



Advisory Panel in Mathematics, the IEEE Circuits and Systems ADCOM, and subcommittees on nonlinear networks, large-scale systems, and computer-aided design. He is the author of over 70 technical papers and two

spending the years 1966 at MIT and 1976 at Imperial College as a visiting professor. He was Assistant Director of Mathematical Sciences at IBM during 1971-1972. He is presently a second-level manager of the Mathematical Algorithms group. He is winner of the IEEE Best Paper award (Circuit and Systems Group), and has received two IBM Outstanding Innovation Awards and an IBM Patent Award. During 1970-1972, he was a NSF Chataqua Lecturer on Mathematical Models and Computing. He has been a member of the NSF

books. His fields of interest have been nonlinear networks, sparse matrices, numerical analysis, stability theory, computer-aided design, character recognition, and optimization. His most recent research has been in the construction and design of an automatic chip compiler, involving logic synthesis and minimization, state assignment, and high level design languages.

Dr. Brayton is a Fellow of the AAAS.

*

Alberto Sangiovanni-Vincentelli (M'74-SM'81-F'83), for a photograph and biography please see page 166 of this issue.

An Integrated Automated Layout Generation System for DSP Circuits

JAN M. RABAEY, MEMBER, IEEE, STEPHEN P. POPE, ROBERT W. BRODERSEN, FELLOW, IEEE

Abstract—An integrated CAD system for the automated design of digital signal-processing (DSP) circuits for audio and telecommunication applications is described. The system uses as unique input a symbolic description of the algorithm. This representation is translated into an actual layout using a two-step process. First, the symbolic input is mapped into the target architecture, which consists basically of a set of concurrent processors and dedicated I/O circuitry. The resulting hardware configuration is compiled into a layout description through a full exploitation of the hierarchy and the modularity of the architecture, calling consecutively a tiler, a floorplanner, and a global placement and routing tool. All these layout generation tools are able to support a wide range of technologies.

The provision of a dedicated register transfer level simulator allows for the efficient debugging and algorithmic checking of the real-time operating signal-processing algorithms.

The efficiency and the usefulness of this design methodology has been demonstrated by multiple examples. Experiments have shown that the use of these techniques can reduce the complete design process to a few months.

I. INTRODUCTION

REAL-TIME DIGITAL signal processing with its continuous data flow and its complex algorithms poses extreme computational demands which, most of the time, cannot be met by general-purpose processors or machines. This has resulted in the development of a range of

dedicated signal-processing architectures with an increased data throughput as a common feature. This can be obtained through a full exploitation of the inherent parallelism in the signal-processing algorithms, using pipelining and concurrency. Increases in the computational performance can also be achieved by the use of functional units optimized for digital signal processing, as parallel multipliers, address arithmetic units, and dedicated I/O processors.

Until recently, a full utilization of these architectural ideas on a single integrated circuit has been prohibited by technological restrictions. As a result of the advances in VLSI technology, the amount of the circuitry and the complexity of the algorithms which can be implemented on a single integrated circuit has increased dramatically. This has resulted in the development of a number of single-chip solutions for digital signal-processing applications. The approaches to design these chips can be divided into general-purpose and custom designs.

The general-purpose signal processor can be considered as the signal-processing equivalent of the microprocessor and basically uses pipelining and a parallel multiplier to increase the data throughput (e.g., [1]). The advantage of this approach is the programmability, which avoids the expensive design times of the custom approach. The generality of these processors is, however, a major handicap: the processor does not fit the specifications and the requirements of a specific algorithm, such as wordlength, datapath, memory size, data throughput, and I/O requirements.

Fully custom designs on the other hand can be optimally designed to meet the requirements of a specific applica-

Manuscript received February 15, 1985; revised April 16, 1985. This work was supported in part by the Defense Advance Research Projects Agency under Contract MDA903-79-C-0429.

J. M. Rabaey is with the Electronics Research Laboratory, the University of California, Berkeley, on leave from the Katholieke Universiteit of Leuven, Leuven, Belgium.

S. P. Pope was with the University of California, Berkeley. He is now with Cyclotomics Corp., Berkeley, CA 94704.

R. W. Brodersen is with the Department of Computer Sciences, University of California, Berkeley.

tion. Among the architectures currently in use, dedicated bit-serial and bit-parallel data paths, concurrent customized signal processors, and systolic arrays seem to be the most popular. The enormous design costs and the multiple design iterations due to the complexity of the circuits, however, make the custom approach unattractive for the medium- and low-volume applications. Standard cell or gate-array methods can shorten the design time, but only with a considerable loss in efficiency. Automation of the design process seems to be the appropriate way to make the custom-design methodology economically viable for those applications. The implementation of a number of these so-called "silicon compilers for signal processing" has been reported recently for bit-serial [2], [3] and bit-parallel hardwired datapaths [4], [5]. These systems take as an input a procedural description of the design and produce a mask-level description of the circuit layout.

This paper describes the realization of an automated design system for audio and telecommunication applications. It uses a number of parallel operating, pipelined, and microprogrammed signal processors as the target architecture. This approach has some apparent advantages over the compiled datapath approaches: decision making is extremely expensive in the hardwired datapath approach, which limits the use of these systems to applications such as filtering, while the use of microprogrammed signal processors allows for the integration of complete signal-processing systems.

Since the signal processors are microprogrammable structures, the input to the compilation system consists of a symbolic programming language. This is much closer to the block diagram description, familiar to the algorithmic designer (who is going to be the main user of the silicon compilers) than the procedural hardware description of the datapath compilers. Standard software compiler techniques can be used to generate the assembly language of the processors from a functional signal flowgraph description language. Another advantage of this architectural approach is that the construction of the processors (and I/O hardware) can be based on the use of macrocells [6]. A macrocell is a large, self-contained functional block of circuitry, such as a RAM or an arithmetic unit, which is parameterizable (e.g., number of words or wordlength). This allows for an exact mapping of size and function of the building blocks to the application in an area-efficient way.

A set of integrated software tools to support this design methodology will be described. This package contains a set of hierarchical layout generation tools and a dedicated, efficient register transfer level simulator, used for the debugging and the algorithmic checking of the input description.

Section II contains a brief description of the target architecture of the layout generator [7], [8]. The complete software system is discussed in detail in Section III, which consecutively handles the design file input, the supporting simulation aids, and the different levels in the layout generation process. The paper is concluded with a number of results, conclusions, and future plans.

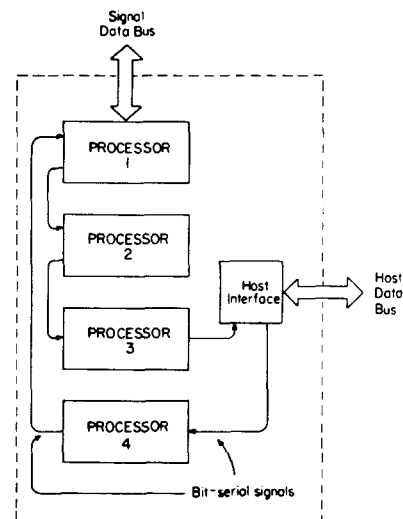


Fig. 1. Organization of four-processor IC with host interface.

II. THE TARGET ARCHITECTURE

This section contains only a rough description of the target architecture of our layout generator, as needed to elucidate the following sections. For a more detailed description, we refer the reader to [7], [8].

The fundamental signal-processing properties of this architecture can be summarized as increased data throughput through concurrency and pipelining and the provision of adequate and customized I/O circuitry.

A. The IC Organization

As illustrated in Fig. 1, the target architecture consists of a number of concurrently operating signal processors. Each of these processors handles a specific part of the algorithm. This is completely consistent with the serial nature of most signal-processing algorithms, in the sense that the algorithm is composed of a sequence of computational tasks. Due to the flexibility of the macrocell approach, each of the processors can be customized to its specific function (wordlength, memory, . . .).

Interprocessor communication is performed over bit-serial lines with parallel-serial (and serial-parallel) conversion and databuffering at the processor side. This protocol was found to be the most efficient in terms of area and circuit efficiency. Once again, the communication circuitry is customized under software control, which generates only the minimal number of interprocessor connections. These are defined in the design file under abstract form as "global variables." The timing and the implementation details of those globals are completely hidden to the user.

An important feature of the architecture is the provision of adequate I/O interfaces as needed in signal processing. The present version provides two types of interfaces: the sampled data inputs and outputs enter the IC over a bidirectional parallel bus at sample rate. A lower (frame) rate, interrupt-driven interaction is allowed with a host microprocessor and is handled by the host interface macrocell. Work is underway to include A/D and D/A conversion ca-

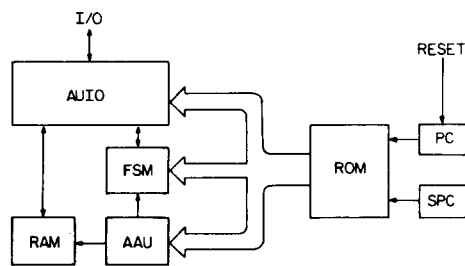


Fig. 2. Macrocell composition of a single processor.

pabilities on the chip based on sigma-delta modulation techniques [9], which are fully compatible with the macrocell approach.

B. Processor Architecture

An in depth consideration of the applications of interest (real-time audio band signal processing) showed that the basic processor architecture could be kept extremely simple in terms of the number of functional units and their interconnections. This approach (together with the exploitation of concurrency) was found to have the computational capabilities of the general-purpose processors with a substantial reduction in hardware complexity.

Hence, a fully configured processor contains only seven macrocells, as illustrated in Fig. 2: AU10 (arithmetic unit + I/O units), RAM (local variables + constants), AAU (address arithmetic unit for address indexing), FSM (finite state machine for decision making), ROM (program memory), and PC and SPC (program counter and subprogram counter). The AAU, FSM, and SPC are optional, depending upon the application [8].

The fundamental properties of the processor architecture are as follows.

- 1) There is almost complete separation between data and control path.
- 2) The parallel, bitsliced arithmetic unit (Fig. 3) contains four pipeline registers. The pipeline segmentation allows for a simultaneous memory access, shift operation, accumulation, and I/O operation. The multiply-accumulate operations, essential in every signal processor, are performed in a SHIFT-ADD fashion. This configuration avoids the implementation of an area-expensive parallel multiplier and takes advantage of the fact that most of the signal-processing algorithms use low-precision fixed coefficients. All arithmetic is performed in two's complement notation.
- 3) The control sequencer is kept simple and contains the ROM stored microcode. The program consists of a main program, followed by a fixed number of subprogram iterations. No branching is allowed, which means that the total execution time of the program is fixed and not data-dependent. This simplifies the architecture of the control sequencer and the synchronization of the interprocessor communications in a substantial way.
- 4) Due to the lack of conditional branches in the control flow, all decision-making capabilities are con-

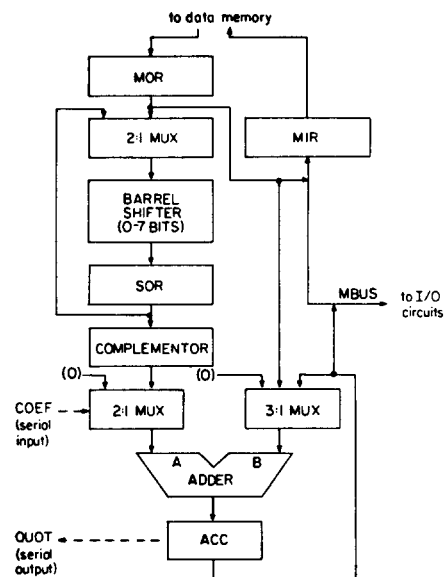


Fig. 3. Processor data-path architecture.

centrated in the user-defined PLA-based finite state machine (FSM). This FSM controls a conditional write operation by setting a condition code bit, depending upon the status of the processor. The conditional assignment statement is as general as the conditional branch, but can be less efficient if the test selects between two long, substantially different pieces of code. Fortunately, this situation seldom arises in signal-processing design.

- 5) The local variables are stored in the RAM macrocell, as well as the constants, which are stored as "read-only RAM words" (using modified RAM cells).

Since the present cell library is designed for a maximum clock rate of 5 MHz and the processor architecture is most efficient for a clock to sample rate ratio between 50 to 1000, it follows that the present architecture is best suited for sample rates between 5 kHz and 100 kHz, which covers most speech and telecommunication applications.

III. THE SOFTWARE TOOL PACKAGE

Fig. 4 displays a graph of the complete software system currently in use. All software is written in the C programming language and runs under the 4.2 BSD version of the Unix operating system. The graphic output support tools have been developed for a 68 000-based Unix workstation.

This section consecutively discusses the design file input description, the simulation aids, and the layout generation tools. It concludes with a brief description of a high-level front end, based on the Silage language and compiler [10], which are currently under development.

A. The Design File Description

The user input to the system is a single text file, called the "design file." The information contained in this file serves as the *unique* user input to the layout generator and the emulator.

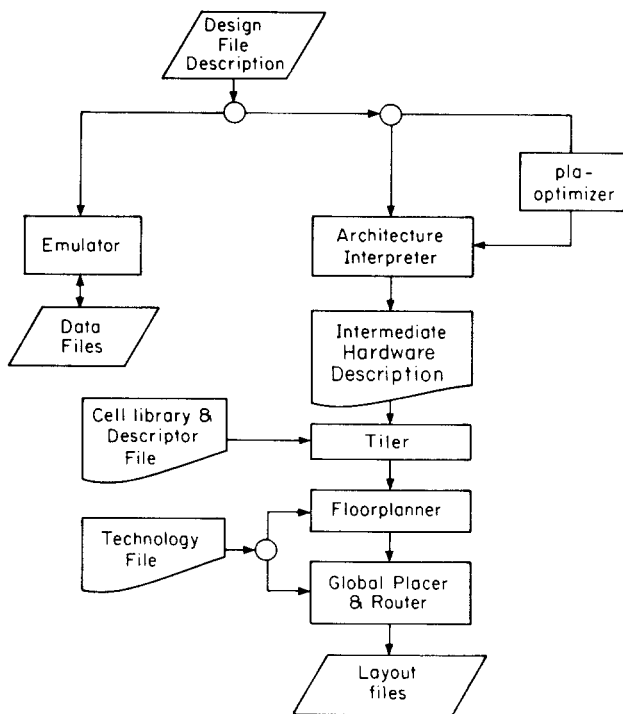


Fig. 4. Flowgraph of layout-generating software system.

For the formal definition of the design file syntax [11], the following meta-notation is used: semantic constructs are denoted by English words between the angular brackets $\langle \rangle$. These words are suggestive of the nature or the meaning of the construct. Production rules use $:: =$ to define a construct as an expression or a combination of constructs; curly brackets $\{ \}$ indicate repetition any number of times including zero; square brackets indicate optional factors (i.e., zero or one repetition); parentheses $()$ are used for grouping.

The basic entities in our architecture are the processors, the I/O interfaces, and the interprocessor communications. This is reflected in the overall format of the design file:

```

< design file > :: =
  < global definitions >
  < I/O definitions >
  ( < processor specification > { < processor specification >
  [ < constraints > ]

```

A global is variable which is shared between different processors or which is shared with the outside world. The definition of a global variable has to contain the wordlength of the variable and provides the necessary information to the compiler for the setup of the appropriate processor I/O structures.

The \langle I/O definitions \rangle block selects some of the global variables for communication with the outside world. It determines if an I/O variable is to be routed to (or from) the signal bus (at sample rate), or to the host interface or eventually to the A/D-D/A conversion block. The processor specification is described below, while the optional constraints block gives the user some additional control over

the processor timing (e.g., synchronization of several processors, external synchronization, . . .).

A processor unit is completely defined by the wordlength of the datapath, the number of local variables and constants (stored in the RAM macrocell), the optional definition of a finite state machine (as a set of logical expressions), and the assembly code (which unambiguously defines the control sequencer, being ROM, PC, and SPC, and the structure of the AAU). This leads to following syntax for the processor block:

```

< processor specification > :: =
  .processor : < processor-name < wordlength > >
  begin
    < local variables >
    [ < local constants > ]
    [ < fsm definition > ]
    < microcode block >
  end

```

The \langle local variables \rangle and \langle local constants \rangle blocks declare a set of identifiers and determine the local storage requirements (size of RAM macrocell). In the \langle fsm definition \rangle , a set of user-defined commands are declared. These commands are logical expressions that use the status of the processor and the states of the FSM as input and manipulate the states of the FSM. An important state is the CC (condition code) bit, which controls the conditional write operations.

```

example :
  SET__IF__POSITIVE: cc = !sign;

```

This statement defines a command, named SET__IF__POSITIVE, which checks the sign-bit of the accumulator and sets the condition code appropriately. This command can then be used freely in the microcode.

The \langle microcode block \rangle contains the actual programming of the processor. The assembly language itself is straightforward with only the pipelined structure (multiple simultaneous instructions) as a complicating feature. Also interesting is the absence of branching and looping with the exception of the mainprogram-subprogram division discussed previously.

The structure and the contents of the design file are illustrated with the simple example of a low-pass filter (Fig. 5). The desired response is realized with a 32-tap finite impulse response filter. The design file for the example, as given in Fig. 6, shows that the algorithm has been mapped into a single processor with a wordlength of 16 bits. Notice the split-up of the microprogram into mainprogram (input-output, initialization) and 32 iterations over the subprogram, which implements the basic multiply-accumulate operation and also performs the updating of the delay line. The multiplication is implemented as a set of shift-add operations. A faster implementation is possible by coding the coefficients directly in the microcode, but this increases the size of the microcode ROM substantially.

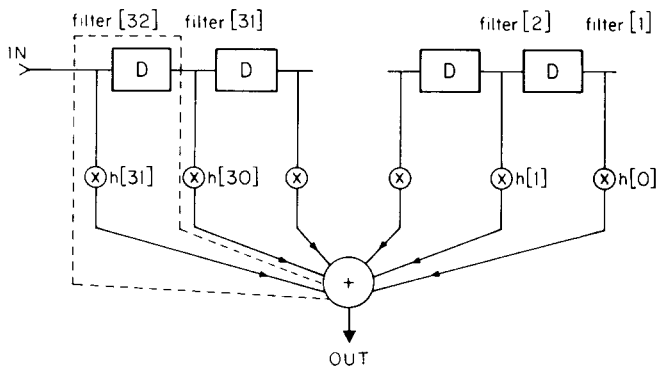


Fig. 5. Blockdiagram of 32-tap FIR filter.

```

global          /* definition of global variables */
begin
in<12>, out<12>, tap<12>;
end

io <12>         /* definition of input/output requirements */
begin
in : signal_in;
out : signal_out;
end

processor : one <16> /* definition of single processor */
begin
.local        /* definition of local variables */
begin
dly[33], result;
end

.constant    /* tap weights scaled for 12 bit accuracy */
begin
weight[32] = 0,-1.2,5,-8,-12,18,25,-35,-47,62,84,-117,-174,301,919,
            919,301,-174,-117,84,62,-47,-35,25,18,-12,-8,5,2,-1,0;
end

.main_pr     /* initialization, in- and output handling */
begin
r(result), mbus = in, le;          /* input new sample */
w(dly[32]), mbus = mor, out = mbus, acc := 0; /* output last result */
w(result), le;                    /* clear result */
end

.sub_pr <32> /* implements one tap of the fir */
begin
rx(weight);
rx(dly[1]), mbus = mor, tap := mbus;
r(result), sor:=mor;              /* start multiplication */
sor:=sor>1,acc:=mor + coef, sor.coef = tap;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
sor:=sor>1,acc:=acc+coef,sor;
acc:=acc-coef,sor;                /* update delay line */
rx(dly[1]), sor:=sor>1,
mbus = mor, le,
wx(dly),
w(result), le;
end
end

```

Fig. 6. Design File description of 32-tap FIR low-pass filter.

B. The Simulation/Verification Aids

One of the strong arguments in favor of the automation of the design process is the shift of the simulation efforts to higher, functional levels. In fact, the extensive hardware level simulations, which are standard in the custom-design world, are avoided due to the “correctness by construction” principle of the silicon compiler. This correctness can be assured, since at the time of the software development extensive circuit, timing, and switch-

level simulations were performed on the basic cell library, the macrocells, and the critical parts of the processor and interprocessor architecture. Hardware limitations are built into the compiler (e.g., maximum word length, maximum RAM and ROM size, maximum clocking frequency) so that the proper operation of the circuits is guaranteed.

Note that this approach has a degree of inefficiency, since the circuit elements are designed for the worst-case conditions and will be overdimensioned for some of the applications. It is felt, however, that the increased software complexity, which would be required if the library cells were parameterized, was not justified by the relatively small reductions in circuit area and power dissipation.

The above considerations demonstrate that in this methodology most of the errors will be generated at the symbolic design file level or at the algorithmic level. General register transfer and functional level simulators could be used to check the behavioral correctness at this level, but the complex nature of the applications of interest makes them too slow to be of very much use. Moreover, in order to assume the “correctness by construction” principle, it has to be verified that a design file description doesn’t violate any of the architectural or hardware limitations of the compiler. Therefore, a special-purpose dedicated emulation tool for our multiprocessor architecture has been developed.

This emulator/verifier, called *Demon*, serves a triple goal: *debugging* of the microcode in an interactive mode, *verification* of the hardware realizability, and checking of *the performance of the algorithm* within the scope of the target architecture through full and realistic emulation on large datafiles (e.g., speech). *Demon* can be considered as a register-transfer simulator with the following extra features to fulfill the above goals:

- Syntax and semantic checks on the design file description.
- Verification of the description in terms of architectural and hardware limitations, such as maximum RAM, ROM, and FSM dimensions, maximum processor word length in function of the sampling frequency, consistency of the addressing techniques used, connectivity of the interprocessor communication network.
- Setup of the timing and the basic structure of the control sequencer. Using an identical procedure as the Architecture Interpreter (see below), *Demon* automatically computes the basic cycle and the number and the timing of the control signals needed. In fact, in order to guarantee a complete mapping between software emulation and hardware realization, the emulator is “microprogram driven”: this means that the sequencing and the control flow of the program are controlled only by the ROM and FSM contents of the processors. For instance, the looping in the emulation is controlled by the “return,” “jsr,” and “reset” signals, as stored in the microprogram

ROM, while conditional operations are controlled by the status of the finite state machine.

- Setup of the timing and the structure of the interprocessor communications. The delays, connected with the serial data-transfer are checked and warnings are issued if these delays result in extra sample delays.

Demon differs from classical processor emulators by its consistent handling and support of concurrency. This is achieved by monitoring the interprocessor data flow using a buffering strategy, which is identical to the protocol used in the actual hardware realization.

In the debugging mode, Demon provides a full set of interactive facilities: it supports features such as tracing, breakpointing, single stepping, and the examination and setting of registers, global, and local variables.

While new algorithms are more efficiently developed and analyzed on a functional level (using, e.g., the Silage description and simulator, as described in Section III-D), it is also important to examine the influence of an actual implementation on the algorithm performance. Especially the effects of finite word lengths, truncation and saturation have to be considered. This often requires lengthy simulations on enormous data files. In fact, the quality of most of the speech-related algorithms is basically decided by human perception factors and can't be judged solely by arithmetic considerations. For instance, the quality of a vocoder implementation is judged by an actual audio test on a speech file, which has been passed through the analysis and the synthesis parts of the vocoder.

These simulations can be performed in the batch mode of Demon. The dedicated nature of the emulator allows for an optimization of the performance using, e.g., data structures that are linked directly to the architecture. This implementation strategy makes lengthy system simulations possible in a reasonable computation time, with the simulation time increasing linearly with the total number of microinstructions.

Demon has been used to emulate a complete LPC-10 vocoder system [12], including the analysis, synthesis, and pitch-tracking part (Fig. 7). The algorithm has been implemented in the architecture using three processors (each with 494 cycles per sample) and a host interface unit. For this system, a ratio of 4000 between simulation time and real time was obtained.

C. The Layout Generation Process

The actual core of the system, the layout generator, consists of a set of programs, each of them handling different levels in the design hierarchy. Some general remarks should be considered before the detailed handling of each of those tools.

- 1) The layout generation process, in spite of high automation, still has to remain interactive. A "push the button" compilation has to be possible, but the user must have the possibility to interfere at any level of the layout process.

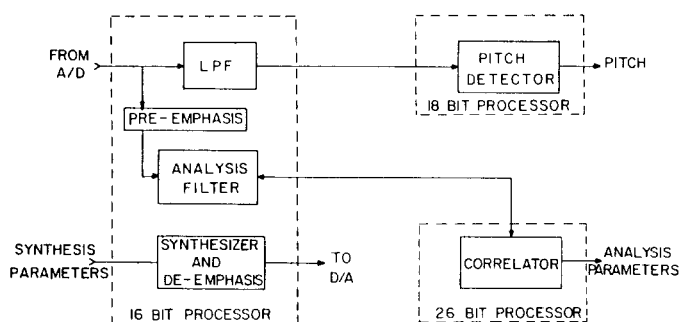


Fig. 7. Block diagram and multiprocessor organization of LPC vocoder.

- 2) The compilation process is iterative: the fast feedback of layout sizes allows the user to make architectural or algorithmic decisions using area considerations (e.g., distribution of the functions over the processor, selection between different implementations of an algorithm, etc.). This is a major advantage of the automated design concept over a conventional custom design, where the effects of algorithmic decisions on the layout are often only discovered at a stage in the design process, which is too late for major design changes to be made. In order to take full advantage of this feature, the entire software process needs to be fairly efficient in terms of computation time.
- 3) Fast and clear feedback of geometrical data (placement, size of routing channels) is essential for a full exploitation of the interactive and iterative features of the layout generator. A basic component of the system is, therefore, a layout display tool, which allows the user to have an immediate view on the different levels of the design hierarchy at every instant in the design process. An example of such a view (at the processor level) is given in Fig. 11. "Layoutool" runs under the window system of a 68 000-based work station and makes full use of the design hierarchy to speed up the plotting process.
- 4) The software is able to handle minor technology changes without adaptation. The fundamental technology parameters are stored in a technology file, which is accessed by the layout generation tools. This approach allows us to handle technology scaling and to span NMOS as well as CMOS without software adaptations. However, major technology changes (as, e.g., the addition of an extra routing layer) currently need a revision of the software. For more specific details on the technology independency, we refer to the subsequent discussions on the different tools.

The layout procedure is composed of the following consecutive steps: the translation of the symbolic design file into an intermediate hardware description, the tiling of the macrocells, the floorplan based assembly of the processors, and finally the chip assembly using standard placement and routing techniques.

C.1. The Architecture Interpreter: The ARCHitecture interpretER (Archer) forms the linking bridge between the

design file and the layout generation. This step, which is crucial in the compilation process, translates the symbolic input language into a hardware description.

Archer has to know how to map symbolic data into architectural properties and how to select between architectural alternatives. This includes the dimensioning of the memories and the datapath, the construction of the control sequencer and the programming of the control ROM, the setup of the interprocessor communications (timing, hardware units), the configuration of the address arithmetic unit, and the programming of the FSM's. The architecture data and the decision rules are currently hardwired in the software.

More versatile implementations using knowledge-based techniques and external decision rules are being considered. This would allow Archer to choose between architectural alternatives.

The interpreter must have access to the construction details (floorplans) of the different macrocells. In other words, it must know how to translate the hardware requirements (e.g., a RAM memory of n words, each m bits wide) into an array of basic library cells and where to connect the data and control wires. The macrocell floorplans are described in a procedural way using a C-based language. This eases the addition of new macrocells (e.g., additional I/O interfaces) or the adaptation of existing macrocells to architectural changes (e.g., column-decoded ROM's and RAM's). The basic constructions in this description are "AddCell()" and "AddPort()" which, respectively, position a library cell in the array and add an external terminal of a certain type to the macrocell boundaries.

The present system supports nine basic macrocells, being RAM, ROM, FSM, AUIO, AAU, PC, SPC, HOST, and PADS. An example of a simple macrocell floorplan (PC macrocell) and its actual layout are given in, respectively, Fig. 8(a) and (b).

The output of Archer is a hierarchical hardware description language (hdl), which is illustrated in Fig. 9 for the case of the *pc* of processor *pr0* of design *lpfir*. It contains the following information:

- Definition of the hierarchy: division of the design in modules (processors, io units, pads, intermodule nets) and the composition of modules in terms of macrocells and intramodule nets.
- The macrocell descriptions, consisting of arrays of basic library cell references. This description does not contain any dimensional information.
- The netlists for the intraprocessor and the interprocessor connections. A terminal on a cell instance can be defined unambiguously by using the macrocell array indices: for instance, *net__43* in Fig. 9 describes a net connecting the *reset* terminal of the cell *pc.ct1* (on row 0 and column 9 of the PC macrocell) to an output of the ROM macrocell of processor *pr0*.

The tool diagram of Fig. 4 shows that the minimization of the finite state machines is performed in a preprocessing

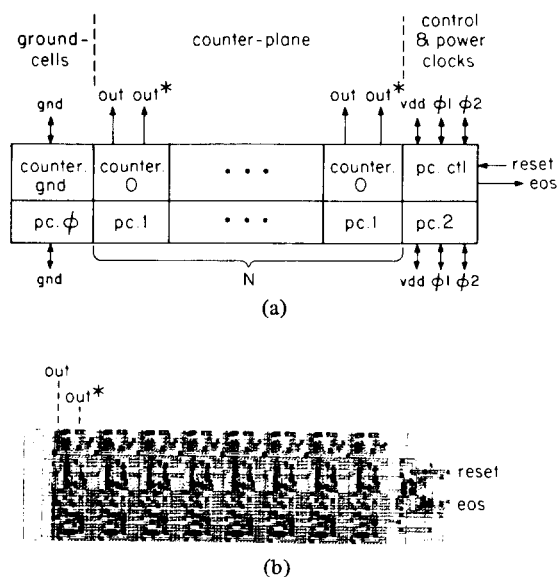


Fig. 8. Macrocell floorplan for PC (program counter) macrocell. (a) Floorplan description: cell and terminal definitions. (b) Corresponding layout view.

```

begin lpfir
  pr0
  pads
  net
end lpfir

begin pr0
  pc
  spc
  ram
  rom
  auio
  aaui
  net
end pr0

begin pr0pc
  counter(gnd..0..0..0..0..0..0..0) pc.ct1 /* row 0 */
  pc(0..1..1..1..1..1..1..1..2) /* row 1 */
end pr0pc

begin pr0net
  net_36 : pr0pc[0.9]eos pr0aaui[3.7]eos /* reset */
  net_43 : pr0pc[0.9]reset pr0rom[20.1]out1 /* eos */
  net_49 : pr0pc[1.1]out pr0rom[0.18]inbtm /* bit 0 */
  net_50 : pr0pc[1.1]out* pr0rom[0.18]inbtm*
  ...
  net_63 : pr0pc[1.8]out pr0rom[0.25]inbtm /* bit 7 */
  net_64 : pr0pc[1.8]out *pr0rom[0.25]inbtm*
  ...
end pr0net

```

Fig. 9. Intermediate hardware language, defining the PC instance (of dimension 8) of processor 0 and its connections.

pass. This program calls some standard tools for pla-handling (Eqntott, Pop) [13], [14] and passes the minimized truth tables to Archer.

Notice that this first pass of the compilation process does not handle any geometrical data. The only inputs are the design file input, the optimized truth tables of the FSM's, and the floorplan construction rules for the various macrocells.

C.2. The Macrocell Tiler: The first step in the actual layout generation process is the assembly of the macrocells from the intermediate hardware description. These are generated by stacking library cells in two-dimensional arrays, based on the predefined macrocell floor-

plan, and making the internal connections by abutment. Besides a set of layout files for the individual macrocells, a simplified database is produced, where the macrocells are replaced by boxes of defined dimensions with terminals located on the boundaries. This serves as the basic information for the subsequent placement and routing programs.

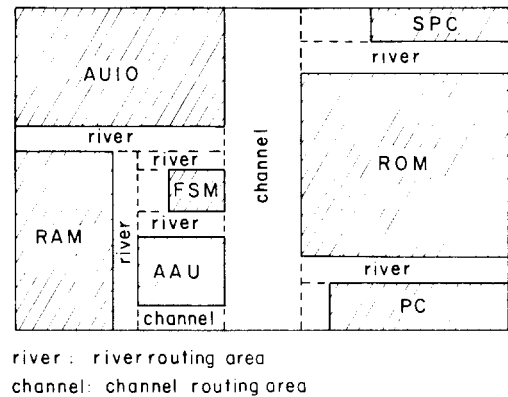
The Tiler extracts the geometrical properties of the basic library cells (size and terminal locations) from a "descriptor file," which is specific to each implementation of the library. This concept makes the Tiler itself technology independent. At this time, a NMOS library is available and a CMOS library approaches completion.

The small number of basic library cells (200 in total), which are designed only once for each technology, allows for extensive area and performance optimization. This makes it possible for most of the active circuitry to be essentially equivalent to full custom layout and is the major reason that the macrocell approach can yield such efficient layouts. The assembly of the macrocells consists of merely stacking the cells, which restricts the interchangeability of the leaf cells, but does not consume extra space. This is in contrast with more general functional block compilers, which use extra routing between library cells (as, e.g., in the macrocell generators used in [17]).

C.3. Floorplan-Based Processor Assembly: An important step in the layout generation process is the placement and interconnection of the macrocells. The problem has been subdivided into two phases, in conformity with the hierarchical levels in the chip architecture: in a first pass, the individual processors and I/O interfaces are assembled using a floorplan-driven strategy. The second pass consists of the total chip assembly using general placement and routing (GP&R) techniques. This split is motivated by the following considerations.

- 1) The compilation of the individual processors is a restricted problem: the layout generator "knows" the basic structure of the processor and the nature of the internal interconnections. GP&R techniques ignore this architectural knowledge and start from scratch.
- 2) Specialized routing constraints are handled by GP&R with great difficulty (e.g., single level routing for certain critical busses, power, ground, and clock routing).
- 3) Usage of GP&R at the lowest level can require hours of computer time, which is not compatible with an iterative design process.

To evaluate the feasibility of floorplan-based processor generation, a first version of the floorplanner was implemented using a single, hardwired floorplan, which is shown in Fig. 10. This floorplan, applied on the pitch processor of the full duplex LPC vocoder of Fig. 7, produced the circuit layout of Fig. 11, in which the black boxes represent macrocells. Examination of a large number of processor layouts, produced by this floorplanner, demonstrated that a limited set of floorplans (between 3 and 5) would cover almost the complete processor field.



river: river routing area
channel: channel routing area

Fig. 10. Processor floorplan, defining the relative positions of the macrocells and the routing channels.

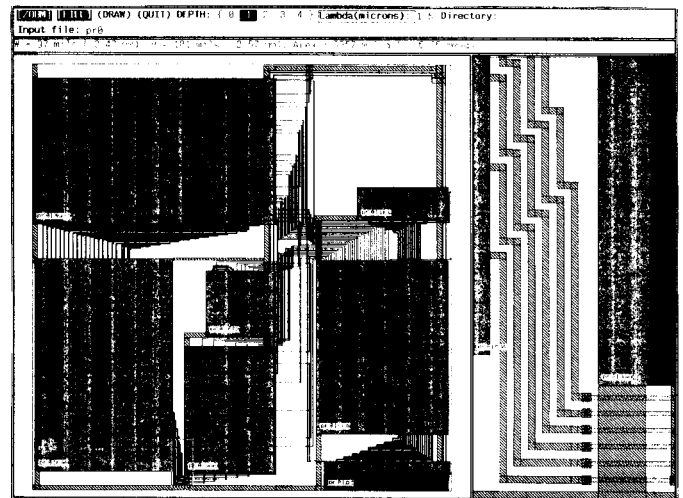


Fig. 11. Instance of processor floorplan (pitch processor of LPC vocoder), as generated by the window-oriented view tool *layout*.

These conclusions were the incentive for the development of a general floorplan-driven placement and routing tool. It takes as input the dimensional data, passed by the tiling process, and a set of floorplan description files. The floorplan of a structural unit (processor, host interface, A/D converter, . . .) basically contains the "knowledge" of the architectural designer and is described using a specialized floorplan description language. The description contains following information:

- The relative placement of the macrocells and the routing channels (floorplan commands: `place()`, `transform()`);
- Global routing guidance. This includes the assignment of the routing cables (being a set of nets with the same source and destination) to the different routing channels and the determination of the routing strategy for each of the sectors (river routing, channel routing) (commands: `route()`, `riverroute()` and `channelroute()`);
- Power, ground, and clock routing information (command: `powerroute()`).

A set of alternative floorplans can be provided for each unit (e.g., about three floorplans are sufficient for our sig-

nal processor) so that the floorplan tool (called Flint) can select the optimum strategy for a particular case. Flint first performs the absolute placement for each of the floorplans, based on graph manipulation techniques. An exact estimation of the required routing space is made. For the one-layer routing, Flint uses an estimation algorithm which is linear with the number of terminals (based on principles described in [18]). The estimator for the channel router executes the complete channel routing algorithm to determine the needed number of tracks. At the end of this phase, the crossings of nets and routing channel borders have been fixed so that the detailed routing phase is a strictly local process.

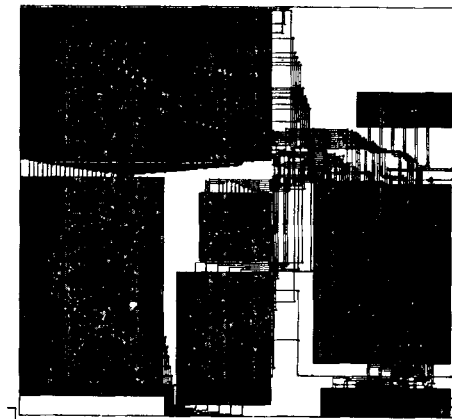
After the absolute placement phase, Flint performs a floorplan selection based on area considerations and moves to the detailed routing. Both river and channel routers were developed for rectangular channels and perform on a gridless base, which allows for denser routing and which fits with the arbitrary placement of terminals on the macrocell boundaries. The river route algorithm can be considered as an extension of the separation algorithms, described in [18], to rectangular channels with terminals on the four sides. The channel router is based on an algorithm due to Kuh [19], but has been modified to eliminate the fixed grid assumptions and to handle cyclic constraints. This router, as well as the river router, is driven by a set of design rules (as described in the technology file). The minimal distance between the border terminals is poly-to-contact, while the track distance is contact-to-contact.

Fig. 12 compares two Flint realizations of the same processor for a different placement of the FSM and using different routing strategies. A total freedom was given to Flint to choose its routing strategy (channel or river routing) in the channels of the processor of Fig. 12(a). For the second processor, routing restrictions were imposed for most of the channels, as can be seen by the large number of river routed channels in the layout plot.

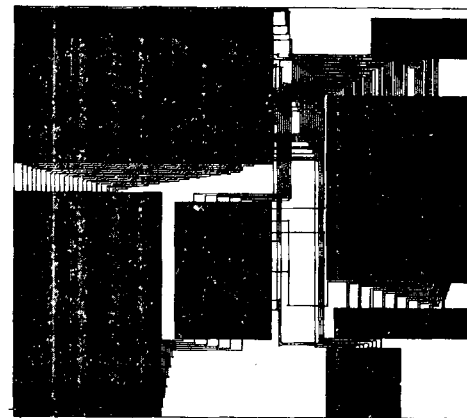
C.4. Chip Assembly: Our current approach to the chip assembly problem is to use a fixed floorplan (Fig. 13). This floorplan consists of two rows of blocks with an optimal partitioning, ordering and rotational orientation for the various blocks. Five subsequent calls of the channel router handle the interprocessor routing and the pad connections. This technique simplifies the power, ground, and clock routing problem and is identical to the procedure described by Moulton [15].

This strategy delivers reasonable results for the technologies used in the present cell libraries. In fact, at this level in the hierarchy, the number of blocks and interconnections is few and fits easily in the fixed floorplan.

With scaling technologies, however, the number of processors (or other modular units) is likely to grow, so that the use of a fixed floorplan will become unacceptable. Therefore, more flexible approaches, based on GP&R, are currently under investigation. While the usage of GP&R proved to be inappropriate at the macrocell level, it is the proper way to handle the total chip assembly. All inter-



(a)



(b)

Fig. 12. Flint-generated floorplans using different routing strategies. (a) Free routing. (b) Floorplan restricted routing.

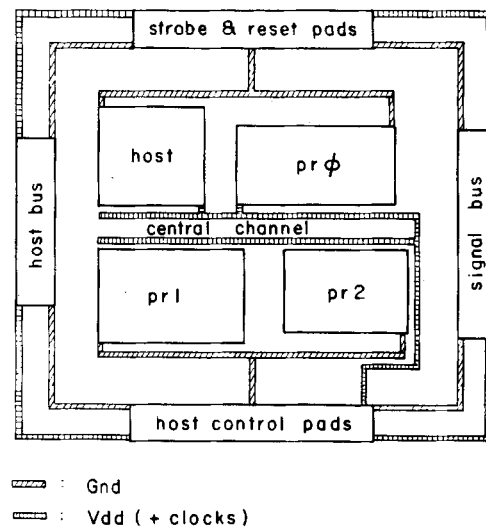


Fig. 13. Chip floorplan, including power, ground, and clock distribution.

processor and pad connections have been designed with enough driving capabilities so that global routing guidance and special routing precautions (with exception of the power and clock routing) are unnecessary.

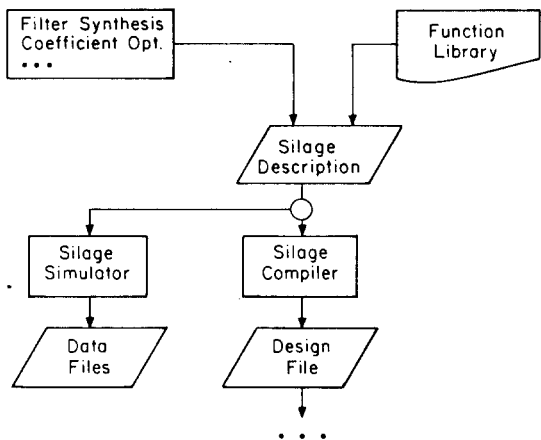


Fig. 14. Software graph of Silage-based front-end system.

D. The Functional Front-End System

Although the design file language allows the user to describe his algorithm at a symbolic level without knowledge of the underlying hardware, he still has to be aware of the architectural details of the target architecture. Moreover, programming at the microcode level is a labor-intensive and error-prone job, while the register-transfer simulation is too slow to be used at the algorithm development level.

The development of a language that allows for a functional description of signal-processing algorithms is, therefore, appropriate. This upper level input can then be compiled into the symbolic description of the target architecture using standard software compilation and optimization techniques. The abstractness and the hardware independence of such a description make it perfectly suited as the input language for the extensive function simulations at the algorithm development level.

A diagram of such a front-end system, based on a language optimized for describing signal-processing algorithms called "Silage," is given in Fig. 14. Silage [10] has been designed as an "applicative language" in an attempt to describe the signal-flow nature of the signal-processing algorithms. An applicative language is a language whose fundamental operation is function application and that has no variables or assignments. This is equivalent to the signal flowgraph, where nodes represent instances of functions and arcs represents the paths followed by the signals.

A Silage description of the low-pass filter of Fig. 5 is given in Fig. 15. The statements that describe the function FIR are to be interpreted as simultaneous equations and not as assignments.

The development of the Silage compiler and simulator is currently under way [10]. An important aspect of the compiler design is the optimization of the generated microcode to the pipelined nature of the datapath and the parallelism in the architecture.

IV. PRESENT RESULTS

The automated design system has been used to generate a considerable number of applications, including an audio equalizer, a decision feedback equalizer with timing re-

/* 32 tap Lowpass Fir filter with fixed coefficients */

```

#define internal num <12>
#define external num <8>
#define NR_OF_TAPS 32

my_coefs = { 0,-1,2,5,-8,-12,18,25,-35,-47,62,84,-117,-174,301,919,
            919,301,-174,-117,84,62,-47,-35,25,18,-12,-8,5,2,-1,0 };

func main(in: external): external =
begin
return = external( fir(internal(in), my_coefs, NR_OF_TAPS) );
end;

func fir(in: internal; coefs: num []; nr_of_taps: num): internal =
begin
return = Sum[0];
Sum[nr_of_taps] = 0.0;
(i: 0 .. nr_of_taps-1)::
Sum[i] = Sum[i+1] + coefs[nr_of_taps-1-i]*in[i];
end;
  
```

Fig. 15. Silage description for 32-tap low-pass FIR filter.

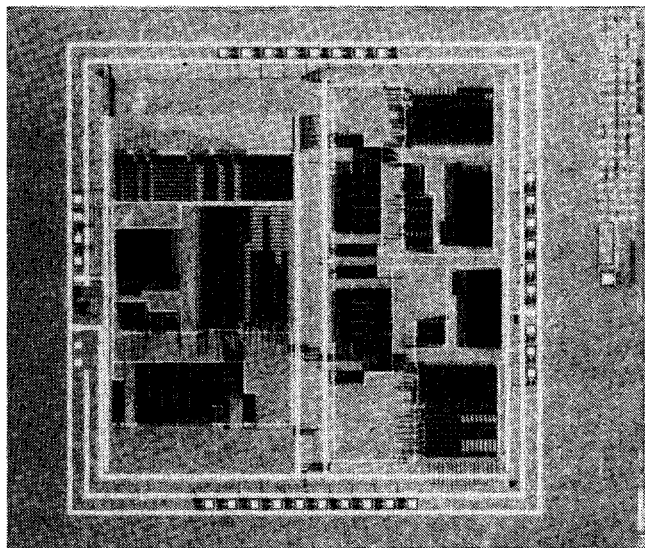


Fig. 16. Circuit plot of LPC vocoder (7 mm x 7 mm).

covery, a full duplex LPC vocoder [8], a full duplex 300-baud modem [16], and a filter bank for speech recognition. Other applications are currently under investigation, such as a 1200-baud modem, an ADPCM coder, a channel vocoder, and a speech scrambler. The number of parallel operating processors in these applications ranges between one and three, while the LPC-10 vocoder and the audio equalizer also require a host interface.

The layout generator presently uses a 3-4-μm NMOS library, while a 1.5-3-μm CMOS library is near completion. The results of the layout generation process for the LPC-10 vocoder (7 mm x 7 mm, 27 400 transistors, 8-kHz sampling frequency) and the 300-baud modem (4.1 mm x 7 mm, 20 000 transistors, 9.6-kHz cycle) are shown in, respectively, Figs. 16 and 17. Close examination of the drawings reveals that higher circuit density can be achieved by considering a larger set of processor floor-plans (in contrast with the single one which was used in this realization) and by allowing a more flexible pad placement.

The importance of the silicon compilation concept is stressed by this observation: the complete 300-baud mo-

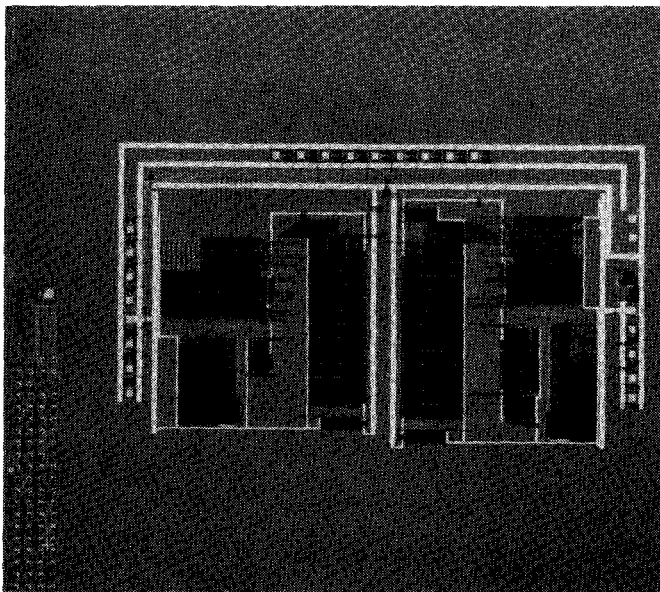


Fig. 17. Circuit plot for full duplex 300-baud modem (7 mm × 4 mm).

dem (algorithm development, design file coding and debugging, layout generation) was generated by a student in less than two months [16].

V. CONCLUSIONS

An integrated automated layout generation system for digital signal-processing circuits is presented. The present system, which was considered to be a test vehicle, allows us to draw some important conclusions concerning the target architecture and the software tools needed in the layout generation process.

- 1) The macrocell-based automated-design methodology allows for the generation of complex signal-processing chips with far less effort than manual design and layout, substantially reducing the probability of errors. The implementation of the complete software system (including the development of the tools) took about one man-year. A few successful applications easily outweigh the development costs of the software and the cell library.
- 2) A strong argument in favor of automated layout generation is the technology independency of the software. This minimizes the redesign efforts when changing or scaling the technology.
- 3) The modular and the hierarchical approach allows for a simple adaptation of the software to changes in the target architecture. In fact, the nature of the developed layout tools (tiler, floorplanner, global placer, and router) makes them suited for other macrocell-based architectures, concentrating all architectural details in the Architecture Interpreter. Future research efforts will be concentrated on the generalization of this software module.
- 4) The modularity, combined with the efficiency of the tools, also supports the iterative nature of the silicon compiler: due to the almost immediate feedback of

dimensional data, the designer can optimize the silicon realization of his application, e.g., decisions at the processor floorplan level or even at the algorithmic level can be based on chip area and power considerations.

- 5) The unique input file to both layout generation and simulation assures the consistency between simulation results and hardware realization.
- 6) The present target architecture is very flexible and proved its ability to handle most of the applications in the 5–150-kHz range. Some weaknesses in the architecture have become apparent and are currently under examination: this includes more support for variable-variable multiplications for, e.g., adaptive applications, a more flexible control structure (e.g., a more powerful FSM) and extension of the I/O interface (A/D–D/A conversion, serial I/O, digital strobe lines). The requirements of an architecture for higher speed applications are also under consideration.

The most important aspect of the compilation methodology is that the design files (or the higher level Silage input files) can be prepared by algorithmic designers without experience in the field of hardware design. This not only opens the world of custom-designed chips to these designers, but should have a considerable influence on the development of the signal-processing algorithms itself.

ACKNOWLEDGMENT

The authors wish to thank P. Ruetz, M. Torkelson, J. Tzeng, B. Abbott, and P. Hilfinger for their contributions to the project.

REFERENCES

- [1] S. Magar, E. Caudel, and A. Leigh, "A microcomputer with digital signal processing capability," in *Proc. ISSCC 1982*, pp. 32–33, Feb. 1982.
- [2] P. Denyer, D. Renshaw, and N. Bergmann, "A silicon compiler for VLSI signal processors," in *Proc. ESSCIRC 1982*, Brussels, Belgium, pp. 215–218, Sept. 1982.
- [3] J. Vandewalle *et al.*, "A unified box of VLSI building tools for digital signal processors," in *Proc. ICCD 1984*, Portchester, NY, Oct. 1984.
- [4] J. Fox, "The MacPitts silicon compiler: A view from the telecommunications industry," *VLSI Design*, pp. 30–37, May/June 1983.
- [5] J. Schuck, M. Glešner, and H. Joepen, "Algic—A flexible silicon compiler system for digital signal processing," in *VLSI Signal Processing*. New York: IEEE Press, 1984, pp. 216–227.
- [6] S. Pope and R. Brodersen, "Macrocell design for concurrent signal processing, in *Proc. Third Caltech Conf. on VLSI*, pp. 395–412, Computer Science Press, 1983.
- [7] S. Pope, J. Rabaey and R. Brodersen, "Automated design of signal processors using macrocells," in *VLSI Signal Processing*. New York: IEEE Press, 1984, pp. 239–251.
- [8] S. Pope, "Automated generation of signal processing circuit," Ph.D. dissertation, Memo UCB/ERL M85/11, U.C. Berkeley, Feb. 1985.
- [9] M. Hauser, P. Hurst, and R. Brodersen, "An MOS A/D convertor-filter combination requiring no precision analog components," in *Proc. ISSCC 1985*, pp. 80–81, Feb. 1985.
- [10] P. Hilfinger, "Silage: A high-level language and silicon compiler for digital signal processing," in *Proc. CICC 1985*, Portland, OR, May 1985.
- [11] J. Rabaey, "Lager: An automated layout generating system for digital signal processing circuits, User Manual V1.3," Memo UCB/ERL M85/5, U.C. Berkeley, Feb. 1985.

- [12] G. Kang, "Application of linear predictive encoding to a narrowband voice digitizer," Naval Research Lab. Rep. 7779, Washington, DC, 1974.
- [13] B. Cmelik, *EQNTOTT Reference Manual*, 1983 VLSI Tools, Rep. UCB/CSD 83/115, CS Div. of EECS, U.C. Berkeley, 1983.
- [14] P. Simanyi, A. Newton, and A. Sangiovanni-Vincentelli, "The POP PLA optimization program," in preparation.
- [15] A. Moulton, "Laying the power and ground wires on a VLSI chip," in *ACM IEEE 20th Design Automation Conf.*, pp. 754-755, 1983.
- [16] W. Abbott, "Design of a 300 baud FSK modem using customized digital signal processors," Memorandum no. UCB/ERL M84/93, U.C. Berkeley, Aug. 1984.
- [17] M. Buric, C. Christensen, and T. Matheson, "The Plex project: VLSI layouts of microcomputers generated by a computer program," in *Proc. IEEE-ICCAD '83*, Santa Clara, CA, pp. 49-50, Sept. 1983.
- [18] J. Ullman, *Computational Aspects of VLSI*. Computer Science Press, 1984, pp. 441-450.
- [19] T. Yoshimura and E. Kuh, "Efficient algorithms for channel routing," *IEEE Trans. Computer-Aided Design*, vol. CAD-1, pp. 25-35, Jan. 1982.

*



circuits.

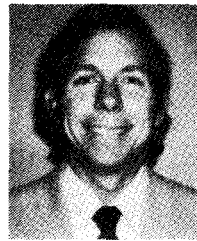
Jan M. Rabaey (S'80-M'83) was born in Veurne, Belgium, on August 15, 1955. He received the E.E. and the Ph.D. degrees in applied sciences in 1978 and 1983, respectively, from the Katholieke Universiteit Leuven, Belgium, where he worked on computer-aided design tools for switched capacitor circuits.

Since 1983, he has been a Visiting Research Engineer at the University of California, Berkeley. His current interest are in computer-aided analysis and automated design of digital signal-processing

Stephen P. Pope received the B.S. degree from the California Institute of Technology, Pasadena, California, in 1978, and completed his doctoral work at the University of California, Berkeley, in 1985.

He is currently on the engineering staff of Cyclotomics Corp., Berkeley, CA, where he is studying IC implementations of error detection and correction algorithms.

*



Robert W. Brodersen (M'76-SM'81-F'82) received the B.S. degree from California State Polytechnic College, Pomona, in 1966, and the E.E., M.S., and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge, in 1968, 1968, and 1972, respectively.

From 1972 to 1976, he was with Texas Instruments, Inc., Dallas, TX, studying the operation and application of charge-coupled devices. In 1976, he joined the faculty of the University of California Berkeley, where he is a Professor. He is studying the application of IC technology to signal processing, image processing, and user interfaces.

AROMA: An Area Optimized CAD Program for Cascade SC Filter Design

EDGAR SÁNCHEZ-SINENCIO, SENIOR MEMBER, IEEE, AND JAIME RAMÍREZ-ANGULO

Abstract—An area optimized computer-aided cascade switched-capacitor (SC) filter design program called AROMA is presented. The program contains several user-selectable filter approximation techniques. AROMA permits the user to make tradeoffs between several design parameters such as passive sensitivity, op-amp output voltage swings, clock frequency, and the total capacitance of the filter. The program has default values of the tradeoffs, however, each block in the cascade filter can be modified by the user to meet specific requirements. AROMA incorporates recent results of practical interest to the SC filter designer. The flexible structure of the program allows for easy addition of new results, i.e., refined efficient building blocks, additional filter approximations, etc. It can be used by the novice as well as by experienced designers. Furthermore, the program is user-oriented. It can start, in its simplest mode, with a set of frequency specifications. The output gives a circuit description file that can be layed out by an analog

circuit layout generator program. Examples illustrating the practicalness of AROMA are given.

I. INTRODUCTION

SWITCHED-CAPACITOR (SC) filters have reached certain maturity [1]-[4], especially in voice-band applications. However, the need for better performance, reasonable sensitivity values, and mainly reduced design and chip cost is still a problem. Motivated by the above problem, the computer program AROMA was developed.

The purpose of this paper is to describe a computer-aided design approach of cascade SC filters. Several important practical results for the SC filter designer have been incorporated. The end result of the program is a circuit description file that can be layed out by a layout generator program.

Manuscript received January 3, 1985; revised April 11, 1985.

The authors are with the Department of Electrical Engineering, Texas A&M University, College Station, TX 77843.