del sistema. El segundo paso consiste en dividir un estado en dos, lo cual reduce el ciclo de reloj (aumentando la velocidad del circuito) a la vez que incrementa la reutilización de las unidades funcionales (reduciendo con ello el area).

El sistema HAL[PaKn89a][PaKn89b][Pa89b](desarrollado en la Universidad de Carleton, Canada, por Paulin y Knight) realiza una localización del "data-path" preliminar y utiliza la información resultante para realizar la asignación de estados. A continuación, la localización es repetida con la asignación realizada con anterioridad, lo que permite realizar un análisis detallado de la velocidad y el coste del sistema, asi como preseleccionar las unidades operacionales. El algoritmo de asignación de estados es nuevamente ejecutado, produciendo la asignación final asi como un uso optimizado de las unidades operacionales preseleccionadas.

Este sistema utiliza en la asignación de estados un algoritmo "dirigido por fuerza" ("force-directed")[Pa89][PaKn89b], que tiene en cuenta datos tales como el area y velocidad de las unidades operacionales asi como las interconexiones y registros utilizados.

4º) Asignación de estados con "pipeline".
La asignación optimizada de descripciones del comportamiento con "pipeline" es un tema de desarrollo reciente en síntesis de alto nivel[PaPa88a][MaDe90]. Dos tipos simples de "pipeline" han sido estudiados [PaPa88a]:

- "Pipelining" estructural.
- "Pipelining" funcional.

En el primer tipo, la descripción del sistema es dividida en secuencias de operaciones que serán realizadas de forma concurrente. Secuencias sucesivas seran desplazadas en el pipe, de tal forma que diferentes instrucciones del algoritmo serán ejecutadas de forma solapada sobre el mismo "data-path".

El sistema Sehwa [PaPa88a] desarrollado por Park y Parker es una de las primeros metodologias que es capaz de implementar sistemas con este tipo de pipelining. Esta metodologia utiliza dos algoritmos de asignación de pipeline:

- Asignación posible: realiza una asignación con restricciones en el coste total de la implementación.
- Asignación maxima: asignación que proporciona la implementación más rápida, sin restricciones de coste.

Sehwa tambien incorpora un algoritmo exhaustivo para obtener la asignación optima. En él, el tiempo de busqueda es reducido utilizando las técnicas antes comentadas para determinar los limites del proceso. Esta programa realiza asignaciones sucesivas, y

después de cada una de ellas, calcula una estimación de la velocidad y el coste de la implementación, que permitirá guiar la proxima asignación.

Este tipo de pipeline ha sido tambien utilizado en la versión actual de Elf [Gi87]. La técnica de desenrollado de lazos ha sido utilizada para obtener una implementación del sistema con pipeline funcional. Este tipo de algoritmos separa cada iteración del lazo y pasa a ejecutarlas en paralelo, dando lugar a una implementación más rápida.

Otro método utilizado para mejorar la velocidad del sistema, consiste en utilizar unidades operacionales con pipeline. A este tipo de pipeline se le conoce con el nombre de pipeline estructural. En él, varias operaciones son ejecutadas en una misma unidad operacional de forma solapada (por ejemplo, dos productos pueden ser realizados en un multiplicador con dos fases de pipeline de forma solapada).

El sistema SLICER, anteriormente mencionado, fue uno de los primeros en asignar unidades operacionales con este tipo de pipeline. Al igual que en la asignación normal, el sistema utiliza el concepto de movilidad para guiar el proceso de asignación con pipeline.
El pipeline funcional y estructural son soportados por el sistema HAL, mediante una modificación del algoritmo dirigido por fuerza, anteriormente mencionado. El sistema utiliza la técnica de desenrollado de lazos para poder obtener un pipeline funcional, introduciendo ciertas consideraciones en el caso de sentencias condicionales. El pipeline estructural es soportado de una forma sencilla: modificando la definición del grafo de distribución.

Otra tarea básica en la síntesis de alto nivel es la localización del "data path"[MFPa88]. En una descripción a nivel algorítmico, el sistema digital es descrito en terminos de variables, operaciones y transferencias de datos entre unas y otras. En una descripción a nivel estructural, sin embargo, existen módulos hardware encargados de almacenar los datos, relizar las operaciones o comunicar los distintos módulos. La síntesis del "data path", tiene como misión determinar en que elementos hardware se localizan cada uno de los elementos de la descripción de alto nivel. Por esta razón, tres son las principales tareas que se diferencian en el proceso[MFPa88]:

1º) Localización de variables en registros.
2º) Asignación de operaciones a unidades operacionales.
3º) Localización de interconexiones.

Se han propuesto distintos algoritmos para realizar las tareas antes reseñadas. Estas

metodologias suelen ser clasificarlas en dos grandes grupos:

1º) Iterativos/constructivos.
Seleccionan la operación, variable o interconexión que será asignada, la asignan y a continuación repiten el proceso con otro elemento. Un ejemplo de programa que utiliza este algoritmo lo constituye el sistema EMUCS [Th86]. Para seleccionar la proxima decisión a tomar, el sistema emplea una función de coste de tipo "minimax"[MF83]. Técnicas similares han sido utilizadas en el sistema SUGAR [Th88][ThDr89] (desarrollado, al igual que EMUCS, en CMU). El sistema FIDIAS [Ti90] utiliza igualmente un método recursivo, mediante el cual se explora todo el espacio de diseño. Utilizando diversas técnicas, la busqueda es recortada, lo que permite obtener buenos resultados en un tiempo razonable.

2º) Globales.
Estos algoritmos utilizan técnicas matemáticas, como la "mixed-integer linear programing" [HaPa83] o técnicas de teoría de grafos [Ch84][Ch81][St90]. Tambien han sido empleados algoritmos basados en "Simulated Annealing" [NeSV86], modelando el problema de asignación como un problema de localización de operaciones en un espacio bidimensional. Esta técnica ha sido utilizada en el sistema desarrollado en la Universidad de Berkeley por Devadas y Newton [DeNe89].

El sistema Emerald/Facet [TsSi84] fue uno de los primeros en utilizar técnicas de teoria de grafos para realizar la síntesis del "data path"[TsSi86]. En estas metodologias, los elementos a localizar (variables, operaciones o transferencias), se modelan como nudos de un grafo, en el cual los arcos representan las dependencias entre dichos elementos. Dos nudos estan unidos por un arco cuando los elementos a ellos asociados, pueden ser localizados en el mismo módulo hardware. El problema de asignación es modelado entonces como un problema de división de un grafo en subgrafos fuertemente conexos. Dado que este es un problema NP-completo, diversas técnicas heuristicas han sido desarrolladas para realizar dicha partición en un tiempo razonable.

Otras metodologias utilizan técnicas matemáticas, como la programación lineal con variables 1-0 para realizar la localización [Ba88][PaKo90]. En ellas, las posibles decisiones se modelan como una variable que puede tomar dos valores: 0 (la decisión no se toma) o 1 (se realiza esa acción). Las restricciones que se imponen sobre el problema son modeladas como ecuaciones sobre dichas variables. Para resolverlas se utilizan técnicas de programación lineal. Estas técnicas han sido utilizadas en la localización de registros en memorias multipuerto o en ficheros de registros[Ba88]. Algoritmos similares, fueron utilizados por Hafer y Parker para localizar todos los elementos del "data path" [HaPa83]. Recientemente, Papachristou y Konuk han propuesto una técnica basada en programación

lineal, que realiza el "scheduling" y "allocation" simultaneamente [PaKo90]. Estas técnicas son costosas desde el punto de vista computacional, razón por la cual se estan introduciendo métodos heuristicos en su implementación.

Los sistemas de síntesis automática de alto nivel pueden clasificarse en dos grandes grupos, teniendo en cuenta su rango de aplicación. Por un lado, aquellos especializados en un tipo concreto de diseño, como por ejemplo SUGAR[Th88][ThDr89] (CPUs), CATHEDRAL-II [DM86][DM88][DM89][Go90][DM90][DeMA90] (procesadores digitales de señal), ADVIS[Mo87] (arrays systólicos) y APOLLON[JaJe85] (microprocesadores), entre otros. Por otro lado, sistemas con amplio espectro de aplicación como FACET , HAL, SPLICER y YSC.

Una alternativa de gran interes consiste en la utilización de técnicas de inteligencia artificial para realizar la síntesis de alto nivel. Las técnicas de busqueda heuristica empleadas en muchos de los sistemas antes reseñados estan empezando a ser sustituidas por programas basados en la utilización de sistemas expertos [IDT89b][Sc87][Su86][Ue85][Bi89] [PaMa87][KuKu89]. Uno de los primeros fue DAA. Este sistema ha alcanzado cierto renombre ya que fue el primero en generar un diseño (el de un IBM 370), que fue considerado como correcto por los ingenieros de IBM que lo diseñaron. Sistemas expertos han sido utilizados igualmente en programas de síntesis de alto nivel orientados al diseño de arquitecturas de procesado digital de señal (por ejempo el sistema DDL/SX desarrollado en la Universidad de Waseda, Tokio)[ShTa88].

Recientemente ha aparecido una nueva metodologia de síntesis de alto nivel en la cual los sistemas expertos juegan un papel importante. Estas técnicas se conocen con el nombre de Compilación de Silicio Inteligente [PaGa87b]. En ellas existen dos elementos o módulos que actuan de forma paralela: planificador y evaluador. El evaluador genera medidas de la calidad del diseño, tales como area, coste, velocidad, frecuencia de uso de los recursos hardware, etc. Estas medidas son utilizadas por el planificador para escoger un estilo de diseño (por ejemplo, el tipo de pipeline, o de unidades operacionales) y a continuación decide la estrategia de diseño se va a adoptar para obtener una implementación del sistema que después será catalogada por el evaluador. Un ejemplo de sistema que utiliza esta filosofía lo constituye la metodología S(P)LICER[Paga87b][PaGa87a], desarrollada por Pangrle y Gajski en la Universidad de Illinois. En este sistema el papel del evaluador es realizado por el diseñador. En el sistema Chippe [BrGa90], desarrollado por Gajski y Brewer, esta tarea es realizada por un sistema experto, el cual, en función de una serie de parámetros que evaluan la calidad del diseño, fija las restricciones con las cuales SLICER y SPLICER deben realizar una nueva síntesis del sistema.

Otro sistema que sigue una filosofia similar es BUD [MFKo90], desarrollado en el entorno CMUDA por McFarland y Kowalski. Este programa utiliza al sistema DAA como herramienta de síntesis del Data Path. BUD evalua el efecto de las decisiones de diseño por su efecto en el diseño físico, en vez de utilizar medidas abstractas que ignoren el layout, conexionado, clocking y otros factores importantes.

Por otro lado, se han propuesto metodologias que sacan el máximo partido a la tecnologia VLSI, mediante arquitecturas regulares, con conexiones muy localizadas. Estas técnicas se basan en un mapeado del flujo de datos en una arquitectura de array de procesadores o array sistólico [MoFo86][Ku88][LaMo85]. El sistema resultante de la síntesis se caracteriza por la ausencia de una unidad de control clásica (si existe circuiteria de control, ésta esta distribuida entre los elementos del array) y una máxima explotación del paralelismo de la descripción. Sistema como ADVIS [Mo87] y SYS3 [HaLe88], por ejemplo, permiten síntetizar una descripción de alto nivel en una array sistólico.

Un tema que no puede ignorarse en cualquier estudio sobre la síntesis de alto nivel, es la conexión de estos sistemas con herramientas de síntesis en otros niveles de abstracción. Algunos sistemas como DAA, HERCULES [KuDM90], ADAM o YSC son capaces de transformar la estructura resultante del proceso de síntesis en un circuito integrado, ya que estan integrados en entornos de diseño automático.

Un ejemplo de este tipo de entorno es el desarrollado en el proyecto ESPRIT I CVS [CVS86], en el cual participaron 10 centros de investigación europeos. En este proyecto se desarrolló el sistema BACH [Bo88a][Bo88b] como herramienta de síntesis estructural. Esta herramienta permite obtener el "layout" final del sistema a partir de la descripción de su arquitectura realizada mediante el lenguaje CVS_BK[CVS88]. Para poder integrarse en este entorno, el resultado de la síntesis de alto nivel debe de representarse en dicho lenguaje, desarrollado dentro del entorno KARL en la Universidad de Kaiserslautern (RFA).

Como se ha podido comprobar en la presente memoria la síntesis de alto nivel es un tema de interes creciente en donde, hasta ahora, no existen sistemas comerciales, solo prototipos universitarios, a pesar del enorme interes que estos temas despiertan en la industria. De hecho, algunos prototipos comienzan a ser integrados en entornos industriales, como el sistema CADDY [KrRo90] o SAW [SaDo90]. Estos sistemas constituyen un tipo de herramienta de diseño automático de desarrollo reciente, en el que aun quedan muchos aspectos por determinar y comprender totalmente. Se puede afirmar que la síntesis de alto nivel y la síntesis a nivel de sistema constituyen dos de los grandes retos en diseño automático de los años 90.

El objetivo de la tesis doctoral, que ahora se presenta, es el estudio, análisis e

implementación de algoritmos de síntesis de alto nivel que desde una descripción inicial del comportamiento del sistema generen la arquitectura a nivel de transferencias de registros que la implemente.

Para ello, se han realizado las siguientes tareas:

a) Estudio de los algoritmos mas importantes descritos en la bibliografia.

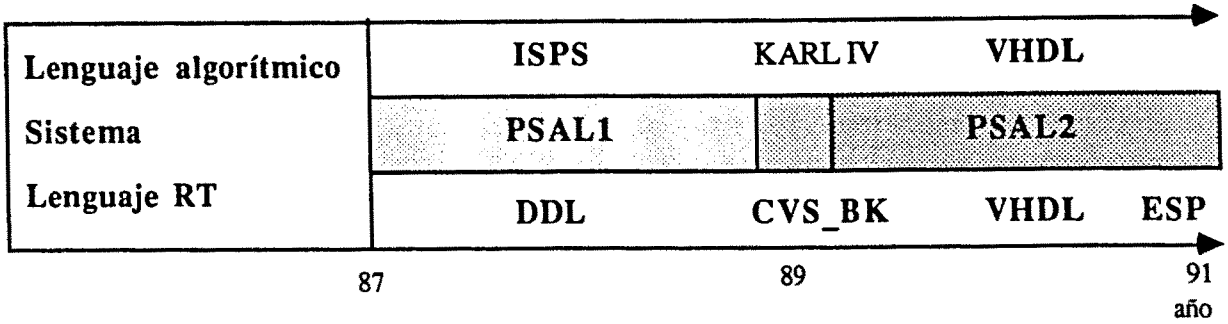
b) Elección de los lenguajes de entrada y salida con objeto de que la utilización del sistema sea lo más amplia posible. Las notaciones de entrada y salida deben tener un caracter general y estar conectadas a sistemas CAD completos.

c) Desarrollo de una representación intermedia que soporte la síntesis. Además, es preciso implementar compiladores que permitan, a partir de la descripción inicial en el lenguaje de descripción a nivel algorítmico, generar la estructura de datos intermedia elegida. Tambien es necesario desarrollar generadores de descripciones a nivel RT, en los lenguajes elegidos a tal efecto, a partir de la representación intermedia.

d) Desarrollo e implementación de algoritmos de localización de recursos hardware ("allocation") y de asignación de estados ("scheduling").

El trabajo recogido en la presente Tesis refleja los resultados obtenidos en el campo de la síntesis de alto nivel por el Grupo de Microelectrónica de la Universidad de Cantabria. Esta linea de investigación, que nace con este trabajo, esta sustentada en la experiencia que el grupo posee en el diseño de circuitos integrados y en la utilización de lenguajes de descripción de hardware en la simulación y test de dichos sistemas. Por esta razón, el trabajo se inició utilizando un lenguaje de descripción de hardware a nivel algorítmico utilizado anteriormente en el grupo: ISPS [BrMa88]. La arquitectura resultante del proceso de síntesis es descrita mediante el lenguaje DDL, el cual fue igualmente utilizado por el grupo con anterioridad con fines de simulación, test y síntesis de sistemas digitales descritos a nivel de transferencias de registros [AlBr86][AlBr87].Como consecuencia de estos trabajos, nace el sistema PSAL1 [SaVi91], el cual dio lugar a la Tesina de Licenciatura titulada "Analisis y Síntesis de Sistemas Digitales descritos a nivel algoritmico" [Sa89]. El sistema fue desarrollado con un formato intermedio muy adaptado al lenguaje ISPS y con unos algorítmos de síntesis en la linea de los desarrollados en CMU. Estas técnicas, además de utilizar ISPS como entrada poseian, cuando se inicio este trabajo, una gran actualidad y desarrollo. De PSAL1 se extrajeron importantes conclusiones, como por ejemplo, la necesidad de contar con un formato intermedio mucho mas flexible y desligado de ISPS, asi como de lenguajes de entrada y salida mas homogeneos. Además, durante el desarrollo de

PSAL1 y hasta la actualidad, la síntesis de alto nivel atraviesa una etapa de fuerte expansión, produciendose un elevado número de aportaciones y nuevas ideas, lo que obligaba a reconsiderar muchos de los algorítmos propuestos en PSAL1. Estos hechos produjeron la evolución en el sistema que aparece reflejada en la figura.



Como se puede observar la evolución no solo afecta al programa, sino tambien a los lenguajes de entrada y salida. Una de las consecuencias obtenidas de PSAL1 fue la necesidad de contar con lenguajes homogeneos, pertenecientes al mismo entorno. Se opto entonces, por utilizar lenguajes pertenecientes a la familia KARL, lo cual permitia además introducir PSAL1 dentro del conjunto de herramientas desarrolladas en el projecto CVS[CVS86], antes mencionado. Entre dichas herramientas se encuentra el sistema BACH [Bo88a][Bo88b][BoIt89], capaz de generar el layout del sistema a diseñar a partir de su descripción a nivel de transferencia de registros. Se escogio KARL IV como lenguaje de descripción a nivel algoritmico y CVS_BK [CVS88] como medio para representar el sistema a nivel de transferencia de registros. En la elección de estos lenguajes conto además, la experiencia del grupo en la utilización de otro de los miembros de esta familia: el lenguaje KARL III [SaAl89]. Con objeto de estudiar los lenguajes de este entorno y su conexión con el sistema PSAL, el autor de la presente Tesis se desplazo a la Universidad de Kaiserslautern (Alemania) invitado por el Profesor Hartenstein (autor de los lenguajes KARL III, KARL IV y CVS_BK), por un periodo de 3 meses. Durante esta estancia se llego a la conclusión de que KARL IV no se encontraba lo suficientemente desarrollado como para poder ser utilizarlo, debido en gran medida a la mayoritaria aceptación de un lenguaje estandar multinivel que había aparecido unos años antes: VHDL. Como consecuencia de ello, se decidió utilizar dicho lenguaje para describir tanto la entrada como la salida del sistema de síntesis.

De la experiencia de PSAL1 se extrajeron dos conclusiones pricipales. En primer lugar, la necesidad de un formato intermedio más flexible, independiente de los lenguajes de entrada y salida. En segundo lugar, la necesidad de incorporar los avances en las técnicas y algoritmos de "scheduling" y "allocation" que se estaban produciendo en ese momento. Surge asi PSAL2[SaVi90a][SaVi90b], caracterizado por una base de datos de gran

flexibilidad, la utilización de VHDL como lenguaje estandar multinivel y la incorporación de nuevos algoritmos de "allocation" y "scheduling". El sistema PSAL2 parte de una descripción en VHDL[SaRa90] y es capaz de expresar el resultado de la síntesis en los lenguajes CVS_BK, VHDL y ESP. Esta diversidad de lenguajes es consecuencia de la evolución histórica del sistema anteriormente comentada. La linea de investigación de síntesis de alto nivel no solo ha tenido como resultado la presente Tesis, sino que tambien ha permitido desarrollar un sistema de síntesis a nivel de transferencia de registros, conocido como ESP [RaSa91]. PSAL2 dispone de una conexión con dicho sistema, lo que permite disponer de un entorno que partiendo de una descripción a nivel algorítmico es capaz de generar el layout que lo implementa.

El sistema dispone de diversos algoritmos de síntesis, pudiendo el usuario seleccionar en cada caso el que desee. Se han implementado 3 algoritmos de "scheduling" ("forced directed", "List scheduling" y asignación global) y el mismo número de algoritmos de "allocation" (asignación global de registros, localización iterativo/constructiva de unidades operacionales y un algoritmo de síntesis del data path basado en probabilidad).

El trabajo recogido en la presente Tesis aporta nuevos algoritmos de síntesis como la técnica de asignación de operaciones en estados teniendo en cuenta las construcciones de control y el algoritmo de síntesis del data path basado en probabilidad, ademas de mejoras en los algoritmos tradicionales que serán mas tarde comentadas. Al mismo tiempo, se ha desarrollado un formato intermedio flexible capaz de soportar todos estos algoritmos. Por otro lado, el sistema utiliza un lenguaje estandar, VHDL, tanto para modelar el sistema a diseñar como su salida pudiendo, por lo tanto, ser utilizado en entornos de diseño que utilicen dicho lenguaje. PSAL2 no es un sistema cerrado: mas bien es un banco de pruebas en donde se han probabo y se probarán nuevos algoritmos y estrategias de síntesis de alto nivel.

La Tesis se encuentra dividida en 5 capítulos y esta introducción. En el primer capítulo se describen diversos algoritmos de síntesis de alto nivel que han sido analizados a la hora de diseñar los sistemas PSAL1 y PSAL2. En el segundo se hace una breve reseña a la utilización que de los lenguajes de descripción de hardware se hace en los sistemas desarrollados en la presente Tesis. El tercer capítulo contiene una descripción del sistema PSAL1, y el cuarto del sistema PSAL2. El quinto muestra los resultados obtenidos por dichos programas en la síntesis de varios ejemplos. Por último, en los apéndices A y B se muestra un ejemplo del formato intermedio de PSAL1 asi como los códigos de operación de dicho formato. En el apendice C se muestra el contenido de la base de datos de PSAL2 asi como los códigos de las sentencias de dicho formato.

CAPITULO 1

TECNICAS Y ALGORITMOS UTILIZADOS EN SINTESIS DE ALTO NIVEL

I) Introducción

En este capítulo serán expuestas las técnicas y algoritmos más importantes propuestos en sistemas de síntesis de alto nivel. En primer lugar se analiza extensamente el sistema de diseño automático de la Universidad de Carnegie Mellon. Esta metodología [SnSi78] [PaSi78][DiPa81][Th86][ClTh90] despertó nuestro interés por varias razones. En primer lugar, la Universidad de Carnegie Mellon fue uno de los primeros centros en donde empezaron a desarrollarse las metodologias de síntesis de alto nivel. Los primeros trabajos (EXP [Ba73]), datan de principio de los 70. En 1979, aparece por primera vez la representación VT [SnSi78]. Esta representación va a ser empleada por muchos de los programas desarrollados en dicha Universidad, algunos de los cuales serán comentados en el proximo apartado. El proyecto de investigación en el campo de la síntesis de alto nivel en CMU se ha mantenido hasta la actualidad, dando lugar a numerosos sistemas que reflejan distintas formas de afrontar la síntesis de alto nivel, todas ellas con un entorno común.

En segundo lugar, utiliza como lenguaje-fuente el mismo lenguaje que se emplea en el sistema PSAL1: ISPS [BaNo80]. Por ello, este sistema fue estudiado en profundidad, teniendo cierta influencia en la filosofia y algoritmos de PSAL1.

Posteriormente, serán expuestos algunos de los algoritmos y estrategias de síntesis de alto nivel mas importantes propuestos en los últimos años y que han sido considerados en la implementación de PSAL2: algoritmos de asignación de operaciones en estados dirigidos por fuerza, asignación de operaciones basadas en caminos, compilación inteligente, programación lineal y técnicas de "scheduling" y "allocation" simultaneos.

II) El sistema de diseño automático de la Universidad de Carnegie Mellon:

1) Introducción

El sistema CMU-DA está compuesto por un conjunto de herramientas software con un punto de partida común : el lenguaje de descripción de hardware ISPS y una representación de flujo de datos, denominada "value trace" (VT) [SnSi78].

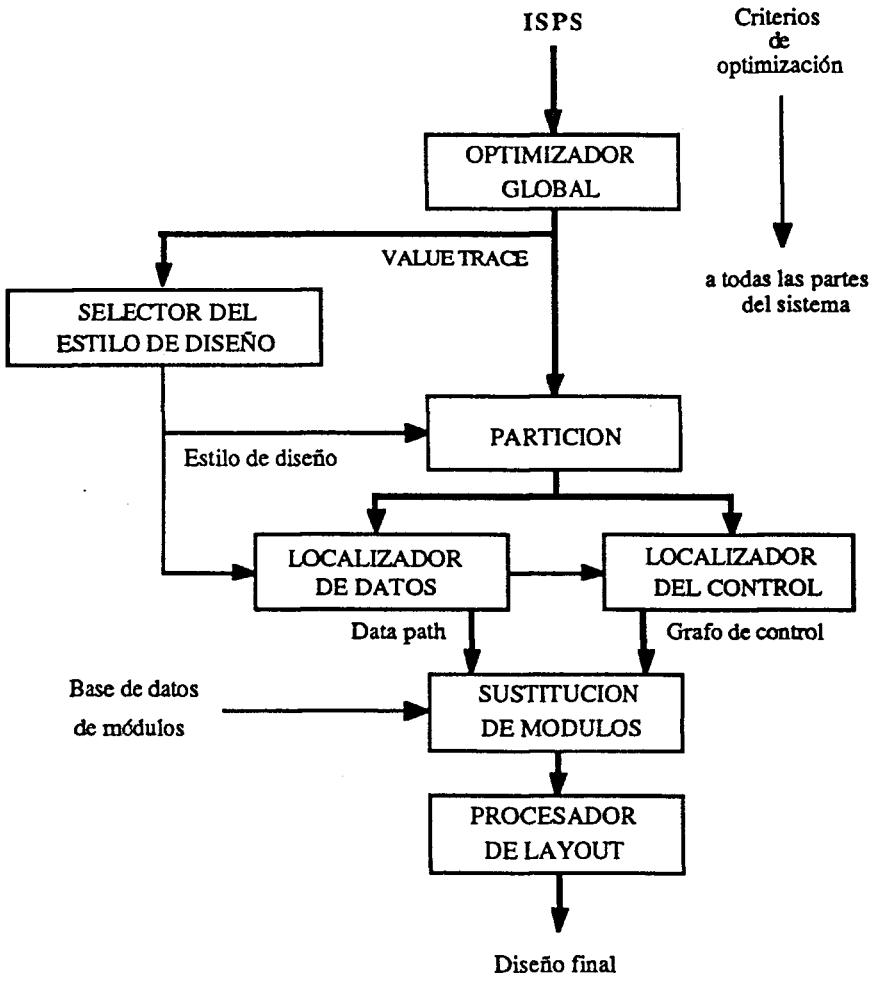


Figura 1.1

La estructura general de este sistema aparece en la figura 1.1 [DiPa81]. Algunas herramientas (como el sistema experto DAA, del cual hablaremos posteriormante) no se ajustan estrictamente a este esquema, pero en cualquier caso, la filosofía es parecida.

El proceso de diseño se encuentra dividido en una serie de niveles, de forma que el paso de uno a otro implica un aumento del nivel de detalle del sistema. Se diferencian 3 niveles:

A) Nivel algorítmico.
El proceso de diseño se inicia a partir de la descripcion algorítmica del circuito. En este nivel se realizan una serie de transformaciones que tienen por objeto producir una implementación hardware más eficiente. La filosofía que se utiliza en este paso es similar a la utilizada en el diseño de compiladores. Dado que la expresión concisa de un algorítmo, descrito en un lenguaje de alto nivel, no es generalmente óptima en código máquina, los compiladores implementan una serie de algoritmos que permiten aumentar la eficiencia del código generado. Estos algoritmos pueden ser usados igualmente, para optimizar la descripción formal del sistema digital que está siendo diseñado.

En este nivel, se encuadraría el Optimizador Global [SnSi78], que aparece como primer paso del proceso de diseño en la figura 1.1. Como vemos, este programa parte de la descripción que genera el compilador del lenguaje ISPS y produce una representación intermedia, en forma de grafo de flujo de datos, optimizado. Esta descripción intermedia, conocida como "value trace" (VT) sirve, no solo para realizar las transformaciones que se realizan a este nivel sino tambien como descripción de entrada de muchas de las herramientas del sistema (incluido el sistema experto DAA).

De las transformaciones implementadas en este módulo destacan:

a) Sustitución de las llamadas a los procedimientos por el cuerpo del procedimiento (sustitución "in-line"). Esta transformación ha sido estudiada y probada en el curso de esta investigación, proporcionando resultados interesantes.

b) Agrupación de sentencias condicionales de tipo IF o CASE.

En la próxima sección, se analizan detalladamente varias transformaciones que se realizan en este nivel.

B) Determinación de la estructura del circuito.
La descripción ISPS no posee información estructural. Cuando en este lenguaje se define un registro, no se hace referencia a un elemento físico, sino mas bien a un tipo de variable. Por ello, se hace necesario un paso de síntesis en el cual se obtenga, de la descripción del comportamiento del sistema,los módulos físicos necesarios para su implementación.

Como se muestra en la figura 1.1, dependiendo de las especificaciones del problema (ej:

coste y velocidad) y de la estructura del algoritmo, el Selector de estilo de diseño elige un tipo de implementación (estilo bus, distribuido, pipeline, etc). Una vez ha sido escogido el estilo, el programa indica que "localizador" debe ser utilizado.

Antes de mapear las variables del programa en registros físicos, tiene lugar uno de los pasos más importantes del proceso: la partición del sistema. En este paso se especifica que grupo de operaciones se realizan en cada paso de control, es decir, en este módulo se determina el flujo de control del programa.

Una vez terminado este proceso se realiza la localización, tanto de los datos como del control, como se observa en la figura 1.1.Veamos que papel juega cada uno de estos programas:

a) Módulo de asignación de variables (datos/memorias) en elementos de hardware.
En esta etapa se determina el número y tipo de operadores de datos, multiplexores (o buses) y registros necesarios para implementar la parte de datos del sistema.
La salida de este módulo es un grafo cuyos nudos son elementos tales como registros o sumadores. A este grafo se le conoce como gráfo de la unidad de datos ("data path graph").

b) Asignación de los estados del sistema.
En este módulo se genera una máquina de estados finitos (FSM) que controla el "data path" producido en el paso anterior. Este programa posee una opción que permite implementar la unidad de control del sistema con diferentes filosofías: microprogración, PLA, lógica random, etc. La salida de este módulo es un grafo de control, en el cual los nudos son estados.

C) Selección de módulos.
En este nivel se selecionan, de una base de datos, los módulos que implementarán físicamente el "data path" y el grafo de control producidos en el paso anterior. La base de datos contiene la descripción de los módulos, disponibles para el diseño del sistema, y puede ser actualizada con los avances de la tecnología. Estos módulos necesitan estar descritos en una gran variedad de niveles, desde el funcional (por ejemplo: un sumador) hasta el nivel de layout, pasando por el nivel lógico y el de circuito.

A continuación, el procesador de layout realiza una partición de los componentes en tarjetas (si se ha utilizado un diseño con chips estandard), decide la localización de componentes, realiza las interconexiones y prepara la documentación del sistema.

Algunas de las herramientas mas importantes del sistema CMU-DA son las siguientes:

*Optimizador Global [SnSi78]. Este programa, desarrollado por E.Snow, tiene 3 componentes conceptuales:

- Una representación abstracta del diseño.
- Un conjunto de transformaciones que permiten manipular dicha descripción.
- Una estrategia de exploración del espacio de diseño, necesaria para decidir en cada momento que transformación aplicar.

*Selector de estilo de diseño. Los estilos considerados en [SiTh77], con objeto de extraer datos estadísticos con los que evaluar la importancia de este módulo en el proceso de diseño son:

-Distribuido: Utiliza un conjunto de chips TTL, interconectados mediante conexiones simples y multiplexores. No se permiten buses.

- Buses: Este estilo se caracteriza por poseer un bus al cual están unidos los registros y operadores de datos. Problemas tales como la asignación de registros para obtener sistemas de altas prestaciones y la selección de tipo y tamaños de operadores de datos son típicos de este estilo de diseño.

- Microprocesador: Solo hay un operador de datos, el microprocesador. El problema consiste en obtener un programa que implemente el algoritmo en el microprocesador.

2ª)La representación "value trace " (VT).

Esta descripción aparece en el sistema CMUDA con objeto de soportar el análisis y las transformaciones de diseño realizadas a nivel algoritmico [SnSi78].

El VT es una representación en forma de flujo de datos del algoritmo. En un modelo tradicional de flujo de control secuencial (máquina con arquitectura Von Neumann), el control (indicación de qué operación debe realizarse) pasa de una sentencia a la siguiente. Esto significa que la descripción del sistema determina la secuencia de operaciones. En un modelo de flujo de control paralelo aparecen operadores especiales, que permiten activar más de una operación al mismo tiempo. Además, el paso de datos entre las diferentes operaciones del programa se realiza por medio de celdas de memoria, compartidas por varias operaciones. Este hecho, presenta el inconveniente de tener, a priori, que asociar cada dato a una posición de memoria. En un modelo de flujo de datos [HwBr84], no existe este obstáculo. La ejecución de una operación se produce, no por el paso del control desde

una operación anterior, sino por la disponibilidad de los datos, los cuales pasan de la operacion en la que se definen a las operaciones en las que se usuan. Esto significa que una operación se ejecutará cuando los datos que necesitan esten listos. Por ello, la secuencia del programa esta determinada únicamente por las relaciones de dependencia entre datos, lo cual permite explotar al máximo el paralelismo de la descripción.

Las ventajas de una representación como la VT son las siguientes:

Por un lado en vez de expresar operaciones realizadas sobre registros, como hacen los lenguajes de transferencia de registros, la VT representa operaciones sobre los valores en si mismos, sin hacer refencia a los elementos hardware en los cuales reside. El mapeado de valores en elementos físicos es realizado por el asignador de datos.

Por otro, esta representación permite eliminar la ordenación serial innecesaria. La ordenación esencial es establecida únicamente por la generación y uso de valores en la VT. La relación de precedencia indica que una operación que produce un valor, debe ser realizada antes que cualquier otra que use dicho valor.

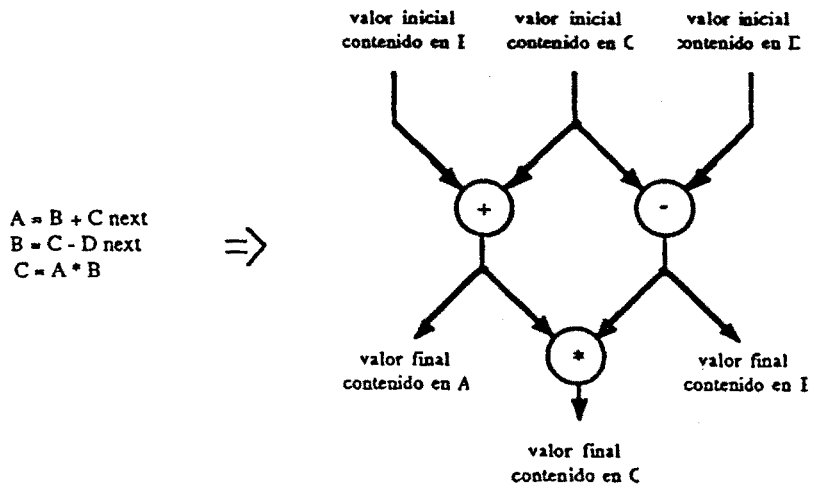


Figura 1.2

En la representación VT, las operaciones reciben el nombre de actividades. Cada actividad tiene un conjunto de valores como entradas y produce uno o mas valores como salida. En la figura 1.2 se muestra un ejemplo de correspondencia entre la descripción ISPS (figura

1.2.a) y su representación VT (figura 1.2.b).

La descripción ISPS consta de 3 operaciones que se realizan en serie. La ordenación serial es necesaria porque los valores contenidos en los registros B y C deben de ser usados antes de ser modificados. En la representación VT no aparecen restriciones respecto de la ordenación relativa de la suma y la resta (que aparecen en la descripción ISPS), dado que el valor generado por una de ellas no influye en el calculo de la otra. Precisamente, el hecho de fijarse únicamente en la dependencia de datos, hace que la VT sea idónea para explorar el paralelismo en una descripción. El problema de la asignación no-conflictiva de datos en registros es resuelto por otro programa.

Pero la representación contiene, no solo operaciones lógicas y arítmeticas, sino tambien acciones condicionales. Estas acciones utilizan una operación especial, llamada "selección". Esta operación es básicamente un conmutador que escoge un conjunto de valores de los datos (de los varios que toma como entradas) y lo transmite a la salida. Esta elección es realizada de acuerdo con el valor de otra entrada, llamada " selector".

Un ejemplo de acción condicional en ISPS y su correspondiente expresión en VT aparece en la figura 1.3 Los valores contenidos en los registros X e Y, despues de la acción condicional, dependen del valor contenido en la variable COND, cuando la condición es ejecutada. La selección escoge uno de los dos conjuntos posibles de valores de las señales X e Y, transmitiendolo a la salida.

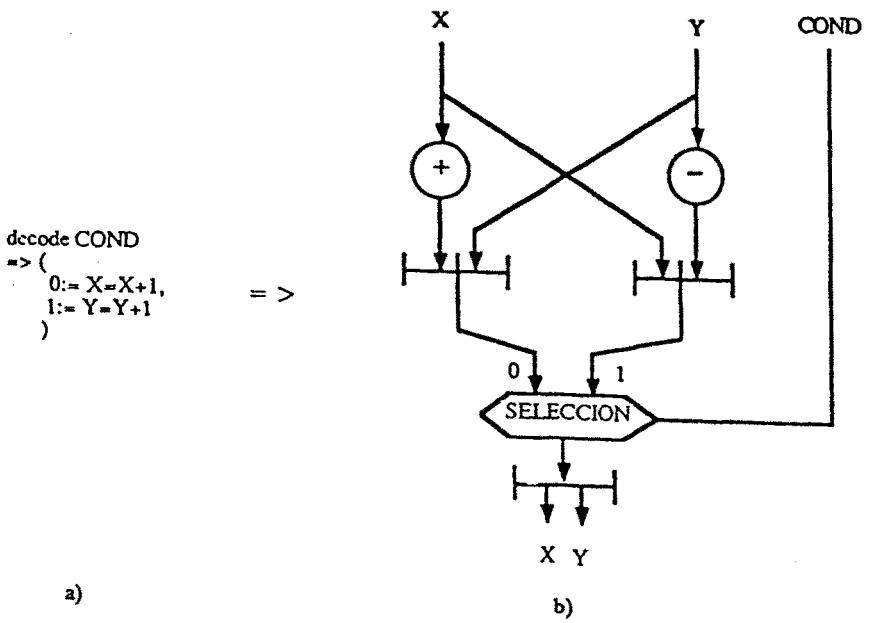


figura 1.3

Con objeto de permitir expresar subrutinas y acciones iterativas, nacen dos construcciones: el VT-cuerpo y la referencia a VT-cuerpos.

Un VT-cuerpo es una descripción VT, con un conjunto definido de variables de entrada y de salida. Gracias a ellos,se puede manejar toda una descripción, como un solo elemento.

Un instalador de un VT-cuerpo es un símbolo que muestra la localización que debe tener una copia del VT-cuerpo en una descripción. El VT-cuerpo sería el equivalente a la definición de una subrutina, en un lenguaje de alto nivel, mientras que el instalador del VT-cuerpo sería una llamada a dicha subrutina.

3) Algoritmos empleados por el sistema CMUDA

En esta sección vamos a hacer una breve descripción de los principales algoritmos utilizados por las dos primeras etapas del sistema de diseño de Carnegie-Mellon, en las que se centra nuestro interes.

A) Algoritmos empleados por el Optimizador Global [SnSi78].
Todas las transformaciones, en este nivel, operan sobre la representación VT. Estas transformaciones tienen por objeto optimizar el coste, velocidad y otras prestaciones del diseño. Las transformaciones, que se realizan a este nivel, se dividen en 4 categorias :

- Transformaciones que afectan a las actividades.
- Transformaciones en los VT-cuerpos.
- Transformaciones en las operaciones de selección.
- Estrategias para aplicar las transformaciones anteriores.

Veamos brevemente las más importantes:

a) Transformaciones que afectan a las actividades.
Las transformaciones más importantes que afectan a las actividades son:

a.1) Reducción de constantes.
Dado que el valor de las constantes es conocido en tiempo de compilación, podemos, en el caso de que todas las entradas a una actividad sean constantes, conocer el valor que tomará la variable de salida. Con ello, hemos transformado una operación aritmética en una asignación. Este cálculo de operaciones en tiempo de

simulación es típico de los compiladores.

Un ejemplo de aplicación de esta propiedad se muestra a continuación:

a = 0 next		a = 0 next
b = 1 next	se transforma en	b = 1 next
c = a+b next		c = 1 next

a.2) Introducción de la identidad.
La idea de esta transformación es introducir una actividad nula que no modifica el resultado final del VT pero facilita la extracción de actividades de un VT-cuerpo o la simplificación de la operación selección.

a.3) Eliminación de actividades muertas.
Como consecuencia de otras transformaciones, suelen aparecer en el VT, actividades cuyo resultado nunca es utilizado. Estas operaciones, no utilizadas, son eliminadas en esta transformación. Esta metodología es equivalente a la eliminación del código no empleado, que realizan los compiladores, como se ve en el ejemplo:

a = b + c next		
a = h + j next	se transforma en	a = h+j next

a.4) Eliminación de actividades redundantes.
Cuando dos o mas actividades realizan la misma operación y tienen idénticas entradas, todas menos una son redundantes, dado que todas asignan el mismo valor. Veamos un ejemplo:

1 Q <= A next		Q <= A next
2 R <= A+B next		R <= A+B next
3 A<= C next	se transforma en	A<= C next
4 S <= Q+B next		S<= R next
5 T <= A+B next		T <= A+B next

En este caso, las operaciones 2 y 4 son redundantes, ya que tienen las mismas entradas.

a.5) Introducción de actividades redundantes.
Es el paso contrario al anterior, y sólo se emplea para poder extraer operaciones de un VT-bloque o de una operación de selección.

b) Transformaciones en los VT-cuerpos

b.1) Eliminación de VT-cuerpos muertos.
Esta transformación es parecida a la a.3). El objetivo es eliminar aquellos VT-cuerpos que no son utilizados, debido a la secuencia del programa.

b.2) Eliminación de VT-cuerpos redundantes.
Cuando dos o más VT-cuerpos realizan las mismas operaciones y sus entradas son iguales, todos salvo uno de los VT-cuerpos son eliminados.

b.3) División de VT-cuerpos.
Esta transformación es la opuesta a la anterior, ya que crea VT-cuerpos redundantes. Esta transformación tiene por objeto ayudar en la extracción de VT-cuerpos de operadores selección.

b.4) Movimiento de actividades en VT-cuerpos.
Esta transformación permite mover actividades del interior del VT-cuerpo al exterior o viceversa y tiene gran importancia a la hora de simplificar estructuras de control tales como lazos. La extracción de operaciones, cuyo resultado es el mismo en todas las iteraciones, permite reducir el número de operaciones en el cuerpo del lazo. Así, si tenemos una operación de este tipo en un lazo que se ejecuta 1000 veces y la extraemos, nos habremos ahorrado realizarla 999 veces.

c) Tranformaciones de la operación "selección".
Básicamente lo que se pretende es simplificar esta operación, extrayendo sentencias comunes en todas las opciones o dividiéndola en otras "selecciones" mas pequeñas.

c.1) Factorización de la selección.
Divide la selección en un conjunto de selecciones menores. Tres son los objetivos buscados:

c.1.1) Una selección compleja puede ser dividida en operaciones más simples realizadas en diferentes pasos de control, lo que facilita su implementación.

c.1.2) Selecciones más simples prodrían ser implementadas como un circuito combinacional en vez de como uno secuencial. Con ello ganaríamos tiempo y en general el coste se reduciría.

c.1.3) El movimiento de actividades a través de la selección podría ser posible

después de una factorización.

c.2) Movimiento de actividades a través de las selecciones.
La idea es extraer una sentencia, que aparece en todas las opciones de la selección, colocandola tras la ejecución de la operación. Con ello se reducen las operaciones a realizar en las diferentes opciones, simplificando la complejidad del control. Además, al extraer actividades, una operacion secuencial (selección) puede transformarse en una operación combinacional. Más aún, podría permitirse la realización de transformacio-nes antes inaplicables.

d) Estrategias para aplicar las transformaciones anteriores. Existen diferentes metodologías que permiten transformar el algoritmo de forma que sea posible aplicar las transformaciones anteriores. Entre estas estrategias destacan:

d.1) Sustitución de las llamadas a las subrutinas por el cuerpo de éstas [Th86].
Las llamadas a las subrutinas impiden la aplicación de las transformaciones anteriores, entre actividades que se encuentran dentro y fuera de la subrutina. Hay casos en los cuales esta transformación puede tener efectos negativos, pero en la mayoría se obtienen ventajas. Esta transformación facilita sobre todo la aplicación de transformaciones entre actividades.

d.2) Sustitución de operaciones selección por sentencias condicionales.
Esta sustitución permite reducir la complejidad del control del sistema. En el análisis de esta transformación se encuentran casos en los cuales una subrutina entera es sustituida por una simple operación combinacional. La idea aparece en el siguiente ejemplo:

El algoritmo

```
a<>, c<>
...
b = 1 next
if not c =>
  if a =>
    b = 0
```

equivale a la operación

```
b = (not a) or c
```

d3) Transformación de lazos.

La idea es sustituir un lazo con n-iteraciones por otro con (n-1)-iteraciones, precedido por la ejecución de las operaciones que se realizaban en la primera iteración. En la figura 1.6 se observa como la aplicación de esta estrategia, permite transformar el lazo en una secuencia de operaciones. En figura 1.6.a se muestra una descripción ISPS. La aplicación de la transformación anteriormente reseñada tiene como consecuencia la extracción de una llamada a la entidad "acción" del lazo, y la reducción del número de iteraciones de este en una unidad. La aplicación sucesiva de esta estrategia produce como consecuencia la descripción de la figura 1.6.b, en la cual el lazo ha sido transformado en una secuencia de operaciones.

```
COUNT=0 next
   LOOP := begin
      acción() next                          acción() next
      COUNT=COUNT+1 next                     acción() next
      if COUNT < 3 => restart LOOP           acción() next
       end
```

a) b)

Figura 1.6

B) Algortimos empleados para la síntesis del "data path" [MF83] [HaPa78] [Th86].

A continuación se comentan los sistemas Emeral y DAA como representantes de las dos aproximaciones de síntesis mas importantes del sistema CMU-DA:

a) El sistema Emerald [ChTs86].

Esta metodología ve la descripción VT como una secuencia de código, en la cual se especifica tanto el flujo de control como el de datos. Dentro de esta secuencia de código se distinguen bloques básicos, es decir, secuencias de código que tienen un único punto de entrada y un único punto de salida. La metodología utiliza tres algoritmos:

- El primero asigna las variables a elementos de hardware.
- A continuación, otro algoritmo localiza los operadores de datos que aparecen en el circuito en ALUs.
- Por último, se determinan los multiplexores que deben aparecer en el sistema para unir los registros y las ALUs.

Las tres metodologías se basan en la misma idea: obtener una partición óptima de un grafo. Una característica de esta metodología es su globalidad, es decir, que afecta a todos los bloques básicos del sistema al mismo tiempo. Además, es muy simple, por lo que requiere poco tiempo de ejecución. En el fondo es un método heurístico para resolver un problema NP-completo: la partición óptima de un grafo. En cuanto a sus resultados, se indica en la bibliografía que es la que mejor localiza las variables. Veamos los diferentes procedimientos de esta metodología:

a.1) Compactación de variables.

Este algoritmo asigna a cada variable una localización física. Para ello toma la secuencia de operaciones y numera cada paso de control. Una vez hecho ésto, mediante un análisis de las variables, obtiene la tabla de vida, es decir, una matriz en la cual se indica para cada variable y paso de control si dicho dato está vivo o muerto. Un dato está "vivo" si va a ser empleado posteriormente. Está "muerto" en el caso contrario. A partir de esta tabla se obtiene el grafo de compatibilidades. En este grafo, los nudos son las variables del algoritmo y los arcos indican que dichas variables nunca están vivas en un mismo paso de control, salvo que en dicho paso, una de las variables sea operando y la otra resultado de la misma sentencia.
Los arcos se dividen en dos categorías:

- Arcos asociados a tranferencias puras de datos.
- Arcos asociados a operaciones.

Los arcos se caracterizan por un par de valores, que indican que nudos unen. Con ellos se construye una lista de arcos.
El obetivo de esta metodología es obtener una partición del grafo, de forma que las variables de una clase de la partición puedan ser localizadas en el mismo registro.
El algoritmo empieza a agrupar los nudos, considerando primero los arcos de la primera categoría. Para agruparlos calcula dos valores:

- Numero de vecinos (nudos unidos por un arco) de cada nudo, es decir, el número de arcos que contienen a dicho nudo.
- Para cada arco, computa el número de nudos que son "vecinos" únicamente de uno de los dos unidos por el arco (a esta cantidad se le denomina número de vecinos excluidos). Calcula igualmente el número de vecinos comunes.

Una vez hecho esto, busca entre los nudos de la categoría que estemos agrupando, aquel que tenga más vecinos. A continuación, toma el arco que más vecinos comunes tenga. Si hay varios, se toma el que menos vecinos excluya. Los nudos

que une dicho arco serán agrupados en la misma clase de la partición. Cada clase constituye un "nudo" del grafo, que sustituye a los nudos que la forman. Los nudos se identifican mediante un índice, el cual se utiliza para reducir el cálculo, ya que al ser los arcos no dirigidos, el arco (2,3) es igual que el (3,2), por lo que se establece una ordenación arbitraria de los nudos en los arcos {(i,j),i<j} que permite reducir los cálculos. Los nudos asociados a una clase tienen por indice el menor de los indices de los nudos que agrupan. Se dice entonces, que estos nudos tienen un "nudo cabecera" p. En el grafo, este nudo estará unido a todos los vecinos comunes de los nudos que forman la partición. En este nuevo grafo, se repite el proceso, tomando siempre un arco que contenga a p, hasta que dicho nudo desaparezca de la lista de arcos (sea un nudo aislado). En este caso, se busca otro nudo de la misma categoría y se repite el proceso, hasta que no queden nudos en esa categoría, caso en el cual se pasa a otra. El resultado será un conjunto disjunto de clases de nudos. Cada clase se asocia a un registro.

a.2) Asignación de los operadores de datos.

La idea es muy parecida a la del paso anterior. Partiendo de la secuencia de código, se numeran, no los pasos de control, sino las operaciones. Cada operación recibe un número. Acto seguido se construye un grafo en el cual cada nudo será una operación. Los arcos conectan operaciones que no se realizan en el mismo paso de control. Con ello, los nudos unen operaciones que podrían estar agrupadas en una misma ALU de forma muy similar al caso anterior. En aquel, se unian variables susceptibles de compartir la misma localización física. En este caso, se establecen unas categorías en función de la similitud de las operaciones, datos y resultados. El algoritmo es similar al empleado en el caso anterior, salvo que en este caso, al agrupar nudos, se produce una reclasificación de las categorías (pueden aparecer similitudes).

a.3) Determinación de unidades de interconexión.

Este algoritmo tiene por objetivo determinar qué multiplexores son necesarios en el circuito. Por ello lo que intenta es agrupar los registros de forma que los datos en ellos contenidos tengan siempre el mismo destino.

El primer paso del algoritmo consiste en sustituir cada operacion por tres transferencias puras:

- Dos transferencias en las cuales, se pasan los datos a los registros de entrada de la ALU correspondiente.
- Otra en la cual se tranfiere el valor contenido en el registro de salida de la ALU al registro destino.

Con ello, la secuencia de código aparece formada únicamente por transferencias puras de datos. A continuación se crea una tabla, en la cual aparecen en la primera columna, el registros de destino, y en la segunda el de origen. Se aplica (cuando es posible) la propiedad conmutativa para intentar que un registro acceda siempre al mismo registro de la ALU, reduciendo con ello la tabla. Los elementos de la tabla serán los nudos de un grafo cuyos arcos indican que las interconexiones que unen no se realizan en el mismo paso de control. Se dividen las conexiones en dos categorías, según tengan o no elementos (fuente o destino) comunes, y se aplica un algoritmo similar al analizado en el primer apartado. Las clases de la partición indican que interconexiones pueden agruparse en el mismo multiplexor.

b) El sistema experto para diseño VLSI "Design Automation Assistant" (DAA) [KoTh85][MFKo90]

En los últimos años se está observando un notable interés en la aplicación de sistemas expertos en el campo del diseño VLSI [Ue85][Su86][BrGa90].

DAA es un ejemplo de estos sistemas. Este programa fue implementado utilizando OPS5 (herramienta para construir sistemas expertos basados en el conocimiento). Los sistemas expertos se caracterizan por poseer una base de conocimientos y un motor de inferencias, en lugar de la estructura clásica del software tradicional formada por algoritmo y datos, tal y como se refleja a continuación [Fo84] :

Software tradicional:	Programa = Datos + Algoritmo
Sistema Experto:	Sistema = Conocimiento + Inferencia

Un sistema experto es la mejor solución para problemas en los cuales no se conoce un algoritmo que lo solucione (o si éste existe es demasiado complejo), pero sin embargo hay expertos humanos que lo resuelven. Este sistema trata de utilizar el mismo sistema que utilizan los humanos para resolver un problema: solucionando el nuevo problema de la misma forma que se soluciona un problema "parecido" resuelto satisfactoriamente con anterioridad. Se trata de usar los conocimientos de los expertos humanos, expresados de una forma conveniente, para resolver los problemas. Un medio comunmente utilizado en la representación del conocimiento son las reglas [Fo84].

En DAA estas reglas se agrupan por tareas, tal y como muestra la figura 1.7. El sistema divide el problema en cuatro partes.

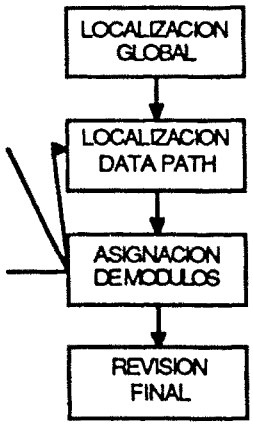


figura 1.7

b.1) Localización global. Localiza todos los elementos que aparecen en la descripción VT y que fueron declarados en la descripción.
b.2) Localización del flujo de datos. Este módulo determina las conexiones necesarias para implementar el flujo de datos, así como las operaciones que deben realizarse.
b.3) Asignación de módulos. En esta parte se van asignando los diferentes elementos que aparecen en la etapa anterior a elementos físicos del hardware.
b.4) Revisión final. En esta etapa se van eliminando todos los elementos sobrantes de la descripción.

El programa emplea adicionalmente algunas reglas para realizar funciones comunes a todos los módulos. El sistema está constituido por 314 reglas.

Este programa propuso un diseño del computador IBM S/370 estimado aceptable por sus diseñadores[KoTh84]. En la actualidad este sistema forma parte de una metodología mas general: el sistema BUD. Este programa evalua el efecto de las decisiones de diseño por su efecto en el diseño físico (en vez de utilizar medidas abstractas que ignoren el layout, conexionado, clocking,etc) y utiliza el sistema DAA para realizar la síntesis del data path.

III) Algoritmos de asignación de operaciones en estados dirigidos por fuerza.

Aparte de los algoritmos utilizados en el sistema CMU-DA que acabamos de describir, en los últimos años se han propuesto técnicas de "scheduling" muy efectivas algunas de las

cuales ya fueron comentadas en la introducción. Una de ellas es la técnica de asignación de estados "dirigida por fuerza" ("forced directed scheduling") desarrollada en la Universidad de Carleton, Canada, por Paulin y Knight, como parte del sistema HAL. Vamos a dedicarle una cierta atención ya que una versión mejorada ha sido implementada en PSAL2.

El objetivo del algoritmo es reducir el número de unidades funcionales, registros y buses del sistema, mediante un adecuado balance en la asignación de estados, sin aumentar el tiempo de ejecución. El procedimiento es del tipo iterativo-constructivo, es decir, localiza una operación en cada paso.

El algoritmo determina, en primer lugar, el camino crítico. Recibe este nombre el mayor conjunto de operaciones incompatibles (no localizables en el mismo estado) que es posible encontrar en la descripción. Dicho conjunto determina el tiempo de ejecución de la descripción. Si no se desea incrementar el número de estados del sistema, debemos asociar cada operación a un estado en el cual ya haya sido localizada una operación del camino crítico. Inicialmente a cada una de esas operaciones se las asigna a un estado.

El siguiente paso es determinar los marcos temporales de cada operación. Recibe esta denominación el rango de estados en el cual una operación puede ser localizada. Para determinar dicho marco se realiza una asignación de tipo ASAP ("as soon as possible") seguida de una asignación ALAP ("as late as possible"). La diferente asignación que cada operación recibe al utilizar dichos algoritmos determina el marco temporal de esta (el estado mínimo en el cual es localizable es proporcionado por la asignación ASAP y el máximo por la ALAP). En la figura 1.9, pueden observarse los marcos temporales iniciales, ocupados por las operaciones del algoritmo descrito como ejemplo en la figura 1.8.

```
while z < a loop
        zl:= z+dz;
        ul:= u- (3 * z * u * dz ) - ( 3 * y * dz );
        yl:= y + ( u * dz );
        z := zl; u:=ul; y:=yl;
end loop;
```

figura 1.8

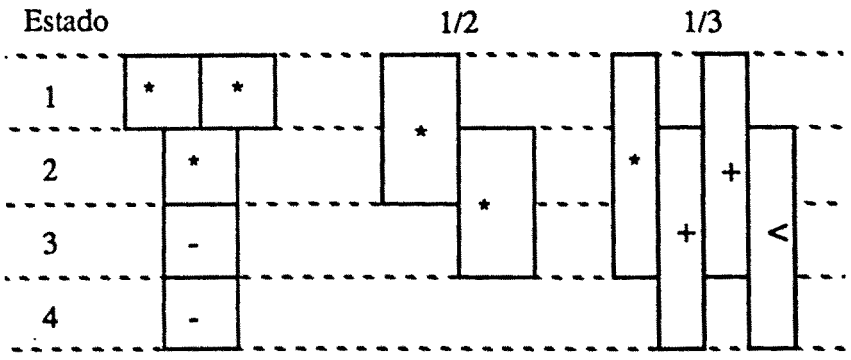


Figura 1.9

En la primera columna (camino crítico) se encuentran las operaciones que solo pueden ser localizadas en un estado. En la segunda, aquellas que pueden ser localizadas en 2 estados, y en las últimas las que podrían ser asignadas en 3 estados.

El siguiente paso es determinar el grafo de distribución, o medida de la posiblidad de que un determinado tipo de operación sea localizado en un estado. Dicho grafo se irá modificando a medida que las operaciones sea asignadas. El grafo de distribución se define como:

$$DG(i) = \sum_{\substack{\text{Operaciones} \\ \text{de tipo Opn}}} Prob(\ Opn,i)$$

Siendo "Prob(Opn,i)" la probabilidad de que una operación de tipo Opn sea localizada en el estado i. Si suponemos que inicialmente es indiferente en donde se localiza una operación, tendremos que, por ejemplo para el estado 1, el grafo de distribución para la operación de tipo producto será:

$$DG(1)_{producto} = 1 + 1 + 1/2 + 1/3 = 2.83$$

El aspecto más característico del algoritmo (y del cual toma su nombre) es que para decidir que operación localiza en un estado, utiliza el concepto de fuerza. Su definición es similar a la conocida Ley de Hooke:

$$F = K * x$$

En donde "K" es el Módulo de Young y "x" el alargamiento del material. En este algoritmo la expresion utilizada es:

$$Fuerza(i) = DG(i) * x(i)$$

Siendo:

- Fuerza(i) = dificultad de la asignación de una operción al paso i.
- DG(i) = valor del grafo de distribución para el tipo de la operación que estemos analizando en el estado i.
- x(i) = Variación del marco temporal sufrido por dicha operación al ser localizada en el paso i.

Suponiendo que la operación pudiera inicialmente ser asignada en un rango $t \leq j \leq b$, su asignación al paso j, tendrá asociada una fuerza:

$$Fuerza\_propia(j) = DG(j) - \sum_{i=t}^{b} (DG(i)/h)$$

Siendo:

$$h = (b - t + 1)$$

La "Fuerza_propia" es la fuerza debida únicamente a la modificación del marco temporal de la operación que estemos analizando. Sin embargo la asignación considerada modificará tambien los marcos temporales de las operaciones predecesoras y sucesoras de la que estemos analizando. De esta modificación se derivará una fuerza que tendremos que tener en cuenta, sumandola a la "Fuerza_propia". El algoritmo asignará aquella operación cuya fuerza sea menor. El procedimiento puede resumirse en los siguientes pasos:

Repetir hasta que todas las operaciones hayan sido localizadas

1. Evaluar los marcos temporales.
   1.1. Encontrar la asignación ASAP.
   1.2. Buscar la asignación ALAP.
2. Determinar el grafo de distribución, de acuerdo con las asignaciones realizadas.
3. Calcular las fuerzas asignadas a cada operación.
4. Añadir las fuerzas asignadas a los sucesores y predecesores de cada sentencia.
5. Localizar la operación con menor fuerza.

Este algoritmo puede ser facilmente extendido para minimizar tambien el coste de los

registros, interconexiones o unidades funcionales. Para ello solo es necesario incluir en la fuerza, estimaciones del coste de estos parámetros al localizar una operación en un determinado estado. Además soporta facilmente la introducción de operaciones encadenadas, operaciones con pipeline estrutural u operaciones cuya ejecución conlleva mas de un ciclo. La técnica puede igualmente ser extendida para soportar pipeline funcional. Algunas de estas posibilidades se han implementado en PSAL.

Por otro lado, el concepto de fuerza puede ser utilizado como criterio de selección de operaciones en un algoritmo de tipo "list scheduling". En este tipo de algoritmos, las operaciones son ordenadas en orden topológico usando las precedencias dictadas por las dependencias de datos y control. A continuación, y de forma iterativa, se asignan todas las operaciones que serán realizadas en un determinado estado. Para decidir que operaciones de entre las posibles serán localizadas en el estado analizado en ese paso, se puede utilizar el concepto de fuerza descrito anteriormente. Otros sistemas utilizan otras funciones, como la movilidad o la urgencia. El algoritmo puede entonces ser descrito como:

1.- Inicializar el máximo temporal con la longitud del camino crítico.
2.- Para cada paso de control, desde el primero hasta el señalado por el máximo temporal.

2.1.- Determiar los marcos temporales.
2.2.- Determinar las operaciones que pueden ser realizadas en el paso de control que estamos analizando.
2.3.- Mientras que el número de posibles operaciones a localizar sea mayor que el número máximo de unidades operacionales señaladas por el usuario:

2.3.1.- Si todas las operaciones pertenecen al camino crítico:

2.3.1.1.- Incrementar el máximo temporal.
2.3.1.2.-.Redefinir los marcos temporales.

2.3.2.- Calcular las fuerzas.
2.3.3.- Seleccionar la operación con menos fuerzas, y marcarla como no localizable en el paso de control actual.

2.4.- Asignar las operaciones restantes al paso de control analizado.

Esta técnica permite además introducir restricciones tales como número máximo de operaciones localizables en OU en un estado, o limitar el número de operaciones de un tipo en un estado, lo cual permite reducir de forma muy eficiente el coste del sistema.

IV) Asignación de operaciones basada en análisis de caminos

Estas técnicas han sido desarrolladas por Camposano y Bergamaschi [CaBe90][CaBe89] y actualmente estan implementadas en el sistema de síntesis de alto nivel de IBM. Se definen dos estrategias: estrategia de "spliting" (originalmente implementada en el YSC) y la técnica AFAP ("as fast as possible").

La primera técnica localiza inicialmente todas las operaciones en un único estado, realizandose a continuación diversos cortes ("split") con objeto de satisfacer las dependencias y los requerimientos solicitados por el usuario. El algoritmo determina todas las restricciones examinando todas las posibles secuencias de ejecución, o caminos, en el grafo de flujo de control. De esta forma, en el ejemplo de la figura 1.10, la restricción "A" impide que las operaciones 2, 3 y 4 sean ejecutadas en el mismo estado. El algoritmo fue implementado con un "backtracking" recursivo, que analiza todos los posibles cortes para cada camino y busca la solución con mínimo coste. Dado que para ejemplos reales este algoritmo podría tener un tiempo de ejecución muy elevado, se implementaron 3 heuristicas diferentes (por lo cual no se asegura que la solución sea óptima):

a) Cortar cuanto antes. Siempre que una restricción permita un rango de puntos de corte, se selecciona el primero. Es equivalente a la técnica ALAP. En la figura 1.11 esta metodologia esta reflejada en el 1º corte, (el corte en 2).

b) Cortar cuanto mas tarde. Dado un rango de puntos de corte (en el ejemplo de la figura 1.11, el rango es : 2, 3 , 4) se selecciona el ultimo punto de corte. Es equivalente a la técnica ASAP. En el ejemplo de la figura 1.11 equivale al corte en 4.

c) Corte adelantado. Un parámetro del sistema (definido por el usuario) es usado para limitar la profundidad de la recursión del algoritmo. La función de coste utilizada es la suma del número de estados y registros necesarios.

En esta metodología, lazos y operaciones condicionales tienen un tratamiento especial dado que introducen condiciones de corte adicionales.

La ventaja de este algoritmo es que el concepto de restricciones es muy general y contiene desde dependencias de datos hasta restricciones temporales o de coste.

La técnica AFAP ("as fast as possible") realiza una asignación de estados encontrando una solución global optima: encuentra el mínimo número de pasos de control, para unas restricciones dadas, para todas las posibles secuencias de operaciones.

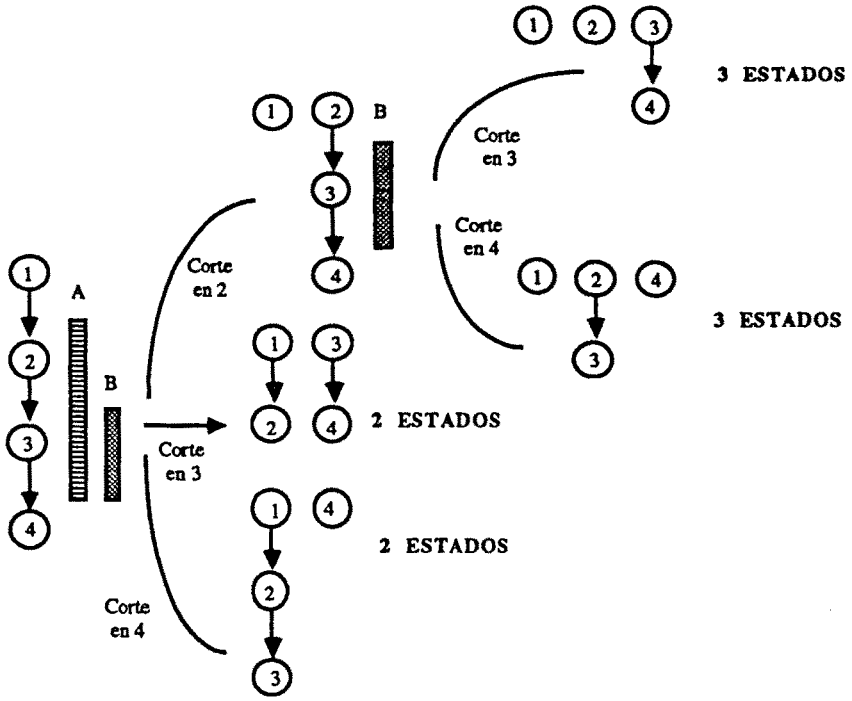


figura 1.10

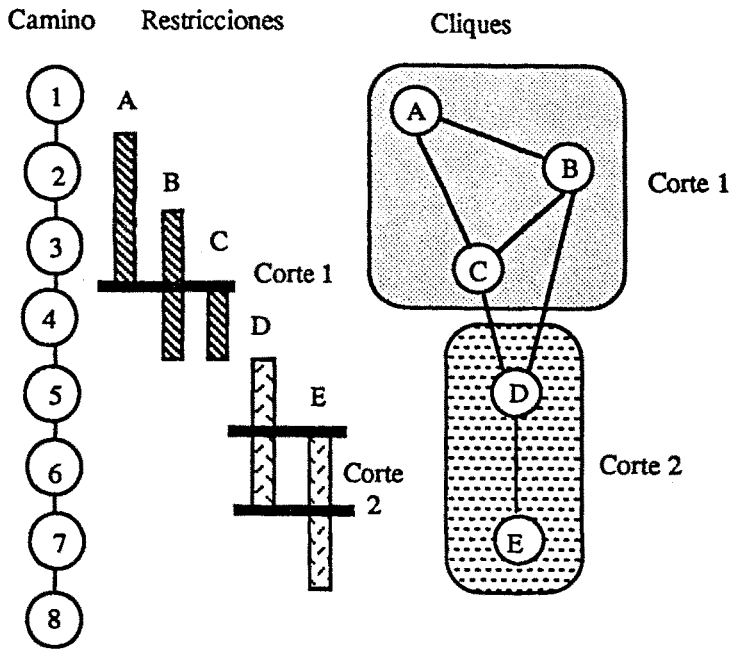


Figura 1.11

Este algoritmo computa, en primer lugar, todos los posibles caminos. Los lazos son abiertos, de forma que que el primer nudo sea el primer nudo de un nuevo camino. Al igual que en el algoritmo anterior las restricciones se modelan como intervalos, en los cuales debe ser localizado un corte.

Estas restricciones pueden ser representadas como un grafo, en el cual los nudos representan restricciones, y los arcos, la posibilidad de un corte común a las restricciones cuyos nudos asociados, dicho arco conecta (es decir, cada arco modela el solapamiento entre dos restricciones). Este grafo de intervalos permite definir el minimo conjunto de cortes necesarios para satisfacer todas las dependencias en un camino. Para ello es necesario encontrar la partición que produzca el menor número de cliques.

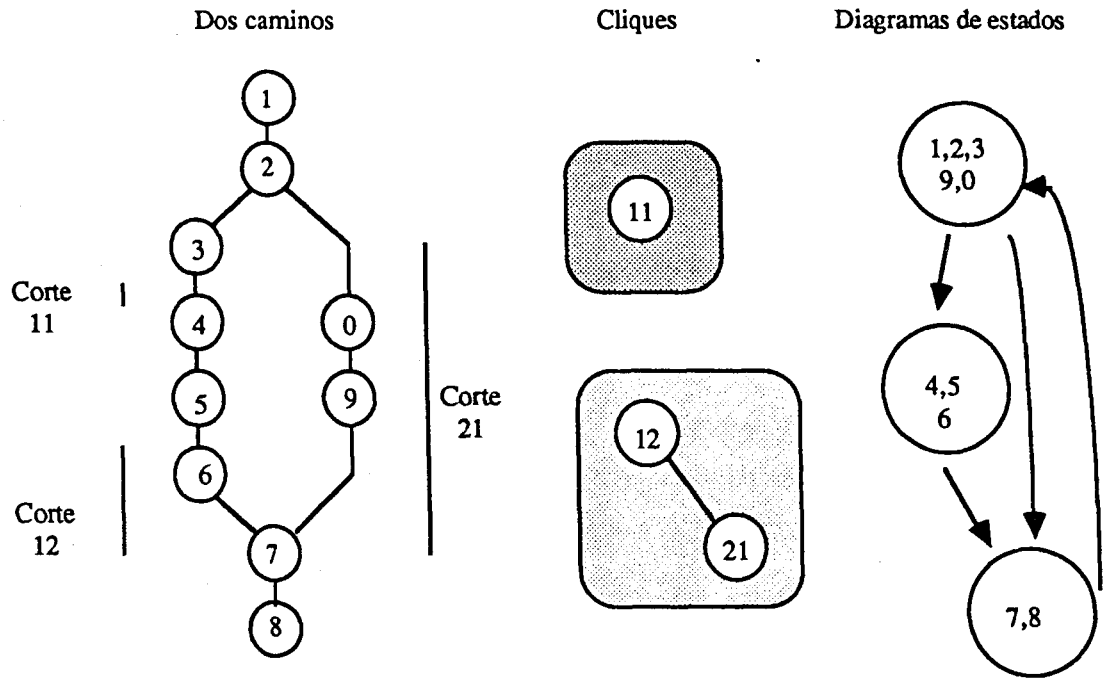


Figura 1.12

Cada clique en dicha partición es asociado a un corte (ver figura 1.12). Para encontrar el minimo número de estados en todos los caminos, se forma un nuevo grafo, en el cual los nudos son los cortes de los grafos de intervalos en cada camino y los arcos modelan nudos comunes a diferentes caminos. Una partición en un minimo número de cortes es entonces realizada, obteniendose con ello el menor número de cortes necesarios para satisfacer las restricciones y, con ello, el menor número de estados.

V) Compilación inteligente

La "compilación inteligente" constituye una técnica novedosa, que conjuga el uso de sistemas expertos y algoritmos tradicionales. En esta metodologia existe un modulo encargado de evaluar la situación actual del diseño y otro encargado de realizar la síntesis. Estos modulos se ejecutan sucesivamente, dando lugar a una ejecución en lazo cerrado (a una evaluación sigue una síntesis, tras la cual se realiza una nueva evaluación, repitiendose de nuevo el proceso). Esta secuencia, evaluación-síntesis, se repite hasta que se verifican las restricciones impuestas por el diseñador. Esta aproximación ha sido utilizada en los trabajos de D. Gajsky y ha tenido como último resultado el sistema CHIPPE. En este sistema, el evaluador estima si se verifican las restricciones y en caso negativo modifica los recursos hardware de que dispone el sistema de síntesis y reconstruye nuevamente el diseño. CHIPPE usa un sistema experto dirigido por restricciones globales y el estado actual del diseño para obtener las restricciones que serán utilizadas por las herramientas de síntesis algoritmica. La comunicación entre el sistema experto y las herramientas se realiza siguiendo una filosofia de "knobs" y "gauges" (botones e indicadores). Los "gauges" son las medidas que dan una idea de la calidad del diseño actual. Los "knobs" son implementados como restricciones a las herramientas de síntesis.

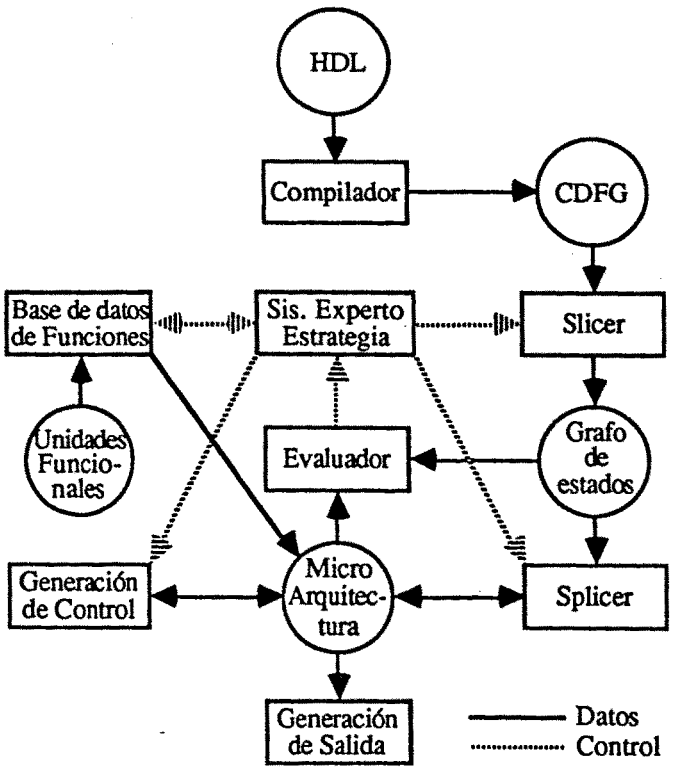


Figura 1.13

El esquema del sistema CHIPPE aparece en la figura 1.13. Partiendo de un HDL (similar a ISPS y PASCAL), el sistema genera un CDFG. Este grafo, junto con la estructura del diseño parcial configuran el estado actual del diseño. Los sistemas SLICER y SPLICER son utilizados para realizar la asignación de estados y la localización del data-path.

Hay dos tipos basicos de funciones de evaluación en CHIPPE :

- Funciones que soportan comparaciones globales
- Funciones que cuantifican el diseño.

Ambas funciones toman en consideración la probabilidad de ejecución de cada camino condicional, lo que en terminos de microprocesadores equivale a la probabilidad de ejecución de una instrucción.

Las funciones de comparación global miden parametros tales como coste de interconecciones, del control, etc. Ejemplos de medidas de calidad son las estadisticas de uso de las unidades funcionales, medidas de solapamiento de unidades, tiempos de reloj muerto, y medidas relativas de uso de unidades funcionales, multiplexores, buses y unidades de control.

VI) Técnicas de síntesis de alto nivel mediante modelos de programación lineal con variables 0-1

Los métodos de programación lineal han sido utilizados tanto en técnicas de asignación simultanea de operaciones en estados y de variables en registros [PaKo90], como en la síntesis del data path [HwHs90][Ba88]. Un ejemplo de utilización de estas técnicas lo constituye la metodologia propuesta en [Ba88] para la localización de variables en memorias con multiples accesos, en vez de registros. Estas técnicas son también aplicables a la localización de buses.

Estos algoritmos asumen que en una memoria con m accesos puede ser localizado un conjunto de registros si no se accede simultaneamente a más de m registros del conjunto. El algoritmo trata de agrupar las variables reduciendo, además del número de memorias, las interconexiones entre éstas y las unidades operacionales.

Supongamos una memoria M con m accesos, de los cuales r son de solo lectura, w de solo escritura y rw de lectura-escritura (rw= m- (r +w)). Supongamos que el diseño involucra

k pasos de control, S1,...,Sk, y N registros, R1,...,RN. Para un registro Ri y para un estado Sj se definen:

- aij . Esta variable valdrá 1 si Ri es leido en el estado j. Valdrá 0 en el caso contrario.
- bij . Valdrá 1 si Ri es cargado en el estado j. Tomará el valor 0 si no lo es.
- cij . Esta variable valdrá 1 si Ri es leido o cargado en el estado j. Será 0 si no es así.

A continuación se asocia una variable entera 0-1, xi, al registro Ri. Esta variable tomará el valor 1 cuando dicho registro pertenezca al conjunto de registros localizado en la memoria M. Valdrá 0 en caso contrario.

Encontrar el máximo conjunto de registros que pueden ser agrupados en la memoria M, puede ahora ser tratado como el problema de programación linear 0-1 siguiente:

$$\max \sum_{i=1}^{N} xi$$

teniendo en cuenta:

$$\sum_{i=1}^{N} xi*aij \leq r+ rw \qquad j=1,...,k$$

$$\sum_{i=1}^{N} xi*bij \leq w+ rw \qquad j=1,...,k$$

$$\sum_{i=1}^{N} xi*cij \leq m \qquad j=1,...,k$$

Una vez determinados los registros que pueden ser localizados en la memoria, se debe determinar la asignación registro-acceso de forma que se reduzcan las interconexiones. Se denomina Pt (t=1,...,m) a los accesos a la memoria, y como Eq (q=1,...,d) a los terminales de las unidades operacionales a las cuales el registro debe de ser conectado. Se definen, para el estado Sj, las siguientes variables 0-1:

- CEiqj . Esta variable será 1 si Ri necesita ser conectado a Eq.

- CPitj . Esta variable será 1 si Ri es asignado al acceso Pt.

- EXtqj . Esta variable será 0, si en algún estado Sv ($1 \leq v \leq j-1$) anterior a Sj, se estableció una conexión entre Pt y Eq, como consecuencia de asociar el registro Ri ($1 \leq i \leq N$) al acceso Pt, teniendo que estar conectado al terminal Eq.

La asignación del registro Ri al puerto Pt en el estado Sj, genera una nueva conexión entre Pt y Eq si la variable CSitj toma una valor mayor que 0 en la expresión:

$$CSitj = \sum_{q} CEiqj * EXtqj$$

A esta variable se la denomina coste de asignación de Ri a Pt en el estado Sj. Para realizar una asignación variable-acceso que miminice el coste de la interconexión se utiliza el siguiente algoritmo:

inicialización: $EXtq1 = 1$ $t=1,...,m$ $q=1,...,d$
for cada estado Sj, $j=1,...,k$, **do**
begin
Calcular CSitj ($i = 1,...,N$ $t = 1,...,m$).
Resolver el problema de programación lineal 0-1, que se plantea a continuación, de forma que los coeficientes CSitj encontrados, minimizen el coste de las interconexiones ∂:

$$\partial = \sum_{i} \sum_{t} CSitj * CPitj$$

con las restricciones:

$$\sum_{t} CPitj \geq cij \qquad i=1, \ldots ,N.$$

$$\sum_{i} CPitj \leq 1 \qquad t=1, \ldots ,m.$$

Calcular:

$$EXtqj+1 = EXtqj - \sum_{i} (CPitj * CE\,iqj) \qquad t=1, \ldots ,m. \quad q=1, \ldots,d.$$

end

Este algoritmo tanbien es aplicable a la localización de interconexiones en buses.

VII) Técnicas de localización simultanea de registros, unidades aritméticas e interconexiones y de asignación de estados

Se han propuesto diversas técnicas que premiten realizar una síntesis global, es decir, que afrontan todas las tareas de localización de UO's, registros, interconexiones y operaciones en estados simultaneamente[SaZa90][PaKo90][BePa90][Gr90][ClTh90]. Estas técnicas utilizan varias metodologias para afrontar la síntesis:

- Las técnicas basadas en fuerzas [ClTh90] constituyen refinamientos del algoritmo de loacalización de operaciones basado en fuerza expuesto en el apartado III de este capítulo. En ellas, una vez seleccionadas las operaciones a ejecutar en un ciclo de reloj se realiza una localización de los elementos hardware (registros, unidades operacionales, etc). Esta asignación de recursos hardware será tenida en cuenta a la hora de realizar nuevos calculos de la fuerza. El calculo de dicho parámetro incluye, además de los factores comentados en en apartado III, factores relacionados con la concurrencia de operaciones y la compatibilidad de conexiones. Otras técnicas unen a estos conceptos, técnicas de programación lineal [PaKo90].

- Técnicas que utilizan "simulated annealing" [SaZa90]. En ellas, un cierto parámetro (denominado temperatura), determina que soluciones deben de ser desechadas. El algoritmo determina el coste de las posibles soluciones. Cuando dicho coste supera un cierto valor (determnado por la temperatura), la solución es desechada. En esta función de coste se tienen en cuenta aspectos relacionados con el "scheduling" (como el número de ciclos de reloj) y con la localización (como el coste de los diversos módulos).

- Técnicas que utilizan métodos de "Branch and Bound"[BePa90]. El sistema "SCHALLOC" desarrollado por Pangrle y Berry utiliza un algoritmo "branch-and-bound" que es utilizado en el "scheduling". La asignación de recursos hardware es realizada en cada paso intermedio del algoritmo. Los costes de esta localización son usados para dirigir la busqueda a la solución óptima. El programa utiliza funciones heuristicas para estimar el coste del sistema con la asignación de estados parcial.

Por otro lado se han propuesto técnicas que realizan una localización global de los recursos hardware [ScGr90][Gr90][NeDe87]. Estas técnicas tratan de sintetizar el "data path", de forma que se minimize una cierta función de coste f, dependiente del tiempo de ejecución y

del coste del hardware[NeDe87]. La síntesis del "data path" puede ser modelada como un problema de localización de microinstrucciones en un espacio bidimensional (tiempo - unidad operacional), de forma que se reduzca una cierta función de coste f. La existencia de sentencias condicionales en la descripción provoca la aparición de operaciones disjuntas. Decimos que dos operaciones son disjuntas si no se pueden ejecutar simultaneamente, debido al flujo de control de la descripción. Por ejemplo, en una sentencia condicional del tipo "IF", la clausula "THEN", se ejecuta cuando no lo hace la "ELSE", y viceversa. Las operaciones de la clausula "THEN" serán, por lo tanto, disjuntas de las de la clausula "ELSE". Dos operaciones disjuntas pueden ser localizadas en el mismo estado y en la misma unidad operacional, es decir, en una misma posición del espacio bidimensional tiempo - unidad operacional. Por ello se introduce una tercera dimensión, que permite reflejar esta situación de forma adecuada ya que las distintas operaciones disjuntas son asociadas a distintos niveles de esta tercera dimensión.

Para localizar las operaciones en este espacio tridimensional son varias las técnicas citadas en la bibliografía:

- Simulación "annealing" [NeDe87].
- Algoritmo "branch and bound"[Gr90].
- Técnicas de síntesis "bottom-up"[SrGr90]

En la primera técnica deben de tenerse en cuenta dos aspectos básicos:

- Generación de nuevos estados. Un nuevo estado puede ser generado de tres formas:

    * Intercambiando dos operaciones.
    * Localizando una operación en otro estado.
    * Intercambiando operandos en una operación conmutativa.

- Función de coste que el algoritmo debe de minimizar. En esta función se tendrá en cuenta el coste de ejecución y el localización de recursos hardware (registros, OU's e interconexiones).

En cuanto a la segunda técnica, esta inspirada en algoritmos similares empleados para localizar programas en uno de los N procesadores de un sistema multi-procesador. Existen, sin embargo, dos diferencias:

- El número de "procesadores" no es conocido.
- La función coste no depende solamente del tiempo de ejecución.

El algoritmo va buscando iterativamente la configuración en la cual se minimize el coste total de registros, operaciones-ALU´s y buses-interconexiones.

La última técnica comentada [ScGr90] esta en la linea del sistema BUD-DAA. En ella el "data pah" se va transformando sucesivamente utilizando un método de "clustering" jerarquico, considerandos además el efecto de las restricciones de scheduling. Al final de la síntesis el scheduling no esta completamente definido, permitiendofuturas optimizaciones.

VIII) Metodologias orientadas a la explotación del paralelismo

Se han propuesto metodologias que tratan de sacar el máximo partido a la tecnologia VLSI, mediante arquitecturas regulares, con conexiones muy localizadas. Una parte importante de estos algoritmos han sido desarrollados para compilar programas que debían ser ejecutados en un sistema multi-procesador. Otra parte la constituyen aquellas destinados a generar, a partir de un algoritmo, una estructura hardware especial que los implemente, aprovechando al máximo el paralelismo implícito en la descripción (ejemplo: arrays sistólicos). Pero en ambos casos, los algoritmos ponen el acento en la optimización de un tipo de estructura: los lazos. Esto es debido, a que los lazos son los máximos candidatos a consumir tiempo de entre las estructuras que aparecen en el algoritmo. Existen también trabajos [GeMo86], que difieren un tanto de la filosofía de los algoritmos que vamos a ver. Estas metodologías utilizan modelos estadísticos para el análisis del sistema. Estas técnicas tienen una prometedora utilización en síntesis de alto nivel.

7.1) Técnicas de compilación para sistemas multiprocesadores [PaKu80][BaSh79].
Estas técnicas suelen limitar el número de tipos de sentencias. Las sentencias consideradas en el caso mas general son :

- go to
- if
- asignación
- for

Además, se supone que el programa se constituye como una secuencia que "fluye hacia adelante" ("forward-flow sequence"). Es decir, no se permiten saltos hacia zonas del programa ya ejecutadas. En esta metodología aparecen una seria de definiciones básicas. Dadas dos sentencias S y T de un algoritmo, tales que S precede a T en la secuencia, se dice que:

a) T es dependiente de los datos de S si y sólo si existe una variable simple v que

recibe un valor en la sentencia S y luego dicho valor es usado en la sentencia T.

b) T es antidependiente de S si y sólo si existe una variable v tal que T asigna un valor a v y S lo utiliza.

c) T es dependiente de la salida de S si y sólo si existe una variable v tal que ambas sentencias asignan valores a v.

d) T es dependiente del control de S, si la ejecución de T depende del valor que toma la variable que se asigna en S. Este tipo de dependencia se incluye para poder analizar sentencias condicionales. Estas sentencias sufren una transformación previa. En el siguente algoritmo puede verse un ejemplo:

```
IF A<5  THEN  B=3;  GO TO 4
    B=4;
4:  ...
```

Al aplicarle las transformaciones reseñadas se obtiene:

```
S   b = A<5
T   B=3 when b
C   B=4 when not b
```

En este ejemplo, las sentencias T y C dependen del control de S.

Para poder aplicar esta nomenclatura a grafos, estas definiciones se extienden. Supongamos que tenemos un algoritmo formado por varios lazos anidados:

```
for i1=1 to n1 do
   for i2=1 to n2 do
   ...
      for ik=1 to nk do
       ...
        S : ...
       ...
       end
   ...
   end
end
```

Sea S una sentencia interna a dichos lazos. Entonces, definimos una "instancia" de S y la representamos como $S_{l_1 l_2 \ldots l_k}$, como el valor de S cuando $i_1=l_1,\ldots,i_k=l_k$. Entre estas instancias, se producen las mismas dependencias que entre sentencias. En este contexto decimos que una sentencia T es dependiente de datos, control, salida o antidependiente de una sentencia S si y sólo si existen instancias $S_{l_1 l_2 \ldots l_k}$ y $T_{l_1 l_2 \ldots l_k}$ que poseen la misma dependencia. Asi dado el siguiente algoritmo:

```
for i=1 to n do
    for j=1 to n do
        S:   A(i,j)=A(i-1,j)+C(j)+C(j-1)
        T:   C(j)=D(j)+A(i,j)
    end ;
end ;
```

Las dependencias que se establecen son:

S es dependiente de los datos de S
T " " " " " " " S
T " " " " " " " T
T es dependiente de la salida de T
T es antidependiente respecto de S

Con objeto de reducir el número de dependencias se aplican una serie de técnicas:

a) Expansión escalar.
Esta técnica consiste en sustituir las variables escalares por variables vectoriales.Asi el algoritmo:

```
b=a(i)+1
h(i)=b+c(j)
```

se transforma en:

```
b(i)=a(i)+1
h(i)=b(i)+c(i)
```

Con esta transformación se eliminan antidependencias y dependencias de la salida.

b) Renombrado

Se actua sobre las variables que toman mas de un valor en una iteración. Lo que se hace es cambiar de nombre a la variable en la segunda asignación. De esta manera el algoritmo:

b=a+c
t=b+1
b=o
p=b+1

se transforma en :

b=a+c
t=b+1
bb=o
p=bb+1

Gracias a estos cambios, en el algoritmo solo habrá dependencias de datos (las de control no las consideramos).

Asociado a toda dependencia de datos existe un vector distancia. Dadas dos instancias $S_{i_1 i_2 \ldots i_k}$ y $T_{j_1 j_2 \ldots j_k}$, tales que la primera es dependiente de datos de la segunda, se define el vector distancia como:

$<m_1, m_2 \ldots m_k> = <i_1 - j_1, \ldots, i_k - j_k>$

Asi la sentencia:

A(i,j)=A(i-1,j+1)+A(i-1,j)

genera, respecto de si misma, dos vectores distancia de valor

<1,-1>,<1,0>

Las dependencias entre sentencias suelen representarse en forma de grafo. Así, S depende de los datos de T se representa:

Se denomina "grafo de dependencias " a un grafo en el cual se representan las relaciones existentes entre sentencias. Se define un Π-bloque como un conjunto de sentencias fuertemente conectadas en el grafo. Supongamos que tenemos el grafo de dependencias de la figura 1.14:

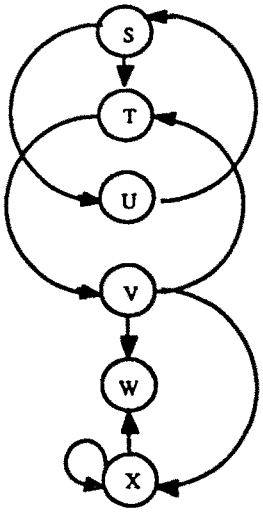


figura 1.14

En este grafo se pueden diferenciar cuatro Π-bloques:

$\Pi 1=\{S,U\}$
$\Pi 2=\{T,V\}$
$\Pi 3=\{W\}$
$\Pi 4=\{X\}$

Entre Π-bloques también se pueden establecer dependencias (se extiende a los Π-bloques las dependencias existentes entre las sentencias que lo forman). Así el $\Pi 2$ es dependiente de datos de $\Pi 1$ (ya que T lo es de S).

Uno de los principales usos de los Π-bloques es la distribución de lazos. Dado el

siguiente algoritmo :

```
for i=1 to n do
S:    A( i ) = B( i-1 ) + 1
T:    C( i ) = A( i ) + d( i-1 )
U:    B( i ) = A( i ) + 1
V:    D( i ) = C( i ) + 1
W: E( i ) = D( i ) + F( i-1 )
X:    F( i ) = D( i ) - F( i-1 )
end
```

Su grafo de dependencias es el que aparece en la figura 1.14. Conociendo los Π-bloques, podemos rescribir el algoritmo,modificando los lazos. El resultado sería:

```
for =1 to n do
    S:    ...
    U:    ...                    Π1
end
for i=1 to n do
    T:    ...
    V:    ...                    Π2
end
for i=1 to n do
    W:    ...                    Π3
end
for i=1 to n do
    X:    ...                    Π4
end
```

Las transformaciones básicas que se pueden realizar para aumentar el paralelismo aparecen a continuación:

a) Transformación "forall".

En este caso se convierte cada iteración del lazo, en una tarea para un procesador. En un determinado instante habrá varios procesadores que estén realizando operaciones del lazo, pero de tal forma que el i-ésimo procesador realice sólo operaciones correspondientes a la i-ésima iteración. Al aplicar esta transformación al algoritmo anterior se obtiene la siguente tabla de distribución temporal de tareas:

PROCESADOR	TIEMPO ->							
1	S1	U1	T1	V1	X1	W1		
2		S2	U2	T2	V2	X2	W2	
3			S3	U3	T3	V3	X3	W3
...					...			

La secuencia de operaciones en los distintos procesadores se obtienen a partir de los Π-bloques.

b) Transformacion "pipeline".
La filosofia "pipeline" consiste en repartir los Π-bloques entre los procesadores, en lugar de las iteraciones.

Con ellos se reduce el número de procesadores necesarios a costa de aumentar el tiempo de ejecución. Así para el ejemplo anterior, la tabla de asignación temporal de tareas que se obtiene es:

PROCESADOR	TIEMPO->							
1	S1	U1	S2	U2	S3	U3	S4	...
2		T1	V1	T2	V2	T3	V3	...
3			X1		X2			...
4				W1		W2		...

7.2) Mapeado de algoritmos en arrays sistólicos.

Un trabajo clásico en este tema lo constituye el desarrollado por D. Moldovan [Mo82] [MoFo86]. Esta metodología parte de un algoritmo constituido por un conjunto de lazos anidados, y trata de implementarlo mediante un array sistólico. Esta arquitectura consta de un conjunto de celdas distribuidas en forma de filas y columnas con conexiones locales, ésto es, cada celda se comunica únicamente con sus vecinas (figura 1.15).

El resultado de la síntesis proporciona el número de filas y columnas que se necesitan asi como la función que realiza cada celda y las interconexiones que existen entre ellas.
Este metodología se podría resumir de la siguiente forma: las operaciones en un lazo se realizan en "orden lexicográfico", ésto es, en el orden que marcan los índices. Ellos son quienes señalan la secuencia de operación. En un algoritmo convencional damos a estos índices un sentido temporal (esta operación se realiza antes que la otra). Es posible, sin embargo, asignarles un significado espacial. Así la operación a(3,4)=a(3,3)+1 podría entenderse como el incremento en una unidad de la señal de salida de la celda (3,3). La

operación se realizará en la celda (3,4), siendo este resultado utilizado por las celdas adjacentes.

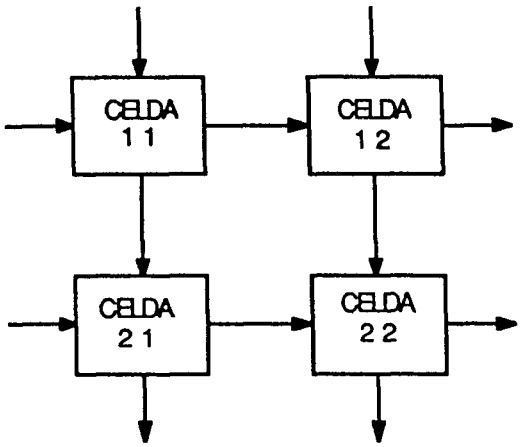


figura 1.15

La finalidad de la metodología es encontrar una transformación que produzca un nuevo conjunto de índices, en el cual, el primero indique (por ejemplo) el instante en el cual se realiza la operación, y los restantes, las cordenadas X e Y de la celda en la cual se realizan. Este nuevo conjunto de índices debe respetar las dependencias de datos del algoritmo primitivo. Sea T la transformación encontrada, y sea D la matriz cuyas columnas son los vectores distancia, d, asociados a las dependencias de datos. La matriz de transformación T debe verificar la condición:

$d>0 \Rightarrow r=T(d)>0$

Esto significa que la transformación preserva el sentido de las dependencias de datos. Si una operación B depende de los datos de A ($d>0$) en el algoritmo primitivo, en el nuevo, este orden de ejecución debe mantenerse ($r>0$). En esta metodología se divide la matriz T en dos partes asignandose a una un sentido temporal y a la otra espacial.

Este método fue ampliado [MoFo86], con objeto de que el array sistólico pudiera tener un tamaño fijo. Para ello se realiza una partición del conjunto de índices en planos Π, de forma que el array procese un único plano cada vez. Ello hace que el tiempo de ejecución aumente, pero mejora el coste y el rendimiento del sistema permitiendo el mapeado del algoritmo bajo las restricciones de área impuestas por el usuario.

IX) Algoritmos aplicados en compiladores.

Existen técnicas de optimización de código utilizadas por los compiladores de lenguajes software que han sido utilizadas en los sistemas de síntesis de alto nivel. Además de optimizaciones algunas de las representaciones intermedias utilizadas en compiladores sistemas han sido empleadas como base de las utilizadas en los sistemas de síntesis. Por esta razón será realizado una breve descripción de estas técnicas, dado que han servido de base a varias transformaciones utilizadas los sistemas de síntesis.

Los compiladores de lenguajes de alto nivel suelen utilizar, para representar el algoritmo descrito en el programa, un grafo denominado "grafo de flujo" [AhSe86]. Los nudos de este grafo son bloques básicos y los arcos (dirigidos), reflejan el flujo de control. Un bloque básico es un conjunto de sentencias cuya secuencia de operación tiene un único punto de entrada (la primera operación) y un único punto de salida. Generalmente, la última sentencia de un bloque básico es una instrucción que implica una bifurcación del flujo de control. En ocasiones, en la primera sentencia confluyen varias secuencias de control. Comentaremos a continuación aquellas transformaciones que no fueron mostradas anteriormente:

a) Renombrar variables temporales.
Esta transformación ya fue vista en el punto anterior (renombrado). La diferencia es que en esta ocasión lo que se intenta es transformar las variables virtuales, que se generan en el compilador al representar en nomenclatura polaca (u otra equivalente) expresiones complejas. El objetivo es que dichas variables sólo se definan una vez. Con ello cada variable virtual sólo tomará un valor. Se definen los bloques básicos que verifican esta propiedad como bloques en forma normal. Esta transformación permite la reducción de variables temporales.

b) Intercambio de sentencias.
Supongamos que tenemos un par de sentencias consecutivas en un bloque básico. El orden de dichas sentencias será intercambiable si la variable definida en la primera no es usada en la segunda y viceversa.

c) Optimización de "peephole".
Esta técnica intenta mejorar las características del programa, optimizando pequeñas secuencias de operaciones que reciben el nombre de "peephole". Es una optimización local pero muy efectiva. A continuación serán mostradas algunas transformaciones de este tipo:

c.1) Eliminación de lecturas y almacenamientos redundantes.

Si nos fijamos en la siguiente secuencia:

```
b=c
c=b
```

observamos que la segunda operación es redundante.

c.2) Eliminación de código desechable.
Esta propiedad es similar a la reducción de constantes. Simplemente se calculan, en tiempo de simulación, todas las operaciones que sean posibles.

c.3) Optimización del flujo de control.
En este paso se intentan eliminar saltos innecesarios. Así el algoritmo:

```
if a<b then go to L1
  ...
  L1:  go to L2
```

se puede transformar en:

```
if a<b then go to L2
  ...
  L1:  go to L2
```

con lo que hemos eliminado un salto innecesario.

c.4) Simplificación algebraica.
Varias propiedades algebraicas pueden ser utilizadas para simplificar el algoritmo, como por ejemplo :

```
x=x+0
x=x*1
```

Otras propiedades, como la conmutatividad o la propiedad distributiva, pueden utilizarse para aumentar la eficiencia del algoritmo. Asi x*(y+z) es una sentencia más eficiente que x*y+x*z.

c.5) Reducción de potencia.
Trata de realizar operaciones muy costosas de forma más barata. Por ejemplo:

$y=x^2$

se transformaría en

y=x*x

c.6) Utilización de operaciones especiales.
Existen operaciones que pueden ser realizadas más eficientemente con un hardware especial. Así:

a=a/2

equivale a desplazar hacia la derecha una posición los bits del registro a.

d) Optimizaciones en lazos.
El primer paso en la optimización de los lazos es la determinación de la dominancia. Se dice que un nudo n domina a un nudo p, si todo camino que parte de n llega a p. En la figura 1.16, por ejemplo, el nudo 1 domina al 2 y al 3 y el 2 domina al 3.

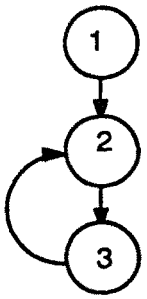


figura 1.16

Un lazo natural se define como una estructura en la que:

- Posee un único nudo de entrada llamado cabecera, que domina al resto de los nudos del lazo.
- Debe haber al menos un camino que comunique el último nudo del lazo con la cabecera.

Además, con el fin de simplificar el análisis del flujo de datos, se introduce el concepto de grafo reducible. Este grafo verifica dos propiedades:

- Los arcos que no vuelven sobre nudos anteriores forman un grafo no cíclico, en el cual es posible encontrar un camino desde el primer nudo del grafo de flujo a un nudo cualquiera.
- Los arcos que vuelven sobre los nudos anteriores forman todos parte de algún lazo.

En los lazos, además de las optimizaciones ya vistas, se suelen utilizarn la inducción de variables y la reducción de potencia. La idea es simplificar el cuerpo del lazo teniendo en cuenta que se conoce tanto el valor inicial como el incremento que sufre la variable de control del lazo. Consideremos el siguiente algoritmo:

```
        j=n
        t1=4*n
        r=a[t1]
        ...
B:      j=j-1
        t4=4*j
        t5=a[t4]
        if t5>v go to B
```

En este algoritmo la sentencia t4=4*j equivale a escribir t4=t4-4, siempre que t4 se inicialice con el valor 4*j. El algoritmo quedará:

```
        j=n
        t1=4*n
        r=a[t1]
        ...
        t4=4*j
B:      j=j-1
        t4=t4-4
        t5=a[t4]
        if t5>v go to B
```

Hemos realizado una inducción y una reducción de potencia (sustituir una operación cara por otra más barata).

X) Algoritmos de análisis del flujo de datos [AhSe86][AlO180].

Como ya se ha visto, el análisis del flujo de datos es un paso previo imprescindible para muchos algoritmos. Incluso para poder crear la representación VT se hace necesario realizar éste análisis. El objetivo es llegar a determinar en cualquier punto del grafo de flujo, qué variables están siendo utilizadas y cuales no. Para ello se definen una serie de conjuntos.

Sea S un nudo (bloque básico) de un grafo de flujo. Definimos :

in[S] = conjunto de variables activas a la entrada del nudo S.
out[S]= conjunto de variables activas a la salida del nudo S.
gen[S]= conjunto de variables generadas en el bloque S.
kill[S]= conjunto de variables que se dejan de utilizar a partir del bloque S.

Para cada estructura del programa se establecen unas ecuaciones entre dichos conjuntos. En la figura 1.17 aparecen un ejemplo.

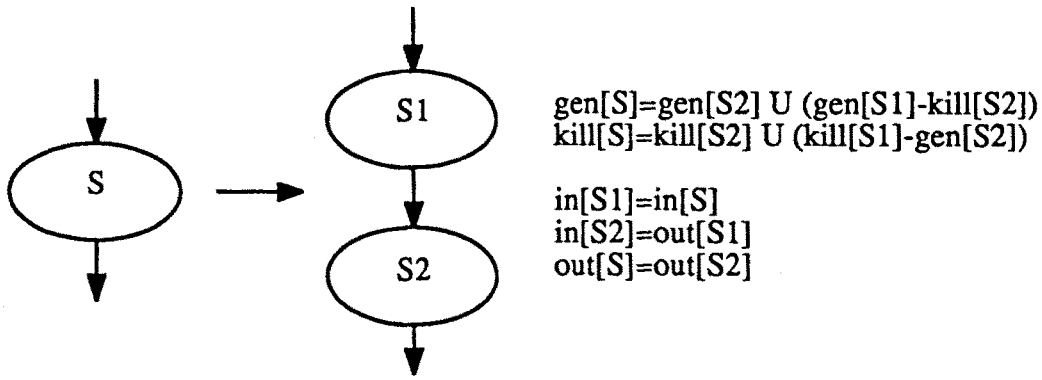


figura 1.17

Estas ecuaciones se resuelven por métodos iterativos. Dichos algoritmos, suponen inicialmente vacios los conjuntos anteriores. Iterativamente los van completando, de tal forma que se verifiquen todas las ecuaciones involucradas. El hecho de que, por ejemplo, los lazos sean reducibles, simplifica enormemente el cálculo de dichas ecuaciones. El resultado de este análisis serán unas tablas que nos indican que variables están activas en cada instante de programa. También se podría, con estos procedimientos, calcular cuando se generan y cuanto se usan las variables [AhSe86], con lo cual podríamos obtener el grafo de flujo de datos (o lo que es casi lo mismo, la representación VT).

CAPITULO 2

LENGUAJES DE DESCRIPCION HARDWARE

I) Introducción

Los Lenguajes de Descripción de Hardware (HDL's), Lenguajes de Descripción Hardware de Computadoras (CHDL's), Lenguajes de Transferencias de Registros (RTL's) ó también Lenguajes de Diseño (DL's), representan distintas acepciones que engloban aquellas notaciones designadas para describir diferentes aspectos o propiedades de un sistema digital. Se trata por tanto, de lenguajes orientados a la descripción de sistemas digitales. La gran variedad de lenguajes propuestos desde 1954 así como las distintas aplicaciones para las que se han diseñado han motivado la aparición de las anteriores acepciones.

En este capítulo, veremos algunas de las propiedades básicas de los lenguajes de descripción de hardware, para posteriormente centrarnos en el estudio de los lenguajes utilizados por los programas PSAL1 y PSAL2 para describir el sistema a diseñar. Como ya fue comentado en la introdución, el primero de estos sistemas utiliza ISPS como lenguaje de entrada a nivel algoritmico y DDL a nivel de transferencias de registros. La necesidad de contar con lenguajes de entrada y salida homogeneos, detectada en PSAL1, tuvo como consecuencia la elección de KARL IV y CVS_BK como lenguajes de entrada y salida del sistema PSAL2. Los problemas comentados en la introducción con KARL IV obligaron a buscar un nuevo lenguaje, optandose por un lenguaje estandar capaz de describir el sistema digital en diferentes niveles de abstracción, VHDL. PSAL2, por otro lado, parte de una descripción VHDL proporcionando la descripción de la arquitectura en cualquiera de los lenguajes CVS_BK, VHDL o ESP. Este último lenguaje fue introducido para conectar el sistema como las herramientas de síntesis estructural operativas en el grupo. De todos estos lenguajes se realizará una breve descripción de sus características para luego comentar su influencia en el proceso de síntesis y su utilización en PSAL. De VHDL se hará un análisis mas detallado ya que su utilización en síntesis de alto nivel requiere un estudio previo.

1) Niveles de descripción

Durante el proceso de diseño el sistema digital es descrito en diferentes niveles de abstracción, con distinto grado de detalle. El diseño generalmente se inicia en un nivel de gran abstracción y poco detalle, como el nivel algorítmico o el PMS y finaliza en el nivel "layout", con la total definición del conjunto de máscaras de fabricación del sistema. En los primeros niveles se proporciona información sobre el comportamiento del sistema sin dar detalles sobre su estructura y su topología.

Niveles de Descripción y Simulación

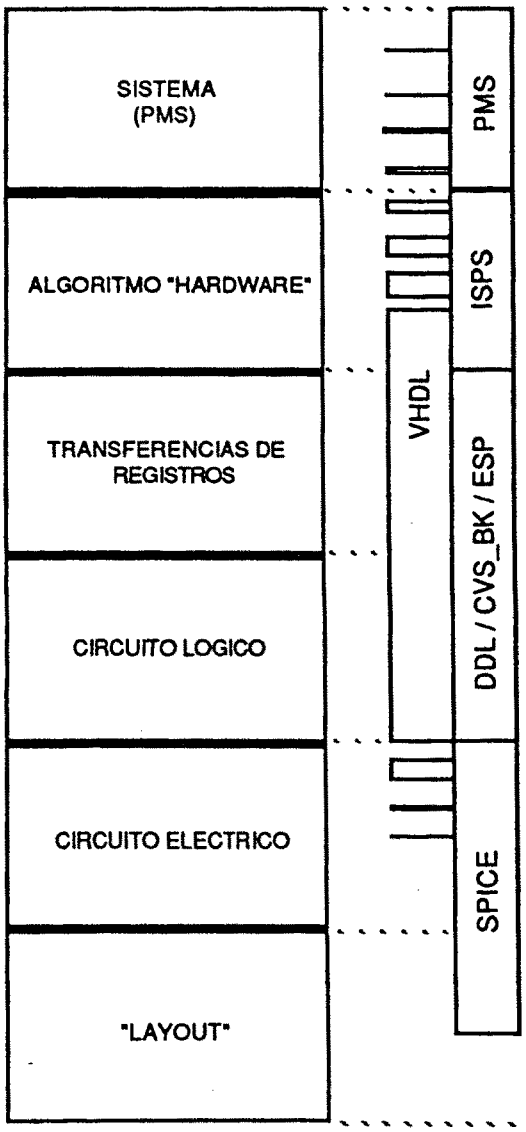


figura 2.1

Conforme avanza el proceso de diseño, se van añadiendo estructura, componentes y topologia al sistema, aumentando su grado de detalle, el cual es máximo en los niveles más bajos, en los cuales todo esta definido (existe poca abstracción).

En la figura 2.1 puede observarse una representación de estos niveles y de algunos de los medios de descripción que se disponen para ellos. Los niveles estan ordenados de mayor abstracción a mayor detalle.

Los medios de descripción y las herramientas CAD han sufrido una evolucion de abajo hacia arriba a medida que crece la complejidad de los sistemas tratados. Aun hoy, predominan en el proceso de diseño las herramientas próximas al silicio. La explotación completa de las posibilidades que la tecnología VLSI ofrece, requiere del uso de herramientas de diseño a niveles de abstracción mayores. A lo largo del proceso de diseño, desde la definición del algoritmo "hardware" hasta el "layout", es necesario seleccionar en cada caso el lenguaje idóneo para una determinada aplicación.

Los HDL's pueden ser contrastados a partir de las siguientes dimensiones:

* Complejidad de los componentes. Esta complejidad puede caracterizarse a partir de los elementos primitivos asumidos por el lenguaje (puertas, flip-flops, registros, unidades funcionales, procesadores, sistemas).

* Complejidad estructural. Es decir, el detalle con el que describe la colocación de los componentes, su interconexión, tamaño, e incluso, consumo, niveles de tensión, etc.

* Complejidad funcional. Esta complejidad puede medirse a partir de las operaciones primitivas permitidas en el lenguaje (booleanas, aritmeticas, relacionales, de control: selección y secuenciación, "timing", etc.).

* Sintaxis y semántica. Usualmente, al usuario se le suministra una descripción informal del significado de las construcciones permitidas en el lenguaje. Sin embargo, la mayoria de los lenguajes poseen una sintaxis formal descrita usualmente en alguna metanotación como la Backus-Naur-Form (BNF).

2) Principales lenguajes

Varios de los lenguajes mas importantes y de mayor difusión son los siguientes:

*VHDL (VHSIC Hardware Description Language). Se trata del lenguaje surgido del programa VHSIC y establecido como estandard por el IEEE. Es un lenguaje de una gran riqueza sintáctica, deribada del ADA, concebido como entrada a una base de datos de diseño sobre la que actuan las distintas utilidades (análisis, simulación, síntesis, ATPG, etc.). Cubre todos los niveles de descripción desde el sistema al nivel lógico en sus aspectos funcional y estructural.

* PMS (Processor-Memory-Switch). La notación PMS fué introducida en 1971 por Bell y Newell con objeto de describir sistemas de cómputo en los niveles mas altos de abstracción. Se trata principalmente de una notación gráfica destinada a describir la estructura del sistema de cómputo a partir de sus componentes (Procesadores, Memorias, etc.). La estructura del sistema se describe a partir de sus componentes y su interconexión. El lenguaje carece de cualquier tipo de declaración funcional.

*ISPS (Instruction Set Processor Specification). El ISP fué introducido inicialmente por Bell y Newell y posteriormente implementado como ISPS por Barbacci. Se trata de un lenguaje procesal orientado principalmente a la descripción de procesadores a partir de su conjunto de instrucciones. Desde el momento en el que no existe una correspondencia única entre la descripción y su implementación "hardware", se trata de un lenguaje de descripción a nivel algorítmico.

*AHPL (A Hardware [Hill] Programing [Peterson] Language). Es un lenguaje típicamente procesal deribado del lenguaje de programación APL (A Programing Language). De éste extrae un gran numero de operadores orientados a la descripción de "hardware".

*DDL (Digital [Donald] Design [Dietmeyer] Language). Es un lenguaje orientado a la descripción modular del sistema. Permite combinar estamentos de tipo procesal con otros de tipo no-procesal. Estos últimos adquieren la forma de maquinas de estados finitos.

*KARL (KAiserslautern Register transfer Languaje). Lenguaje desarrollado en la Universidad Alemana de Kaiserslautern por el Prof. Hartenstein a principio de los 80's. Es de tipo no-procesal de tal forma que la descripción del circuito se configura como una interconexión de celdas. KARL tiene asociado un lenguaje gráfico (ABL), cuyo editor ABLED, permite la descripción del circuitos de forma gráfica interactiva similar a la captura de esquemáticos a nivel lógico.

* CVS_BK (CVS Behavioral KARL). Extensión del lenguaje estructural KARLIII a la descripción de comportamiento, definida en el proyecto ESPRIT CVS. Fue

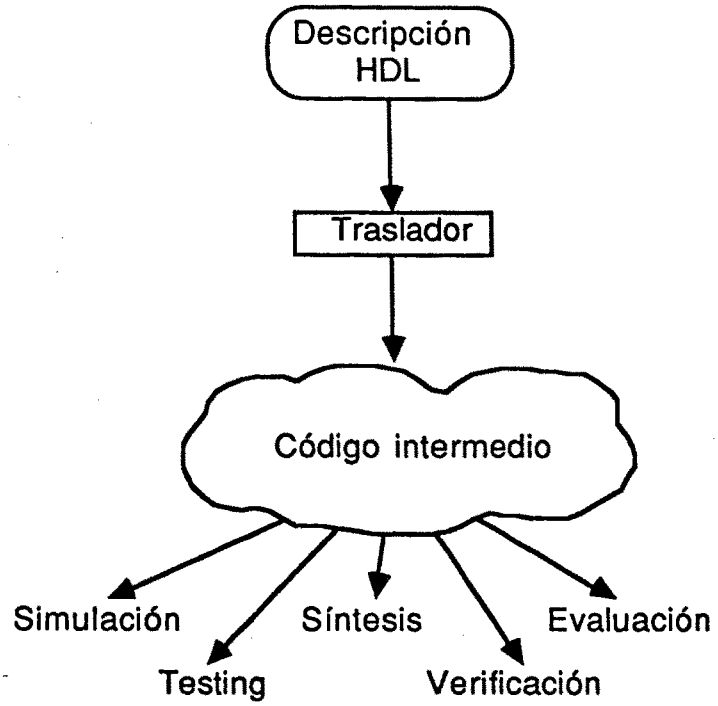
desarrollado por el mismo equipo que desarrollo KARLIII, que además de definir el lenguaje, extendio el simulador de KARLIII para poder incluir la nueva representación.

3) Aplicaciones

Las principales aplicaciones de los HDL's son las siguientes:

* Descripción. La utilidad mas inmediata de estos lenguajes es la de describir formalmente el comportamiento y/o la estructura de los sistemas digitales. La descripción del sistema digital mediante un HDL tiene tres utilidades principales:

a) Medio de entrada a utilidades de cómputo. Es sin duda la utilidad mas importante. La descripción HDL del sistema digital es el medio de introducción del diseño en el computador para arrancar distintas utilidades. Usualmente, estas utilidades no trabajan directamente sobre la descripción HDL sino que ésta se somete a una traslación inicial con objeto de detectar posibles errores sintácticos y crear un fichero intermedio que es el que sirve de entrada a las distintas utilidades:



b) Documentación. La descripción HDL del sistema digital propuesto se usa en la documentación del proceso de diseño al nivel funcional.

c) Comunicación entre diseñadores. La descripción HDL del sistema digital sirve como medio de intercambio de ideas en las etapas iniciales del diseño.

*Síntesis. La síntesis es el proceso de traslación del conjunto de especificaciones iniciales a una representación física tal que ambas respondan al mismo comportamiento y la representación física satisfaga las constricciones impuestas por los elementos que utiliza (fan-in, reglas de diseño, etc.) y por el diseñador (costo, velocidad, etc.). Los HDL's cubren tres etapas básicas en el diseño de sistemas digitales:

a) Definición del algoritmo "hardware" a implementar a partir del conjunto de especificaciones iniciales.

b) Sintesis de la arquitectura óptima capaz de implementar el algoritmo "hardware" previamente definido. Se trata de un proceso de diseño desde una descripción a nivel algorítmico a otra al nivel Transferencias de registros.

c) Sintesis Lógica que a partir de la definición del circuito al nivel transferencias de registros proporcione la implementación lógica del mismo.

*Simulación. Es la forma mas simple de verificar la corrección de un diseño a cualquier nivel. Descrito el sistema en el lenguaje del que se trate, se le somete a un conjunto de entradas seleccionado y se analizan las respuestas proporcionadas por el computador. Existen dos métodos principales de simulación. Uno se basa en la utilización de un programa de simulación de próposito general que genera las respuestas del circuito a partir del codigo intermedio obtenido de su descripción HDL y un fichero de estimulos (DDL y CVS_BK). Otra se basa en la utilización de un programa de compilación que a partir del código intermedio genera un ejecutable que simula el funcionamiento del circuito ante cualquier secuencia de entradas (ISPS).

*Simulacion de fallos. La inyección de fallos a nivel funcional presenta la ventaja de que un único fallo puede cubrir multitud de fallos distintos en niveles inferiores siempre que su modelado sea correcto. La simulación de fallos se ha utilizado tanto para determinar coberturas de secuencias de "test" como medio de generación de estas secuencias.

*Verificación del diseño. La simulación, como medio de comprobación del

comportamiento de un sistema digital presenta graves inconvenientes ya que tanto la generación de una secuencia de entradas que cubra todos los posibles fallos de diseño como el análisis de la correspondiente secuencia de salida son tareas extraordinariamente complejas sino imposibles. Por ello, en los últimos años han aparecido técnicas de verificación formal del diseño en las tres fases antes comentadas. Estas técnicas se basan en comparar la descripción del sistema a un nivel con la descripción al nivel superior de la que la anterior se obtuvo. Las técnicas de comparación pueden ser estrictamente matematicas (calculo de predicados, lógica temporal, etc.) o extraidos del análisis de "software" como la ejecución simbólica.

*Evaluación. Los HDL's han sido usados en la evaluación de las prestaciones de distintos sistemas digitales. Así, ISPS se ha empleado como medio de seleccionar computadores para un tipo de aplicación. Determinados unos programas de "test", la evaluación se efectuaba a partir de tres parametros: S≈Número de bytes utilizados para almacenar el programa de "test", M≈Número de bytes transferidos entre el procesador y la memoria principal durante la ejecución del programa de "test' y N≈Número de bytes transferidos entre determinados registros internos del procesador durante la ejecución del programa.

*Testing funcional. La generación de la secuencia de "test" a nivel funcional presenta ventajas similares a la simulación de fallos: se realizan a un nivel en el que los detalles innecesarios de la lógica no se consideran. Esto permite abordar circuitos de complejidad muy superior. Los lenguajes no-procesales permiten extender técnicas bien establecidas como el algoritmo-D. En los lenguajes procesales es necesario tener en cuenta la secuenciación de estados determinada por la Unidad de Control. Técnicas extraidas del "software testing" como la ejecución simbolica han sido también propuestas.

El trabajo desarrollado en la presente Tesis nos ha permitido estudiar y utilizar diferentes HDLs. Esta experiencia de uso nos ha permitido extraer valiosas conclusiones que se han visto reflejadas en la definición del lenguaje de la base de datos de PSAL2; un lenguaje capaz de modelar sistemas descritos en cualquiera de los lenguajes estudiados, dentro del ambito de la síntesis de alto nivel. Además hemos podido evaluar y comparar diversos HDL. Una primera conclusión es que la mayoría de los HDL estudiados estaban orientados al modelado de hardware real, por lo que sus primitivas tenían un significado muy claro en terminos de síntesis. La excepción es el lenguaje VHDL, en el cual existen construcciones orientadas principalmente a simulación, sin un significado claro en síntesis. Por contra, este lenguaje es el mas poderoso de los estudiados, el único capaz de modelar el sistema en

diferentes niveles de abstracción y por ello, el único capaz de expresar tanto el sistema a diseñar como el resultado de la sínesis de alto nivel. La aceptación de un estandar universal multi-nivel puede representar un progreso de gran importancia en un campo caracterizado tradicionalmente por un número elevado de lenguajes. Esta situación, que fué descrita como "Torre de Babel" [COM77], puede ser superada con la aparición de VHDL. Las principales ventajas que se obtendrían con este lenguaje se pueden resumir en los siguientes puntos:

1º Desde el punto de vista del usuario se eliminaría la necesidad de conocer y manejar distintos lenguajes.

2º La utilización de un único lenguaje para describir del sistema a diseñar en los distintos niveles de abstracción por los que este pasa durante el proceso de síntesis permitiría mejorar y dar coherencia tanto a la documentación del proyecto como a la comunicación entre los distintos grupos de ingenieros que trabajan en él.

3º La conexión entre distintas herramientas EDA se podría ver notabemente simplicada por el uso de un único lenguaje estandar.

En cuanto a los HDL utilizados a nivel algorítmico ,VHDL y ISPS, el segundo presenta la ventaja de estar orientado a la simulación y síntesis de sistemas descitos a este nivel, por lo que su sintaxis esta muy especializada y adaptada a dicho nivel de abstracción. Esto no ocurre en VHDL, por lo cual es necesario restringuir el uso del lenguaje al tiempo que se definen un conjunto de normas sobre el significado en síntesis de algunas de sus primitivas. Sin embargo, VHDL dispone de un conjunto de instrucciones de control mucho mas rico que el empleado por ISPS. De hecho, la rigidez en las construcciones de control de ISPS, generan problemas en los algoritmos de síntesis lo cual limita de forma notable los resultados obtenidos por la herramienta. Esta es una de las razones que justifica los mejores resultados de PSAL2 sobre PSAL1. La utilización de VHDL elimina muchas de las restricciones que introducía ISPS.

En cuanto a los lenguajes utilizados a nivel de transferencia de registros (DDL, VHDL, CVS_BK y ESP) debe destacarse que casi todos estan orientados a la sintesis de sistemas digitales secuenciales. En todos los casos el sistema se modela a traves de una máquina de estados finitos, poseyendo primitivas especiales para definir elementos clásicos en este tipo de descripciones, tales como registros, terminales o estados. Una vez mas, la excepción es VHDL, que obliga a limitar y clarificar su sintaxis para poder ser utilizado a este nivel con fines de síntesis del sistema. A cambio, posee las primitivas mas potentes y por ello la mayor capacidad de modelado.

El lenguaje DDL (utilizado como salida por PSAL1), es un lenguaje RT clásico, que ha sido empleado con fines docentes (por ejemplo en el libro "Logic Design of Digital System" de D. Dietmeyer [Di79]) y de investigación. Su principal inconveniente es la asignación de caracteres especiales a las operaciones, lo que hace que las descripciones no tengan una lectura sencilla.

CVS_BK, en cambio, posee una sintaxis y una semántica muy clara, al tiempo que proporciona la posibilidad de disponer de herraminetas de síntesis a otros niveles. Es unos de los lenguajes mas orientados a síntesis de los estudiados. El problema de CVS_BK es la dificultad existente para poder obtener herramientas que los utilicen, dado que fue desarrollado dentro de un programa ESPRIT, y su distribución fuera del proyecto no es facil.

El lenguaje ESP ha sido desarrollado por nosotros dentro de un entorno de diseño automático, por lo que esta muy orientado a síntesis. Además, al haber sido desarrollado en el mismo entorno que PSAL2, dispone de una buena conexión con las herramientas descritas en la presente Tesis.

A continuación serán descritos los diferentes HDL estudiados. El análisis ha sido dividido en tres apartados:

- Lenguaje de descripción de sistemas a nivel algorítmico: ISPS

- Lenguajes de descripción a nivel de transferencias de registros: DDL, CVS_BK y ESP.

- Lenguaje estandar multinivel: VHDL.

II) Lenguaje de descripción de sistemas digitales a nivel algorítmico: ISPS

1º) Introducción

La notación ISP (Instruction Set Procesor) fue introducida por Bell y Newell en su libro "Computer Structures : Reading and Examples " publicado en 1971. Un subconjunto de esta notación fue posteriormente implementada como el lenguaje de procesadores del conjunto de instrucciones, ISPL(Instruction Set Procesor Languaje). Mas tarde ISPL fue reimplementado y aumentado, dando lugar en 1977 al lenguaje de descripción de computadoras ISPS (Instruction Set Procesor Specification) [SiBe82][BaNo80][Ba81]. Este lenguaje nació como resultado del proyecto SMCD (Symbolic Manipulation of Computer Descriptions), financiado por el DoD a los Departamentos de Ciencias de la Computación e Ingeniería Eléctrica de la Universidad de Carnegie-Mellon.

Los diseñadores utilizan notaciones y lenguajes como herramientas para manipular abstracciones. El significado de estas abstracciones esta basado en un conjunto de nociones predefinidas en el dominio del problema que el diseñador trata de resolver. En el proyecto SMCD, los autores de ISPS intentaron diseñar y construir sistemas que operaran en el dominio de las descripciones de computadoras. Ellos necesitaban abstracciones para describir computadoras.

Cuando alguien trata de diseñar un lenguaje orientado a un problema, se le presentan dos opciones. En primer lugar, se puede usar una notación para cada area del problema y, cuando nuevas areas se conviertan en foco de investigación, se desarrollarán nuevos lenguajes. Por otra parte, se puede hacer una conjetura y desarrollar un lenguaje que incorpore cualquier posible abstracción que pueda necesitarse. Es facil ver que ninguna de estas soluciones es satisfactoria. En la primera se presenta el problema de verificar la equivalencia entre dos descripciones realizadas con diferente notación. En la segunda, debido al número y disparidad de posibles areas de aplicación, se tiende a producir lenguajes complicados, dificiles de entender y no-implementables.

ISPS es un lenguaje de kernel, que proporciona al usuario herramientas para definir abstracciones que dependen de la aplicación partiendo de una notación basica. ISPS no interpreta las abstracciones. En el fichero que crea el compilador aparecen los mismos operadores y la misma estructura que en la descripción de partida. De hecho es posible, a partir del fichero intermedio, obtener la descripción fuente. Son los programas de aplicación los que interpretarán las abstracciones.

Aunque ISPS esta orientado hacia la descripción de computadores, contiene un cierto numero de construcciones que pueden ser usadas para describir cualquier otro tipo de sistema digital a nivel algorítmico.

La filosofía de ISPS estaba guiada por dos principios: flexibilidad y simplicidad.
En concreto, era deseable diseñar un lenguaje de descripción de computadores que pudiera ser apropiado para diversas aplicaciones : diseño automático, simulación (tanto de software como de hardware) y generación automatica de software (en particular compiladores de compiladores)[Ba81].

Por lo tanto, aunque ISPS puede ser visto como un lenguaje de programación, el objetivo de la notación es la descripción de computadores y otros sistemas digitales, no necesariamente algoritmos computacionales.

2º) Aplicaciones:

El lenguaje ISPS es analizado por el Parser. Su salida es una descripción en forma de árbol, la cual es utilizada como entrada a distintos programas de aplicación. El esquema de este proceso de compilación aparece reflejado en la figura 2.2.

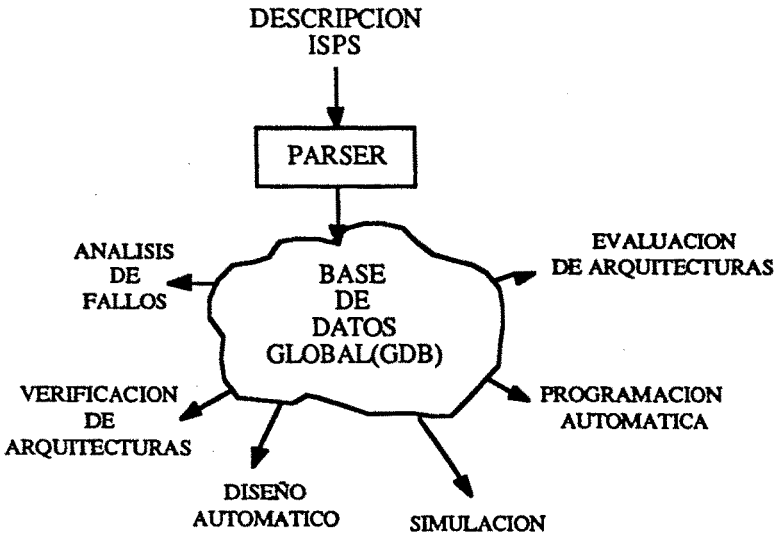


figura 2.2

El fichero que contiene la descripción ISPS tiene la terminación .ISP. El Parser genera como salida, un fichero terminado en .GDB. La descripción contenida en dicho fichero se denomina "Base de Datos Global"(GDB). Esta representación será empleada por los programas de aplicación. Actualmente tenemos diponible uno de estos programas de aplicación : el simulador. Su funcionamiento aparece esquematizado en la figura 2.3.

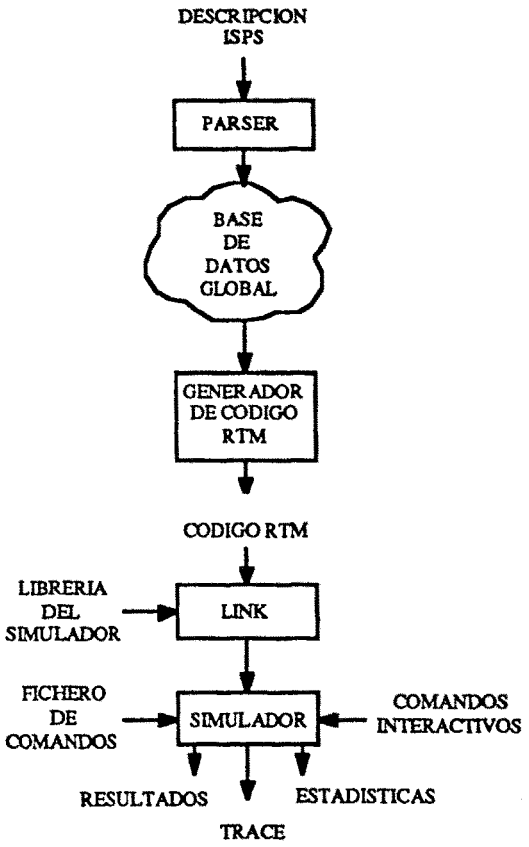


figura 2.3

3º) Utilización de ISPS en PSAL1

En ISPS una entidad es una unidad de descripción funcional o estructural. De hecho toda la descripción es una entidad, en la cual se definen nuevas entidades. Para que PSAL1 pueda compilar la descripción es preciso señalar cual de estas entidades es la primera en ejecutarse, utilizando el atributo "main". En la figura 2.4 puede verse un ejemplo de descripción en ISPS.

```
MARK1 :=
   begin
!      The Manchester University Mark-1 Computer (circa 1948)
!This is the ISPS description  of the first version of the machine, as
!reported in  [Lavington, S.H. "A History  of Manchester  Computers",
!National Computing Centre Publications, Manchester, England, 1975]

** Mp.State **
   M[0:8191]<31:0>,
** Pc.State **
    CR\Control.Register<12:0>,
    ACC\Accumulator<31:0>,
** Instruction.Format **
  PI\Present.Instruction<15:0>,
       f\function<0:2> := PI<15:13>,
       s<0:12> := PI<12:0>,

** Instruction.Execution **
   icycle\instruction.cycle {main} :=
       begin
       REPEAT
          begin
          PI = M[CR]<15:0> next
          DECODE f =>
             begin
             #0 := CR = M[s],
             #1 := CR = CR + M[s],
             #2 := ACC = - M[s],
             #3 := M[s] = ACC,
           #4:#5 := ACC = ACC - M[s],
              #6 := IF ACC lss 0 => CR = CR + 1,
             #7 := STOP()
             end next
          CR = CR + 1
          end
       end
    end
```

figura 2.4

El comportamiento funcional de una entidad viene dado por expresiones que especifican la secuencia de operaciones. Las transformaciones que se producen en una entidad pueden realizarse en serie (una detras de otra) o en paralelo (al mismo tiempo). Dos acciones se desorrollan en serie cuando estan separadas por un "next". Si las acciones estan separadas por ";", dichas acciones se ejecutan en paralelo. En la descripción siguiente aparece un ejemplo:

```
mark1.1 () ; mark1.2 ()
```

Lo que significa que las entidades mark1.1 y mark1.2 deben ejecutarse en paralelo . Este tipo de construcciones pueden presentar problemas durante la síntesis, debido a que es dificil estalecer las dependencias de variables entre los procesos que se ejecutan en paralelo. PSAL1 no tiene ninguna dificultad con este tipo de construcciones siempre que no se utilicen construcciones de control. En este caso PSAL1 producirá un error al tratar de generar la descripción DDL, aunque la mayoria de sus algoritmos lo soportan.

En cuanto a las sentencias condicionales (IF y DECODE), solo se restringe el uso de don't care. Estos valores no pueden utilizarse en PSAL1. En el lenguaje ISPS sirven para indicar que una acción debe realizarse independientemente del valor del bit don't care, por ejemplo :

```
DECODE . . . =>
        begin
  '1????? \... : = ... ( esta acción se realiza cuando primer el bit de la variable de
entrada
  esta a 1 )
            ("alias" )
  '01???? \ ... : =  ...  ( idem, pero cuando empiece los dos primeros bits sean 01 )
     ...
        end
```

Se prohibe igualmente la utilización de la construcción TERMINATE. Esta primitiva aborta una entidad que se esta ejecutando en paralelo con la que la detiene. La razón de esta exclusión radica en la dificultad de determinar en que punto de la entidad abortada debe detenerse la ejecución.

PSAL1 admite todos los tipos de operaciones definidos en ISPS en cualquiera de las representaciones predefinidas (sin signo, con signo, complemento-1 y complemento-2). El sistema introduce, sin embargo, restricciones en los calificativos. Estas instrucciones estan destinadas a los programas de aplicación. Algunos son reconocidos por el Parser y forman parte de la descripción. El resto se introducen en la GDB y estan destinados a ciertos programas de aplicación. Los calificativos que reconoce PSAL1 son:

- PROCESS: Permite que una entidad sea activada sin que la entidad activadora se detenga.

- MAIN: Indica la entidad que debe de ejecutarse en primer lugar.

-Calificadores aritmeticos : Modifica la representación de los números binarios. Por defecto es complemento-dos.

Los calificativos no tolerados son:

- CRITICAL: Cuando se intenta re-ejecutar una entidad con este calificativo la segunda llamada espera a que la primera se complete.

- PTIME: Especifica el tiempo que una entidad tarda en ejecutarse.

- REF: Iguala el parametro formal y el actual de una entidad. Es equivalente a un paso de parámetros por referencia.

- INCREMENT: Permite modificar el tamaño de la palabra de una memoria.

Existen una serie de entidades predefinidas en el lenguaje. Algunas de estas entidades no son aceptadas por PSAL1, como es el caso de:

- Count.one (expresión) <...> : Devuelve el número de unos que hay en la expresión.

- Delay (...) : Detiene la ejecución de la entidad durante un tiempo.

- First.one (expresión) <...> : Retorna el número de ceros que hay antes del primer uno de

la expresión.

- Is.running (entidad) <...> : Devuelve '1 si la entidad esta actualmente activada y '0 si no lo esta.

- Last.one (expresión) <...> : Retorna el número ceros que hay después del último uno de la expresión.

- Mask.left (expresión1 ,expresión2) <...> : Devuelve una palabra del tamaño de expresión1, la cual tiene puestos a cero tantos bits, por la izquierda, como indica expresión2.

- Mask.right (expresión , expresión2) <...> : idem que la anterior, pero por la derecha

- Parity (expresión1) <...> : Retorna '1 si expresión1 tiene un número par de unos .

- Time.wait (expresión1, expresión2) <... > : Detiene la ejecución hasta que expresión1 es distinta de cero, o hasta que haya transcurrido un tiempo mayor que expresión2 .

PSAL1 acepta las siguientes entidades predefinidas:

- Stop () : Termina la activación de todas las entidades.

- No.op () : No hace nada.

- Undefined () <...> : Retorna un número de valor y longitud indefinida, después de un tiempo no concretado. Esta entidad se modela en PSAL1 como una transferencia a un estado "don't care".

- Unpredictable () : Puede pasar cualquier cosa. Esta entidad se modela en PSAL1 como una transferencia a un estado "don't care".

- Wait(expresión) <...>: Detiene la ejecución hasta que la expresión sea diferente de '0.

III) Lenguajes de descripción a nivel de transferencias de registros: DDL, CVS-BK y ESP.

1 º El lenguaje DDL

Como ya se ha comentado, este lenguaje es un típico ejemplo de lenguaje de transferencias de registros en el que es posible encontrar una relación muy estrecha entre la descripción y el circuito que la implementa. Se trata de un lenguaje orientado a la descripción modular del sistema. Permite combinar estamentos de tipo procesal con otros de tipo no-procesal. Estos últimos adquieren la forma de maquinas de estados finitos.

DDL fue el lenguaje de descripción a nivel de transferencia de registros utilizado en el sistema PSAL1. A continuación, se realizará un breve análisis sobre el estilo de diseño que utiliza para describir la arquitectura resultante del proceso de síntesis.

La descripción se inicia con la primitiva <SY> seguida del nombre del sistema a diseñar. Esta declaracion abre la descripción del sistema completo y portanto define el bloque que engloba al resto de los bloques de que consta el sistema. A continuación se pueden definir condiciones que producen la detención del sistema. PSAL1 utiliza esta posibilidad para modelar el comando STOP. Esta primitiva es modelada mediante la introducción automática de un registro (_STOP), el cual es cargado con el valor 0 cuando el sistema tiene que detenerse, como se ve en el ejemplo:

```
<SY> pdp8: _STOP:
        ...
        <RE> ...
                _STOP[1],...
        ...
        <ST>
        ...
                S24: ... _STOP <- 0D1,...
```

Una vez especificado el nombre del sistema se declaran los registros con la primitiva <RE>, los terminales con <TE> y las memorias con <ME>. PSAL1 añade ademas la definición de

un reloj (utilizando la primitiva <TI>), necesario en la definición de la máquina de estados finitos, como se ve en el ejemplo:

```
<SY> edc:
<RE> ACC[16],
          ...
<TE> INPUT[16], ...
          _M[10].
<ME> M[1024:16].

<TI> clk_edc.
<AU> CPU_edc: clk_edc:
```

El sistema PSAL1 añade un terminal por memoria, para modelar el bus de direcciones. El terminal _M es el único objeto que se puede utilizar como dirección de la memoria M. Por lo tanto solo se utilizan la expresiones:

```
... <-M[_M]
```

para modelar la lectura en memoria y :

```
M[_M] <- ...
```

para modelar la escritura en dicho elemento.

Como se puede observar, los elementos que define PSAL1 empiezan por '_', con objeto de que sean facilmente diferenciados del resto.

Para describir el resultado del proceso de síntesis en forma de máquina de esados finitos, PSAL1 utiliza la primitiva AUtomaton (<AU>). Para definir un automata, deben señalarse un registro de estados y un reloj. Ambos elementos son introducidos por PSAL1 de forma automática. De esta forma en el ejemplo anterior el "registro de estados" es "CPU_edc", y el reloj, "clk_edc". Una vez han sido especificados se define la lista de estados que forma el automáta. Dicha lista se inicia con el comando <ST>. Un ejemplo de autómata descrito a partir de los estados lo tenemos en el contador módulo-6 siguiente :

```
<SY>  COUNT :
      <AU>  CON :
            <RE>  MQ[2], QC.
            <TE>  CL, ABC[3].
            ABC = MQ ' QC, | MQ[2] | QC <- ~QC..
                  <AU>  MQ[2] : CL :
                        <ST>  S1[00] : MQ <- 2D2.
                              S2[10] : MQ <- 1B2.
                              S3[01] : MQ <- 0B2.....
```

En cada estado se reflejan las acciones u operaciones que la máquina realiza cuando se encuentra en él. Para representar las operaciones PSAL1 utiliza tres estrategias. Las operaciones asociadas al control y , por lo tanto, no localizables en una unidad operacional, son representadas utilizando el conjunto de operadores de DDL. En DDL las expresiones son de tipo lógico, no existen por lo tanto operaciones aritméticas ni relacionales. En la tabla siguiente se muestran los operadores lógicos aceptados:

Operador	Símbolo	Sintaxis	Función que realiza (asumiendo A y B de dimensión n)
Concatenación	'	A ' B	construye un conjunto con todos los elementos de A y B
Complementación	~	~A	complemento bit a bit de A
Reducción	/	ø / A	A[1]øA[2]ø...øA[n] (donde ø es un operador AND u OR)
AND	*	A * B	producto lógico bit-a-bit*
NAND	~*	A ~* B	complemento del producto lógico bit-a-bit*
OR	+	A + B	suma lógica bit-a-bit*
NOR	~+	A ~+ B	complemento de la suma lógica bit-a-bit*
OR-exclusiva	@	A @ B	or-exclusiva bit-a-bit*
NOR-exclusiva	~@	A ~@ B	equivalencia bit-a-bit*

*Si en cualquiera de estas operaciones uno de los operandos es un escalar, la operación se realiza entre el escalar y cada uno de los bits del vector. Si ambos operandos son vectores deben de tener la misma dimensión.

Dentro de las operaciones no asignadas a unidades operacionales existe un subconjunto formado por operaciones que se realizan en el mismo registro sobre el que actuan, como por ejemplo la incrementación o decrementación. Estas operaciones son modeladas mediante operaciones especiales. En DDL existe una primitiva, <OP> ("operator") que proporciona

un medio de definir bloques lógicos combinacionales dentro del sistema. En la descripción generada por PSAL1 aparecen este tipo de elementos, pero no su definición, ya que esta se encuentra en otros ficheros (los terminados en ."ous"). En el siguiente ejemplo se puede observar la utilización de dos operaciones definidas por PSAL1 (DEC e INC) para modelar las operaciones de incremento y decremento:

```
<SY> system :
...
<ST>
        State_i: ACC <- INC(ACC) ...
        ...
        State_j: ACC <- DEC(ACC) ...
```

La operaciones localizables en OU se modelan igualmente mediante "Operator". Estas reciben como argumentos, no solo las operandos sino tambien el código de la operación que deben realizar. PSAL1 no asigna los códigos a las operaciones, por lo que introduce un símbolo que indica la operación, como se ve en el ejemplo:

```
ACC <- OU0(ACC,IR, <<< + >>>) ....
```

El símbolo "<<< + >>>" tendrá que ser sustituido por el código asignado a la operación '+' en la unidad operacional 1. En el nombre de la operación aparece una cifra que indica a que OU se refiere dicha operación.

Para modelar operaciones que esperan se verifique una condición para continuar la ejecución de la FSM (equivalente al "wait" en ISPS), PSAL1 introduce condiciones en la ejecución de un estado concreto. En el ejemplo puede observarse como hasta que el registro IN no tome el valor 1, la ejecución de la FSM estará detenida en el estado S2:

```
<SY> pdp8 : _STOP:
...
<ST>
...
        S2: IN: ...
```

La estructura: " If...Then...Else...Endif ", se describe en DDL con la sintaxis:

Be = | Be_1 | Be_2 ; Be_3 .

Si la condición Be_1 se cumple ("|" = "If") la fuente será Be_2 ("|" = "Then"); sino, será Be_3 (";" = "Else" y "." = "Endif"). Puede que no exista la operación "else", con lo cual no aparecerá el ";".

Otro tipo de estructura condicional es "case" o "if-value". La sintaxis es en este caso:

! Be #va_i coc_i .

donde cada va_i es un entero decimal positivo ó un rango de éstos. Si Be toma el valor (o los valores) indicados por va_i se realizará el correspondiente conjunto de operaciones coc_i. Así, por ejemplo:

```
<TE>   A, B, C, D[0:7], W, Y, STROBE.
<ID>   ABC = C ' B ' A.
<BO>   Y = | ~STROBE |
           ! ABC #0 D[0] #1 D[1] #2 D[2] #3 D[3] #4 D[4]
                #5 D[5] #6 D[6] #7 D[7],W = ~Y.
```

2º El lenguaje CVS-BK

El lenguaje de descripción de hardware CVS-BK fue desarrollado en el proyecto ESPRIT, CVS. Este lenguaje nace como una evolución de los lenguajes KARL III, IRENE y KARENE. Muchos de sus conceptos hardware y características fueron tomados de sus antecesores. Desde un punto de vista gramatical, CVS-BK puede ser considerado como una extensión de KARL III. Esta es la razón de su nombre: CVS-BK (CVS Behavional KARL).

Una de las razónes de introducir en PSAL2 el módulo, GEN_CVS_BK, encargado de expresar el resultado de la síntesis en CVS-BK, fue el poder conectar PSAL2, con las herramientas de síntesis desarrolladas en el proyecto CVS, tales como BATCH. GEN_CVS_BK iba a formar parte de un paquete que permitiría a PSAL2 disponer de lenguajes homogeneos como entrada (KARL IV) y salida (CVS-BK) del sistema. El pobre estado de desarrollo de KARL IV hizo que dicho paquete este unicamente compuesto por GEN_CVS_BK. Dicho módulo fue desarrollado durante una estancia en la Universidad de Kaiserslautern (Alemania), en la cual fueron desarrollados los lenguajes de la familia KARL (KARL III, KARENE, CVS-BK, etc.).

CVS-BK permite utilizar dos estilos para describir el sistema digital: estructural y de comportamiento. Ambos estilos se pueden observar en la descripción del contador up/down que aparece a continuación.

Descripción estructural:

```
CELL  up_down_counter
   (        RIGHT IN  fct [1..0]                    {function selector}
            FRONT IN  preset [15..0]                {preset value input}
            RIGHT IN  clk                           {clock input}
            BACK OUT  count [15..0] ) ;             {count value output}

CONSTANT zero [15..0] .= 0 ;                        {reset value}
REGISTER count [15..0] ;                            {count register}

BEGIN
  AT clk DO
    count := CASE fct OF
                                0 : inc ( count )
                                1 : dec ( count )
                                2 : zero
                                3 : preset
                ENDCASE ;
  ENDAT ;
END.
```

Descripción del comportamiento:

```
MODULE up_down_counter
   (        RIGHT IN  fct [1..0]                    {function selector}
            FRONT IN  preset [15..0]                {preset value input}
            BACK OUT  count [15..0] ) ;             {count value output}

CONSTANT zero [15..0] .= 0 ;                        {reset value}
REGISTER count [15..0] ;                            {count register}
STATE upc_states = (init, wait, up, down);          {module states}

BEGIN
  WHENEVER fct [1] GOTO init;                       {global condition}

  SEQUENCE                                          {start of states}
    init : IF fct [0]                               {init states}
         THEN count := preset                       {fct = '11 -> load preset}
         ELSE count := zero                         {fct = '10 -> load 0}
         ENDIF ;
         goto wait
    wait : CASE fct OF
             0 : GOTO up
             1 : GOTO down
         ELSE GOTO wait
         ENDCASE
    up :   count := inc ( count ) ;                 {increment count}
         GOTO up                                    {stay in state up}
    down : count := dec ( count ) ;                 {decrement count}
         GOTO down                                  {stay in state down}
END
```

La descripción estructural es similar a la descripción de tipo flujo de datos en VHDL. Los puertos de la celda son descritos en la definición de la CELL (celda). En los puertos no solo se indica la dirección del puerto, sino también su posición en los márgenes de la celda (información muy útil para "componer" celdas, poniendo en contacto sus márgenes). Todas las variables utilizadas en la descripción representan arrays de bits (bit-vector en VHDL). Las variables pueden ser de varios tipos:

- Constantes.
- Registros.
- RAM.
- ROM.
- BUS.
- Node, lightnode: concepto similar a los terminales en DDL.
- Channel. Estas variables permiten comunicar procesos concurrentes.
- State.

PSAL2 utiliza variables de tipo registro, RAM (para modelar memorias), ROM (para modelar constantes), BUS, NODE (para modelar terminales) y STATE (para modelar estados).

En la descripción estructural, la sentencia AT permite señalar cuando los registros serán cargados, mientras el CASE permite escoger el valor a asignar en función de una condicion.

La descripción de tipo comportamiento permite modelar FSM de una forma muy sencilla, razón por la cual fue elegida en el módulo GEN-CVS-BK.

En el ejemplo de la página siguiente puede observarse como PSAL2 describe el resultado de la síntesis de un sistema que implementa la ecuación:

$$r = \sum_{i=0}^{255} (x[i] - y[i])$$

La descripción se inicia con la definición de un módulo (ejemplo1). Un "module" es similar a una "cell". la diferencia es que una celda contiene información estructural (descripción no-procedural) mientras que un módulo describe un comportamiento (descripción procedural). En la definición del módulo, se señalan, no solo el nombre del sistema, sino también la dirección y nombre de los puertos. En el ejemplo, se define un terminal de entrada, denominado "$psal_reset_terminal".

```
module ejemplo1 (
    in
        $psal_reset_terminal      );
     clock $psal_clock
         default;
    ram
          mem_x[255..0;15..0];
          mem_y[255..0;15..0];
    node
         t1[15..0];
         t3[31..0];
         t2[15..0];
    register
         $psal_reset;
         r[31..0];
         j[7..0];
         address_x[7..0];
         address_y[7..0];
    constant
          $psal_const_11[7..0].=1;
          $psal_const_15[7..0].=1;
          $psal_const_12[31..0].=0;
    state
         PSAL_ejemplo_state_list = (
              PSAL_ejemplo_state_1,
              PSAL_ejemplo_state_2,
              PSAL_ejemplo_state_3  );

    begin

     whenever ( $psal_reset or $psal_reset_terminal ) goto PSAL_ejemplo_state_1;
    sequence
    PSAL_ejemplo_state_1:
           if ($psal_reset and $psal_reset_terminal ) then $psal_reset:='0 endif;
         r:=$psal_const_12;
         j:=$psal_const_11;
         goto PSAL_ejemplo_state_2
    PSAL_ejemplo_state_2:
        if not(equ(j))  then
        address_x:= j;
        address_y:= j;
          goto PSAL_ejemplo_state_3
        else
          $psal_reset:='1;goto PSAL_ejemplo_state_1
        endif
    PSAL_ejemplo_state_3:
         t1.=mem_x[ address_x] { t1 <- dat_bus_x } ;
         t2.=mem_y[ address_y] { t2 <- dat_bus_y } ;
           t3.= t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]
. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1[15]. t1 -  t2[15]. t2[15]. t2[15]. t2[15]. t2[15]. t2[15].
t2[15]. t2[15]. t2[15]. t2[15]. t2[15]. t2[15]. t2[15]. t2[15]. t2[15]. t2;
         r:= t3 + r;
          j:= j + $psal_const_15;
          goto PSAL_ejemplo_state_2
end.
```

Ejemplo de descripción resultante del proceso de síntesis en CVS_BK

A continuación aparece el cuerpo del módulo. En él se pueden diferenciar dos partes:

- Parte de declaraciones.
- Parte intena del bloque.

En la primera se definen los objetos que serán utilizados en la descripción. PSAL2 utiliza objetos de tipo"NODE" para modelar terminales y "REGISTER" para los registros. Las memorias son definidas utilizando el objeto "RAM". En esta definición, el primer rango, muestra las palabras que tiene la memoria, y el segundo el rango de bits de cada palabra. Además se definen constantes, estados y el reloj.

En una definición CVS-BK, el reloj debe ser explícitamente definido. La razón es que dicho elemento es usado para determinar cuando se actualizan las variables de estado. Dado que esta información no aparece en la descripción algorítmica, PSAL2 introduce un reloj ("$psal_clock"). Este reloj valdrá "1" durante la transición de estados y "0" el resto del tiempo (opción "default").

En cuanto a la declaración de estados debe diferenciarse entre una declaración general y una declaración fija. Una declaración fija es utilizada para "refinar" una declaración general, esto es, para definir subestados de un estado. Estos subestados deben de estar asociados a un determinado reloj, que señala cuando deben realizarse las transiciones. El módulo GEN_CVS_BK no hace uso de esta posibilidad, utilizando una definición general de estados, en la cual se indica el nombre de la lista de estados (a la cual se denomina : PSAL_"nombre del sistema"_state_list), así como los estados que forman parte de dicha lista. En el ejemplo la descripción posee 3 estados. PSAL2 utiliza la siguiente regla para nombrar los estados:

PSAL_"nombre del sistema"_state_"nº del estado"

CVS-BK es un lenguaje especialmente orientado a la síntesis de sistemas secuenciales descritos en forma de máquina de estados finitos. Por esta razón, la vista interna del módulo (contenida entre las primitivas BEGIN y END) está formada por dos componentes principales:

- Parte del "Whenever".
- Parte de instrucciones.

En la primera se señalan las condiciones que producen una reinicialización de la FSM, de forma independiente del estado. La segunda, formada por la secuencia de estados, describe

el funcionamiento del sistema en cada estado. CVS-BK permite, en la vista interna del módulo, introducir información estuctural (mediante el comando MAKE) y jerarquia (mediante el uso de subprogramas definidos como rutinas, ROUTINE). PSAL2 no hace uso de estas 2 últimas posibilidades.

La parte del "Whenever" es definida automáticamente por PSAL2. Para ello utiliza el siguiente modelo de máquina secuencial: el sistema define un registro ("$psal_reset") y un terminal ("$psal_reset_terminal") que permitirán detener las ejecucion de la FSM. El registro "$psal_reset" será cargado con un valor alto siempre que sea necesario detener el sistema. De esta forma, una instrucción "STOP" en la base de datos de PSAL2 es interpretada por GEN_CVS_BK, como la carga de un valor alto en el registro de "reset". Al ser utilizado dicho elemento en la parte del "Whenever", la carga anteriormente reseñada producirá un disparo en la condición de reset deteniéndose la FSM hasta que se produzca un cambio de alto a bajo en el terminal "$psal_reset_terminal". De la misma forma, un cambio del nivel bajo al alto en dicho terminal provocará la detención del sistema. Se dispone por tanto, de un sistema para detener la ejecución de la FSM, con la ventaja de que después de dicha interrupción la máquina continua ejecutando un estado conocido (el primer estado de la descripción), lo que simplifica el test del circuito final, al tiempo que elimina lógica redundante, ya que la condición de reset, que es común a todos los estados, está claramente definida y puede ser fácilmente reconocida por la herramienta de síntesis lógica. La estructura del "Whenever" es:

```
WHENEVER reset GOTO init_state ;
```

En ella se indica el estado al cual se debe de ir en caso de que se produzca una condición de reset.

A continuación aparece la parte de instrucciones. Dicha parte es introducida por la palabra clave "SEQUENCE", y está formada por la especificación de las acciones a realizar en cada estado. Cada grupo de acciones está marcado con una especificación del estado en el cual se realizan:

```
SEQUENCE
    PSAL_ejemplo_state_1:
                ...
                acciones a realizar en el estado PSAL_ejemplo_sate_1
                ...
    PSAL_ejemplo_state_2:
                ...
```

En cada una de dichas especificaciones se señalan acciones que deben de ejecutarse en paralelo. Pueden aparecer diversas instrucciones como:

1) Instrucciones de asignación. existen dos tipos de asignaciones en CVS_BK:
   - Asignaciones a terminales (:=).
   - Asignaciones a registros (:=).

2) Instrucciones de tipo IF.
   Estas expresiones tienen el formato:

```
IF "expresión" THEN
    acciones
  [ ELSE acciones ]
ENDIF.
```

   La clausula ELSE es opcional.

3) Instrucciones de tipo CASE.
   Las cuales tienen el formato:

```
CASE "expresión" OF
    ...
  lista_de_valores:
            lista de acciones
    ...
  [ ELSE acciones ]
ENDCASE.
```

   La clausula opcional ELSE, permite reseñar las acciones que se ejecutan en el caso de que la "expresión" no coincida con ninguna "lista_de_valores".

4) Instrucciones GOTO.
   Esta primitiva permite señalar cual será el próximo estado de la FSM.

CVS-BK posee un rico conjunto de operaciones, por lo cual PSAL2 no necesita especificar en que unidad operacional se realizan las operaciones (esta tarea es realizada por BATCH). Tampoco es necesario especificar la conectividad entre los distintos módulos, por lo que PSAL2 no señala cuales son las interconexiones definidas en el sistema y que elementos conectan.

Por otro lado existen ciertas operaciones en PSAL2 que carecen de equivalente en CVS-BK, tales como operaciones de desplazamiento, incrementación, decrementación y complementación. En estos casos PSAL2 especifica una función, cuya definición no se

encuentra en este módulo, sino en una librería adicional, que permite modelar la nueva instrucción.

En cuanto a la semántica, en CVS-BK es necesario que los operandos de una asignación tengan la misma dimensión, el mismo número de bits. Por ello PSAL2, extiende el operando de la derecha para que su dimensión coincida con el de la izquierda utilizando el operador de concatenación ("."). En el ejemplo, y dado que las operaciones se realizan en complemento-2, el bit más significativo del operando de la derecha es repetido hasta que se obtiene un elemento con igual dimensión que el operando de la izquierda.

3º El lenguaje ESP

1) Introdución

El lenguaje ESP ha sido desarrollado por el Grupo de Microelectrónica de la Universidad de Cantabria, como lenguaje de entrada de un compilador de silicio que genera el layout de sistemas digitales a partir de su descripción a nivel de transferencia de registros [RaSa91]. Este compilador toma directamente la descripcion a nivel RT generada por la herramienta de síntesis de alto nivel (PSAL2) y la convierte en una descripción de circuito en EDIF que sirve como entrada para que el sistema EDGE de CADENCE genere el layout detallado del mismo. Para ello utiliza algunos programas de las OCTTOOLS [Sp89] como herramientas de sintesis y optimización del circuito generado.

El diseño del layout puede realizarse utilizando librerias de "standard cells" celdas parametrizadas, PLA's, "gate array" o modulos predefinidos dependiendo de la estrategia utilizada en el compilador.

2) Descripción del entorno de diseño

El compilador se planteó como un nexo de union entre un sistema de sintesis de alto nivel que utiliza descripciones de circuitos escritos en VHDL, (PSAL2) y un CAD-Framework que puede generar el layout de los circuitos a partir de descripciones en EDIF como es el sistema EDGE de CADENCE. Ademas, el sistema hace uso de programas de los OCTTOOLS desarrolladas por la Universidad de California en Berkeley como herramientas para algunos pasos de sintesis y optimización del circuito.

Las OCTTOOLS [Spi89] son una colección de más de 90 programas y librerías, sin embargo es de hacer notar que aunque individualmente algunos de estos programas son herramientas muy poderosas (como misII [BrRu87] y Mustang [DeHi88]), el conjunto en sí no puede definirse como un sistema CAD, ya que falta un "entorno" de trabajo que permita manejar los diferentes programas en forma sencilla para el usuario.

3) El lenguaje ESP

El lenguaje de entrada al compilador es un superset de BDSYN, (el lenguaje de descripción de harware de las OCTTOOLS) desarrollado por el Gupo de Microelectrónica de la Universidad de Cantabria con el objeto de permitir describir circuitos a nivel RT .

Como dijimos anteriormente BDSYN solo acepta descripciones de circuito combinacionales, mientras que los circuitos descritos a nivel RT son generalmente circuitos secuenciales, con elementos tales como registros, buses y memorias, que no pueden representarse en BDSYN. Esta deficiencia se puede subsanar si se considera un circuito secuencial como la unión de un circuito combinacional y un grupo de registros (parte secuencial), y que parte de las entradas del circuito combinacional provienen de los registros y parte de sus salidas son las entradas de los mismos.(figura 2.5).

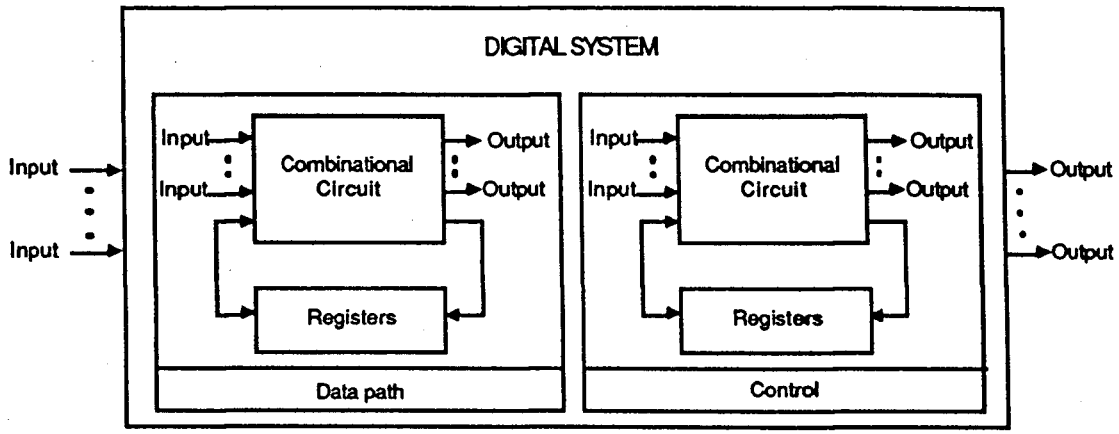


figura 2.5

Una primera parte del programa ESP se encarga de generar el código, en lenguaje BDSYN, que permite el manejo directo de registros, memorias y buses, esto es, las definiciones de todas las señales de entrada, salida y control que corresponde a estos elementos, asi como las secuencias de eventos necesarios para leerlos o cargarlos, dando lugar a una descripción

en lenguaje BDNET de las interconecciones entre el circuito combinacional y los registros, memorias y buses.

Para realizar esto, el lenguaje ESP, además de todas las instruciones del lenguaje BDSYN, incluye primitivas para definir y manejar los elementos antes citados en forma automatica. Esto permite al diseñador describir el funcionamiento del sistema completo usando las nuevas instrucciones para la lectura y escritura en los registros, memorias y acceso a buses sin preocuparse de generar la logica combinacional que ello requiere. De esto ultimo se encarga el preprocesador.

La descripción generada por PSAL2 esta formada por varios módulos. El principal contiene la descripción del sistema en forma de tabla de estados. El resto de los módulos modela las unidades operacionales.

El sistema resultante del proceso de síntesis presenta un comportamiento especial cuando se encuentra en modo "reset". Este modo es introducido automáticamente por PSAL2 y permite llevar al sistema a un estado definido mediante una señal externa. En la descripción ESP existe una sentencia "if" que determina el modo en el cual se encuentra el sistema. El efecto de esta construcción es similar al que produce la sentencia "whenever" en el lengaje CVS-BK. El modelo de reinicialización introducido por PSAL2 ya fue comentado al hablar de la sentencia "whenever" (en CS_BK) por lo que no será repetido aquí.

Cuando el sistema no se encuentra en estado de reinicialización su comportamiento es descrito mediante una tabla de estados. En ella se señalan las acciones a tomar asi como cual será el próximo estado a ejecutar. Dado que las unidades operacionales son implementadas en módulos aparte, cada vez que se precise utilizar una de ellas, se especificará en la descripción del estado la transferencia de los operandos al módulo en el cual se realiza la operación, asi como el elemento en el cual debe de cargarse el resultado obtenido. Además, de los operandos debe ser especificado el código de la operación.

Aunque el lenguaje ESP dispone de subrutinas, PSAL2 no modela las entidades como subrutinas del lenguaje sino utilizando la estrategia comentada en el lenguaje DDL. La razón de no utilizar la primitiva del lenguaje radica en que éstas estan orientadas al modelado de lógica combinacional.

4) Organización del sistema

Un esquema de la organización del sistema completo puede verse en la figura 2.6.

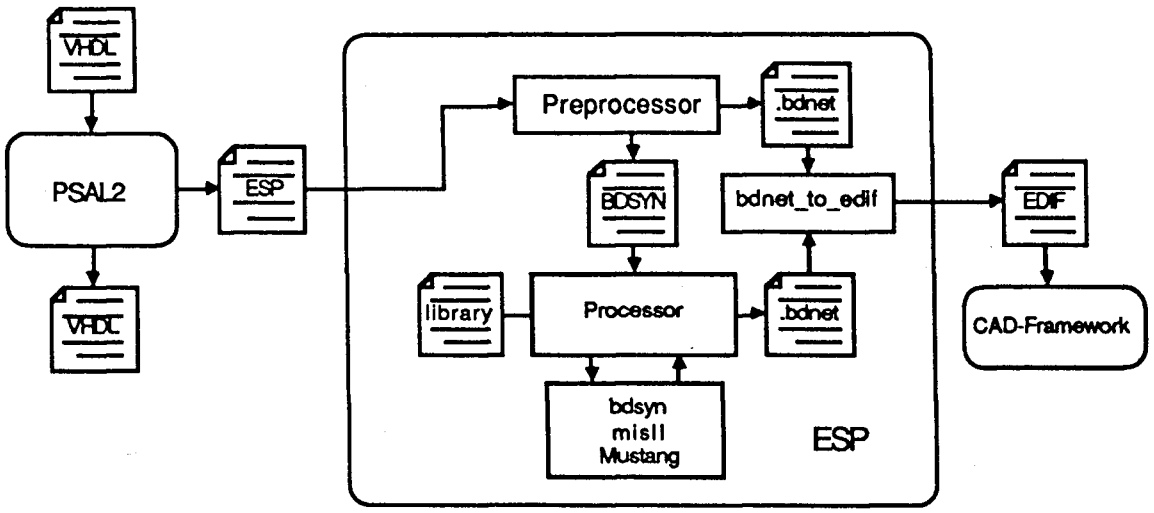


Figura 2.6

Una descripción a nivel comportamiento del circuito a implementar descrito en VHDL es procesado por PSAL2, obtiendose una descripcion a nivel RT en lenguaje ESP.

La descripción en lenguaje ESP es preprocesada por el programa para generar la parte combinacional del mismo, escrita en BDSYN, que será sintetizada usando los programas Bdsyn [Se89] y MisII [BrRu87]. Simultaneamente se genera una descripcion de la parte secuencial, que incluye los registros, los buses y las memorias escrito en BDNET.

Si se utiliza un control mediante maquina de estados finitos (FSM) se obtiene la tabla de transiciones de la misma para poder realizar una optimización en la asignación de estados utilizando Mustang [DeHi88].

Los resultados obtenidos, unidos para generar una descripción del circuito completa en formato BDNET mapeado sobre una libreria definida por el usuario es traducida a EDIF y utilizada por EDGE para generar el "layout" definitivo. El circuito asi generado puede ser simulado, dentro del entorno EDGE, para compararlo con la descripción original dada a PSAL2.

IV) Lenguaje estandar multinivel: VHDL.

1) Introducción

A raíz del programa "Very High Speed Integrated Circuits" (VHSIC) puesto en marcha por el Departamento de Defensa de los EEUU con objeto de reducir el tiempo de diseño de los circuitos integrados y permitir la introducción efectiva de la tecnología VHSIC en los sistemas militares, surge la necesidad de un medio estandard de comunicación y documentación de la masiva cantidad de datos asociados al diseño de los dispositivos de la escala y complejidad deseadas.

Los requerimientos a ser exigidos al nuevo lenguaje de descripción de hardware fueron analizados durante 1981; la organización del programa fue formulada durante 1982 y el programa "VHSIC Hardware Description Language" (VHDL) dio comienzo en 1983. Tras varias versiones intermedias, revisadas extensivamente, tanto por el gobierno de los EEUU, como por las industrias y Universidades involucradas, y que dieron lugar a la versión 7.2 del lenguaje, el IEEE publicó el estandard IEEE Std 1076-1987.

El entorno de diseño VHDL tiene la estructura mostrada en la figura 2.7.

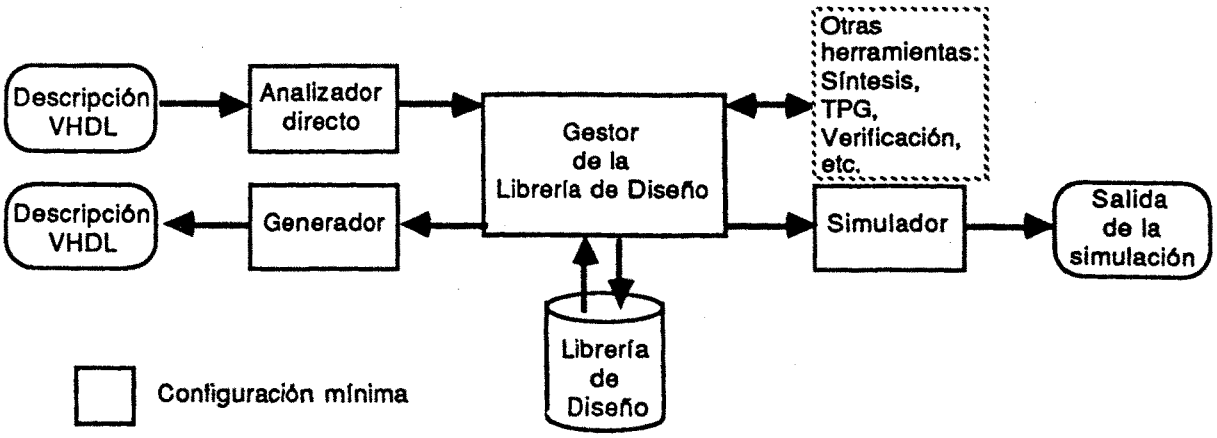


figura 2.7

Libreria de diseño. El lenguaje VHDL se ha diseñado, como tendremos ocasión de comprobar mas adelante, con objeto de satisfacer los siguientes requerimientos:

a) Permitir la descripción del circuito digital desde el nivel de sistema hasta su implementación lógica.

b) Permitir la combinación de descripciones de tipo estructural con otras de comportamiento.

c) Soportar el diseño "bottom-up" y "top-down" como un proceso en el que un equipo de diseñadores interacciona sobre un mismo conjunto de datos.

d) Permitir la descripción modular y jerárquica del sistema.

Todo ello hace que una determinada entidad (la abstracción hardware principal en VHDL), pueda contener otras entidades y que todas ellas puedan ser descritas y utilizadas de distintas formas y por distintos diseñadores a medida que el proceso de diseño evoluciona.

La libreria de diseño VHDL es una estructura de datos única compartida por todos los usuarios del entorno de diseño en la que se almacenan ordenadamente las unidades de diseño VHDL una vez analizadas. El resultado del análisis es una representación de la unidad de diseño en una notación intermedia.

Todas las operaciones propias de la manipulación y mantenimiento de la libreria de diseño están encomendadas a un gestor. Este programa es compartido por el resto de las herramientas con objeto de acceder a los ficheros en la librería.

Análisis. El analizador acepta la descripción VHDL y la chequea con objeto de detectar posibles errores sintácticos y errores semánticos estáticos (aquellos que son evidentes sin necesidad de proceder a la simulación). Caso de que no existan errores, produce como salida el fichero en la libreria de diseño.

Generación. El generador es el encargado de recuperar la descripción VHDL original de una unidad de diseño a partir de la notación intermedia de la unidad de librería correspondiente. Permite, por tanto, que las herramientas actuen directamente sobre la libreria de diseño y, posteriormente, generar la descripción VHDL correspondiente.

Simulación. Es la forma mas simple de verificar la corrección de un diseño a cualquier nivel. Descrito el sistema en el lenguaje del que se trate, se le somete a un conjunto de entradas seleccionado y se analizan las respuestas proporcionadas por el computador.

Síntesis. El lenguaje VHDL ha sido utilizado en el sistema PSAL2 como medio para describir el sistema digital tanto a nivel algorítmico como a nivel de transferencia de registros. En la implementación actual, el programa utiliza un módulo que permite transformar la descripción VHDL, que describe el sistema a diseñar, en una descripción en

formato PSAL. Esta última representación puede ya ser compilada y utilizada por el sistema PSAL2. El modulo VHDL2PSAL transforma una descripción VHDL en una representación PSAL, aceptando un subconjunto de VHDL adaptado a la descripción de sistemas a nivel algorítmico que será definido en el proximo apartado. El módulo VHDL2PSAL realiza la transformación directamente, sin generar la libreria de diseño, lo cual obliga a que el sistema a diseñar sea descrito en una única entidad.

Una de las principales ventajas de VHDL es su capacidad para describir sistemas con diferentes niveles de abstracción. Ello permite utilizar VHDL para describir, no solo el algoritmo de entrada al sistema de síntesis, sino tambien la arquitectura resultante. El módulo GEN_VHDL, encargado de esta tarea, realiza una descripción del sistema en VHDL a partir de los datos contenidos en la base de datos de PSAL2. En la actualidad estamos trabajando en un nuevo módulo que permita introducir estos datos en la libreria de diseño DLS, de CLSI, siendo el generador de código de dicha libreria el encargado de generar el nuevo código.

2) Utilización de VHDL en el modelado de sistemas a nivel algorítmico

VHDL permite el modelado de sistemas digitales utilizando estilos de descripción estructural, flujo de datos y comportamiento. En síntesis de alto nivel una descripción a nivel algorítmico del sistema a diseñar es utilizada como entrada a la herramienta de síntesis, la cual produce una representación estructural como resultado del proceso de síntesis.

Por ello, las primitivas que en VHDL modelan comportamiento, pueden ser utilizadas para describir inicialmente el sistema a diseñar [CaTa89]. Esta descripción [SaVi90] estaría formada por primitivas de tipo "process" conteniendo sentencias secuenciales con una estructura similar a la que poseen lenguajes secuenciales tales como C, PASCAL, ISPS, etc. El uso de un lenguaje estandar en síntesis de alto nivel permite utilizar metodologías de síntesis a nivel estructural y lógico para implementar la descripción resultante del proceso de síntesis de alto nivel, lo cual es un paso importante hacia sistemas de síntesis completos, comenzando en el nivel algorítmico y terminando en la layout final del sistema [Me89a] [Le89][Ma89][NaSa86]. En estas metodologías la comunicación entre los distintos módulos se realizaria utilizando el lenguaje VHDL. Estos entornos de síntesis podrían, igualmente, hacer uso de las librerías VHDL con las que diversas compañías modelarán sus productos, lo cual permitirá alcanzar uno de los objetivos de la síntesis de alto nivel: la independencia tecnológica (las celdas requeridas para implementar el sistema serán escogidas de la librería seleccionada por el usuario). Otra ventaja del uso de VHDL en síntesis de alto nivel, es la posibilidad de comparar resultados entre distinas herramientas, evaluando las distinas metodologías y estilos de diseño.

Como ya fue anteriormente comentado, un sistema de síntesis de alto nivel podría usar una descripción funcional como entrada, utilizando las sentencias secuenciales de un proceso. Esta filosofía permitiría modelar la jerarquía del diseño utilizando subprogramas que a su vez llaman a otros subprogramas.

Es posible describir el sistema utilizando varios procesos, siempre que las dependencias establecidas entre objetos de diferentes procesos puedan ser claramente definidas por la herramienta de síntesis, con objeto de garantizar la equivalencia funcional entre la descripción inicial y la final [SaVi90].

Una de las posibilidades más interesantes que ofrece VHDL es la de modelar con diferente nivel de detalle, diferentes partes del mismo sistema en una misma arquitectura. Por ello, el usuario podría describir algunos módulos a nivel algorítmico y el resto con menor nivel de detalle. En estas situaciones, la herramienta de síntesis generará la estructura que implemente el comportamiento descrito a nivel algorítmico, mientras los módulos restantes pasarán directamente de la representación inicial a la final.

VHDL fué desarrollado inicialmente como un lenguaje orientado a simulación [Co88][ML89] [NuMe89] poseyendo algunas primitivas dificilmente implementables con hardware actual [Li88] [St89]. Para poder sintetizar un sistema descrito en VHDL, sería deseable restringir el uso de estos elementos [Me89b] [Bh88] [We90]. Ejemplos de primitivas del tipo anteriormente mencionado lo constituyen los ficheros I/O, la localización dinámica y los programas recursivos. Otras construcciones no poseen un significado claro en síntesis, como por ejemplo, las primitivas utilizadas en VHDL para modelar retrasos. Es muy difícil para una herramienta de síntesis determinar si el sistema resultante del proceso de síntesis es funcionalmente equivalente al descrito inicialmente, si los retrasos son mayores o menores que los especificados en la descripción algorítmica. Dado que es muy difícil generar lógica con exactamente el mismo retraso que el especificado en la descripción inicial, la herramienta de síntesis no es capaz de asegurar que la estructura por ella generada funcione como preveía la simulación de la original. Por esta razón el uso de estas construcciones en síntesis de alto nivel debe de ser limitada.

La lista de sensibilidad de un proceso es otra primitiva que puede causar problemas. Un proceso es una sentencia concurrente que se ejecuta de forma asíncrona. La lista de sensibilidad indica al simulador cuando debe iniciar la ejecución del proceso. Esto puede incrementar la eficacia en simulación pero no tiene una contrapartida general en lógica real. Para poder alcanzar una gran eficiencia en simulación el tiempo que mantiene la posibilidad de implementar la descripción, pueden ser definidos atributos que indiquen, a la herramienta de síntesis, que ignore una señal que ha sido usada para proporcionar una simulación más eficiente:

```
entity FHT is                                              -- FHT description
   port (    Datain   :  in     DATA;
             Dataout  :  out      DATA;
             Start    :  in     BIT;
             Clock    :  in     BIT  );
   type DATA is range -128 to 127;
   subtype synthesis_type is STRING;
   attribute synthesis_at : synthesis_type;                -- attribute declaration
   attribute synthesis_at of Start  : signal is "ignore";  -- attribute specification
   attribute synthesis_at of Clock : signal is "ignore";   -- attribute specification
   begin                                                   -- body
      assert Clock_CYCLE < 50 ns;
      assert Data_ARRIVAL_TIME = Clock_CYCLE;
   end;
architecture ALGORITHM of FHT is
   begin
   main : process( Start )
   ...
```

Por ejemplo, en la descripción anterior, la señal "**start**" indica al simulador cuando debe reiniciar la ejecución del proceso, pero no tiene significado durante la síntesis, razón por la cual se le ha asignado el atributo "**ignore**". Los atributos son un medio muy simple de proveer a la herramienta de síntesis de información adicional o restricciones sobre los objetos de la descripción.

El uso de la lista de sensibilidad conlleva igualmente restricciones en el uso de la primitiva "**wait**", por lo cual es recomendable sustituir dicha lista por un "**wait**", en el cuerpo del proceso, como muestra la siguiente descripción.

```
entity FHT is                                              -- FHT description
...
architecture ALGORITHM of FHT is
   begin
   main : process
      ...
                  wait on Start ;
      end process main;
   end;
```

En VHDL existen tres tipos de objetos: variables, señales y constantes. De ellos, las señales pueden presentar problemas en síntesis de alto nivel, ya que, mientras las variables actualizan su valor tan pronto son asignadas, las señales solo son actualizadas al final del ciclo de simulación, lo cual conlleva múltiples problemas a la hora de determinar las dependencias entre datos. Para resolver este problema pueden plantearse dos soluciones:

La primera es no utilizar señales en la descripción. Esta solución tiene el inconveniente de no poder utilizar puertos en la entidad, ya que estos objetos son señales por definición. Por

lo tanto, los puertos del sistema serán modelados por variables, las cuales deberán tener asociado un atributo, que indique en la herramienta de síntesis su carácter de puerto del sistema, así como su dirección tal y como muestra el ejemplo:

```
entity FILTER is                                          -- IIR Digital Filter
  begin
  ...
     process                                              -- passive process
       variable Dout : DATA;
       variable Dat  : DATA_ARRAY ( 0 to 4 );
       ...
       attribute synthesis_at of Dout : variable is "output_port";
       ...
         begin
       ...
       Dout := Dat(4);                                    -- new data
    ...
  end FILTER;
```

En este caso, el proceso es pasivo y tiene que ser definido en el cuerpo de la definición de la entidad. Despues de la síntesis de alto nivel, el algoritmo será implementado como una arquitectura con puertos de entrada y salida. La segunda posibilidad permite usar señales, pero incluyendo una sentencia "**wait**" cuando sea necesario actualizar inmediatamente el valor de la señal, como se puede ver en el ejemplo adjunto:

```
entity FILTER is                                          -- IIR Digital Filter
   port( Clock : in  BIT ;
         Dout  : out DATA ;
      ...
end FILTER;

architecture behavior of FILTER is
  begin
   Working : proccess
         variable Dat : DATA_ARRAY ( 0 to 4 );
         ...
           begin
         ...
         Dout <= Dat(4);                                  -- new data
         wait until ( Clock='1'and not Clock'stable ) ;
      ...
```

Se puede observar como una sentencia "**wait**" es utilizada para suspender la ejecución del proceso hasta la llegada de la transición activa del reloj (**Clock**). Esta señal solo tiene sentido durante la simulación, por lo cual posee el atributo "**ignore**". Es recomendable restringir el uso de señales a aquellos elementos que modelan la conexión del sistema con el mundo exterior.

Otro comando que debe ser tratado con cuidado es la primitiva "**wait**", cuando es utilizado con una señal que no debe ser ignorada por la herramienta de síntesis. De las tres posibilidades que ofrece el comando ("**wait on**", "**wait until**" y "**wait for**") solo el comando "**wait until**" tiene un reflejo directo hardware, aunque con un significado especial: se asume que el sistema generado sólo verificará la condición durante la transición activa del reloj, como es típico en los sistemas síncronos.

Con objeto de dar información adicional a la herramietna de síntesis, dos comandos pueden ser empleados: "**assert**" y "**attribute**". El segundo puede ser empleado para dar propiedades específicas a los objetos de la descripción (señales, variables, etc), mientras que el primero puede ser utilizado para dar directivas generales al proceso de síntesis tales como periodo máximo de reloj o el número máximo de unidades operacionales que pueden ser utilizadas.

Las variables usadas en la descripción algorítmica pueden ser de dos tipos "**integer**" y "**bit-vector**". Otros tipos, como "**bit**", "**positive**" y "**natural**" son modelados a partir de los anteriores. En general el usuario utiliza tanto el tipo entero como el "**bit-vector**". La desventaja de utilizar uno u otro reside en el hecho de que VHDL tiene operaciones pre-definidas solo en un tipo: las operaciones aritméticas son definidas sobre "**integer**" y las lógicas solo sobre "**bit-vector**". Una solución sería el uso de una librería estandar conteniendo la declaración de operacions aritméticas sobre "**bit-vector**", así como las funciones de conversión de "**integer**" a "**bit-vector**" y viceversa. La librería estará compuesta por una serie de "**packages**", cada uno asociado a un tipo específico de código binario (Signed-Magnitude, Ones-Comp, Twos-Comp, etc) [Sh85][Li86]. En el ejemplo se puede observar como se señala que todas las operaciones de la descripción deben ser realizadas en complemento-dos.

```
library OPER; use OPER.TWOS_COMP.all;                    -- Two Complement library
entity pdp8 is                  -- PDP8 description
  ...
architecture ALGORITHM of pdp8 is
  begin
   process
      variable M   : MEMORY ( 0 to 4095 );               -- memory
      variable PC  : WORD; -- Program.Counter
      variable eadd: WORD;-- effective address
        ...
     begin
        ...
      M( intval(eadd) ) := PC;          -- intval converts bit_vector to integer
      PC := eadd + O"0001";                                -- JMS instruction
        ...
```

El sistema PSAL2 dispone de un módulo que permite utilizar VHDL como descripción inicial del sistema a diseñar. El módulo VHDL2PSAL está formado por más de 8.000 líneas de código y, básicamente, es un compilador VHDL, en cuya especificación han sido utilizadas las herramientas de ayuda al diseño de compiladores LEX y BISON. Una serie de funciones escritas en C permiten, no solo mantener las tablas de símbolos y los datos necesarios en el proceso de compilación, sino también escribir dichos datos en formato PSAL2. La descripción generada por VHDL2PSAL puede ser entonces leída directamente por la herramienta de síntesis. Este módulo introduce ciertas restricciones en el uso de VHDL, con objeto de que el código generado pueda ser implementado por PSAL2. Algunas afectan a la filosofia del lenguaje, como por ejemplo que se suponga que las señales son actualizadas de forma inmediata, y por lo tanto, tengan sentido descripciones de comportamiento, en las cuales existan diversas asignaciones secuenciales a una misma señal (en VHDL estandard solo la última permanecería), o distintas lecturas de un mismo puerto en una sentencia interna de un process en un mismo ciclo de simulación (en VHDL, siempre se obtendría el mismo valor). PSAL2 mantiene el orden de acceso a una misma señal, pero no la interrelación entre ellos. Esto significa que para una señal concreta (por ejemplo "A"), se accederá a su valor o se le actualizará en la descripción final en el mismo orden que en la inicial. Sin embargo no se garantiza que si en la descripción inicial se actualiza la señal "B" antes que la señal "A", en la descripción final este orden se mantenga. Si el usuario desea mantener dicho orden, puede utilizar una sentencia "**wait**" entre ambas sentencias, lo que evitaría que la herramienta PSAL2 invirtiese el orden de las sentencias.

Por otro lado la herramienta supone que todas las operaciones aritmético-relaciones se realizan en complemento-dos y que pueden realizarse sobre operandos del tipo "**bit-vector**". Con ello se elimina la necesidad de funciones de conversión. PSAL2 trabaja internamente con datos de tipo "**bit-vector**", por lo cual todos los tipos de datos son transformados en dicho tipo. Por esta razón, se admiten "**bit-vector**" como índices de memorias (en vez de tipo entero), ya que PSAL2 no diferencia entre ambos tipos.

Siguiendo la misma filosofía adoptada en otras aplicaciones basadas en VHDL (como el interfase de simulación VHDL, XGHDL, desarrollado por GENRAD), se irá comentando que primitivas del estandar son adoptadas y con qué significado, utilizando para los diversos apartados la misma numeración que adopta el "IEEE Std 1076-1987".

3) Descripción del subconjunto de VHDL utilizado por PSAL2 para describir el sistema a nivel algorítmico

Capítulo 1.- Entidades de diseño y configuraciones

1.1.- Declaración de entidad.

El sistema será descrito mediante una única entidad. No se permite modelar jerarquia utilizando entidades.

1.1.1.- Cabecera de la entidad

1.1.1.1.- Genéricos

No permitidos en la implementación actual.

1.1.1.2.- Puertos

Los puertos pueden ser de tres tipos:

- in
- out
- inout

Su tipo tiene que ser obligatoriamente de un tipo predefinido ("integer", "positive", "natural", "bit-vector" o "bit")

Existen ciertos nombres de puertos que son reconocidas por la herramienta. Así una señal de salida denominada "psal_reset" es identificada como una señal de "reset" del sistema. Cuando este puerto de salida tome el valor 1, PSAL2 implementará un reset del sistema saltará al estado inicial.

1.1.2.- Parte declarativa de la entidad.

En la entidad se pueden declarar:

subprogramas
tipos
subtipos
constantes
atributos

Dichas declaraciones serán globales en la arquitectura.

1.1.3.- Parte de sentencias de la entidad

No se permiten comandos en este área.

1.2.- Cuerpo de la arquitectura

Sólo se permite una arquitectura en la descripción

1.2.1.- Parte declarativa de la arquitectura

Las restricciones son las mismas impuestas en la parte declarativa de la entidad (1.1.2). Estas declaraciones son globales al proceso que modela el sistema.

1.2.2.- Parte de sentencias de la arquitectura.
Como en el estandar.

1.3.- Declaración de configuración
No permitida.

Capítulo 2.- Subprogramas y paquetes.

2.1.- Declaración de subprogramas

En la versión actual no se permiten las declaraciones de subprogramas: toda especificación debe ir acompañada por la parte declarativa del subprograma y su parte de sentencias. Para definir subprogramas debe de usarse la regla 2.2.

2.1.1.- Parámetros formales

Como en el estandard.

2.2.- Cuerpos de subprogramas.

En la parte declarativa se admiten declaraciones de:

- subprogramas
- subtipos
- tipos
- constantes
- variables
- alias
- atributos

2.3.- Redefinición de subprogramas

Si existen dos subprogramas con el mismo nombre, el sistema utilizará el definido en último lugar

2.3.1.- Redefinición de operadores

No permitida

2.4.- Funciones de resolución

No permitidas.

2.5.- Declaraciones de paquetes

No permitidas.

2.6.- Cuerpo de paquetes

No permitido

2.7.- Reglas de conformidad

No verificadas.

Capítulo 3.- Tipos

3.1.- Tipos escalares

Sólo se permite el tipo entero. En la implementación actual el valor máximo está limitado por 2^{32}. No se permiten valores o rangos negativos, ni expresiones en la especificación del tipo.

3.2.- Tipos compuestos

3.2.1.- Tipos de Array

No se permiten arrays de subtipos que a su vez sean arrays (como máximo se permiten array unidimensionales).

3.2.1.1.- Restricciones de indices y rangos discretos

El array modela memorias, en las cuales el índice modela las direcciones y la indicación de subtipo las palabras. No se permiten expresiones en las especificaciones. En la descripción se podrá utilizar un elemento de tipo

"**bit-vector**" como índice, sin necesidad de funciones de conversión, ya que esta es realizada automáticamente por PSAL2.

3.2.1.2.- Tipos de array predefinidos

Sólo se reconoce el subtipo "**bit-vector**".

3.2.2.- Tipo record

No permitido

3.3.- Tipo acceso

No permitido

3.4.- Tipo fichero

No permitido

Capítulo 4.- Declaraciones

4.1.- Declaración de tipo

Sólo se permiten declaracions de tipo completas. Se pueden definir tipos escalares o compuestos. Los rangos deben de ser valores naturales.

4.2.- Declaración de subtipo

No se permiten funciones de resolución. Los rangos o valores deben de ser naturales.

4.3.- Objetos

4.3.1.- Declaración de objetos

Se pueden declarar:

- constantes
- variables

No se permiten señales.

4.3.1.1.- Declaración de constantes

El valor de la constante debe de ser natural. Si se desea expresar otros valores puede utiliarse literales del tipo bit-vector literal.

4.3.1.2.- Declaración de señales

No permitida

4.3.1.3.- Declaración de variables

No se permiten inicializaciones.

4.3.2.- Declaración de ficheros

No permitida.

4.3.3.- Declaración de inerface

Sólo se permiten los modos:

- in
- out
- inout

4.3.3.1.- Lista de interfaces

No se permite el no-terminal **generic_interface_list** utilizado en la regla 1.1.1.1.

4.3.3.2.- Lista de asociación

En los elementos de esta lista, no pueden aparecer los parámetros formales. Son por lo tanto elementos "posicionales". No se permite el parámetro "**open**" para indicar la no consideración de un parámetro.

4.3.4.- Declaración de alias

Como en el estandard

4.4.- Declaración de atributos.

Sólo se reconoce una declaración

attribute synthesis-attribute : string ;

4.5.- Declaración de componentes

No permitida

Capítulo 5.- Especificaciones

5.1.- Especificación de atributos:

Sólo se identifica un valor para el atributo "**synthesis-attribute**": "**ignore**". Este atributo solo puede ser especificado para objetos. Su significado ya fué comentado en la introducción.

5.2.- Especificación de la configuración.

No permitido

5.3.- Especificación de la desconexión

No permitido

Capítulo 6.- Nombres

Se permiten todos los tipos fijados en el estandar salvo "**selected_name**" y "**attribute_name**". En los "**indexed_names**" sólo se puede utilizar un elemento ya que los arrays son uni-dimensionales.

Capítulo 7.- Expresiones

7.1.- Expresiones

Como en el estandar.

7.2.- Operadores

Se consideran definidas para todos los tipos aceptados por el sistema. Se considera que retornan un objecto de tipo "**bit-vector**".

7.2.1.- Operadores lógicos

Se admiten las definidas por el estandar.

7.2.2.- Operadores relacionales

Se admiten los definidos por el estandar.

7.2.3.- Operadores de suma

Se admiten los definidos por el estandar con las salvedades reseñadas en el punto 7.2.

7.2.4- Operadores de multiplicación

Se admiten las definidas por el estandard con las salvedades reseñadas en el punto 7.2.

7.2.5.- Operadores Misceláneos

No se admiten

7.3.- Operandos

7.3.1.- Literales

Se permiten:

- literales numéricos abstractos.
- literales "**bit-vector**".

7.3.2.- Agregados

No permitidos

7.3.3.- Llamadas a funciones

A diferencia del estandar, la llamada a funciones siempre tiene que tener asociada una lista de parámetros actuales. En el caso de no existir parámetros en la función, dicha lista estará vacía y por lo tanto no existirán elementos entre los paréntesis.

7.3.4.- Expresiones cualificadas

No permitidas

7.3.5.- Conversión de tipo

No tiene sentido. PSAL2 solo utiliza un tipo internamente, por lo cual no es necesario realizar transformaciones de tipos.

7.3.6.- Localizadores

No permitidos

7.4.- Expresiones estáticas

Sólo se permiten literales y constantes.

7.5.- Expresiones Universales

No se tienen en cuenta los tipos de los objetos a la hora de analizar las operaciones, ya que el sistema solo utilizada datos de tipo "**bit-vector**" y la información referente al tipo se pierde en el proceso de compilación.

Capítulo 8.- Sentencias secuenciales

8.1.- Wait

Sólo se permite la construcción:

"wait until ..."

Si la señal sobre la que se aplica posee el atributo "**ignore**" dicha sentencia es ignorada.

8.2.- Sentencia Assert

No permitida

8.3.- Asignación de señal

La asignación secuencial de señales es permitida con varias restricciones:

- Sólo se permite un wareform_element.
- No se permiten retardos. Luego no puede utilizarse la contrucción "**after**".
- No se modelan las desconexiones del driver, por lo cual no se permiten sentencias del tipo "**null**".
- No se permite utilizar el modificador "**transport**", ya que no se pueden modelar retardos con esta sentencia.

8.4.- Asignación de variables.

La única diferencia con el estandar es que no existen restricciones en cuanto a los tipos a ambos lados de la expresión.

8.5.- Llamadas a procedimientos

Se han modelado conforme al estandar. A diferencia de las llamadas a funciones, en las llamadas a procedimientos puede omitirse la parte de parámetros actuales.

8.6.- Sentencia if.

La única diferencia con el estandard es que la condición no tiene que ser de tipo "**Boolean**". Si la condición es "0" el "**if**" no se ejecuta. En cualquier otro caso se ejecuta.

8.7.- Sentencia case

Se ha modelado como en el estandar. Se permite utilizar la opción "**others**", y no se permite el caracter '|' para modelar rangos.

8.8.- Sentencia loop

De los tres posibles tipos de lazos ("**loop**", "**while**" y "**for**") solo se ha modelado el lazo de tipo "**while**". Como en la sentencia "**if**", la condición no tiene por qué ser

"boolean". Para su ejecución se siguen las normas fijadas en la regla 8.6.

8.9.- Sentencia Next.

Ha sido modelada como en el estandar.

8.10.- Sentencia exit

Ha sido modelada como en el estandar.

8.11.- Sentencia return

Ha sido modelada como en el estandard

8.12.- Sentencia null

Esta sentencia es ignorada por el compilador, pero permite crear caminos alternativos con objeto, por ejemplo, de cubrir las diversas posibilidades de una sentencia condicional.

Capítulo 9.- Sentencias concurrentes

Sólo se permite una sentencia concurrente, el proceso ("process"). Su definición (regla 9.2) del estandar, ha sido modificada:

- No se permite la utilización de la lista de sensibilidad.
- Sólo se permite un proceso.

El sistema a diseñar es modelado mediante un único proceso, el cual es traducido en una función en PSAL2, cuyo nombre es el nombre de la entidad. Esta función (la función principal de la descripción; la primera que se ejecuta), posee como última instrucción una sentencia de tipo "restart" (reinicio de la función). Este comando, introducido automáticamente por el programa VHDL2PSAL, tiene por objeto modelar la reejecución indefinida que caracteriza a un proceso en VHDL.

Capítulo 10.- Alcance y visibilidad de las declaraciones

En los elementos VHDL admitidas por el módulo VHDL2PSAL se siguen las reglas de visibilidad y alcance reseñadas en este capítulo del estandar.

Capítulo 11.- Unidades de diseño y su análisis

Como ya hemos comentado, la descripción del sistema digital está formada por una única entidad, con una única arquitectura. Por lo tanto el fichero de diseño estará formado por una única unidad de diseño, en la cual existirá una unidad de librería formada por dos unidades: una declaración de entidad (unidad primaria) y un cuerpo de arquitectura (unidad secundaria).

Capítulo 12.- Elaboración y ejecución

Este capítulo hace referencia principalmente a aspectos relacionados con la simulación de la descripción. Algunos de estos aspecptos y los problemas que conllevan en la síntesis de alto nivel, fueron comentados en un apartado anterior. Aspectos generales relacionados con elementos concretos han sido comentados durante la descripción del elemento particular. Por ello no se hacen referencias adicionales a este capítulo.

Capítulo 13.- Elementos léxicos

El subconjunto propuesto no admite los siguientes elementos:

- literales decimales
- literales con base
- literales carácter
- literales de tipo "string"
- No se utiliza el conjunto de caracteres propuesto para reemplazar a limitadores en entornos que carezcan de estos últimos.

Capítulo 14.- Entorno del lenguaje predefinido

14.1.- Atributos predefinidos

No se permiten

14.2.- Paquete Standard

Se reconocen los tipos:
- bit
- integer
- natural
- positive
- bit-vector

14.3.- Paquete TEXTIO

No se reconoce ningún elemento de dicho paquete.

4) Utilización de VHDL a nivel de transferencia de registros

En una descripción a nivel de transferencia de registros, el diseñador define explícitamente los registros y unidades operacionales (OUs) del sistema asi como su interconexión. VHDL no dispone de una definición explícita de registros, aunque es posible definir señales con "**signal_kind**" de registro.

Por otro lado se han propuesto dos convenciones para modelar las unidades operacionales. La primera define una OU como un bloque (primitiva "**block**") sin señal de guarda. El código de operación es definido como una señal de tipo enunciativo, cuyo valor final será asignado por la herramienta de síntesis lógica. La señal que modela la salida de la OU es generada como una función de los valores de entrada y del código de operación en un bloque que modela lógica combinacional, tal como OU1 en el siguiente ejemplo:

```
...
architecture behavior of FILTER is
    signal OU1_S : BIT_VECTOR ( 0 to 31 );        -- OU output
    signal OU1_1 : BIT_VECTOR ( 0 to 31 );        --  first input of OU
    signal OU1_2  : BIT_VECTOR ( 0 to 31);        -- second input of OU
    signal OU1_C : OU1_control;                   -- OU control signal
       type OU1_control is ('+','*');             -- control signals
...
begin
    ...
    OU1: block                                    -- OU1 is defined as concurrent block
       begin
       with OU1_C select
          OU1_S <= OU1_1 + OU1_2 when '+', ...
       end block OU1;
```

Otra posibilidad consiste en utilizar funciones para describir las OUS. La filosofía es similar a la anteriormente descrita, pero sin la necesidad de definir las señales de entrada de la unidad operacional:

```
-- Other possibility : OU1 is declared as function
 function OU1 ( Dat1: BIT_VECTOR(0 to 31);    Dat2: BIT_VECTOR(0 to 31);
      control : OU1_control )                  return BIT_VECTOR (0 to 31);
```

Dado que PSAL2 realiza la localización de operacions en OUs, no son necesarias operaciones aritméticas en la descripción a nivel RT. Como consecuencia de la versatilidad de VHDL, se han propuesto varios estilos distintos en la descripción de sistemas a nivel RT [VHDL89][Bh88][St89][SaVi90]. Estos estilos se pueden agrupar en dos categorias:

La primera utiliza una descripción de tipo "flujo de datos" basada en bloques [SaVi90]. Una posibilidad es utilizar dos bloques: "Data Path" y "Control Unit". Ambos bloques usan expresiones de guarda para determinar cuando los registros son cargados. Estos objetos son modelados utilizando señales con guarda, mientras los terminales son modelados como señales sin guarda. En el bloque "Data Path" los nuevos valores de los registros son definidos utilizando asignaciones de señales condicionales. Estas asignaciones tienen la peculiaridad de que si la condición no es cierta, un cierto valor por defecto es cargado. En los ejemplos mostrados, el valor por defecto es el valor actual de la señal.

Para impedir la generación de lógica innecesaria por parte de la herramienta de síntesis lógica, esta debe identificar la carga de una señal con su valor actual como una situación en la cual no se cargan nuevos valores en el registro. El valor del próximo estado y las señales de control de las OUs (cuando se describen utilizando bloques) son generadas en la unidad de control.

Dado que el registro de estados debe ser conocido por la herramienta de síntesis lógica, puede utilizarse un atributo para identificarlo. Las transferencias de estado se modelan como cargas del nuevo estado en el registro de estados. Además, se utilizan valores simbólicos para modelar los diferentes estados, dejando a la herramienta de síntesis lógica la libertad de asignar dichos estados a los valores más convenientes para reducir el coste total del diseño [WhNe90]. Los atributos y "assert" son usados en estos ejemplos de la misma forma que lo eran en la descripción algorítmica, proporcionando información adicional a la herramienta de síntesis. Esta clase de descripción es similar a la utiliada en algunas herramientas de síntesis lógica como BDSYN y es fácil obtener una a partir de la otra:

```
...
architecture behavior of FILTER is
    signal Suma   : BIT_VECTOR ( 0 to 31 ) register;          -- Suma is a register
    signal estado : state register;                          -- state register
       type state is (S0,S1,S2,S3,S4,S5);                    -- state list
    attribute synthesis_at of estado : signal is "state_register";  -- attribute specification
...

begin

    Data_path: block ( Clock='1' and not Clock'stable)
       begin
          ...
          Suma <= guarded
                    OU1_S when
                          ((estado=S0 and Reset='0' and Prog='0' ) or ... else
                     Suma;
                       ...
        end block Data_path;

 Control_part: Block ( Clock='1' and not Clock'stable )
       begin
          estado <= S0 when ...
          OU1_C <= '+' when ...
       ...
```

La técnica descrita anteriormente presenta dos inconvenientes: en primer lugar es difícil comprender cual es el funcionamiento del sistema, dado que se trata más de una descripción de funciones lógicas, que de una representación RT clásica. En segundo lugar, la misma condición es repetida en múltiples asignaciones, lo que puede provocar la generación de lógica innecesaria al tiempo que hace difícil determinar qué asignaciones se ven afectadas por una condición.

Existen varias alternativas: por ejemplo, dividir la descripción en bloques, cada uno asociado a un estado [VHDL89]. La expresión de guarda de un blqoue es activada cuando el estado actual es el asociado al bloque. Esta metodología genera un único bloque en vez de los dos bloques ("Data Path" y "Control Unit") propuestos en el esquema anterior. En esta metodología se incluyen todas las acciones que se ejecutan en el mismo estado en el mismo bloque. Por lo tanto, en él se incluyen tanto transferencias de datos como de control. Esta descripción es del tipo utilizado en RTLs clásicos como DDL:

```
...
architecture algorithm of FHT is
 signal count : BIT_VECTOR ( 9 downto 0 ) register;
 ...
 begin

 system : block
          constant cero10 : BIT_VECTOR ( 9 downto 0) := B"0000000000";
           ...
          begin
           S0 : block ( estado=S0 and Clock='1' and not Clock'stable)
              begin
                  count <= guarded cero10;
                     ...
                  estado <= guarded S1;
               end block S0;
               ...
```

Otra alternativa al uso de descripciones de tipo flujo de datos a nivel RT es la basada en el uso procesos [SaVi90]. En esta alternativa, los registros pueden ser modelados de dos formas: La primera utiliza señales para modelar los registros y variables para modelar los terminales. La segunda utiliza variables para modelar tanto los registros como los terminales. En este caso, si una variable es utilizada en un proceso con reloj antes de ser definida, entonces dicha variable modela un registro. Si es usada despues de ser definida, entonces modela un terminal [St89].

En este estilo de descripción, las OUs son modeladas utilizando funciones. El proceso comienza con una sentencia "**wait**", para sincronizar la carga de los registros con el reloj. Este es el único comando de este tipo permitido en las descripciones. A continuación un comando del tipo "**if** ... **then** ... **else** ..." es utilizado para introducir las condiciones de "reset" del sistema. Dichas condiciones son necesarias no solo para inicializar el sistema, sino también para facilitar el test final del circuito. A continuación se describe el autómata de control utilizando una primitiva del tipo "**case**", en la cual el registro de estado es la variable de selección. Cada estado del sistema es modelado como una alternativa de dicho comando "**case**". Este estilo de descripción emula construcciones de algunos RTL clásicos, como el "AUtomaton" de DDL o el "**sequence**" de CVS-BK.

```
entity pdp8 is          -- PDP8 RT description
    port (
      Clock : in    BIT ;
      Stop  : inout BIT ;
...
architecture ALGORITHM of pdp8 is
 signal PC     : BIT_VECTOR( 11 downto 0 ) ;          -- Program counter register
 signal eadd   : BIT_VECTOR( 11 downto 0 ) ;          -- Effective address register
 signal estado : state ;                              -- State register
...
 begin
    main: process
       begin
          wait until ( Clock='1' and not Clock'stable);
          if( Stop='0') then                          -- Reset condition
             estado <= S0;
           else-- finite state machine
             case estado is
                ...
                when S12 =>                           -- state S12
                PC <= eadd;
                estado <= S30;                        -- Next state = S30
       ...
```

Existe otra forma alternativa de utilizar la sentencia "**process**" para modelar sistemas a nivel RT. Esta aproximación obliga a utilizar una señal y un proceso para modelar cada estado [Bh88]. En cada uno de estos procesos se definen las acciones a realizar en dicho estado. La señal asociada al estado aparece en la lista de sensiblidad del proceso asociado, por lo tanto, un cambio en el valor de la señal implicará que el proceso sea ejecutado. La transición de estados se define, por lo tanto, como una simple inversión de la señal asociada al próximo estado.

```
...
architecture ALGORITHM of pdp8 is
 signal PC     : BIT_VECTOR( 11 downto 0 ) ;          -- Program counter register
 signal eadd   : BIT_VECTOR( 11 downto 0 ) ;          -- Effective address register
 signal S12 : BIT ;                                   -- State S12
 signal S30 : BIT ;                                   -- State S30
 ...
 begin
    ...
    S12: process ( S12)                               -- State S12
       begin
          PC <= eadd;
          S30<= not S30;                              -- Next state = S30
       end process S12;
 ...
```

5) Modelo de descripción a nivel de transferencias de registros utilizado por el sistema PSAL2

Despues de estudiar y analizar las diversas alternativas propuestas en el apartado anterior, decidimos adoptar una filosofía de descripción basada en el uso de un único proceso, para representar el sistema resultante del proceso de síntesis a nivel de transferencia de registros en VHDL.

La herramienta de síntesis PSAL2 utiliza señales para modelar registros y variables para los terminales. Una sentencia de tipo "**wait**" (la única que de este tipo en la descripción), al final del proceso, señala el instante en el cual todos los registros son actualizados. Las unidades operaciones son modeladas mediante funciones, utilizándose variables simbólicas para modelar los códigos de operación.

El registro de estados es identificado mediante un atributo, para que de esta forma sea reconocido por la herramienta de síntesis estructural. Esta señal de tipo enunciativo toma valores simbólicos, que mas tarde, otra herramienta reeemplazará por valores reales (una vez la asignación secundaria haya sido realizada).

El único "**process**" que modela el sistema consta de dos elementos: por un lado, una sentencia "**if**", que señala el comportamiento del sistema en modo de "reset" (cuando la condición de la sentencia es cierta) o en modo normal (cuando la condición no es cierta). En segundo lugar, aparece una sentencia "**wait**", señalando cuando se cargan los registros. La sentencia "**if**", posee como condición, las situaciones que provocan un estado de "reset" en el sistema. Para ello se utilizan 2 señales ("psal_reset" y "psal_reset_terminal") que son introducidas por el sistema de forma automática. En esta situación el sistema no ejecuta ninguna instrucción, limitándose a esperar el cambio de la señal "psal_reset_terminal", que indique cuando debe comenzar a funcionar el sistema.

Cuando el sistema no se encuentra en la situación anteriormente descrita (parte "**else**" del "**if**" anteriormente comentado), su comportamietno es modelado utilizando sentencias secuenciales. Dado que el comportamineto del sistema es modelado mediante una máquina de estados finitos, el sistema PSAL2 utiliza las opciones de una sentencia de tipo "**case**" para representar las acciones que se realizan en cada estado de la FSM. Dicha sentencia posee como selector el registro de estados, modelando cada alternativa un estado. Las transferencias de estado son modeladas mediante cargas en el registro de estado. El sistema puede utilizar prácticamente todas las sentencias de tipo secuencial para modelar el diseño. Al igual que a nivel algorítmico algunas construcciones de VHDL no pueden utiliziarse a nivel RT, como por ejemplo, los ficheros I/O, las expresiones de retraso, etc.

CAPITULO 3

PSAL1 : SISTEMA DE SINTESIS ALGORITMICA

I) Introducción

En este capítulo, expondremos las técnicas y algoritmos cuya implementación ha dado origen al sistema PSAL1 (Programa de Síntesis ALgorítmica, versión 1). Esta herramienta implementa una descripción a nivel algorítmico, escrita en ISPS, en una arquitectura a nivel de transferencias de registros proporcionada en DDL. PSAL1 se encuentra dividido en dos módulos (figura 3.1) :

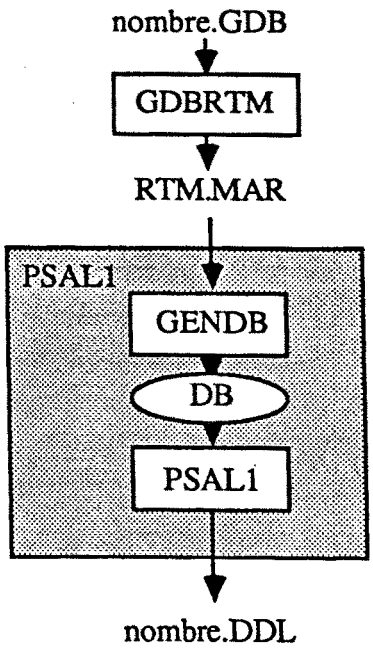


Figura 3.1

a) Parser (GENDB) . La misión de este programa es transformar la descripción ISPS en un grafo que permita posteriormente el análisis y síntesis de la descripción. Este parser además de generar el código intermedio, realiza algunas optimizaciones relacionadas con la sintaxis de ISPS. La salida del parser es un fichero escrito en formato binario que contiene toda la información del grafo.

b) Programa de análisis y síntesis automática (PSAL1). Este módulo utiliza los datos generados por el parser para obtener una descripción a nivel de transferencias de registros. En él se realiza un análisis de variables, una síntesis de registros y unidades operacionales así como una asignación de operaciones a estados.

Dos son los condicionantes que determinan el tipo de sistema que PSAL1 va a sintetizar. Por un lado, las metodologías de síntesis del programa basadas en un determinado estilo de diseño. Por otro los lenguajes utilizados para describir el sistema antes y después del proceso de síntesis. En este sentido, el programa solo sintetizará correctamente aquellos circuitos que admitan una descripción algorítmica en ISPS. Como ya hemos comentado, aunque ISPS permite la descripción algorítmica de cualquier tipo de sistema, se trata de un lenguaje adaptado principalmente a procesadores de conjunto de instrucciones. Estas características del lenguaje de entrada influyen en el propio proceso de síntesis. El lenguaje de salida también condiciona el tipo de sistemas que se pueden sintetizar.

Vamos a resumir a continuación los condicionamientos del proceso de síntesis del programa PSAL1, debidos a las causas antes comentadas y que deben de tenerse en cuenta a la hora de su utilización:

a) La implementación final va a ser síncrona.

b) En la arquitectura del circuito sintetizado van a diferenciarse dos unidades. Por un lado, la unidad de datos (UD) en la que van a tener lugar las transferencias de información especificadas en el algoritmo de entrada. Por otro, la unidad de control (UC) encargada de determinar que transferencias de información deben de ejecutarse en cada instante de tiempo. Esta arquitectura se adapta perfectamente al lenguaje de salida DDL, en su declaración <AU>.

c) Las unidades operacionales admitirán dos operandos como máximo.

d) En DDL existen dos tipos de transferencias de información: una asociada a registros y controlada por reloj y otra a terminales. En la arquitectura RT ciertas señales van a tomar un valor combinacional en función de las entradas y el estado. Los registros, sin

embargo, actualizan su valor en el instante de la llegada del ciclo de reloj. PSAL1 implementa todas las variables ISPS en registros, salvo que el usuario le indique el hacerlo en terminales, con objeto de implementar puertos del circuito.

e) No se permiten encadenamientos de operaciones, salvo en el caso de operaciones de control. La razón es que en caso de permitirlo, aparecerían unidades operacionales encadenadas, estilo de diseño que no hemos contemplado en la implementación actual de PSAL1.

En PSAL1 la descripción se encuentra representada en forma de grafo. Para crear dicha representación, se realizó un estudio del tipo de grafo que mejor se adaptaba al tipo de representación utilizado y a las transformaciones que en ella se debían de efectuar. La decisión final fue la de un grafo de flujos por los siguientes motivos:

- Esta representación es capaz de soportar todas las transformaciones consideradas y ademas de la forma más simple.
- Por otra parte, era la más sencilla. Incluso es fácil pasar de esta representación a la VT del sistema CMUDA.

El grafo que utilizamos presenta algunas diferencias con un grafo de flujos tradicional. La más importante es que admite que varias operaciones se realicen al mismo tiempo.

Una vez decidido el tipo de grafo, el siguiente paso fue determinar de donde extraer dicha información. En principio aparecian dos opciones: utilizar la descripción ISPS o el árbol que genera el traductor, denominado "Base de Datos Global" (GBD). Inicialmente se decidió utilizar la Base de Datos.

```
GDB:I;ISPS V6(12)-MI;MARK1.ISP; 1-MAR-1988 21:11:56.52;
(ISPSDECLARATION
  (EDECLR
    (EHEAD
      MARK1
      (FCSET))
    (SECTIONLIST
      (SECTION
        MP.STATE
        (EHEAD
          M
          NIL
          (: 0   8191)
          (: 31  0)))
```

figura 3.2. Extracto de la descripción GDB del MARK1

En la figura 3.2 aparece un ejemplo de dicha representación. Después de estudiar cómo funcionaban los generadores de código intermedio y los analizadores sintácticos [AhSe86] de los compiladores, empezamos a obtener los grafos sintácticos de la representación GDB, paso previo para el desarrollo del compilador. Sin embargo, un estudio más profundo de los programas que forman el simulador de ISPS, hizo que dicha elección fuese modificada. Se decidió utilizar el programa "gdbrtm" que, tomando la descripción GDB, obtiene unas tablas de datos, las cuales son empleadas posteriormente por el simulador. Estas tablas son almacenadas en un fichero llamado RTM.MAR, y se expresan en lenguaje MACRO (macroensamblador del VAX). Tal elección nos obligó a estudiar los programas fuente de "gdbrtm" (escritos en BLISS-32), con objeto de encontrar el significado de la información contenida en el fichero RTM.MAR.

Este trabajo inicial nos permitió implementar un pequeño traductor, programa escrito en C que toma el fichero RTM.MAR y lo transforma, logrando que la información contenida en las tablas sea más fácil de manejar. Tal herramienta ha sido muy importante en el desarrollo del sistema ya que nos ha permitido determinar la filosofía y códigos de operación de las tablas de datos almacenadas en RTM.MAR.

Una vez determinada la estructura de las tablas empezamos a desarrollar el "Parser" de nuestro sistema. Como dicho programa iba a estar escrito en lenguaje C, necesitábamos introducir en el fichero RTM.MAR, unas variables que permitiesen al programa en C leer las tablas escritas en ensamblador .

La filosofía de la primera parte del sistema PSAL1 es muy parecida a la del simulador de ISPS, como se observa en la figura 3.3. Una vez el programa "gdbrtm" ha generado, a partir de la Base de datos (GDB), el fichero RTM.MAR {en la bibliografía se le denomina fichero de código RTM (register transfer machine)}, se ejecuta el programa "trans.c", el cual introduce en el fichero RTM las variables que necesita el "Parser". A continuación se compila el fichero resultante (escrito en MACRO), y el código objeto resultante es "linkado" con el "Parser" y los programas de optimización. Esto produce un fichero, cuya ejecución permite crear el grafo de flujo y optimizarle.

Actualmente, la salida del sistema es un fichero en donde aparecen los distintos bloques básicos que forman el grafo de flujo, así como las conexiones entre ellos. El sistema genera además un fichero con formato binario, que permite una rápida utilización de los datos.

Los cuatro programas que forman este primer módulo, serán comentados con más profundidad más adelante, pero de todas formas vamos a señalar brevemente su función:

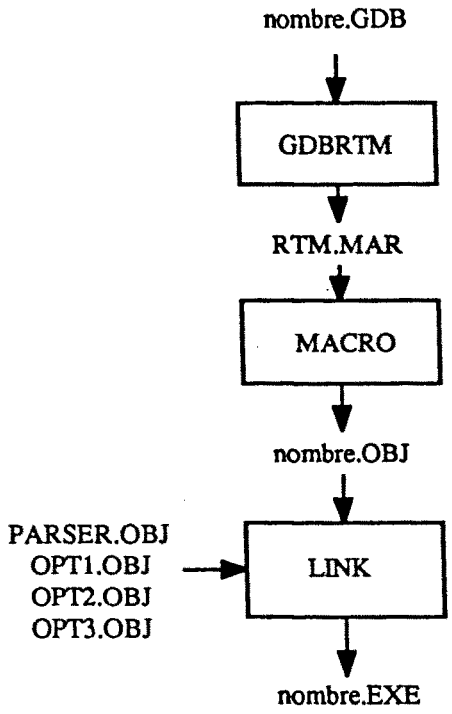


figura 3.3

a) Programa principal.

Su misión es crear el grafo de flujos. Se diferencia de un "Parser" tradicional, en que sólo traduce aquellas entidades que son utilizadas en la descripción. Si una entidad es definida pero no utilizada en la descripción ISPS, es ignorada por este programa. Esto hace que si, por ejemplo, el usuario no marca una entidad con el calificativo "main", el programa diga que no hay nada que traducir, ya que no sabe qué entidad tiene que ejecutar en primer lugar.

b) Primera optimización.

Este programa tiene por objeto reducir el número de entidades de la descripción. Para ello trata de sustituir la llamada a la entidad por el cuerpo de ésta (sustitución de subrutinas "in-line"). Unicamente se aplica si dicha entidad sólo es llamada una vez. Además en el caso de realizar la sustitución, reconvierte las sentencias LEAVE, RESUME y RESTART que aparecen en la entidad.

c) Segunda optimización.

Tiene como misión reducir el número de bloques básicos, aprovechando la capacidad del grafo para expresar la ejecución simultánea de varias operaciones dentro de un

bloque básico. Para ello realiza una emulación de la ejecución del programa, con objeto de garantizar que el orden de ejecución en el grafo sea el mismo que el de la simulación.

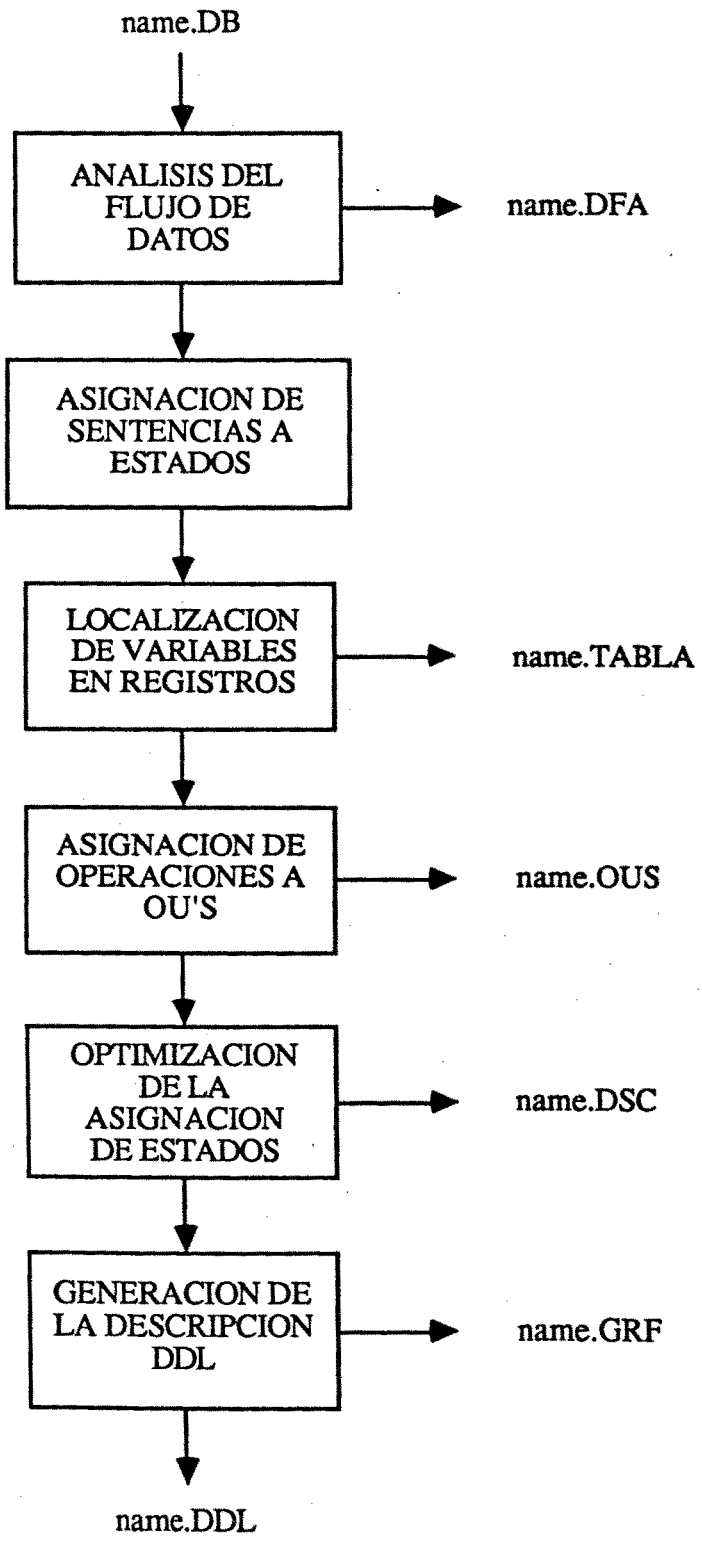


figura 3.4

d) Tercera optimización.

Antes de que el grafo sea escrito en el fichero, tiene lugar una última optimización. En ella se eliminan bloques que nunca son ejecutados. Además se realiza una revisión del grafo, asociando bloques cuando esto sea posible, con lo cual se compacta la descripción.

La segunda parte del programa, toma la base de datos, escrita en el fichero "nombre.dB" y procede a su síntesis. Como se ve en la figura 3.4, sobre la base de datos generada por el Parser se realizan tareas tales como:

a) Análisis de variables.

Este módulo determina las dependencias entre variables, realizando al mismo tiempo diversas optimizaciones tales como eliminación de operaciones redundantes o sustitución de operaciones comunes.

b) Asignación de sentencias a estados.

El sistema permite realizar una asignación de estados controlada por el usuario. Tres son las opciones que ofrece:

1) Respetar el orden que las operaciones tenían en la descripción ISPS. Esta opción permite obligar al sistema a realizar una determinada asignación, lo que facilita la verificación de otros módulos.

2) Asignar sin limite en el número de unidades operacionales. El usuario que escoge esta opción desea obtener la implementación más rápida del sistema, sin importar el coste de las unidades operacionales.

3) Asignar con restriccion en el número de unidades operacionales. En un sistema digital, las unidades operacionales suelen ser los elementos más costosos desde el punto de vista del area que consumen. Una forma de limitar el coste en area del sistema es, por lo tanto, limitar el número de unidades operacionales. Una vez fijada esta limitación, el sistema asocia las operaciones a estados con esa restricción.

c) Síntesis de registros.

Este módulo es el encargado de localizar cada variable de la descripción en un registro.

d) Localización de operaciones en unidades operacionales.
Las operaciones son localizadas en unidades operacionales. Para ello se desarrolló un algoritmo iterativo-constructivo que evita los efectos laterales de este tipo de métodos.

e) Optimización del control y generación de la descripción DDL.
En el paso b) las operaciones fueron asignadas a estados. Dicha asignación es optimizada en este módulo, reduciendo el número de estados. Una vez completada dicha tarea, el sistema es descrito utilizando el lenguaje DDL.

Con ello el sistema es capaz de implementar la descripción algorítmica ISPS original describiendo la arquitectura resultante en DDL.

II) La descripción RTM (Register Transfer Machine).

La descripcion RTM es utilizada por PSAL1 como entrada. En este apartado se comentan algunos problemas surgidos en el uso dicho formato. Una descripción más detallada se encuentra en [Sa89].

La descripción se encuentra contenida en el fichero RTM.MAR, en forma de estructuras descritas en lenguaje macroensamblador y consta de once secciones. Antes de ser utilizada, la descipción RTM es preprocesada por el programa "trans", encargado de reemplazar ciertas instrucciones en macroensamblador por la definición de una serie de variables que precisa el programa principal (escrito en C) para poder leer la estructura.

La descripción del sistema se encuentra en una de las secciones, en forma de tabla de sentencias. El simulador ejecuta un elemento de la tabla en cada paso (si el algoritmo no tiene operaciones en paralelo). Cada elemento es una instrucción RTM. Por lo tanto, después de ejecutarse la estructura i, se ejecutará la i+1, salvo que la i sea una estructura de control, e indique que la ejecución deba continuar en otras estructuras. Existen 132 códigos de operación reconocidos en la representación los cuales, básicamente, coinciden con las operaciones definidas en ISPS. Estos códigos son los mismos que utiliza el sistema PSAL1.

La tabla de sentencias posee en su organización algunos aspectos destacables por su influencia en el proceso de síntesis. Asi, por ejemplo, la definición de una entidad que se inicia con una operación denominada "begin" y finaliza con la instrucción "end"puede

realizarse en cualquier parte de la descripción. Generalmente, si en la operación i se llama a una entidad que aún no ha sido definida, la estructura i+1 es una operación de salto y la i+2 el inicio de la definición de dicha entidad. La instrucción de salto i+1 señala a la estructura que se encuentra después de la operación "end" de la nueva entidad. Este hecho dificulta la inteligibilidad de la descripción original. En el grafo de flujos del sistema PSAL1 se ha evitado este problema.

La operación IF tiene asociada una estructura, denominada "smerge", en la cual convergen los dos caminos de la operación. Uno de los índices sucesores en el IF es el correspondiente a la estructura smerge, y el otro a una secuencia de operaciones que también termina en el smerge. El nombre de esta operación indica que la ejecución debe continuar cuando alguna rama ha terminado su ejecución ("serie-merge"). Esto implica que las sentencias IF no tienen clausula ELSE, lo que hizo necesario desarrollar un algorítmo especial en PSAL1. Las sentencias DECODE tienen un tratamiento muy parecido. En la figura 3.5 se observa como se representa esta operación.

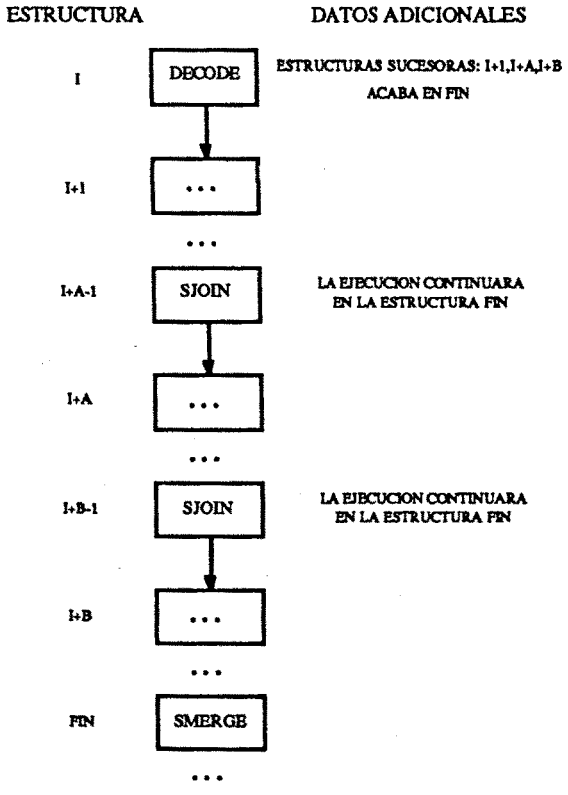


figura 3.5

ESTRUCTURA	OPERACION	INFORMACION ADICIONAL
1	DIVERGE	SUCESORAS: 2,4,6 FINALIZA EN 7
2	a=0	
3	PJOIN	CONTINUAR LA EJECUCION EN LA ESTRUCTURA 7
4	b=1	
5	PJOIN	CONTINUAR LA EJECUCION EN LA ESTRUCTURA 7
6	c=0	
7	PMERGE	

figura 3.6

Las acciones que se ejecutan en paralelo se representan en la tabla de sentencias de la misma forma que las acciones condicionales. Estas acciones van precedidas de una operación

llamada "diverge" asociada a una del tipo "pmerge". En la figura 3.6, aparece la representación que tendría la sentencia:

a=0; b=1; c=3 next

El simulador, cuando encuentra la operación "pmerge", espera a que todas las secuencias sucesoras del nudo "diverge" terminen su ejecución antes de seguir. PSAL1 utiliza un paso de optimización para simplicar esta estructura.

Por último debe señalarse que todas las operaciones que se realizan en la tabla de sentencias son binarias (al igual que en PSAL1), empleándose la nomenclatura polaca inversa para representar operaciones más complejas.

Al igual que en los compiladores, en la descripción RTM existe una tabla en la que se almacenan los nombres de todas las variables utilizadas. En la tabla de símbolos se almacenan además otros datos, tales como qué bits se utilizan de una variable (en el sentido RTM) en una operación, o qué relación existe entre variables (en el caso de que hayamos dado un nombre a algunos bits de una variable). Este array está formado por estructuras en las cuales los campos más importantes son:

- Tipo de símbolo. Se distinguen 9 tipos:

  a) Memoria.
  b) Variable.
  c) Constante.
  d) Nombre de entidad.
  e) Máscara.
  f) Flag.
  g) Registro temporal, creado por el programa gdbrtm.
  h) Flag temporal.
  i) Constante de más de 32 bits.

- Flag. Este campo es empleado especialmente para el simulador. El programa PSAL1 extrae básicamente dos informaciones de este campo:

  * Si un argumento de una entidad tiene o no modificación REF (incluso aunque no sea considerado en síntesis).
  * Si el símbolo es la definición de un registro hijo. Llamamos registro hijo a aquel que utiliza sólo una parte de los bits de otro registro, que llamamos padre.

- Nombre del registro, memoria o entidad.

- Campo de argumentos de la función .

- En los símbolos asociados a los registros, puntero hacia la sección d) del fichero RTM.MAR, con lo cual podemos saber qué hijos tiene dicho símbolo.

- Campo que apunta a la sección b) del fichero RTM.MAR indicándonos los bits y las palabras del elemento que representa.

- En los registros hijos, campo que nos dice en qué símbolo se encuentra su relación con el registro padre.

- Campo que señala en qué entidad se ha definido el símbolo.

El tipo de símbolo e) está asociado con una "máscara" ya que en la descripción ISPS pueden aparecer expresiones como:

```
a<2:10>,                ! definición del registro a
...
... + a<6:8> next       ! utilización de una parte de la variable a
```

La expresión a<6:8>, aparece en la tabla de sentencias como una operación ("read byte"). En dicha operación se carga en un variable (generalmente virtual) el resultado de dicha operación. Los operandos de "read byte" son una variable (en este caso a) y una máscara. El formato intermedio de PSAL1 permite modelar operaciones en las cuales se utilizan partes de una variable, sin necesidad de introducir instrucciones equivalentes a "read byte". Por esta razón, la operación antes mencionada es asociada a una simple asignación en el formato intermedio de PSAL1.

Por otro lado, la tabla de símbolos contiene un elevado número de variables virtuales introducidos por el parser de ISPS .Estas variables son producidas por:

- El acceso a partes de una variable o de una memoria.
- La transformación de expresiones aritméticas complejas en operaciones en nomenclatura polaca.
- La identificación del acceso a memoria como una operación más.

Estas variables tienen el mismo tratamiento que las definidas explicitamente en la descripción de entrada.

III) Generación de la base de datos .

1º) El grafo de flujos.

En primer lugar vamos a analizar el formato interno que utiliza nuestro sistema. En este formato se definen tres tablas y un grafo. Veamos en primer lugar la información contenida en las tablas.

a) Tabla de entidades.
En este array de estructuras, cada elemento se asigna a una entidad descrita en el algoritmo a analizar. Se indica en dicho elemento el nombre de la entidad, el contexto (entidad en la cual se define y utiliza), el valor del parámetro "PTIME" asociado a dicha entidad y se señala dónde se inicia la definición de la entidad. Esta tabla es muy útil en algoritmos de análisis y síntesis de entidades, ya que permite acceder rápidamente a todas ellas. Así, el primer paso de optimización trabaja la mayor parte del tiempo sobre ella.

b) Tabla de nombres.
En este array de estructuras se incluyen todos los registros que son necesarios para implementar el sistema. Esta tabla está orientada hacia la implementación e indica el tamaño del registro o memoria, cuál es su padre, en qué contexto se define y usa, y su nombre. Para las memorias, se señala igualmente el valor del parámetro INCREMENT. Los flags de esta tabla aportan gran cantidad de información sobre el elemento, por ejemplo, si tiene asociado un hardware especial (ej: si es capaz de autoincrementarse o de desplazar sus bits) o el tipo de elemento (registro, flag, memoria, padre, hijo). En esta tabla, el campo "contexto" es un puntero a la tabla de entidades (todos los "contextos" lo son), mientras que el campo "padre" apunta a un elemento de la tabla de símbolos en la cual se indica la relación existente entre el registro padre y el hijo.

c) Tabla de símbolos.
En ella se indica qué partes de un registro se utilizan. Si la tabla anterior estaba relacionada con la implementación física de las variables de la descripción, ésta está relacionada con las señales de control. Si un mismo registro es referenciado por tres elementos de la tabla (por ejemplo), podemos entender que en el algoritmo se accede a

tres partes del registro y por lo tanto necesitamos tres señales para identificar a que parte del registro nos referimos. En la tabla de símbolos se encuentran también todas las constantes descritas en el algoritmo, así como las relaciones existentes entre registros padres e hijos. Con esta tabla, podemos también saber qué variables son utilizadas como dirección de una memoria.

Los símbolos están ordenados, con lo cual, todos los que se refieren a un registro están juntos (cosa que no pasaba en las tablas de símbolos de la descripción RTM), y el orden de los "registros referenciados" es el mismo que tienen en la tabla de nombres. En esta tabla se indica:

- El elemento de la tabla de símbolos RTM, en donde se definía el símbolo.
- Unos flags que nos indican el tipo de símbolo.
- El registro asociado. Este campo es un puntero a la tabla de nombres.
- El rango de palabras que emplea este símbolo.
- El rango de bits empleados.

Seguidamente se describe brevemente el elemento más importante de esta representación intermedia: el grafo de flujo que modela el flujo de control del algoritmo que se esta estudiando. Los nudos de este grafo se representan en el computador por estructuras, y los arcos entre nudos, por punteros. Existen además, otros punteros que unen los nudos formando una cadena: la tabla de sentencias. Esta lista tiene por objeto poder acceder a cualquier nudo. El sistema PSAL1 genera un fichero en donde se refleja el grafo de flujo, ordenado segun la lista de nudos.

En el Apendice A, se incluye el fichero que el sistema PSAL1 genera como salida al analizar la descripción del computador MARK-1. La tabla de sentancias está compuesta por un conjunto de nudos, separados unos de otros por una línea a trazos (---). En cada nudo, después de indicar el índice por el cual nos referiremos a él, señalamos el número de antecedentes de dicho nudo en el grafo de flujo, el número de operaciones contenidas en ese bloque básico y los índices de los antecedentes. Cada nudo del grafo es un bloque básico formado por una secuencia de operaciones. Esta secuencia aparece entre una línea puenteada (...) y otra a trazos (===). Para cada operación se indica:

- Código de operación. (Así por ejemplo el código 23 es "lee de memoria").
- Flags asociados con el grafo. Indican cuando una operación es la primera o la última del bloque básico, o si se ejecuta en paralelo con otra. Además indican a qué operandos accede la operación.
- Puntero al símbolo del resultado.
- Puntero al símbolo del primer operando.

- Puntero al símbolo del segundo operando.

Cuando una operación se realiza en paralelo con la anterior, la segunda operación aparece precedida por una "p" en la salida de PSAL1.

Después de la línea a trazos (===), aparece la sentencia que provoca la ruptura del flujo, y por lo tanto el fin del bloque básico. En primer lugar, aparece el código de la operación (DECODE, por ejemplo, es el código 301). A continuación se encuentran:

- Indice asociado a la tabla de sentencias del fichero RTM.MAR.
- Puntero cuyo valor depende de la sentencia:

  * Sentencias de tipo IF o DECODE.
    Puntero a la tabla de símbolos. Indica cuál es la variable de control.
  * Sentencias CALL.
    Esta variable indica en qué sentencia se inicia la descripción de la entidad a ejecutar.
  * Sentencias BEGIN.
    Puntero a la tabla de entidades.

- Cuando se produce una división del flujo de control, por un nudo IF, DECODE o DIVERGE, existe otro nudo en el cual este flujo se vuelve a unir. Este puntero señala el nudo en el cual se realiza la fusión. En una sentencia de tipo BEGIN, apunta al nudo END.
- Número de sucesores.
- Puntero a sucesor. En el caso de que haya más de uno se dan cuatro valores:

  * Tipo de condición.
    Así, por ejemplo, el código 1 significa que la condición es un valor.
  * Sentencia sucesora asociada a dicho valor.
  * Valor de la condición asociada con el sucesor anterior.
    En caso de que la condición sea un rango, valor máximo de éste.
  * Valor mínimo del rango.

El grafo de flujo se inicia por un nudo con código "inicio" y finaliza siempre en un nudo marcado con el código "fin". Entre ambos van apareciendo las distintas entidades que forman el algoritmo. Las entidades aparecen unas detrás de otras, ordenadas según su uso, es decir, aparecen antes las que antes se usan.

Existen algunos nudos cuyos bloques básicos están vacíos. Esto puede ser debido, bien al

grafo de flujo, bien al tipo de sentencias. Hay sentencias de control (como "pmerge" y "smerge", que ya vimos en el apartado anterior), cuyo bloque básico asociado está siempre vacío. Otras, como la sentencia "call", dan al bloque básico un sentido especial. En esta sentencia, su bloque básico contiene la asignación de parámetros actuales a los parámetros formales de dicha entidad. El nudo"comienzo de entidad" tambien da un significado especial al bloque básico. En él aparecen los argumentos que tiene la entidad que encabeza.

2º) Parser del sistema PSAL1.

Este programa está formado por más de 2.000 líneas de código C. Su misión es transformar la descripción RTM en el grafo de flujos que acabamos de describir. La descripción de partida esta contenida en unas tablas de valores, dentro de un programa escrito en lenguaje MACRO. La utilización de dichas tablas por un programa escrito en C no es directa. Es necesario , no sólo introducir en el programa en MACRO unas variables (misión que realiza trans.c), sino también cambiar los formatos de otras. A la hora de desarrollar el Parser, aparecieron varios de estos problemas. Uno de ellos es la conversión de strings de un formato al otro. Estas diferencias de formato hacen que esta parte de PSAL1 sea dependiente de la máquina ya que aprovecha características propias de C sobre μVAX. La lectura de la tabla RTM por el Parser se basa en la equivalencia de algunas directivas MACRO con los tipos de variables del C. De esta forma, una tabla de valores con formato WORD en MACRO, se maneja como un array de enteros sin signo en C.

En la tradución de la descripcion RTM al grafo de flujo, el Parser realiza un análisis del algoritmo eliminando todas las entidades que nunca son usadas en la descripción, pero cuya definición aparece en ella. El número de estas entidades es importante si se tiene en cuenta que el programa gdbrtm introduce en la descripción la "definición" de todas las funciones del lenguaje. Estas definiciones constan únicamente de un nudo "begin" y otro "end".

El programa está formado por 23 funciones, de las cuales destacan la encargada de crear las entidades y la que extrae las operaciones de las tablas de la representación RTM :

* La función "crea entidad". Este programa mira si la entidad que se le da como argumento existe o no. Si existe, devuelve la dirección donde se inicia su descripción, sinó la crea. Lo primero que hace, en este caso, es crear los nudos begin y end de la entidad y acto seguido inscribe a la nueva entidad en la tabla de entidades. Después de crear un nudo vacío, como sucesor del nudo begin, llama a la función "siguiente índice", que se encarga de generar el grafo de flujo de la entidad.

* La función "siguiente índice". Esta función recibe como argumentos un nudo incompleto y un índice de la descripción RTM. Lo que hace es ir completando el

bloque básico asociado a dicho nudo. Para poder hacerlo, clasifica las operaciones de la descripción RTM en dos grupos:

- Operaciones de control. Son aquellas instrucciones que rompen la secuencia de código. Por ello tienen que ser la última operación de un bloque básico.
- Operaciones que pueden pertenecer a un bloque básico.

Si el índice de la tabla de sentencias de la descripción RTM que esta función recibe como argumento se corresponde con una operación del segundo tipo, dicha operación es traducida al formato PSAL1 e introducida en el bloque básico. La función devuelve en este caso el índice de la siguiente operación RTM que debe ser traducida.

Si la operación es de control, se traduce e introduce en el nudo. Acto seguido se crean los sucesores que sean necesarios. Existen ciertos tipos de sentencias de control que precisan un tratamiento más complejo. Destaca sobre todo la operación "call". En el bloque básico de una sentencia de este tipo, aparecen las asignaciones de valores a los argumentos de dicha entidad. Pero entre esas operaciones, pueden aparecer llamadas a otras entidades. Estas instrucciones no pueden aparecer en un bloque básico. Por ello se hace precisa una transformación de la secuencia de operaciones, que tiene por objeto obtener un grafo de flujos que verifique las restricciones del sistema. El programa tiene una función, llamada "lock", que se encarga de realizar esta tarea. "Lock" analiza las operaciones que aparecen como argumentos de una función. Supongamos que en la descripción ISPS aparece la llamada:

ent1(a,ent2(b,c),c) next

La función "lock", en este ejemplo, lo primero que hace es detectar que, entre los argumentos de la entidad "ent1", existe una llamada a "ent2". Por esta razón, crea un nudo (llamado "ent1"), en el cual se localizan la llamada a la entidad "ent1", y su respectiva asignación de argumentos. Acto seguido, "lock" busca las asignaciones de valores a los argumentos de la entidad "ent1". Cuando encuentra una (en el ejemplo arg1=a), cambia la variable destino (arg1) por una variable virtual (llamada virt), e introduce la asignación resultante (virt=a) en el nudo que se esta completando.

Acto seguido introduce la asignación arg1=virt , en el nudo ent1. Con esta estrategia se ha conseguido que, aunque en la entidad "ent2" se modifique la variable "a", el valor de "arg1" sea el correcto y se verifiquen las condiciones del grafo. Después, la función "lock" traduce la llamada a la entidad "ent2" (como un nuevo nudo del grafo). Una vez finalizada dicha traducción, examina de nuevo los argumentos de la entidad "ent1".

```
pru:=
    begin
        ** primera **
        a<10:0>,
        b<10:0>

        ** segunda **
        c{main}:=
            begin
            d(a,e(b),b)
            end,

        d(s<10:0>,t<10:0>,r<10:0>):=
            begin
            b=t+s
            end,

        e(h<10:0>)<10:0>:=
            begin
            a=h+1
            end
    end
```

figura 3.7.a. Descripción ISPS

numero	nante	noper	ante oper	rtm	control oper	fin flag	result	nsuc dato1	suc dato2	OPERACION
1	0	0	1							
				342	0	1	3	1	2	INICIO C
2	1	7	1							
				5	21	6	5	0		virt=a
				5	1	7	8	0		h=b
				101	3	5	7	1		a=h+1
				5	1	2	6	0		s=virt
				5	1	3	9	0		t=e
				5	1	4	8	0		r=b
				101	43	8	3	2		b=t+s
				206	19	0	0	1	3	
3	1	0	2							
				205	0	0	1	0		FIN

figura 3.7.b. Grafo de flujo de la descripción ISPS de la figura a)

Como ya no hay llamadas (solo queda la asignación arg3=c), coloca el nudo "ent1" detrás del nudo que llamaba a "ent2", y finaliza la traducción de la llamada a "ent1". En la figura 3.7 aparece un ejemplo de esta transformación.

La descripción ISPS, que aparece en la figura 3.7.a, ha sido optimizada por OPT1, por lo que en el grafo (figura 3.7.b) ya no existen las llamadas a las entidades. Sin embargo, se puede observar que la segunda operación era la asignación de argumentos a la entidad "e", mientras que la siguiente sentencia es el cuerpo de dicha entidad. Al aplicar la estrategia descrita anteriormente, hemos conseguido que la modificación de la variable "a" por parte de la entidad "e", no afecte al funcionamiento de la entidad "d" (cuyas asignaciones de argumentos aparecen en las sentencias 4, 5 y 6).

Al igual que la operación "call", los nudos BEGIN tienen un tratamiento especial. En los bloques básicos asociados a ellos, aparecen listados los argumentos que tiene la entidad. Además, los antecedentes de un nudo BEGIN son todos aquellos nudos que se refieren a la entidad que encabeza, es decir, todos los nudos que la llaman o contienen una operación de tipo LEAVE, RESUME, RESTART, o TERMINATE dicha entidad. Por lo tanto, mirando los antecesores del nudo BEGIN, podemos saber que nudos llaman o interrumpen la ejecución de la entidad que dicho nudo encabeza.

Existen operaciones de los bloques básicos que también necesitan una transformación previa para ser introducidas en el grafo. Destaca sobre todo el acceso a un bit genérico. En ISPS es posible encontrar expresiones como:

b<a>

En ella nos referimos al bit del registro b, indicado por el valor decimal del registro a. En la descripción RTM, esta instrucción precisa dos operaciones y la definición de una variable virtual. En el grafo de flujo es una simple operación.

Además de introducir las operaciones en sus bloques básicos correspondientes, se generan la tabla de símbolos y la de nombres. La filosofía de generación de estas tablas es sencilla: si se necesita un elemento que no existe, se crea. Debe destacarse la transformación que sufren las máscaras. Este tipo de símbolos aparecían en la descripción RTM. En la tabla de símbolos se transforman de forma que no existe un registro por un lado y una máscara por otro, sino un símbolo que representa una parte del registro. El problema del "nombre" de los bits en las máscaras se ha solucionado, evitando la doble definición que aparecía anteriormente.

El Parser comienza su ejecución creando un nudo (el inicial) y acto seguido llama a la función "crea entidad". Esto es debido a que en la descripción RTM, todo el algoritrmo está contenido en una entidad denominada PRELUDE. Una vez el grafo ha sido finalizado (debido a la filosofía del Parser, cuando la función "crea entidad" ha terminado de generar la función PRELUDE, el grafo de flujo ya está completo), se llama a las funciones de optimización, y por último, a la función encargada de escribir en un fichero los datos que contiene el grafo.

3º) El primer paso de optimización.

Ya vimos en un capítulo anterior, que una de las estrategias para optimizar algoritmos era la sustitución de las llamadas a las subrutinas por el cuerpo de éstas. El programa OPT1 implementa este algoritmo.

Con el fin de evitar efectos secundarios indeseados (un incremento excesivo del número de estados), se sustituyen únicamente aquellas entidades que son llamadas una sola vez en toda la descripción. Esto significa, que en el grafo sólo existe un nudo de tipo "call" que llama a dicha entidad. Esta condición es necesaria, pero no suficiente. Para poder reemplazar la entidad deben verificarse además las siguientes condiciones:

a) No debe existir en el grafo una operación de tipo TERMINATE, referida a dicha sentencia.

b) Si dicha entidad tiene cualificador "process", no puede haber código que se esté ejecutando al mismo tiempo que la entidad. A efectos prácticos ésto significa que el nudo sucesor de la llamada a la entidad sea el que indica el fin del grafo.

c) Si en la descripción existen instrucciones del tipo LEAVE o RESTART, referidas a dicha entidad, dichas operaciones deben de estar contenidas en el cuerpo de la entidad.

El programa OPT1 lo primero que hace es analizar todas las entidades para ver si cumplen o no las condiciones. A cada entidad le asigna un código que indica que reglas viola. De esta forma, si una entidad es llamada más de una vez, no vuelve a ser chequeada, con lo cual se ahorra tiempo de ejecución. A continuación OPT1 sustituye aquellas entidades que cumplen las condiciones. No sólo realiza una simple sustitución sino que transforma las sentancias LEAVE, RESTART y RESUME, en sentencias BRANCH (equivalente al "goto" de algunos lenguajes de alto nivel). Estas transformaciones se basan en un análisis del significado de estas sentencias. Así:

* Toda sentencia LEAVE, referida a la misma entidad que la contiene, equivale a un salto a la última sentencia de dicha entidad, es decir, al nudo "end".

* Toda sentencia RESTART, referida a la misma entidad que la contiene, equivale a un salto a la primera sentencia de dicha entidad, es decir, al nudo "begin".

* Toda sentencia RESUME, referida a una entidad que a su vez llama a la que contiene dicha sentencia, equivale a una instrucción LEAVE referida a la segunda. Es decir, si la entidad A llama a la entidad B y en esta última contiene la instrucción RESUME A, dicha operación equivale a LEAVE B.

Lo primero que hace OPT1, es transformar los RESUME. A continuación cambia los LEAVE y los RESTART, y por último coloca la entidad en el lugar que ocupaba el call. Una vez ha extraido las entidades, vuelve a analizar la tabla de entidades, buscando nuevas candidatas a la sustitución. El programa se repite hasta que, al analizar las entidades, concluya que no se puede sustituir ninguna entidad.

Es interesante destacar el método por el cual el programa detecta si una sentencia se encuentra en el cuerpo de una entidad. El programa conoce la dirección del nudo en el cual se encuentra la operación a detectar (un LEAVE, RESUME o RESTART), ya que esta dirección aparece en el "begin" de la entidad a la cual se refiere dicha operación. Cada vez que OPT1 sustituye una entidad, numera todos los nudos, de forma que si un nudo pertenece a una entidad, el valor a él asignado es intermedio el valor asignado al nudo "begin" y el asignado al nudo "end". El programa conoce tanto el valor asignado a la operación en cuestión, como todos los valores asociados a los nudos "begin" y "end". Aplicando la propiedad de la numeración de nudos, anteriormente expuesta, detecta rápidamente en qué entidad está la operación.

El proceso de eliminación de una entidad lleva aparejado varios efectos laterales como:

- Eliminación de dicha entidad de la tabla de entidades.
- Cambio de los contextos en dicha tabla. Las entidades definidas en la sustituida pasan a estarlo en la que la llamaba.
- Cambio de los contextos en la tabla de nombres. Igual que el punto anterior, pero para registros.
- Cambio de función de los nudos begin y end. En ocasiones se transforman en nudos con varias entradas (si existían LEAVE o RESTART dentro de la entidad), pero generalmente se eliminan. Algo parecido le pasa al nudo call. Generalmente se transforma en un bloque básico que contiene una serie de asignaciones, la antigua

asignación de parámetros. En PSAL1, el nudo al cual llegan varios arcos del grafo, tiene asociada una operación smerge. La razón de esta elección es facilitar la detección de lazos.

Después de estas transformaciones, el nudo inicial toma por nombre el de la entidad más externa. Esta es la razón de que el nudo inicial tenga un formato parecido al nudo "begin".

Los resultados de esta optimización han sido espectaculares en la mayoría de los ejemplos que hemos utilizado, correspondientes a sistemas descritos en ISPS en la Universidad de Carnegie-Mellon. Se ha llegado a sustituir el 80% de las entidades. La razón es simple: los diseñadores suelen dividir los algoritmos en pequeñas funciones con objeto de simplificar su comprensión.

4º) El segundo paso de la optimización.

En este paso lo que se pretende es aprovechar al máximo las propiedades de nuestro grafo. Para describir la sentencia:

a=0; b=0 next

se necesitan inicialmente cuatro nudos del grafo. En cambio, utilizando las propiedades de nuestros bloques básicos (la posibilidad de expresar operaciones en paralelo en uno de ellos), se puede expresar como una operación más de un bloque. Con esta transformación, serían eliminados al menos tres nudos en el ejemplo anterior.

Esta optimización ha sido implementada en el programa OPT2. Lo primero que hace OPT2 es buscar los nudos en los cuales la sentencia de control (operación que divide el flujo), sea del tipo "diverge". Entonces comprueba si dentro de esta estructura hay otras sentencias de este tipo. El programa procede de esta manera, ya que en el caso de existir estructuras de este tipo anidadas, lo mas lógico es empezar la compactación por la mas interna, ya que ello permite transformar el resto mas facilmente. Supongamos que el programa ha encontrado una estructura "diverge", que no contiene más operaciones de este tipo. Entonces, OPT2 determina que instrucciones deben ejecutarse en el mismo instante. Acto seguido, busca estas operaciones en los bloques básicos asociados a los nudos sucesores del nudo "diverge". Si las encuentra, las expresa como un conjunto de operaciones que se ejecutan en paralelo y las introduce en el bloque básico asociado al nudo "diverge". Repite el proceso hasta que ya no quedan operaciones en los nudos sucesores, o hasta que encuentra una sentencia de control. La operación de control no es extrafble porque no podemos determinar, a priori, qué sentencias se ejecutarán en paralelo a partir de ella. La secuencia de operación depende, a partir de ese momento, de la operación de control.

```
dir:=
    begin
        ** p1 **
        a<10:0>,
        b<10:0>,
        c<10:0>,
        d<10:0>,
        e<10:0>,
        f<10:0>

        ** p2 **

        pru{main}:=
            begin
                a1();a2()  next
                a=0
            end,
        a1():=
            begin
                a=d;b=d+1 next
                c=a+b next
                e=c*b next
                a4()
            end,
        a2():=
            begin
                e=1;f=3 next
                f=f-a
            end,
        a4():=
            begin
                d=e-c;f=e+c;b=e*c
            end
    end
```

figura 3.8 . Descripción ISPS de dir

Con este programa se consiguen reducciones de hasta el 25 % de los nudos del grafo de flujo. Este algoritmo, aparentemente sencillo, presenta problemas de implementación, ya que la determinación de la secuencia de operaciones se realiza sobre la descripción RTM, que no tiene la simuliud con el grafo que tenia antes de la primera optimización. Han desaparecido del grafo, llamadas a algunas entidades y en su lugar aparece su definición; las estructuras "diverge" tienen otro formato, incluso aparecen operaciones nuevas añadidas

por el "Parser".

En la figura 3.8 se muestra una descripción ISPS. Dicha descripción fue introducida en el sistema PSAL1, produciendo como salida el grafo que aparece en la figura 3.9. Se observa como el programa OPT1 ha sustituido todas las llamadas a entidades por el cuerpo de estas y OPT2 ha compactado el resultado, pasando de 20 nudos a 3 (como se ve en la figura 3.9).

TABLA DE SENTENCIAS

Numero= 3

numero	nante	noper	ante	oper	rtm	control oper	fin flag	nsuc result	suc dato1	dato2	OPERACION
1	0	0	1								
				342	0	1	3	1	2		**Inicio de grafo**
2	1	5	1								
				5	25	4	3	0			**a=d;**
				p	101	1	5	3	1		**b=d+1;**
				p	5	1	7	1	0		**e=1;**
				p	5	11	8	2	0		**f=3 next**
				101	7	6	4	5			**c=a+b;**
				p	102	13	8	8	4		**f=f-a next**
				103	3	7	6	5			**e=c*b next**
				102	7	3	7	6			**d=e-c;**
				p	101	3	8	8	6		**f=f+c;**
				p	103	13	5	7	6		**b=e*c next**
				1	40	4	0	0			**a=0**
				206	14	0	0	1	3		**bloque**
3	1	0	2								
				205	0	0	1	0			**Fin de grafo**

Grafo de flujo de la descripción ISPS de la figura 3.8

figura 3.9

3º) Resultados

Una vez el generador de la base de datos del sistema PSAL1 estuvo implementado, fue sometido a varias pruebas. Tres eran los objetivos buscados:

1º) Determinar si la interpretación de la descripción RTM era correcta.
Como ya hemos comentado, no disponiamos de información directa de la descripción RTM. Por esta razón, para poder decodificarla, tuvimos que someter al compilador a gran número de ejemplos, que cubrieran todas las posibles construcciones semánticas del lenguaje. Asi mismo, se quería comprobar que la captura de datos por parte del programa era correcta. PSAL1 está diseñado de forma que si detecta un caso no previsto da un aviso. La introducción del nuevo caso es sencilla, debido a la estructura del Parser (parecida a la de un analizador sintáctico de un compilador).

2º) Obtener medidas estadísticas de la importancia de las optimizaciones.
En la bibliografía[AhSe86] se indica que la importancia de las transformaciones depende fuertemente del lenguaje de descripción. Por ello, deseabamos conocer cual era la eficiencia de nuestro sistema trabajando con descripciones proporcionadas por otros autores.

3º) Medir el tiempo de ejecución del módulo.
Aunque este parámetro no sea crítico actualmente, es importante saber su orden de magnitud.

Para realizar este test, se escogieron cuatro descripciones de entre las que la Universidad de Carnegie- Mellon distribuye como ejemplos de utilización del lenguaje ISPS. Se trata por lo tanto, de descripciones de sistemas digitales reales. Las descripciones escogidas fueron:

- Descripción del computador VAX 11 / 780.
- Descripción del computador IBM S / 370.
- Descripción del computador PDP11F.
- Descripción del computador RISC II.

Como se puede observar en las tablas adjuntas, las transformaciones implementadas producen importantes simplificaciones en el sistema:

	RISC II	PDP11F	S370	VAX11/780
NUDOS				
PARSER	766	1889	2368	4958
OPT1	617	1609	1760	4164
OPT2	573	1268	1517	3103
ENTIDADES				
PARSER	70	136	246	349
OPT1	18	41	35	69

En estas tablas, se representan el número de nudos que tiene el grafo (primera sección) y el número de entidades que hay en la descripción, después de cada paso del sistema. En resumen, se observa:

a) La primera optimización reduce de forma expectacular el número de entidades. En los ejemplos anteriores, la reducción media es del 79.6%. En la descripción del S/370, dicha reducción llega a ser del 86%. Esta optimización simplifica tambien el grafo. Se eliminan el 18.3% de los nudos como media. La mayor reducción porcentual se produce tambien en el S/370; el 26% de los nudos son eliminados.

b) La segunda optimización reduce el número de nudos del grafo, mediante una mejor utilización del mismo. En este paso se eliminan, en los ejemplos anteriores, un 20.7% de los nudos. En el mejor caso, el VAX 11/780, dicha reducción llega a ser del 25%.

Teniendo en cuenta ambas optimizaciones, se eliminan el 35% de los nudos que había generado el Parser. La descripción que mayor reducción porcentual sufre es la del VAX 11/780, en donde se eliminan el 37.41% de los nudos.

Por último, los tiempos de CPU invertidos por el sistema en transformar los ejemplos propuestos aparecen en la tabla adjunta. El concepto "subtotal", reflaja el tiempo de CPU que necesita el Parser, OPT1 y OPT2. En el concepto "total", se incluye el tiempo que tarda el sistema en escribir el grafo en el fichero de salida.

	RISCII	S/370	PDP11F	VAX 11/780
PARSER	2"	20"	16"	-
OPT1	<1"	3"	6"	-
OPT2	<1"	1"	1"	-
SUBTOTAL	3"	24"	23"	-
TOTAL	20"	1':30"	1':13"	11':19"

Se observa que la descripción del VAX 11/780 (de la que no se dispone de los datos por función) necesita mucho mas tiempo que el resto. De este ejemplo es posible deducir que la versión actual de PSAL1 sobre μVAX tiene problemas para manejar grafos tan grandes. De hecho el "page fault" de dicha ejecución es muy alto (500.000).

IV) Analisis de variables

1 º) Introducción:

Para poder analizar y sintetizar un sistema digital descrito a nivel algorítmico, PSAL1 necesita conocer como se modifican los datos en el sistema para de esta forma asegurar que la máquina resultante de una transformación es funcionalmente equivalente a la inicial. El análisis del flujo de datos proporciona al sistema esta información.

Para PSAL1, dos descripciones son equivalentes cuando ambas manipulan los datos de la misma forma. Esto significa que si la operación OP1, utiliza un dato definido en la operación OP2, después de realizada un transformación, dicha relación debe mantenerse. Es decir, el dato antes reseñado (utilizado en OP1) no puede, en la nueva descripción, depender de una operación que no sea equivalente a OP2.

Como puede verse, la relación entre los datos va a ser el criterio que deben respetar todas las transformaciones. Por ejemplo, el algoritmo de síntesis de variables, que describiremos posteriormente, sólo podrá agrupar dos variables cuando en ningún momento ambas variables contengan simultáneamente un dato que será posteriormente utilizado.

Básicamente, el análisis de variables consiste en determinar dónde se se define un dato y dónde se usa. Este análisis puede realizarse solucionando una serie de ecuaciones del flujo de datos. Por ejemplo, para un bloque básico B, podemos escribir:

$$in[B] = use[B] \cup (out[B] - def[B])$$

siendo:

in[B] = conjunto de variables vivas antes de iniciarse la ejecución de B
use[B] = conjunto de variables usadas en B, antes de ser definidas en dicho bloque
out[B] = conjunto de variables vivas despues de ejecutarse B
def[B] = conjunto de variables definidas en B

Una variable se dice viva, cuando el valor que contiene va a ser posteriormente utilizado. Un dato se dice usado, cuando es empleado en alguna operación. Un dato se dice definido, cuando es resultado de alguna operación.

La ecuación anterior dice que el conjunto de variables vivas antes de la ejecución de un bloque básico B, es igual al conjunto de variables usadas antes de ser definidas en dicho bloque, unido al conjunto de variables vivas después del bloque, excluídas aquellas definidas en el bloque.

El análisis de variables se divide en dos pasos:

1.- Análisis de los bloques básicos
En este paso se calcula para cada bloque el conjunto de variables usadas y definidas, asi como la relación entre variables dentro del bloque.

2.- Análisis global
Una vez determinados para cada bloque los conjuntos use[B] y def[B], se procede a determinar los conjuntos in[B] y out[B], para todos los bloques. Ello conlleva resolver las ecuaciones de análisis del flujo de datos.

2º) Analisis de los bloques básicos

Las técnicas de análisis del flujo de datos en bloques básicos son de uso común en compiladores de lenguajes de programación, sin embargo, su aplicación a la síntesis de alto nivel obliga a definir una serie de conceptos nuevos.

Los lenguajes de programación trabajan con tipos de datos predefinidos. Gracias a ello, el diseñador del compilador conoce de antemano qué clase de datos pueden aparecer. Además, se utilizan las variables como primitivas: generalmente no se puede operar directamente con campos de bits.

En ISPS, no existen tipos de variables. El usuario puede definir para cada variable la anchura en bits que desee (nombrándolos además como quiera), e incluso puede utilizar en una operación el rango de bit que necesite, sin que ello implique operaciones adicionales (desde un punto de vista hardware, lo único que ello conlleva es la existencia de multiplexores a la salida del registro para escoger los datos adecuados, no implicando por ello más estados).

Esta libertad en la descripción complica el análisis. Para resolverlo, el módulo de PSAL1

encargado de esta tarea, ANBLOCK, hace uso del concepto de segmento de bits. Un segmento de bits es un conjunto de bits consecutivos de una variable, que siempre aparecen juntos, es decir, en cualquier referencia que se haga a dicha variable, si existe uno cualquiera de los bits del segmento, aparece el resto.

En el siguiente ejemplo se muestra como se calculan los segmentos de bits de una variable. Supongamos que tenemos una descripción en la cual se utiliza la variable **a**, la cual tiene 10 bits . En la figura 3.10.a se muestran los bits de dicha variable que se emplean en cada uso. De acuerdo con la definición anterior, en dicha variable se diferencian 6 segmentos, los cuales aparecen reflejados en la figura 3.10.b. Cualquier utilización de **a** puede expresarse como unión de dichos segmentos, los cuales constituyen el minimo conjunto que verifica esta propiedad. Por ejemplo, a<7:1> se puede escribir como :

segmento 2 U segmento 3 U segmento 4 U segmento 5

El análisis del flujo de variables, no se realiza por lo tanto sobre variables, sino sobre segmentos de variables.

	segmento 1: bits 10:8
a<3:2>	segmento 2: bits 7:6
a<10:1>	segmento 3: bits 5:4
a<5:0>	segmento 4: bits 3:2
a<7:1>	segmento 5: bits 1:1
	segmento 6: bits 0:0

figura 3.10.a

figura 3.10.b

El número de variables en una descripción ISPS puede ser de por sí muy elevado (por ejemplo, la descripción del computador VAX 11/750 tiene 1.246 variables y 1.678 símbolos). El análisis puede, por lo tanto, ser muy complejo desde un punto de vista computacional. Para evitarlo se introduce el concepto de variable local a bloque.

Una variable se dice local a un bloque cuando nunca está viva fuera de él. Estas variables, por lo tanto, nunca pertenecerán a los conjuntos in[B] y out [B], por lo cual no es lógico incluirlas en use[B] y def[B]. Esta consideración reduce fuertemente el tamaño del problema. Así en la descripción del VAX 11/750, sólo existen 200 variables no locales a un bloque con 406 segmentos.

ANBLOCK realiza en primer lugar un preanálisis, para identificar dichas variables locales a bloques. Una vez hecho ésto, procede a calcular los segmentos de todas las variables, para a continuación realizar el análisis. Para realizar el análisis, el sistema cuenta con una estructura de datos en la cual se indican los segmentos que están en determinado momento activados, y las operaciones en las cuales se activan. Esta estructura se desplaza dentro del bloque permitiendo asignar a cada dato utilizado la operación en la cual se define. Completa, además los conjuntos use[B] y out[B]. Por ejemplo, dado el bloque básico de la figura 3.11, supongamos que la variable **a** es local al bloque. Los datos de partida aparecen en la figura 3.12.

1. a = b+c
2. h = a+i

figura 3.11.

a,b,c,h,i = status desconocido
use[B] = Ø
def[B] = Ø

figura 3.12

En la primera operación se usan **b** y **c**, y se define **a**. Como **b** y **c** no son locales, dependerán de una operación definida en otro bloque. Luego deben de ser incluídas en use[B]. La situacion de las variables y de los conjuntos use[B] y def[B] despues de este análisis, aparece en la figura 3.13.

a = definida en 1
b,c = definidas antes del bloque
h,i = status desconocido

use[B] = { c,b }
def[B } = Ø

figura 3.13

a = definida en 1
b,c,i = definidas antes del bloque
h = definida en 2

use[B] = { c,b,i }
def[B] = Ø

figura 3.14

Procedamos a analizar la segunda operación: En ella se usa el valor de **a** definido en 1, así como el de **i** definido anteriormente. Entonces este uso de **a** se relaciona con la definición mediante un puntero. La variable **i** es incluída en el conjunto use[B]. La situación despues de analizar esta operación aparece en la figura 3.14. Al finalizar el análisis del bloque, el sistema sabe qué variables no locales han sido definidas (**h**) y las introduce en def[B]. Como consecuencia del análisis, el sistema ha extraido los siguientes datos :

use[B] = {**b**, **c**, **i**}
b en la operación 1 depende de una operación externa
c en la operación 1 depende de una operación externa
i en la operación 2 depende de una operación externa
a en la operación 2 depende del resultado de la primera operación
def[B] = { **h** }

Internamente, estas relaciones se representan mediante arcos entre operaciones, por lo cual las operaciones dentro de un bloque básico forman un grafo de flujo de datos. Los conjuntos use[B] y def[B], se representan mediante vectores de bits. Cada bit es asignado a un segmento. Que un bit esté a 1 significa que el elemento pertenece al conjunto y si es 0, que no pertenece. Internamente por lo tanto, use[B] = {1110} y def[B] = {0001}, suponiendo que asignamos los 3 primeros bits a **i**, **b** y **c**, y el último a **h**.

ANBLOCK, al mismo tiempo que realiza el análisis, optimiza la descripción. Tres optimizaciones tienen lugar:

- eliminación de operaciones redundantes
- transformación de operaciones repetidas
- propagación local de constantes.

En la figura 3.15, puede verse un ejemplo de eliminación de lecturas y almacenamientos redundantes. La variable **a** es sustituida en la segunda operación por **b**, dado que ambas tienen el mismo valor. La primera operación puede ser eliminada si el valor de **a**, en ella definido, no es utilizado en el bloque básico y **a** es local (por eso aparece entre paréntesis). Si no es local es preciso contar con el análisis global para poder eliminar dicha operación. Si **b** fuese una constante asistiríamos a un ejemplo de propagación local de constantes.

a = b	(a = b)
...	...
h = a+c	h =b+c
Antes de la optimización	Despues de la optimización

figura 3.15

ANBLOCK realiza además ciertas operaciones en tiempo de compilación, tales como sustitución de operaciones en las cuales aparece el elemento neutro de la operación por transferencias. Un ejemplo aparece en la figura 3.16 (aplicación de la propiedad del

elemento neutro de la suma). Por último, este módulo detecta igualmente operaciones repetidas, transformándolas de forma adecuada, tal y como se observa en la figura 3.17.

a = 0	(a = 0)
...	...
h = a+i	h =i
Antes de la optimización	Despues de la optimización

figura 3.16

a = b+c	a = b+c
...	...
h = b+c	h = a
Antes de la optimización	Despues de la optimización

figura 3.17

En la descripción del VAX 11/750, estas optimizaciones eliminan el 10,4% de las operaciones de la descripción. En descripciones de menor tamaño, como la del PDP8, se elimina el 5% de las operaciones.

3º) Análisis Global

Una vez determinadas para cada bloque básico, los conjuntos use[B] y def[B], el siguiente paso es la resolución de las ecuaciones del flujo de datos, obteniendo como resultado los conjuntos out[B] e in[B].

Existen dos estrategias para lograr este objetivo:

- Análisis del flujo de variables basada en la sintaxis. Esta técnica utiliza ecuaciones especiales para cada una de las posibles construcciones sintácticas del lenguaje.

- Método iterativo. Este método es independiente de la sintaxis.

El primer método es el más eficiente pero tiene el inconveniente que sólo vale para una determinada sintaxis. Con objeto de no restringir la utilidad de PSAL1 a un único lenguaje, se escogio la segunda opción.

GLOANA es el método de PSAL1 que implementa la resolución iterativa de las ecuaciones de flujo de datos. La idea básica del algoritmo aparece reflejada en la figura 3.18.

```
for cada bloque básico B do in[B] = Ø
   while existan cambios en algún in do
      for cada bloque básico B do
         begin
            out[B] = U in[S]                          [1]
                     S sucesor
                     de B
            in[B] = use[B] U (out[B] - def[B])        [2]
         end
```

figura 3.18

Todos los conjuntos se representan como conjuntos de bits, por lo cual, las operaciones que aparecen en el algoritmo pueden realizarse mediante operaciones lógicas. Así la operación [2] puede escribirse:

in[B] = use[B] **OR** (out[B] **AND** (**NOT** def[B]))

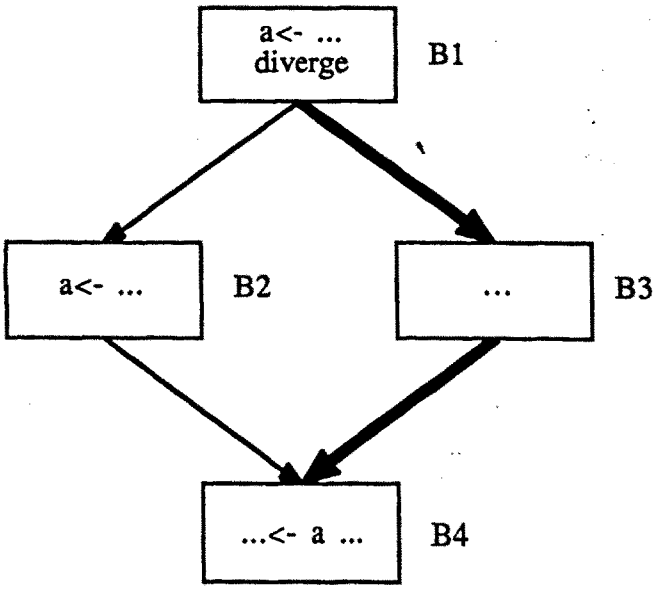


figura 3.19

El principal problema que se presentó al tratar de aplicar dicho algoritmo en el módulo GLOANA, fué la existencia de bloques básicos ejecutándose en paralelo en la descripción. El algoritmo antes reseñado funciona bien con descripciones secuenciales, pero cuando aparece paralelismo, falla.

En la figura 3.19 se observa una descripción en la cual existe paralelismo (los bloques básicos B_2 y B_3 se ejecutan simultáneamente). En el bloque B_2 se define una variable **a** que luego es usada en el B_4. En B_1 dicha variable también es definida. Aplicando el algoritmo anterior, el sistema llegaría a la conclusión de que el valor de **a** utilizado en B_4 es el generado en B_2 o en B_1, lo cual es falso (es el generado en B_2). Ello es debido a que la "necesidad" del valor de **a** es propagado por la segunda rama (con trazo grueso), lo cual conduce a un resultado absurdo.

GLOANA soluciona el problema permitiendo que por cada rama del grafo sólo se propaguen aquellas variables que son usadas o definidas en la rama. Las variables no usadas en ninguna rama se propagan sólo por una de ellas (la primera por ejemplo). Para ello GLOANA define un "vector máscara" que refleja qué variables son usadas y definidas en uno de los procesos que se ejecutan simultáneamente y modifica la ecuación [2] de la figura 3.18 de la siguiente forma:

$$out[B] = \bigcup_{\substack{S \text{ sucesor} \\ \text{de } B}} (in[S] \textbf{ intersección } \text{máscara})$$

El algoritmo anterior debe sufrir una modificación mas para poder analizar descripciones en las cuales existen entidades. En estas descripciones existen tres tipos de operaciones que tienen un tratamiento especial:

- Las llamadas a entidades, en las cuales en bloque sucesor es el comienzo de la entidad llamada. En el caso de que la entidad sea del tipo PROCESS, cada llamada tendrá dos sucesores: el comienzo de la entidad llamada y el bloque básico sucesor de la llamada.

- El final de una entidad, cuyos sucesores son todos los bloques básicos sucesores de una llamada a dicha entidad. En el caso de una entidad del tipo PROCESS, despues de terminar su ejecución, el control no vuelve a ningún punto, por lo que esta entidad no tiene sucesor.

- Las instrucciones de control que producen un abandono o una reejecución de la entidad. Estas operaciones precisan un tratamiento especial, segun la operación. La

operación mas compleja es el RESUME. Ello es debido a que para conocer cual es el bloque básico sucesor de dicha secuencia, es necesario saber que entidades se estan ejecutando en el momento de realizarse el RESUME. Para conocer este dato el sistema construye un árbol en el cual el nudo raiz es la entidad que contiene al RESUME. Las entidades que llaman a este nudo forman el siguiente nivel del árbol. Cada nudo de este nuevo árbol es expandido siguiendo el mismo proceso. La expansión de un nudo se detiene cuando dicho nudo corresponde a la entidad señalada en el RESUME. Ello nos permite conocer cuales son los bloques sucesores del RESUME.

En el análisis de entidades aparecen propagaciones indeseadas, que se pueden evitar impidiendo que por la entidad se propaguen variables que no son definidas ni usadas en ella, y que variables locales a una entidad se propaguen fuera de ella. Para ello se utilizan máscaras con el mismo uso anteriormente descrito.

GLOANA, además, avisa de aquellas operaciones en las cuales el valor definido nunca es usado, y que por lo tanto podrían ser eliminadas. El programa genera además un fichero (nombre.dfa) en el cual se almacenan los valores de los conjuntos in[B], out[B], def[B] y use[B] para todos los bloques de la descripción.

Cabe destacar el bajo número de iteraciones que el sistema necesita para encontrar la solución. Esta depende fuertemente del número de entidades. Así la descripción del PDP8 necesita solo 33 iteraciones. El ejemplo que más iteraciones ha precisado ha sido el VAX 11/750 con 78.

V) Algorítmo de asignación de operaciones a estados.

El módulo SCHE2 es el encargado de asignar las operaciones a estados, teniendo en cuenta las restricciones impuestas por el usuario. Una vez realizada esta tarea, PSAL1 asigna las variables a registros, las operaciones a OUs y por último, una optimización de la asignación realizada por SCHE2. El módulo LASTCOMPAC, por último, realizará una optimización del flujo de control.

Este módulo utiliza la técnica ASAP ("as soon as possible") para realizar dicha asignación. En ella se intenta localizar cada operación en el primer estado en el que sea posible, teniendo en cuenta la restricción impuesta por el diseñador. Una operación puede ser localizada en un estado si no viola las dependencias de datos, ni el limite en el número de unidades operacionales.

Antes de analizar cómo funciona el módulo antes reseñado, veamos brevemente que forma tiene en este momento el grafo de flujo. Este grafo está formado por bloques básicos. Dentro de estos bloques, las operaciones están agrupadas en niveles. Esta agrupación por niveles es realizada por PSAL1 a partir de la descripción ISPS, e indica qué operaciones desea el usuario que se realicen en paralelo. Esta asignación será la elegida en el caso de que el diseñador haya escogido la primera opción (respetar el orden de ejecución de las operaciones). En este caso el algoritmo asignará a todas las operaciones del mismo nivel al mismo estado y pasará a analizar otro nivel. Además calculará el número minimo de OUs que se necesita con vistas a utilizarlo posteriormente.

En el caso de haber seleccionado otra opción, el algoritmo de asignación de estados a operaciones utilizará una tabla, en la cual, se van a colocar las operaciones asignadas a cada estado. Esta tarea será realizada por la función SCHE2_OPER, la cual se encarga de determinar la asignación de estados en un bloque básico. La tabla de estados está inicialmente vacía, es decir, no hay operaciones asignadas a ningún estado. El algoritmo va tomando las operaciones de cada nivel y las va asignando al estado correspondiente. El orden de asignación de las operaciones de cada nivel se determina a partir de cuatro tipos de relaciones establecidas entre ellas:

a) Operaciones independientes. Los estados en las cuales pueden ser asignadas no deben verificar ninguna dependencia especial. Dos operaciones op1 y op2, son independientes, si la variable definida en op1 no es utilizada en op2 y viceversa.

b) Operaciones con dependencia de tipo 1. Dos operaciones op1 y op2 presentan dependencias de tipo 1, cuando el valor definido en op2 es usado en op1. Una dependencia de tipo 1, obliga a que op1 sea localizada en un estado anterior o igual a op2, pero jamás posterior.

c) Operaciones con dependencia de tipo 2. Dos operaciones op1 y op2, presentan dependencia de tipo 2, cuando el valor definido en op1 es usado en op2. Este tipo de dependencia obliga a que op1 sea localizado en un estado posterior o igual a op2.

d) Operaciones anidadas. Estas operaciones deben de ser localizadas en el mismo estado. Este tipo de relación surge cuando el valor definido en una de ellas es usado en otra y viceversa.

En los siguientes ejemplos se muestran estos tipos de relaciones:

- Independencia: a = b + c; h = i + j
- Dependencia 1: a = b + c; b = i + j
- Dependencia 2: a = b + c; h = a + j
- Anidamiento: a = b + c; c = a + j

Estos problemas de orden surgen porque SCHE2_OPER va localizando las operaciones en una tabla en la cual ya están localizadas operaciones de niveles anteriores. Si localizamos las operaciones de un nivel sin tener en cuenta las relaciones anteriores, es posible que se violen las dependencias de datos. Veámoslo con un ejemplo en el cual supondremos que no existe ningún limite en el número de OUs. En la figura 3.20 se representan las operaciones de un nivel. Supongamos que localizamos primero a = b + c. Como los datos de esta operación no dependen del resultado de la anterior (i = t + j), SCHE2_OPER las asigna al primer estado. Cuando más tarde trata de asignar h = a - i, como la variable i depende de la operación i = t + j, asignará la operación al segundo estado. Con ello la descripción resultante sería la que aparece en la figura 3.21. Como se ve, ha violado la dependencia de la variable a, con lo cual la descripción de la figura 3.21 es distinta de la de la figura 3.20.

state 1	i = t + j
state 2	a = b + c ; h = a - i

figura 3.20

state 1	i = t + j ; a = b + c
state 2	h = a - i

figura 3.21

Para encontrar el orden en el cual las operaciones de un nivel deben ordenarse, SCHE2_OPER utiliza el siguiente algoritmo:

1º) Crea una lista con todas las operaciones del nivel.

2º) Crea una tabla en la cual van a ser colocadas las operaciones ordenadas.

3º) Recorre la lista de operaciones, buscando aquella que no dependa del resto. Esta operación será independiente del resto, y por lo tanto puede ser colocada en el primer lugar de la tabla de operaciones ordenadas. Esta tabla está constituída mediante estructuras unidas por punteros, lo cual permite insertar o cambiar de sitio elementos sin más que introducir o cambiar de sitio estructuras. Cada operación independiente ocupa un lugar (una estructura) de la tabla de operaciones ordenadas.

4º) Una vez todas las operaciones independientes han sido extraídas, se inicia el análisis de las operaciones que quedan en la lista. Estas operaciones presentan algún tipo de dependencia entre ellas. La idea es extraer conjuntos de operaciones entre las cuales existe dependencia, pero que son independientes del resto. Para ello se toma

una operación cualquiera de la lista de operaciones y se pasa a la primera posición de la tabla de operaciones ordenadas. Mediante un simple procedimiento iterativo, es posible extraer el conjunto de operaciones interdependientes que contiene el elemento extraído. Este conjunto se representa como una lista de operaciones en la tabla de operaciones ordenadas.

5º) El paso 4 es repetido hasta que todas las operaciones han sido asignadas a una posición de la tabla de operaciones ordenadas. Con ello en dicha tabla tendremos conjuntos independientes de listas de operaciones dependientes.

6º) A continuación se analiza cada una de estas listas. Cuando en una de ellas se encuentra una operación que presenta una relación de tipo 1 con el resto, dicha operación es extraída del conjunto y colocada en la posición inmediatamente anterior al conjunto. Si la relación es de tipo 2, la operación es colocada después del conjunto que estamos analizando. En el caso de que la relación sea de anillamiento o que presente los dos tipos de dependencia, la operación queda en el conjunto.

7º) El paso 6 es repetido para todas las listas.

Veamos con un ejemplo cómo funciona el algoritmo. Supongamos que en la figura 3.22 se representan las operaciones de un nivel.

a = b+c ; b = a + h ; i = a * t ; m = not i ; k = h

figura 3.22

Tras aplicar los tres primeros pasos, en la tabla de operaciones ordenadas sólo hay una operación, k=h, la única independiente. Para aplicar el paso 4º, tomaríamos una operación de las que quedan en la lista de operaciones, y la pasariamos a la tabla de operaciones ordenadas. Podemos tomar a = b + c, por ejemplo. Como el resto de las operaciones del nivel dependen de ella o entre ellas, todas formaran parte de la lista que contiene a la operación anteriormente escogida. Tras aplicar los pasos 4º y 5º, en la tabla de operaciones ordenadas aparecen 2 entradas, tal y como se vé en la figura 3.23.

entrada 1	a = b+c ; b = a + h ; i = a * t ; m = not i
entrada 2	k = h

figura 3.23

Al aplicar el paso 6º a la primera entrada, se extrae en primer lugar m = not i que presenta una dependencia del tipo 1 con el resto de las operaciones. Por esta razón, la operación anteriormente reseñada es incluida en la posición inmediatamente anterior a la lista que estamos analizando. Ello permitirá localizarla antes que al resto, lo que evitará violar las dependencias de datos. Al repetir el paso 6º, se extrae i = a * t (dependencia de tipo 1 con el resto), no pudiéndose extraer más operaciones, ya que las demás están anidadas. En la figura 3.24 se muestra el estado final de la tabla de operaciones ordenadas. Este será precisamente el orden en el cual serán asignadas las operaciones por el algoritmo.

entrada 1	m = not i
entrada 2	i = a * t
entrada 3	a = b + c ; b = a + h
entrada 4	k = h

figura 3.24

Como vemos, las entradas de la tabla de operaciones ordenadas tienen asignadas listas de operaciones. El siguiente paso es asignar a cada una de esas listas, un estado. Evidentemente todas las operaciones de la lista son asignadas al mismo estado. Para ello se siguen los siguientes pasos:

a) Determinación del mínimo estado en el cual la lista puede ser asignada respetando la definición de las variables usadas en las operaciones de la lista. El programa determina cual es el máximo estado en el cual se define una variable que luego es empleada en la lista. Si llamamos minuso a este estado, la lista podrá localizarse como mínimo en minuso+1.

b) A continuación se determina cual es el mínimo estado en el cual la lista puede ser localizada, teniendo en cuenta las variables en ella definidas. Básicamente, lo que hace el programa es encontrar el máximo estado (mindef) en el cual se usa una variable que más tarde será definida en la lista.

c) Se escoge, como mínimo estado en el cual la lista puede ser localizada, al máximo{minuso + 1, mindef}. A este estado se le llama estado actual.

d) A continuación, se comprueba si en la lista se utiliza una unidad operacional que ya es empleada en las operaciones asignadas al estado. En el caso de que la respuesta sea positiva, pasaremos a considerar como estado actual al actual + 1 y repetiremos el paso d). Si la respuesta es negativa y no hemos superado el límite de OUs (si existe),

asignaremos las operaciones de la lista al estado actual. En caso de que necesitasemos más operaciones de las señaladas por el usuario, para poder localizar la operación en dicho estado, pasariamos a considerar como estado actual al actual + 1 y repetiriamos el paso d).

El proceso se repite para cada lista de la tabla de operaciones ordenadas. Como se ve, hemos asignado cada operación al primer estado en el cual esta asignación era posible (técnica ASAP), teniendo en cuenta las restricciones impuestas.

VI) Asignación de variables en registros

El objetivo de este proceso es asociar a cada variable de la descripción algorítmica un registro, en el cual estará alojada la información que contiene la variable. Lógimente varias variables pueden asociarse a un mismo registro, con lo cual se reduce el costo del hardware. Este va a ser uno de los objetivos de la síntesis: lograr reducir el coste del sistema, lo que conlleva reducir el número de registros.

Antes de explicar como el módulo VARSIN de PSAL1 logra este objetivo, debe reseñarse que el sistema distingue dos tipos de variables:

a) Variables agrupables. Sobre estas variables va a trabajar nuestro sistema.

b) Variables no-agrupables, sobre las que no se va a aplicar el procedimiento de síntesis. Se diferencian 4 grupos:

b.1) Memorias. Las variables del tipo memoria (arrays), son localizadas automáticamente en una memoria hardware.

b.2) Variables asociadas a terminales. Existen en la descripción ISPS ciertas variables que están asociadas a líneas de control o de transferencia de información del sistema digital. Estas variables no son registros desde el punto de vista hardware, sino simplemente terminales o señales. Un ejemplo típico son las variables asociadas a las líneas de interrupción externa en microprocesadores. En ISPS no se dispone de un procedimiento standard para identificar estas variables, por lo que el usuario debe indicarle al Parser que variables desea asociar a terminales. Estas variables son ingnoradas por VARSIN.

b.3) Variables especiales. En la descripción ISPS pueden existir ciertas variables

que es preciso que se localicen directamente en un registro. Estas variables pueden servir, por ejemplo, para comunicar procesos concurrentes o para chequear el sistema. El usuario dispone de un mecanismo para indicarle al algoritmo qué variables desea sean automáticamente localizadas en registros.

b.4) Como veremos posteriormente, existen en la descripción operaciones encadenadas. Dos operaciones se dice que están encadenadas, cuando el valor definido en la primera operación es usado en la segunda, no existiendo un registro que lo almacene. En este caso, la variable, lo único que indica es un paso de un dato entre operaciones en un mismo estado, sin necesidad de registro. En la figura 3.25 se puede ver un ejemplo de una variable de este tipo. En dicha descripción, la variable c no almacena ningún dato, es una simple transmisión del resultado de una operación y la otra en el mismo ciclo de reloj.

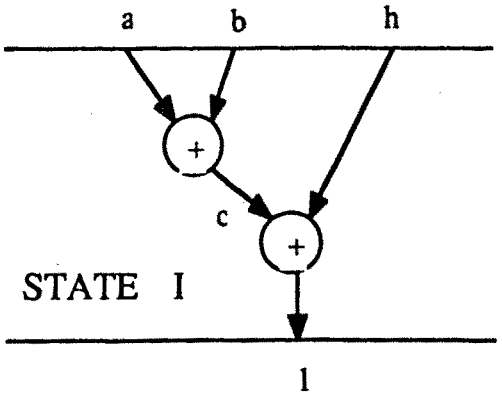


figura 3.25

VARSIN utiliza una técnica de localización global, basada en la teoría de grafos. En ella, las variables de la descripción son representadas como nudos de un grafo. Los arcos del grafo indican que las variables asociadas a los nudos que unen pueden ser agrupadas en un registro, porque nunca estan vivas simultaneamente. El programa trata de realizar una partición del grafo en un conjunto mínimo de agrupaciones. Los nudos asignados a una agrupación verifican la propiedad de que entre cualquier par de nudos de la agrupación, siempre existe un arco. Cada agrupación es asociada a un registro.

El módulo VARSIN se encuentra dividido en tres partes:

a) Generación del grafo (módulo GENTALIVE)
b) Partición del grafo en agrupaciones (módulo VARSINTESIS)
c) Agrupación de variables en registros (módulo UNEVAR)

Para poder generar el grafo de dependencias, GENTALIVE hace uso de la información generada por el análisis del flujo de datos. Partiendo de un grafo (en el cual los nudos son las variables localizables) en el que existen todos los arcos posibles, el algoritmo va eliminando aquellos arcos que unen variables que almacenan simultáneamente un dato que será posteriormente empleado.

El programa no realiza el análisis a nivel de variables (nombres) sino que lo realiza a nivel de segmentos. Esto significa que dos variables no pueden ser agrupadas cuando existen segmentos en cada una de ellas activas simultáneamente.

Supongamos que las sentencias de la figura 3.26.a representan el contenido de un bloque básico. Supongamos que para dicho bloque, los conjuntos in[B] y out[B] toman los valores que aparecen en la figura 3.26.b. El grafo de dependencias tendría en este ejemplo siete nudos (a, b, c, j, k, m, e) e inicialmente cada par de nudos estaría unido por un arco.

[1]	a = b+c	in[B] = { b , c , k , m }
[2]	e = b+a ; j = k+m	out[B] = { j , e , m }
	a)	b)

figura 3.26

Como b y c pertenecen a in[B], ambos contienen datos simultáneamente (durante la primera operación). Por ello, el arco {b , c} es eliminado. Lo mismo ocurre a los arcos {b, k}, {b,m}, {c, k}, {c, m} y {k, m}. En la primera operación se observa que se usa c y ya no se vuelve a utilizar. Como en esa operación se define a, entonces a y c nunca contienen un dato al mismo tiempo, luego el arco {a, c} se mantiene. Este mismo razonamiento se podría aplicar al arco {b, e}. En la figura 3.27 aparece el grafo resultante.

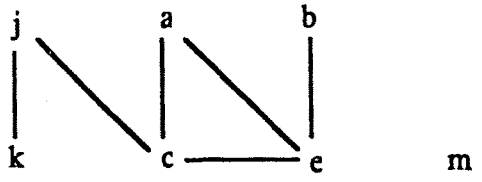


figura 3.27

Una vez generado el grafo, VARSIN busca el conjunto mínimo de agrupaciones en los cuales sería posible dividir el grafo. Dado que la partición en agrupaciones es un problema

NP-completo, el programa utiliza un procedimiento heurístico para encontrar la mejor partición. Este algoritmo utiliza como criterio de coste el número de vecinos comunes y excluídos por un arco. Dado el arco {a,b}, se dice que dicho arco tiene un vecino común, si existen los arcos {a,c} y {b,c}, siendo a, b y c nudos del grafo. Decimos que el arco {a,b} excluye un arco, si existe el arco {a,c}, pero no el {b,c} o viceversa. En el ejemplo anterior, el arco {a,c} tiene un vecino común (e) y excluye un arco (el {c,j}). VARSIN implementa el siguiente algoritmo:

a) Para cada arco del grafo, calcular el número de vecinos comunes y excluídos.

b) Tomar el arco que más vecinos comunes tenga. En el caso de que haya varios, el que menos excluya.

c) Agrupar los dos nudos de dicho arco, en un único nudo. Los arcos excluídos por el arco escogido son eliminados del grafo, mientras que se crean arcos entre el nuevo nudo y los vecinos comunes.

d) Considerar los arcos que contienen el nuevo nudo. Para cada uno de éstos arcos calcular los vecinos comunes y excluídos. Pasar posteriormente al paso b). Si el nuevo nudo generado en el paso c) no tuviese más arcos, se comprueba si aún existen arcos en el grafo. En caso positivo se vuelve al paso a), si no, el algoritmo finaliza su ejecución.

El programa genera como resultado un grafo sin arcos, en el cual cada nudo está formado por un conjunto de nudos del grafo original. Cada nudo es un agrupación y será asociado a un registro. Veamos como se aplica este algoritmo al grafo de la figura 3.27. El primer paso consiste en calcular los vecinos comunes y excluídos de cada arco. La figura 3.28 recoge estos datos.

arco	vecinos comunes	vecinos excluidos
{ a,c }	1	1
{ a,e }	1	1
{ b,e }	0	2
{ c,e }	1	2
{ c,j }	0	3
{ k,j }	0	1

figura 3.28

Como se podrá observar el nudo m ya es un nudo aislado. Aplicando el paso b) escogemos el nudo {a,c} (podríamos haber cogido igualmente el {a,e}). A continuación aplicamos el paso b) generando un nuevo nudo (que llamaremos {a,c}). El grafo resultante aparece en la figura 3.29.

Repitiendo el proceso se obtiene el conjunto de agrupaciones que aparece en la figura 3.30. El sistema necesita por lo tanto, cuatro registros.

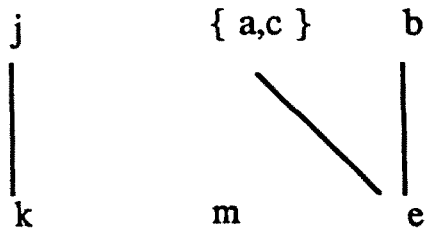


figura 3.29

{ a,c,e }
{ k,j }
{ b }
{ m }

figura 3.30

El último paso consiste en reformar la descripción del sistema digital incluyendo estos resultados. El módulo UNEVAR se encarga de modificar las tablas de nombres y símbolos, así como las referencias de las operaciones, de forma que a partir de este momento ya se puede afirmar que cada nombre del grafo de flujo está asociado a un registro. VARSIN no sólo realiza una síntesis de variables, sino que al mismo tiempo que genera el grafo, calcula cuál es el mínimo número de registros que la descripción necesita. Para ello VARSIN calcula el número máximo de variables que se encuentran activadas en un punto de la descripción. El número de registros necesario para implementar el algoritmo debe de ser obviamente mayor o igual que aquel.

Además de compactar las variables, VARSIN analiza la utilización que se hace de las variables en los procesos concurrentes. El programa no permite que una variable sea usada y definida en dos procesos concurrentes, salvo que la variable sea del tipo especial (no-agrupable). PSAL1 trata con ello de evitar violaciones de las dependencias entre datos que podrían producirse en los procesos concurrentes. El módulo GENTALIVE, escribe en un

fichero el grafo de dependencias si el usuario así lo solicita.

VARSIN calcula el mínimo número de registros en los cuales pueden localizarse las variables de la descripción. En el caso del PDP8, el algoritmo proporciona una solución con 7 registros, que coincide con la implementación mínima a partir del análisis de dependencia de datos.

VII) Localización de unidades operacionales

Una vez las variables han sido asignadas a registros, el siguiente paso es localizar las operaciones. El hardware necesario para realizar una operación es costoso, por lo cual se intenta diseñar circuitos capaces de realizar el mayor número de operaciones con el mínimo hardware. Esto permite una mejor utilización del hardware disponible, una minimización de las interconexiones y por otro lado una reducción de costes. A estos módulos, capaces de realizar diferentes operaciones, se les denomina unidades operacionales (OU's).

Sin embargo, no todas las operaciones de la descripción son localizadas en OU. Existen tres clases de este último tipo:

a) Operaciones de carga o almacenamiento, tales como accesos a memoria, concatenaciones, lectura de registros, etc, que no necesitan ser localizados en OU's.

b) Operaciones realizadas en los registros. Existen tres tipos de operaciones, cuya implementación está unida al registro sobre el cual operan:

- operaciones de incremento-decremento.
- cálculo del complemento-1 del registro.
- operaciones lógicas en las cuales el registro en cuestión es resultado y operando al mismo tiempo.

Antes de realizar la síntesis de OU's, el programa detecta estas operaciones, y las señala de forma que no sean tenidas en cuenta en la asignación posterior.

c) Operaciones asociadas con el control. En la descripción del sistema digital existen operaciones cuyos resultados son generalmente utilizados posteriormente en la descripción. Otras, por contra, generan datos cuya única misión es controlar el funcionamiento del sistema. Estas operaciones relacionadas con el control no tendrían

en principio por qué estar implementadas en el "data-path", podrían estarlo en la unidad de control.

El flujo de control es modificado en la descripción por ciertas variables. Por ejemplo, en una operación condicional (un if), se realizarán unas acciones si una determinada variable, toma un valor y otras en caso contrario. Las decisiones siempre se toman según el valor de una variable de la descripción. En la operación de control dicha variable aparece como dato, e indica que, según su valor, el flujo del programa tomará un camino u otro. Estas operaciones de control permiten tener en cuenta que las variables correspondientes son utilizadas para controlar la máquina. PSAL1 no localiza en OU's aquellas operaciones susceptibles de ser localizadas en la unidad de control. Estas operaciones son de tipo lógico, y se caracterizan porque su resultado sólo sirve para determinar el próximo estado. Por lo tanto, el cálculo de dicha variable y su uso tienen lugar en el mismo estado, con lo cual, dicha variable de control es una variable no asociada a un registro (caso de operaciones encadenadas).

PSAL1 dispone de un módulo denominado DEFCHAIN, el cual se encarga de detectar estas operaciones encadenadas asociadas con el control. Antes de realizar la síntesis de variables, DEFCHAIN analiza la descripción buscando dónde se definen las variables que se usan en las operaciones de control. Si dicha operación verifica ciertas condiciones la operación que define dicha variable es marcada como no localizable. Si la variable de control sólo es utilizada en ese caso, dicha variable es señalada como no-localizable en registro (sólo sirve para transferir datos en un estado y, por lo tanto, representa una conexión interna del sistema). Si la variable es usada en otras operaciones se define una nueva variable, no-localizable en registro. Para que una operación asociada a una variable de control sea no-localizable debe de verificar:

1.- Ser una operación lógica o una concatenación de variables.
2.- El valor definido en la operación sólo debe de ser usado en la operación de control.
3.- Las variables usadas en dicha operación no serán redefinidas una vez realizada ésta en el bloque básico que se esta analizando.

Dado que la operación de control es la última operación del bloque básico que se realiza, la condición 3 tiene por objeto que, al encadenar la operación escogida y la de control, en el último estado del bloque, no se violen las dependencias de datos.

Sólo se permite encadenar una operación. La razón es simple: la localización de operaciones en la unidad de control reduce el tamaño de las OU's en el data-path y elimina una cantidad importante de variables a la hora de asignarlas a registros, pero a costa de aumentar el tamaño de la unidad de control. Se trata con ello de reducir el número de operaciones que se

realizan en dicha unidad. Un criterio para reducirlo puede ser localizar como máximo una operación por cada condición. El análisis de los resultados nos demuestra que con ello se simplifica notablemente el data-path, no complicándose mucho la unidad de control. Hemos observado que no es posible, muchas veces, integrar más operaciones. En el PDP8, DEFCHAIN detecta 9 operaciones encadenadas. Ello le permite eliminar 4 variables de 33 (12%). El resto (5 variables) son variables definidas para la ocasión.

Una vez ha quedado establecido qué operaciones son localizables en OU's y cuales no, ANALU (el módulo de PSAL1 encargado de esta tarea), inicia su ejecución. El objetivo de este programa es obtener el conjunto mínimo de unidades operacionales, que agrupando todas las operaciones de la descripción, minimiza una cierta función de coste definida por el usuario.

ANALU utiliza un método iterativo/constructivo para obtener esa partición de las operaciones. La técnica consiste en ir localizando las operaciones paso a paso, de tal forma que en cada paso se localice aquella que más minimiza el coste.

Estas técnicas iterativas/constructivas pueden presentar efectos laterales. Así, cuando se localiza una operación en una OU, se está impidiendo localizar en esa unidad otras operaciones, y aunque el coste de inserción de la primera operación sea el más bajo, el coste de integración del conjunto puede ser demasiado alto. Denominamos coste de integración, a la penalización resultante de introducir una operación en una OU, medida en unidades definidas por el usuario.

Dos operaciones no pueden ser incluídas en una misma OU cuando ambas se están ejecutando al mismo tiempo. ANALU hace uso de este hecho (la existencia de conjuntos excluyentes) para evitar los efectos laterales antes reseñados. El programa, en vez de localizar operación a operación, localiza un conjunto de operaciones excluyentes en cada paso. Dos operaciones pertenecen a uno de estos conjuntos cuando se excluyen, o lo que es lo mismo, se ejecutan al mismo tiempo. Si consideramos que el sistema se encuentra en un estado determinado en cada ciclo de reloj, podremos concluir que cada estado representa un conjunto de operaciones excluyentes. Además, en el mejor de los casos, el número máximo de operaciones que aparecen en estos conjuntos, nos indica el número mínimo de OU's que son precisas para implementar el sistema. ANALU trata de localizar los conjuntos excluyentes en ese número mínimo de OU's minimizando una cierta función de coste, que tiene en cuenta no sólo el coste de la operación, sino el de las conexiones que hay que establecer y la similitud de la nueva operación con las ya localizadas.

Lo primero que hace ANALU es calcular los conjuntos de operaciones excluyentes. Este

proceso presenta ciertas dificultades en el caso de procesos concurrentes, ya que cualquier operación que use uno de los procesos, excluye a todas las operaciones de los otros. Al mismo tiempo que se calculan los conjuntos, se determina el número máximo de operaciones que aparecen en ellos. Además, se almacena el conjunto máximo de operaciones excluyentes. Es importante señalar, que los conjuntos excluyentes dependen totalmente de la descripción de entrada. Gracias a ello, el usuario puede indicar que operaciones pueden realizarse en OU's diferentes, al mismo tiempo que señala al sistema el número máximo de OU's permitidas (que será igual al numero máximo de operaciones no localizables en OU's que aparezcan ejecutandose en paralelo en la descripción inicial).

El algoritmo de síntesis sigue los siguientes pasos:

1.- Se asigna a cada OU una de las operaciones del conjunto con mayor número de elementos. A esta asignación se la denomina asignación por defecto.

2.- Para cada conjunto de operaciones excluyentes se calcula su coste de integración en las OU's.

3.- Se selecciona aquel conjunto de operaciones excluyentes que tenga menor coste. Si un conjunto tiene un coste de integración 0, se selecciona automáticamente.

4.- Se asignan los conjuntos excluyentes a las OU's. Si aún quedan conjuntos, el algoritmo continúa en el paso 2. En caso contrario, termina.

El algoritmo, para calcular el coste de integración asociado a cada conjunto de operaciones excluyentes (o estado), procede de la siguiente forma:

1.- En primer lugar se calcula el incremento de coste que se produciría al localizar cada operación del estado en una de las OU's. Ello produce una tabla en la cual se refleja para cada operación y OU's, su coste de integración,

2.- Una vez han sido calculados dichos costes, se asocia a cada operación, la OU en la cual su coste de integración es menor.

3.- En ocasiones hay varias operaciones asignadas a una misma OU. Para estas operaciones se calcula la variación de coste que se produciría al asociar dicha operación a una OU distinta y cuyo coste de integración sea el mínimo (excluyendo, claro está, la OU en la cual está actualmente integrada la operación). La operación para la cual dicha variación sea mínima, modifica su OU asignada.

4.- El paso anterior se repite hasta que cada operación ha sido asignada a una OU diferente.

El coste de integración de un conjunto es la suma de los costes de integración de las operaciones que lo forman.

Una vez seleccionado un estado, se localizan las operaciones en las OU's de acuerdo con las asignaciones antes realizadas. Para ello, el algoritmo guarda, para cada conjunto de operaciones excluyentes, la asignación operación-OU de menor coste.

La función de coste es el medio que tiene el diseñador de especificar la mayor o menor dificultad de implementación de distintas operaciones en una OU. Ello otorga al sistema una gran flexibilidad y extiende su aplicación a la resolución de problemas de diseño diferentes. La función de coste es una función escrita en C, que recibe de ANALU unas tablas, en las cuales se le indican las operaciones (y la anchura de la palabra en bits) actualmente localizadas en las OU's así como las conexiones ya establecidas entre registros y OU's. El sistema envía a la función una operación y espera que ésta le devuelva el coste que a juicio del usuario esa integración tiene.

La función actualmente implementada, tiene en cuenta no sólo el coste intrínseco de las operaciones y de las conexiones, sino también la similitud entre operaciones. La similitud es una función que mide la cantidad de hardware que dos operacions pueden compartir. Así por ejemplo, el coste de integración de una adición en una OU en la cual hay localizada una substración puede ser menor que si se localiza en una OU sin operaciones, ya que en el primer caso ambas operaciones comparten la mayor parte de las puertas.

Con un ejemplo se muestra como funciona ANALU. Consideremos el conjunto de estados y operaciones asociadas reflejado en la figura 3.30. Supongamos que el resto de los estados de la descripción carecen de operaciones susceptibles de ser localizadas en OU's.

```
        ...
S1 : v1 = v2 + v3, v4 + v1 * v2
        ...
S2: v5 = v2 - v3, v6 =  v4 or v3
        ...
S3: v1 = v5 * v3, v2 + v1 and v2, v5 + v5- v6
        ...
```

figura 3.30

```
OU1  v1 + v5 * v3
OU2  v2 = v1 and v2
OU3  v5 + v5 - v6
```

figura 3.31

El estado, con un mayor número de estados es S3 con 3. Así pues, el número mínimo de OU's que se precisan es tres. A continuación localizamos (asignación por defecto) cada operación de dicho estado en una OU. En la figura 3.31 se muestra el resultado de dicha localización. Supongamos que se ha definido la función coste tal y como aparece en la figura 3.32. La similitud suma/resta muestra la reducción de coste que significa agrupar dichas operaciones en una OU, debido a las puertas que comparten

coste de una conexión	3		**OU1**	**OU2**	**OU3**
coste de una operación lógica	1	**op1**	13	7	8
coste de una suma/resta	10	(v1=v2+v3)			
coste de un producto	100	**op2**	12	103	112
similitud suma/resta	8	(v4=v1*v5)			

Figura 3.32

Figura 3.33

El siguiente paso consiste en determinar el coste de localización de las peraciones en cada estado. En la figura 3.33 aparecen los costes de integración de las operaciones del estado S1. Veamos a continuación, como se calcula el coste de integración de la operación op 1 en la OU_3.

a) En primer lugar, en dicha operación se realiza una suma. Como este tipo de operación no está localizado en la OU_3 (sólo hay una resta), la asignación de esta operación incrementará el coste en 10 unidades. Como entre la suma y la resta existe una similitud de 8 unidades, el incremento de coste debido al tipo de operación sería de 2 unidades.

b) El objetivo del algoritmo no es sólo minimizar el coste de las operaciones, sino también el de las interconexiones. La operación op1 precisa una conexión entre el registro v2 y el primer operando de la OU. Como tenemos localizada una conexión entre dicho registro y el segundo operando de la OU y op1 es conmutativa, conmutamos sus datos, con lo cual reduciremos el coste de conexión (consideraremos v3 + v2 en vez de v2 + v3). Al hacer esta suposición necesitamos sólo 2 nuevas interconexiones: una entre v3 y el primer operando de la OU_3, la otra entre la salida de dicho módulo y v1. Por todo ello, el coste debido a las interconexiones es de 3 x 2 = 6 unidades.

De esta forma, el coste de integración de op1 en OU_3 resulta de 8 unidades. En la figura 3.33 se reflejan los costes de integración de todas las operaciones del primer estado. Se observa que la asignación que añade menos coste es aquella que localiza op1 en OU_2 y op2 en OU_1. El coste asociado al estado es por lo tanto de 19 unidades. Al repetir el proceso para el otro estado se comprueba que el coste asociado es de 10 unidades. Como este segundo coste es menor, se selecciona el segundo estado como candidato a ser localizado, asignando a continuación sus operaciones a las OU's. Repitiendo el proceso, se selecciona S1 en el siguiente paso, tras lo cual ya no quedan más estados, por lo que la localización se da por concluída. Las asignaciones realizadas por el algoritmo, aparecen en la figura 3.34.

OU1	OU2	OU3
v1 = v5 * v3	v2 = v1 and v2	v5 = v5 - v6
v6 = v4 or v5		v5 = v2 - v3
v4 = v1 * v2		v1 = v2 + v3

Figura 3.34

- Una vez ha finalizado la asignación, ANALU escribe en un fichero (llamado nombre.ous) los resultados de ésta. En el fichero se indica:

  - Número de OU's necesarios para realizar las operaciones
  - Operaciones y anchura en bits de cada operación
  - Conexiones establecidas entre las OU's y los registros del circuito.

En la figura 3.35 se pueden observar los resultados que ANALU proporciona para el PDP8.

Para señalar qué conexiones se han establecido, se utiliza una tabla para cada OU. Cada línea de la tabla indica qué conexiones se establecen entre el registro que aparece al final y la unidad operacional. Esta relación se expresa mediante un número con 3 campos: el primero está asociado a la salida de la OU, el segundo con el primer puesto de la OU, y el tercero con el segundo. Un 0 en un campo significa que la conexión no existe, un 1 que si existe y una x que las interconexiones son intercambiables.

ASIGNACIONES OPERACION-ALU REALIZADAS POR PSAL
Modulo: pdp8
Numero de ALUs: 2

ALU 1

operacion	bits
eql	12
neq	12

ALU 2

operacion	bits
slr	13
srr	13
+	13
lss	12
geq	12

INTERCONEXIONES DEFINIDAS
formato 123... 1 salida de la alu,2 input 1,3 input 2
valores 0 .. conectado,1 .. desconectado,x .. optativo

ALUs ----> 0 ... Maxalu

000 100	PC
000 101	CPAGE
100 110	EADD
100 110	T75
010 110	LAC
001 001	CONSTANTES

figura 3.35

VIII) Compactación de operaciones

Una vez han sido asignadas las operaciones a OU's, las variables a registros y las conexiones a buses, se realiza una última compactación de las operaciones. LASTCOMPAC es el módulo encargado de esta optimización de la asignación de estados, antes de obtener la descripción DDL. Este módulo optimiza en primer lugar la asignacion operación estado en un bloque básico, para a continuación realizar una reasignación global.

Las optimizaciones realizadas en este modulo tratan, por ejemplo,de sacar ventaja del hecho de que en un ciclo de reloj se pueden asignar acciones que sólo se realizan si se cumple una condición (if-condicional). Estas acciones condicionales necesitan varios bloques básicos para poder ser expresadas en el grafo de flujos y sin embargo son asociadas a un mismo estado.

Por otro lado, ya hemos señalado que existen operaciones encadenadas asociadas con el control. El algoritmo identifica estas operaciones y las asigna al último estado del bloque básico. El resto de las operaciones son reasignadas (si es preciso), para mantener la coherencia.

```
...
x = b + c next
DECODE x = >
   begin
      0 := h = k,
      1 := h = k + 1,
      2 := h = k - 1
   end next
y = t + k next
...
```

figura 3.36

El siguiente paso es realizar la asignación global de estados. De esta función se encarga COMPACBLOQUE. Este algoritmo va recorriendo el grafo asignando estados según el flujo de control. Dentro de un bloque básico el orden ya está especificado, por lo cual, lo que hay que determinar es el offset a añadir a las operaciones del bloque básico o, dicho de otra forma, como el menor estado en un bloque básico es el 0, el estado que será asignado al primer nivel del bloque básico. Para ello COMPACBLOQUE recuerda las operaciones actualmente asignadas al último estado analizado. Si las dependencias entre datos lo permiten, el primer estado del bloque básico será asignado al último estado analizado. Una vez las operaciones del estado han sido asignadas, el programa determina qué operaciones han sido asignadas al último estado del bloque, para de esta forma continuar la asignación.

En la figura 3.36 vemos una parte de una descripción ISPS. Su representación en forma de grafo de flujos aparece en la figura 3.37.

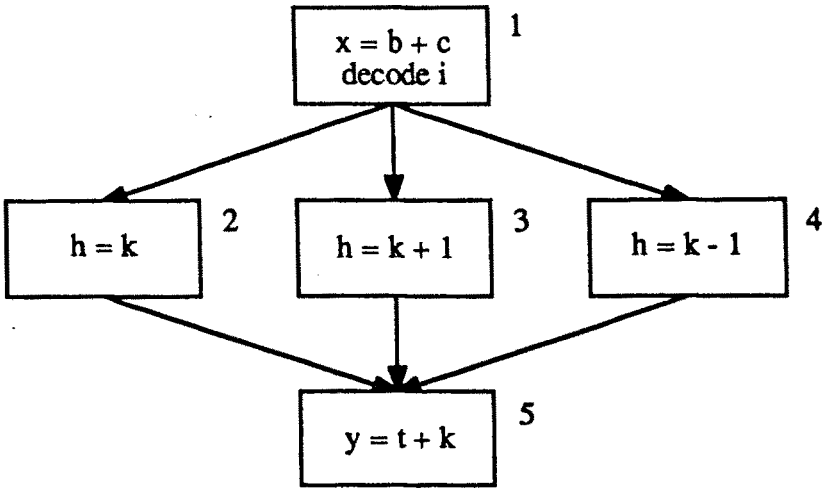


figura 3.37

El algoritmo, despues de asignar las operaciones del bloque 1, llega a la conclusión de que en el último estado debe existir una operación (x = b+ c) y un "decode". Llamaremos a este conjunto, conjunto inicial. A continuación se analizan los bloques 2, 3 y 4, siendo asignados al mismo estado que el conjunto inicial..

Supongamos que en el conjunto inicial en vez de la operación x=b+c apareciese la expresión k=b+c. En este caso la variable k sustituye a la x, por lo que las operaciones h=k, h=k+1 y h=k-1, no podrían ser localizadas en el mismo estado que k=b+c. En este caso el "decode" y las tres operaciones antes reseñadas, serán asignadas al próximo estado (figura 3.38). A continuación se procede a analizar el bloque 5.

```
state i     k = b + c, -> state i+1
state i+1 ! i # 0 h = k   # 1 h = k + 1   # 2 h = k - 1
```

figura 3.38

El algoritmo identifica también qué ocurre cuando aparecen instrucciones de control, como "return", llamadas a entidades, etc.

En ISPS, las instrucciones **if** no admiten **else**. El algoritmo de compactación de bloques identifica de forma automática estas situaciones. Existen ciertos casos en los cuales las instrucciones que aparecen despues del **if** jamás serán realizadas si se realiza el **if**. En la figura 3.39 se describe un ejemplo de estos casos. En la figura 3.39.a se observa que si no

se ejecuta el if, la instrucción h = i + j no se ejecuta. Luego la descripción puede ser reescrita como aparece en la figura 3.39.b (estamos abusando un poco de la sintaxis ISPS, ya que en este lenguaje no existe el else). Esta reasignación permite aprovechar mejor las posibilidades sintácticas de DDL minimizando la implementación final.

```
if a =>                          if a =>
    begin                        begin
        ...                              ...
        restart ...                      restart ...
    end next                         end
    h = i + j next               else
                                     h=i+j next

        a)                               b)
```

figura 3.39

En el caso del PDP-8, este módulo asocia las operaciones en 33 estados. Si se tiene en cuenta que el grafo de flujo del PDP-8 tiene 119 sentencias y 108 operaciones se concluye que han sido asociadas una media de 3.27 operaciones por estado, sin contar la de control, y de 6,89 si dichas operaciones se consideran.

IX) Generación de la Descripción DDL

Una vez las operaciones han sido localizadas en estados, el siguiente paso es escribir la descripción DDL resultante. ESDDL es el generador de código DDL del sistema PSAL1. Este programa va agrupando las operaciones en los estados en los cuales han sido asignados. El programa no sólo busca y escribe en cada estado las operaciones a él asignadas, sino que también calcula cual será el próximo estado. Con el fin de mejorar la descripción, ESDDL realiza una optimización del flujo de control. Con ésta, el sistema trata de evitar que existan estados que sólo sirvan de puente entre estados, sin que la máquina haga nada (en la figura 3.40 se puede observar un ejemplo; el estado S2 ha sido eliminado, ya que sólo servía para pasar el control a otro estado). La eliminación de estados innecesarios es realizada por ESDDL en dos pasos:

- Señalización de estados candidatos a ser eliminados
- Cálculo del próximo estado de forma optimizada.

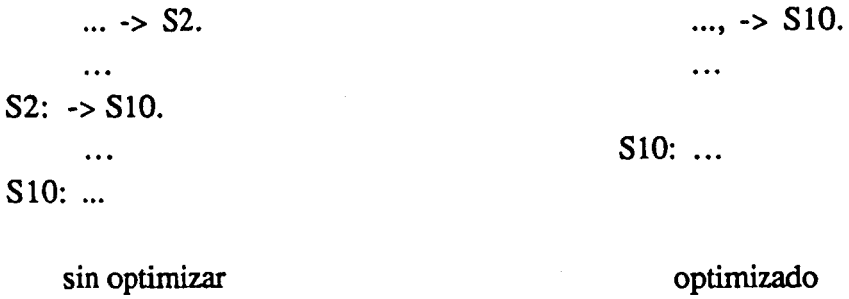


figura 3.40

En principio, cualquier bloque en el cual no se realice ninguna acción (tan sólo sirva para trasladar el control entre bloques básicos), es señalado con una marca, de forma que dicho bloque sea ignorado cuando ESDDL busque las operaciones de un estado. Evidentemente si el programa no encuentra acciones, no escribe el estado. Por esta razón, en las descripciones DDL, los estados no se nombran secuencialmente, sino que faltan ciertos valores.

La segunda tarea que realiza el programa para optimizar el control, es realizar una búsqueda exhaustiva del próximo estado. Cuando ESDDL ha escrito todas las acciones asociadas a un estado, calcula cual es en principio el próximo estado. Si resulta que el próximo estado ha sido marcado en la tarea anterior, el programa considera como próximo estado el estado que se ejecuta despues del estado marcado y repite el proceso. Lógicamente, el proceso se acaba cuando se encuentra un estado en el cual se realicen acciones. ESDDL además de optimizar el control, realiza una reconversión de las entidades. En DDL, no existe ninguna primitiva que guarde paralelismo con las entidades de ISPS. ESDDL realiza la implementación de las entidades asociando a cada una un registro. En él, va a ser almacenado un valor, que indica el punto en el cual la ejecución debe continuar una vez concluida la subrutina. La llamada a la entidad se modela como una transferencia de estados, en donde el próximo estado se corresponde con el primero de la descripción de la entidad. Una vez la ejecución de los estados de la entidad ha finalizado, la utilización del valor del registro asignado a la entidad, permitirá al sistema conocer cuál es el próximo estado.

En la figura 3.41 se puede observar un ejemplo. En la descripción ISPS se realizan dos llamadas a la entidad a. En la descripción DDL aparecen dos estados, cuyo sucesor será el primer estado de la descripción de la entidad (SBEGIN). Como se puede observar, antes de saltar a la entidad, una variable asociada a la entidad (_a) es cargada con un valor, dependiendo de cual de las dos llamadas se ejecuta. Tras finalizar la ejecución de los estados de la entidad a (en el estado SEND), el valor de dicho registro es utilizado para conocer el

próximo estado.

```
        ...                                  ...
a() next                                Si: ... , _a <- 0D1, -> SBEGIN.
        ...                                  ...
a() next                                Sj: ... , _a <- 1D1, -> SBEGIN.
        ...                                  ...
! definicion de la entidad a           < CO > estados de a.

a():=
        begin                           SBEGIN: ...
        ...                                     ...
        end                             SEND: ..., ! _a
                                                #0 -> Si+1
                                                #1 -> Sj+1.
```

figura 3.41

Cuando describimos el funcionamiento del modulo de asignación operación-estado, señalamos que el programa realiza una identificación automática del **else** en algunos **if**. ESDDL reconoce esa identificación y la escribe (hasta entonces solo se había reflejado en la asignación de estados). ESDDL siempre que accede a una memoria, lo hace a través de un terminal (un terminal especial dedicado únicamente a direccionar una memoria).

El módulo de asignación operaciones-estados, genera para el PDP8 una descripción con 33 estados. ESDDL, produce una descripción con sólo 31, por lo cual han sido eliminados durante la compactación, el 6.45 % de los estados.

X) El sistema PSAL1

El conjunto de módulos anteriormente descritos constituyen el sistema de síntesis PSAL1, el cual desde una descripción algorítmica en ISPS propone una arquitectura a nivel de transferencias de registros descrita en DDL.

El sistema PSAL1 esta constituido por alrededor de 15000 líneas de código C. Actualmente esta implementado en un μVAX II, bajo sistema operativo VMS.