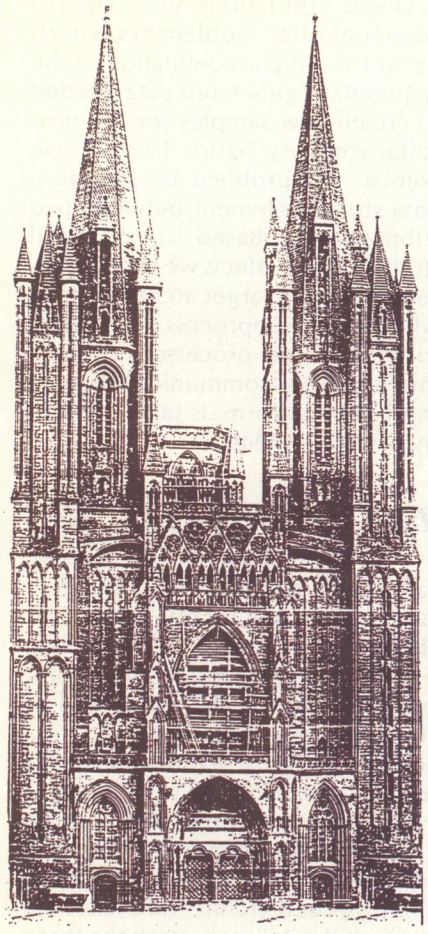


Cathedral-II: A Silicon Compiler for Digital Signal Processing

H. De Man, J. Rabaey, P. Six, and L. Claesen
IMEC

The article describes the status of work at IMEC on the Cathedral-II silicon compiler. The compiler was developed to synthesize synchronous multiprocessor system chips for digital signal processing. It is a continuation of work on the Cathedral-I operational silicon compiler for bit-serial digital filters.

Cathedral-II is based on a "meet in the middle" design method that encourages a total separation between system design and reusable silicon design. The CAD system includes a rule-based synthesis program, a procedural program, and a controller synthesis environment. Processors are synthesized in terms of modules called from automated reusable module generators. Chip layout is done on a floor planner. An expert subsystem verifies correctness during silicon design and generates functional and timing models for verification at the module and chip levels.

We can easily define a silicon compiler in general terms as a software system supporting chip layout synthesis starting from a behavioral description at the algorithmic level. However, standardization beyond this simple definition can be risky. We believe that "the" silicon compiler simply cannot exist, any more than "the" software compiler can exist. It is this belief that led us to develop an application-specific silicon compiler —that is, a compiler necessarily tied to a particular application, in this case, digital signal processing.

Our compiler, called Cathedral-II, is based on what we call a "meet in the middle" design method. Its name describes the separation of system design and reusable silicon design and the gradual move toward a middle ground during the synthesis process.

The specific function of Cathedral-II is to synthesize synchronous multiprocessor chips for digital signal processing. It was developed following a series of five steps:

1. Define a wide, but concise, class of system design applications.
2. Define, based on manual design exercises, a target silicon architecture and its associated layout style.
3. Define a design strategy based on available designer skills.
4. Define the behavioral language and the silicon modules.
5. Then and only then develop the CAD tools, with emphasis on the "D" and the "A."

0740-7475/86/1200-0013$1.00 © IEEE

THE APPLICATION

Cathedral-II is based on complex algorithms in digital signal processing, which is playing an increasing role in modern VLSI-based systems. DSP spans data throughput from 1K bps up to 100M bps. Therefore, even this area is too wide for a single design style or silicon compiler.

The low-throughput end of DSP is well served by the new generations of general-purpose DSP chips. However, while the first implementable algorithms addressed such simple cases as digital filtering, today's applications require throughput from audio sample range up to 1M-bps sampling rates. Consequently, we need very high precision or highly complex algorithms involving block data processing, matrix manipulations, multiple data rates, and a lot of decision making besides number crunching. Examples are digital audio, speech processing, smart modems, and robotics. Also, these applications would be a lot more attractive if they included adaptable I/O periphery on the same chip.

General-purpose digital signal processors are not very well-suited for the implementation of these algorithms. On the other hand, a full custom solution is too costly in design time and, more important, in time to enter a highly competitive market.

Because of the highly specialized nature of such algorithms, we assume that the algorithm designer is able to do the silicon implementation. Therefore Cathedral-II addresses highly complex, block-oriented DSP algorithms in the audio to near video frequency range.

Figure 1 shows a pitch extraction algorithm for speech.[1] This example has been used to study the design process and to define the tools in Cathedral-II. Real and imaginary parts of blocks of 64 frequency components from a discrete Fourier transform processor are first transformed into an amplitude spectrum. By averaging, a threshold is computed to eliminate irrelevant spectral components. From the remainder, the maxima in the spectrum are computed and then compared to 49 sieves with meshes at octave distances. Finally, the best match is computed as the pitch value. As the figure shows, this problem, typical for third-generation DSP algorithms, naturally decomposes into a number of subprocesses, which are fairly independent of each other.

These algorithms give rise to communication bottlenecks, which occur from the accumulation of the sequentially generated data needed to create new samples for the next subprocess. As Figure 1 shows, we combat the problem by putting a data storage element between two subprocesses. Based on a careful study of these effects we have therefore defined a target architecture in which each subprocess is assigned to a dedicated processor, and the interprocessor communication, in its most general form, is taken care of by switched RAMs.

THE ARCHITECTURE

We have selected a flexible, tailorable multiprocessor architecture as the target for our design synthesis. The architecture is flexible enough to evolve into a single, general processor at the lower end of the frequency spectrum and into a set of parallel, hard-wired data paths at the upper end. As such, it spans many applications—speech, audio, robotics, telecommunications, and image processing. The basic idea of the proposed architecture is to add enough flexibility to attack the three basic bottlenecks that normally limit

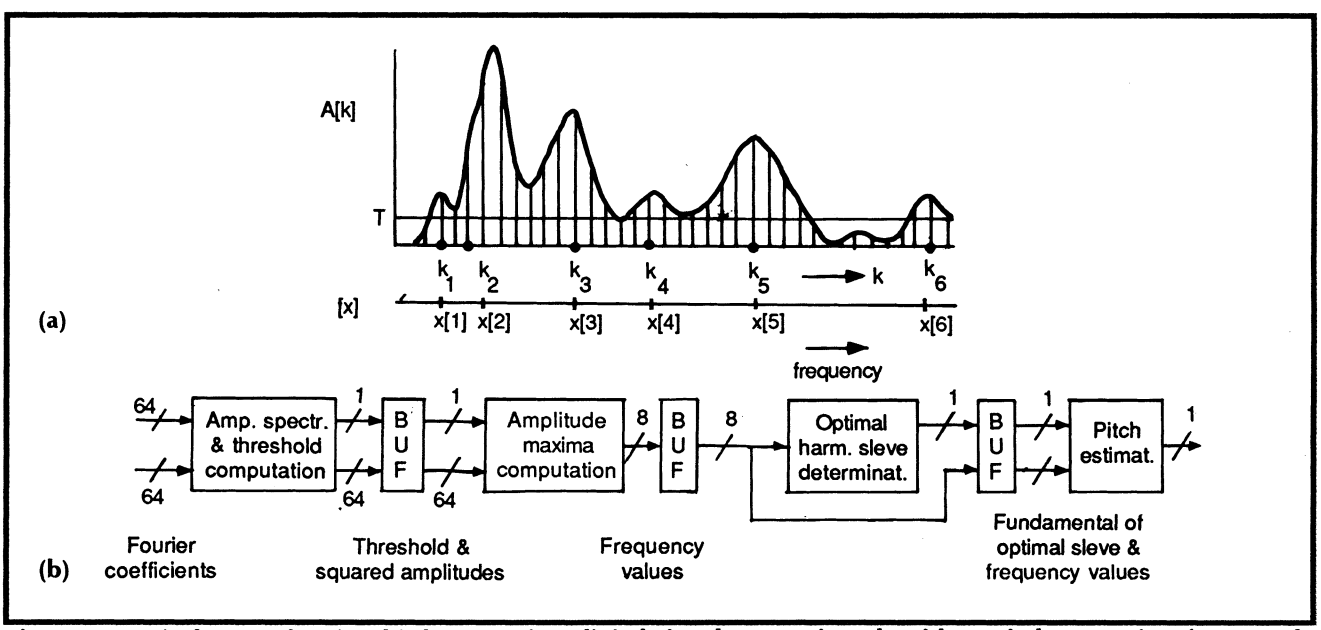


Figure 1. Typical example of a third-generation digital signal processing algorithm, pitch extraction for speech analysis (a). The block structure of the algorithm (b) consists of independent processes, each of which is assigned to a dedicated processor.

data throughput in general-purpose signal processors: arithmetic throughput, congestion in data transfer, and controller delays when performing conditional operations (frequent in third-generation algorithms).

At the highest level, the proposed architecture is composed of a set of concurrent operating processors. Each executes one subtask of the algorithm and is optimally tuned to perform just that one task (Figure 2). Each processor operates relatively independently of protocols, exchanging only data that is global among processors.

Depending on the data exchange rate and the amount of buffer needed, different protocols can be selected: synchronously switched RAM buffers, FIFOs, or request- and acknowledge-based synchronization. Communication with the outside world is over an I/O frame that can support a large range of I/O protocols (parallel to serial, synchronous to asynchronous).

Each processor consists of a dedicated data path and a controller. The data path is optimized for the task(s) it has to perform and is assembled from a set of selected execution units (EXUs), interconnected by a restricted number of customized buses. Each EXU contains a register file (of variable size) at its input side. This structure makes it possible to avoid the arithmetic and data transfer bottlenecks. Studies have shown that the following EXUs are sufficient to span most of the target application space:

- a general-purpose (but rather inefficient) ALU/Shift unit
- an address computation unit (ACU) with modulo-counting capabilities
- a parallel multiplier/accumulator
- a parallel-serial divider
- a comparator (for max-min computations)
- a scaler/normalizer (for fixed-point/floating-point conversions)

The first two units are *general-purpose blocks,* while the others are *accelerators.* All these units have been designed with changeable parameters and have been implemented in the module generation environment. Typical parameters are word length, shift dimension, size of register files, and size of multiple array.

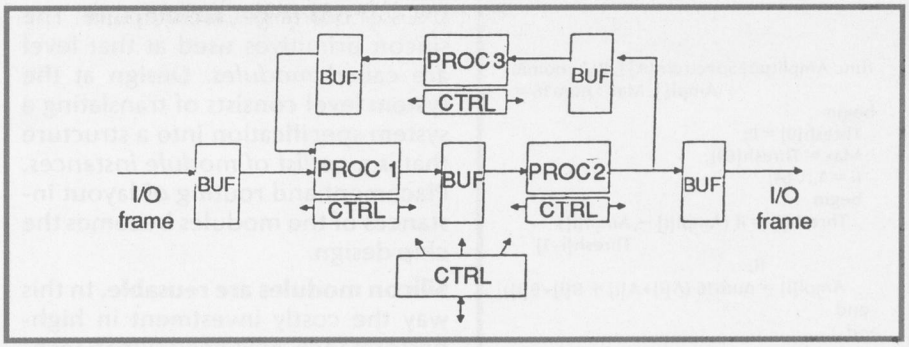


Figure 2. A multiprocessor architecture.

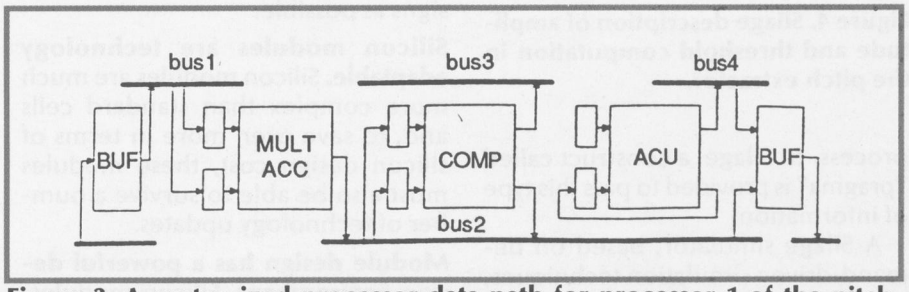


Figure 3. A customized processor data path for processor 1 of the pitch extraction algorithm in Figure 1.

An example of a processor data path built with this strategy is shown in Figure 3. It is used to compute the amplitude spectrum of a signal, given the complex frequency domain spectrum, and at the same time to determine the maximum amplitude. It consists of three concurrent units: a multiplier/accumulator, a comparator, and an ACU. This data path can perform an amplitude computation and a maximum update in two cycles (average).

We chose a multibranch, microcode-based controller to control data flow through the data path. This structure can handle a large span of algorithms flexibly and efficiently. It can support algorithms that require heavy decision-making as well as those that are repetitive. Some EXUs can also have a local controller to help reduce the complexity and the size of the central controller. An example is the decoder of the register files. Another controller is needed at the highest level to control the flow of data between processors and from the processors to the outside world.

THE LANGUAGE

Different applications obviously require different specification languages; a microprocessor designer will use conceptual constructs different from those a modem designer might consider. Therefore we have selected Silage,[2] a language optimized for high-level description of signal processing algorithms, as the design language for Cathedral-II.

The basic object in Silage is the signal, which is a vector whose components are samples in the time domain from infinity to actual time. The basic operation is a functional application of those signals. In this way, a Silage description of an algorithm resembles a signal flow graph, where nodes are instances of functions and arcs are the signals. Silage supports time domain operations such as decimation and interpolation and allows for the description of equations, decisions, iterations, hierarchy, and finite word-length effects. The Silage description of the algorithm implemented on the data path of Figure 3 is given in Figure 4.

In some cases, the designer may want to pass some structural hints to the compiler to guide the synthesis

```
func AmplitudeSpectrum(A[], B[] : num8)
                    Ampl[], Max : num16 =
begin
  Thresh[0] = 0;
  Max = Thresh[64];
  (i = 1 . . 64)
  begin
    Thresh[i] = if (Ampl[i] → Ampl[i])
                              Thresh[i-1]
                fi;
    Ampl[i] = num16 (A[i]*A[i] + B[i]*B[i]);
  end
end
pragma Processor (1,AmplitudeSpectrum);
```

Figure 4. Silage description of amplitude and threshold computation in the pitch extractor.

process. In Silage, a construct called "pragma" is provided to pass this type of information.

A Silage simulator, based on demand-driven simulation techniques, is currently under development. Once the system is successfully simulated, synthesis can start.

MEETING IN THE MIDDLE

After choosing the application and defining the target architecture the next step is to select a design strategy. We are typically addressing the quick-turnaround design of complex systems on chips with about 300,000 devices. The complexity of the algorithms to be implemented far exceeds that of the circuit or even the logic gate level. Moreover, since silicon designers are so scarce, we cannot exploit the potential of such systems, unless they are directly designed by the system engineers, without detailed knowledge of silicon implementation.

Therefore, silicon design knowledge (at the micron level) must be localized in reusable modules at the MSI/LSI level familiar to the system designer. Figure 5 shows a design scheme that satisfies these requirements. We call it *meet-in-the-middle* design methodology, and it can be characterized as follows:

System design is separated from silicon design. The interface is located at the level of arithmetic/logic operator blocks, data storage, controllers, and I/O units. These are the EXUs of our target architecture. The silicon primitives used at that level are called *modules*. Design at the system level consists of translating a system specification into a structure that is a *netlist of module instances*. Placement and routing of layout instances of the modules becomes the chip design.

Silicon modules are reusable. In this way the costly investment in high-performance, advanced silicon technology design is limited, and the cost is written off over as many designs as possible.

Silicon modules are technology adaptable. Silicon modules are much more complex than standard cells and, to save even more in terms of silicon design cost, these modules must also be able to survive a number of technology updates.

Module design has a powerful design environment. Silicon modules, however, are not enjoyed by a particular foundry or CAD vendor. Since the competitive edge between system houses will not be in the technology but in the architectural technique and in its implementation, we expect module design to require a powerful design environment for a local team of silicon designers. This may not yet be the case, but we expect it to happen in the future.

In this method, system design is top-down to the usual intermediate level. Silicon designers compose, in the usual bottom-up fashion, LSI-level modules from functional building blocks, which in turn are composed of logic leaf cells at the transistor level.

Hence, both parties "meet each other in the middle" of the design abstraction levels. Scarce talent is optimally used and the design process corresponds to the usual patterns.

There are some fundamental deviations from classical design at the silicon and system levels, however. These usually stem from the fact that module generators are generally software procedures rather than fixed geometrical structures. Also, we expect system designers to think at the algorithmic rather than the structural level.

THE CAD SYSTEM

Figure 6 shows the CAD toolbox used in Cathedral-II. The figure shows that in the middle of the design abstraction is a separation between the silicon and the system designers. The link between them is a "call" to a limited set of EXUs as defined in the target architecture.

The system designer defines the system at the behavioral level in Silage. Coupled to this language is the high-level simulator to verify the behavioral correctness of the algorithm. Using the throughput requirements and a set of expert design

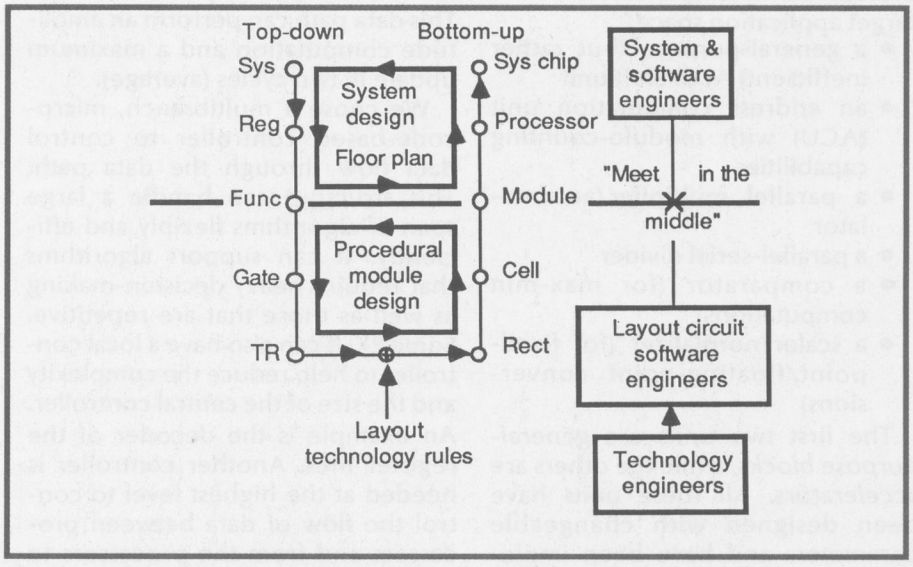


Figure 5. "Meet in the middle" design methdology.

rules, Silage code is first optimized with respect to the functions of the target architecture. The optimized code is then compiled directly into structure (a netlist in terms of the limited set of predefined EXU silicon modules). Clearly most of this synthesis is dictated by the target architecture, and therefore quite a large part of it is rule-based. In Cathedral-II, it was implemented in Prolog, which is discussed later.

We also, in principle, allow the designer to specify the design at intermediate levels, all the way down to structure, but at the price of increasing the amount of lower level simulation, the redesign risk, and the time to the market. When all calls to the modules are successful, the chip is ready for floor planning. This can be done either interactively or by automatic placement and routing techniques.

In Cathedral-II, we include the synthesis of the algorithm directly into data paths and control logic. That is, we are aiming at a true silicon compiler, in contrast to most commercial systems today, which are limited to floor planning and to module generation.

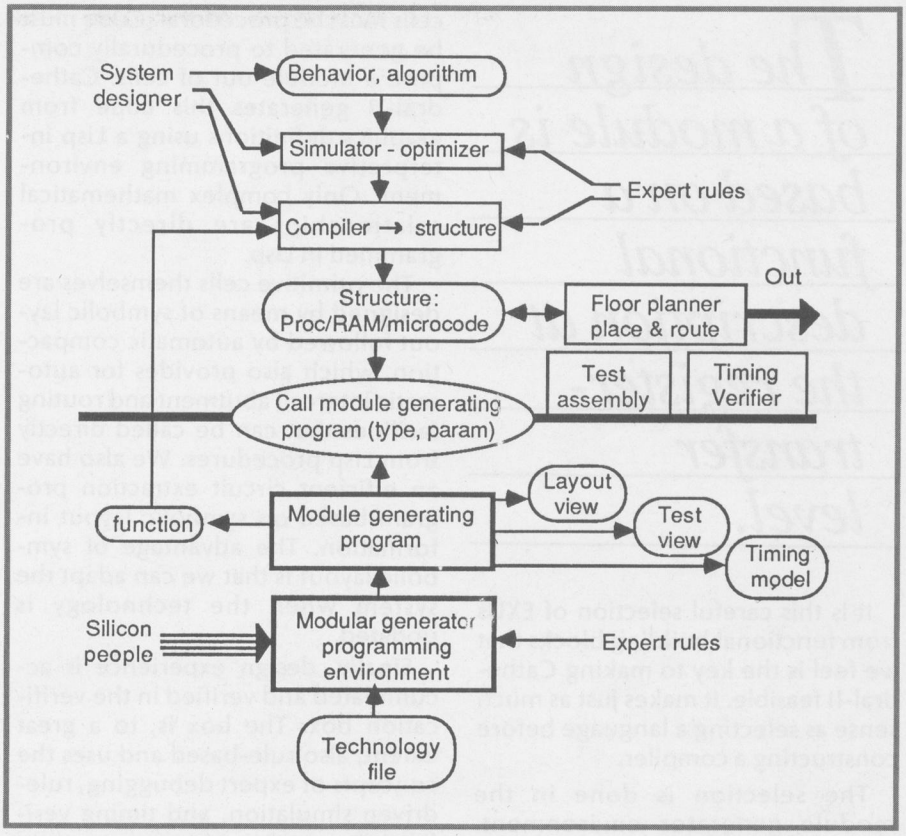


Figure 6. The CAD toolbox in Cathedral-II.

How does this scheme work? First, just as in a software language compiler, success is intimately linked to a clear definition of the set of modules. Consequently, we must generally limit the variability of the modules and carefully consider the application when choosing EXUs.

Second, in view of the evolving technology, we cannot design in a fixed technology but must be able to adapt to technology changes.

Finally, we need a lot more information from a module generator than just the *layout view,* which in itself should consist of a bounding box view, as well as a full layout view. Other views include a functional view, a timing view, and a test view.

A *functional view* is an RTL-level function, in case the design was done at the structural level, requiring simulation. Even if full synthesis is used, designers will still want to simulate the actual chip. This simulation is possible in our system by the Hilarics-Logmos program,[3] which is a register-transfer simulator that allows modules to be included and symbolic microcode descriptions to be defined.

A *timing view* is necessary, since although the synthesis compiler can take first-order throughput requirements into account, it is only after placement and routing that a full performance check can be done.

In Cathedral-II, we are following a bottom-up hierarchical generation of timing models in a knowledge-based program, called Slocop, which closely follows the composition procedure of a module. At the floorplan level, interconnection parasitics are taken into account to check global timing. If unsatisfactory, buffer sizes are adjusted. If still not satisfactory placement is changed, or a pragma is formed at the Silage level to call for higher performance modules or to increase parallelism in the algorithm.

Finally, although it is not in the actual Cathedral-II version, a *test view* is needed with each module. We envision that most modules with a high degree of structure will be C-testable. That is, depending on the particular structure, a small set of word-length-independent patterns will guarantee module testability. A *test assembly* program would then be implemented at the level of the floor planner. For the particular architecture, the program would generate total testability for the whole chip.

In contrast to the traditional Calma type of fixed layout or even the tiling of fixed cells, our design style calls for modules to be written as *procedures* with adjustable parameters. Parameters range from simple word length to conditional composition in terms of functional building blocks or size of output buffers. These procedures should also generate the other views needed by the synthesis programs.

This style causes two problems. First, because silicon designers will confront the software more often, and most are not trained to understand the software, we will have to hide the code generation as much as possible. Second, we must carefully restrict the set of modules—a matter of carefully choosing EXUs according to the architecture.

The design of a module is based on a functional description at the register-transfer level.

It is this careful selection of EXUs from functional building blocks that we feel is the key to making Cathedral-II feasible. It makes just as much sense as selecting a language before constructing a compiler.

The selection is done in the module generator environment. EXUs are composed from functional building blocks like adders, comparators, and registers, which are, in turn, composed of logic cells.

Figure 7 shows the anatomy of the module generator in Cathedral-II. The arrow types indicate clearly the "create" (silicon designer), "generate" (call from silicon compiler), and "adapt" (to technology rules) functions needed for such a programming design environment.

The Cathedral-II module generation environment provides not only the layout environment but also a first version of an *expert verification system*. The verification system is needed for the following tasks:

- to generate the functional/timing and test models during the "generate" phase
- to verify the modules during the "create" and "adapt" design phases

The design of a module is based on a functional description at the register-transfer level. This description (and its simulation) can be done by the Hilarics-Logmos system.[3] It is the documentation link between the system and the silicon designer.

In a module generator, the connectivity and relative placement of cells must be procedural. Code must be generated to procedurally compose a module out of cells. Cathedral-II generates this code from graphics definitions using a Lisp interpretive programming environment. Only complex mathematical relationships are directly programmed in Lisp.

The primitive cells themselves are designed by means of symbolic layout followed by automatic compaction, which also provides for automatic intercell abutment and routing facilities that can be called directly from Lisp procedures. We also have an efficient circuit extraction program based on symbolic layout information. The advantage of symbolic layout is that we can adapt the system when the technology is updated.

Finally, design experience is accumulated and verified in the verification box. The box is, to a great extent, also rule-based and uses the concepts of expert debugging, rule-driven simulation, and timing verification.

Design synthesis

Architectural synthesis, as envisioned in Cathedral-II, is based on two cornerstones:

- the targeting of the synthesis toward a single (but flexible) architecture, allowing for considerable pruning of the search space and making it possible to use dedicated optimization techniques to exploit properties of the architecture
- the use of the module generation environment as a module knowledge database that can be queried by the synthesis tools, allowing design decisions to be based on physical implementation details (speed, area, power)

This strategy avoids the inefficiency of previously published design systems,[4,5] which attacked synthesis problems in a purely top-down fashion. These cornerstones are reflected in the outline of our synthesis system, as presented in Figure 8. It consists of several subtasks, which were initially defined through a

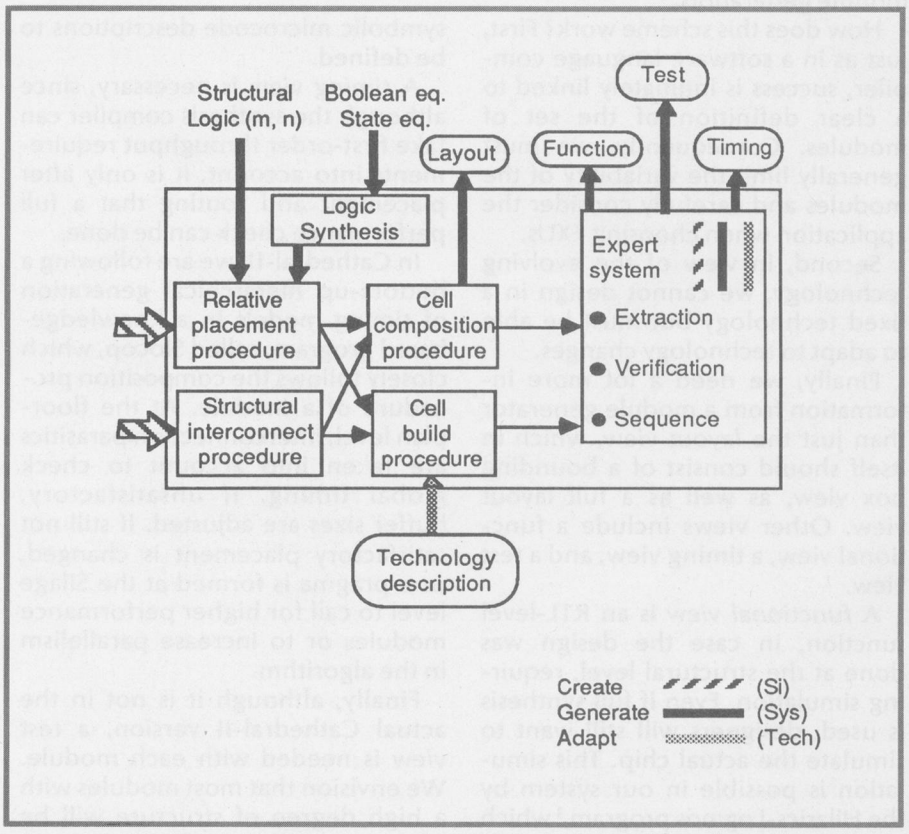


Figure 7. The module generator software environment has three functions: create (silicon), generate (system), and adapt (technology).

number of manual design exercises on real industrial examples. These exercises have shown that synthesis is an iterative and interactive optimization process. Therefore, we envision an open design system that allows the experienced designer to interfere with synthesis at various intermediate levels. Consequently, we need entry points (description languages) and verification means (simulators) at all those levels. However, user interference, should be avoided as much as possible, especially at the low levels, and the system should be intelligent enough to restrict the iteration loops to a local level (if possible).

As Figure 8 shows, input to the design system is Silage. The correctness of this description can be verified using the Silage simulator. The simulator outputs can also be used as a reference to verify the effects of further design steps.

Algorithm partitioning

The first step in synthesis is to partition the algorithm. The Silage description is split into distinct sections, each of which is mapped onto a separate processor. Each processor is then synthesized independently from the others, which is possible because the interprocessor communication protocols are transparent. The partitioning itself is based on a number of often contradictory observations, such as load balancing, data transfer bottlenecks, cut-set analysis, and computational requirements. In most cases, however, an educated guess can be made by simple inspection. Therefore, we have decided for the first version of Cathedral-II to leave the partitioning to the user by means of pragma statements in the Silage description.

Data path synthesis

The applicative description of each processor is transformed into a procedural one in two steps. First, a customized data path is synthesized, and the high-level operations are mapped onto a set of register-transfer-level (RTL) operations for this data path (Figure 8). Second, the optimal timing of the RTs is computed. This is referred to as "microprogram scheduling" (discussed later). The intermediate RT language has nonconflicting procedural (fixed architecture) and applicative (no explicit timing specification) properties.

The synthesis of a processor data path includes several tasks. A limited number of EXUs are stored in the library. Using this knowledge and the throughput specifications, the EXUs required to realize the Silage operations are selected. Dedicated bus connections between different EXUs within a single processor are also chosen.

Finally, the chosen EXUs are stripped of unnecessary operators, and internal parameter values are assigned. These decisions are influenced by area and throughput specifications, which determine the choice between multiplication alternatives (hard-wired or shift-add-based multiplier) or concurrency levels. During synthesis, Silage variables are assigned to data path register files (containing multiple storage fields). In this way, the Silage operations are transformed into RT operations. The result is a "black box" description of the controller, which serves as input to the microprogram scheduler.

An automated synthesis tool supported by knowledge-based techniques is under development. In a first pass, the Silage code is preprocessed. This includes a simplification of the source code and a number of local transformations (loop handling, reordering of singular variables, etc.). The preprocessor also preassigns operations to EXUs. These preassignments are based on a global overview of the problem and are used to guide the data path mapper. The mapping tool itself, implemented in Prolog, tries to unify the high-level Silage equations with internal low-level equations, each of which represents a fundamental operation of an available EXU. Successful unification may mean fulfilling certain conditions, under the form of new high-level equations. Synthesis is governed by a program called *inference engine*, which advises on the unification rule most appropriate to a given high-level equation. The architectural properties of the EXUs are stored as a set of rules, which keep the tool flexible with respect to architectural changes.

Microprogram scheduling

Register transfers are fundamental operations from a control point of view, since each is to be realized in a single machine cycle. In microprogram scheduling, the optimal mapping of RT operations to time (machine cycles) is computed (optimal meaning with a globally minimal cycle count). Conditional RTs and FOR constructs are allowed in the RT language. A FOR loop can be regarded as a pragma by which the designer prescribes a limited ordering of operations. Loops are straightforwardly mapped into the microcode. The algorithms are scheduled statically at compile time, as opposed to dynamically at runtime, thus saving complex data-driven control protocols. The scheduling tool should take into account all implications of the actual controller structure, such as the amount of pipelining and the type of controller logic, as well as all restrictions to prevent conflicts between RTs. Therefore, a preliminary choice between alternative controller types is required (we have selected the multibranch, microcode-based controller).

From the scheduling constraints (data precedence constraints, resource conflict constraints, and controller pipelining constraints), a general integer linear program (ILP) can be set up, which can be solved using standard procedures. Scheduling is NP-complete; therefore large problems require the assistance or replacement of the ILP technique by heuristics. After analyzing the lifetime of the register file variables, the optimal schedule is converted into a readable symbolic microcode.

After scheduling is completed, the cycle count is evaluated, which may force backtracking to the data path synthesis level. Excessive cycle counts may lead to faster implementation schemes or changes of the concurrency level, while low cycle counts may suggest cheaper data path structures.

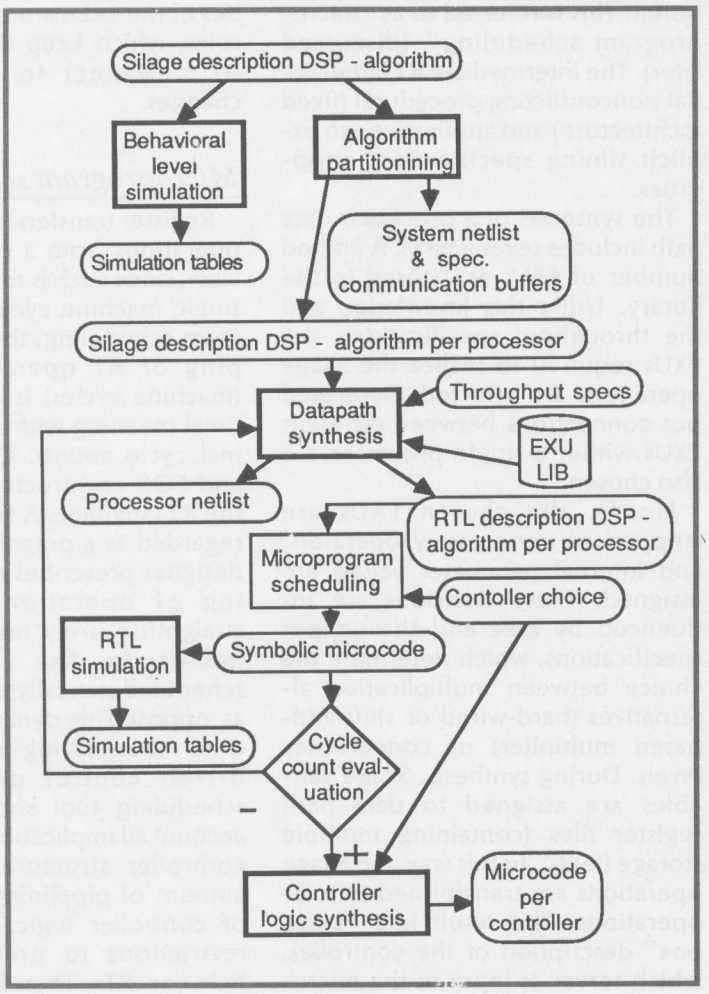


Figure 8. Overview of functions, languages, and simulators in the Cathedral-II synthesis process.

Controller synthesis

From the symbolic microcode, the controller decoding logic is synthesized. To feasibly compare controller realizations (microcode ROM-based controllers, multilevel logic implementations, PLA-based sequencers, etc.), we have a unified environment with a single input language. This environment provides a library of utilities for synthesis, optimization, and layout generation of controller structures such as state assignment, logic minimization, partitioning, and assembly of regular arrays using symbolic layout techniques. In this way, new controller architectures are easily introduced. A prototype version of the environment is being implemented using the multibranch controller as a test case.

This synthesis system has been used to generate the design of a high-quality pitch tracker for audio systems (Figure 2). The result (Figure 9) is a tracker with four processors that occupies only 37 mm^2 in a 3μ CMOS process.

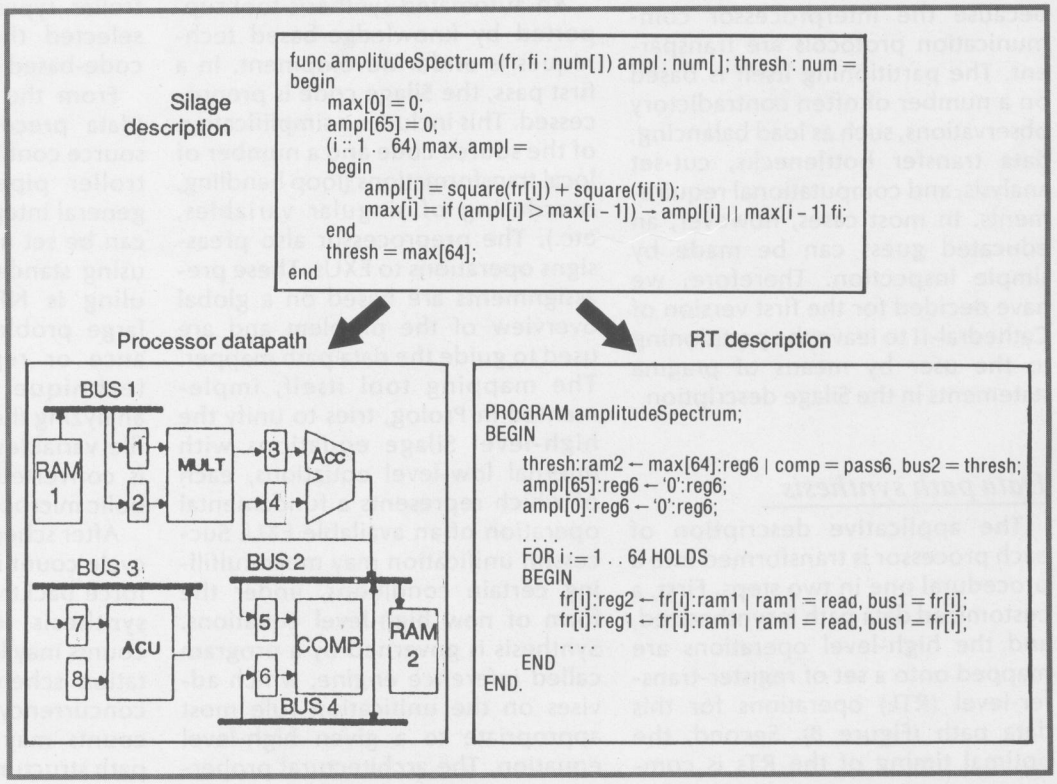


Figure 9. Overview of the datapath synthesis process: translation of Silage into structure and register transfer representation.

Module generator

To define a module generator, we need a system that enables us to design the leaf cells and provide high-level commands to express the composition of a module as a function of the parameters. Since the composition rules may be as complex as

"Fit a number of leaf cells so that their terminals connect by abutment"

or

"Route a set of nets"

the algorithms to support these high-level commands must also be provided in an integrated environment.

Many existing module generators have a delay between definition and execution, introduced by the compilation and linking steps of the traditional programming languages they are implemented in. To overcome this problem, we designed an interactive environment that gives graphical feedback each time a composition rule of the module generator is defined, allowing mistakes to be corrected immediately after they were made.

To describe a module we capture four types of information.

Structural: The submodules

Topological: The relative placement of the submodules

Connectivity: How the submodules must be interconnected and where the input and output terminals of a module are

Construction: How submodules fit together and which interconnections are realized by abutment or routing.

In the first version of the module generator in Cathedral-II, called MGE, structural and topological information is entered at the same time. Components are added on grid points to represent their relative ordering but not their final coordinates. The command (ADD-COMP FA |1..N| 1), for example, adds a horizontal row of length *N* of FA cells. As Figure 10 shows, the cells are represented by their outline at this stage, resulting in a clear picture and fast display.

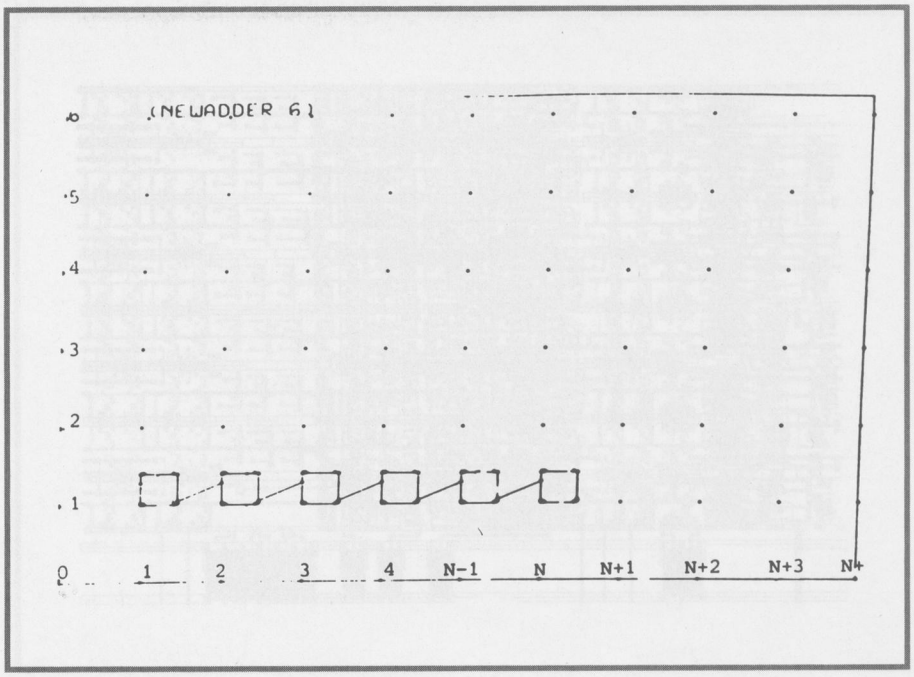


Figure 10. Structure and connections of a ripple adder.

Connections are also easily defined by specifying a net name, and the list of terminals of the connected components. The command (ADD-CON (RIPPLE |2..N|) (COUT |1..(- N 1)| 1) (CIN |2..N| 1)) defines the connections of the ripple signals in the adder example. No assumption is made at this stage as to how the connections will be realized.

Indices of array-like structures can be expressions; (*N*-1) is the last ripple signal, for example. MGE allows us to associate an expression with each horizontal or vertical grid line on the display (see *x*-axis in Figure 10). The creator can then, by

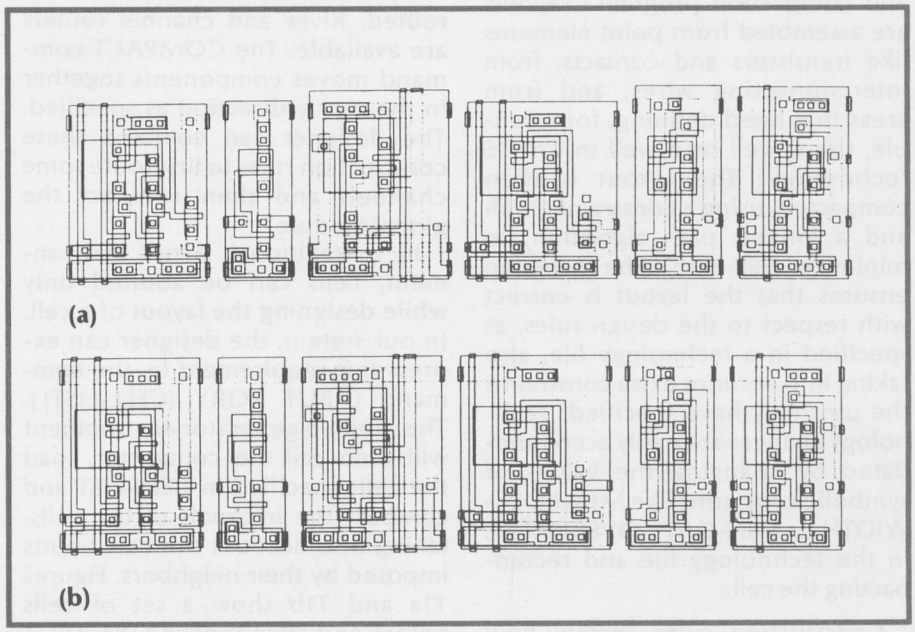


Figure 11. A set of symbolic cells before (a) and after (b) automatic fitting.

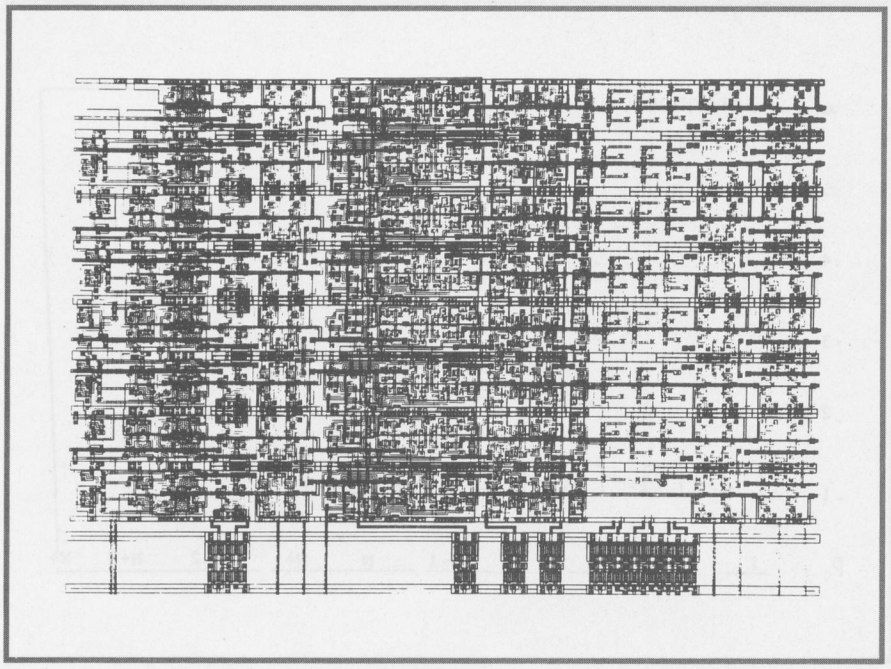


Figure 12. Add-Shift-Compare module as generated by the module generator environment.

simply placing cells and pointing at terminals graphically, automatically generate the Lisp procedure. This is the first step towards graphical definition of module generator code.

For the design of the leaf cells we use Cameleon, our symbolic layout and compaction program.[6] Layouts are assembled from point elements like transistors and contacts, from interconnecting wires, and from areas that need defining, for example, the n-well or p-well in CMOS technology. The layout is then compacted, using a constraint graph and a longest path algorithm, to minimize cell area. The program ensures that the layout is correct with respect to the design rules, as specified in a technology file, also taking into account extra constraints the user may have specified. Technology changes are easily accommodated by changing the values of symbolic constants, like MIN-POLY-WIDTH or MIN-GATE-OVERHANG, in the technology file and recompacting the cell.

Construction rules specify how the real layout of the module must be made starting from the topology, the connections, and the basic cells. Either the cells fit on each other and are pushed together, or routing must be added to make the connections. Both options are available in MGE. The ROUTER command allows the user to define a horizontal or vertical channel and the nets that have to be routed. River and channel routers are available. The COMPACT command moves components together in the *x* or *y* direction as specified. The designer can combine these construction rules to first route some channels and then compact the global module.

In a traditional layout environment, cells can be abutted only while designing the layout of a cell. In our system, the designer can express this requirement by the command (ABUT XORY (CELL-LIST)). The module generator environment will then call the compactor, load the cells specified in CELL-LIST and compact the internals of the cells, taking into account the constraints imposed by their neighbors. Figures 11a and 11b show a set of cells before and after applying the ABUT command.

The final step of an editing session in MGE is to save the commands into the module generator procedure. This is done automatically each time the designer stops working on a module. From this procedure, modules can be generated for any set of parameters in the allowed range.

To avoid inventing a complete new language and to take advantage of the interpretative nature of Lisp, the first version of the module generator environment was implemented as a superset of Lisp. For the more computationally intensive algorithms like compaction and routing, Pascal and C programs are called from Lisp. The system has been tested through the design of five EXUs defined for a particular architecture. The resulting layout of the add-shift-compare is shown in Figure 12.

The use of Lisp as the basis of the MGE provides the creator with all the flexibility of a high-level programming language to define module generators that require more complex composition procedures. An example is the comparator shown in Figure 13.

Module verification

One of the problems with module generators is how to prove their correctness. The traditional way is by simulation. However, simulation is a subjective test method that depends entirely on the patterns defined by the designer to detect expected problems. It is costly in CPU time and is not guaranteed to capture unexpected problems.

To overcome these drawbacks we have developed Dialog,[7] a set of rule-based analysis tools that allows us to detect violations against basic design principles. We have also developed Slocop for timing analysis. These tools are based on the Lisp-like Lextoc language,[7] which provides powerful facilities to express rules about MOS circuits. Figure 14 gives an overview of the verification system.

We begin with *MOS network extraction*. To verify the design we first have to extract from the layout a

model for the cell on the circuit level. This can be done very easily, since the stick diagram contains the devices and the interconnection wires. The internode capacitances can be found by detecting the points where elements cross.

Next from the extracted MOS transistor network the user can decompile and check clocking rules. The system first determines if the network belongs to the class of valid circuit configurations, such as static CMOS, using passive multiplexing trees and prescribed registers. All parts of the circuit violating these rules are flagged as errors. The checking is done by partitioning the network into the basic subnetworks that are possible in a given design style. The description of the topology of those subnetworks (static CMOS gates, pass transistor trees with or without bleeder transistors, etc.) resides in a knowledge base. This knowledge base is written in Lextoc. Therefore, in contrast to Crystal,[8] for example, our system is easily extended with new types of subnetworks, and is applicable to any definable class of digital MOS circuits (Figure 15).

A high-level check on the clocking is now done. Starting from the primary clock information, the derived clocks are found, and latches are detected and separated from combinatorial logic. By assigning the appropriate properties to the nodes in the network, illegal combinations are detected and reported. For example,

- A derived clock can only be formed with signals latched on the same clock phase
- A latch data input must not be a primary clock

Figure 16 shows the response of Dialog to a request to verify a clocking rule. The system has identified and highlighted that a loop exists containing only a single clock phase, which could cause a race problem.

From the primitive circuits found in the previous step, we now search for illegal configurations like opens, shorts, and odd NMOS and PMOS combinations. All acceptable circuits are checked to see if they can guarantee correct logic levels. This check is not restricted to static situations but includes dynamic effects like charge redistribution. An example of a possible problem circuit and the rule involved in the checking procedure are in Figure 17.

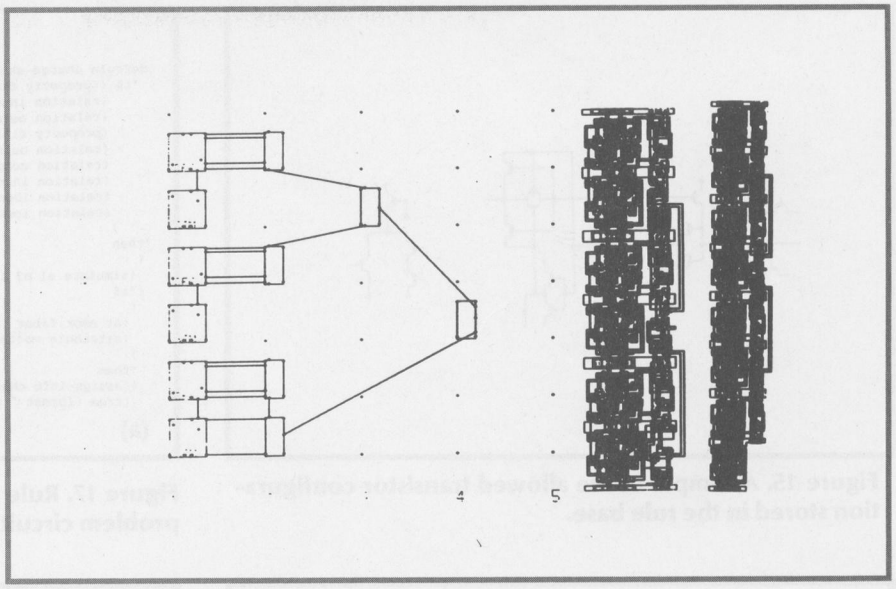


Figure 13. Interpretively generated representation of the structure of a six-bit comparator (a); the layout of the six-bit comparator (b), in which some cells are connected via abutment, some with routing; and the layout of an eight-bit comparator (c), which demonstrates the flexibility of the module generator.

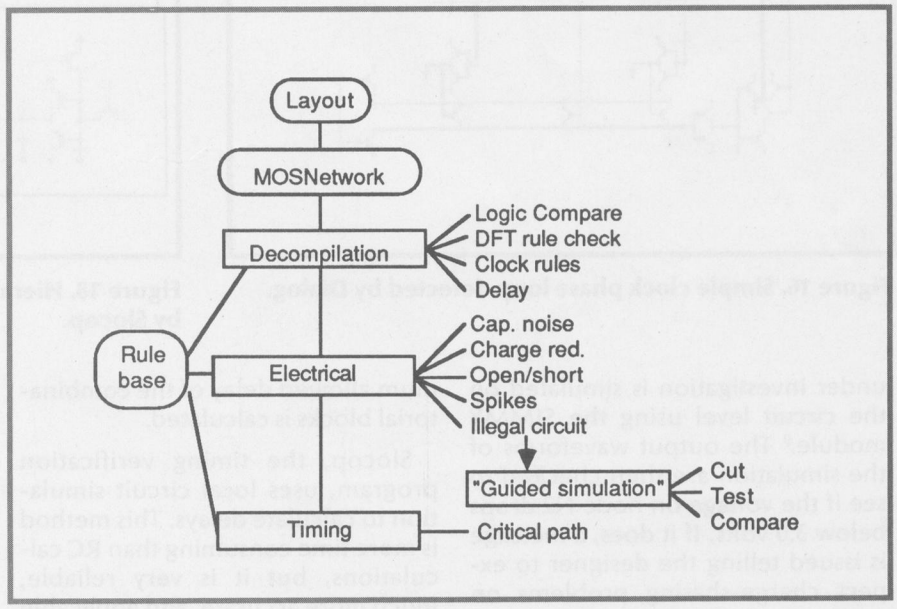


Figure 14. Overview of the Dialog expert verification system.

To diagnose an error on node B caused by charge sharing we have to find the combination of input signals that creates the worst case configuration. This is done by the SETUP-INPUTS procedure. Using these inputs, the part of the circuit

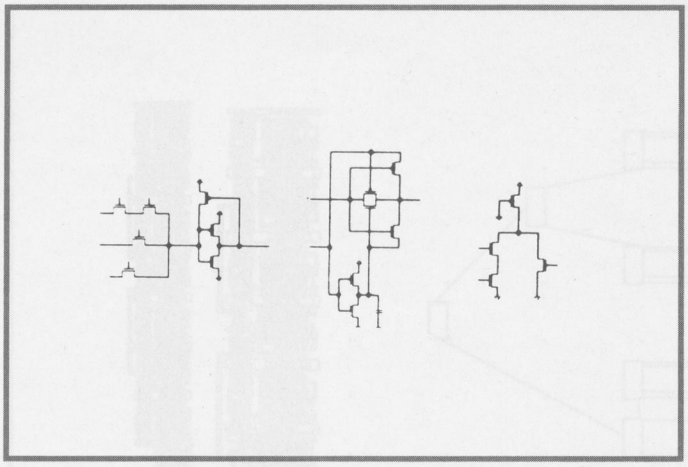


Figure 15. A sample of the allowed transistor configuration stored in the rule base.

```
(defrule charge-sharing-noral ()
( *if ((property fibar-ndyn e1)
       (relation innodes n1 e1)
       (relation outnodes n1 e2)
       (property fibar-pdyn e2)
       (relation outnodes n2 e1)
       (relation compon ecomp  e1 )
       (relation innodes  n1 ecomp)
       (relation ionodes  n2 ecomp)
       (relation ionodes n3 ecomp)
     )
 *then
  (
   (simulate e1 n2 1)
  (*if
   (
    (at exor.fibar (> 3.0) p1)
    (attribute voltage (< 4.0) n2  p1)
   )
   *then
   ((assign-info charge-sharing-error ecomp)
    (true (format " please re-dimension ~a" (instance n3))))))
```

(a)

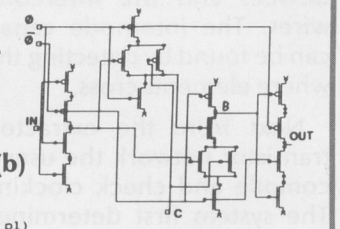


Figure 17. Rule to detect charge sharing (a) and possible problem circuit (b).

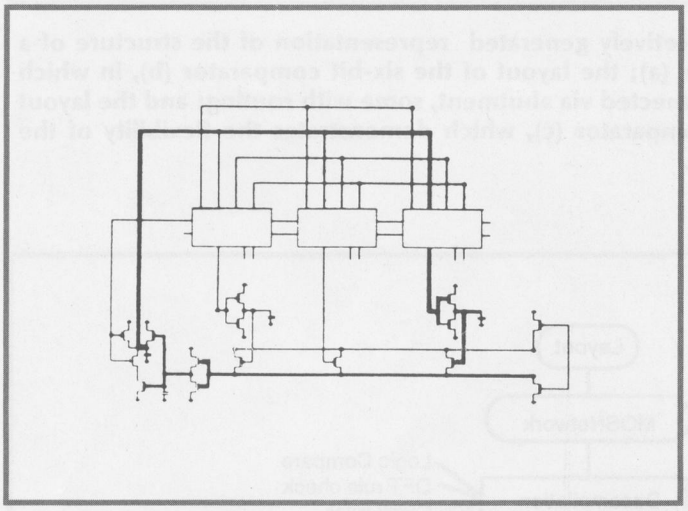


Figure 16. Simple clock phase loop detected by Dialog.

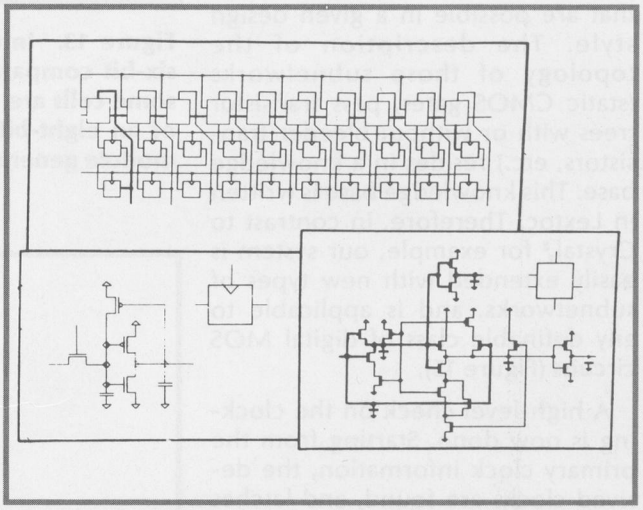


Figure 18. Hierarchical highlighting of critical delay path by Slocop.

under investigation is simulated on the circuit level using the SIMMY module.[9] The output waveforms of the simulation are then checked to see if the voltage on node N2 drops below 3.0 volts. If it does, a message is issued telling the designer to expect charge-sharing problems on node N2. If the designer does not believe this diagnosis, a facility is called to display the conditions creating the problem situation and explain by what successive steps it was detected.

Even if the circuit has passed all the general rules, we must check that it does not violate the timing specifications for the module. From the primary clock specifications, the edges of the clock inputs for the latches are computed and the maximum allowed delay of the combinatorial blocks is calculated.

Slocop, the timing verification program, uses local circuit simulation to calculate delays. This method is more time consuming than RC calculations, but it is very reliable, much more accurate, and applicable to any type of subnetwork that can be described in the program's knowledge base. For global analysis, graph-based algorithms are used, since the large number of paths makes an exhaustive path search impossible. Figure 18 gives a multiwindow output of Slocop, showing the hierarchical annotation of the critical path of an adder module. Slocop uses a critical path algorithm that provides the longest signal propagation paths from inputs to outputs.

CONCLUSION

We have described the concepts and status of the development of an application-specific silicon compiler for highly complex DSP algorithms.

We have shown that development of a silicon compiler is only possible after carefully limiting the target silicon architecture to a restricted application area. Only then can a CAD system be developed. Most parts of the synthesis and module generation system have been successfully prototyped, and the first large chip designs are now being undertaken. The use of the AI programming techniques is becoming more and more important in this activity.

Behavioral silicon compilation as well as the module generators to

support it will require new skills from both system and silicon designers. It remains to be seen how they will react to it in the future.

ACKNOWLEDGMENTS

We recognize contributions from many people, but most of all from all the members of the IMEC VSDM group, particularly K. Croes, L. Rijnders, I. Vandeweerd, M. Pauwels, F. Catthoor, M. Bartholomeus, G. Goossens, J. Vanhoof, E. Vanden Meersch, and I. Bolsens.

Like Cathedral-I, this work was sponsored by the Economic Community's Esprit 97 project, which involves contributions from IMEC, Philips, Siemens AG, Bell Telephone Manufacturing, Silvar-Lisco, and Ruhr University of Bochum.

We recognize contributions from all our Esprit 97 partners, especially Philips, Siemens, Bell, and Silvar-Lisco, particularly J. Van Meerbergen of Philips who greatly influenced Cathedral-II's target architecture.

REFERENCES

1. R. Sluyter, H. Kotmans, and A. Van Leeuwaarden, "A Novel Method for Pitch Extraction from Speech and a Hardware Model Applicable to Vocoder Systems," *Proc. IEEE ICASSP Conf.*, Apr. 1980, pp.45-48.
2. P. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing," *Proc. IEEE CICC Conf.*, May 1985, pp.213-216.
3. E. Marien et al., *Manual of LOGMOS V4.2: A Simulator Covering Register Transfer-Functional-Gate and Switched Level*, available from Silvar-Lisco, Kapeldreef 75, B-3030 heverlee, Belgium.
4. D. Thomas et al., "Automatic Data Path Synthesis," *Computer*, Vol. 16, No. 12, Dec. 1983, pp.59-70.
5. P Marwedel, "The MIMOLA Design System: Tools for the Design of Digital Processors," *Proc. Design Automation Conf.*, 1984, pp.587-593.
6. L. Rijnders et al., *CAMELEON Version 1.1, Users Guide*, report 5-C1-3 of EEC project MR-03-KUL, available from IMEC.
7. H. De Man et al., "DIALOG: An Expert Debugging System for MOS VLSI Design," *IEEE Trans. Computer-Aided Design*, Vol. CAD-4, No. 3, July 1985, pp. 303-311.
8. J. Ousterhout, "Switch Level Delay Models for Digital MOS VLSI," *Proc. Design Automation Conf.*, 1984, pp.542-548.
9. J. Cockx, *User's Manual for SIMMY*, KU Leuven, project MR03KUL report 5-B1-1, Oct. 1984.

ADDITIONAL READING

Catthoor, F., et al., "General Datapath, Controller and Inter-Communication Architectures for the Creation of a Dedicated Multi-Processor Environment," *Proc. ISCAS-86*, Vol. 2, May 1986, pp.730-732.

Catthoor, F., et al., "Investigation of Finite Word-Length Effects on Arbitrary Digital Filters Using Simulated Annealing," *Proc. ISCAS-86*, Vol. 3, May 1986, pp.1296-1297.

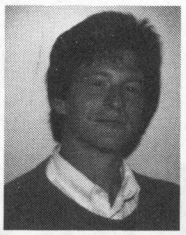
De Man, H., et al., "CATHEDRAL-II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip," NATO summer course on logic synthesis and silicon compilation for VLSI design, July 1986 (to be published).

Fettweiss, A., "Wave Digital Filters: Theory and Practice," *Proc. IEEE*, Vol. 74, No. 2, Feb. 1986, pp.270-327.

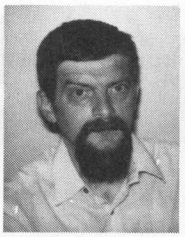
Goossens, G., et al., "A Computer-Aided Design Methodology for Mapping DSP-Algorithms Onto Custom Multiprocessor Architectures," *Proc. ISCAS-86*, Vol. 3, May 1986, pp.924-926.

Jain, R., et al., "Custom Design of a VLSI PLM-FDM Transmultiplexer from System Specifications to Layout Using a CAD System," *IEEE J Solid-State Circuits*, Vol. SC-21, No. 1, Feb. 1986, pp.73-85.

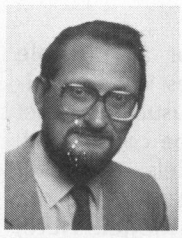
Van Ginderdeuren, J., et al., "A Digital Audio Filter Using Semi-Automated Design," Digest of technical papers of ISSCC conference, Feb. 1986, pp.88-89.



Jan Rabaey heads the Architectural and Algorithmic Strategies group of IMEC's Design Methodologies for VLSI Systems Division. His research interests are computer-aided analysis and automated design of digital signal processing circuits and architectural synthesis. Previously, he was a visiting research engineer at the University of California at Berkeley, where he developed an automated synthesis system for multiprocessor DSP architecture. Rabaey holds a PhD in applied sciences from Katholieke Universiteit Leuven.

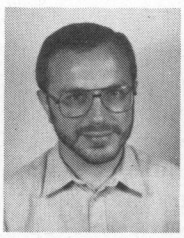


P. Six heads the Silicon Implementation Techniques group at IMEC, where his interests are symbolic layout and compaction, module generators for data paths and controllers, and graphical user interfaces. Previously, he headed the CAD group at Bell Telephone Manufacturing in Antwerp, where he was responsible for the development and integration of CAD tools for design and verification of ICs. Six received a PhD in applied sciences from Katholieke Universiteit Leuven.



Hugo De Man is vice president of the VLSI Systems Design group at IMEC. His research interests are IC design and CAD. Previously, he was an ESRO-NASA research fellow working on computer-aided device and circuit design, a research associate of the Belgian National Science Foundation, a professor at the University of Leuven, and a visiting professor at the University of California at Berkeley.

De Man holds a PhD in applied sciences from the Katholieke Universiteit Leuven. He has been an associate editor for *IEEE Journal of Solid-State Circuits* and *IEEE Transactions on CAD*. He is a fellow of the IEEE.



Luc J.M. Claesen heads research in the Design Management and Verification group within IMEC's Design Methodologies for VLSI Division. His interests are CAD and integrated digital signal processing circuits. Previously, he was a research assistant at the ESAT Laboratory at Katholieke Universiteit Leuven, where he worked in CAD of integrated systems for digital and analog signal processing. Claesen holds a PhD in applied science from Katholieke Universiteit.

Questions about this article can be directed to H. De Man, IMEC, Kapeldreef 75, B-3030, Belgium.