

The Yorktown Silicon Compiler System

R. K. Brayton

R. Camposano

G. De Micheli

R. H. J. M. Otten

J. van Eijndhoven

7.1 Introduction

7.1.1 History of the Yorktown Silicon Compiler

The Yorktown silicon compiler is a result of an ongoing project at the IBM T. J. Watson Research Center at Yorktown Heights, N. Y. The goal of this project is to study all aspects related to automating the synthesis of digital systems to be integrated on a semiconductor carrier. The compiler grew out of two initially independent clusters of activities in that project in the early 1980s.

One of these involved logic synthesis and minimization and began with research in automating the synthesis of logic to be implemented with cascode current switch circuits. In the summer of 1981, the scope of the logic synthesis activity was widened considerably:

- The table method of Ashenurst and Curtis for decomposing switching functions was modified to incorporate knowledge about the arrival times of input signals.
- The observation that unate Boolean functions could be efficiently manipulated led to the idea of recursively expanding functions with respect to variables chosen such that the cofactors are as close as possible to unate functions. The efficiency of simplification, complementation, tautology, and other important operations in logic minimization was considerably improved by applying this idea to most functions encountered in practice.

- Algorithms used in the best heuristic minimization programs available at that time, MINI and PRESTO, were programmed. Better algorithms for several tasks were discovered and implemented. Many of the improvements in efficiency were obtained by applying the result of the previous task, the unate recursive paradigm. The activity resulted in a new package for two-level logic minimization called ESPRESSO.

- Activity in the decomposition of logic functions, aimed at different kinds of cascode circuits, resulted in the discovery of the kernel theorem. This allows one to use a subset of all common divisors for locating all nontrivial common subexpressions. Thus, the extraction of common subexpressions for use as an intermediate became a much more manageable task.

At the end of that summer, it was not difficult to see that a powerful package for multilevel logic synthesis, quite different from conventional local transformation methods, was within reach. This approach would combine many of the ideas discovered during the summer and would be basically independent of the final target technology chosen. By the end of 1982, a very powerful package of algorithms for logic synthesis and minimization existed and, for reasons of attracting other users, was called a *logic editor*.

Simultaneously, but on a smaller scale, a high-level layout design procedure was being developed. The net effect was a refinement and ordering of a given hierarchy. Special routines were invented that, based on eigensolutions of connection matrices, would try to discover a hidden structure and translate this into a slicing structure. The slicing constraint was imposed mainly because of its consistency with hierarchy concepts and its complexity-reducing effects on many optimization and decision procedures that were to follow. This constraint hardly could be disadvantageous if individual cells ("leaves" in the hierarchy) were being assembled according to shape and pin position constraints derived from a top-down floor plan design procedure. This required a characterization of the set of feasible shapes and an efficient algorithm that could determine the optimum geometry of a floor plan using only feasible module shapes. These algorithms were implemented and combined by the end of 1982. In addition, a version of the recently invented "annealing method" was implemented for handling a variety of objective functions in designing floor plans or for various placement problems.

Following development of the logic editor and a program for creating preliminary cell environments, the natural next step was the implementation of an assembler for logic macros. In selecting a technology for implementing the logic, we concluded that the domino circuit was the best option. In addition to the well-known advantages of complementary MOS circuits, the domino circuit added speed, potential to implement a complex single function in a simple discharging network, and the ability to adjust the driving power behind its output signal. An investigation into its properties with respect to testability turned out favorable as well. The problems with this family, reported in the literature, were overcome by IBM's advanced production techniques and special features that could be incorporated in the basic circuit, if necessary. Experience in handling such a large logic family

did not exist, and the technique of local transformations does not lend itself to synthesizing logic in so large a family. Thus, only through a package such as the logic editor could the potential of domino logic be reasonably investigated.

Early exercises in implementing an arithmetic-logic unit (ALU) exclusively with domino circuits had demonstrated the advantages of using linear transistor arrays to realize the discharge network. Thus, algorithms for obtaining area-efficient linear transistor arrays formed the core procedures for generating domino gates. To put these gates together in a module that implemented a multilevel logic specification, we developed the pluricell layout style. At a first glance, pluricell modules seem to be polycell or standard-cell-type layouts. They use the columnar arrangement of the individual gates, with many of the interconnections made in the channels between the columns. Both realize the supply and clock lines by abutment of gates. A pluricell design procedure, however, does not copy the layout of its gates from a cell library. The gates are constructed algorithmically when called for in the result of the logic synthesis procedure. Also, the interconnections between gates bordering different channels are realized in second-level metal. The placement procedure has the task of optimally using the resources (wiring layers) in obtaining a time- and area-efficient layout. The availability of the second-level metal gives the macro better accessibility for global signals. The pluricell layout style requires a channel router that makes signals available in the first-level metal, where they must contact a second-level metal track.

The logic editor with a technology-dependent tail for domino logic, combined with the pluricell layout program, constituted what we called a *macroassembler*. Immediately after its completion, in September 1983, its results were compared with PLA realizations and gate array estimations for the same logic. All the comparisons turned out in favor of the macroassembler [Bray84c].

The next step was the development of a logic language. The first version of this language was available in the spring of 1984. In the meantime, due to experience from demonstrating the macroassembler at several sites, we decided to use the tools developed so far as a basis for a complete environment for designing digital systems. Operated in a fixed sequence, starting with a high-level description of the system and ending with production of data that uniquely determine the instructions for fabricating an integrated circuit, such a set of procedures is called a *silicon compiler*.

Once this decision was made, we needed to add the additional procedures necessary and to select an example as a test case. The 801 architecture was chosen because it was complex enough to exercise all the tools and, because it was the first RISC-type architecture, was of general and growing interest. The logic language was extended with a capability for interpreting tables, which made the specification of control modules easier, and extracting "don't cares" led to better implementations. To communicate the hierarchy of the system, a simple hierarchical specification language was devised,

giving us, in effect, a register transfer level of input. Every combinational module was specified by using the logic language. All this was considered a preliminary solution for exercising the lower-level tools. An input format more acceptable for other users was planned as a future activity.

Simultaneously, the global wiring of the floor plan design tool was rewritten in a compiler language to accelerate the process, and at the same time it was provided with more powerful heuristics. Also, additional work was being done to use the annealing algorithm in a fully automatic way. With improved understanding of the method, more improvements were added over the next two years.

By the end of 1985, we had specified additional interfaces and data structures in order for the various modules of the compiler to communicate with each other. Some verification tools were added for gate-level simulation at the system level, and some preliminary timing analysis was made available. Finally, during 1986, a fully automatic timing optimization strategy was implemented using logic resynthesis, resizing of output devices, and replacement of critical modules. In addition, a behavioral-type input specification was adopted and implemented as the main input method for the compiler. This used the previous register transfer level input as an intermediate format to communicate with the lower-level components of the compiler.

7.1.2 The Yorktown Silicon Compiler

We use the word *compiler* in a way similar to its use in a software context. A compiler processes a program in a language that is expressive in describing hardware. It is reasonably high level in the sense that the programmer need not know assembly code or the details of the final object code—in this case, the gates or the target technology, and the layout design rules. The program in the high-level language is automatically compiled into a valid chip image.

The Yorktown silicon compiler (YSC) project has three major goals. The first goal is to achieve automatic silicon compilation from a behavioral high-level description into a chip image. The compile time should be short and negligible in comparison to the time spent by the designer in describing the chip as a program. This allows the user to experiment with different chip architectures and evaluate in real time the circuit tradeoffs. The second goal is to be able to design chips that are competitive with custom manual design in performance and silicon area. This is made possible by using several optimization techniques that are coupled to the entire top-down synthesis process. As a final goal, the Yorktown silicon compiler provides a design environment whose backbone is the automatic synthesis operation. This environment includes verification techniques and alternative synthesis and optimization paths. Therefore, the Yorktown silicon compiler can be used as a workbench by the CAD researcher to develop new synthesis or optimization techniques and seek their validation in the final image of the chip.

As much as possible, the Yorktown silicon compiler has been kept independent from the target technology. Only the circuit layout phase depends on the implementation technology. This phase is the equivalent of the code generation step in the software context. At present, the target technology is a CMOS implementation. Different compiler back-ends have been implemented to explore different flavors of CMOS styles, such as dynamic and static mode of operation.

The Yorktown silicon compiler views a VLSI chip as a hierarchy of data sets, which we call *modules*. Each module represents a portion of the chip. The level of abstraction of the representation, as well as the hierarchy, are refined during the compilation process. Eventually, each module will correspond to a rectangular area of the chip and will represent the geometries of the masks needed to fabricate that part of the chip.

The hierarchy of modules is constructed by the compiler from the high-level system description, which is given by the circuit designer in the form of a program. The hierarchy may be suggested by the system designer by structuring the program appropriately. In the beginning, each module is a subroutine describing a finite automaton. Therefore, the entire circuit to be implemented can be seen as a hierarchy of finite automata. The first creative step of silicon compilation, called *structural synthesis*, is the processing and optimization of this hierarchy with the goal of devising an interconnection structure of data operators without memory, data storage elements, a set of input/output (I/O) ports, and, possibly, special function units. At this point, this structure determines another hierarchy: the leaf modules, called *cells*, are elementary building blocks of the chip and can be classified into the different types specified previously. Data operators without memory are implemented by combinational logic circuits, and they are referred to as *combinational macros*. Storage elements include various kinds of registers and arrays of read/write memory units (referred to as *register files*). The input/output ports are implemented by off-chip drivers and receivers. Finally, special function units may be described as calls to special procedures, provided by the user. The structural synthesis step optimizes the circuit timing performance by means of architectural-level decisions, such as the selection and the partitioning of data operators and the high-level determination of the control structure.

The Yorktown silicon compiler processes the representation of the combinational modules to obtain an optimized multilevel logic representation. This step is called *logic synthesis*, and it aims at determining a circuit implementation of the combinational module that minimizes the active area and the worst-case switching time. The logic synthesis step provides an estimate of the area needed to implement the combinational module. At this point, it is possible to estimate the timing performance of the chip and determine which interconnection nets are critical for the global circuit performance.

The chip floor plan is designed after the logic synthesis step using the information about the area estimates of the modules, the interconnections among modules, and the critical nature of these interconnections. Modules

are characterized by constraints on their shape during the floor plan design phase. The functional hierarchy is then refined into a *slicing structure*—each module is associated with a rectangular area (see Figure 7.1). The estimated physical position of each module is then used to estimate the length of the interconnections, which will help to obtain a better estimate of the chip timing performance. If the design does not meet the required

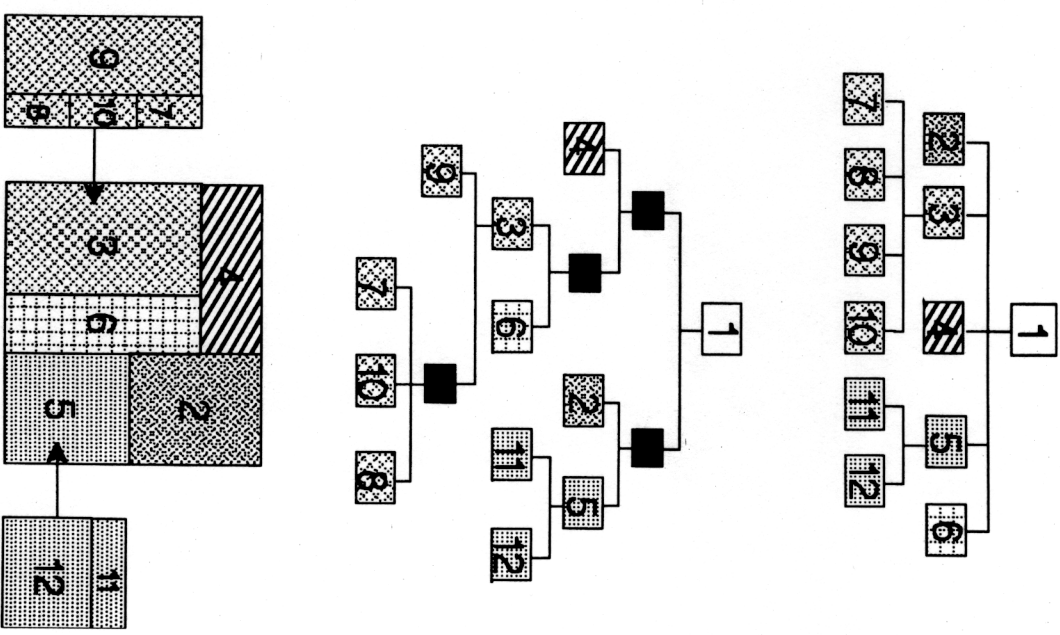


Figure 7.1 Slicing tree and corresponding floor plan.

specification, the circuit is "tuned" by adjusting appropriate parameters corresponding to the final device sizes, by iterating the logic synthesis step, or by repositioning some of the modules.

The circuit layout comprises two steps: (1) the generation of the images of the geometries of the cells and (2) the global chip assembly. The first step is accomplished by the use of *macroassemblers*, which have the ability of building each logic block or storage block in a rectangular area using an estimated length-to-width ratio and indications for the pin positions. Thus it can adjust the implementation of each module to the global floor plan requirements. The combinational logic modules are in pluricell layout style—that is, each multiple-input multiple-output combinational logic function is implemented as an interconnection of generalized gates, with each gate implemented by a linear transistor array. Gates are stacked in rows separated by wiring channels. The structure is determined by the shape constraints, and the microcell position is optimized by a placement procedure. The geometry of each gate is obtained and optimized by a procedure according to its logic functionality. The gates are interconnected by wiring the channels that separate them (see Plate 1, page 000). Other macroassemblers are used for generating the image of the registers and the register file as rectangular blocks, also according to their shape constraints. Eventually, in the global chip assembly step, the macrocells corresponding to each rectangular area are placed on the chip image and interconnected using the channels that separate them.

These steps of the top-down synthesis are represented in Figure 7.2. Circuit verification tools coexist in this environment and support functional-level and gate-level circuit simulation, propagation delay analysis, and connectivity verification.

The strength of the Yorktown silicon compiler, in comparison to other approaches to automated synthesis, relies on the following facts:

1. Synthesis starts from a behavioral description; therefore, it qualifies as a silicon compiler (as opposed to a module generator).
2. Logic synthesis uses powerful algorithms with global optimization routines (as opposed to local rule-based transformations).
3. The floor planning creates preliminary environments for the modules in a top-down sequence without fixing the geometry of the cells to be designed by special procedures.
4. The combinational modules are synthesized as multilevel logic macros laid out in pluricell style. This gives superior results in terms of area and switching time, as compared to two-level logic PLA and gate array implementations.
5. The timing performance of the circuit can be optimized by tuning the circuit in different ways: finding optimal device size, finding an appropriate logic structure of the combinational modules, and by influencing the positions of the macrocells by timing considerations.

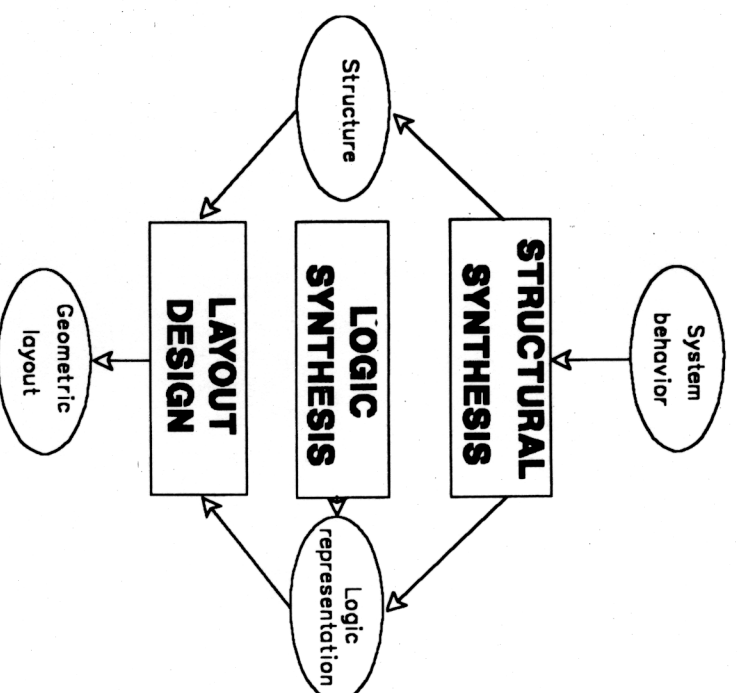


Figure 7.2 Synthesis in the YSC system.

The techniques that are implemented in the major components of the Yorktown silicon compiler will be discussed in Section 7.2. Section 7.3 will explain how the compiler achieves the automatic synthesis of a chip from high-level specifications. And, finally, Section 7.4 will present the results of our experience in using the compiler and the experimental evidence of the strength of the YSC approach in producing competitive chips.

7.2 The Functional Units of the Yorktown Silicon Compiler

7.2.1 Hardware Description

The starting point for the synthesis of a digital system is a system description. This representation captures the system behavior and provides the necessary information to design the hardware as well as the software system to be used in conjunction with the hardware. The description must be such that the hardware and software development can be performed independently. In principle, the hardware description should not imply any implementation choice and, therefore, should be purely behavioral. In

practice, the semantics of most high-level representations used for hardware synthesis tend to suggest some architectural implications: there is a wide spectrum of high-level representations ranging from purely behavioral to merely structural ones [Gold85]. In the case of digital computers, the hardware specification is traditionally given in the *principles of operations*, which is a description in a natural language.

Since the high-level behavior of digital hardware systems can be described by algorithms, there are several advantages in describing them as programs in a programming language. First, the hardware description can be processed by silicon compilers in a manner similar to software programs processed by software compilers. Experience in software compiler technology has been key to the development of the front-end silicon compilers. Second, program descriptions can be simulated before proceeding to synthesis. Third, programs can be used as synthetic documents describing hardware systems and as replacements for the principles of operations texts, just as program descriptions of classical algorithms have replaced textual descriptions.

The choice of the programming language involves a tradeoff between generality and power of expressing hardware. In principle, programming languages such as C, Pascal, or APL could be used. The main advantage is the popularity of these languages: compilers and interpreters already exist for these languages. The main disadvantage in using software programming languages is the lack of constructs that are typical of hardware programming languages (e.g., instructions for handling I/O pins). A solution to this problem is to extend the capabilities of software programming languages. For example, Brenner has developed the Yorktown Logic Language (YLL) as an extension of the APL notation. The YLL will be fully described later in this section.

In the Yorktown silicon compiler, the systems to be designed are specified in the Yorktown Intermediate Format (YIF). The definition of the library elements to be used in synthesis is done in the YLL. Whereas YIF can be used to describe behavior involving both combinational and sequential aspects of a system, YLL is restricted to purely combinational, memoryless circuits.

The introduction of a format (YIF) rather than a language for the initial system specification separates the real silicon compilation task from the front-end processing of a programming language. The former task has all the essential challenges of hardware automatic synthesis; the latter task is analogous to the well-known problems of parsing, syntax checking, and semantic analysis. Like several programming languages, such as C, PLI, or Pascal, YIF has imperative semantics. Statements are evaluated sequentially according to the specified control flow. Consequently, programming languages with imperative semantics can be mapped into a YIF representation in a straightforward manner. For practical reasons, the YSC system supports the compilation of only one language into YIF. We have chosen the V language because of compatibility with other research activities within IBM and the possibility of having access to several examples [Bets85].

The YLL is used to describe the data operators in the module library as well as the control constructs that result after structural synthesis. The need for a module library derives from the frequent use in VLSI design of common modules, such as arithmetic-logic units (ALUs), shifters, and so forth. These modules are often very critical to the final hardware performance, and their behavioral description must have a predictable structure that can be highly optimized. However, instead of using just a small number of predefined library modules, an open book philosophy is followed. The library consists of parameterized YLL lines describing data operators (e.g., an addition). During synthesis, several of these lines are composed to form a YLL program that will yield, after logic optimization and layout, a hardware module that meets exactly the specific needs of the system being designed. Due to its powerful constructs for the description of combinational logic, YLL is very well suited for this purpose.

The Yorktown Intermediate Format (YIF)

YIF is a format to specify the behavior of a digital system at the algorithmic level. The main goal in designing YIF was to allow an easy compilation of programming languages with an imperative semantics into YIF while capturing all the information necessary to start a design. YIF is somehow inspired by the internal form defined in [Camp85]. Related work can be found in [Knap85], [McFa78], [Orai86]. In the YSC system, YIF is the input to structural synthesis.

YIF contains:

- *The definition of the variables necessary to describe the system behavior.* These include variables in the sense of a programming language. Also their observability in the final hardware can be described: architected registers might be specified as registers already at the beginning; variables that represent the interface to the outside world (pins) must be specified.
- *Operations that use the variables as inputs and outputs.* Operation sequences are specified giving the control flow. The basic control flow elements are sequence, conditional branching, and iteration. The operators can be chosen according to the application and must be described in YLL in the operator library.
- *A mechanism for hierarchical and modular design.* An operation can be specified by another YIF description. This corresponds to a procedure call in the behavioral program.

The primary representation of YIF is a textual format with a very simple syntax. This form is used to interface different programs, such as the V compiler with structural synthesis. Structural synthesis keeps an internal representation of YIF in matrices and lists.

A YIF description $Y = (V, S, R, R_i, R_o)$ consists of:

- *A set of operations — V.* Operations represent atomic computations (e.g., and, addition, shift-left, etc). One element, $v_1 \in V$, called the first operation, must be given.
- *A set of variables — S.* They are equivalent to variables in a program. A

- variable j in YIF has a type(j) $\in$ {array, external, internal, register}. The type indicates whether a variable is an array, whether it represents a port to the outside (external), whether it is an internal variable, or whether it is an architected register. During synthesis, the internal variables might be converted automatically into registers. Thus, the initially specified architected registers are a subset of the registers of the final design. (An initial specification might contain no architected registers at all.)
- A *precedence relation* $-R_i \subseteq V \times V$. It defines the control flow for the operations:

$(v_1, v_2) \in R_i$ iff v_1 before v_2

"before" here means "immediate predecessor," as implied by imperative semantics. The relation R_i is not reflexive.

- An *input relation* $-R_i \subseteq S \times V$. This indicates the variables used as inputs (are read) by each operation.
- An *output relation* $-R_o \subseteq V \times S$. This indicates the variables used as outputs (are written) by each operation.

The pair (V, R_i) can be seen as a digraph, the so-called *precedence graph*. Node v_1 is said to be the (immediate) predecessor of node v_2 and vice versa; v_2 is called the (immediate) successor of v_1 . *Conditional branches*—that is, the selection of one among many possible successor operations—are represented by a node with more than one successor, leading to a fork. *Iterations* or *loops* are represented by circuits in the graph. The operational semantics of YIF is defined as follows:

Operations are executed one at a time, starting with the first operation v_i . The next operation is a successor of the executed operation. If there is more than one successor, exactly one of them is chosen depending on the value in one of the variables. An operation v writes a new value to all variables s_j for which $(v, s_j) \in R_o$. It uses the values in the variables s_j such that $(s_j, v) \in R_i$. A variable retains its value until it is overwritten.

For synthesis it is necessary to eliminate the cycles in R_i . This is done by removing some pairs L (edges in the precedence graph) such that

1. The transitive closure A of $(R_i - L)$ is a partial order
2. The graph remains connected so that $\forall v \neq v', \exists v'' (v, v'') \in A$ that is, all operations are still reachable from v_i .

Since a fork in the graph represents the alternative execution of one of the successors, all operations not ordered in A will be called *alternative*.

YIF also includes a mechanism for modular and hierarchical design. One YIF description is referred to as a module. *Modules* are synthesized separately and connected after structural synthesis ("module binding"). To relate the modules to each other a "module call" operation is used. It replaces an atomic computation. The semantics is as in imperative languages: control is passed to the called module until it is ready and returns. There are some restrictions however. Global variables that can be referenced in different

modules are not allowed. All the information flow between the calling and the called module must be stated explicitly by listing the corresponding variables as inputs and/or outputs in the module call operation. In the called YIF description, these variables are of type external. These "parameters" are passed by name; they will be connected during module binding, also passing the required control information.

In addition to this information, YIF includes some attributes such as the width (number of bits) of a variable; its dimension, if it is an array; an index identifying an operation uniquely; the type of the operation (addition, shift, etc.); constraints on the operations; and so forth. A special constraint, the *map constraint*, specifies that one or more operations are to be mapped onto a specific hardware module. Thus, it is possible to specify the implementation of different operations using the same hardware. During synthesis, map constraints will be generated automatically.

Although the semantics of an imperative programming language and YIF are basically the same, some extensions (e.g., handling constraints) and restrictions (e.g., limiting the allowed data types) are necessary. During compilation, variables in the language are mapped on variables in YIF, expressions are decomposed into single operations, and the control constructs in the language are represented by the edges in the graph (V, R_i) . The edges L are obtained directly from the iteration constructs such as DO UNTIL or DO WHILE. Procedure calls are replaced by module calls. The compiler must monitor the passing of all parameters as inputs and/or outputs in the module call; it may also need to monitor the scope of global variables according to the semantics of the language.

To illustrate these concepts, consider the self-explanatory V program in Figure 7.3, which describes a possible instruction fetching stage of the 801 processor.

The V compiler converts this program to the YIF code given partially in Figure 7.4 and plotted in Figure 7.5. The YIF code starts with a SYMBOLTABLE containing the used variables followed by their type and dimensions, the first pair of numbers being the width and the second pair being the array dimension limits. Variables T1 and T2 were generated by the compiler. The NODES represent the operations:

- INDEX is a unique number identifying them.
- TAG represents the kind of operation (e.g., CBR means conditional branch, SO means single operation, etc.).
- OPERATION gives the function of the operation (e.g., TR means transfer, CON means concatenation, etc.).
- LINE and COLUMN give the position in the original program, in this case token numbers indicated in the listing.
- INPUTS is the list of input variables for this operation and their used width and array dimension.
- OUTPUTS is the list of output variables.
- PREDECESSORS are the predecessors.
- SUCCESSORS represents the successors and their CONDITIONS (i.e.,

```

MODULE FETCH (BPC:IN, BRANCH:IN, IBUS:IN, IRE:IN,
1
20
33
36
39
43
49
55
61
68
74
80
83
90
96
102
105
117
123
126
129
135
140
144
148
154
156
160
164
170

EXTERNAL FETCH:
  PCT:OUT, IRT:OUT, IA:OUT);

DCL BPC BIT(32), /* New PC */
  BRANCH BIT(1), /* Use BPC */
  IBUS BIT(32), /* Instruction Bus */
  IRE BIT(1), /* Instruction ready */
  DCL PCT BIT(32), /* PC */
  IRT BIT(32), /* Instruction register */
  IA BIT(32), /* Instruction address */

INTERNAL FETCH:
  DCL PC BIT(32), /* PC for prefetch */
  IR BIT(32), /* Instruction register */
  OLDPC BIT(32), /* PC for output */

BODY FETCH:
  PCT := OLDPC; IA := PC; IRT := IR;
  WHEN (IRE || BRANCH) /* Catenation */
  CASE 2: /* 10 */
  CASE 0: /* 00 */
  DO UNTIL IRE LOOP ENDDO;
  CASE 1,3: /* X1 */
  PC := BPC;
  IA := PC;
  DO UNTIL IRE LOOP ENDDO;
ENDCASE:
  IR := IBUS;
  OLDPC := PC;
  PC := PC+4;
END FETCH;

```

Figure 7.3 Basic instruction fetch in the 801 processor.

the value at the output of this operation to select the corresponding successor).

In the graphic representation, nodes represent an operation. Inside the nodes, the INDEX is given. To the left, OPERATION is found. INPUTS are listed above the nodes. OUTPUTS are below them and to the right. If an operation has more than one successor, the CONDITIONS that apply to each of the edges are also listed. The edges in $L = \{(v_8, v_7), (v_{10}, v_9)\}$ are not drawn. They represent the two UNTIL loops. The loop bodies are empty represented merely by the "dummy" operations v_7 and v_9 , which also have no inputs and outputs (indicated by asterisks). Operation v_{11} is also a dummy operation that "closes" operation v_4 , which represents the CASE statement.

The main features of YIF not yet described are the specifications of constraints and of parallelism. *Constraints* include area, time, power, and mapping. They are given for sets of operations (e.g., a set of operations should be implemented in less than a given area or by some specific library cell). A similar approach can be found in [CaKu86].

Parallelism is introduced using an additional relation, $R_p \in V \times V$. But since we restricted synthesis initially to systems that can be represented by one finite automaton, it is not used yet. The parallelism achieved automa-

```

SYMBOLTABLE:
  IRE EXTERNAL 1 1 0 0;
  IBUS EXTERNAL 1 32 0 0;
  ...
  IR VARIABLE 1 32 0 0;
  T1 VARIABLE 1 2 0 0;
  T2 VARIABLE 1 32 0 0;
  NODES:
  INDEX 15 TAG SO OPERATION TR LINE 164 COLUMN 168;
  INPUTS T2(1..32)[0..0];
  OUTPUTS PC(1..32)[0..0];
  PREDECESSORS 14;
  SUCCESSORS *;

  INDEX 14 TAG SO OPERATION ADD LINE 166 COLUMN 168;
  INPUTS PC(1..32)[0..0];
  OUTPUTS T2(1..32)[0..0];
  PREDECESSORS 13;
  SUCCESSORS 15 CONDITIONS *;
  ...

  INDEX 11 TAG CBE OPERATION * LINE 117 COLUMN 153;
  INPUTS *;
  OUTPUTS *;
  PREDECESSORS 8 10 *;
  SUCCESSORS 12 CONDITIONS *;

  INDEX 10 TAG UTS OPERATION TR LINE 129 COLUMN 132;
  INPUTS IRE(1..1)[0..0];
  OUTPUTS *;
  PREDECESSORS 9;
  SUCCESSORS 9 CONDITIONS 0;
  SUCCESSORS 11 CONDITIONS 1;

  INDEX 9 TAG UB OPERATION * LINE 129 COLUMN 132;
  INPUTS *;
  OUTPUTS *;
  PREDECESSORS 10 *;
  SUCCESSORS 10 CONDITIONS *;
  ...

  INDEX 4 TAG CBR OPERATION TR LINE 117 COLUMN 153;
  INPUTS T1(1..2)[0..0];
  OUTPUTS *;
  PREDECESSORS 3;
  SUCCESSORS 5 CONDITIONS 3 1;
  SUCCESSORS 9 CONDITIONS 0;
  SUCCESSORS 11 CONDITIONS 2;

  INDEX 3 TAG SO OPERATION CON LINE 118 COLUMN 122;
  INPUTS IRE(1..1)[0..0] BRANCH(1..1)[0..0];
  OUTPUTS T1(1..2)[0..0];
  PREDECESSORS 2;
  SUCCESSORS 4 CONDITIONS *;
  ...
END;

```

Figure 7.4 YIF code for the example.

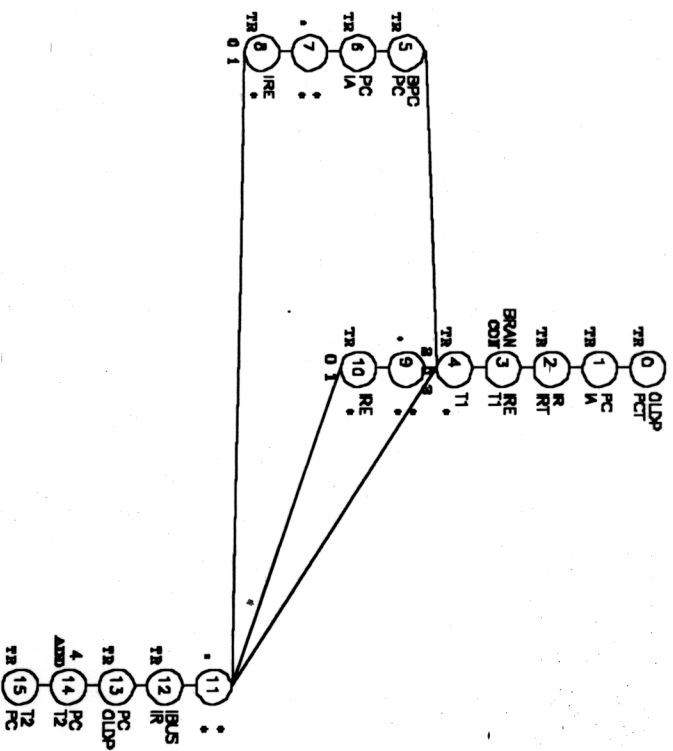


Figure 7.5 VIF represented as the graph (V, R, -L).

ically until now has been obtained by scheduling different operations to be executed in the same cycle. Two or more state machines running truly concurrently must be described as two or more VIF specifications, including some synchronization mechanism.

The Yorktown Logic Language (YLL)

GUIDING PRINCIPLES YLL was developed to describe multilevel combinational logic. Its main guiding principle has been that a block of combinational logic is a sequence of computations on vectors of 0's and 1's, which produce similar such vectors. As such, any software language that can execute these computations would be a candidate for describing the logic. Of course, a language that easily and concisely expresses this computation is preferable. The software language when compiled and executed, or interpreted, processes these 0's and 1's numerically as numbers. A specification language for logic can be immediately derived if all "logic" operations are executed symbolically. In this case, the domain of the symbolic manipulation is the set of logic functions. Since the capability to create and manipulate logic functions already resides in the logic synthesis module, the language processing should be straightforward.

A second principle has been that since the YSC system was to be a tool for silicon compiler development, we needed the capability of extending

the description language easily. Thus we adopted the strategy of embedding YLL in the language, APL, used for all our tool development. This decision brought with it several attractive features:

1. Interaction and easy debugging
2. Ease of manipulating vectors and arrays
3. Definition of powerful operators and their combinations, which could be extended naturally to the logic domain
4. Recursion
5. Easy syntax, simplifying the language parsing task

The idea of embedding YLL in an existing language has not only simplified the language processing task, but has also made language extension easy. For example, new operators can be created by writing an APL subroutine, named for the new operator. The subroutine is written in APL by the language developer, and uses the subroutines and conventions already established for manipulating logic functions for YLL processing.

We also decided, since the operators and combinations of operators of APL had been carefully chosen and constructed based on usefulness, that all those having a natural meaning in the logic domain should be defined. Thus any combination of the ten nontrivial binary logic operations, "AND," "OR," "XOR," and so forth, operating on arrays of arbitrary dimension are allowed. Perhaps this is more power than will ever be needed for specifying logic, but who knows?

Finally, we constrained the YLL processor so that the variables mentioned in the user's YLL source program and the set of logic functions produced by YLL are in exact one-to-one correspondence. This was done because of some initial experience with an input language that produced an equivalent and correct logic macro; however, only the identity of the module inputs and outputs was preserved. We felt that it was important that YLL could be used to describe the internal structure of the logic module in those cases where careful algorithmic design of the combinational logic was desired and where logic synthesis may not be capable of rediscovering this careful design.

YLL LANGUAGE The syntax of YLL is the same as APL, so a user of APL will find it easy to learn YLL. We will assume some knowledge of APL and discuss YLL in terms of its differences with APL. In principle, the YLL program written by the user to describe a module of combinational logic should closely resemble an APL program that performs the same computation numerically with 0's and 1's. Conversely, any APL program that operates in the Boolean domain should define, except for minor variations, a valid YLL program. The differences between APL and YLL are due largely to the fact that the APL program does numerical processing and the YLL program does symbolic processing. Thus, while the result of all statements in the APL computation is a 0 or a 1, the result in the YLL computation is a logic function. Other differences are due to choosing to extend YLL for further convenience.

Variables: There are two kinds of variables in YLL, logic variables and metavariables. *Metavariables* are those used for controlling the flow of the YLL execution and are typically used for parameterizing the subfunctions (e.g., to work with different bit widths). *Logic variables* correspond to those used to describe the hardware. Thus initially, each logic variable can be associated with a signal that will exist in the initial implementation description given to the logic synthesis task. To distinguish a metavariable computation from a logic computation, the user may choose to execute (Φ) a quoted string of characters, such as $(\Phi)'N \leftarrow \rho V'$. The general rule is that if a string of characters in APL has an operator that has a logic meaning, like $\leftarrow, \wedge, \neq, \sim$, but its APL meaning is intended, then the string is quoted and executed. Thus the full power of APL is available to YLL.

Constants: 0 and 1 are the only constants allowed for the logic variables. There is no restriction for metavariables. Reference to an integer that is to represent a binary vector can be provided by an APL function. For example, 32 DB 105 would produce a 32-bit binary representation of the decimal number 105. The function DB can be written trivially by the user.

```
VZ ← D DB N
-[1] Z ← (DB 2) + N
V
```

This is a good example of how easily YLL can be extended: simple or quite complicated new operators can be easily defined and used to extend YLL.

Assignment: Assignment ($\leftarrow$) is a logic operation that takes the logic function defined by the right-hand side and "OR's" this with the current logic function held by the variable of the left-hand side. We assume that all logic functions start with logic function 0 and are built up by this "OR-ing" operation.

Operators: All operators keep their usual meaning, but only those that are Boolean operations will give rise to a hardware computation. For example, $\wedge, \vee, =, >, <, \wedge, \vee, \neq, \leq, \geq$, and $\sim$ are the only true logic operations. The rest of the operators, such as $!, \uparrow, +$, etc., are useful for manipulating metavariables or indexing into or selecting from an array. For example, $VI[(N \div 2) + 1; K;]$ selects a set of rows of the logic array V . N and K are metavariables, and by convention because the operations are inside square brackets, the metavariable computation need not be quoted (in this case, since there is no ambiguity in what is meant, we would not have to quote the metavariable computation anyway).

Combinations: Combinations of logic operations operating on arbitrary arrays of logic functions are allowed along any dimension if such combinations are defined in APL. For example, $(\neq \setminus 3)A \vee \wedge (< \neq B)$ is allowed

as long as the dimensions of A and B are compatible with this operation. The result is an array of logic variables with the same dimension as the corresponding APL result.

Index origin: We have adopted index origin of 0 for compatibility with hardware conventions. Thus a 32-bit vector is indexed 0, 1, 2, 3, ..., 31.

Declarations: It is sometimes necessary to give a dimension to a logic variable before it is used. This is done by $\triangleleft$ (e.g., $A \triangleleft 3$ 5 defines A as a 3-by-5 array of logic variables). The statement $A \leftarrow 3$ 5p 0 would also initialize A , but all bits of A would have a logic function, namely 0. With the other declaration only the dimension is given. By convention, the individual logic variables exist in the macro only if they are assigned to it. Thus if $A \triangleleft 3$ 2 is defined but only $A[29]$ and $A[30]$ are computed, then only these two will be logic functions of the logic macro.

Equivalence: For convenience, we allow equivalencing between names. For example, $B \triangleleft 'A[3; 14]'$ provides the upper left 3-by-4 array of A with another name B . No new variables B are created. $B \leftarrow 1$ is equivalent to the statement $A[3; 14] \leftarrow 1$, but the first is easier to write.

Selection: We use the symbol $(A)C(Z \leftarrow B; Z \leftarrow C)$ is the same as the if-then-else statement; if $(A = 1)$ then $(Z = B)$ else $(Z = C)$. The convention is that each statement on the left, if true, selects the corresponding statement on the right. If there is one more statement on the right, then it is selected by the complement of the disjunction of all the statements on the left. The previous statement can be written also as $Z \leftarrow (A)C(B; C)$. Arbitrary nesting of selection is allowed.

Iteration: We borrow the "foreach" operator, $\cdot$, from APL2. Thus, $'A[\omega;] \leftarrow B[\omega;] \vee C[\omega;]'$ executes the quoted statement 32 times for $\omega = 0, 1, 2, \dots, 31$.

Inputs/outputs: Inputs and outputs are passed to a YLL function, both through arguments, and through global variables. Local logic variables are intermediate variables of the hardware description that may disappear later in the logic synthesis process, but inputs and outputs will always be present in the final hardware implementation.

Consider the following example:

```
VZ ← A FCNY B; C; N
[1]  $\Phi N \leftarrow \lceil \rho A \rceil \div 2'$ 
[2]  $G \triangleleft N$ 
[3]  $C \leftarrow (2 \times N) \uparrow A \wedge B$ 
[4]  $G \leftarrow C[2 \times \rho G]$ 
[5]  $Z \leftarrow (CNT)C(C; G, G)$ 
```

This example illustrates the following:

and process output columns 6, 7, and 8. This represents all the outputs in this case, but a subset could be specified. The inputs for this table are given in the right argument of GTABLE. They correspond to input columns 1 through 5 of the TRAFFIC table and should agree in dimension with the input dimensions for those columns given at the top of the TRAFFIC table.

Note that LIGHTTAB has only three colors; the code point 3 is never used, giving rise to a don't-care point. Also note that LIGHTTAB is used as the encoding table for the output column 6 where we see the three colors RED, YELLOW, and GREEN used as mnemonics in this column.

The APL function GTABLE does several things. First, it extracts all mnemonics for each output column and forms a don't-care function consisting of the complement of all code points appearing with a valid mnemonic. A "?" in an output column serves as an invalid mnemonic and hence the input conditions for that row are don't-care conditions. GTABLE accesses each input encoding table to determine the encoding for each input mnemonic. Thus, if a code point is not used in an encoding table, it also gives rise to a don't-care condition and will become part of the don't-care function derived. The encoding table also specifies the name of the output encoding vector for that column. One logic function for each output encoding bit is derived by GTABLE. Finally, all output functions for that output column are simplified using an ESPRESSO-based minimization procedure with a don't-care set given by the derived don't-care function.

Although it does not occur in the TRAFFIC example, the encoding table may be omitted for an output column. Then each valid mnemonic in that column is derived as a separate logic function, giving rise to a "one hot code" for that output column. Finally, if no output columns are listed in the left argument of the call to GTABLE, all output columns will be processed and a special signal, named TRAFFIC_VALID, will be created to indicate when a valid input was received. This can be used to detect an error condition and cause an interrupt, and indirectly, may be the mechanism used to allow the other signals to be don't care when there is an invalid input.

7.2.2 Structural Synthesis

Structural synthesis has become quite popular recently due to the need for more powerful design aids. Much thought has gone into the planning and the design of such systems, but comparatively little concerning the algorithms has appeared in the literature. Although structural synthesis is not yet a well-understood problem, several subproblems have been identified. Subproblems include scheduling operations given a certain number of resources, data path synthesis, control synthesis, pipeline synthesis, high-level optimizations, and so forth. Other efforts to develop design systems including structural synthesis have been and are being undertaken at the University of Karlsruhe [Wojt79] [Rose85], the University of Kiel [Zimm80] [Marw84], Carnegie-Mellon University [Dire81] [Kowa83] [Tsen86], University of Southern California [Knapp83] [Park86], Bell Labs

[Kowa85] [McFa86], and GTE [Blac85] [Fox85] among others. More general overviews can be found in [Camp85] [Gajs86] [Thom86].

In addition to the usual restriction to synchronous systems described by finite automata, two principles were used in the YSC system in order to more precisely define the goal of structural synthesis:

1. Logic synthesis—that is, synthesis of the combinational parts of the design—will be used extensively at a later stage. This is justified by the powerful logic synthesis tools available in the Yorktown silicon compiler. Moreover, the combinational logic usually takes only a small part of the total chip area. At a high level, not too much effort should be spent to minimize this part of the design.
2. Structural synthesis will be restricted initially by a high-level architectural decision—that is, to minimize the number of control states needed for executing all operation sequences. The tradeoff, between the number of control states (the number of clock cycles) needed to perform a sequence of operations and the time period required to execute the operations within each control state (the cycle time), is always decided initially in favor of the minimum number of control states.

The principle advantages for structural synthesis in using those principles are:

- They yield to a natural separation between the *sequential* and the *combinational* aspects of a system. At the beginning, only sequential aspects of the circuit to be designed must be considered.
- The goal is clearly defined. Initially, unknown parameters such as absolute speed or area, which are often estimated very poorly in structural synthesis systems, are not taken into account.
- Using as few states as possible in the control part also minimizes the number of latches required for storing data used in different states. This can be advantageous if latches are expensive compared to combinational logic, for example if static latches with scan-path capabilities are used.
- The design obtained initially can be modified in subsequent steps using basically one transformation—control state splitting.

The procedure for obtaining a structure from the behavior is based on the following ideas. Assume initially the design contains just combinational logic. If this is correct, the circuit can be obtained just using logic synthesis. The same is true if there are registers present (and identified) but no sequential control is necessary. If more than one control state is necessary, then it must be decided what operations have to be executed in what state (control state assignment). This is done by subsequently splitting an initial state that holds all operations. The operations are distributed among two new states. In each state splitting, as many operations as possible are scheduled in each of the new states. This assures that two states never will be merged again, as far as the number of states for performing the operation sequences is concerned. Operations can be assigned to more than one state.

In the YSC system, the main steps in structural synthesis from YIF are:

1. *Memory assignment.* Variables in the behavioral YIF description that must be implemented by memory arrays are identified and replaced by memory references. The memories themselves are generated only during layout design.
2. *Control state assignment.* A control state machine is built. Operations are assigned to the states of the control automaton; that is, it is decided which operations must be performed in which control states. Also data path registers are allocated in this step. Thus all sequential aspects of the system being designed are defined.
3. *Control state splitting.* After control state assignment, states contain as many operations as possible. Splitting a control state means assigning part of its operations to one or more new control states. This is done for two reasons. First, if sequential operations within one control state are distributed among two or more states, the necessary time to execute all operations within one control state may be shortened (at the cost of a new control state). Second, if two operations that were executed in parallel within one control state are distributed between two control states, they are sequenced and may share hardware, thus trading increased execution time for less area.
4. *Variable unfolding.* Variables are replicated ("unfolded") until single assignment is achieved. Each unfolded variable has only one source and thus corresponds to a net in a network.
5. *Operation folding.* Operations performing the same function can be implemented by the same hardware module if they are alternative or executed in different control states. The mapping of two or more operations to the same hardware module is called "operation folding." Control state splitting heavily influences which operations can be folded.
6. *Register folding.* Registers in the data path are assigned during control state allocation and splitting, by examining each state independently. Just as with operations, values held by registers might be alternative, in which case they can be mapped onto the same register. Register folding (also called "register assignment") is guided by "lifetime" analysis of the variables.
7. *Control and network generation.* Multiplexers are created for nets that get different alternate values. The select signals for these multiplexers as well as enable signals for registers must be added. Operators are chosen from the operator library. This library contains YLL subroutines, which are parameterized logic specifications for each of the operators, not particular implementations. The final network for each module is generated.
8. *Partitioning and module binding.* Cells such as registers and memories are selected from the cell library. The cells in this library are generated during layout design. Unlike operators, they do not need to be logically synthesized. The combinational logic still described as a function in YLL is partitioned into modules of appropriate size for logic synthesis or for the subsequent cell construction during layout compilation.

Steps 3 to 6 can be iterated or redone to obtain different designs. The algorithms for structural synthesis are described here only briefly. More details can be found in [Camp87].

Memory Assignment

Variables in YIF in general will be mapped either on storage elements (e.g., registers or memories) or onto nets (wires) during structural synthesis, depending on whether their value must be kept for more than one control state or not. The type of a variable indicates how to implement it. Variables initially of type register are the so-called *architected registers*—that is, registers already specified as those in the initial specification. Variables of type external are ports (pins) to the outside of the specified system. Variables of type internal will be implemented as registers only if necessary, otherwise as nets (wires). Variables of type array possibly will be converted into memories. Only one-dimensional arrays are allowed. Multidimensional arrays must be converted into one-dimensional arrays during compilation. Notice that each element of an array has a given number of bits (width). Each of these bits can be referred independently.

Example: Let $S_a = \{s \in \mathcal{S} | \text{type}(s) = \text{array}\}$. For $s \in S_a$, basically two situations can be distinguished:

- There is at least one reference to s that uses a variable index whose value cannot be determined during structural synthesis. Although there are special cases in which it would not be necessary, s is mapped onto a memory. For example, let $a \in S_a$ be of dimension m and width w (both constants). If in the YIF description there is a reference $a[l]$, j being another variable, then a will be implemented by a memory of m words of width w .
- All references to s use constant indices, that is, indices whose values can be determined at synthesis time. In this case, either s is mapped onto a memory or it is converted into as many variables as the dimension of s . The system decides which action is taken depending on the dimension of s and some other parameters. Large dimensions usually lead to memories, while small ones lead to single variables. For example, let $a \in S_a$ be of dimension m and width w (both constants). If in the YIF description all references use constant indices and m is small, such as 4, then a will be converted into a_1, a_2, a_3 , and a_4 . All references to $a[l]$ will be replaced by a new variable a_l , and so on. If m is large, such as 1,024, then a will be implemented by a memory of 1,024 words of width w .

Notice that the replication of variables might lead just to nets in the final implementation. Moreover, all variables may be referenced in parallel. Implementations as memories are usually smaller but only a small number of references in each control state (as many as the number of memory ports) is allowed.

Memory assignment gets more complicated if multiport memories are allowed. The decision of which reference to a variable is assigned to which

port of a multiport memory cannot be made at this stage and must be delayed.

In YIF, variables are mapped on memories by changing their type from array to internal and introducing additional operations that represent read and write. The read and write operations refer to the variable using an explicit address (no index). A constraint maps each of them to a memory cell. These map constraints will be taken into account during module binding.

The design after memory assignment is still represented in YIF. Each variable, whose type was array, has now been converted to another type.

Control State Assignment

The aim of state assignment is to create a control automaton and to determine which operation is executed in which state. Since this can be done in many ways for a given YIF specification, additional constraints are necessary. In our case, the initial goal is to minimize the number of control states needed for the execution of each of the operation sequences, starting with the first operation and ending with the last operation. All operations scheduled to one control state are implemented by combinational logic and thus executed "in parallel." Operations scheduled into different control states are executed in sequence; thus they are alternative in time and can be folded.

Given a YIF description $Y = (V, S, R, R_i, R_o)$ and the type of each variable, $\text{type}(s) \in \{\text{external, internal, register}\}$, the control state assignment algorithm works in four steps, constructing a set of states Z and assigning to each state $z_i \in Z$ a set of operations $\text{ops}(z_i) \subseteq V$ to be executed during that state.

1. *Control states due to loops.* For each loop (iteration, cycle in a graph) at least one control state must be introduced to allow the repetition of the operations in it. Loops are eliminated, and the iteration is encoded into the control by providing appropriate state transitions. The algorithm proceeds as follows. It first finds the feedback edges $L \subset R$, such that, in forming the transitive closure, A , of $(R, -L)$, a partial order relation results. In our present approach, L is derived syntactically from operation tags. These tags are generated during compilation by considering the syntax of the iteration constructs of the high-level language (e.g., WHILE and UNTIL iterations).

The operations of the resulting acyclic graph $(V, R, -L)$ are now assigned to states. Let

$$V_L = \{v_j\} \cup \{v \in V \mid \exists_r (v', v) \in L\}$$

V_L contains all those operations that must form the starting point of a state. A state starting point is the first operation scheduled in that state, according to A . Operation v_j (the first operation of the whole YIF) clearly is such an operation. The other operations in V_L are the first operations in each loop; since according to the semantics of YIF loops must possibly be repeated, also these operations must be the starting point of a state.

A first automaton is built with as many states Z as $|V_L|$. The operations executed in each control state $z_i \in Z$ are

$$\text{ops}(z_i) = \{v_j \in V_L\} \cup \{v \in V \mid (v', v) \in A\}$$

Since our goal is to minimize the number of required states to execute all operations sequences, all successors in A of the first operations V_L are initially included in each state. Particularly, all successors from v_j that is, all operations, are scheduled in one state. An example of the states generated by loops is given in Figure 7.6.

2. *Control states due to module calls.* An operation can be either an atomic computation, such as an addition, or a reference to another YIF description. The latter is called a *module call*. If the called YIF description generates a combinational circuit, it can be treated as an atomic computation. If the resulting circuit is sequential, then it cannot be treated as an atomic computation any more and an additional control state in the calling module must be introduced, splitting the original state. In the most general case, the number of control states (cycles) that a module call will need is unknown. For each of these module calls we introduce a new "waiting" control state. This module call waiting state does not perform any operation besides enabling the called module, until a *ready signal* is generated. Ready signals can be generated automatically by identifying control states that contain final operations in a module.
3. *Control states due to constraints.* Timing constraints, such as specifying that an operation should be performed in one cycle, might produce new control states. These new control states are added as specified.
4. *Control states due to data flow restrictions.* The first three steps yield a state assignment that is the starting point for data flow analysis. Data

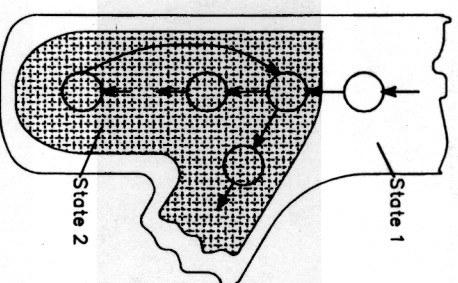


Figure 7.6 Control state assignment for a loop.

- a. Internal variables that are used as inputs without having been used as outputs before in the same state must hold a value that was written in another state. They must be converted into registers. These variables are defined by

$$S_{\text{ir}} = \{s \in S \mid \text{type}(s) = \text{internal} \wedge$$

$$\exists z \in Z, i \in V [v \in \text{ops}(z) \wedge (s, v) \in R, \bigwedge_{v' \in \text{ops}(z)} [(v, v') \in A \vee (v', s) \notin R_0]]$$

They are used as inputs by the operation v in state z without being used as output by any operation v' before v in that state. Thus, the value of these variables must have been written in another state. A storage element is necessary to keep a value from one control state to another in hardware. Hence, the type of all variables $s \in S_{in}$ is changed to $\text{type}(s) = \text{register}$. Unless architected, registers are only introduced where it is necessary

b. Variables with $\text{type}(s) \notin \text{register, external}$ create *data flow restrictions* if they are used as inputs and/or outputs more than a given number of times within one control state (cycle). A register can be written only once in a cycle, an external port (or memory port) might have only one value associated with it in each cycle. Control states in which this limitation is violated must be split into two or more states. For data flow constraints due to multiple use of a variable as output, these control states are obtained as follows. Given a state $z \in Z$ then if for some variable $s \in S$ with $\text{type}(s) \notin \text{register, external}$ the set

$$V_{ZS} = \{v \in \text{ops}(z) \mid (v, j) \in R_o\}$$

has a subset $V' \subseteq V_Z$ such that

$$A_{i_1 \neq i_2, \forall i}[(v_1, v_2) \in A \vee (v_2, v_1) \in A] \wedge |V_i| > c_i$$

then z must be split. c_j is the maximum number of times the variable j can be written (used as output) in one state (during one cycle). Usually $c_j = 1$. The control state z must be split between both references (in this case output references) to the variable j . Input or mixed (input and output) data flow restrictions are treated in an analogous way.

Steps a and b are iterated until no changes occur.

After control state assignment the sequential control is explicit. A register holding the control state is introduced, and the state transitions are given

by operations that load new values into this register. State assignment (i.e., encoding the states) may be done here if desired. The design at this stage is a finite state machine. All paths from a unique first operation to one last operation are performed in one cycle. This first design has the “extreme architecture” that minimizes the number of control states required for all operation sequences.

Since the registers in the data path were also determined during control state assignment, the data path also is designed to some extent. Hence, control and data parts are synthesized together in a single step.

The result is again represented in YIF. Consider the example of Figure 7.4. It produces the YIF description given in Figure 7.7. Here the variable *PC* was converted into a register (and unfolded, see the variable unfolding description that follows), also the state register *STAT* was created, encoding the states by simply numbering them.

Control State Splitting

Further modifications to the basic structure obtained so far are:

- Splitting states that are too long, that is, the combinational part in the data path that is too slow. Possibly new registers are introduced as explained in control state assignment.

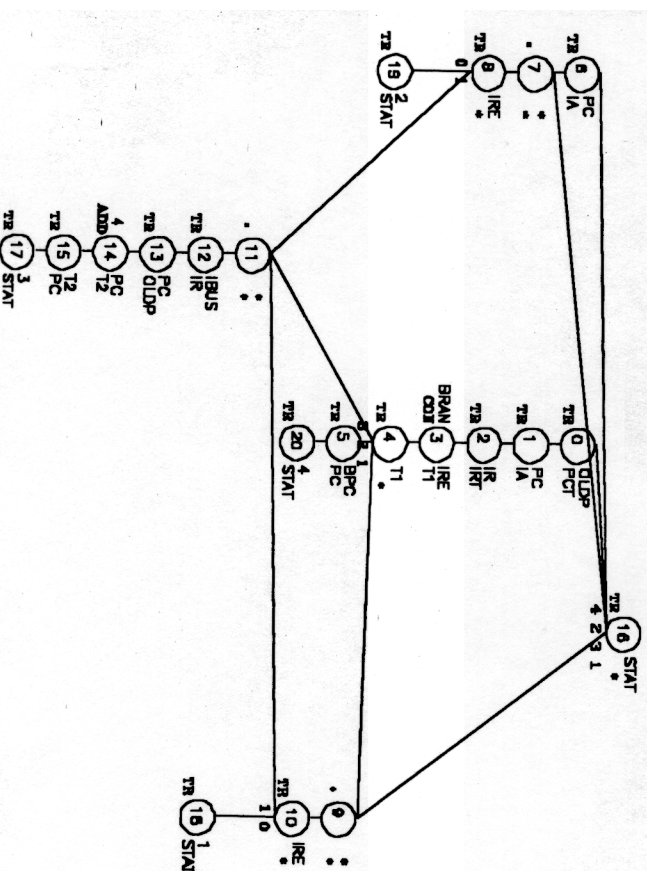


Figure 7.7 YIF representation with explicit sequential control.

- Modifying the design so that the critical paths in the data path assigned to each control state have a similar estimated delay or meet timing constraints.
- Splitting states that contain operations requiring hardware that uses a large area. Since control states are alternative in time, this allows for the mapping of such operations onto the same hardware gaining area.

To identify the states to be split, a goal must be defined, such as a given maximal cycle time (specified as a timing constraint). State splitting already requires an estimation of the delays and areas not yet known at this stage. Successful state splitting can only be achieved with good relative estimates. In the YSC system, area, power, and delay are estimated by using functions of arbitrary complexity stored in the module library or by exercising logic synthesis using only fast algorithms.

Variable Unfolding

In an imperative description, variables hold their values until they are overwritten; therefore, they can be used many times. Variable unfolding introduces additional variables so that single assignment results, and changes the references accordingly. Variable unfolding occurs both for registers and for internal variables implemented by nets.

Storage elements already meet the single assignment criterion due to the data flow restrictions imposed earlier. Nevertheless, it must be identified, if an input or an output is being referenced: outputs are only loaded at the end of a cycle and are available only during the next cycle. The unfolding consists of introducing new variables for the inputs whenever this is necessary. An example of register unfolding is given in Figure 7.8.

Internal variables that will be implemented by nets do not necessarily meet single assignment. For sequential assignment to these variables, new variables must be introduced. In doing this, the necessity of introducing additional multiplexers might arise (see Figure 7.9).

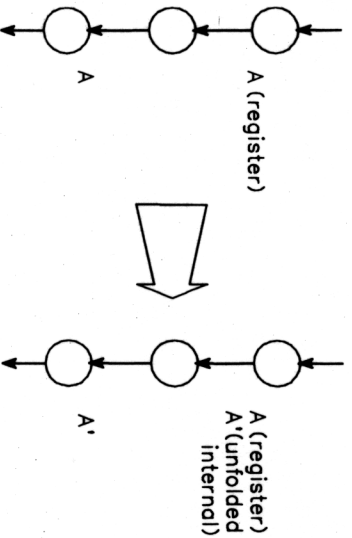


Figure 7.8 Register unfolding.

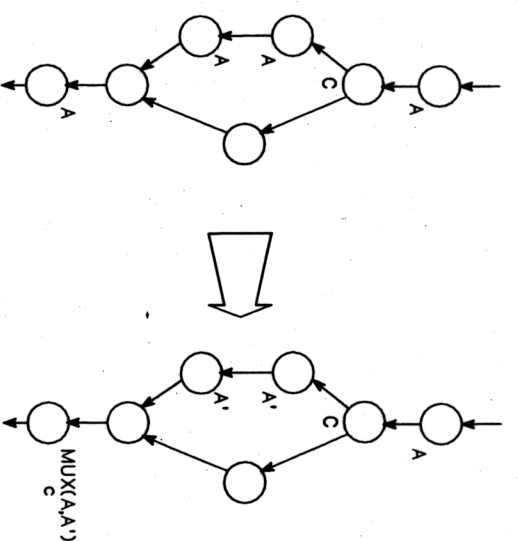


Figure 7.9 Internal variable unfolding.

Operation Folding

Although we relieved structural synthesis from performing so-called high-level optimizations of combinational parts that can be done during logic synthesis, not all optimizations can be done by logic synthesis. Some optimizations arise from the sequential nature of the circuits. As an example, consider the circuits in Figure 7.10.

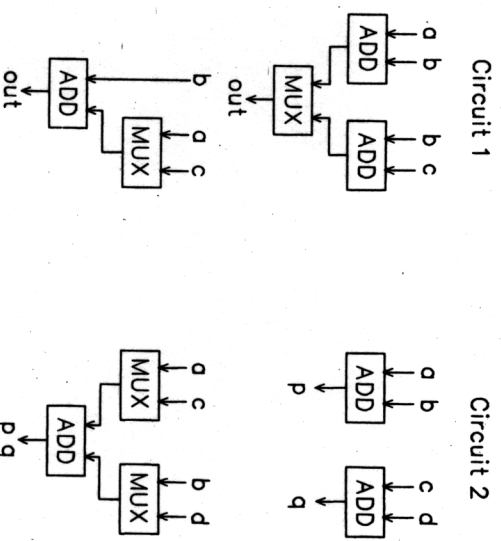


Figure 7.10 Two examples of optimizations of combinational circuits.

The optimization of circuit 1 can be performed by logic synthesis; it will yield an optimal circuit (optimal according to the goals defined in logic synthesis) for any width of the operators, for some of the operators being constants, and so forth. Doing this at a higher level would be extremely difficult since the properties of each functional block (an addition in the example) vary (e.g., they might be commutative or not or they might have a neutral element 0 or 1).

The optimization in circuit 2 cannot be performed by logic synthesis; the operands used in the original circuit are different for both adders. But using the fact that both additions always occur in different control states, that is, they are alternative in time, both adders can be collapsed by adding some multiplexers as indicated. This kind of optimization should be done by structural synthesis. In the YSC system, this is called *operation folding*. Two operations v and v' can be folded, if

$$(v, v') \notin A \wedge (v', v) \notin A$$

that is, if they are alternative. It can be shown that folding more than two operations results in the clique partitioning problem for the complement of the symmetric closure of $AtGolub80$.

Folding should result in some advantage. Our criteria are:

- The operations should be complex compared to a multiplexer. This excludes, for instance, simple gates from being folded.
- The operations should be similar so that folding results in a smaller circuit. Similarity in the simplest sense means requiring the same function (e.g., an addition), only allowing different input and output widths. More refined similarity criteria can be table driven. The number of different functions that an operation can perform is limited to the order of 10^2 , so this approach is practicable.

Operation folding is done simply by appending a *map constraint* to the operations being folded. The design is still represented in YIF. During control and network generation, the multiplexers for the inputs will be created and the outputs will be merged. The combinational circuit for the operations themselves will be obtained and optimized during logic synthesis.

Register Folding

Registers are generated according to the initial specification and during control state assignment and splitting. Besides the architected registers, registers are introduced if the value of a variable must be passed between two or more states. Clearly some of these variables might be "alternative," that is, there is no situation in which they are both "alive." These variables can be mapped onto a single register. Merging two or more of the registers initially generated is called *register folding*. To detect which registers can be

folded, techniques similar to "lifetime" analysis in compiler construction are used.

Control and Network Generation

At this point, the design is considered ready with respect to its sequential aspects. The YIF data format now satisfies the following properties:

- The predecessor-successor graph, which orders the operations (nodes in the YIF graph) is acyclic.
 - The data flow graph defined by R_i and R_o , giving the connection between variables and operations, is acyclic. In this context, the input and output sides of registers are considered separate.
 - Indices selecting parts (bits) of variables are constants. Variable indices have been replaced by operations or memory calls.
 - All operations can be implemented by combinational logic. The only YIF nodes that specify an operation requiring a sequential implementation are module calls.
- The goal at this stage is to produce:
- The description of a network connecting module inputs, module outputs, registers, blocks of combinational logic, and calls to other modules
 - The logic specification of the combinational blocks

Descriptions of the different steps to reach this goal follow.

OPERATION-TO-OPERATOR MAPPING For each operation specified in the YIF graph, a combinational function ("operator") must be chosen. A library of operators is supplied, given by their logic specification, as well as connection and control information. For instance, an ADD operator in the library might not be used only for the "+" operation in the YIF graph, but also for an increment operation "+ +". The library specifies that ADD can be used for incrementing, by connecting the single operand to the first input of the ADD and a constant "1" to its second input. Simultaneously an ALU could be present in the library, capable of performing among others the "+" and "+ + " operation. In the same way, constants will be specified for their control inputs, instructing the ALU to perform the required function. Besides this so-called mapping information, expressions are provided in the library for every feasible operation/operator combination to predict roughly the area, delay, and power requirements. The expressions yield these estimates as a function of the width and the number of operands, and they are used to make a default selection when several mappings are feasible. By default, one combinational function is instantiated for each operation. The choice of the operator might be dictated by previous decisions, such as the folding of more operations onto a single operator.

It is worth noting here that, for example, the implementation of an

increment by an ADD unit is *not* a waste of area or power. The library fixes only the logic behavior of the ADD unit. The actual hardware (gates) is only generated after a logic synthesis step, thus reducing ADD with a constant input to a "real" incrementer.

WIDTH ADJUSTING The widths of variables, constants, and operators need to be matched in order to write a real network definition. Normally a (sign-) extension of the variables and constants (operands) is sufficient to match the inputs of the operator. Sometimes the operator itself must be resized, for instance when operations involving different widths are folded onto each other. Further, the output of the operator must be matched with the specified output variable. This can be done by removing the most significant (carry-) output bit of the ADD unit, for instance.

MULTIPLEXING The insertion of multiplexers is needed to write to nets, which are assigned to more than one operator, and to write to the inputs of operators, which are assigned more than one operation due to folding. The "select" inputs of the multiplexers are decoded, in order to match the control (enable) signals discussed in the next section. The multiplexers are stored in the YIF description in the same way as any other operator, and they are fetched from the same library.

A more complex control, such as for an ALU, is automatically generated by this scheme. The control input of the ALU was assigned a constant for every operation that was folded on it. By multiplexing these constants, and subsequent logic synthesis of this multiplexer with constant (data-) inputs, the proper control logic is obtained.

CONTROL SIGNAL GENERATION Control signals are needed for the select inputs of the multiplexers, for the enable inputs of the registers, and for the synchronization and communication with the "outside world" and other module calls.

Note that the entire YIF graph handled at this stage is only one module out of a hierarchy of modules specifying the intended circuit. For each module, one finite automaton is created. In general, the synchronization between these automata is arranged in such a way that automata above in the hierarchy provide a start (enable) signal for those in the level immediately below (their children), and await their completion, which is signaled by an outgoing ready (enable) signal.

Inside each module, control signals are generated, enabling the paths through the graph that should be executed according to the conditions in the forking nodes. In general, each module gets one enable input, assigned by the outside world, controlling the activity of the module. All starting nodes of the (acyclic) YIF graph are controlled by this incoming enable signal. Proceeding down through the graph, new enable signals are assigned. At the joining of paths, a new enable is created by "or-ing" the enables corresponding to the incoming paths. At a fork, new enable signals are created for the outgoing paths according to the branch decision and the

incoming enable. The different logic functions required for the creation of the control signals are obtained from the same YLL library as the data path operations.

A module output enable signal is created by "or-ing" the enables of a subset of the terminating nodes of the graph, indicating to the outside world the completion of the activity of the automaton. Further, outgoing enables are provided for the control of outgoing data, declared as registers in the outside (parent) module. Of course, module calls have an enable input, assigned by the enable in the current path. The output enable of the module call is used as the enable for the next node(s) downward in the same path.

The enable signals created this way are easily attached to the multiplexer inputs. The register enables might need a final "or-ing" of these signals if they are written by more than one operator. If the writing node was a module call, its corresponding outgoing register enable is used.

Basically, this scheme generates a so-called one hot encoding. For example, the node that implements the branch on the value of the state register will cause one enable signal for every (used) state register value. However, if the logic blocks that generate or use these enable signals are passed as one unit to the logic optimizer, these enable signals remain internal to this unit. They might be removed by minimizing the logic. Thus the large number of enable signals are just part of the logic specification and may not be present in the final realization.

NETNAME ASSIGNMENT For writing the network, a unique name should be given to each net. Normally, the name of the YIF variable can be used, but in practice a large number of operations (nodes) in the YIF graph specify an assignment or a caretate operation. For these operations, no logic is generated; they are resolved instead by a suitable naming, thus specifying the required interconnection. By this mechanism, many YIF variables are removed and replaced by constants.

Partitioning and Module Binding

The previous steps generated a network, connecting I/O variables, registers, and calls to logic functions from a library. As a last step, these logic function calls are grouped into combinational units that will be passed to logic synthesis. Of course, for the best minimization, all the function calls should be grouped into one large block. However, two reasons might exist for a partitioning into several smaller groups of functions. The first is the time and space complexity of logic synthesis, which limits the size of its input. The second reason is that the layout programs might be able to find a better floor plan if several smaller units are created. Thus a partitioning of the logic may be desired, which can be based on connectivity between the resulting groups and on the use of inputs common to the groups trying to construct blocks of a given size. Notice that control and data path are not separated.

The resulting network is written in a simple language, HND, suitable for machine and human reading. It allows a hierarchical definition of the network with separately generated modules. The interconnection through the different levels of the hierarchy is checked and established later. The logic specification of each group of logic functions is written in YLL.

After all different modules of the chip hierarchy are generated this way, a linking phase is done. Recursively, all HND files are processed, checking the consistency of the interfaces. For the calls to combinational logic, the results of logic synthesis are read, because the interface might have been affected by minimization, inputs might have been removed, and outputs might have been set to constants. Further, the hierarchy is expanded, generating one output module for each different instantiation, unless mapping constraints specify mapping of several module calls onto the same instance. The network information is written in a compact binary form suitable for interfacing with the layout compilation step.

The HND result of structural synthesis for the example of Figure 7.3 (also in Figures 7.4 and 7.7) partitioning the required logic into two groups is given in Figure 7.11.

HND consists of modules. Each module has a header indicating its identification (Module_id), its type, its technology, and the date and time it was synthesized. Module types can be:

- *Module*—for modules that are not leaves in the hierarchy and instantiate other modules.
- *Decomp*—for modules that are leaves in the hierarchy and contain combinational logic. For each of these modules a YLL program containing its logical specification is generated.
- *Latche*—for modules that represent latches with an "enable" input. Other types of latches (e.g., without an "enable" input, with "set" and "reset" inputs, etc.) are also available.

For each module, the input and output nets as well as the internal nets (Locals) are indicated. The names are generated, keeping the name in the original YLF wherever possible. A module that is not a leave may instantiate other modules giving them an instance name (first parameter after the keyword Instance). Each instantiated module is connected by giving a list of nets, which is connected in that order to the inputs and outputs as listed in the module definition.

Several things should be explained for the example shown in Figure 7.11. Module FETCH is the root of the hierarchy. It instantiates the decomp modules FETCH1 and FETCH2, which contain the combinational logic partitioned into two groups. A two-bit latch instantiated as STATE_REGISTER.1 corresponding to the state register and three 32-bit latches instantiated as OLDP1 (holding OLDP), PC1 (holding PC), and IR1 (holding IR) were generated. To the inputs of each module a CLOCK was added (SCVS implementation). The inputs and outputs of the

```
Module_id FETCH module scvs 198612102253;
Inputs CLOCK IRE IBUS[0:31] BRANCH BPCT[0:31];
Outputs:
  outputs IAL[0:31] IRTL[0:31] PCTL[0:31];
  locals STATE_REGISTER.[0:1] in.OLDPCT[0:31] in.PCTL[0:31]
  in.STATE_REGISTER.[0:1] enable_OUT_17[2:3] enable_OUT_12[0]
  enable_IN_5 latch_PC1 latch_STATE_REGISTER.1;

Instance OLDP1 LATCHES32 *;
  connects CLOCK enable_IN_5 in.OLDPCT[0:31]
  PCTL[0:31];

Instance PC1 LATCHES32 *;
  connects CLOCK latch_PC1 in.PCTL[0:31]
  in.OLDPCT[0:31];

Instance IR1 LATCHES32 *;
  connects CLOCK enable_IN_5 IBUS[0:31]
  IRTL[0:31];

Instance STATE_REGISTER.1 LATCHES2 *;
  connects CLOCK latch_STATE_REGISTER.1
  in.STATE_REGISTER.[0:1]
  STATE_REGISTER.[0:1];

Instance EQL_OR1 FETCH1 *;
  connects CLOCK IRE BRANCH STATE_REGISTER.[0:1]
  in.STATE_REGISTER.[0:1] enable_OUT_17[2:3]
  enable_OUT_12[0] enable_IN_5 latch_PC1
  latch_STATE_REGISTER.1;

Instance ADD1 FETCH2 *;
  connects CLOCK BPCT[0:31] in.OLDPCT[0:31] enable_OUT_17[2:3]
  enable_OUT_12[0] enable_IN_5
  IAL[0:31] in.PCTL[0:31];
End_of_module
```

Figure 7.11 HND Example.

root module FETCH correspond to the interface specified initially with a CLOCK added.

Module FETCH1 results in just 22 transistors, and module FETCH2 results in 406 transistors after logic synthesis. Without partitioning the logic and using the same logic synthesis procedure, just one module with 417 transistors would result. Besides the latches, this is all the hardware required. Several other alternatives using different state splitting criteria, other partitioning of the logic, different logic synthesis alternatives, and trading time for area can be exercised very fast.

7.2.3 Logic Synthesis

Introduction

The YSC system is an optimizing compiler, and logic synthesis has exploited different optimization techniques. We have also emphasized cer-

rain styles, in implementing the combinational logic, that fit well with the philosophy of the other modules of the compiler. For example, we use multilevel logic, and we have emphasized the use of cells based on linear transistor arrays for the gates. The logic synthesis also has the ability to trade off the speed of an algorithm with the quality of results so that early estimates of area and delay may be obtained quickly. In addition, we are continuing to experiment with new algorithms for minimization, factorization, and decomposition of multilevel logic.

Complicated logic functions or sets of functions are often implemented in practice by using multilevel logic (multistage circuitry). This is because it is easier to control a number of properties of the final implementation, such as its area, delay, input/output pin positions, and flexibility of its layout. Although a set of logic functions is simply a mapping from its inputs to its outputs, if the mapping is defined as a sequence of smaller mappings whose intermediate results are useful, then a more efficient implementation may be obtained. Data flow mappings such as ALUs, rotators, and decoders are such examples. Although "control" logic is generally considered suitable for a two-level implementation, our experience is that control logic also maps well into multilevel logic. Thus by adopting multilevel logic as the design style, nothing is lost, and we gain the advantage that control and data flow can be treated equivalently, allowing for merging the two where appropriate. We felt that in describing logic the user should be allowed the full power of a general programming language. The logic synthesis program must have the ability to deal with the output of such a description, which may intermix control and data flow logic.

Just as with software, there are different algorithms that compute the same thing, and some are more efficient than others. A software compiler generally preserves the overall sequence and flow of the algorithm. With a hardware compiler, a similar option should be available; but hardware and software compilers are different. With hardware, there is more opportunity to optimize since the compile time allowed can be longer and optimization has a bigger payoff. Thus the hardware compiler has the opportunity to change the sequencing and flow in order to optimize further, if the appropriate algorithms can be developed.

The task of logic synthesis in an optimizing silicon compiler is to map the logic equations into the specified target circuitry and to optimize the area while meeting the timing constraints. In the Yorktown silicon compiler, we use automatic synthesis and minimization for multilevel logic based on algebraic factorization and Boolean simplification. The synthesis process is carried out by applying a sequence of operators to a fixed ambient data structure. The operators are such that they can be applied in any order and repeated until no change occurs. They can both remove intermediates and synthesize new ones, guided by measuring their "value" based on area savings or delay reduction. If the incoming logic is already multilevel, the degree to which the initial decomposition is preserved during the synthesis

process can be controlled, thus allowing the designer to specify more and more of the implementation details if desired.

The logic synthesis procedure transforms a more or less arbitrary logic description into a set of circuits that can be implemented in a given target technology. It also carries out the optimization of the circuits themselves. In many circuit families, such as dynamic domino CMOS logic [Erd84][Kram82][Pre84] differential cascode [Hel84], and conventional static CMOS, a complicated logic function can be implemented in a single circuit. Automatic optimization is essential to the success of these complex circuit families, since design by hand is impractical. For a general discussion of the complex CMOS circuit families, see [Chen84].

Although each function in the final result must be realizable, much of the synthesis procedure is technology independent. Even those operations that are guided by hardware considerations often require only a simple subroutine to check whether or not a given Boolean expression is simple enough to be realized as a single circuit. Due to this modularity, it is easy to change from one circuit family to another. One can also carry out synthesis of a single piece of logic several times, varying the target technology; in this way, a comparison between circuit families can be obtained without difficulty. The three types of complex CMOS circuits mentioned previously are among those supported by the logic synthesis programs.

An outline of the logic synthesis procedure follows. We begin with the process of constructing the logic from the YLL description. This creates an initial description of a "Boolean network," which is described in the next section along with transformations to be performed on it. We then detail the algorithms that carry out these operations and explain how they fit together to form a logic synthesis procedure. Finally, we demonstrate the effectiveness of this procedure with a practical example.

Logic Construction: YLL Processing

One way to view the YLL processing is to understand that its purpose is to build a structure and that the YLL program is a description of that structure. The structure is a set of nodes, one for each variable used in the YLL code. Associated with each node is a logic function. The structure is thus an interconnected set of logic functions. The variables of the YLL program are either inputs, intermediates, or outputs of the logic module being described. The inputs and outputs are variables passed into or out of the YLL subfunction, either as arguments or global variables. The intermediates are the local variables of the YLL subfunctions.

Unlike simulation languages, the order in which the lines of the YLL code appear, except for a few cases, is fairly arbitrary. The reason is that each line describes a piece of this structure and the sequence in which these pieces are described is not relevant. Thus a variable can be "used" before its value is computed.

Each variable is assigned a unique positive integer starting with 2. 0 and 1 are reserved for the logic constants. Associated with the integers

2 3 4 ... are the APL variables $\Delta 2 \Delta 3 \Delta 4 \dots$, which will hold the logic functions of those variables, stored in sum-of-products form. Each time a new value (logic function) of a variable is computed, it is "or-ed" with the existing value, which is assumed to be 0 if there is currently no value. Since "or-ing" is commutative, the sequence in which these values are computed is of no relevance. Also if a variable X is used in another logic function, say Y , it does not matter that the value of the logic function for X has not been constructed yet. For example, in the statement $Y \leftarrow X \neq A$, the logic function $XA + XA$ is constructed, so the actual logic function for X is not relevant. Of course, in the final structure for the logic module, the logic function for X will affect the variable Y . For each of the positive integers, the user name for that variable is kept on a namelist, so these names may be passed to the rest of the compiler for other purposes.

We also have to process temporary variables, which arise as intermediate results. For example, in a line of YLL such as $Z \leftarrow (\vee \neq A) \vee \wedge (B \vee C)$, the results $(B \vee C)$ and $(\vee \neq A)$ must be stored in temporary variables. Then the inner product is calculated, using these temporary logic functions, to obtain the logic functions for Z . After this line is completed, the temporary results may be erased. In this way, we ultimately create a structure that has exactly one node for each user-defined variable, and no other nodes. We use the convention of assigning temporary variables negative integers, starting with -1, -2, ... and storing the associated logic function in the APL variables $\Delta 1, \Delta 2, \dots$.

Since YLL is embedded in APL, the processing is made easier. Processing a YLL function consists of two steps: translation and execution. Translation is a fairly straightforward process that alters the given YLL program and produces a similar-looking APL program. The translation is guided by the following table.

LXLITAB	
$\vee$ LOR	$\wedge$ LNAND
$\wedge$ LAND	$\vee$ LNOR
$\sim$ LNOT	$\neq$ LEVALA
$>$ LGT	$\circ$ LOUTER
$\geq$ LGE	$\boxtimes$ LPLA
$<$ LLT	$\top$ LENCODE
$\leq$ LLE	$\cdot 1$ LDECODE
$=$ LEQ	$\ominus$ LPREV
$\neq$ LNEQ	I LRENAME

Line 1 of the table gives the APL function LOR as a replacement for the APL operator $\vee$. LOR was written by the language developer and is a function that manipulates logic functions using the data structures defined. Thus any occurrences of $\vee$ not in quoted expressions are simply replaced by LOR in the translation process. Other symbols (e.g., $\leftarrow, /, \backslash, \cdot, \circ, \neq, \wedge$, etc.) that have logic meaning are similarly translated, but since their syntax

is more complicated, the translation process is embedded in a program. The YLL program FCNY that follows is an example of this translation process.

```
VZ ← A FCNY B;C;N
[1]  $\neq$  'N ← [(pA) ÷ 2]
[2]  $G \Delta N$ 
[3]  $C \leftarrow (2 \times N) \uparrow A \wedge B$ 
[4]  $G \leftarrow C[2 \times pG]$ 
[5]  $Z \leftarrow (CNT) C(C;G;G)$ 
```

This YLL program is translated into the two APL programs that follow:

```
V FCNA
[1]  $\neq$  'N ← [(pA) ÷ 2]
[2] SINK 'G' LALO N
[3] SINK 'C' LSET (2 × N) ↑ A LAND
[4] SINK 'G' LSET C[2 × pG]
[5] SINK 'Z' LSET (CNT) LIMP ' (C;G;G)'
```

```
V FCNAZ ← A FCN B;C;N;Z
[1]  $\varepsilon$  Function names: FCNY
[2]  $\varepsilon$  Global names set: G
[3]  $\varepsilon$  Global names ref: CNT
[4]  $\varepsilon$  Global names alloc:
[5] LOCK
[6] FCNA
[7] UNLOCK
[8] FCNA Z ← LEVALA Z
```

The first program, FCNA, is the direct translation of FCNY. Note that the lines of FCNY and FCNA are in a one-to-one correspondence. The differences are basically that certain logic symbols have been changed to the corresponding subfunction call. Thus

Symbol	Replaced by
$\wedge$	LAND
$\vee$	LIMP
$\leftarrow$	LSET
Δ	LALO

The subfunction SINK is placed at the beginning of each line, containing an assignment, to clean up the temporary variables that may have been created by that line.

The function FCN is the main call to FCNA, and its purpose is to take care of the input and output arguments, the local and global variables, and to do various cleanup chores. The call in FCN to FCNA is surrounded by LOCK and UNLOCK, which lock the current temporary variables required

after FCNA is completed. Then they are unlocked so that they may be erased later. The final operation is to take the temporary result Z returned from FCNA and put it into the correct place, in this case into the dummy logic variable FCNAZ. At the time FCNA returns its values, the variables Z have been assigned a permanent value, denoted by positive integers, but because Z is a dummy variable, these variables are now made temporary by the function LEVALA. Negative integers are returned by FCN, and the corresponding logic functions are stored in the APL variables, e.g., $\Delta 1$, $\Delta 2$, ...

After the translation process has been completed for all appropriate functions, the next phase is to execute the resulting APL program; in the previous example this is done by a call to FCN. If any arguments are to be passed, they must be allocated first; then the function call can be made. For example, in FCN, we would have to allocate the variables for the inputs A and B, and the global variable CNT. This can be done, for example, by another YLL program with lines

```
AA  $\Delta$  BB  $\Delta$  10
CNT  $\Delta$  1
ZZ  $\leftarrow$  AA FCN BB
```

This causes the APL program FCN to execute, and the result is that the block of appropriate combinational logic is created.

When a variable is allocated in YLL, it is assigned an integer and its name is stored on the namelist. A variable may be allocated either the first time it is assigned a value (logic function) or by using an allocation (Δ) statement. When the APL programs execute, the variables in the YLL programs become APL variables. These variables are arrays of integers, positive or negative, with the integers being the numbers allocated to the variable name or a temporary result. Thus the YLL variables may be reshaped, selected from, and generally manipulated like any APL array, since they are simply just arrays of integers.

It is also possible in a YLL program to refer to the value of a variable X using the symbol $\odot X$, but the value obtained is the current logic function associated with X, so care must be taken here to describe what is meant.

If the variable A is allocated the numbers 33 34 35, then if during the execution of the APL programs we requested the APL value given to A, we would get 33 34 35. But if we request its logic value using an auxiliary function called SHOW, we would see the logic functions $\Delta 33$, $\Delta 34$, $\Delta 35$ displayed in symbolic form. For example, suppose that we define the program

```
VMAINX
[1] D  $\Delta$  5
[2] E  $\Delta$  5
[3] CNT  $\Delta$  1
[4] Z  $\leftarrow$  D FCN E
```

and then translate and run MAIN, stopping execution before the fourth line of FCNA:

```
MAIN
FCNA[1]  $\leftarrow$  N  $\leftarrow$  [ (p A)  $\div$  2'
FCNA[2] SIN K ' G' LALO N
FCNA[3] SIN K ' C' LSET (2  $\times$  N)  $\uparrow$  A L AND B
FCNA[4] SIN K ' G' LSET C[2  $\times$  p G]
```

Now we print both the APL value of C and its associated logic function using SHOW.

```
C
16 17 18 19 20 21
SHOW C
C[0] = D[0]  $\odot$  E[0]
C[1] = D[1]  $\odot$  E[1]
C[2] = D[2]  $\odot$  E[2]
C[3] = D[3]  $\odot$  E[3]
C[4] = D[4]  $\odot$  E[4]
C[5] =  $\Phi$ 
```

The process is similar for G.

```
G
13 14 15
SHOW G
G[0] = C[0]
G[1] = C[2]
G[2] = C[4]
```

Continuing execution, after MAIN has completed, we can display the result Z and its logic functions:

```
Z
28 29 30 31 32 33
SHOW Z
ZA  $\Delta$  1[0] = CNT  $\odot$  C[0] + CNT  $\odot$  G[0]
ZA  $\Delta$  1[1] = CNT  $\odot$  C[1] + CNT  $\odot$  G[1]
ZA  $\Delta$  1[2] = CNT  $\odot$  C[2] + CNT  $\odot$  G[2]
```

Note that the name Z in MAIN has been renamed temporarily to ZA Δ 1 because the dummy name Z used in FCNY conflicts with the name Z used in the subfunction MAINX, which will be an output of the combinational logic module created. The variable Z in FCNY is, in effect, a temporary variable and like any temporary variable will not appear.

Conditional clauses associated with C are processed using the notion of an environment. The environment is a logic function that is always "and-ed" with the logic function computed on the right side of an assignment ($\leftarrow$) statement. The environment function is held in a special variable ENV.

As the environment changes, for example when another conditional clause is encountered (if conditional clauses are nested), the new condition is computed and “and-ed” with the current environment, the old environment is pushed onto a stack, and the new one is stored in *ENV*. When a condition ends, the stack is popped reestablishing the previous environment. Environment logic functions are stored in temporary variables (Δn), and the associated negative integers are the ones actually pushed and popped from the stack. A special “locking” and “unlocking” mechanism is used to keep these temporary variables from being swept away when temporary variables are cleaned up in *SINK*.

The final phase is to write the results into a file, using the function *WRITE*. We use logic description format, logic interchange format (*LIF*), but could also store the result in an internal data structure used in the logic synthesis programs. For *LIF*, two steps must be carried out. The first is to gather up each logic function and write it out in a PLA-type format. The second is to determine which variables are the inputs, outputs, and intermediate variables of this logic module. This is determined by which APL variables still exist in the workspace (global variables) and which have associated logic values. The global variables with associated logic functions are the outputs. The variables that exist, that are used by at least one logic function, and that have no logic value are the inputs. The logic functions that exist, that are used, and for which there are no APL variables with those integers as values are the intermediate variables. This information along with the users’ names in the namelist is written to the output file.

Boolean Networks

Some logic synthesis programs are based on data structures that consist of a netlist of qualified gates. Local transformations are performed, mapping one set of such gates into another. The simple notion of a *Boolean network* provides more freedom in developing algorithms that are largely independent of the target technology. A Boolean network is a representation of a combinational module, that is, a block of logic implementing a set of logic functions. Eventually, a Boolean network will be implemented by a macro cell, that is, by an interconnection of cells each implementing a single logic function. Each logic function produces either a primary output of the module, an intermediate result, or both. These functions may depend upon the primary inputs to the module, upon the intermediates just mentioned, or upon both. In order to preserve the combinational nature and avoid race conditions, no cyclic dependencies are allowed. A primary input or an intermediate will be referred to simply as a *variable*. A Boolean network can be represented by a directed graph with a logic function associated with each node and an arc from node *i* to node *j* if function *j* depends on the variable associated with function *i* or input *i*.

The logic function at each node is represented as a sum-of-products Boolean expression. Internally, this expression is given as a matrix, the columns of which correspond to variables and the rows of which represent individual product terms. For example,

$$\begin{array}{rcccl} D & = & AB & + & BC \\ F & = & DE & + & DE \end{array}$$

	A	B	C	D	E
1	1	1	2	2	2
2	1	0	2	2	2
2	2	2	2	1	0
2	2	2	0	1	1

Here we have used + to denote logical “or,” juxtaposition to denote “and,” and underscore to denote negation. Thus *F* is defined to be the exclusive or of *D* and *E*; note that *D* is actually an intermediate defined in terms of *A*, *B*, and *C*. In the matrix representation, variables occurring in their positive form are marked with a 1, those occurring in their negative form are marked with a 0, and a 2 indicates that the variable does not occur. A row defines one product term.

The function at each node is stored in the data base. The data structure used in the Yorktown silicon compiler is a single, large, bit matrix with a set of pointers to the rows associated with each function. Each variable is associated with two columns of the bit matrix, with 00 representing a 2; 01 representing a 1; and 10 representing a 0. As the synthesis proceeds, functions are retrieved from the data base, modified or eliminated, and rewritten back into the data base. Additional information about the function (e.g., its delay, suitability for implementation in a single gate, or if the function has changed recently) is sometimes stored in other arrays.

Input to the synthesis system is typically a two-level Boolean cover, an interconnection of two-level Boolean covers (*LIF*), or a Boolean network of functions produced by a logic description language such as *YIL* and stored in the data structure described previously. Thus the output of the logic synthesis can be an input as well.

A two-level Boolean cover can be interpreted as a Boolean network in at least two ways: (1) as a two-stage network where the product terms in the input plane produce intermediates, sums of which are then used to express the outputs; or (2) as a one-stage network in which each output is expressed as an independent sum of products. In the former case, the Boolean cover can be minimized by a PLA minimizer such as *ESPRESSO-II*, which creates intermediates (in this case, product terms), that may be useful even for multilevel minimization. Thus for some logic, it may be suitable to use *ESPRESSO-II* and direct its output into the multilevel logic synthesis program. In the latter case of the one-stage network, the creation of intermediate expressions to simplify the network is entirely the responsibility of the synthesis program.

Basic Logic Synthesis Operations

In the course of synthesis, a sequence of operators is applied to the Boolean network. There are five kinds of operations performed. They are of two different types—algebraic and Boolean—and different algorithmic variations are provided so that the synthesis process can be tailored to be faster (for estimation purposes) or slower (for more optimal results). The algebraic operations represent methods for increasing the speed of the operations by

treating the logic functions as polynomials. Generally, a sum-of-products representation of a logic function can be viewed as a special type of polynomial—one where no power greater than 1 occurs and where the combination XX never occurs. Also, we normally manipulate only those sum-of-products expressions that are prime and are irredundant. In such cases, very fast manipulation techniques (e.g., for finding common subexpressions) can be found. The Boolean operations (e.g., logic minimization, complementation, etc.) allow the full power of changing a logic expression into a logically equivalent one, but require more time. A typical synthesis procedure is composed of a sequence of these operations, and the procedure is tuned to give a desired tradeoff between speed of the synthesis process and quality of its results. The five kinds of operations are:

1. *Extraction (algebraic)*. A subexpression common to two or more functions is extracted and used to create a new intermediate variable, which then replaces the subexpression wherever it occurs. The new intermediate can be used in both its positive and negative forms.
2. *Elimination (algebraic)*. An intermediate of low "value" is eliminated, and the subexpression it represents is "pushed back" into the functions that use it. This is the inverse of extraction.
3. *Simplification (Boolean)*. An unminimized logic function is replaced by a logically equivalent (for the module) but "simpler" expression. This transformation is essentially "static." Unlike the other two, it does not usually change the structure of the Boolean network, although arcs in the network may disappear or new arcs may appear if dependencies change.
4. *Re substitution (algebraic and Boolean)*. An existing function is substituted or divided into another to see if the second can be usefully simplified. Substitution can be either of the algebraic (weak division) or Boolean type (strong division). Typically, every pair of functions is tried, but appropriate filters are used to quickly rule out some pairs.
5. *Decomposition (algebraic)*. After common subexpressions have been identified by extraction, some of the functions may still be too complex to be implemented in a single circuit of the target technology. Such functions must be further factored and decomposed. This decomposition is carried out locally because it is assumed that global commonality has already been identified by the extraction process.

After the decomposition step, an actual gate of the target technology is selected for each function in the network. For the complex cells used in the domino and static CMOS technologies, this gate is automatically designed with the aid of logic factorization algorithms.

EXTRACTION (ALGEBRAIC). The discovery of common subexpressions appearing in the Boolean network is the "creative" step in the automatic synthesis process. It is achieved by forming partial factorizations of each function and generating a list of *kernels*, which are essential subexpressions from

which algebraic commonality between two or more functions can always be discovered.

To explain the notion of kernels, we must first introduce the idea of an *algebraic quotient* for a pair of Boolean expressions. A *logic expression* $f = f_1 + f_2 + \dots + f_n$ is a particular sum-of-products representation of a Boolean function. Each f_i is a single product term or *cube*. The *algebraic product* of two expressions f and g is only defined when f and g depend upon disjoint sets of variables. It is given simply as the sum of all possible crossterms $f_i g_j$. Since f and g have disjoint variable sets, no zero products can occur (a variable is never multiplied by its complement). In this sense, the product is purely "algebraic."

Similarly, the algebraic quotient (f/g) of two expressions is required to depend on variables other than those on which g depends. It is defined to be the largest (most terms) expression such that

$$f = (f/g)g + r$$

where r is yet another expression (the remainder). Here the product between (f/g) and g is the algebraic product (which is defined), and the right and left sides of the equation are required to agree as expressions, not just logical functions. That is, the same set of cubes must appear on both sides of the equation. The quotient (f/g) is unique. For example,

$$f = AB + AC + AD + BC + BD$$

$$g = A + B$$

$$f/g = C + D$$

$$f = (f/g)g + r = (A + B)(C + D) + AB$$

By using sorting techniques, the computation of the quotient f/g can be carried out very efficiently. In fact, the division requires only $O(n \log n)$ steps, where n is the total number of product terms in f and g . Further, by keeping the product terms of all expressions sorted, which is possible to do, then algebraic division is linear in n .

Clearly the formation of quotients is closely related to the problem of factoring an expression. Similarly, the task of identifying subexpressions common to two or more functions is essentially the problem of finding common divisors. Here g is a divisor of f if f/g is nontrivial. Notice that if g is a divisor of f , then so is (f/g). The motivation for defining the kernels of an expression is to specify a manageable set of divisors that is still rich enough to allow all common subexpressions to be located.

An expression is said to be *cube-free* if no cube can be factored out of it. Thus $ABC + ABD$ is not cube-free (it equals $AB(C + D)$), but $C + D$ alone is cube-free. The set of *kernels* $K(f)$ of an expression f is defined to be the set of all quotients f/c such that c is a cube and f/c is cube-free. Note that a kernel must have at least two product terms, since a single cube is not cube-free.

The number of kernels in an expression is typically much smaller than the number of possible cubes c that can appear in the definition. Indeed, f/c is the empty expression for many choices of c , and for many additional choices it is not cube-free.

Since we represent our Boolean expressions in sum-of-products form, it is fairly easy to identify common divisors that are single cubes. It is much harder to locate cube-free common divisors, and this is the purpose of kernels, as illustrated by the following result [Bray84a].

If two expressions f and g have a cube-free common divisor, then they have a cube-free divisor which is the sum of cubes common to two expressions b and k , where $b \in K(f)$ and $k \in K(g)$.

Thus to find a subexpression common to two or more functions in a Boolean network, we compute the set of all kernels and then build expressions out of cubes common to pairs of kernels. If no common subexpressions are found in this way, then the preceding result guarantees that the functions in the network are "relatively prime"—that is, they have no cube-free common divisors. Consequently, the only remaining common subexpressions are single cubes; these can be located in a straightforward way.

For additional efficiency, we can limit the calculation to "depth zero" kernels (see [Bray84] for more details), a somewhat smaller set that still suffices to find useful new intermediates.

Once the candidate intermediates have been identified, they are given ratings, roughly measuring the number of literals they would save in a factored representation of the Boolean network. If this rating exceeds a critical value, the intermediate is created and the subexpression it represents is replaced by a single new variable wherever it occurs. More precisely, to create an intermediate variable v to represent the expression g , we first add the equation $v = g$ to our Boolean network. Then we replace each expression f already in the network by the new expression

$$f = \kappa(f/g) + v(f/g) + r$$

(Of course, if both g and $\bar{g}$ do not divide f , then f is left unchanged.) We refer to this process as *Algebraic substitution* of v for g in f . (True Boolean substitution will be discussed in the section on simplification that follows.)

The new intermediate consolidates logic that would otherwise be duplicated in many functions. The rating just mentioned usually reflects a savings in transistors in a physical realization of the network, independent of the circuit technology used. For example, consider the three Boolean expressions f, g , and h given by

$$\begin{aligned} f &= AE + BE + CDE \\ g &= AD + AE + BD + BE + BF \\ h &= \overline{A}BC \end{aligned} \quad (1)$$

We compute the set of kernels for each function:

$$\begin{aligned} K(f) &= \{A + B + CD\} \\ K(g) &= \{A + B, D + E, D + E + F\} \\ K(h) &= \phi \end{aligned}$$

Observing that the expression $A + B$ consists of two cubes common to the kernel $A + B + CD$ of f and the kernel $A + B$ of g , we create a new intermediate variable X to represent this expression, and replace $A + B$ with X wherever it occurs. The resulting Boolean network is

$$\begin{aligned} X &= A + B \\ f &= EX + CDE \\ g &= DX + EX + BF \\ h &= CX \end{aligned} \quad (2)$$

Notice that the negation of X has been used to simplify the expression for h .

ELIMINATION (ALGEBRAIC) Elimination is the inverse of extraction. If we eliminate (push back) the intermediate X in the Boolean network (2) above, we obtain the original network (1).

This process is not carried out indiscriminately. Rather, we begin by assigning a value to each intermediate in the Boolean network. This value, like the rating used during extraction, measures the degree to which an intermediate helps to simplify the Boolean network. We then eliminate only those intermediates whose values fall below a certain critical level. This critical level mediates between the otherwise antagonistic extraction and elimination processes.

Elimination tends to reduce the total number of circuits and delay stages in the design. It is also useful in converting a large, sparse network of simple functions into a small, denser network of somewhat larger functions. A sparse network is sometimes produced by the extraction operation; it also might be given as input to the synthesis procedure—certain logic description languages produce such networks. Elimination also enhances the potential for simplification. Thus it is not unusual for elimination to be invoked several times in the course of synthesis.

SIMPLIFICATION (BOOLEAN) A single Boolean function corresponds to many equivalent Boolean expressions. It is obviously desirable to keep all expressions occurring in a Boolean network in as simplified a form as possible. The incoming logical data must be minimized by the synthesis program, and further simplification may be possible after an elimination operation. It is slightly less obvious that an expression involving intermediate variables contains implicit relations that also can be used in its simplification. For example, if it is given that

$$X = AB + BC$$

$$Y = AD$$

then

$$Z = X + CY$$

simplifies to

$$Z = X + Y$$

This simplification results because the condition $\overline{X}CY$ can never occur; it is part of the don't-care set for this network; thus $Z = X + \overline{X}CY + \overline{X}CY = X + Y$.

In general, to any Boolean network we can associate a list of variable combinations that are logically impossible; the totality of all such conditions is the don't-care set for the network. The don't-care set includes, for each intermediate variable v , the condition

$$\overline{v}f + fv$$

where f is the expression represented by v . In the preceding example the don't-care set is given by the expression

$$\overline{X}(AB + BC) + X(\overline{A}B + BC) + Y(AD) + Y(\overline{A} + D)$$

The reader can verify that the cube $\overline{X}CY$ is contained in this don't-care set as claimed. These conditions describe implicit relations between the variables in the network, and the exploitation of these relations is essential to the optimization of multilevel logic.

The simplification routines we employ were abstracted from the ESPRESSO-II minimizer, described in [Bray84b]. These algorithms are based upon the "unate recursive paradigm," a divide-and-conquer strategy that has proven quite efficient. Simplification without a don't-care set can be carried out by a very efficient procedure (called SIMPLIFY in [Bray84b]), which does not guarantee that an irredundant cover of prime implicants will be produced, although this is often the case. If a don't-care set is available, there are two methods for producing an irredundant cover. One is based on tautology, and one is based on complementation of the function and its don't-care set. We have implemented the latter method since it is the general paradigm of the ESPRESSO algorithms, but if the don't-care set is extremely large, a tautology-based method is probably preferable. We use the ESPRESSO-based simplification operations to obtain the simplified result if don't cares are available.

In practice, the don't-care set for the entire network is too large to process at once, so when minimizing a particular function, we only compute the relations between the variables it directly depends upon. Note that elimination tends to bring distant dependencies closer together, so if done before simplification, there is increased potential for simplification.

The idea of what constitutes a "simple" function for multilevel logic

must be discussed. It is not measured simply by the minimum number of product terms in the expression. In general, we would like to obtain an expression that factors simply. Since this is difficult to control with the two-level simplification techniques, we have employed a "minimum variable" or "minimum literal" heuristic to obtain "simpler" expressions.

For example, suppose we want to simplify

$$g = AB + CX$$

where

$$X = A + B$$

We merely simplify f relative to the don't-care set

$$d = \overline{X}(A + B) + X(\overline{A}B)$$

associated to the intermediate variable X .

As a heuristic, we wish to reexpress f in terms of the least number of variables A, B, C , and X required. (A similar heuristic is defined as follows to determine the minimum number of "literals" $A, \overline{A}, B, \overline{B}, C, \dots$ required to express f .) This is done by computing the onset $f = g - d$ and offset $r = \sim g - d$; thus defining an incompletely specified function (f, d, r) . In this example

$$\text{on set } = f = ABX + ACX$$

$$\text{don't-care set } = d = \overline{X}(A + B) + X(\overline{A}B)$$

$$\text{off set } = r = \overline{A}CX + ABX$$

Each of the cubes of f do not intersect with the cubes of r . This means that for each pair of cubes, there must be at least one variable in "conflict." Thus ABX is orthogonal to $\overline{A}CX$, because A conflicts with $\overline{A}$. We then build a "variable-blocking matrix" M , where each row represents a pair of cubes, one from f and one from r and where we put a 1 in the columns that represent the conflicting variables. In the example,

		Variables					
		A	B	C	X	$f\text{-cube}$	$r\text{-cube}$
$M =$	1	0	0	0	1	ABX	$\overline{A}CX$
	0	0	1	0	1	ABX	$\overline{A}BX$
	0	0	1	0	1	$\overline{A}CX$	$\overline{A}CX$
	1	0	0	1	1	$\overline{A}CX$	ABX

It is easily shown that a literal may be removed from a cube of f if there remains at least one 1 in each row of the blocking matrix. For example, B can be removed from ABX . If an entire column of the variable-blocking matrix can be removed while maintaining at least one 1 in each row, then each occurrence of that variable may be deleted from the cubes of f . A minimum variable representation can be found by determining the maximum number of columns that can be removed. This is solved by a minimum "row covering" algorithm—that is, by selecting a subset of

columns so that there is at least one 1 in each row of M . In the example, one determines by inspection that two minimum covers exist, corresponding to the variables A , C , and X or to variables A , B , and C .

If we now restrict f to the last subset of variables, we obtain $g = AB + \bar{A}C$ (the superfluous variable X has simply been discarded). At this point, normal simplification proceeds, but each cube is prime and irredundant, so the final result is

$$f = AB + \bar{A}C$$

Thus the minimum variable heuristic selects that X should be eliminated before any of the cubes of f are expanded to prime.

As an alternative, a minimum literal heuristic can be obtained by building, in a similar way, a "literal-blocking matrix," with a column for each literal appearing in the cubes of f . Thus

<i>Literals</i>					
A	B	C	X	$\bar{A}$	<i>f-cube</i>
1	0	0	0	0	ABX
0	1	0	1	0	$\bar{A}CX$
0	0	1	0	0	ABX
0	0	1	0	0	$\bar{A}CX$
0	0	0	1	1	$\bar{A}CX$
0	0	0	1	1	ABX

We could eliminate the sets of literals $\{B, \bar{A}\}$ or $\{X\}$, obtaining either $f = AX + CX$ or $f = AB + \bar{A}C$, respectively. However, the first solution is the minimum literal solution, and it is the one found by the minimum covering algorithm.

This example illustrates how the minimum literal heuristic can give a preferred answer in terms of multilevel synthesis, since $f = X(A + C)$ is preferred over $f = AB + \bar{A}C$ (less literals in the factored form of f).

RESUBSTITUTION (ALGEBRAIC AND BOOLEAN) Resubstitution is the operation of attempting to use a function that already exists in order to simplify another existing function. There are two versions of resubstitution, algebraic and Boolean. In algebraic resubstitution, we select any pair of existing functions f and $v = g$, and then form $(f/g)v + (f/\bar{g})v + r$ and unless $f = r$, we change f . Since there may be many pairs that have to be tested, it is expedient to eliminate some pairs by "filters"; for example, if the number of terms in g is greater than the number of terms in f , or if g contains a variable not in f , then the pair (f, g) is not tested. The set of algebraic filters eliminates most pairs, so algebraic resubstitution is extremely efficient. In theory, extraction will find any potential algebraic resubstitutions. However, it is less efficient since it is more general than resubstitution; therefore, we employ algebraic resubstitution quite liberally in the synthesis process.

Boolean resubstitution can be more expensive than algebraic and must be used with care. Don't-care simplification, already discussed, can be used to define true Boolean substitution (also called strong division). In general, if we wish to substitute a new variable X with associated logic function g into a given logic function f , we write down a don't-care set,

$$d = Xg + X(\sim g)$$

For example, if we wish to determine if the new intermediate

$$X = \bar{A} + B$$

can be usefully substituted into the existing expression

$$f = AB + \bar{A}C + BC$$

we merely simplify f relative to the don't-care set

$$d = X(\bar{A} + B) + X(\bar{A}B)$$

associated to the intermediate X .

The problem is the same as don't-care simplification—that is, to simplify f using the don't-care set d . We employ the extended notion of "simple" (see the previous section on simplification) using the minimum literal heuristic. Also, since we wish to substitute the intermediate X into the expression f , we require that X be included in any minimum literal subset so computed. In this example, we obtain $f = X(A + C)$.

Boolean substitution can be a powerful tool for multilevel logic synthesis. The following examples of its effectiveness occur in part of a 4-bit ALU description.

$$L = (BF + \bar{B}F)(A + E) + \bar{A}E(\bar{B}F + BF)$$

substituted into

$$R = (BF + \bar{B}F)(\bar{A} + E) + \bar{A}E(\bar{B}F + BF)$$

yields

$$R = \bar{A}(\bar{E}L + EL) + \bar{A}(EL + EL)$$

Here both the positive and negative polarities of L have proven useful in the simplification of the expression for R . On the other hand,

$$R = (BF + \bar{B}F)(\bar{A} + E) + \bar{A}E(\bar{B}F + BF)$$

substituted into

$$L = (BF + \bar{B}F)(\bar{A} + E) + \bar{A}E(\bar{B}F + BF)$$

yields

$$L = \bar{A}(\bar{E}R + ER) + \bar{A}(ER + ER)$$

This shows that R can be useful in expressing L . Remarkably,

$$T = (DH + \bar{D}H)((G + C)(\bar{E} + \bar{A})(B + E) + \bar{B}E + CG) + (D\bar{H} + \bar{D}\bar{H})((G + C)(\bar{A}E(B + F) + BF) + CG)$$

substituted into

$$S = (H + D)((G + C)(\bar{A}E(B + F) + BF) + CG) + DH$$

yields

$$S = D(H + T) + HT$$

This demonstrates that strong division can sometimes discover unexpected and dramatic simplifications.

Boolean substitution is more powerful than algebraic substitution, but it is also more expensive. In practice, similar results are often obtained more

efficiently by algebraic substitution followed by simplification, and this is the technique we normally use. The new intermediate is first incorporated into those existing expressions for which it is an algebraic divisor, and then the more expensive Boolean simplification is carried out only for these expressions. In the model example discussed earlier, the algebraic substitution of X into f yields the expression $AB + XC$. When this is simplified relative to the don't-care set for X , the expression $X(A + C)$ is again obtained.

DECOMPOSITION (ALGEBRAIC) After extraction, it may still be necessary to decompose large functions into smaller pieces. Different technologies demand different decomposition styles; the common problem is to locate within a complicated expression, a large subexpression that can be implemented as a single circuit. Since the task of identifying global commonality has already been carried out, this additional decomposition can be applied to each function independently, without loss of efficiency in the final design.

Decomposition and factoring are basically the same thing. In factoring an algebraic expression, we select a divisor, g , and write the expression as $f = (f/g)g + r$ (where (f/g) is the algebraic quotient) and then recursively factor the three expressions (f/g) , g , and r . Decomposition can be done in the same way, except that after selecting g we change f to $(f/g)w + (f/g)v + r$ and store $v = g$ in the data base. The f and g are recursively decomposed further. This process stops if (1) f or g cannot be factored or (2) f or g meet some stopping criteria.

At this point, there may still be some functions in the data base that do not meet the technology specifications. Such functions are broken down further by (1) splitting a sum of products into two smaller sums or (2) breaking down single product terms, which do not pass the criteria, into the product of two smaller product terms.

After this, all functions will pass the technology criteria that assure that each can be built in a single gate of the target technology. Then, usually, a switch is set indicating that no expression should be subsequently changed into one that does not meet the criteria. The test for passing the technology criteria is coded in a single subroutine. This subroutine depends on the target circuitry used and could be simply a function that receives a logic expression as input and uses table lookup to determine if the expression is acceptable. For complex cell families, such as domino logic, there would be too many entries in the table, so we have used parameterized criteria (e.g., the maximum number of literals in any cube being less than a given value, typically 5). For complex static CMOS, the criteria might be no more than 5 literals in any cube of the function, and no more than 3 literals in any cube of the complement function.

All the operators previously described have been implemented such that when the switch is set, a test is made to see if any function has changed from one that was acceptable to one that is not. If so, the change is rejected and the operator attempts either to find a change that is acceptable or gives up and goes on to the next function. Thus in effect, each operator has two versions, one that knows nothing about technology and one that

knows about technology but only through the simple device of using the single criteria subroutine to test if a change in an expression is acceptable. This allows for the creation of technology-independent algorithms for logic synthesis and a set of programs that can be easily altered to switch from one set of target circuits to another. We have implemented this with a set of overlays to the logic synthesis programs so that the user may select from a menu of different circuit technologies.

A Typical Set of Steps in Synthesis

The general philosophy used in the synthesis sequence is to collapse the logic into a small but highly useful set of logic functions, apply simplification using as much of a local don't-care set as can be allowed, and use extraction to decompose the network further. Thus, when simplification is applied, a more global result can be obtained than is possible by a set of local transformations. Further, the generality of logic simplification, in effect, defines all possible local transformations and is applicable to any type of target technology. After each function is simplified, its resulting sum-of-products form is most always in a canonical form; hence algebraic extraction will identify most of the common subexpressions.

The operations described previously are linked together in a sequence to form a complete synthesis procedure. A typical sequence of operations is as follows:

1. Apply fast simplification (SIMPLIFY) to the incoming data.
2. Apply algebraic resubstitution for maximum sharing of the intermediate or output functions.
3. Apply elimination. As usual, those intermediates whose values exceed a given critical amount are not pushed back. By setting this cutoff higher or lower, we can control the degree to which the original decomposition is preserved.
4. Perform Boolean simplification, using the implicit don't-care sets.
5. Extract common subexpressions. Even subexpressions with fairly low value should be extracted, since they help to disclose other subexpressions.
6. Eliminate again. Any intermediates of little value are now removed.
7. Simplify using don't-care sets again.
8. Decompose the logic into functions simple enough to be implemented in single circuits in the target technology. Set the technology switch.
9. Eliminate one final time. At this point, because the technology switch is set, we do not push back any expression if its removal creates a function that can no longer be realized as a single circuit. At the same time, attempt to reduce the number of circuits and delay stages by trying to push back all but the most valuable intermediates.
10. Select or design the circuits for each function in the network (using factorization for domino or complex static CMOS technologies).

Practical Examples

An APL implementation of the logic synthesis system just described has been in use at IBM T. J. Watson Research Center since 1982. Boolean

networks with as many as 1,000 functions and PLAs with 30 inputs and outputs and over 2,000 product terms can be treated with reasonable expenditure of resources. A Boolean network that has not been minimized can be synthesized into a circuit family such as domino logic in roughly one to three times the amount of CPU time required to produce a minimized PLA for the same logic. When the input to the synthesis procedure is known to be a minimized PLA, the processing is faster because some Boolean operations can be suppressed.

Various stages in the synthesis of a small piece of logic extracted from an IBM System 38 CPU will be illustrated here. The incoming data represent six output functions in the form of a single PLA; there are no intermediates initially. The PLA depends upon 27 inputs, which are labeled A through Z, and A'. Intermediates introduced during synthesis will be labeled B', C', and so forth; the output signals are labeled 1 through 6.

This example illustrates the functioning of extraction, elimination, and a form of decomposition. The original PLA has already been minimized (as a PLA), and subsequent simplification has no effect.

The original expressions for the six outputs are shown below. For brevity and ease of comprehension, we show a factored form of each of these expressions, although internally each is represented as a sum of products. The factorization algorithm is itself based on kernel generation, so it is not surprising that the factored forms already reveal a good deal of the structure present in the network.

Input Data

$$\begin{aligned}
 1 &= (HI + GHNFBCD + E) + A + N(BCD + E + A)) \\
 &\quad + (E + CDB(GPH + D) + GKLO) + (E + CDBGM(H + D) + GKLE) \\
 &\quad + (I + HG(AP + AM) + GKLAO + AJ) \\
 2 &= (HI + GHNFQU(T + VW) + R) + Q + N(SU(T + VW) \\
 &\quad + R) + Q) + (I + HG(MSU(T + VW) + R) + Q) \\
 &\quad + P(SU(T + VW) + R) + Q) + GKJ(SU(T + VW) + R) \\
 &\quad + Q) + O(SU(T + VW) + R) + Q) \\
 3 &= (ABCDENQX + ABCDEFQX)(HI + HD) + (ABCDERQX \\
 &\quad + ABCDENQX)(H + D) + GK(ABCDERQX + ABCDEQX) \\
 4 &= ABCDEX(GPH + D) + G(KLO + HIN) + HIN \\
 5 &= ABCDEX(FHI + GHI) + G(MH + D) + G(KL) \\
 6 &= (Z + Y)(ABCDENQSTUVWXA' + ABCDEFQSTUVWXA') \\
 &\quad (GHI + HD) + (ABCDERQSTUVWXA' \\
 &\quad + ABCDENQSTUVWXA')(G(H + D) + GK(ABCDERQSTUVWXA' \\
 &\quad + ABCDEQSTUVWXA'))
 \end{aligned}$$

Next we illustrate the results of extraction. Five new intermediate variables have been introduced. In addition, the program has discovered that primary outputs 4 and 5 can also be used to simplify some of the other functions. This could have been discovered by resubstitution as well.

Result of Extraction

$$\begin{aligned}
 1 &= D'(BCD + E) + A) + E'(BCD + E) + A) \\
 2 &= D'(SU(T + VW) + R) + Q) + E'(SU(T + VW) + R) + Q) \\
 3 &= QF' + QG' \\
 6 &= (QRSTUVWA'F' + QRSTUVWA'G'XY + Z) \\
 B' &= I + H \\
 C' &= E' + GHI \\
 D' &= JH' + FC' + GMB' \\
 E' &= OH' + NC' + GPB' \\
 4 &= F' = ABCDEXE' \\
 5 &= G' = ABCDEXD' \\
 H' &= G + L + K
 \end{aligned}$$

The elimination process reduces the seven intermediates above to four somewhat larger ones, shown below. The degree to which the complexity of the original network has been reduced is evident when compared to the input data.

Result of Elimination

$$\begin{aligned}
 1 &= D'(BCD + E) + A) + E'(BCD + E) + A) \\
 2 &= D'(SU(T + VW) + R) + Q) + E'(SU(T + VW) + R) + Q) \\
 3 &= QF' + QG' \\
 6 &= (QRSTUVWA'F' + QRSTUVWA'G'XY + Z) \\
 D' &= F(HI + GHI) + G(MI + H) + GKLE \\
 E' &= G(PI + H) + G(HIN + KLO) + HIN \\
 4 &= F' = ABCDEXE' \\
 5 &= G' = ABCDEXD'
 \end{aligned}$$

This network is now decomposed, with domino logic playing the role of the target technology. In this circuit family, the main constraint on feasibility of a given logic expression is the length of its longest product term. The decomposition operation for this family discovers that if the primary outputs of the network are all complemented, then every expression becomes feasible. (Complementation is allowed in this case because the outputs are feeding a latch.) The new network, with the intermediates relabeled, follows.

Result of Decomposition

$$\begin{aligned}
 1 &= B'(A(EG + D) + B) + C') + AC'(EG + D) + B) \\
 2 &= B'(Q(R(T(V + W) + U) + S) + C') + QC'(R(T(V + W) + U) + S) \\
 3 &= QD' + QE' \\
 6 &= Q(R + S + T + U + V + W + X + A') + Q(R + S + T + U
 \end{aligned}$$

$$+ V + W + A') + QR + S + T + U + V + W + A') + F' + YZ$$

$$B' = (HI + HI + EGJ + K + L) + G(MH + D) + EHD$$

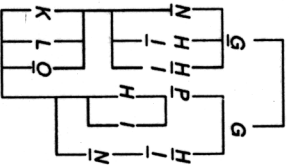
$$C' = (N + HI + H)G(K + L + D) + G(EH + D) + HIN$$

$$^4 = D' = A + B + C + D + E + X + C'$$

$$^5 = E' = A + B + C + D + E + X + D'$$

Finally, series-parallel circuits are constructed for each of the preceding functions. This operation, which is essentially the same as factorization, is also performed by the synthesis system. The series-parallel graph for the intermediate C' follows.

Circuit for C'



The Boolean network, together with the topological circuit descriptions, is now available for further processing by the layout program.

7.2.4 Layout Design

Introduction

The dominating guideline in the layout design part of the YSC system is to postpone implementation decisions in order to avoid committing the program prematurely to a specific implementation. Each decision should leave enough freedom for the remaining stages to satisfy the constraints it created and at the same time should rearrange the available data such that further meaningful decisions are possible in the next step. So, concurrent with the gradual stiffening of the design, the information is progressively organized so that more detailed decisions can be derived. Postponing implementation decisions is the characteristic of what has been called *stepwise refinement*. The usual perception of the principle is as a progressive decomposition of a task into subtasks in a sequence of refinement steps. This sequence of refinement steps terminates when all undecomposed tasks are straightforward translation routines. This method thus entails a hierarchical structure.

The principles of stepwise refinement obviously apply to any complex design task based on a top-down strategy rather than to a process of combining independently developed subdesigns. In general, completely specified subdesigns are difficult to handle because the flexibility and the information for adapting them to their environment are often not available

when they are designed. Therefore, the concepts of stepwise refinement and a hierarchical structure are particularly compatible with the environment of automatic layout design in a silicon compiler.

The Environment of Layout Design

THE INITIAL HIERARCHY The input to the Yorktown silicon compiler and the structural synthesis in the early stages have imparted hierarchical structure to the design data. Each module in this hierarchy contains a possibly empty collection of submodules with an incidence structure (a net list). If the collection of submodules is empty, the module is a cell. In all other cases, the module is a compound. The hierarchy can be represented as a rooted tree. The modules are represented by the nodes. The root node represents the whole system. The leaves represent the cells. The internal nodes represent the compounds.

The hierarchy probably reflects some high-level viewpoints such as the functional interdependence of the modules. Functional interdependence and connectivity are often highly correlated, and the latter is an important basis for decisions in layout design. The functional hierarchy, therefore, represents useful data for the layout design part. In general, however, the considerations that lead to a functional hierarchy ignore important aspects of layout design. Logic decomposition and data path definition are just two examples of decisions that have a significant influence on the final layout. Many decisions that are functionally almost equivalent may have completely different consequences for the layout and its design process. Therefore, the layout design part may choose to ignore parts of the decomposition, initially or throughout. Nevertheless, it is assumed that after some clustering around seeds and some pruning the design is completely hierarchically structured. That structure is considered part of the initial data for layout design.

No layout design system is complete without algorithms to generate the geometrical specification of each cell (these algorithms are called *cell assemblers*), and algorithms for combining geometrically specified modules into supermodules. However, isolating the cell assembly tasks from the combination task is dangerous because of their mutual dependence. Taking this dependence into account by iterations over several design tasks is undesirable because of the time complexities involved and the convergence problems. In the YSC system, we have chosen an approach in which cell assembly is preceded by algorithms that in a top-down sequence create preliminary environments for submodules, so that part of the input for the cell assembler is the preliminary environment of the cell to be generated.

Cells can be of various types. The type determines the cell's flexibility (used in the creation of preliminary environments) and the cell assembler to be used for obtaining the complete layout. The most rigid cell type is an *inset* cell. Its layout and pin positions are fixed and stored in a library or are completely implied a priori by the algorithm generating the cell. The layout design system can only assign a location and an orientation to such a cell. Other cells have pointers to algorithms that can produce their layouts. These cells may have much more flexibility and therefore can be more easily

adapted to a given environment. The most flexible kind of this type of cell is a *macro*. These cells have a decomposition of their own into circuits of a particular family. They are the ones used primarily in the Yorktown silicon compiler for implementing combinational logic, not only because high degrees of flexibility are desirable in the context of layout design in a silicon compiler, but also because with the appropriate choice of family they usually outperform other implementation styles. In addition, there are cells generated by algorithms, depending on a few parameters, that construct special-purpose subcircuits such as ALUs, rotators, and adders. These cells have limited flexibility (e.g., they stretch in one direction).

THE RECTANGLE CONSTRAINT The ultimate task of a layout design system is to produce a *layout*—that is, a set of masks that completely specifies the geometry of the circuit. For present-day technologies, the geometrical specification of eight to fifteen masks suffices. Lithography techniques quite often require orthogonal artwork. This leads to masks that are unions of perpendicular rectangles. Rarely is a restriction to rectangles and combinations thereof detrimental, whereas the cell design algorithms and layout data bases profit from such a restraint. The rectangle is therefore the basic construct in the YSC layout system.

The rectangle constraint is also imposed for the compounds and cells of the hierarchy. Consequently, each hierarchy will be a rectangle dissection in the final layout, that is, a rectangle subdivided into nonoverlapping rectangles. The restriction to rectangles might seem rather arbitrary; however, a constraint on the shape of a region is of great value in determining the shape and positions of its parts. If these parts have a high degree of flexibility, then the restriction to rectangles is of little consequence since the parts can be fit into the environment by using their flexibility. On the other hand, choosing rectangles as the only constructs simplifies the formulation and derivation of the design decisions.

It is also expedient that the results of the layout design process are stored in a way that is compatible with the data representation delivered by previous design procedures. The initial data available in the layout design stage is hierarchically structured. We use a layout design procedure that preserves that structure, although refined and ordered. *Refinement* here means that a module with all its submodules can be replaced by a tree with the same root and the same leaves, but with a number of additional internal nodes.

DESIGN RULES To improve chances for successful integration of the circuit and to increase yield when the circuit goes into production, patterns are required to satisfy certain rules, the so-called *design rules*. A first classification distinguishes roughly two classes: (1) *numeric rules* that quantify spacings between patterns in a plane or in combinations of planes and (2) *structural rules* that enforce and prohibit certain combinations.

There usually are a large number of numeric rules. Very few, however, are critical to the layout algorithms. The rules are formulated as minimum

rules only, because it is assumed that the layout design techniques will try to keep the total chip small. Good algorithms working with these rules produce valid layouts in a wide range of values. Of course, the algorithms do not produce optimal layouts for all combinations of values in these rules, but they should produce acceptable solutions for all practical sets of values. In the YSC system, the numerical rules are stored as a vector. Specifying the technology causes values to be assigned to the components of this vector. The vector is accessible by all procedures in the layout design stage.

The structural rules are much more difficult because changes in these often require completely different decisions. These rules usually increase the dependence between different masks. This is particularly so if the metal layers are involved. Rules that forbid or enforce certain overlaps between patterns in the metal layer masks and other masks affect the wiring routines, because these routines are often based on generic algorithms solving some cleverly isolated interconnection problem. Introducing structural constraints often invalidates the assumptions made in the design of the algorithm. In the YSC system, the generic algorithm is usually followed by a mask generator in which the implications of the structural rules are incorporated. We hope that changes in structural rules will only affect the mask generator, which must be adapted in that case.

Floor Plan Design

SHAPE CONSTRAINTS For every cell in the hierarchy there is an algorithm that tries to adapt the cell to its estimated environment while generating its detailed layout. This preliminary environment must be created on the basis of estimates of the area needed by each cell, feasible (rectangular) shapes for it, and the external interconnections. The size and the shape of a cell are constrained by the amount and type of circuitry that must be accommodated in that cell. It is reasonable to expect one dimension of the enclosing rectangle not to increase if the other dimension is allowed to increase. Constraints satisfying that requirement are called *shape constraints*. Examples of shape constraints are shown in Figure 7.12.

Inset cells have piecewise linear shape constraints. Such constraints can be conveniently represented by a sequential list of their breakpoints. This

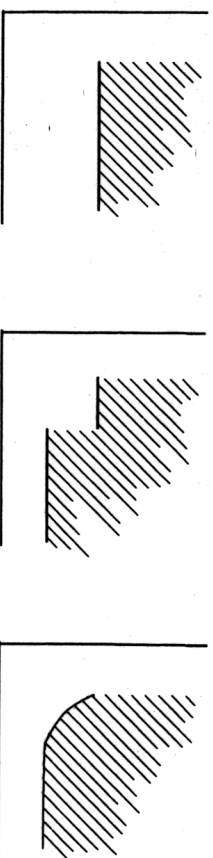


Figure 7.12 Examples of shape constraints. The set of feasible dimension pairs is indicated by shading. The examples are (a) an inset cell with a fixed orientation, (b) an inset cell with free orientation, and (c) a cell with a minimum area and minimum dimensions.

is not the case for flexible cells, and possibly other cell types occurring in practice. Of course, any shape constraint can be approximated by a piecewise linear bounding function with arbitrary accuracy. Usually, a piecewise linear approximation with three breakpoints will suffice, considering the limited accuracy of most area estimations.

Useful shape constraints of compound modules in the functional hierarchy can be derived in an obvious way from the shape constraints of its cells.

PLACEMENT VERSUS FLOOR PLAN DESIGN The estimation of the rectangle, in which the module is to be realized, is controlled by the shape constraints. Guidelines for the position of such a rectangle among all the other rectangles are contained in the functional hierarchy and in the incidence structures associated with the modules. In the context of layout, these incidence structures are called *net lists*.

Utilizing the data (shape constraints, net lists, and functional hierarchy), the cells must be arranged in a rectangle. This enclosing rectangle is often desired to be as small as possible; sometimes it is constrained in the aspect ratio or completely specified. If the cells were fixed objects, this would be a placement problem. However, in this context, the cells are allowed to take any shape allowed by its shape constraint. This generalization of placement is called *floor plan design*.

Both floor plan design and placement are guided by a number of objectives, which are not easy to formulate in a single function. This can be illustrated by the following typical combination of objectives. The first objective is primarily concerned with the realization of the interconnections. A common figure of merit is total wire length, often estimated by summing the perimeters of the rectangles that enclose all module centers connected to the same net. At the same time, the second objective is to give the cells optimal rectangular regions, minimizing such objectives as deformation, dead area, wiring space, and so forth. The first objective is topological in nature, while the latter is geometrical. To relate the two objectives, an additional refinement step (using an intermediate structure capturing much of the data affecting the topological objectives) is used in the Yorktown silicon compiler.

FLOOR PLAN TOPOLOGIES It has already been observed that in the final floor plan, the modules will be rectangle dissections in which each submodule is either a rectangle dissection itself, or, as in the case of cells, a rectangle. Creating a preliminary environment for the cells amounts to generating certain aspects of the rectangle dissection in which each cell is an undivided rectangle. Since the shape of the cells is not yet known at this stage, the geometrical details of the rectangle dissection cannot be determined. However, less restrictive aspects of a rectangle dissection are its neighbor relations—that is, which cells share a particular line segment in the dissection. The set of neighbor relations is called the *floor plan* of the rectangle dissection. A floor plan is useful information that can be generated at an intermediate stage of the refinement process. Usually, enough freedom is left for the cell-

assembling procedures after fixing the floor plan, and further decisions concerning the environment of the cells can be derived from it. Therefore, the first task will be designing a floor plan. A reasonable decomposition of that task, certainly in the light of the discussion in the previous section on the layout design environment, is to take one module at a time, starting with the root of the hierarchy and progressing downward such that no module is treated before its supermodule. This translates the hierarchies into nested rectangle dissections [Orte82a].

It is desirable to design a floor plan that minimizes the chip area while meeting the shape constraints of its cells. A concise and common method of representing the floor plan of a rectangle dissection is by a *polar graph*. Many floor plans designed in practice and all the floor plans of the successful, more specialized layout styles have polar graphs that are two-terminal series-parallel digraphs. The optimization problem described previously can be efficiently solved for this class of floor plans.

A first observation is that a floor plan that can be represented by a two-terminal series-parallel graph can also be represented by a more manageable data structure, namely an ordered tree. By restricting floor plans in this way, the floor plan design procedure essentially orders and refines the initial hierarchy, and the hierarchy and the layout structure can be consistently represented and stored in a natural way during the entire procedure. A rectangle dissection, with a two-terminal series-parallel polar graph, is a rectangle dissected by a number of parallel lines into smaller rectangles that might be dissected in the perpendicular direction. Such structures are called *slicing structures* (see Figure 7.1), and the associated tree is called a *slicing tree*. Each vertex of the tree represents a *slice*. Each slice either contains only one cell or is a juxtaposition of its *child slices*. In the latter case, that slice is said to be the *parent slice* of its child slices, and these child slices are the *sibling slices* of each other. The sibling slices are ordered according to their position in the parent slice (e.g., left to right and top to bottom).

When a (preliminary) geometry is associated with the slicing tree, the *longitudinal dimension* of a slice is the dimension it inherits from its parent slice. The other dimension is its *latitudinal dimension*. The latitudinal dimension of the common ancestor slice is the length of the side it transmits to its child slices. The other dimension is its longitudinal dimension.

IMPLEMENTATIONS There are several ways of obtaining slicing structures. A well-known method is the min-cut algorithm. If applied in its pure form, it leads to binary slicing trees. However, it can be extended to produce general slicing trees. The min-cut method does not use an intermediate structure that captures globally a large part of the topological aspects of the input. Each dissection divides the problem into smaller problems, but it is difficult to take into account decisions in one part when handling the other parts.

There are also methods that use an intermediate structure [Orte82b]. One such structure is a point configuration in which the topological properties of the input are somehow translated into a closely related geometrical concept,

namely distance, and since the configuration will be embedded in a plane, the geometrical concept is usually distance in the two-dimensional euclidean space. High connectivity is reflected in relatively short distances. A useful metric in this context is the so-called *dutch metric*. To obtain the distance between two modules under the dutch metric, first calculate a proximity by taking the quotient of the sum of weights of the net that connect the two modules together and the sum of the weights of the net that connect each of the modules with another module. Obviously, the proximity can never exceed 1. The distance is the square root of 1 minus that proximity. The weights of the nets can be used to grade the relative importance of nets. The distance space can always be embedded in a euclidean space, although seldom in the euclidean plane. However, it is straightforward to construct a two-dimensional point configuration in which the distances d_p are such that $\sum (d_i^2 - d_j^2)$ is minimum. This is done by determining the eigenvectors with the two largest eigenvalues of an associated matrix, the *schoenberg matrix*. Also by calculating a partial eigensolution, the weighted sum of distances can be minimized under the constraint that the configuration must have a certain spread. However, these are euclidean distances, and wiring is generally orthogonal, requiring a different metric. It is also difficult to take into account more detailed information (e.g., pin positions on the perimeter) in assessing the wiring objective.

Recently, a probabilistic algorithm has become very popular for handling combinatorial optimization problems [Kirk83]. This algorithm, called (simulated) annealing, can also be used for generating point configurations in the present context [Orte84]. The following formulation is illustrated in Figure 7.13. Each configuration allowed in the algorithm is represented by a pair of permutations over the set of modules. To obtain the score of that configuration, the two permutations are used to obtain the horizontal and vertical coordinate for each module. The horizontal coordinates, for example, are obtained by placing an interval between two neighboring elements of the horizontal permutation, proportional to the sum of the two areas of the associated modules. Thus the horizontal and vertical coordinates provide a point configuration in the plane for all the modules. The perimeter of a box that contains all the points representing modules connected by a particular net is used as an estimate for the size of that net. Several refinements (e.g., correction for nets connected to many modules, different weights for different directions, weights for nets, etc.) are possible. The sum of these size estimates is the score of the configuration. To generate a new configuration, a pair of modules is transposed in one of the permutations. The score function is then evaluated for this new state and is either accepted or rejected according to acceptance rules of the annealing algorithm. In this way, an optimal or near-optimal point configuration is obtained.

Properties of the final rectangle dissection are derived from this point configuration and the shape constraints. The topological considerations are taken into account by preserving relative positions in the point configuration and keeping modules close together if they are represented by points

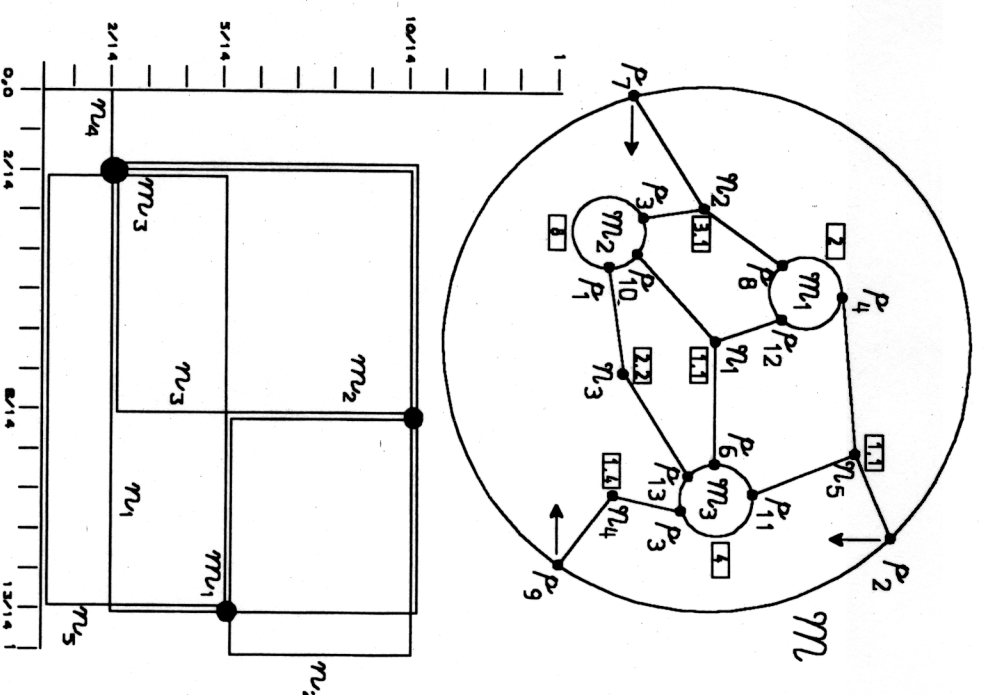


Figure 7.13 Determining the score of a configuration.

at a small distance from each other. The geometrical aspects are taken care of by keeping track of deformations implied by the dissections, for example. Shrinking is a method that efficiently achieves these goals for slicing structures. The discussion of shrinking will be limited to designs with exclusively flexible modules. For the most part, nearly square shapes are preferred for flexible modules because square shapes generally require less wiring space. Now, assume that a given slice with two or more submodules is to be sliced into a number of child slices. This slice is separated from its sibling slices by lines parallel to one axis, and the slicing must now be tried parallel to the other axis. Figure 7.14 is a visualization of a process

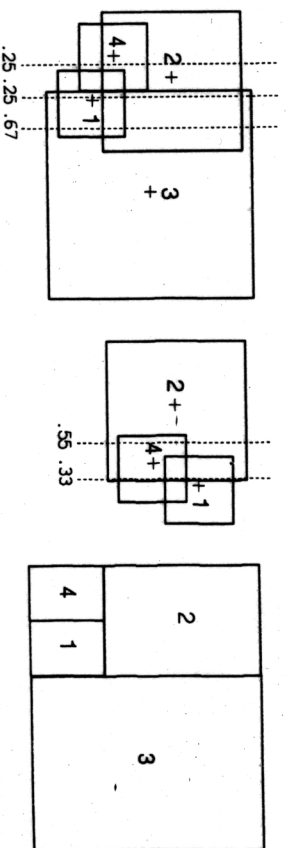


Figure 7.14 The determination of the shrink factors. In this example, the squares around the module centers are drawn in their position before the shrinking process. The amounts of shrinking necessary to keep the vertical dashed lines from intersecting are given for each line. After accepting the line with the highest shrink factor, the three modules at the left are treated in the same way.

of determining the shrink factor for possible slicing lines with the given orientation.

Think of a module as a square around the point representing it and with its sides parallel to the axes. The sizes of these squares are such that each pair of squares has overlaps and the ratio of their areas is equal to the ratio of the area estimates given to the corresponding modules. The squares are shrunk simultaneously, leaving their centers fixed and preserving the area ratios. At some point during this shrinking process, there will be a line with the given orientation that divides the modules into two blocks without intersecting any of the squares. The amount of shrinking necessary to reach that point is the *shrink factor* to be assigned to the corresponding line. Shrinking continues, and other lines with the same orientation and separating other blocks are determined along with their shrinking factors. The lines with relatively high shrink factors are candidates for slicing lines. The accepted lines partition the modules into groups. Each of these will be treated recursively in the same way. The selection of the lines usually considers other factors. One of these, *deformation*, will be discussed next.

The preferred shape for flexible cells is a square. However, in general it is not possible to accommodate cells as squares in a slice with the area being the area estimate for the supermodule. The deviation from the square can be measured by a simple function. The calculation of deformation on the basis of such a function starts with deriving the outer dimensions from the total area and the aspect ratio. One of these dimensions is inherited by the child slices from the parent. The other dimensions of these child slices are obtained by dividing the sum of the areas of the modules in each of them by the inherited dimension. Proceeding in this way, each slice inheriting the dimension calculated for its parent slice will yield the dimensions of all slices, which enables one to calculate the minimum amount of deformation (under the given function) incurred by each cell. Since the computation is performed top-down, this is applied during the slicing procedure. For

every decision, proximity information (shrink factors) and shape information (deformation) is available. For floor plans with a large number of modules, of similar size, the first slicing lines are mostly determined on the basis of proximity information. Later, when few modules are allocated in a slice, deformation usually dominates the selection. In the final stage, one may even consider determining the deformation incurred by all possible slicing trees.

FLOOR PLAN OPTIMIZATION Given the slicing tree and the shape constraints of the leaf modules, the best rectangle dissection corresponding with that tree must be determined. By *best*, we usually mean area or perimeter of the overall rectangle; however, a more general objective function, with the outer dimensions as arguments, can be used as well. Thus we can construct the smallest rectangle dissection with (1) a given slicing structure and given shape constraints for its undivided rectangles and (2) with or without constraints on the aspect ratio (e.g., a given aspect ratio, or a lower and upper bound on the aspect ratio). The time complexity of the algorithm is polynomial.

The shape constraint of a compound slice can be derived from the shape constraints of its child slices. In the final configuration, these child slices must have the same longitudinal dimension, which is the latitudinal dimension of their parent. The inverse of a compound's shape constraint is only defined on an interval in which the shape constraints of all its children are defined. Its smallest possible longitudinal dimension for a given feasible latitudinal dimension x is the sum of the values of the shape constraints of the children at x . So, the shape constraint of a compound is obtained by the addition of the shape constraints of its children in the interval in which they are all defined and by the determination of the inverse of this resulting function. These operations are easy for piecewise linear shape constraints, represented by a list of their breakpoints ordered according to the respective longitudinal dimensions (see Figure 7.15).

The ability to obtain the shape constraints of a slice by adding the shape constraints of its child slices and inverting the result enables us to obtain the shape constraint of the entire enveloping rectangle. Appropriate objective functions (quasi-concave functions that are monotonically increasing in their arguments) always assume their minimum value at the boundary of the associated shape constraints. This is a consequence of the monotonicity of shape constraints and the objective function. For piecewise linear shape constraints, that minimum will be assumed at one of its breakpoints. To find an optimum pair of dimensions for the common ancestor slice, the objective function only has to be evaluated at the breakpoints of its shape constraint in the convex set of permissible pairs.

Given the longitudinal dimension of a slice and its shape constraint, the latitudinal dimension can be found by evaluating the shape constraint at the given longitudinal dimension. After deriving the shape constraint for the common ancestor and determining a dimension pair for which the objective function assumes a minimum, the longitudinal dimensions

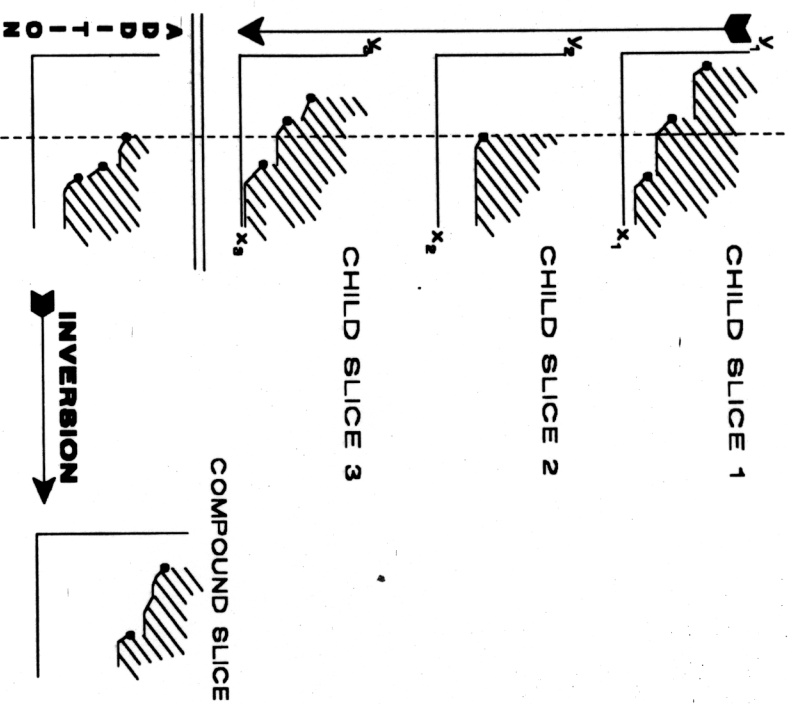


Figure 7.15 Addition and inversion of shape constraints.

of its children are known. For each of them, the latitudinal dimension, which in turn is the longitudinal dimension of its children, can be calculated. Continuing in this way will yield the dimensions of all slices in the configuration.

The algorithm for finding the optimum dimensions for all slices consists of three parts:

1. Visit the nodes of the slicing tree in depth-first order; and, just before returning to the parent, determine the shape constraint by adding the shape constraints of its children and inverting the result.
2. Evaluate the objective function for each breakpoint of the derived shape constraint of the common ancestor, and select a dimension pair that corresponds to the smallest value.
3. Visit the nodes of the slicing tree in depth-first order; and, before going to any of its children, determine the latitudinal dimension by evaluating its shape constraint at the inherited longitudinal dimension.

Completing all three steps will yield the dimensions of all slices in an optimum configuration for the given floor plan and cell shape constraints. To determine the position coordinates of the slices from these dimensions and the floor plan is straightforward. It is also easy to determine what orientation the inset cells can have in this optimum configuration.

NET ASSIGNMENT The more complex the circuit is, the more dominant the wiring is in the final layout. Although part of the wiring can be realized on top of active devices, particularly when there is more than one metal layer, a considerable portion of the chip is used exclusively for wiring (wiring space). This part of the chip will be implemented using "wiring cells." If these wiring cells are realized in rectangular regions, the wiring space can be seen as the union of nonoverlapping rectangles. The selection of the rectangles for the wiring space affects the efficiency of the wiring procedures. It constrains the sequence in which the wiring can be generated, determines the algorithms that can be used for the wiring, and determines the number of different algorithms needed to perform that task.

Again, slicing structures have considerable advantages over general rectangle dissections. First, because they imply a decomposition of the wiring space into the minimum number of rectangles: the slicing lines are, in effect, the wiring rectangles. To distinguish these undivided rectangles from the ones that correspond to the cells in the functional hierarchy (in the slicing tree both kinds are represented by leaves), they are called *junction cells*. Second, a feasible sequence for generating the wiring can be easily derived from the slicing tree. Third, all these rectangles can be wired by using the same kind of algorithm, usually called a *channel router*.

Since the wiring consumes a high percentage of the total area, it is useful to have early estimates for this space so that the sequence of floor plan calls can take these estimates into account when designing the nested floor plans. Several objectives may be important in realizing the interconnections, and many of these are directly related to the length of these nets. This immediately presents a problem since a floor plan is a topology rather than a geometrical configuration. Yet, in order to measure the length of a net, a metric is necessary.

A commonly used structure for approaching these problems is the plane graph determined by the rectangle boundaries in the rectangle dissection. Each rectangle corner is a vertex, and the line segments between them are the edges. This graph depends on the geometry of the rectangle dissection, and this geometry is not known in the floor plan design stage. A closely related graph can be defined for slicing structures. For there is a one-to-one correspondence between the junction cells of rectangle dissections with the same slicing structure. Also, the relation generated by the T-intersections between junction cells is invariant over the rectangle dissections with the same slicing tree. Based on these observations, an environment can be created in which any steiner tree heuristic can be used for routing each individual net.

The refinement steps described in the previous section determine a topology for every compound of the functional hierarchy. These topologies are restricted to slicing structures, so floor plan design replaces each subtree, of which the vertices represent a certain module and all its submodules, with another tree, of which the root represents the selected module and the leaves its submodules. Another type of refinement is the replacement of a leaf in the functional hierarchy (a *function cell*) by a tree decomposition. The reasons for not having this decomposition in the initial tree can be quite diverse. For example, the decomposition suitable for the functional design may be far from optimal for layout design. In that case, such a hierarchy is pruned, and a data base problem must be resolved when this happens. However, most often there is no need for further decomposition from the functional design point of view, but flexibility is increased if the layout design procedures use some inherent decomposition.

Algorithms for designing cells, possibly using such a decomposition, are called *cell assemblers*. The task of a cell assembler is to determine the internal layout of its cell (with respect to a reference point in that cell's region) on the basis of a suitable specification and data about its environment. There may be quite a diversity of cell assemblers in a silicon compiler system. The applicability of the silicon compiler is highly dependent on the set of implemented cell assemblers. While most of the decisions during floor plan design are largely technology independent, cell design is dominated by the possibilities and limitations of the target technology. The numeric design rules are stored as numbers, the value of which is assigned to certain variables in the cell assembler. The structural rules are mostly incorporated in mask generators.

The layout of a slice is obtained by first determining the layout of all its child slices, except junction cells, and then by calling the appropriate assemblers for the junction cells. Visiting the slicing tree in depth-first order and performing the above operations when returning to the parent slice enables the program to determine the *chip coordinates* (coordinates with respect to a unique point on the chip) of all layout elements in the parent slice before leaving the corresponding vertex. The translation of that result into the rectangles of the various masks is also performed at this point. This translation is straightforward.

FUNCTION CELLS The task of a cell assembler is simple for inset cells since the internal layout is stored in a master or user library. From the topology (determined in the floor plan design process) and the shape constraints of all cells (including the junction cells), an estimate of the rectangle of that module has been derived. Reasonably accurate data about the position of the nets to be connected to that cell have been generated by the net assignment process. On the basis of that data, the assembler must decide which orientation is given to the inset cell and how it is to be aligned with its sibling slices.

For function cells, where the layout is not stored in a library, the layout may be still largely implied by the specification. For example, if a cell is a programmed logic array, the specification is either a personality matrix or a set of Boolean expressions. In the former case, the assembler does nothing but translate into a layout, so the result is obtained in the same way as the stored inset cells. If the array is specified by a set of Boolean expressions, the assembler uses a so-called *pla-generator*. The shape of the resulting array is still difficult to control; but the pin positions can be adapted to the results of the net assignment, which are performed during the floor plan design stage. Some sophisticated *pla-generators* use techniques such as row and/or column folding to make the area of the array smaller. This constrains the choice in pin positions considerably and might lead to a higher area consumption because of the complex wiring around that array. A *pla-generator* in a cell assembler should be able to take the results of the net assignment into account [Demi85].

Regular arrays (e.g., programmed logic arrays and memories) are heavily constrained, and it is not easy to manipulate the shape of the array. Other cells, called *macros*, have a natural or given decomposition that can be used for that purpose. They are decomposed into circuits that either are selected from a prespecified catalogue or can be designed with a simple algorithm from a function specification. The reason for having such a macro as a cell assembler, rather than part of the floor plan design, is that the circuits have certain properties that make special layouts very efficient. For example, the catalogue or the simple algorithm may have a constraint that gives all cells in the macro the same width and the same positions for power supply and clock pins. In that case, a *pluricell* layout style is suitable for the macro. It forces the cells to be distributed over columns; but the number of columns can be chosen freely. Therefore, the aspect ratio of the macro can be influenced. Also, the pin distribution around the periphery can be prescribed on the basis of environment data. In the YSC system, we use predominantly a cell generator that implements combinational logic in pluricell style [Bray 84c].

Pluricells are similar to polycell or standard cell layouts as can be seen in plate 1. Both have their circuits placed in columns, either with two sides accessible for signal nets or with one side accessible and the other abutting cells in a neighboring column. The latter is chosen in the YSC system. Supply and clock lines are realized by abutting subsequent cells in each column. The signal nets are realized mainly in channels between the (double) columns. Two wiring layers, polysilicon and first-level metal, are used to realize the nets in the channels.

One important difference is in the way that the individual cell layouts are obtained. Polycell layouts find their cells in a master library that is built and maintained independently from the individual applications. Pluricell cells are constructed ad hoc. In case of a domino circuit, the switching network is realized as a one-dimensional transistor array along the edge of the channel. Such an array is formed by crossing a strip of n -diffusion with

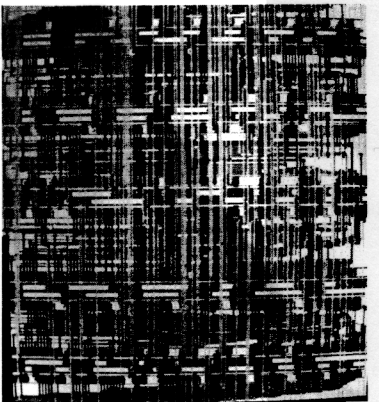


Plate I

polysilicon tracks carrying the associated logic signal. This is not always possible without modifying the network, that is, duplicating transistors, introducing transistors with grounded gates, changing sequences in series connections, and so forth. As long as the logic function and the electric properties of the result are still acceptable, these modifications may be used for making the diffusion strip short and the number of tracks to realize the network topology small. The rest of the circuit is realized between the power supply and ground lines that run parallel to the diffusion strip on the side opposite from the channel. The sizes of those devices are adapted on the basis of critical path information generated by the algorithms in the timing optimization of the Yorktown silicon compiler. Between those two parts of the circuit runs the clock line. Any number of additional first-level metal tracks can be allocated between that line and the ground line of the inverter. However, one is always there without increasing the circuit area. This is used primarily for sense node. Another track, which is free because of the spacing rules, is between the clock line and the transistor array and is used for making the output available via first-level metal. If this cannot be used, the output will enter the channel via polysilicon (see plate II).

Of course, other types of circuits can be incorporated in pluricell layouts as long as their periphery is compatible (i.e., pitches of clock and supply lines are matched, signal pins are on the correct side). Carrying this to the extreme with completely specified circuits from a library leads to polycell layouts. Therefore, pluricells can be seen as a generalization of polycell styles.

To connect parts of nets in different channels, polycell layouts use either so-called feedthrough cells or the net may leave the channel at one end and make the required connection outside the macro. In pluricell layouts, the feedthrough connections are made by second-level metal tracks. This also allows signals to enter the macro from all four sides. Each signal requiring a second-level metal connection has exactly one track assigned to it. Channels in a pluricell layout are different from channels resulting from

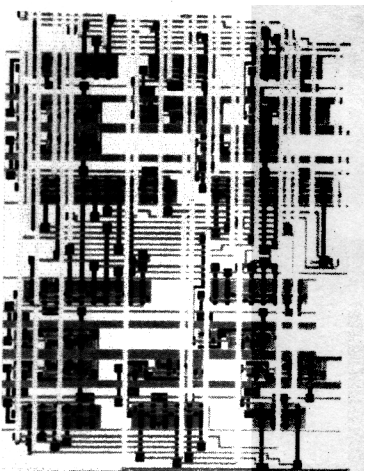


Plate II

a *classical* channel router, because signals can be dropped into the channel by contacts between first-level and second-level metal. Thus there are three ways of bringing signals into the channel, whereas the formulation has only two—the sides and the ends of the channel. The channels shown in plate II differ in one more aspect from classic channels. Design rule sets have developed smaller and smaller widths for metal and polysilicon tracks. However, the area taken by contacts between polysilicon and first-level metal has not decreased as fast. Currently, with minimum pitch wiring under the presence of these contacts, there is enough space for a first-level metal track in both directions. One router implemented in the YSC makes use of this fact by allowing twice as many tracks in the channel as would be allowed by a classic channel router, but contacts in neighboring tracks are not allowed. Moreover, in latitudinal direction, the router uses the extra space for adjusting the position of a net by first-level metal jogs. Therefore, the wiring space is reduced by a factor of about two, as compared to what a classic channel routing would produce.

The results of the logical synthesis and timing are available to the layout program. Also, information about desirable device sizes is given to the layout part. For example, functions in a critical path may be marked so that the associated domino circuit gets a more powerful inverter. All this information is stored in a specific format readable by the layout procedures. After the design of the whole chip has entered the layout stage and after floor planning and the global net assignment, this information is read to obtain information about the environment of the macro. The construction of a pluricell style layout is performed in the following steps.

1. The topology of the switching network. The construction of the domino circuit in a pluricell layout is executed by first performing a topological analysis of the logic function and encoding the result in a compact vector from which periphery data, such as dimensions and pin positions, can be derived. Domino circuits are specified as factored logic functions. These have an obvious translation into a two-terminal series-

parallel network of switches. The translation of a switching network in a linear transistor array is illustrated in Figure 7.16. This translation process is possible for the network in Figure 7.16, because the graph of the switching network has a special property; only two vertices have an odd degree. Switching networks that do not have this property cannot be translated into a linear transistor array by the process illustrated in Figure 7.16. However, different graphs may be used to realize the same logic function. In the case of switching functions with a series-parallel graph, the commutativity of the AND operation makes the sequence of the series connections irrelevant. Also, duplicating a transistor and connecting the two in parallel does not change the switching function. Thus any series-parallel logic function can be transformed to one that has at most two odd nodes. An interesting problem now is to find one that realizes the given logic function with the minimum number of transistors, using the permutation of the series combinations and duplication of transistors. This corresponds with a transistor array of minimum length under the given constraints. In practice, however, we also want to restrict the number of metal tracks required to connect the various diffusion regions to realize the required graph. Typically, we try to keep the number of metal tracks to three in order to keep all the circuits in a column of the pluricell at the same width. Using an APL

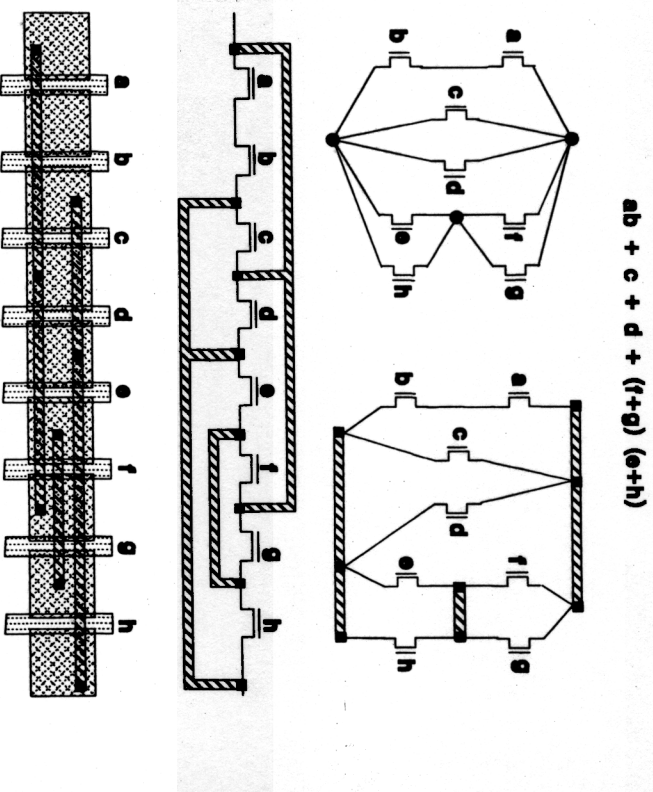


Figure 7.16 Translating an AND-OR switching function in a linear transistor array.

implementation, all the circuits of a controller represented in plate I were created in less than one second of CPU time on an IBM3081. The controller had 81 single output switching functions, which were realized by almost 800 transistors in the linear arrays.

2. Placement in (double) columns. Once the circuits are designed, a placement procedure consistent with the structure rules of the slicing structure is used. First, a configuration of points, each representing a domino circuit, is created in the plane on the basis of proximity data such as net length and delay. Slicing will preserve the relative position within each slice. The determination of the final slicing structure is based on approximately equal-length double columns.

3. Second-level metal assignment. Once the distribution of the circuits over the columns is known, their position in the column is fixed. This enables the program to determine the intervals at which the nets must occur for each pair of double columns in order to connect the corresponding pins along the sides of the channel. If the same net occurs in several channels or enters the macro from one of the directions perpendicular to the columns, a second-level metal track is reserved for that net. The position of these second-level metal tracks is optimized with respect to net resistance and net size as permitted by the environment of the macro. This problem has been formulated as a bipartite matching problem.

4. Mask generation. After finishing the second-level metal track assignment, the data for the channel-routing problems are complete. Pluricell floor plans are, as a slicing structure, always represented by a five-level slicing tree. The leaves of that tree represent the circuits and the channels. By generating the mask data of these leaves in the same sequence as they are visited by a depth-first search over that tree, the coordinates of the elementary mask elements are immediately assigned. Consequently, circuit geometry and wire specification are interleaved, and in one pass the mask data are complete.

Decompositions in macros occur very often; however, for special cells that are frequently used, it sometimes is worthwhile to implement a special algorithm producing a highly optimized layout. In word-organized digital systems, these special cells often process a number of bit vectors. The layout as a whole may benefit from aligning these cells so that the wires carrying these bit vectors do not have to be matched to the pitch of each individual cell. Also, wires that pass over such a cell without making any contact must be accommodated. These requirements imply a certain kind of flexibility, such as stretchability and variable pitch. If possible, such an algorithm should be able to produce these highly optimized cells for several bus dimensions and a range of performance requirements.

JUNCTION CELLS. Junction cell assemblers are closely related to channel routers [Burs85], because of the way they are isolated and called in the layout process. The junction cell is a rectangular area of which the latitudinal sides are part of the longitudinal sides of junction cells that are represented

in the slicing tree by vertices closer to the root. When the assembler is called in a bottom-up sequence for a certain junction cell, the longitudinal coordinates of the entering nets are known. There are several ways a net may enter the junction cell: from the longitudinal sides of that cell, from a higher metal layer, from the latitudinal sides, and perhaps in still other ways. The task of the assembler is to realize all the required interconnections in a rectangle with the smallest possible latitudinal dimension. The longitudinal dimension must be commensurate with the latitudinal dimension of the parent slice. Therefore, increasing the longitudinal dimension of the channel should be avoided if possible.

7.2.5 Verification Tools

The design synthesis should be a strictly automatic process. However, its effectiveness depends on a compatible debugging environment where the "program" can be run and errors corrected. The YSC system is equipped with a number of facilities for accomplishing this. Two gate-level verification tools are described here: a logic simulator and a timing verifier. With the logic simulator designs can be verified by inspecting the contents of the registers for a given input stream. Information about the arrival times of the signals can be obtained by means of the timing simulator. This information can be used for verifying whether the circuit operates within the required cycle time as well as for providing timing information to the timing-performance optimization tools.

The Gate-Level Logic Simulator

A gate-level logic simulation provides a means of verifying the logic behavior of the circuit being designed. The simulator allows the monitoring of the logic values (1 or 0) of some of the variables in the original description. The values of the variables that are always observable are those that are latched into registers (architected or not), those corresponding to I/O pins, and the primary inputs and/or outputs of the blocks of the hierarchical circuit structure (e.g., I/Os of combinational modules).

Gate-level logic simulation uses the information of the structure of the circuit provided by the structural synthesis step and the logic representation of the combinational modules generated by the logic synthesis step. The behavior of these modules is characterized by the Boolean function that they implement. This information is easily accessible after logic synthesis. The logic simulator assumes zero-delay in each circuit block. The behavior of special-purpose modules, or circuits, external but interacting with the hardware being designed (e.g., a memory unit) may be modeled by an APL program.

Simulation consists of loading the registers with a set of logic vectors representing the "initial conditions" and then feeding the circuit primary inputs with a stream of input data. The stream of values of some (or all) the observable variables may be monitored or stored. In particular, if the circuit being designed has a synchronous operation, the simulation may be halted at each cycle.

Simulation may be used to verify the correctness of the input to the YSC system. In the case of processor design, it provides a mechanism for verifying the architecture. In particular, architectural verification may be achieved by "running" a stream of instructions and by monitoring the circuit primary outputs and the values of the architected registers.

The Gate-Level Timing Simulator

The gate-level timing simulator provides information about the propagation delay through the circuit. It serves two purposes. First, it checks a circuit against its specifications. For sequential synchronous circuits in particular, it is important to know whether the circuit meets the cycle time specification. Second, it evaluates the signal delays needed by the time-performance optimization programs.

The delay computation requires several assumptions on the circuit. An accurate timing evaluation requires not only the knowledge of the circuit topology, but also layout data such as the length of the interconnections. However, it is important to obtain timing information before the final layout is available. Such information may be used for early verification or for driving the layout procedures to produce images that satisfy timing requirements. For example, if timing information is available before the floor plan is designed, critical nets can be identified and incorporated into the objective function of the floor plan.

The timing simulator of the YSC system has been designed for different stages of the synthesis process, using all the information that is available at that stage. Just after the logic synthesis and before the layout stage, the wiring load is either estimated from the fan-out information or neglected. After the floor plan has been determined, the wiring length is computed on the basis of the position of the modules. Of course, when the layout has been completed and the wiring lengths are known exactly, the simulator can provide more accurate timing data.

The timing simulator is capable of simulating combinational circuits or the combinational portion of sequential circuits, including multiple-clock circuits. Combinational logic is described to the simulator as an interconnection of *logic gates*, by means of *nets* carrying *logic signals*. The hierarchy in the circuit structure is not required to explain the timing model. Therefore, we can assume that there is no hierarchy for the sake of this description. However, the hierarchical information is retained in the actual implementation of the simulator for computational efficiency.

The circuit being described has a set of primary inputs and outputs. The primary inputs to this network are also interconnected to the logic gates by means of nets. The primary outputs are identified by the gates generating the corresponding signal. We consider each signal stored in a register as both a primary input and output to the combinational subsystem. Each logic gate is assumed to be unidirectional because of the implementation technology. Therefore, we can associate a source and one (or more) sinks to each net. The interconnection can be modeled by a direct graph $G(V, A)$, whose node set $V = \{v\} = V^s \cup V^r = \{v^s\} \cup \{v^r\}$ is in one-to-one

correspondence with the set of logic gates (V^g) and primary inputs (V^i) and whose edge set A is in one-to-one correspondence with the source-sink pairs of the nets. The circuits synthesized by the logic synthesis stage are unidirectional to avoid race conditions, that is, $G(V, A)$ is acyclic. A node v_i is said to be a *predecessor* (successor) of gate v_j if there is a directed path from v_i to v_j (from v_j to v_i) in $G(V, A)$. A predecessor (successor) is said to be direct if the path has a length of 1.

The signal propagation is modeled by associating a *propagation delay*, $d(v^g)$, to each logic gate or, equivalently, a weight to each node in the set V^g . The propagation delay through a physical gate is modeled by an empirical equation. Although the procedure for computing the delays is fairly general, the propagation delay model for a gate depends on the target implementation technology. Delay equations for different logic families and technologies can be obtained by: (1) determining a certain number of circuit parameters that characterize the gates (e.g., gate sizes, capacitive loading, etc.); (2) simulating a wide set of gates with different parameters using a circuit simulator such as ASTAP to compute the propagation delay; and (3) using regression analysis to tabulate the delay as a function of the characteristic parameters. Therefore, the propagation delay $d(v^g)$ can be computed quickly from the parameters describing the gate and its interconnection.

For each node, we compute a *data ready time*: $t(v^g)$, $i = 1, 2, \dots, |V^g|$. The data ready time of a gate is the time at which the signal generated by that gate is ready. Similarly, we associate to each primary input a data ready time: $t(v^i)$, $i = 1, 2, \dots, |V^i|$. For our purposes, we synchronize the computation of the data ready times to the system clock. We assume the data ready time to be zero for each primary input corresponding to a register. The data ready times of the remaining inputs are set to the delay of the corresponding input signal with regard to the system clock. (The extension to multiple-clock systems is straightforward.) The data ready times at the logic gates can be computed by tracing forward the signal propagation—that is, by computing:

$$t(v_j) = d(v_j) + \max_{k \in K} t(v_k) \quad K = \{k | (v_k, v_j) \in A\}$$

for each node v_j corresponding to a gate in a sequence consistent with the partial order represented by the graph.

The gate-level timing simulator computes the set of data ready times for all gates in the circuit. It allows comparison of the computed arrival times of the primary outputs with the expected arrival time. With this information, the user can determine whether the circuit being synthesized satisfies the circuit specifications or not. It also provides the basic information for the timing optimization procedures.

7.2.6 Timing Performance Optimization

We present here some algorithms to improve the timing performance of the circuit being designed by the YSC system. Circuit performance is related to

the worst-case propagation delay of signals between two register boundaries, because the system clock must be adjusted to allow the arrival of each signal at the destination registers within the clock cycle. In this context, the optimization of circuit performance is equivalent to the minimization of the critical path delay. The circuit timing depends on the structure of the circuit (e.g., register boundaries). Therefore, timing optimization can already be considered at the structural synthesis level (see section 7.2.2.). We assume now that the global interconnections among combinational logic, registers, and I/Os are frozen, and we survey timing optimization techniques that are compatible with this assumption. We refer the interested reader to [DeM86] for the details as well as for a set of references to other work in the field.

The logic synthesis stage in the YSC system can be programmed to optimize the silicon area taken by each combinational module, the timing performance of a module, or both. Logic synthesis of each combinational module is performed one module at a time, and timing information during synthesis is limited to the domain of the module being synthesized. A gate-level timing simulation of an entire circuit can be done only after all its combinational logic blocks have been synthesized as an interconnection of gates. Since the measure of the timing performance is not an additive function (e.g., silicon area), the circuit being synthesized may not be optimal in terms of timing performance.

Similarly, the layout design stage (in particular, the floor plan design and the module and gate placement) can be programmed to optimize some figure of merit of the design, such as total wiring length or wiring density. Timing considerations may be included in the layout, by considering, for example, the length of the wires carrying signals that are critical to the circuit performance. However, these critical wires can be identified only after a timing simulation, the accuracy of which improves as more information about the actual layout is available. Similar considerations apply to the selection of the sizes of some devices in the macrocells. The timing performance can be optimized by choosing the gate sizes appropriately; however, the optimal choice depends on the structure of the logic network.

For these reasons, we consider a global approach to timing optimization which involves operations at the logic, topological, and layout levels of circuit description. This approach exploits the hierarchical circuit representation used by the YSC system as a collection of modules. In particular, at the logic level, timing optimization is achieved by modifying the internal structure of the logic gates and their interconnection inside each combinational module. At the topological level, we position (or reposition) the modules and, as a consequence, the corresponding gates; this reduces the delay on the wires along the critical paths. At the layout level, we select the gate sizes to improve the switching speed. These operations are interlaced with the synthesis steps of the YSC system and can be seen as the "code optimizer" part of the compiler that may be invoked when compiling circuits with critical timing performances.

In principle, the strategy for timing optimization can be viewed as an

iteration of two steps: (1) evaluate the critical path delay; and (2) modify the circuit appropriately. These steps are repeated until a satisfactory performance is obtained. Because of the relations between the synthesis steps and the accuracy in the delay evaluation, the timing performance optimization is done as a stepwise refinement. The data ready times are estimated first, after the logic design phase, by neglecting the capacitive load due to wiring through the use of the gate-level timing simulator described previously. The corresponding logic representation of the circuit optimizes the number of gates and their internal complexity, which correlate to the optimization of the silicon area. Similarly, the devices are assumed to have minimum size, because this choice corresponds to a minimal area implementation. The results of the timing estimate can then be used to identify critical nets that can be assigned an appropriate weight in the objective function of the floor plan design. After the floor plan is designed and the modules are placed, a more accurate estimate of the delays can be obtained, because the positions of the modules are known. If the performance is not satisfactory, the circuit may be redesigned for optimal performance by:

1. Resizing the active devices
2. Resynthesis of the combinational modules
3. Repositioning the modules

These operations are iterated until a satisfactory performance is achieved or no improvement is detected (see Figure 7.17). Other figures of merit in the design, such as silicon area and wiring length, are also taken into account.

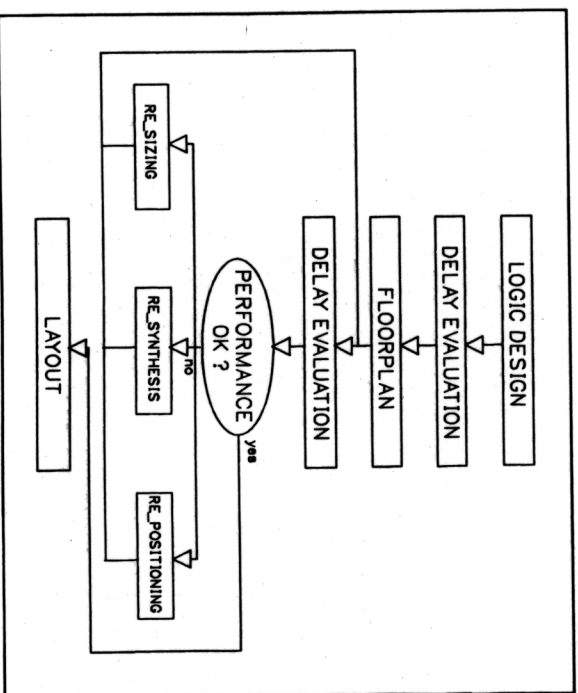


Figure 7.17 Timing optimization in the YSC system.

The performance-oriented redesign algorithms trade off these figures of merit for a faster timing performance. Due to the complexity of this approach and the interrelations among the effects of the changes at the logic, topological, and layout levels, it is not possible to guarantee an optimal procedure in rigorous terms. However, the heuristic procedures used for resizing, resynthesis, and repositioning have shown to be very effective.

Critical Paths

For a given circuit description, the gate-level timing simulator is used to compute the signal delays at each gate of the circuit. For this reason, we will use the terminology and notation introduced in the section on the gate-level timing simulator throughout this section. Important information about signal timing includes the slacks of the signals generated by the gates (also called slacks at the gates). The *slack* of each primary output (or at the gates generating a primary output) is defined as the difference between a chosen time $t(v_i)$ (e.g., the minimum system cycle time) and the computed data ready time $\hat{t}(v_i)$. For the other gates, the slack $s(v_i)$ at node v_i is defined to be:

$$s(v_i) = \min_{j \in J} \{t(v_j)\} + \max_{k \in K} \{\hat{t}(v_k)\} - \hat{t}(v_i)$$

$$J = \{j | (v_i, v_j) \in A\} \quad K = \{k | (v_k, v_i) \in A\}$$

The slack at each gate measures how much additional delay each signal can tolerate, while satisfying the relation $\hat{t}(v_i) < t(v_i)$ for each node v_i corresponding to a primary output signal. The slacks can be computed by tracing the signal propagation backward in the circuit—that is, by computing the slack at each gate in a reverse order consistent with the partial order represented by graph $G(V, A)$:

Let $\epsilon \geq 0$ be an arbitrary constant. The set of *critical nodes* is the subset of nodes $C_\epsilon = \{v \in V | s(v) \leq \epsilon\}$. The *critical graph* $H(C_\epsilon, B)$ is the subgraph of $G(V, A)$ induced by C_ϵ . A *critical section* (of the critical graph) is a maximal subset of nodes $\mathcal{C} \subseteq C_\epsilon$ with no mutual predecessor/successor relations. A *critical path* is a maximal directed path in $H(C_0, B)$. Any critical path intersects any critical section.

The meaning of critical node, critical graph, and critical path depends on the choice of $\hat{t}(v_i)$ at each node v_i corresponding to an output signal and on the parameter ϵ . Let $\hat{t}(v_i)$ be the required circuit cycle time (or a required arrival time for circuit output signals), and let ϵ be a number taking into account safety margins and tolerances. Then, the circuit meets the timing specification if and only if the set of critical nodes is empty. If this set is not empty, its elements indicate those gates for which data ready times are to be reduced to meet the timing specifications. Reducing the data ready time at the noncritical nodes does not help in meeting the circuit timing specification. Therefore, a timing optimization procedure may concentrate its efforts on the set of critical gates (nodes).

The concept of critical nodes and a critical graph is also useful in detecting those gates that limit the timing performance of a circuit, regardless of a

timing requirement. Let us consider the case in which the circuit output nodes of interest are latched into registers. A design strategy may try to achieve the best timing performance of that circuit and then adjust the system clock accordingly. In this case, let t^* be the largest data ready time among the circuit primary outputs of interest. The system clock period is bounded from below by t^* . Timing performance optimization aims at reducing t^* . Let $h(v_i) = t^*$ for each node v_i corresponding to an output signal. The critical graph $H(C_0, B)$ is the set of nodes having the property that any independent variation in the data ready times of any critical section implies a variation in t^* . Therefore, the nodes (gates) along the critical graph are "critical" because the timing performance of the circuit (limited by t^*) can be improved by decreasing the data ready time at critical nodes. This can be done, for example, by reducing the propagation delays at those nodes. Note also that a variation of the data ready time of a critical node v_i may influence t^* by very little if the direct successor of that node has a noncritical direct predecessor v_j , such that $|h(v_j) - h(v_i)|$ is small. In other words, node v_j is "almost critical." Considering "almost critical" nodes is important in a timing optimization procedure, as far as computational efficiency is concerned. For this reason, "almost critical" nodes are made "critical" by choosing $\varepsilon > 0$. The choice of positive values for parameter ε widens the set of critical nodes.

Although the definition of the critical nodes and critical graph is fairly general, we consider in the following sections the problem of minimizing the maximum data ready time t^* ; therefore, we choose $h(v_i) = t^*$ for each node v_i corresponding to an output signal, and we set ε to a proper fraction of t^* . We then use critical sections to determine those gates for which data ready times must be decreased to reduce t^* .

Delay Modeling and Electrical Considerations

We consider here only one family of circuits, namely dynamic CMOS circuits, operating in the domino mode. We consider only the *evaluation phase* of the domino cycle—that is, we assume that the timing of the *precharge phase* is correct. The gates consists of (1) a series-parallel connection of transistors that provides a discharging path to ground from a precharged node and (2) one (or more) *driver*, implemented by a static CMOS inverter. The size of the device implementing the driver can be adjusted by changing its width. The width is a linear function of the parameter w , the device length is kept constant. The sizes of the devices of the discharging path are kept constant, because the capacitance of the sense (precharged) node is negligible with regard to the gate and wire capacitance connected to the driver output.

The propagation delay through a physical gate is modeled by a delay equation that depends on a set of characteristic parameters, namely: (1) the size w of the drivers; (2) the capacitive loading c at the output, which in turn depends on the fan-out of the gate and the wiring capacitance to ground; and (3) the structure of the discharging path and in particular the maximum number of devices in a discharging path l . It is convenient to express the

propagation delay as $d = \alpha(w) + \beta(w)c + \gamma(w)l$. The coefficients α , β , and γ are tabulated for a finite number of values of w . These coefficients are computed by simulating a wide set of gates with different parameters w , c , and l with a circuit simulator, such as ASTAP. Note that l is known for each gate after having completed the logic design phase of the silicon compilation process. The parameter w is set to its minimal value during logic synthesis; it is then adjusted appropriately by the resizing procedure. The evaluation of the capacitive load c on a gate requires the knowledge of the gate fan-out and the wiring load. The fan-out information is also known after the logic design phase. The wiring information is known exactly only after the layout design is completed; an estimate is used before.

From the model of the circuit and its propagation delay, it is clear that the timing performance can be improved by changing the sets of parameters $\{w\}$ (driver size), $\{c\}$ (load capacitance), and $\{l\}$ (gate structure) and/or the structure of the graph $G(V, A)$. Device resizing aims at optimizing the set of parameters $\{w\}$. Circuit resynthesis improves the timing performance by changing the structure of the graph $G(V, A)$ and, as a by-product, the parameters $\{l\}$ and $\{c\}$. The module replacement aims at speeding up the circuit by modifying the parameters $\{c\}$ through changing the wire lengths.

Device Resizing

Before introducing the device sizing strategy, it is important to consider the particular delay model for domino CMOS circuits and how the device sizing affects the propagation delay. As pointed out in the previous section, the only design parameter of interest is the width of the driver. We limit the choice of the driver device widths to a finite number for practical reasons. Therefore, we assume that the parameter w can take a finite set of values $\{1, 2, \dots, p\}$. In our delay model, the functions $\beta(w)$ and $\gamma(w)$ are monotonically decreasing with w , while $\alpha(w)$ is monotonically increasing with w . This is consistent with the facts that the current flowing through the MOS drivers is directly proportional to the gate width (the higher the current, the lower the propagation delay) and that the driver gate capacitance increases as the size increases (the higher the gate capacitance, the higher the propagation delay). As a net result, for typical values of c and l , the propagation delay is monotonically decreasing with w —that is, $\Delta^d(w) \equiv d(w + n, c, l) - d(w, c, l)$ is always negative for each node $v \in V^d$ and $n > 0$. Note also that the gate input capacitance does not change with w , because only the driver size changes. Therefore, an increase in the parameter w is always beneficial in reducing the propagation delay through the corresponding gate and does not increase the data ready time at any other gate—that is, the mapping $w \rightarrow t^*$ is monotonically decreasing.

With this model, it is clear that a minimum value of t^* can be achieved by selecting the maximal size w for each gate. However, the drawback of this trivial size assignment is the cost in silicon area ($\mathcal{A} = f(w)$) and power consumption, which are a monotonic increasing function of the sum of all driver widths, that is, Σw . Conversely, assigning a minimal value to the size w for each gate would optimize these last two figures of merit but

would also correspond to a maximum value of t^* . Therefore, the problem of device resizing can be seen as finding an assignment of gate sizes that yields a good area/speed tradeoff. For example, the problem can be cast as finding the gate size assignment corresponding to a minimal value of t^* with minimal area consumption.

We consider two heuristic approaches to the solution of the problem.

A first algorithm starts by setting all device sizes to their minimal size and then raises those that limit the circuit performance. The algorithm iterates the three steps: (1) compute the set of critical nodes; (2) determine a particular critical section $\mathcal{C}$; and (3) raise the size of the drivers in $\mathcal{C}$ by 1. The detection of the critical nodes is achieved by computing the data ready times and the slacks, as described previously. The algorithm then constructs a critical section $\mathcal{C}$. The critical section is required to include a critical section of $H(C_0, B)$ whose corresponding drivers have $w \leq \phi$. If no such set exists, the algorithm terminates. The critical section $\mathcal{C}$ is constructed according to a greedy strategy by looking at the values $\Delta^1 d(v)$ of the critical nodes. The rationale is to maximize the minimum value of $\Delta^1 d(v)$, $v \in \mathcal{C} \cap C_0$, which in turn is an upper bound on Δt^* . Then all the sizes corresponding to the gates in $\mathcal{C}$ are incremented. At this point, all the data ready times and slacks need to be recomputed; this is done at the next iteration of the algorithm.

The algorithm terminates when there is not a critical section of $H(C_0, B)$ whose corresponding drivers have $w \leq \phi$. Equivalently, the algorithm terminates when the maximum data ready time t^* is determined by a critical path with maximum-sized drivers. At this point, the maximum data ready time cannot be improved by resizing other devices. Note, however, that the algorithm does not guarantee the minimality of the silicon area—that is, $\mathcal{S} = f(w)$.

A second algorithm uses an opposite heuristic strategy. All device sizes are set initially to their maximal size, and then those device sizes that do not limit the circuit performance are decreased. These gates, which we can informally call "oversized," correspond to the nodes such that $s(v) \geq \Delta^{-1} d(v)$. The algorithm iterates the three steps: (1) compute the data ready times and slacks; (2) determine a subset $\mathcal{V} \subseteq V$; and (3) lower the size of the drivers in $\mathcal{V}$ by 1. The algorithm selects $\mathcal{V}$ as a maximal set of nodes that have the following properties: (1) $s(v) \geq \Delta^{-1} d(v) \forall v \in \mathcal{V}$; (2) no minimal-sized driver (i.e., with $w = 1$) is included; and (3) no mutual predecessor/successor relations exist among the elements of $\mathcal{V}$. If no such set exists, the algorithm terminates. The size of the drivers corresponding to the nodes in $\mathcal{V}$ are reduced by one.

The algorithm terminates when there are no more "oversized" gates—that is, when the gate size assignment corresponds to a local minimum of silicon area. Note that the value of t^* is not affected by the algorithm, because the data ready times at the successors of $\mathcal{V}$ cannot increase.

In the program implementing the sizing procedure, the first algorithm is used to determine a gate assignment corresponding to a minimal value of t^* . Then, this assignment is used as the starting point for the second

algorithm, which ensures a local minimum of silicon area. In the actual implementation, for practical reasons, there are two other criteria that can terminate both algorithms. The algorithms terminate if the number of outer iterations reaches a predefined quantity. The first algorithm terminates also if the area estimate $\mathcal{S} = f(w)$ reaches a predefined bound $\mathcal{S}_{\max}$. Note that by adding these two other criteria for termination, the optimality of the solution cannot be claimed. However, a "good" solution can be found with limited computing time and/or silicon area and power requirements. In addition, the quality of the solution can be traded off for computing time by choosing the parameter ε when running the first algorithm. Note that by choosing $\varepsilon = 0$, only the sizes of the gates that affect t^* are incremented. This corresponds to using a local strategy. Local optimization might be very inefficient, as far as speed of computation is concerned, when it aims at decreasing one output data ready time and when another output gate has a close data ready time. This other gate would probably generate the signal with the largest data ready time at the next iteration of the algorithm, and the algorithm would take many iterations to improve the circuit timing performance. A global strategy is achieved by choosing $\varepsilon > 0$. There is a trade off in the choice of ε : by increasing ε , fewer iterations are needed to achieve a given timing performance, but possibly more devices are resized.

Note that both algorithms terminate at a point when the maximum data ready time is constrained by the technological limitation on the size of the drivers on a critical path. To improve further the timing performance of the circuit with the given technological constraints on the maximum size of the drivers, it is necessary then to change the structure of the circuit by resynthesis or to change the position of the gates, as described in the next two sections.

Circuit Resynthesis

We consider in this section the problem of improving the circuit performance by changing the gate interconnection. We attempt to optimize the circuit by modifying the graph $G(V, A)$ by adding and/or deleting nodes and/or edges. Note that a change in the graph structure affects both the gate fan-out (and therefore the set of parameters $\{c_i\}$) and the gate fan-ins (and therefore the gate structure and the set of parameters $\{l_j\}$).

We consider different strategies for resynthesis. They can be classified as global or local. *Global strategies* apply a set of transformations to the entire circuit simultaneously. *Local strategies* assume a partition of the circuit into blocks and are applied to each circuit block in a sequence. Clearly, global strategies can achieve results that are at least as good as those obtained by applying local strategies. However, their computational cost increases as the size of the circuit, and it may be impractical to apply a global strategy to a large circuit. Local strategies can be made very effective by using an appropriate circuit partition into blocks. Resynthesis strategies may preserve the technological constraints on the gate structure or not. For example, in the case of CMOS domino circuits, gates are subject to a

constraint on the longest discharging path, that is, $l \leq l_{\max}$. A strategy that preserves the technological limitation would use transformations such that no gate at any step has $l > l_{\max}$. A strategy that does not preserve the technological constraints would allow more general transformation that would at some point permit gates with $l > l_{\max}$, for example. These gates would be eventually decomposed to satisfy the technological limitation. Note that the set of transformations is richer in the case of a strategy that does not preserve technological constraints. However, in this case, it is hard to determine a sequence of transformations corresponding to a descent in the maximum data ready time t^* . In fact, gate decompositions, which may be required at the end of the resynthesis procedure, may affect the maximum data ready time.

We consider two basic circuit transformations: gate elimination and gate factoring (extraction and decomposition), as introduced earlier. Note that by using a sequence of these two transformations, the structure of the interconnection of gates represented by $G(V, A)$ can be modified arbitrarily to any other structures implementing the same combinational function. Note also that by changing the circuit structure, the optimality of the gate size assignment no longer holds. Therefore, gates are assumed to have maximal size during circuit resynthesis—that is, the circuit speed is assumed to be limited by the structure of $G(V, A)$. A gate resizing is then applied after resynthesis.

As far as circuit timing is concerned, gate elimination and gate factoring can be represented as follows. Let gate j be a direct predecessor of gate i . The elimination of gate j into i is the replacement of gate i by gate i' . The inputs to gate i' are all the signals that are input to gate i and to gate j , except for the output of gate j . Gates i and i' are equivalent in the sense that their outputs coincide for any valid combination of the circuit primary inputs. Note that gate j need not be implemented if it has no successor and if its output is not a primary output of the circuit. In this case, gate j is deleted from the network—that is, node j is deleted from the graph $G(V, A)$. A gate elimination may violate a technological constraint, $l' > l_{\max}$. If technological constraints are required to be preserved, such an elimination cannot be considered.

Gate factoring is the transformation opposite to elimination. Gate i is replaced by gate i' . A new gate, j , is added to the network—that is, a new node, j , is added to graph $G(V, A)$ —and it is a direct predecessor of gate i' . The inputs to gate i' are some of the signals that are input to gate i and the output of gate j ; the inputs to gate j are some other inputs to i . Also in this case, gates i and i' are equivalent in the sense that their outputs coincide for any valid combination of the circuit primary inputs. Note that factoring cannot introduce a violation of the technological constraint $l \leq l_{\max}$.

Gate transformations change the number of stages, or logic levels, needed to implement one (or more) primary output of the circuit. In general, for a gate elimination (extraction), gate i' is more complex (simpler) than gate i . Therefore, its propagation delay may get larger (smaller). The data

ready times of all the gates that are successors of gate i are affected by the transformation. Moreover, the fan-out of the direct predecessor of gates i' and j may change, and, as a result, so may the data ready times at some other circuit outputs. Let $t'(v)$ denote the data ready time after the transformation, and let $\Delta t(v) \equiv t'(v) - t(v)$. A transformation of gate i is said to be "locally favorable" if in the modified circuit: (1) $\Delta t(v_i) + t(v_j) < 0$, that is, the data ready time decreases; and (2) $\Delta t(v_j) < 0 \forall j \neq i$, that is, any increase in the data ready times is less than the slack. Since a gate transformation affects the entire circuit, it is important to measure its global impact on the timing of the circuit. Let $\Delta \underline{t} \equiv [\Delta t(v_1), \Delta t(v_2), \dots, \Delta t(v_n)]$. Then a figure of merit of the global impact of the transformation is measured by $\eta^T \cdot \Delta \underline{t}$, where η is a vector of coefficients. Gate transformations affect the silicon area (estimated by $\mathcal{S}$) of the circuit implementation, because of the change in the number of gates and in their internal structure. Therefore, a comprehensive figure of merit is $\Delta e \equiv \eta^T \cdot \Delta + U \theta \Delta \mathcal{S}$, where θ is a scalar coefficient.

We consider first an example of an algorithm that uses a global strategy. This strategy guarantees a descent of the maximum data ready time at each iteration. If, in addition, we require the strategy to preserve the technological constraints, the transformed circuit can be implemented without additional gate decompositions; therefore, its performance is guaranteed to be better than the original one.

Circuit resynthesis with a global strategy that preserves the technological limitations can be done by applying a sequence of gate eliminations and/or factoring to a circuit. An algorithm for circuit resynthesis can be implemented by an iteration among the steps: (1) compute the set of critical nodes; (2) determine a particular critical section $\mathcal{C}$; and (3) apply the "best" transformation to the gates represented by $\mathcal{C}$. The detection of the critical nodes is achieved by computing the data ready times and the slacks, as described before. The algorithm then constructs a critical section $\mathcal{C}$ as in the case of the first resizing algorithm. The critical section is required to include a critical section of $H(C_0, B)$ the corresponding gates of which have a locally favorable transformation (according to the technological limitations). If no section exists, the algorithm terminates. The critical section $\mathcal{C}$ and the type of transformations are constructed according to a greedy strategy by examining the values $\Delta t(v)$ of the critical nodes. Then all the gates corresponding to the gates in $\mathcal{C}$ are transformed. At this point, all the data ready times and slacks need to be recomputed; this is done at the next iteration of the algorithm.

As mentioned previously, this algorithm guarantees a descent of the maximum data ready time at each iteration. However, it is inefficient to apply it to large circuits because for each node there are several possible transformations to be evaluated. Because of the computational complexity, only few transformations can be considered in practice. On the contrary, it would be useful to use other circuit transformations that may be beneficial, such as applying simplification after elimination or factoring the gates

with the goal of determining common subexpressions. For this reason, it is interesting to develop algorithms with local strategies. In this case, the resynthesis of the circuit is done on circuit blocks, and a richer set of transformations may be applied. Unfortunately, while resynthesizing a circuit block, it is not possible to compute the effect of a transformation on the circuit data ready times and on the slacks, if we are monitoring only the structure of a block. Therefore, some heuristic rules must be used to determine which transformation to apply and where.

The strategy for circuit resynthesis, which has been implemented, is a local one. It exploits the hierarchical structure of the circuit. The circuit is partitioned into blocks, and this partition corresponds to the one used by the YSC system for synthesis purposes. The blocks of the partition are the combinational modules specified in the hierarchical circuit description. Transformations are applied only inside the combinational modules. Each combinational module is described by a subgraph $G^m(V^m, A^m)$ of $G(V, A)$. A *critical module* is a module whose subgraph has critical nodes.

At first, the critical graph for the entire circuit is computed. Then, the critical modules are redesigned one at a time. The data ready times and slacks are computed for the whole circuit after the resynthesis of each block. Circuit blocks are sorted by first choosing those subcircuits with nodes that have no predecessor in any other critical module that has not yet been resynthesized. The reason is not to change the data ready times at the input gates of a module after its resynthesis, since the circuit structure has been tuned to this particular input signal arrival-time distribution. For the resynthesis of each subcircuit, each subgraph corresponding to a critical module is considered along with its boundary conditions, which are the data ready times at the module inputs and the slacks at its outputs. While doing resynthesis of a module, the goal is to reduce the data ready times at the critical nodes. Note that since the module is "extracted" from the whole circuit, the slacks cannot be recomputed after each transformation. Therefore, the critical nodes are computed only once, and the transformations are applied to reduce the data ready times at these nodes.

The overall goal of the algorithm is an area effective and time effective synthesis. An elimination leads to a more compact implementation, if the node being eliminated is not implemented. Therefore, if the eliminated node had more than one direct successor, it would be convenient to have it eliminated into its direct successors, even if these eliminations would not improve the timing performances, since they would improve the silicon area. The factoring of a gate requires the implementation of an additional gate; the cost of the additional area can be (partially) offset by the internal simplification of the successor gate. For this reason, it is convenient to consider the extraction of common subexpressions of the logic functions implemented by two (or more) gates. In this case, any node extraction would lead to the simplification of the two (or more) direct successors of the generated node and eventually to a reduced loss (or possibly a saving) of silicon area.

The algorithm for the resynthesis of a subcircuit can be summarized as follows: (1) label the critical nodes; (2) eliminate labeled nodes; (3) simplify the gates where an elimination took place; (4) extract common subexpressions; and (5) perform areadelay tradeoff. The critical nodes are detected at the beginning. All subsequent operations attempt to decrease the data ready time at these nodes. Direct predecessors of labeled nodes are eliminated into them only if the transformation is locally favorable and the estimated area $\mathcal{A}$ is less than a bound $\mathcal{A}_{\max}$. Simplification is applied to those gates where an elimination took place. Then, for each subexpression common to at least one labeled node, a gate is added to the circuit to implement the subexpression if the extraction is locally favorable and $\mathcal{A} \leq \mathcal{A}_{\max}$. Finally, areadelay tradeoffs are exploited by moving subexpressions across gate boundaries to reduce gate and device count.

The algorithm may be constrained to preserve the technological constraints or not. In the latter case, some gate decomposition may be required as a last step. Even though the algorithm does not guarantee a descent of the data ready time of t^* , experimental results have shown that performing resynthesis with local strategy (with or without preserving the technological constraints) yields a considerable improvement in the circuit timing performance.

Circuit Repositioning

The repositioning algorithm aims at improving the timing performances by reducing the capacitive load $\{c\}$ of the critical gates. This is done by shortening the length of the nets carrying critical signals. In the YSC system, each module is designed as a rectangular macrocell consisting of an interconnection of smaller cells each implementing a logic gate. Gates are interconnected by wires that run inside and outside the macrocell. We assume that the capacitive loading on a gate depends primarily on the length of the wires that go across the module boundary. Therefore, we restrict our attention to the intermodule wiring, and we estimate the wire lengths by the macrocell positions.

The goal of the algorithm is to reduce the length of the critical nets by changing the mutual positions of the modules. The rationale of the repositioning algorithm is to bring the critical nodes (gates) closer to their direct successors in other (not necessarily combinational) modules. Since any change in a module position affects the length of all the nets connected to it, we require that any change in the module position correspond to an improvement in timing performance. For this reason, we introduce the *geometrical slack* $g(v_i)$ for each net connected to the output of the gate denoted by v_i . The geometrical slack $g(v_i)$ is derived from the slack $s(v_i)$ and represents the additional length that each net can tolerate while satisfying the relation $t(v_i) \geq t(v_j)$ at each output node v_j .

The repositioning algorithm is based on pairwise module interchange. The moves that are allowed are position interchanges between modules with compatible shape, such that the increase in wire length for each net connected to each gate v_i of the modules under consideration is less than

$g(v)$). With this choice, each move cannot increase f^* . The objective function is a weighted sum of the net lengths connected to the critical nodes. The algorithm uses a greedy strategy in determining a sequence of allowed moves corresponding to a maximal decrease of the objective function.

The algorithm computes first the critical graph $H(G_e, B)$, the set of critical modules M , the slacks $s(v)$, and the geometrical slacks $g(v)$. Then it iterates the following steps: (1) select a critical module; (2) determine the set of allowed position interchanges for the selected module; and (3) select and perform a position interchange. The algorithm first determines a candidate module for interchange by using a weighted sum of the parameters of the critical nets connecting it to other modules of compatible shape. Then the set of allowable moves is determined by considering the geometrical slacks. If no move is possible, the candidate is rejected. Otherwise, the local best interchange is performed and another interchange for that module is searched for. When all the critical modules have been examined as candidates, another pass is done if at least one module interchange has been performed at the present pass. Otherwise, the algorithm terminates. If the original placement was obtained by minimizing the total wire length, this number is monitored during the interchange. The algorithm trades off the total wire length for the wire length of the critical nets. An additional termination criterion that may be used is reaching a bound on the total wire length.

7.3 Using the YSC System

7.3.1 The YSC System Implementation

The YSC system implements the ideas and the synthesis algorithms presented in section 7.2. The system consists of a coherent set of programs that perform the synthesis of a digital system from its description in the hardware language to the specifications of the geometries of the masks needed to fabricate the chip. We describe here some implementation details and how the Yorktown silicon compiler is used to synthesize a VLSI chip.

Since the YSC system is experimental, we have chosen to implement most of its tasks in the APL language. APL is an interpreted language, and the APL notation allows for complex operations specified by simple statements. Thus APL programs are generally compact, easy to write, and easy to debug. The various synthesis tasks of the YSC system are each organized around a set of functions that operate in an environment called an *APL workspace*. The APL workspaces provide facilities for interaction and easy debugging of code. As a result, APL permits fast code development and easy experimentation with new ideas. On the other hand, a few synthesis tasks are based on well-established, computationally expensive algorithms (e.g., floor planning using simulated annealing). These tasks are still implemented by APL functions in APL workspaces, but the computationally expensive

parts are executed through function calls to compiled programs, written in Pascal, for faster execution.

Each major task of the YSC system is implemented as an APL workspace. The workspaces communicate with each other by writing intermediate files to the operating system file mechanism (in this case, CMS operating system). These files may contain binary or textual information and are linked by name through a hierarchical set of references. This hierarchy corresponds to the initial hierarchy of the hardware description or its refinement during the synthesis process. The file names are those given by the user or those created by the structural synthesis process. Different file types are used to specify the type of information contained in the file. For each intermediate stage of the synthesis process and for each module, there is a set of files containing the corresponding intermediate data. The set of files at a given stage of the synthesis represents a view of the system being designed, which corresponds to a particular level of abstraction. For example, a hardware system can be described by a set of files containing either a YIF or a YIL description. This data will be transformed into another set of files describing the logic equations of each module, then into files describing the topology of the transistor implementation, and eventually into a set of files containing the layout geometries.

The compilation of a digital system consists of a sequence of calls to the appropriate APL workspaces. This sequence will be described in section 7.3.4. The result is the automatic transformation of the chip description from the high level into a set of geometries representing the mask data.

7.3.2 Describing Hardware to the YSC System

The hardware description principles were described in section 7.2.1. We consider here the task of the hardware designer in describing an architecture to the YSC system. The Yorktown silicon compiler supports several approaches to hardware descriptions, ranging from pure behavioral to pure structural and including combinations of both.

A behavioral description can be given by a program in the V language or, equivalently, in a YIF file. In this case, the circuit structure is determined during the structural synthesis phase of the YSC system. On the other hand, purely structural descriptions can be given by a hierarchical set of HND and YIL files. In this case, the HND files specify directly a system partition into combinational logic modules, registers, drivers, receivers, and so forth. Structural synthesis is thus bypassed, being performed by the human designer directly. Of course, a behavioral description frees the designer from having to describe the details of the circuit implementation. However, it is sometimes desirable to specify the structure of the circuit more directly. For example, computer architects sometimes have in mind an implementation when they specify an architecture at the behavioral level. Of course, it would be desirable that the structural synthesis stage discover an equivalent or better implementation. In general, this is not

yet possible. For example, structural synthesis does not support the detection and efficient use of concurrency required for an efficient pipelined implementation. Therefore, to be able to synthesize chips with specific structures, it is necessary to specify the structure directly. This is a limitation of the present knowledge and algorithmic development of the structural synthesis stage, which can be expected to improve in the near future.

In the YSC system, it is also possible to use a high-level hardware description that is neither purely behavioral nor purely structural. Structural implications in the hardware specification may be achieved by arranging the description as a set of module calls (in either V or YIF). Each module would correspond to either a combinational block or a finite automaton. The distinction between a module call and a call to an operator in the YLL library is that the operator call is similar to a macroexpansion mechanism, while a module call is similar to a procedure call. Thus a module call produces a separate piece of hardware, and an operator call is expanded and merged with the logic of the present module. Our mechanism for describing concurrent automata is to describe each as separate modules, using a user-written HND module to connect them through their I/O names. Note that such a partition is forced by the design specification and is preserved through synthesis. It is also possible to specify systems with mixed behavioral and structural representations by using two (or more) descriptions, one structural and one behavioral, which communicate through their I/O ports and are connected with an appropriate HND module description. One description may be purely in V or YIF, and the other purely HND or YLL.

7.3.3 Application-Specific Cell Descriptions

Everything that can be described in the input languages of the YSC system can be synthesized by the programs. This does not always lead to acceptable implementations. The libraries are available for cells that are quite common and in which special features must be realized. The user can extend the library with cells that he or she wants to be treated in a special way. This can be done at several levels.

If, for example, the user wants a special logic implementation of an arithmetic unit, a YLL description can be added to the library. In this description the user must make sure that the intermediates wanted for this implementation are referenced as global variables. The logic synthesis system will view these as outputs of the logic module and, therefore, will preserve them in the gate implementation of that module. During the linking their intermediate character will become apparent again, and proper connection to the environment is guaranteed.

If the requirements are more specific, even down to the layout level, more work must be done. If the layout is completely fixed, there is the possibility of adding an inset cell and its periphery description to the system. This will suffice for the incorporation of the cell in the final layout. If this cell is to take part in simulations that are carried out for verification purposes, a consistent functional description is needed as well.

If there is still some flexibility needed in the cell, a special cell assembler must be written. In the YSC system experience described in section 7.4 such a cell occurs; it is called the general-purpose register. We have supplied a cell assembler for that module. Part of that module, the address decoder, is described in YLL but is passed through the logic editor, without doing any optimization effort, in order to obtain the appropriate data structures describing that gate implementation. The layout program treats each decoder as a normal logic macro with the constraint that all circuits must be placed in one column. The other part of the module contains the latches, the bit line prechargers, the sense amplifiers, the write drivers, and the shut-off feedback circuitry. The layout of all these circuits, except the feedback circuitry, is completely implied by the procedure as soon as the basic cell dimensions and the number of read and write ports are given. The feedback chain, however, can be completely tuned—that is, every transistor in that chain can be sized individually on the basis of simulation, experience, or other criteria. The two parts, the address decoding part and the latch array with its supporting circuitry, are connected together by a generic river routing routine followed by the right mask generator.

In the high-level input language, all these application-specific cells must be treated in the same way as the more permanent library cells.

7.3.4 The Synthesis Stream

Silicon compilation must be capable of being a completely automatic process. The YSC system provides both an automatic and a hand-driven synthesis mode of operation. The latter is useful for stepping the compiler through its various stages and for verifying whether the circuit synthesis is correct and optimal. Errors may be introduced in the hardware description that may be detected after the first stage of the compilation. The compiler developers may wish to study new algorithms for synthesis and optimization and verify their correctness. Also, decisions about which procedures to apply may be made by the user as certain intermediate results become available.

Each major task of the YSC system is implemented by a set of functions in an APL workspace and can be executed by a single command. This command loads the appropriate APL workspace and executes a main execution program. The exact mechanism used in the CMS operating system is the Rexx interpreter executive command language. The Rexx program initiates the APL processor, loads the appropriate APL workspace, and starts the main program. The workspace also contains all the parameter and option settings for executing the corresponding silicon compilation task. The main program is stored along with the workspace and need not be known or understood by the user of the YSC system. However, it is of extreme importance to its developers, since the option settings affect the quality of the results and should be set properly. For example, in the case of logic synthesis, the execution program specifies the sequence of operations (e.g., extract, eliminate, simplify, etc.) used to optimize and synthesize

the multilevel logic. Certain default settings for the synthesis procedure can be made to depend on the name of the macro being synthesized. In this way, an ALU can be treated differently from a piece of control logic or a multiplexer for example.

The entire silicon compilation process can also be executed by a single command, which invokes a main Rexx program that is a sequence of commands each related to a synthesis step. As an example, a main execution program could be as follows:

- Execute structural synthesis
- Execute logic synthesis
- Estimate delays
- Execute floor planning
- Estimate delays
- Execute timing optimization
- Execute macrogeneration
- Execute layout assembly

A "standard" main execution program can be invoked by a user of the YSC system who does not need to know its content. However, other main execution programs may be executed either in the development and experimental phase or to compile chips with special features or requirements. For example, if the design is a microprocessor and the designer wants to keep all the data flow portion in a stack organized with a particular structure, an appropriate command must be introduced (execute data flow stack construction) in the main execution program.

This programmability feature of the compilation process allows the compiler to be extended easily to incorporate new ideas. It is conceivable that different users of the YSC system will have different interests and capabilities. The novice may need to know only how to describe the hardware and how to invoke the main execution program. A more sophisticated user may want to understand the mechanism of the compiler and explore the use of the different execution sequences. An expert in hardware and software design may want to add a special feature to the compiler for his or her own needs (e.g., an on-chip self-test pattern generation module) by writing a program for a specific task and invoking it from the main execution program.

7.4 Experience with Using the YSC System

As an exercise for the compiler and as an opportunity to learn more about system-level issues, we have developed a design for the IBM 801 processing unit [Radi83] and described it in the languages used as input to the YSC system. We chose the 801 architecture for two reasons. First, since the 801 was developed at IBM Watson Research Center, we had access to a detailed description of its operation as well as to the developers themselves. Second, the 801 is sufficiently complex to provide a good test for the compiler and,

at the same time, sufficiently simple to be understood by novice system designers.

This section describes the process of spanning the gap between the architectural specifications and the input to the silicon compiler. We present two descriptions of the 801 processor as inputs to the SC system. The first is a behavioral description in the V language and implies only a few features of the circuit structure. The second is a description in the HND/YLL notation, which allows the user to specify the structure in detail. We also report the results obtained by the Yorktown silicon compiler in synthesizing the chip.

7.4.1 The 801 Architecture

The architecture of a system defines its attributes as seen by the programmer—that is, the conceptual structure and sequential behavior of the machine as distinct from the organization, the logical design, the layout, and the performance of any particular implementation. Typical parts of an architecture specification are the instruction set, the addressing capabilities, the registers visible to the assembly language programmer, and the interrupt handling. In spite of the clear separation provided by this definition, it certainly would be unwise to isolate architecture design from its target operating system and compiler on one side and from its hardware implementation on the other. Therefore, it was not surprising that during this exercise we were confronted repeatedly with the fact that the 801 architects had the application and the implementation constantly in mind. Their decisions were mainly guided by three basic design principles:

1. A compiler should be able to produce object code with an efficiency comparable to the best hand code so that assembly language programming is never needed for performance.
2. Each instruction should be able to be executed in one short machine cycle, and yet the path lengths of the computations should not be commensurately larger than those required by machines with more complex instructions.
3. The idle time of the processing unit due to storage access should be kept short.

What follows is meant only as a brief introduction to the architecture of the 801 processor. For more details the interested reader is referred to [Radi83].

The 801 is a 32-bit minicomputer architecture. The arithmetic and logic operations deal with 32-bit words. Shifts and rotates can have lengths of up to 32 bits. Addresses are 32 bits long. The 801 architects foresaw a compiler capable of a very effective utilization of a large number of registers, and therefore decided to prescribe a register file of 32 full-word general-purpose registers (GPR).

Instruction length and format greatly influence the complexity of the hardware for decoding and the efficiency of a pipeline if implemented [Mar84]. Therefore, accepting some alignment constraints can be very beneficial with respect to hardware simplification and pipeline

implementation. This was recognized in the architecture design of the 801. Each instruction is exactly one word long and is aligned on word boundaries. Only a few instruction formats are used. All operands are aligned according to their size. These constraints, the emphasis on register utilization, and the single-cycle instruction principle, move the 801 into the class of streamlined architectures [Henn84]. Although many of its properties are shared with the so-called reduced instruction set computers (risc), the size of the instruction set alone would make such a qualification a misnomer in the case of the 801.

Typical instructions are provided for storage access, address computation, branching, and comparing. Consistent with the third basic design principle, the processor is allowed to continue with instruction execution after the initiation of a storage access. The processor will only stall if the memory system has not yet provided the requested data when the content of the affected register is needed. To make better use of this facility, the optimizing compiler can reschedule the storage access instructions to reduce the number of stalls. Also, a branch with execute instruction (similar to the delayed branch in reduced instruction sets) allows the compiler to move inevitable instructions into the normally idle period following a taken branch instruction.

Arithmetic operations are done using 32-bit two's complement, with the usual add and subtract instructions as well as some special instructions for computing the minimum and the maximum of two values. There are also several instructions not normally found in a reduced instruction set, such as "the multiply step" and "the divide step," which allow complex operations to be easily decomposed into sequences of simple instructions. Also, a rich set of rotate and shift operations, controlled by a register or an immediate field are provided.

The third basic principle immediately led to a store-in-cache strategy. But instead of a single conventional cache that delivers a word every cycle, an instruction cache and a data cache were both included separately, thus effectively doubling the cache bandwidth and allowing asynchronous fetching of instructions and data. Explicit instructions for cache management are introduced to make a reduction in unnecessary loads and stores of cache lines possible at the software level.

Finally, the number of interrupts defined for the processing unit is reduced by imposing software protocols for interrupt handling and prescribing an external interrupt controller. Protection is achieved by making some instructions privileged in the sense that they can be executed only in a special state that is not accessible to application programs.

7.4.2 System Description

An Initial Behavioral System Description

The *Principles of Operation*, a document describing in precise and verbose detail the 801 architecture, was our starting point. In order to condense this

bulk of information (as well as to provide us with an understanding of the operation of the machine), we first translated these principles of operation into a high-level behavioral description in V, specifying the effect of each instruction on the architected registers, the details of when interrupts may occur and how they are processed, and so forth. This description is also very useful as the system documentation; for example, from within an editor, we could easily locate all of the instructions affecting a particular special-purpose register.

The main routine is nothing more than an infinite loop. For each cycle this loop fetches (P8RI) an instruction, executes (P8EXE) it if no instruction storage interrupt has occurred, and tests the various interrupt conditions (P8CHI) as shown in Figure 7.18.

Architected registers were modeled by variables in the program. The general-purpose registers (GPR) were modeled by an array with 32 elements. Other variables not mentioned in the principles of operations were necessary to hold intermediate results or to "remember" certain special situations.

The V module P8RI roughly corresponds to the example presented in sections 7.2.1 and 7.2.2. In the execution module P8EXE, each instruction was coded as a CASE. The decoding of the instruction was done by first separating the two main instruction formats with an IF statement and then decoding the appropriate bits with a CASE statement (see Figure 7.19).

Interrupt checking (P8CHI) is done by examining various interrupt flags and external pins in the order established by interrupt priorities. The flags are set during instruction fetch and execution according to the allowed interrupt conditions. In the case of an interrupt, P8CHI will load a new machine state and save the old one in some special-purpose registers.

Besides the interrupts, the most difficult aspects to model using only an imperative language were:

- *The GPR management.* All data memory access goes from and to GPRs. To minimize the idle time due to data memory requests, these operations

```

MODULE P801                                /* This is the main 801 module */
...
BODY P801
DO INFINITE LOOP                          /* Fetch-Execute-Check-interrupt loop */
P8RI;                                     /* Fetch */
IF ~IN_STOR_FLAG THEN P8EXE; ENDIF;      /* Execute */
P8CHI;                                    /* Check interrupts */
ENDDO;
END P801;

```

Figure 7.18 Main V program for the 801 processor.

```

MODULE P8EXE;
...
BODY P8EXE;

IF ~( (IR::0,6)=63)
  THEN
    WHEN (IR::0,6)
      CASE 16;
        ...
        CASE 18;
          ...
        ENDCASE;
      ELSE
        IF ~( (IR::21,4)=0)
          THEN PGM_FAULT:=0; /* not defined */
          ELSE
            WHEN (IR::25,7)
              CASE 16;
                ...
                CASE 18;
                  ...
                ENDCASE;
              ENDIF;
            ENDIF;
          END P8EXE;

```

Figure 7.19 Program structure for the execution module.

can be initiated during an instruction and finished when the requested data arrive. Hence, for memory pending read operations, it is necessary to implement a GPR management that controls which registers contain valid data and which still must be actualized.

- *The instruction storage access.* Instructions are prefetched, and so it is necessary to keep track of whether a prefetched instruction can be used or must be discarded because of a branch or an interrupt. Also, an instruction fetch might result into a processor stall if the instruction cannot be (pre)etched fast enough.
- *The data memory access.* Data memory access is done in parallel to instruction execution. Each access might imply a processor stall if a previously initiated data access is still occupying the data bus.

The technique used to model these three aspects consisted of writing one module for the management of the GPR, one for the instruction storage bus, and one for the data storage bus. To use these resources the respective module call was issued.

This description could be used as the starting point for structural synthesis. However, this raises the following problems:

- the design is not pipelined. We have no means of automatically constructing a pipeline out of a nonpipelined description and including all

the pipeline control (e.g., flushing the pipe in the case of an interruption or a branch, feeding back results to previous stages of the pipe without storing them first in registers to avoid processor stall, etc.) To our knowledge, no such methods have been exercised in practice with realistic designs.

- The GPR originally was modeled as a single port memory. In practice, this is not a viable solution because it results in very bad performance. Multiport memories are necessary. Although we can handle multiport memories, the problem of assigning ports to accesses of a single array representing the memory is not done automatically in our system yet. To use multiport memories, each port must be modeled as a separate variable, thus deciding in each access (by specifying the particular variable) which port will be used.
- The execution module coded as presented results in a huge number of control states. The reason for this is that during each instruction that uses registers or the data storage bus (i.e., almost all instructions), a processor stall is possible. Since waiting states are created automatically in this case, a different waiting state for each instruction (basically “remembering” the particular instruction) results. Since many instructions access more than one register, the total number of these control states is huge. This is not a problem in principle—that is, the state transitions are quite simple and result in a small amount of logic, the memory required to hold these states is small, and everything is generated automatically. Only the running time of structural synthesis will increase.

As a consequence, the performance of a processor synthesized in this way would not be acceptable. As an exercise and a test for the Yorktown silicon compiler, we synthesized the structure of the 801 in this version. The results are given later in this section. The real design required some modifications, such as the introduction of a pipeline and the separate representation of each port of the GPR. As will be seen, this also solved the problem of the excessive number of control states.

System Decomposition

To introduce a pipeline, we decomposed the system into pipeline stages, specifying each stage as a separate design. An additional module implemented the pipeline control. The pipeline design was done manually. Since some of the 801 instructions seen naturally tailored to a pipelined implementation, the first behavioral description could be easily decomposed into what seemed to be a natural four-stage pipeline. This choice was different from the design discussed in [Radi83].

The stages, as illustrated in Figure 7.20, are:

1. *Fetch (F-stage)*—Load into the instruction register the instruction pointed to by the program counter, a branch address, or an interrupt address. Load the program counter with an appropriate new value.
2. *Decode (D-stage)*—Load the appropriate registers (RA,RB,RS) with the contents of the general-purpose registers pointed to by the instruction.

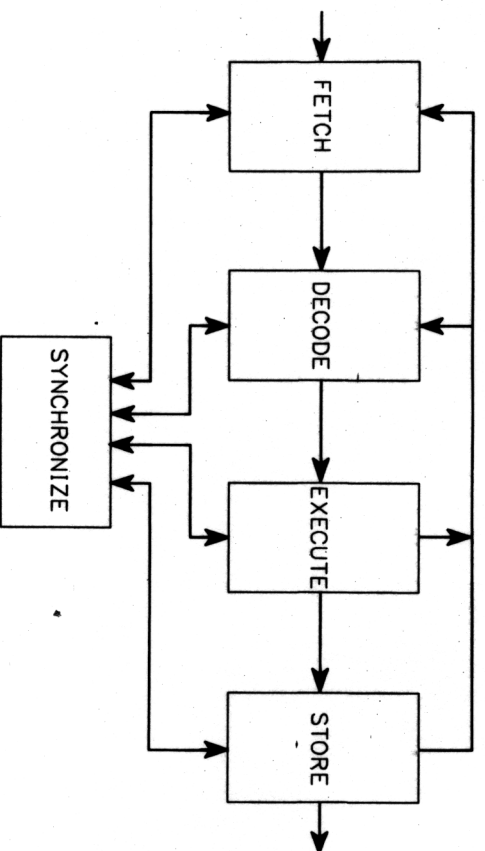


Figure 7.20 Pipeline structure for the 801 example.

3. *Execute (E-stage)*—Generate control signals from the operation code, make the arguments available to the execution unit, and latch the results. Generate a branching address if necessary.
4. *Store (S-stage)*—Place the result in a register. Initiate requests to main memory.

The behavioral description of the pipeline consists of five V programs, one for each stage plus one for pipeline control. Each of these programs is synthesized independently, obtaining five finite automata that run concurrently. Each of them has only a small number of states, because the part chosen to be performed by one stage naturally corresponds to parts that had fit in one state in the initial description! Instructions are decoded in each of the last three stages, so it is not necessary to encode the instruction in the control states. Additionally, each stage decodes the instruction differently, allowing optimization to a large extent (e.g., the logic needed by the decode stage uses only a few transistors).

The pipeline works as follows. In the absence of branches and interrupts and if data are delivered fast enough from the data and the instruction cache, each stage of the pipeline performs its operation in one cycle. In the case of a branch instruction, if the branch is taken, the program counter is loaded with the target address at the end of the E-stage. The partially executed instructions in the other stages of the pipeline are now incorrect; in an unpipelined design, they never would have been fetched. To handle this situation, each stage of the pipeline has a "kill flag," which is propagated from one stage to the next as the instruction flows down the pipe. A branch taken sets the kill flag for all the other instructions currently in the pipe.

When such an instruction reaches the S-stage, its effect on architected storage is inhibited because the kill flag is set. In this way, a branch instruction effectively flushes the contents of the pipe.

Interrupts are very similar to branches. In fact, the action of the processing unit under an interrupt is to store the current status in special-purpose registers and then to branch to a fixed memory location. One additional consideration is required because interrupts can be generated at any stage of the pipe. To handle this, each stage has interrupt bits that flow along with the instruction in the same way as the kill flag described previously. As interrupts are generated, these bits are set; when the instruction reaches the S-stage, the interrupt is serviced.

An instruction can generate an interrupt in the D-stage, for instance, and then be killed by a branch instruction ahead of it. In this case, the instruction in the D-stage never should have been executed and the interrupt it generated is ignored.

Another consideration is "waiting for data." For example, when loading a register from external memory, the load instruction generates a memory request that is sent off-chip. Execution continues normally until a reference is made to the register that was supposed to be loaded. If the memory request has not yet been serviced, we must wait for it before proceeding. The waiting function is accomplished by inhibiting the transfer of data from one stage of the pipeline to the next. In this way, we effectively halt the pipeline until the external request is serviced.

The flow through all four stages of the pipeline is altered somewhat by certain bypasses in the pipeline. These speed the flow of the instructions through the pipe. For instance, if an instruction in the D-stage requires the contents of a register that is the target of a load instruction or the target of the result just computed in the E-stage, the data will be routed into the appropriate latch on the boundary between the D- and E-stages. On the next cycle these data are stored in the correct register but are also already in the E-stage. This allows, for example, a sequence of additions to be executed without interruption. Another complication is that there are instructions that refer to writing data directly into some of the architected latches. This could be done at the end of the S-stage, but this might cause some delay in the instruction flow. If we know that no interrupt will be processed on the next cycle, then this store can be performed at the end of the E-stage. In general, these decisions corrupt the clean design of the pipeline, complicate the control logic, and may affect the cycle time of the machine.

A Structural System Description

Although the behavioral description is sufficient input for the YSC system, we also described the complete structure of the 801 in HND and the combinational logic in YTL. In addition to the historic reasons for this further description [Bray85], it was very useful to validate the results of structural synthesis by comparing them with this "hand design."

The major steps in defining the structure were:

- *Identification of all registers.* In addition to the specified architected registers, many others were necessary.
- *Definition of all the necessary control signals.* The control within each pipeline stage was treated locally. Local control is convenient because it reduces the wiring. Moreover, it can be efficiently designed by exploiting the unused or inappropriate operation codes as don't-care conditions for the local control blocks.
- *Partitioning of the combinational logic.* While writing the behavioral description of the 801, the frequent operations were identified and coded as a separate module. This already represents a first structure, which guided the partition of the logic. Sometimes further partitioning was advantageous, such as when the execution unit was divided into an arithmetic-logic unit and a rotating and merging logic.

After the structure was obtained, it was directly coded into HND. Writing YLL programs for each logic macro in the 801 turned out to be a fairly straightforward and modular programming task. For the logic involved in the control part, YLL's ability to handle tables proved to be very convenient. Our structural design led to the diagram in Figure 7.21. The major difference between our behavioral description and our structural description is that the latter does not include an explicit module to

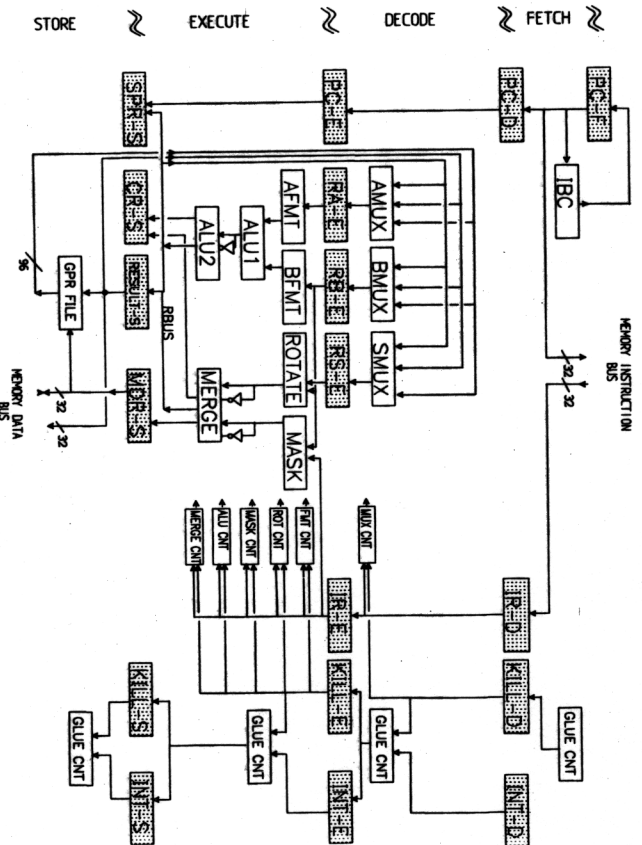


Figure 7.21 Schematic of our structure of the 801 processor.

synchronize the pipeline. This module only controls the synchronization aspects of the pipeline stages, and consequently it is small.

7.4.3 Design Statistics

In this section, we summarize very briefly some results obtained by using the YSC with our 801 processor design.

Input Code

Our two behavioral descriptions and the structural description resulted in the input code given in Table 7.1. The operator library included only 16 operators coded in YLL, enough to implement any of the YIF operators. The cell library contained only a register file and bit images for single latches, drivers, and receivers. The data in Table 7.1 contain the entire chip description.

Structural Synthesis

The synthesis of the nonpipelined version led to the results shown in Table 7.2. "Operations" refers to the number of YIF nodes before control and network generation, "states" refers to the number of generated control states, and "latches" refers to the number of bits requiring a latch. The module EXE contains the complete data path, but the control was generated for only 40 percent of the instructions.

The results obtained for the pipelined version are given in Table 7.3. They do not include some purely combinational modules that are also necessary.

The pipeline stages automatically synthesized led to the combinational logic shown in Table 7.4. The transistor count given includes only the transistors required for the gate logic (pull down tree). Those were the best results achieved concerning transistor count. Trading 15 percent transistor count for 25 percent of speed improvement was possible, for example, for FETCH. The delay is estimated for a 1 μ CMOS domino (single voltage cascade switch or SCVS) implementation. A straightforward implementation of the logic specification generated by structural synthesis would have

Table 7.1 Input Code Statistics

Version	Modules	Code (lines)	Code (statements)	Tables (lines)
Behavioral, unpipelined	13	V: 1,487 YIF:10,212	V: 1,370	
Behavioral, pipelined	10	V: 1,820 YIF:12,481	V: 1,190	
Structural, pipelined	YLL: 64 HND: 132	YLL: 1,877 HND: 132	YLL: 771	YLL: 421

Table 7.2 Results of Structural Synthesis for the Nonpipelined Version

Module	Operations	States	Latches
CHD	34	4	5
MAIN	29	5	289
CC	70	0	0
EXE	706	113	114
CHI	55	10	4
RD	54	5	3
EFFA	72	9	91
GPR	20	2	1
INTE	82	2	1
SELBIT	66	0	0
RI	34	4	2
MASK	36	0	0
DATA	44	1	67

contained up to nearly four times more transistors! The pipeline stages DECODE and EXECUTE were partitioned automatically because their combinational logic resulted too large.

Our manual decomposition led to a total of 118 different modules. The combinational logic modules have been synthesized to the gate level. Also a global net list was created for the chip. These data have been used for gate-level simulation and timing optimization at the chip level.

Some statistics of the synthesis results have been entered in Table 7.5. The totals in this table do not include the internal powering of signals (e.g., by clock repeaters), but they do include all the off-chip drivers and receivers. In compiling the statistics, all output buffers are assumed to be of minimum size. (The driver-sizing factor is implemented by inserting parallel transistors of minimum width.) As a result, the transistor totals for the domino circuits include only the transistors required for the gate logic and five transistors for the buffer and clock. The decomposition of the logic was done with the possibility of distributing the control to or near to the modules being controlled. Thus, of the 58 combinational logic modules, 25 generate control signals.

Table 7.3 Results of Structural Synthesis for the Pipelined Version

Module	Operations	States	Latches
FETCH	16	4	98
DECODE	329	2	181
EXECUTE	806	1	180
STORE	50	4	35
SYNCHRONIZE	312	1	99

Table 7.4 Logic Generated for the Pipelined V Description

Module	Inputs	Outputs	Levels	Trees	Transistors	Delay(ns)
FETCH	68	69	11	85	417	23
DECODE	219	131	—	320	2,098	—
EXECUTE	255	152	—	1,607	10,414	—
STORE	73	44	4	46	191	8
SYNCHRONIZE	202	167	13	212	1,107	28

Timing Optimization

The programs for timing optimization have been tested on the microprocessor synthesized from the structural description (Table 7.5). The maximum data ready time in the circuit, t^* , was 72.7 ns before optimization. A pass of a resynthesis and a resizing step could reduce the maximum data ready time to 53.2 ns, or 26.9 percent. The critical path of the processor was related to the generation of condition codes, as a result of a trap condition based on the outputs of the processor ALU. Therefore, the path affected 9 critical modules, or 15 percent of the total.

The repositioning algorithm was applied only to the part of the chip consisting of the glue logic circuits implementing the interrupt and global processor control. One pass of repositioning and resizing reduced the maxi-

Table 7.5 Design Statistics for the Pipelined Structural Description

	Total blocks	Total gates	Min/Max Gates in block	Min/Max Transistors in block	Total transistors
Combinational logic blocks	58	1,415	1/110	10/1,344	17,660
Inverter blocks	8	150	1/97	3/291	450
Subtotals	66	1,565			18,110
Regular latch blocks (LSSD)	5	38	1/32	24/768	912
Latch/Enable blocks (LSSD)	31	514	1/32	32/1,024	16,448
Register file	1	1,184	1,184	19,000	19,000
Subtotals	37	1,736			36,360
Driver blocks	4	69	1/32	4/128	276
Receiver blocks	10	41	1/32	0/0	0
Tristate driver/receiver	1	32	32	320	320
Subtotals	15	142			596
Totals	118	3,443			55,066

mum data ready time by an additional 2.1 ns, so that the total improvement in one pass was about 30 percent. This corresponded to a decrease of the total estimated length of the critical nets of about 4.9 percent.

After repositioning (and the corresponding resizing), the set of critical modules included only two modules that were not critical before. Resynthesis could reduce their delay by about 1 ns, which corresponded to an equivalent reduction of t^* . For this circuit example, no other timing improvement could be achieved; the maximum data ready time was 49.9 ns and the total reduction of t^* was 31.3 percent. At this point, the performance of the circuit was limited by the structural decomposition on which these timing techniques have no control.

Layout Design

The chip image, except for the bonding pads which form the perimeter of the chip, is shown in Figure 7.22. The total area with bonding pads is 72 mm² and has been constructed using a 1.3 micron production standard CMOS process with two levels of metal. The major modules seen in the figure are a stack, a general-purpose register (GPR) file and the remainder, which comprises mainly the control logic and related latches. One of the

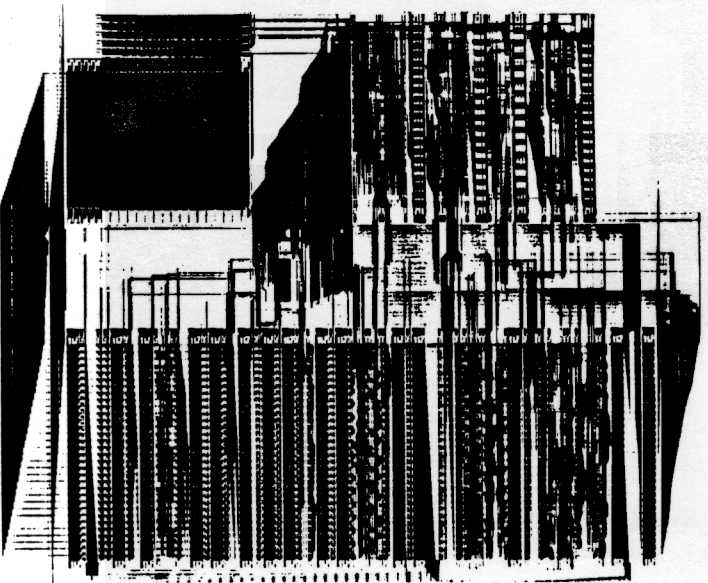


Figure 7.22 Chip image (excluding bonding pads) of the 801 processor.

major constraints dictating the floor plan at this level was that the stack, which ideally would contain the GPR (more 160 nets connecting the top of the stack with the top of the GPR), had to be folded or else the chip would be too long in one dimension. The remainder was originally decomposed so that it could be distributed with the macros being controlled. However, once the stack is folded, the space under the GPR is free, and hence, in this case, is used for the remainder for this chip floor plan. Finally, the remainder was constructed as a stack because a large clock driver was required for each macro to ensure that clock signals were well distributed. Clearly, more area can be made free by distributing the control as part of the stack along one side, thus essentially eliminating the two major global wiring channels. However, for better area efficiency more function should be added to this chip.

Nevertheless, the chip as is, in area and delay, is slightly smaller and somewhat faster than another chip containing the same functionality and technology ground rules, but designed by hand. We attribute these results to:

- a. The use of domino logic, a two-phase domino clock, and linear transistors for the gates
- b. The extraction and use of don't cares for logic minimization
- c. The automatic design of a normal stack which includes much more than a normal data flow stack

Even with the floor plan shown, some improvements should be tried, something that is quite easy with a silicon compiler. For example, better area can be obtained by turning the GPR so that its 160 data nets enter the side of the stack directly. Similarly, the remainder should be rotated 90° clockwise so that its nets enter the middle vertical channel directly. Both of these changes require reconstructing the stack. This is a relatively easy task for the YSC system, but would be quite difficult for other design methods, including other silicon compilers.

7.5 Conclusions

The YSC system has been designed to satisfy its primary goals: automatic compilation, competitive chip design, and creation of a design environment for experimenting with microsystem architecture and CAD algorithms. The synthesis algorithms have been chosen because they are guaranteed to complete without conflict or the need for intervention. In other words, they support automatic silicon compilation. The synthesis and optimization algorithms are the strength of the YSC system, because they allow the user to explore different system solutions and they offer leverage over manual design procedures. A key part of the system is the ease with which it can be enhanced, thus offering the possibility of future extensions as algorithms are improved.

The compiler structure and hardware description languages are sufficiently general to allow a wide range of digital systems to be designed with relative ease. The compiler input languages allow the user to accurately control the implementation if desired. Otherwise, the user need only specify the system behavior and let the compiler make the major decisions about the system implementation.

Our original goals have been fulfilled by the results of the compilation of a medium-scale, 32-bit processor. We could achieve different system solutions, explore the benefit of pipelined architectures, and eventually achieve a chip image that compares favorably with manual design of the same architecture.

7.6 Acknowledgments

Several people contributed to the Yorktown silicon compiler system. We take this opportunity to express our appreciation to all those who helped us in building the system. In particular we would like to acknowledge the contribution of Curtis McMullen to the logic editor (YLE) and the linker, of Norman Brenner to the YLL language and processor, and of Lukas van Ginneken to the annealing algorithm and the global routing. Within IBM Research, Marian Mack supported the layout design programs, C. L. Chen provided us with insight into the design and technology problems, and Yiannis Yamour helped in understanding logic and architectural issues. Several visitors to IBM Research helped in writing experimental tools and in preparing the description of the 801 processor: Willem van Bokhoven, Jochen Jess, Jacob Karzenelson, Richard Rudell, and Jan-Edzard Talsma. Within IBM Corporation, we also benefited from interactions with other IBM divisions, especially Jim Davis at Boca Raton, Bill Griffin and Ralph Kilmoyer at Burlington, and Gunther Machol at Los Garos. We would also like to thank Carla Otter for helping us in preparing this manuscript.