

CATHEDRAL-II — a computer-aided synthesis system for digital signal processing VLSI systems

by Prof. H. De Man,

Katholieke Universiteit Leuven

Prof. J. Rabaey

University of California, Berkeley

and J. Vanhoof, G. Goossens, P. Six and L. Claesen

IMEC

This paper describes the concepts and the status of the CATHEDRAL-II silicon compiler for digital signal processing systems. It is shown that efficient layout synthesis is possible, starting from a very high-level behavioural description of a system, owing to the careful definition of a target architectural design style and an application domain. An overview is presented of the different synthesis tools which have been or are being developed, both for architectural synthesis and optimisation, and for module and layout generation. With the underlying design methodology, the world of silicon design will become accessible to system engineers.

Introduction

This paper reports the status of the work at IMEC on a computer-aided synthesis system for synchronous multiprocessor system chips realising digital signal processing applications. This system is referred to as CATHEDRAL-II.

The system supports chip layout synthesis starting from a behavioural description at the algorithmic level, and as such differs from most existing so-called 'silicon compilers', which only support module generation, followed by placement and routing based on a structural description. The work is part of ESPRIT projects 97 and 1058, which involve contributions from Philips, Siemens, Bell Telephone Manufacturing Company, Silvar-Lisco and the Ruhr University of Bochum.

The work is based on our philosophy that 'the' silicon 'compiler' simply does not and probably never will exist, just as 'the' software compiler does not exist, since there

are many source and target languages, each of which are optimised for a given task. Therefore we believe that efficient silicon compilers will *necessarily* be strongly tied to a *particular target application area*.

Moreover, since silicon design represents a delicate trade-off between time, area and power, we propose in this paper not a pure compilation system, but one in which the user has a high degree of interaction. Furthermore, adaptation to changes in architecture is possible by means of rule-based programming techniques. We call such a system a computer-aided synthesis system rather than a silicon compiler.

CATHEDRAL-II is therefore an application-specific synthesis system. It is our experience that the development of such a system needs to follow a well defined sequence of steps, which we identify as follows:

- Define a wide, but concise, class of system design applications.

- Define, based on manual design exercises, a target silicon architecture and its associated layout style.
- Define a design strategy based on available designer skills.
- Define the behavioural language and the silicon modules.
- Only then develop the 'CAD' tools, with emphasis on 'D' and 'A'.

The organisation of this paper is as follows. In the following section we define the application field of CATHEDRAL-II, which is highly complex digital signal processing (DSP) algorithms in the audio frequency spectrum. The next section then defines the target architecture for such algorithms based on a practical example. A chip will consist of a number of concurrent processors, input/output modules and interprocessor communication by data storage elements.

The paper then introduces a design method called 'meet-in-the-middle', which encourages a total separation between system design and re-usable silicon design. Based on that, we then describe the actual status of the CATHEDRAL-II computer-aided design (CAD) system. The system design starts from a high-level behavioural language called SILAGE that is oriented towards DSP. It is supported by a 'compiled-code' simulator. From this SILAGE description first a rule-based synthesis is made of the data path of the processors. A heuristic scheduling program then generates the microcode for the processor controllers and interprocessor communication network. The layout of the processors is

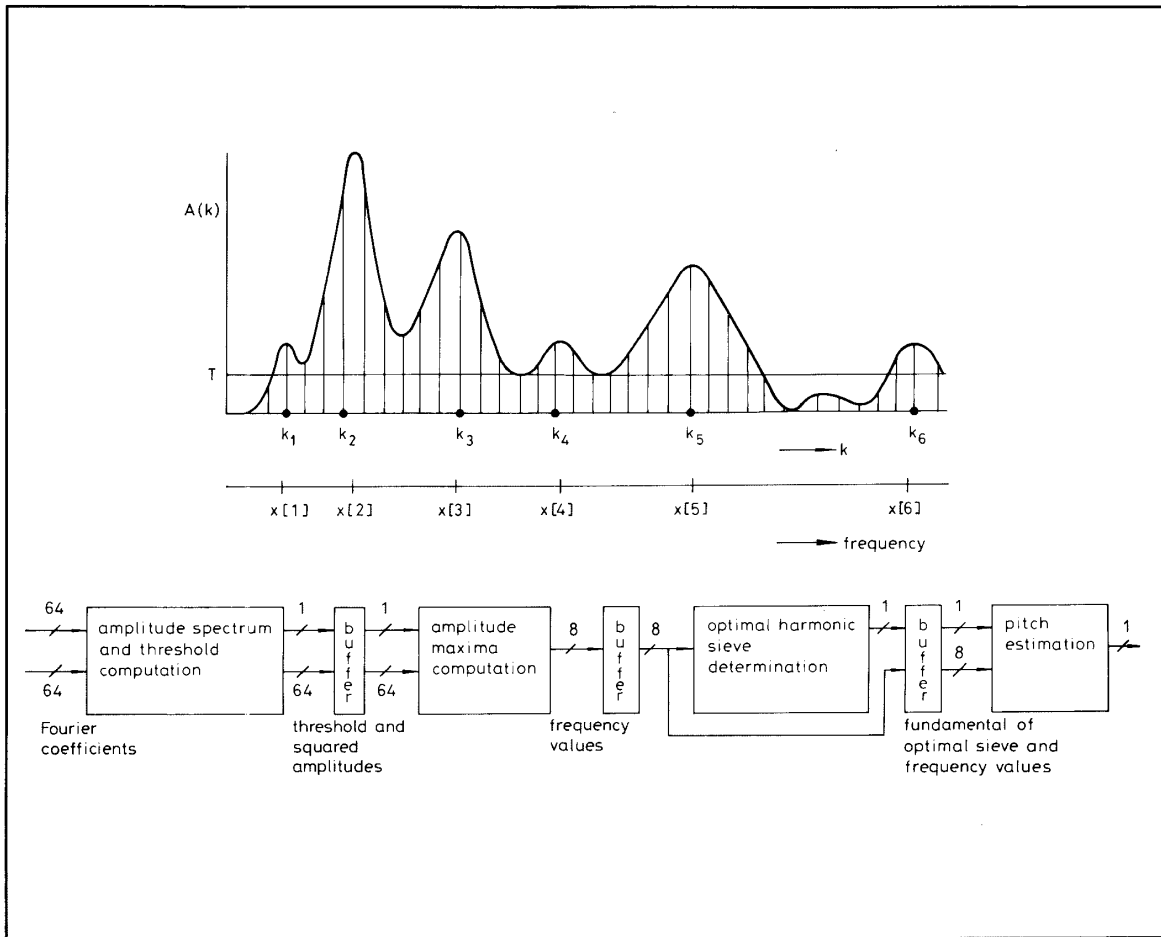


Fig. 1 Typical example of a third-generation DSP algorithm — pitch extraction for speech analysis

Note the block structure of the algorithm, consisting of independent processes which will each be assigned to a dedicated processor

synthesised in terms of modules called from automated re-usable module generators created by silicon designers and adaptable to new layout rules. Chip layout is done on a floorplanner.

The application field of CATHEDRAL-II

CATHEDRAL-II is oriented towards digital signal processing. This field is playing an increasing role in modern very large-scale integration (VLSI) based systems but in itself is still very wide, spanning data throughput from 1kbit/s up to 100 Mbit/s. Therefore even this area is too wide for one single design style or for one 'silicon compiler'.

The low-throughput end is well served by the new generations of general-purpose DSP processor chips. However, whereas the first implementable algorithms addressed such simple cases as digital filtering, today there is a need to address many applications in the audio sample range up to a 1Mbit/s sampling rate, which require either 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, robotics etc. Furthermore, these applications would be a lot more attractive if they also included adaptable input/output periphery on the same chip. The general-purpose DSP processors are not very well suited for the implementation of these algorithms. On the other hand a full-custom solution is far too expensive in design time, and, most importantly, the time to market for these usually highly competitive applications is too large. As a result of the highly specialised nature of such algorithms, we start from the conjecture that the algorithm designer must be able to do the silicon implementation him/herself. Therefore CATHEDRAL-II addresses *highly complex, block-oriented DSP algorithms* in the *audio to near-video-frequency range* as the application field.

As a typical example, Fig. 1 shows a pitch extraction algorithm for speech, as described in Ref. 1. This example among others has been taken as a vehicle to study the design process and to define the tools in CATHEDRAL-II. Real and imaginary parts

of blocks of 64 frequency components resulting 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 (decision making!), and these are then compared to 40 'sieves' with meshes at octave distances. Finally, the best match is computed as the pitch value. As Fig. 1 shows, this problem, typical for third-generation DSP algorithms, naturally decomposes into a number of subprocesses which are fairly independent of each other.

This situation is typical and is related to the fact that in most algorithms communication bottlenecks occur owing to an accumulation of sequentially generated data necessary to create new samples for the next subprocess. As shown in Fig. 1 this gives rise to the need for a data storage element between the two subprocesses. Based on a careful study of these effects we have therefore defined a target architecture in which each of such subprocesses is assigned to a dedicated processor and the interprocessor communication, in its most

general form, is taken care of by switched RAMs. This architecture will be discussed in more detail in the next section, together with the description language SILAGE (Ref. 2), used for the high-level system definition from which the multiprocessor structure will be synthesised.

The target architecture

In most present-day synthesis tools the target of the compilation is rather ill defined. In order to generate a high-performance, area-efficient DSP chip, it is obviously insufficient to consider only hard-wired or even folded data paths containing registers, busses, multiplexers and arithmetic logic units (ALUs) (Refs. 3 and 4).

The key to efficient silicon compilation is the selection and precise definition of a specialised target architecture and its parameters (Ref. 5). This should be combined with a highly specialised rule base to translate high-level language constructs into efficient register transfers to be executed on that architecture. Only in this way can higher-order constructs such as delay management, memory addressing and multiplication strategies be translated efficiently, and dedicated optimisation strategies be applied. Architecture-independent generation of structure cannot exploit the particular properties of a target library and is therefore bound to be sub-optimal in most cases. In order to cope with various levels of technological updates, a variety of mechanisms have been provided within CATHEDRAL-II. Updates to rather closely related (for example scaled) technologies can be handled by the module generators. Updates to completely new libraries are handled by writing a completely new translation rule base. This is only feasible if a powerful expert system shell is available. Updates to new design methodologies inevitably require a completely different system and different optimisation tools.

A careful study of many DSP applications, conducted in co-operation with the industrial partners in the ESPRIT projects, shows that DSP algorithms can be considered as structured computer programs to be executed in real time. They can be described by a set of rather independent but concurrent 'functions' with strictly local variables. The global input/output variables are exchanged between the functions by data storage elements which perform the necessary data shuffling and data rate transformations between two functions. This consideration has led to the definition of the target architecture as shown in Figs. 2 and 3.

At the highest level, the proposed architecture is composed of a set of concurrently operating processors. Each of those executes one particular subtask of the algorithm and is optimally tuned to perform

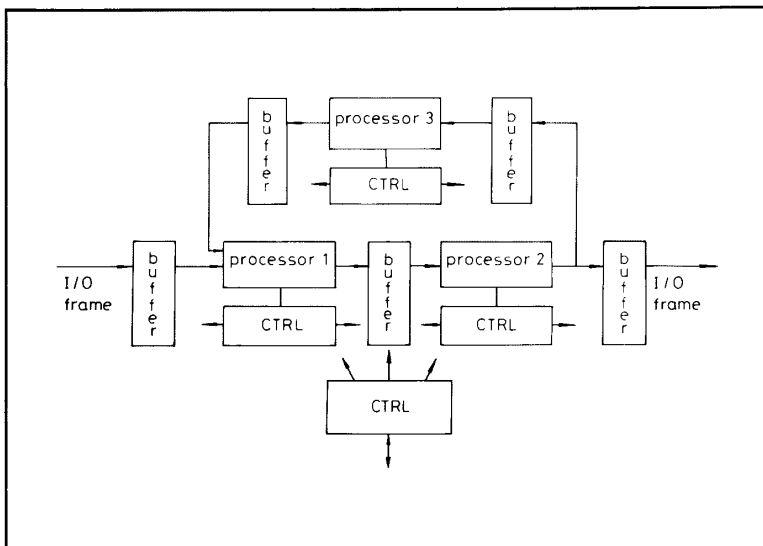


Fig. 2 Target multiprocessor architecture

Processors are locally dedicated data paths communicating global variables through data stream buffers

just that one task (Fig. 2). Each of those processors operates relatively independently from protocols, exchanging only data which is global between them. Dependent upon the data exchange rate and the amount of buffering needed, different protocols can be selected: synchronously switched RAM buffers, first-in, first-out (FIFO) buffers or request and acknowledge based synchronisation. The communication with the outside world proceeds over an input/output frame, which can support a large range of input/output protocols (ranging from parallel to serial, from synchronous to asynchronous).

One level deeper, as shown in Fig. 3, each of the processors consists of a dedicated data path and a controller. The data path is optimised for (only) the particular tasks it has to perform, and is assembled from a set of selected *execution units* (EXUs), interconnected by a restricted number of customised busses. Each of the EXUs contains two register files with up to a maximum of eight registers at its input side. This composition 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 data path — an ALU shifter unit
- dedicated accelerators — an array multiplier (with or without adding/subtracting accumulator), an iterative divider, a comparator unit for fast decision-making algorithms, a normaliser unit for accelerating a limited set of floating-point operations
- memory — ROM, RAM, FIFO and an address calculation unit (ACU).

The modules are stored in a procedural way in module generators (Refs. 6 and 7) with a well defined set of parameters (word length, number of overflow bits, optional presence of a shifter, ranges of programmable and fixed shifters, number of pipeline stages). Each of the EXUs consists of two input register files (of at most eight register fields) and some combinatorial logic to generate the desired function. The registers provide local storage of variables.

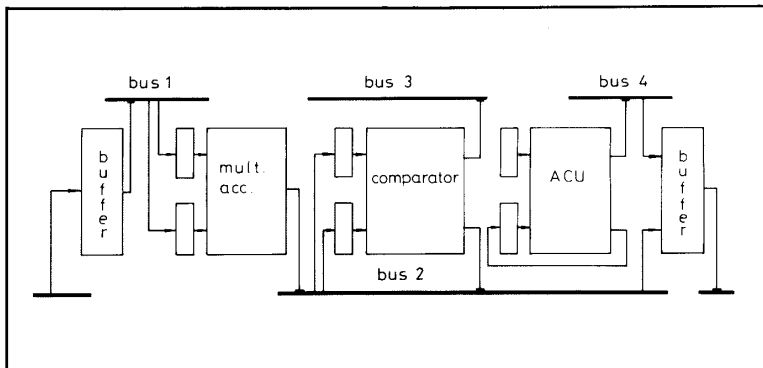


Fig.3 Example of a dedicated data path constructed from a restricted set of parametrisable EXUs — ALU, ACU, multiplier/accumulator, normaliser, divider and comparator

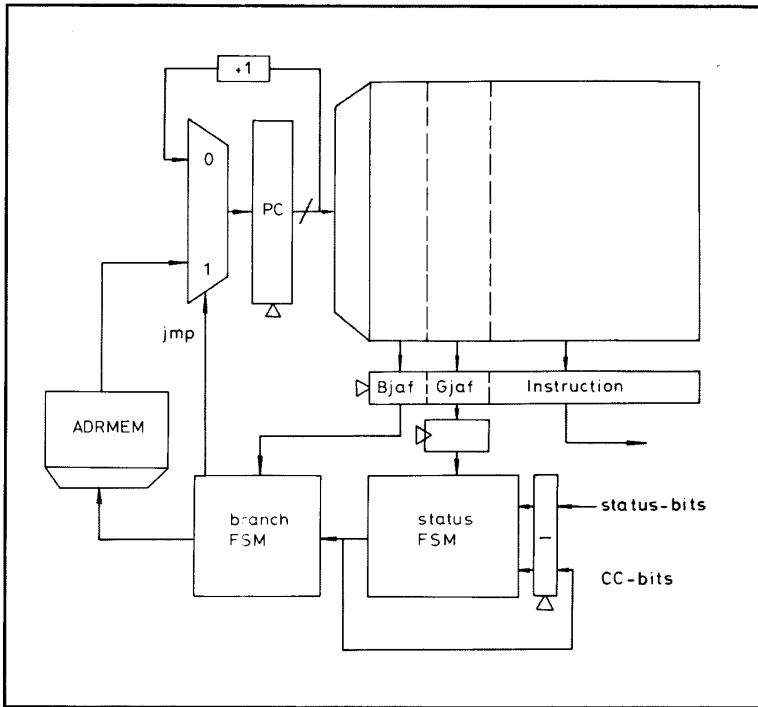


Fig. 4 Multibranch controller architecture for the data path

RAM is used as background memory.

Scan-path registers and self-test structures for embedded RAMs are included in the modules. In the module generation environment a test view is included, which allows the generation of all test vectors as a function of the parameters of the module.

The maximum clock frequency depends on the library that is being used (10MHz

for the current 3 μ m CMOS library). It is assumed that the module generators supply modules which can be clocked at this given frequency. The module generators will insert pipeline stages if the internal logic causes too much delay (for example because the word length is too big) and they will select faster logic structures (for example a carry-look-ahead instead of a

carry-ripple adder) whenever appropriate. As a consequence, the synthesis tool only considers machine cycles and does not have to consider circuit delays of the EXU logic. However, it has to know the total number of machine cycles needed for an operation on a certain EXU.

A multi-branch, microcode-based controller, as shown in Fig. 4, has been selected to control the data flow through the data path. This structure is flexible and powerful enough to handle a large span of algorithms in a flexible and efficient way. It can support heavily decision-making-oriented as well as regular, repetitive algorithms. Some of the EXUs can also have a local controller. This helps to reduce the complexity and the size of the central controller. An example of such a controller is the decoder of the register files. Note also that another controller is needed at the highest level to control the data flow between the processors and to the outside world.

An example of a processor data path constructed using the proposed strategy is shown in Fig. 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 an update in the maximum calculation in two cycles (average).

The 'meet-in-the-middle' design strategy

The complexity of the algorithms to be implemented far exceeds the circuit or even the logic gate level. Moreover, since silicon designers are so scarce, we can not exploit the potential of such systems if they cannot be used directly by system engineers having no detailed knowledge of the silicon implementation.

It is therefore necessary that silicon design knowledge (at the micrometre level) be localised in re-usable modules at the MSI/LSI level familiar to the system designer. This leads to the design scheme shown in Fig. 5, which we have defined as the *meet-in-the-middle* design methodology. This methodology can be characterised as follows:

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

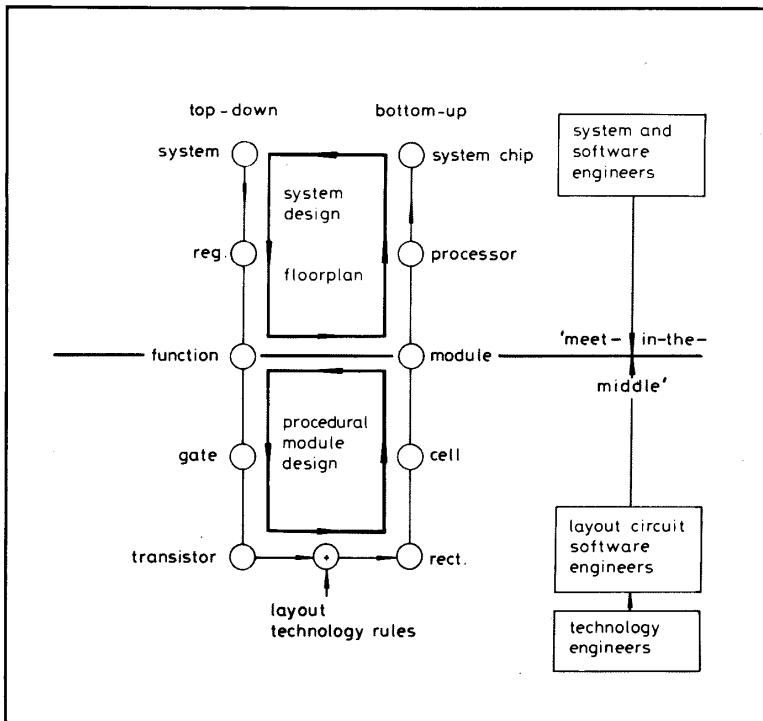


Fig. 5 Meet-in-the-middle design methodology supported by CATHEDRAL-II

modules is then the chip design.

- Silicon modules are *re-usable*, just as standard cells are. 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.

- However, 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 capable of surviving a number of technology updates. Therefore modules must be technology adaptable.

- Silicon modules, however, are not the privilege of 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 the implementation of it, we expect that *module design will require a powerful design environment in itself* for a local team of silicon designers. This may not yet be the case today, but we expect this to happen in the future.

Note that, in this design style, system designers design in a top-down fashion down to their usual intermediate level. The silicon people design in their usual bottom-up fashion, composing the LSI level modules from functional building blocks (see below), which in turn are composed of logic leaf cells at transistor level. It is as if both parties meet each other in the *middle* of the design abstraction levels. In this way scarce talent is optimally used, and the design process corresponds to the usual patterns.

However, owing to the above characteristics, some fundamental deviations from classical design at silicon and system level do occur. They are related to the fact that module generators will be software procedures rather than fixed geometrical structures, and also the fact that we expect system designers to think at the algorithmic rather than the structural level. This will become clearer after giving next the outline of the CAD toolbox as we are developing it for CATHEDRAL-II.

The CATHEDRAL-II CAD system

Outline of the system

Fig. 6 gives an outline of the CATHEDRAL-II CAD system. System design is first described at the behavioural level in the SILAGE language (Ref. 2). This is a very high-level language especially oriented towards DSP system specification, as will be discussed below. It is at this high level that we also have a very efficient simulator in order to verify the correctness of the proposed algorithm to be implemented.

Based on this system description we describe below the actual synthesis part of the CAD system. The synthesis system will translate the behavioural description into

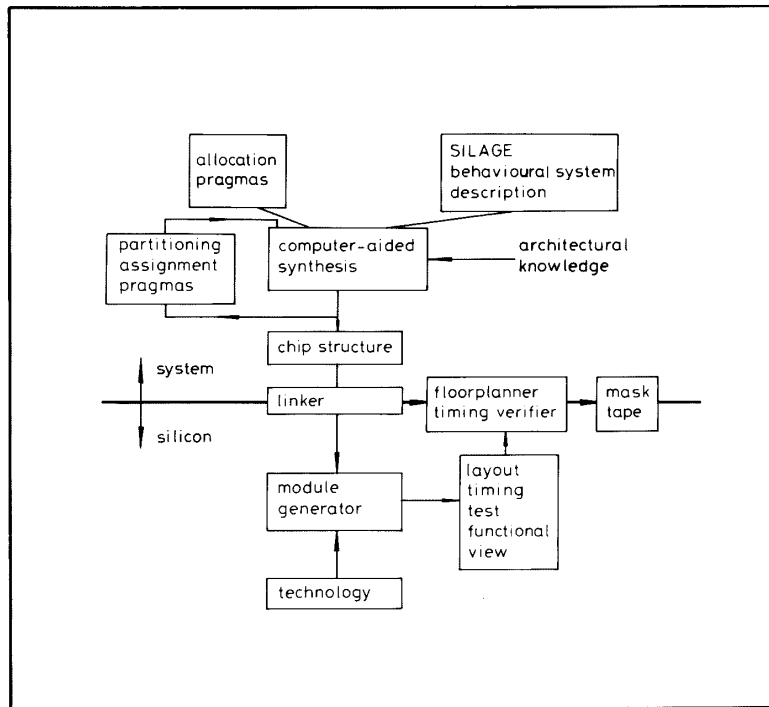


Fig. 6 Outline of the CATHEDRAL-II CAD system

the hardware structure of the chip. The latter is defined in terms of the EXUs, as discussed in the section on the target architecture. It is this part that forms the cornerstone of our system. We believe that an efficient translation can only be done if it takes the properties of the target architecture into account. Furthermore, a strong interaction with the designer must be possible. Therefore we have not opted for a fixed type of compiler, but rather for an interactive system based on a rule-based Prolog translator to synthesise the structure of the data paths of the processors. As we will see, the user can influence the compilation by using so-called pragma statements. These statements do not modify the original behaviour (as this would violate the principle of design correctness 'by construction') but only the way in which the behaviour is mapped into the hardware representation.

Besides the mapping into the data path, a lot of procedural optimisation also has to take place, such as scheduling of microcode, register count minimisation and bus merging. This is done using C programs called from the Prolog environment. The synthesis part then generates the hardware structure in terms of instantiated versions of the EXUs and the controllers. The call for instantiation of the EXUs, controller ROMs and programmable logic arrays (PLAs) as well as the communication RAMs and FIFOs is passed through a procedural linker to the module generators. As will be discussed later, these are software procedures capable of generating the necessary views to build the final chip

layout, its characterisation test and documentation. These views are sent to a floorplanner. This floorplanner allows for interactive placement of the modules and automated routing based on the netlist produced by the synthesis system.

Finally, we will explain how the module generators can be adapted to a new technology file by the use of symbolic layout. Also new techniques to verify correctness during the design of new module generators are provided. Note also on Fig. 6 the bold line in the 'middle', which is the boundary of the activity of the silicon designer, who provides the library to the many system designers using the CAD system for a large variety of system design jobs. We will now discuss each of the parts of the system separately.

System specification language — SILAGE

Unlike most so-called silicon compilers, which are basically only layout generators from structural description, CATHEDRAL-II starts from a behavioural specification language. The system designer, just as would a high-level language programmer, describes his/her DSP algorithm in a pure behavioural language without any structural biases. Other behavioural compilers (Refs. 4 and 8) use a procedural language (Pascal or C-like). In that case the control flow is implicit to the code and, in case a fixed microprocessor architecture is used, the compiler is similar to a traditional software compiler. However, in such cases the potential throughput of silicon is not exploited. The other extreme is to use a

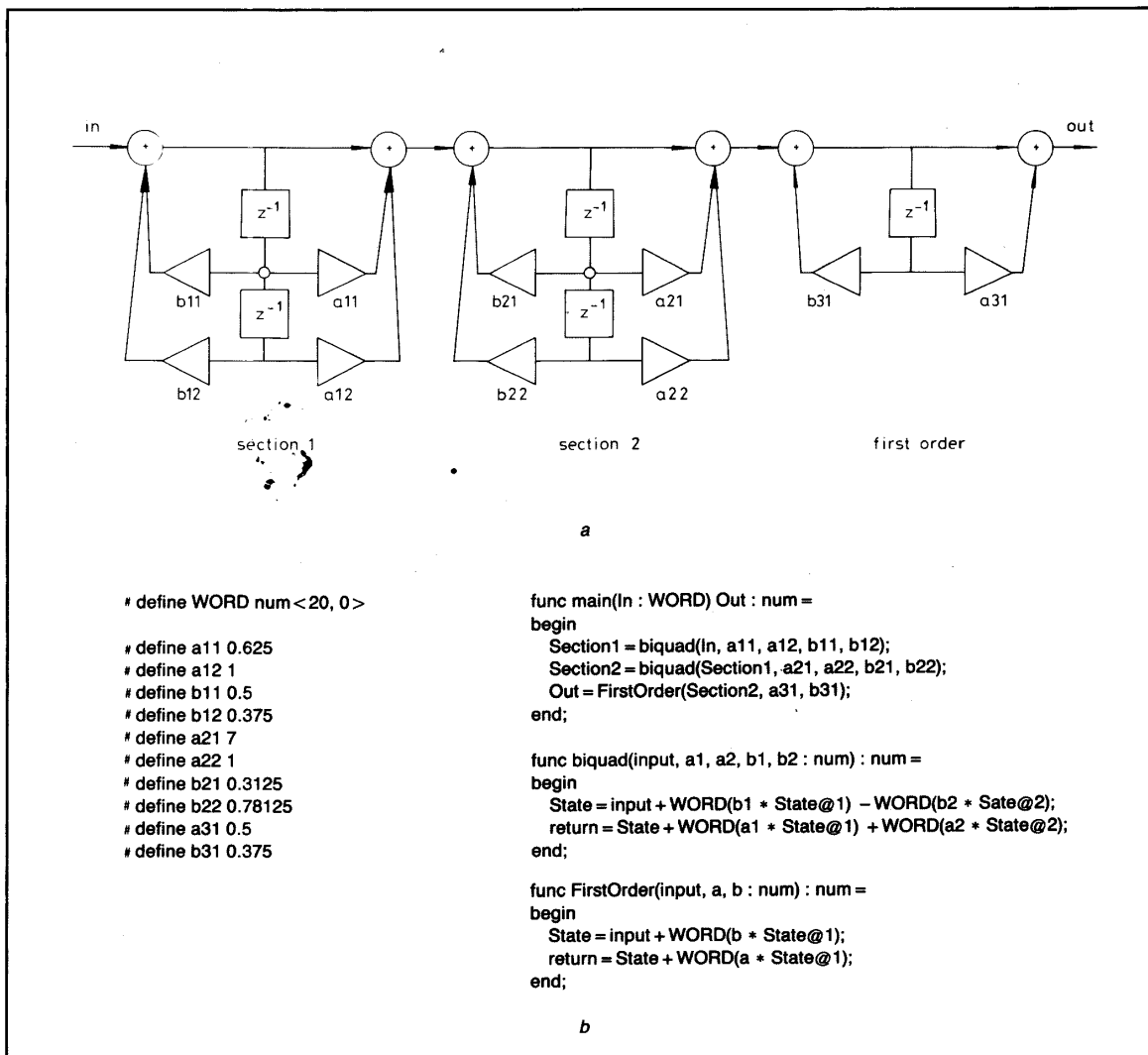


Fig. 7 SILAGE description example

a Signal flow graph of a fifth-order PCM filter

b SILAGE description of the fifth-order PCM filter example — note the applicative nature of SILAGE as well as the finite word length types

register-transfer level description where, by definition, no control flow is given and the maximum concurrency is given in the language. In such cases the implementation efficiency is very heavily influenced by the way the behaviour is described.

In contrast to these two approaches, we use a description which is applicative and therefore purely behavioural, and we add programs for interactive mapping into hardware. Therefore the input language to CATHEDRAL-II has the following two separate parts:

- For describing the behaviour of the algorithm, the applicative language SILAGE (Ref. 2) is used. The main idea of SILAGE is to capture the signal flow graph of a signal processing algorithm, such as illustrated by the fifth-order PCM filter example shown in Fig. 7a. Fig. 7b shows the SILAGE description of the PCM filter. A SILAGE

description contains no structural information and does not enforce any degree of concurrency, nor does it contain information about the implementation of the control flow. This gives the compiler the full flexibility to select between the fastest and the cheapest implementation, starting from the same behavioural input description. We consider this to be of prime importance, because of the excessive CPU times needed to simulate the behaviour of complex DSP algorithms. In SILAGE, signals are represented by variables, which can be thought of as infinite arrays in time. The implicit time index is never written. The values of a variable at any moment in the past can be recovered by using a delay operator '@'; so instead of writing $a[t]$ and $a[t-2]$ one writes a and $a@2$.

Relations between variables are expressed by explicit time-discrete equations. The ordering of these equations is irrelevant.

Each equation has to be considered true at any moment in time. Equations can be conditional. Loops can be used as a compact notational format (repetition) but they do not imply any control flow. Between the equations inside a loop, there might be precedence or not. A loop

```

(i : 1 .. N) ::
begin
  .....
end;

```

is equivalent to the quantifier $\forall i: 1 < i < N$. Operations on variables are represented by function calls. A large number of functions suited for describing typical DSP operations are predefined. Word lengths of signals and finite word length characteristics of operators can be fully specified (at this moment for fixed-point operations only). Typing is almost completely implicit, but

the designer can change the type of any variable by using coercion.

It is to be noted that in CATHEDRAL-II a 'compiled-code' simulator is available to simulate the SILAGE description. This simulator generates C code, which can be compiled and executed. Once a SILAGE description is checked for correctness, no further lower-level simulation is necessary, in principle, owing to the use of the interactive synthesis tools.

- The second part of the input description is made up of *compiler directives*, also called '*pragmas*'. Since SILAGE is applicative, the compiler must generate both a structure and the detailed timing of the micro-program. As discussed before, it will only be able to do so by using a very well defined target architecture. This allows both the compiler and the designer to make abstract decisions on a very high level and still perform a relevant cost evaluation, because bottom-up design information is continuously used when making high-level decisions. In the *interactive* concept of CATHEDRAL-II the system designer is able to give *structural hints* to the compiler at as high a level as possible. This can be done by adding high-level directives, called '*pragmas*', to the behavioural description.

For our target architecture, three types of pragmas are supported:

- pragmas for splitting up an algorithm into processors — pragma

'pole, processor, 2'

forces the function 'pole' to be implemented on processor 2

- hardware allocation pragmas — pragma

'alloc(alu, 3)'

allocates three arithmetic logic units (ALUs) in the data path

- pragmas for assigning an expression or a class of expressions to an execution unit instance — pragma

'assign(pole(a, _).(_ * y), alu, 2)'

forces any multiplication using the local variable 'y' in any function call 'pole' with first argument 'a' to be executed on ALU instance 2. Note that the underscore '_' is used as a wild character.

The synthesis system — mapping and optimisation

Based on the SILAGE specification we are now faced with the task of generating the data path and the controller of the processors as well as their intercommunication network. We refer to Fig. 8 to illustrate the following different tasks:

- *Data path synthesis*: The synthesiser, JACK-THE-MAPPER, deduces a data path structure, consisting of a set of EXUs, from the SILAGE description. The methodology

adopted in JACK is to use a mixture of automated tools and user interaction to solve this extremely complex optimisation and search problem. In fact, our experience has shown that a system designer often has a good insight into the complexity and the computational bottlenecks of an algorithm. Therefore he/she is perfectly capable of estimating the required amount of parallelism and the acceleration units needed. The most-time consuming and error-prone job is not located in the allocation task, but in the *operator assignment, the controller generation and the minimisation of the execution time, the register usage and the bus count*. This is where the synthesis task has to come in and has to be extremely good in order to be acceptable.

Based on these considerations, we have deduced the following synthesis strategy (Fig. 8):

- The SILAGE description of the algorithm is first passed to a *preprocessor*. The tasks of the preprocessor are to parse the SILAGE description, to perform syntax and semantic checks, to determine the data types of all signals and to perform a number of local transformations, common to most general-purpose software compilers (such as, for example, the elimination of common subexpressions). In addition to the behavioural part of the SILAGE description, the preprocessor also inputs a set of user-defined *allocation and assignment pragmas*

(which will steer the compilation process).

- In order to transform the preprocessed applicative high-level language description of the algorithm into a customised processor structure, the mapping tool has to *assign primitive SILAGE operations to execution units*, define the bus structure and assign the SILAGE and intermediate variables to register files and background memories. This task can be divided into a translation and a number of optimisation subtasks.

The *translation* step transforms behavioural primitives into architectural primitives. This step is of extreme importance, since it will determine how efficiently the architectural properties can be exploited. In order to cope with architectural changes and expansions, this tool has to be flexible and expandable by an inexperienced user. Therefore it has been implemented in a rule-based fashion. This rule base captures the knowledge of the architecture designer and addresses the real creative step in the synthesis process.

The translation might be straightforward for a simple addition, but is far more complicated for constructs such as multiplication (parallel, parallel-serial, constant multiplication), algorithmic delays, matrix operations, repetitions, floating-point operations, double-prec-

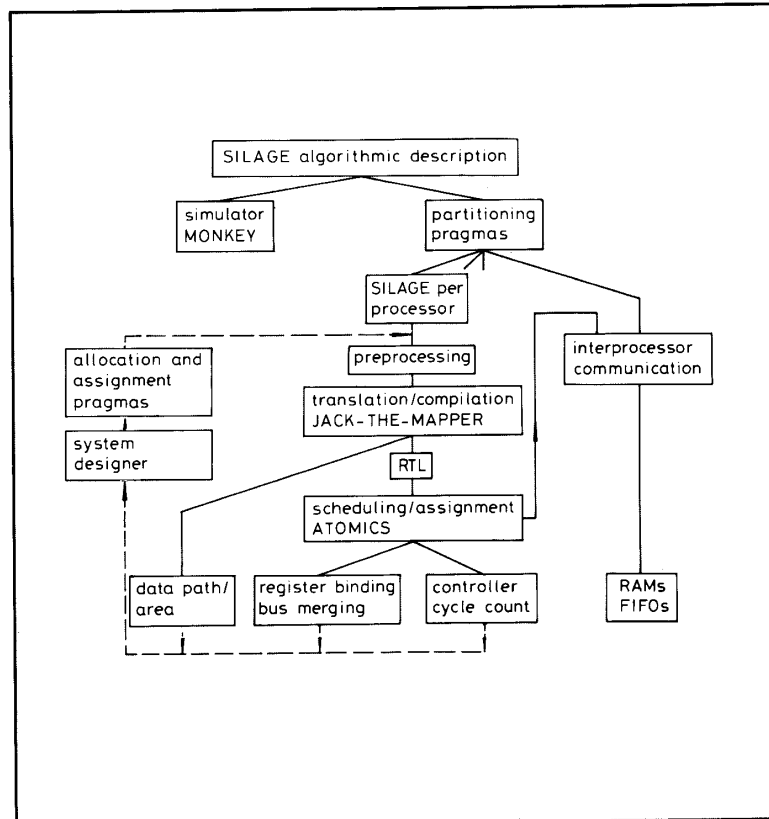


Fig. 8 The synthesis part of the CAD system of CATHEDRAL-II

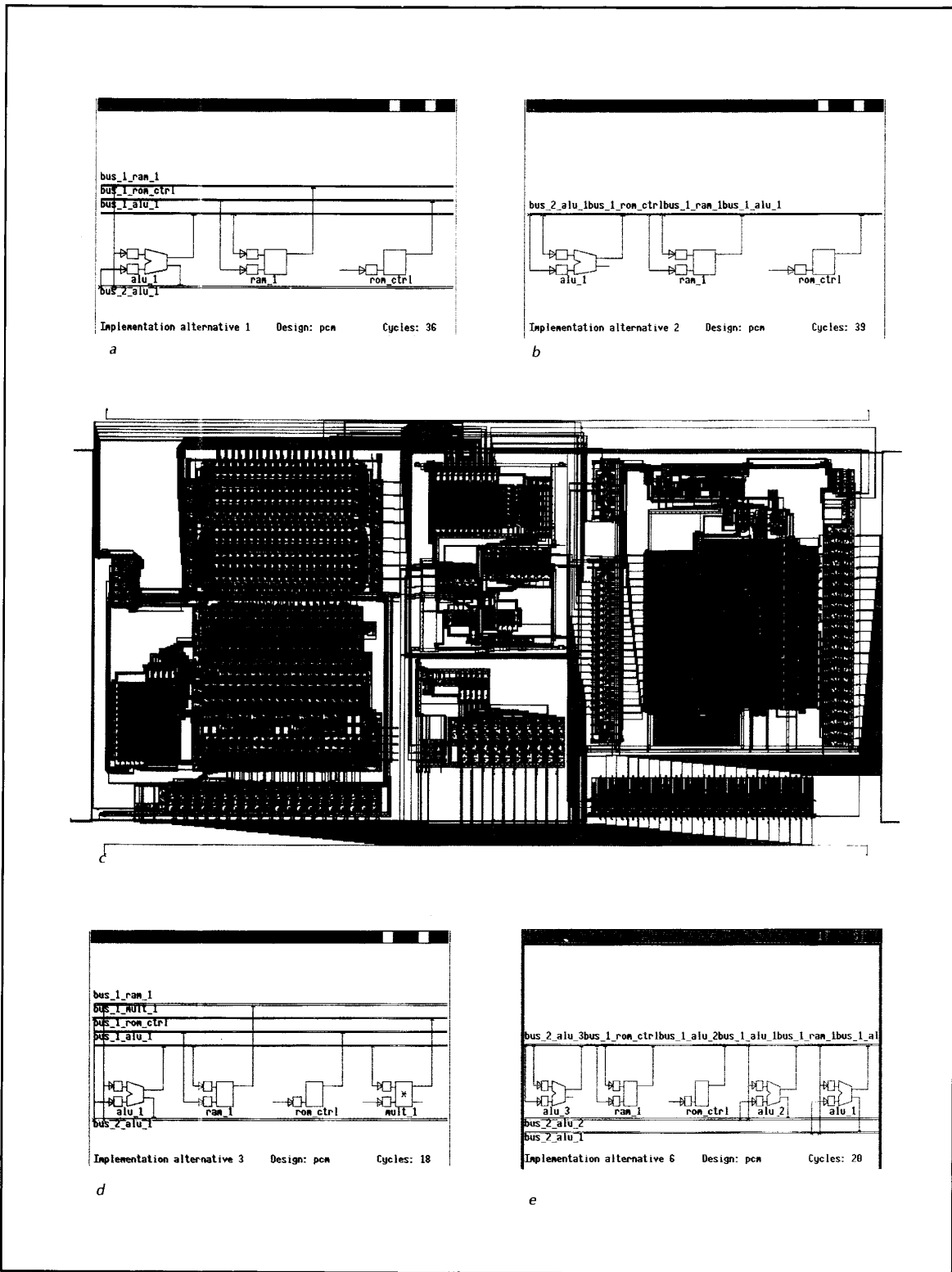


Fig. 9 Processor synthesis example

- a Default data path generated by JACK-THE-MAPPER — this solution with four busses is scheduled by ATOMICS in 36 machine cycles
- b Same data path as in Fig. 9a after bus merging — rescheduling now shows the need for three more cycles (Fig. 10 shows the scheduled register-transfer code)
- c Layout generated for the processor of Fig. 9b — from left to right: ALU with register files, controller circuitry, RAM
- d Redesign by JACK and ATOMICS of the processor based on the pragma 'assign (_ * -, mult, 1)', whereby all multiplications are executed on a multiplier and scheduled in 18 cycles
- e Redesign based on a re-partitioning pragma, whereby a three-ALU solution is scheduled in 20 cycles

sion arithmetic etc. A second set of rules implements the interconnection strategy. It generates the necessary busses, input multiplexers and tri-state output buffers. The current rule base for the multiprocessor architecture consists of more than 100 rules, but new ones are being added regularly (as our experience grows). A user-friendly knowledge acquisition system is being developed to ease the introduction of new rules.

- *Scheduling/assignment* (Ref. 9): As a result of the translation step, the SILAGE description has been transformed into a data path structure and a register-transfer (RT) description of the algorithm. In the RT description no timing is imposed on the operations. A number of *assignments* (for example the binding of an operation to a particular EXU instance) are *also* left open. The tasks of the scheduling operation thus are:

- the ordering of the RT operations on the time axis in such a way that the execution of the algorithm takes a minimal number of cycles
- the binding of the undefined assignments so that the allocated hardware is used in an optimal way.

A graph-based scheduling tool called ATOMICS has been developed. Special features of ATOMICS are its capability to schedule *repetitive programs* and to handle input/output *constraints*. ATOMICS has been used to schedule extensive programs and has proven to achieve the optimal schedule in most cases. Related, additional optimisation steps are:

- *Register binding/bus merging*: Once the exact timing schedule of RTs has been determined, the dimensioning of the register files as well as the assignment of the variables to precise register fields (based on a lifetime analysis of the variables) and the minimisation of the bus count (bus merging) have to be performed. Procedural optimisation functions have been developed for both tasks and have been integrated in the JACK environment.

- *Communication hardware synthesis*: After the derivation of the processor hardware and the controller timing, it is possible to synthesise the interprocessor communication hardware and to derive the structure and contents of the central controller. The tasks of this *synthesis tool*, which is currently being developed, are to select the cheapest communication protocol (FIFO or RAM based/single or double buffering), to dimension the buffer arrays and to determine the exact timing of the control signals needed. It must be mentioned that the selection of a certain protocol can result in a number of extra constraints on the processor timing or hardware, so that a re-iteration on the processor synthesis might be needed.

```

Potential 0:
x35 : reg_2_alu_1 ← x31@1 : reg_1_alu_1 | alu_1 = passa,
      bus_2_alu_1bus_1_rom_ctrlbus_1_ram_1bus_1_alu_1 = x35 <0>;
x35 : reg_1_alu_1 ← x31@1 : reg_1_alu_1 | alu_1 = passa,
      bus_2_alu_1bus_1_rom_ctrlbus_1_ram_1bus_1_alu_1 = x35 <0>;

Potential 1:
x31 : reg_2_alu_1 ← x17@1 : reg_1_alu_1 | alu_1 = passa,
      bus_2_alu_1bus_1_rom_ctrlbus_1_ram_1bus_1_alu_1 = x31 <0>;
.
.

Potential 37:
x1 : reg_1_ram_1 ← x7 : reg_2_alu_1, x3 : reg_1_alu_1 | alu_1 = add,
      bus_2_alu_1bus_1_rom_ctrlbus_1_ram_1bus_1_alu_1 = x1 <0>;

Potential 38:
x1 : reg_3_ram_1 ← x2 : reg_2_ram_1, x1 : reg_1_ram_1 | ram_1 = write <0>;

Total of 39 potentials

Total of 39 machine cycles

a

< 0> /*Startup*/
      UPD CREG[0] = sregStart, CREG[1] = CREG[0];
      JMP IF CREG[1] → 2 || !CREG[1] → IF CREG[0] → 1 || !CREG[0] → 0 FI FI

< 1> /*Potential 0*/
      CTL reg_2_alu_1 : W[0, 4], reg_1_alu_1 : R[5, 7], alu_1 : passa,
      reg_1_alu_1 : W[6, 7], reg_1_alu_1 : R[5, 7], alu_1 : passa; UPD; JMP 3;

< 2> /*Potential 1*/
      CTL reg_2_alu_1 : W[1, 4], reg_1_alu_1 : R[4, 7], alu_1 : passa; UPD; JMP 4;

< 3> /*Potential 2*/
      CTL reg_2_ram_1 : W[0, 0], rom_ctrl : const[1, 1]; UPD; JMP 5;
.
.

< 39> /*Potential 38*/
      CTL reg_2_ram_1 : R[0, 0], reg_1_ram_1 : R[0, 0], ram_1 : write; UPD; JMP 0;

b

```

Fig. 10 Excerpt from the register-transfer code and the symbolic microcode

- a Register-transfer code
- b Symbolic microcode

An example of processor synthesis

The following is an example of the synthesis of the fifth-order PCM filter in Fig. 7a, as described in the SILAGE code in Fig. 7b. When this code is sent through JACK without any pragmas, it will first produce the cheapest possible silicon solution in terms of EXUs. The data path structure generated is shown in Fig. 9a. It contains an ALU for all operations, a small RAM to store the state variables and to perform the communication with the input/output, and a block 'ROM-CTRL' which forms the link to the controller over which immediate addresses can be fetched.

Note that in this solution four busses are used and a scheduling using ATOMICS shows that 36 cycles are needed to execute the filter algorithm. This is the fastest possible execution time using three EXUs. However, when using the bus-merging algorithm, after about one minute of CPU time on an Apollo 3000 system, we get a new solution, as in Fig. 9b, where all the busses have been merged into a single bus at the expense of only three additional cycles.

Fig. 10a also shows an excerpt of the scheduled register-transfer code produced by ATOMICS. It contains all the transfers

that have to take place during the 39 cycles; the part of the code following the vertical bars is a symbolic representation of the control signals required by the transfers. Fig. 10b shows part of the finite-state machine description, or *symbolic microcode*, derived from the RTL description. This code is the input for a program that generates the ROM and PLA matrices of the controller. When the designer is satisfied with this result he/she can call the module generation environment to generate the ALU, ROM, RAM and controller instances. They are then placed and routed on the floorplanner to give the layout shown in Fig. 9c. In this way very fast turnaround design is possible. However, if, for example, the speed is not

sufficient, the designer can add pragmas to the SILAGE description and generate new solutions quickly. In Fig. 9d the pragma

```
'assign ( _ * _ , mult, 1)'
```

states that all multiplications should take place on a single multiplier. The solution now contains a multiplier and runs in only 18 cycles.

Fig. 9e shows a re-partitioning pragma, whereby the designer forces the use of an ALU for each section in the filter and for all additions and multiplications. This leads to a solution in 20 cycles. The reason for this high speed in spite of not using a multiplier is the result of the fact that JACK

contains rules to decompose fixed coefficients into a minimum number of additions and shifts, and this clearly shows the advantage of the rule base intelligence. Finally note how such a synthesis system allows a very rapid scan of the design space, since each debugged microcode redesign costs an average designer around one week instead of about five minutes in CATHEDRAL-II. In the next section we describe the module generation environment for CATHEDRAL-II.

Module generator design environment

To allow the definition of a parametrizable module generator, the *creator* of it needs a system that enables him/her to design the leaf cells and provides 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 set of nets', the algorithms to support these high-level commands must also be provided in an integrated environment.

A drawback of a lot of the existing module generators is the 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 have designed an interactive environment that gives feedback in graphics form each time a composition rule of the module generator is defined, allowing mistakes to be corrected immediately after they are made.

To describe a module we have to capture four types of information:

- structural, defining the sub-modules of a module
 - topological, defining the relative placement of the submodules
 - connectivity, defining how the submodules must be interconnected and where the input and output terminals of a module are
 - construction, defining how submodules fit together and which of the interconnections are realised by abutment or routing.
- In the first version of the module generator in CATHEDRAL-II, called MGE (Module Generator Environment), 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 co-ordinates. Fig. 11, for example, shows a copy of the screen while defining an ALU module generator. To add a column of N cells of type GENERAL (shaded in Fig. 11) at $x=8$ and $y=1,2,\dots,N$ we use the command

```
(ADD_COMP GENERAL 8 | 1 . . . N |)
```

As can be seen from Fig. 11, the cells are represented by their outline at this stage,

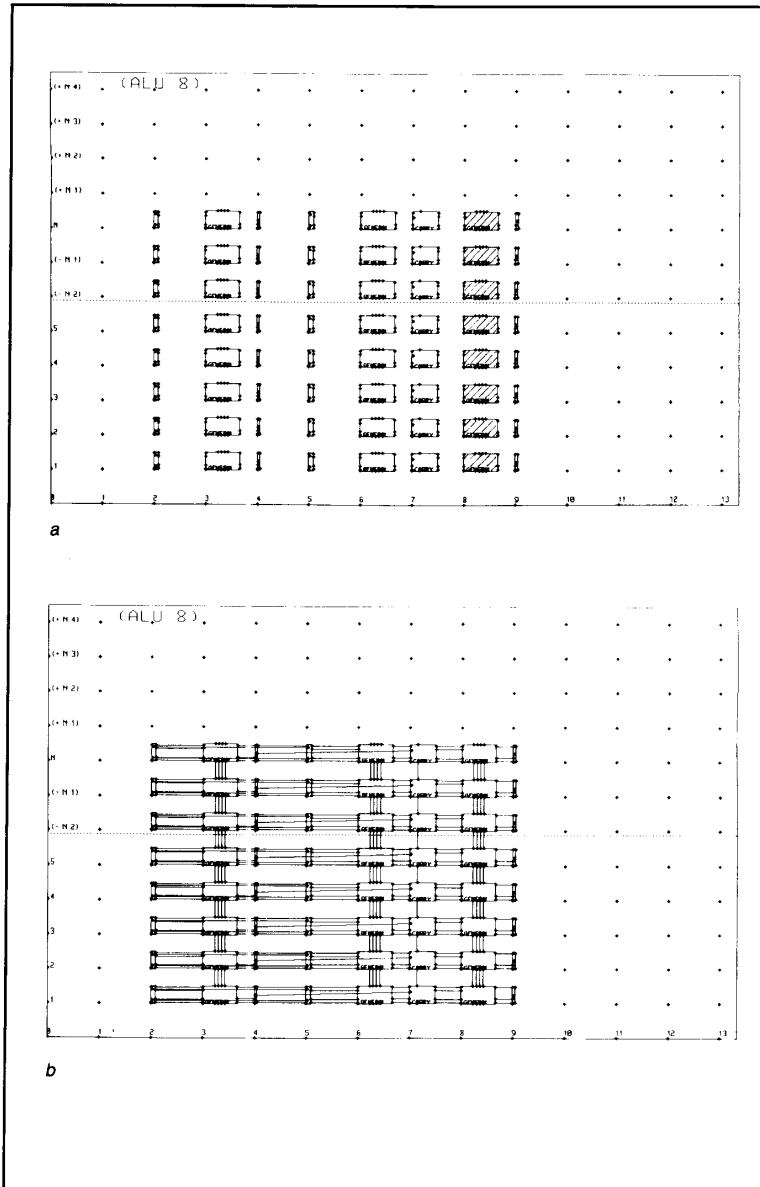


Fig. 11 ALU module generation

- a Graphical definition on the screen while defining a parametrizable ALU module generator after adding the topology of the components
- b Graphical definition on the screen while defining a parametrizable ALU module generator after adding the interconnection

resulting in a clear picture and fast display. Connections can also easily be defined by specifying a net name, and the list of terminals of the connected components. The command

```
(ADD_CON (OUT3 | 1...N |)
 (OUT | 8...8 | | 1...N |)
 (POUT | 9...9 | | 1...N |))
```

defines the N connections with net names $OUT3[1] \dots OUT3[N]$ between the pins OUT of the cells in column 8 and pins $POUT$ of the cells in column 9. Note that no assumption is made at this stage of how the connections will be realised.

Note that indices of array-like structures can be expressions, like $(N-1)$ being the last ripple signal of the adder in the ALU. MGE allows the association of an expression with each horizontal or vertical grid line on the display (see the y -axis in Fig. 11). The creator can then, by simply placing cells and pointing at grid points graphically, use the associated expressions as coordinates in the automatically generated 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 (Ref. 10). Layouts are assembled from point elements such as transistors and contacts, wires to interconnect them, and area definitions, such as the n - or p -well in CMOS technology. The layout is then compacted, using a constraint graph and a longest-path algorithm, to minimise the area of the cell. The program assures that the layout is correct with respect to the design rules, as specified in a technology file, also taking into account extra constraints that the user may have specified. Technology changes can easily be accommodated by changing the values of symbolic constants, like $MIN-POLY-WIDTH$ or $MIN-GATE-OVERHANG$, in the technology file and compacting the cell again.

The construction rules specify how the real layout of the module must be made, starting from the topology, the connections and the basic cells. Two basic options exist: either the cells fit on each other and can be 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. A river and a channel router are available. The $COMPACT$ command will move 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 abutment of cells can only be achieved by taking into account, while designing the

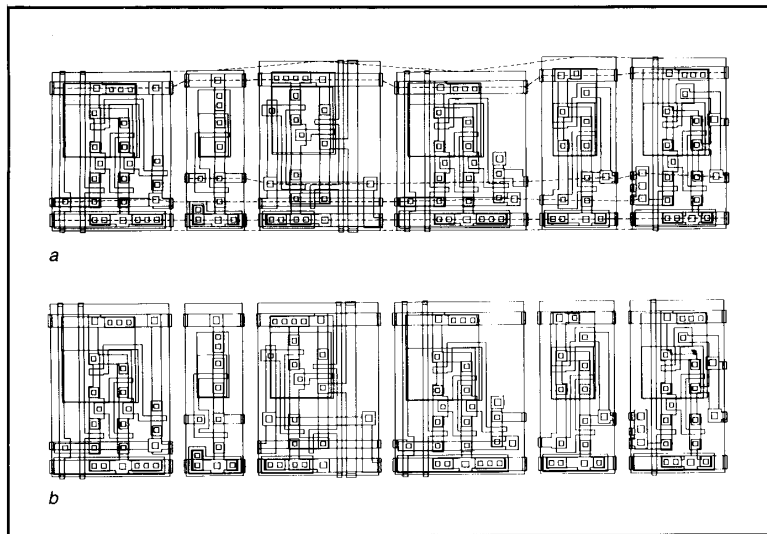


Fig. 12 Example of the automatic pitch matching of a set of data path cells

- a Original symbolic layout
- b Cells after terminal pitch matching

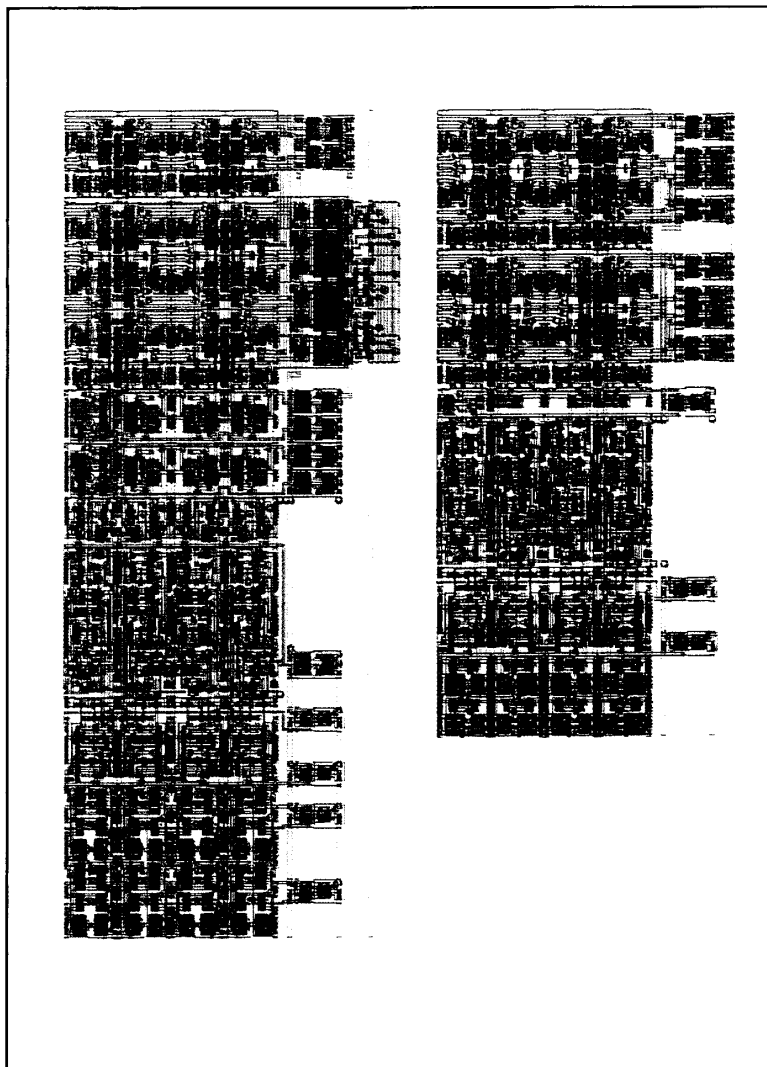


Fig. 13 Layout of two address calculating units generated by specifying two different sets of parameters to the module generator

layout of a cell, the requirement to abut to the other cells. 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 neighbours. Figs. 12a and b 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 the trouble of inventing a complete new language and taking advantage of the interpretative nature of Lisp the first version of the MGE was implemented as a superset of Lisp. For the more arithmetic-intensive algorithms such as compaction and routing, Pascal and C programs are called from Lisp. The system has been tested through the design of six EXUs defined for the target architecture in CATHEDRAL-II. The resulting layout of two instances of the ACU is shown in Fig. 13.

The use of Lisp as the basis of MGE provides the creator with all the flexibility of a high-level programming language, enabling him/her to define module generators that require more complex composition procedures. An example of this is the comparator shown in Fig. 14.

Conclusions

This paper describes 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 a careful limitation of the target silicon architecture for 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 artificial intelligence programming techniques plays an ever more important role in the prototyping activity. Behavioural silicon compilation, as well as the module generators to support it, will require new skills from both system and silicon designers and it remains to be seen how they will react to it in the future.

Acknowledgments

The authors would like to acknowledge contributions from many people: in the first place all the members of the IMEC VSDM group and in particular K. Croes, L. Rijnders, I. Vandeweerd, M. Pauwels, F. Catthoor, M. Bartholomeus, J. P. Robin, E. Vanden

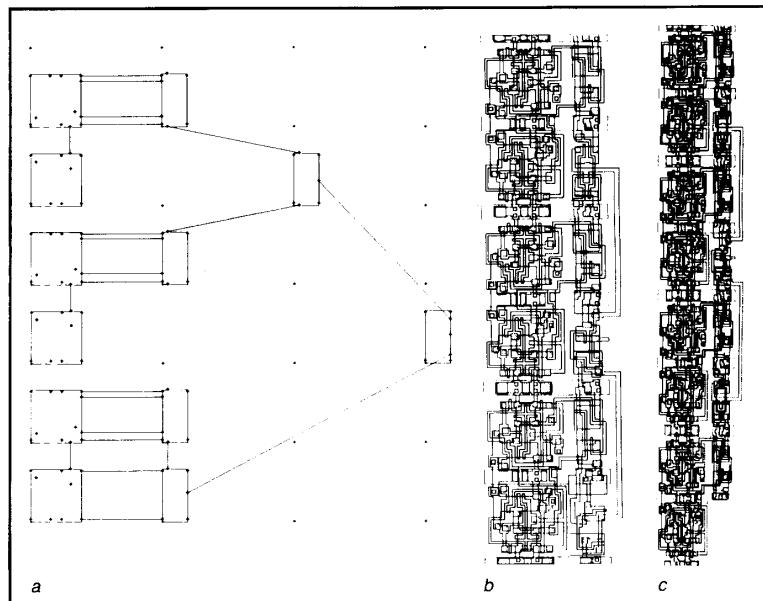


Fig. 14 Comparator module generation

- a Interpretatively generated representation of the structure of a 6 bit comparator
 b Layout of the 6 bit comparator — some cells are connected via abutment, others with routing
 c Layout of an 8 bit comparator, proving the flexibility of the module generator created

Meersch and I. Bolsens. We also wish to acknowledge contributions from all our ESPRIT 97 partners, especially Philips, Siemens, Bell Telephone Manufacturing

Company and Silvar-Lisco. In particular we are grateful for significant inputs to the target architecture from Dr. J. Van Meerbergen of Philips.

References

- 1 SLUYTER, R. J., KOTMANS, H. J., and VAN LEEUWAARDEN, A.: 'A novel method for pitch extraction from speech and a hardware model applicable to vocoder systems'. Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, Denver, CO, USA, April 1980, pp. 45-48
- 2 HILFINGER, P.: 'A high-level language and silicon compiler for digital signal processing'. Proceedings of IEEE Custom Integrated Circuits Conference, Portland, OR, USA, May 1985, pp. 213-216
- 3 KOWALSKI, T. J., GEIGER, D. J., WOLF, W. H., and FICHTNER, W.: 'The VLSI design automation assistant: from algorithms to silicon', *IEEE Design & Test of Computers*, 1985, 4, (2), pp. 33-43
- 4 TRICKEY, H.: 'Flamel: a high-level hardware compiler', *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 1987, CAD-6, (2), pp. 259-269
- 5 CATTHOOR, F., RABAEY, J., GOOSSENS, G., VAN MEERBERGEN, J. L., JAIN, R., DE MAN, H. J., and VANDEWALLE, J.: 'Architectural strategies for an application-specific synchronous multiprocessor environment', *IEEE Transactions on Acoustics, Speech & Signal Processing*, 1988, ASSP-36, (2), pp. 265-284
- 6 SIX, P., CLAESEN, L., RABAEY, J., and DE MAN, H.: 'An intelligent module generator environment'. Proceedings of 23rd ACM-IEEE Design Automation Conference, Las Vegas, NV, USA, July 1986, pp. 730-735
- 7 SIX, P., VANDEWEERD, I., and DE MAN, H.: 'An interactive environment for creating module generators'. Proceedings of 12th European Solid-State Circuits Conference, Delft, Netherlands, Sept. 1986, pp. 65-67
- 8 MARWEDEL, P.: 'A new synthesis algorithm for the MIMOLA software system'. Proceedings of 23rd ACM-IEEE Design Automation Conference, Las Vegas, NV, USA, July 1986, pp. 271-277
- 9 GOOSSENS, G., RABAEY, J., VANDEWALLE, J., and DE MAN, H.: 'An efficient microcode-compiler for custom DSP-processors'. Proceedings of IEEE International Conference on Computer-Aided Design, Santa Clara, CA, USA, Nov. 1987, pp. 24-27
- 10 CROES, K., DE MAN, H., and SIX, P.: 'CAMELEON: a process tolerant symbolic layout system'. Proceedings of 13th European Solid-State Circuits Conference, Bad Soden, West Germany, Sept. 1987, pp. 193-196

Prof. H. De Man is with the Inter-University Microelectronics Centre, Katholieke Universiteit Leuven, Naamestraat 22, 3000 Louvain, Belgium, Prof. J. Rabaey is with the Department of Electrical Engineering & Computer Science, University of California, Berkeley, CA 94720, USA, and J. Vanhoof, G. Goossens, P. Six and L. Claesens are with IMEC vzw, Kapeldreef 75, B-3030, Leuven, Belgium.