

Synthesis of Pipelined DSP Accelerators with Dynamic Scheduling

Patrick Schaumont, Bart Vanthournout, Ivo Bolsens, and Hugo J. De Man, *Fellow, IEEE*

Abstract—To construct complete systems on silicon, application specific DSP accelerators are needed to speed up the execution of high throughput DSP algorithms. In this paper, a methodology is presented to synthesize high throughput DSP functions into accelerator processors containing a datapath of highly pipelined, bit-parallel hardware units. Emphasis will be put on the definition of a controller architecture that allows efficient run-time schedules of these DSP algorithms on such highly pipelined data paths. The methodology will be illustrated by means of an image encoding filter bank.

I. INTRODUCTION

COMPLEX digital systems such as the videophone terminal of Fig. 1 typically consist out of a heterogeneous mix of hardware blocks [1]: processor cores, general purpose macroblocks, and dedicated accelerator processors. These accelerator blocks are required to execute high performant DSP functions such as motion estimation and DCT/IDCT functions.

In this paper we will concentrate on the generation of such application specific accelerator processors. We will highlight both the design issues and the architecture characteristics. The requirements of such accelerator processors are as follows.

- High throughput requirements impose the usage of pipelined data paths.
- Area can be saved through the hardware sharing of different micro-instructions.
- The accelerator processor has to be embedded in an overall system architecture.
- The accelerator functions can execute both at a manifest rate and a nonmanifest rate or *Data Introduction Interval* (DII [2], [3]). An example of the former is the processing of a data stream out of an A/D converter. An example of the latter is the processing of data out of a processor core inside the system.

The support of a nonmanifest DII allows to split the development of the system control component schedule and the accelerator processor schedule. The software executed by the system control component thus can be revised even after the accelerator processor was developed. This is an essential feature in the presence of today's complex algorithms.

As seen from the system control software, the accelerator component is a function call. Execution of this call spawns off the system control thread into the accelerator, executes

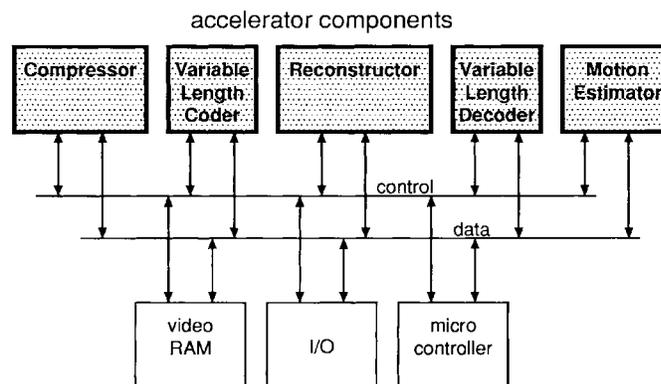


Fig. 1. Architecture of a video phone.

the accelerator operations, and resynchronizes with the system control thread. Such a scheme is used by some commercial numerical coprocessor components [4].

The presented synthesis system allows to generate such components, but with a much higher complexity and processing power.

II. OVERVIEW OF THE WORK

Automated synthesis systems for pipelined datapaths have been reported previously: [2], [3], and [5]. All of these assume a fixed DII. Therefore, they fix the runtime schedule at compile time. For an accelerator function, the fixed rate assumption does not hold and more flexibility is needed.

For this purpose, our work has concentrated on the following issues. The accelerator algorithm is defined by means of a signal flow graph (SFG). The accelerator datapath is defined as a set of *application specific units* [6]. An ASU is a bit-parallel hardware operator, able to execute one or more micro-instructions. The micro-instructions are defined by subsets or *clusters* of the SFG. Each cluster corresponds to one micro-instruction, and the set of all clusters covers the complete SFG. This way, the accelerator algorithm corresponds to a *sequence* of micro-instructions.

An efficient interconnect network, consisting of pipeline registers, will take care of moving data from one ASU to the other without creating a communication bottleneck.

A simple and fast controller structure is defined that organizes the run-time sequencing of the micro-instructions on the ASU's.

Finally, a complete design flow, from algorithm specification to implementation is defined. For the synthesis, including pipelining and retiming of ASU components, we rely on

Manuscript received July 31, 1996.

P. Schaumont, B. Vanthournout, and I. Bolsens are with the Interuniversity Micro-Electronics Center (IMEC), Kapeldreef 75, B-3001 Leuven, Belgium.

H. De Man is with the Interuniversity Micro-Electronics Center and Professor at the Katholieke Universiteit Leuven, Belgium.

Publisher Item Identifier S 1063-8210(97)00734-8.

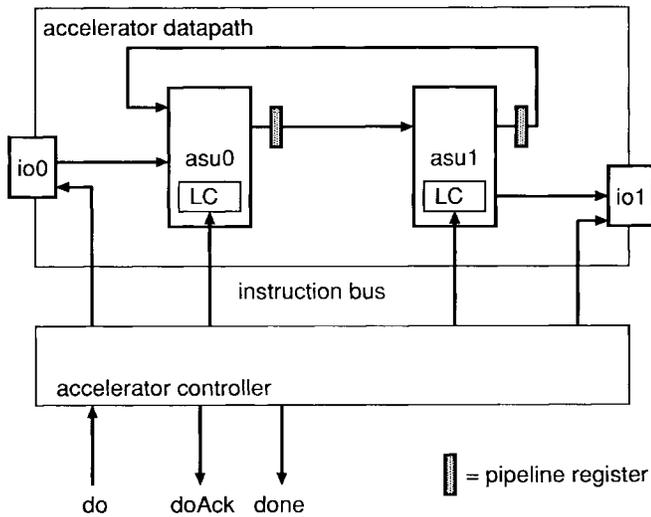


Fig. 2. Architecture of the accelerator processor.

existing data path synthesis tools and retiming tools [7], [8]. The design flow is integrated into a software system called DOLPHIN.

In the next sections, we will further detail the design steps to realize a data path architecture composed of ASU's. After an overview of the synthesis process, we examine the design steps that lead to the accelerator datapath in Section III-A and III-C. Following this, the architecture of the run-time controller is elaborated in Section III-D, and the operation of the controller is explained in Section III-E. Finally, the described architecture and method will be demonstrated in Section IV by synthesizing a quadrature mirror filter bank used for image encoding.

III. ACCELERATOR ARCHITECTURE

Fig. 2 shows the overall structure of the pipeline processor. Two parts can be distinguished: an *accelerator data path* and an *accelerator controller*.

The accelerator datapath consists of ASU's and interconnection buses with pipeline registers. All connections run point-to-point, and the use of latch registers that require read-write signals is avoided. This way, the *interconnection strategy* avoids that either a multiplexed bus or an interconnect storage register can become a pipeline bottleneck.

The accelerator controller is steered by the system level controller through a processor interface. It also generates control signals for the accelerator datapath, as well as strobe signals for the input and output buses on that datapath. The accelerator controller does not provide looping and branching support: it sequences micro-instructions to the datapath.

A. The Accelerator Data Path

The different steps taken in the design of the accelerator data path are illustrated in Fig. 3, along with a small example.

- 1) Using the SFG specification of the accelerator function, the accelerator data path is defined by *clustering* the SFG. A cluster can contain functional operations (additions, shifts, ...), or else signal flowgraph inputs and outputs. SDF semantics [9] are assumed. The clustered

graph must be a *directed acyclic graph* (DAG). This implies that loops in the SFG, such as algorithmic feedback loops, must be enclosed within one cluster.

- 2) These clusters are assigned to hardware. Clusters containing functional operations are assigned to ASU operators, while clusters containing input/output operations are assigned to input/output strobes. The SFG data precedences that cross the borders of the clusters define the *interconnection buses* of the accelerator data path.
- 3) The set of clusters assigned to one ASU define the ASU micro-instruction set and composition. It consists of a local controller (LC) and a bit-parallel data path.

The local controller handles micro-instruction decoding and local decision making. As a consequence, there is no global decision making and thus no condition evaluation circuitry in the accelerator controller.

The ASU bit-parallel data path is obtained using the CATHEDRAL-3 data path synthesis tools [7], [8], which are capable of mapping the SFG operations inside a cluster to a bit-parallel data path using sharing where possible.

The available bit-parallel operators and some corresponding SFG operations are listed in Table I. These operators are a library of parameterized descriptions that are instantiated in standard cells during synthesis.

Aside from ASU definition tools, retiming software is used to insert pipeline registers in the ASU data path. Also, netlist optimization tools [10], [11] are used to tune the instantiated bit-parallel operator to an application specific one.

The I/O timing behavior on the data and control ports of an ASU is known as the *timing view*. It is expressed as a number of clock cycles, representing the latency between ASU input consumption and output production.

- 4) To find the number of interconnection pipeline registers we proceed as follows. The *cluster latency* is expressed as the number of clock cycles needed to evaluate that cluster. Using the cluster latencies as operation lengths, and the (operator, micro-instruction) tuples as conflicts, the clustered graph is scheduled. The cluster latency is equal to the ASU latency incremented by one. The increment of one ensures that at least one pipeline register will be present on an interconnection bus between two ASU clusters. The maximum combinatorial delay, or *critical path length*, of the accelerator datapath will therefore comply the timing specs that were used for the individual ASU's. A list scheduling algorithm is used to perform the ordering. The cluster schedule is also used to model the micro-instruction sequence of the overall data path in a table, with one row per ASU and one column per clock cycle. Such a structure is called a *reservation table*, which is the basic data structure in the design of the accelerator controller.

B. Cluster Scheduling

The cluster scheduling algorithm applied influences both the controller specification (the reservation table) and the

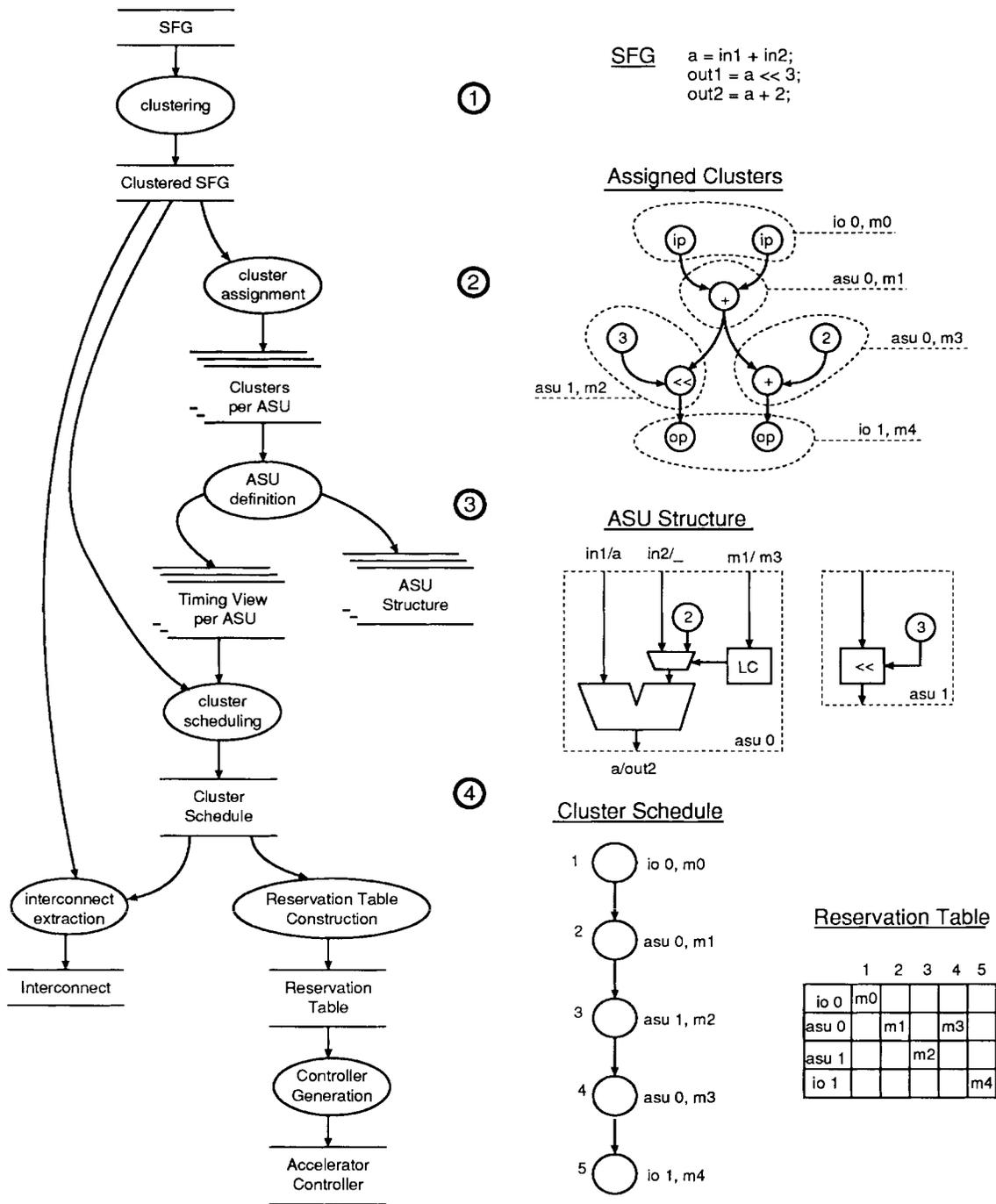


Fig. 3. Design flow for the accelerator processor.

interconnect specification (the number of pipeline registers on each interconnect bus).

We wish to distinguish between the scheduling as applied here to obtain the data path interconnect, and the scheduling needed to decide upon the DII at runtime. Most scheduling authors fix the DII at compile time, and apply sophisticated techniques [12]–[15] to arrive at a minimum cost solution for both interconnect and DII.

Unfortunately, these cannot be applied to the problem at hand—an accelerator requires the DII to be selectable at runtime. The controller that will be presented is able to do so, within the performance capabilities of the pipeline.

Therefore, we only consider scheduling as used to obtain the data path interconnect. Currently, a list scheduling algorithm is used. This approach only takes micro-instruction resource conflicts into account. In order to obtain a more optimal solution, one of two approaches can be followed.

- 1) Optimization for a *desired DII*. It has been shown [16] that an arbitrary DII or DII cycle from a pipeline can be obtained provided that performance of the pipeline is not exceeded, and the adequate number of data path interconnect delays are introduced.
- 2) Optimization for *interconnect cost*. Minimization of the number of pipeline registers residing on the interconnect

TABLE I
THE AVAILABLE BIT PARALLEL ASU OPERATORS

SFG operation	ASU operator	description
$+, -$ (binary)	ADDSUBFBB	adder-subtractor
*	MULTFBB	booth multiplier
delay	REGFBB	register
delay	REGFILEFBB	register file
&	ANDFBB	bitwise AND
	ORFBB	bitwise OR
\wedge	XORFBB	bitwise XOR
\ll, \gg (fixed)	hardwired	shift operator
\ll, \gg (variable)	SHIFTFBB	barrel shifter
==	COMPFBB	comparison
$>, <, >=, <=$	ADDSUBFBB	flag generation
== 0, == 1	ZERO_ONE_DETECFBB	detection of 0, 1
$z \leftarrow if \dots$	MUXFBB	signal selection
- (unary)	HADDERFBB	inversion
++	HADDERFBB	increment

buses can be done in polynomial time [17]. At that time, the DII rate can no longer be arbitrarily chosen.

These two methods have conflicting goals, and depending on the application one might prefer one or the other.

C. Clustering

The SFG clustering and assignment process, that leads to the definition of the data path operators, will be further detailed in this section. Current state of the art clustering strategies [18] indicate that there is no single unifying approach to obtain these clusters automatically. Rather, the ideal approach is believed to be a toolbox of functions that aid the designer in clustering an SFG.

Therefore, our current approach to SFG clustering and assignment for the accelerator processor is a manual process. In this section, we will identify the costs involved in this process and derive the guidelines that will steer it. It will be shown that, for the application area of DSP, small clusters introduce excessive multiplexing cost, and therefore that large clusters are preferable.

First, the hardware costs inside an ASU processor are identified. The hardware cost is primarily defined by the active silicon area. There are three sources of active area as follows:

- the data path cost, determined by the operator area needed to implement the SFG operations;
- the multiplexing cost, which is the multiplexer area needed to implement hardware sharing on an operator;
- the interconnection cost, consisting of the area of all pipeline registers that carry signals from one cluster to the next.

Through SFG clustering and cluster assignment the operation-operator binding is fixed.

- 1) The multiplexing cost and data path cost are determined by the set of clusters assigned to the same ASU.
- 2) The interconnection cost is expressed in the number of SFG signals crossing the boundary of the clusters, and the lifetimes of these signals. The latter is determined by the cluster schedule.

TABLE II
CELL AREA IN EQUIVALENT 2-NAND GATES

Cell	Function	Location	gates/bit
FADD1	Full Adder	data path	5.5
MUX2	2-Multiplexer	sharing	2.3
DFF	MS-flipflop	interconnect	4.6
LA	Latch	interconnect	3.5

Hardware sharing will only be effective if the savings in data path cost exceeds the extra cost introduced by the sharing. All the operations shown in Table I except for the delay operations are shareable. The delay operations are needed to implement algorithmic state, which is common in DSP algorithms.

We now show why we need a large *shareable* cluster size within the target accelerator processor architecture.

In Table II, the hardware cost of different operators is expressed in terms of equivalent two-input NAND-gates in the target technology. These figures were obtained by averaging the properties of two $0\mu 5$ standard cell CMOS libraries. The bit-parallel operators introduced by the data path synthesis tools are described in terms of these cells. For example, a ripple carry adder is a chain of FADD1 cells, while a multiplier is a matrix of FADD1 cells. The table also lists the multiplexing operator and the interconnect operators, the flip-flop and the latch. The proposed accelerator uses the edge-sensitive flip-flop on the interconnect.

Consider the SFG snippet of Fig. 4. It is assigned in two different ways. The leftmost one is intended to have no sharing, and has an area cost of 11 gates/b. The rightmost one shares the addition, but it has an area cost of 14.7 gates/b. Both implementations have the same performance: the rightmost implementation has half the critical path of the leftmost one, but it needs two clock cycles to perform the operation. We conclude that sharing is not useful for this SFG: the multiplexing cost exceeds the sharing gain.

If the operator area cost gets higher, as for example when using multipliers, sharing might provide area gain. The data path synthesis tools try to maximize the cluster subgraph

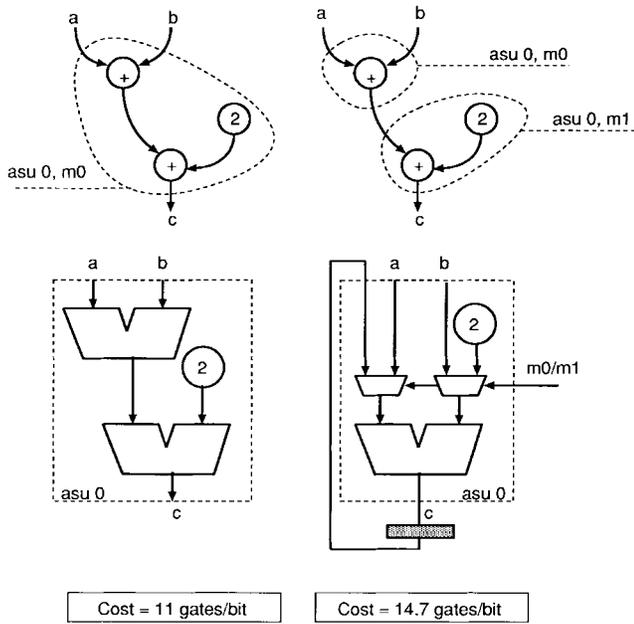


Fig. 4. Sharing cost example.

that can be shared among different micro-instructions, such that sharing area gain will exceed multiplexing cost. Given the nature of DSP algorithms however, this is not an easy task.

- Complex operations are often expanded to simple ones: A constant multiplication is implemented as an expansion of add-shift operations which have irregular structure. Such structures are difficult to share.
- Many operations like shift, bit-select, and bit-reverse are implemented hard-wired and have virtually nil operator size.
- In DSP algorithms, algorithmic delay is a key element. This delay cannot be shared, and cuts down the maximal cluster subgraph that can be shared.

We conclude that clustering is a process that should be done with care, and that the designer, who knows the application at hand, is in the best position to do it.

As Table II shows, one might be inclined to use a latch LA instead of a flip-flop DFF in order to gain interconnect area. It is not done for the following reason. Using latches to separate pipeline stages, and a double phase clocking scheme, only half of the pipeline stages are filled at maximum throughput [19]. To get the same performance as with edge triggered flip-flops, we need a dual latch between each pipeline stage. The figures in the table indicate that the dual-latch solution is no better than the flip-flop solution.

D. The Accelerator Controller

After a discussion of the data path synthesis process, we focus on the accelerator controller. The accelerator controller must perform ASU micro-instruction sequencing according to the cluster schedule, as represented in the reservation table.

In Fig. 5, the operation of the controller is illustrated by an example. The reservation table that was derived in Fig. 3 is

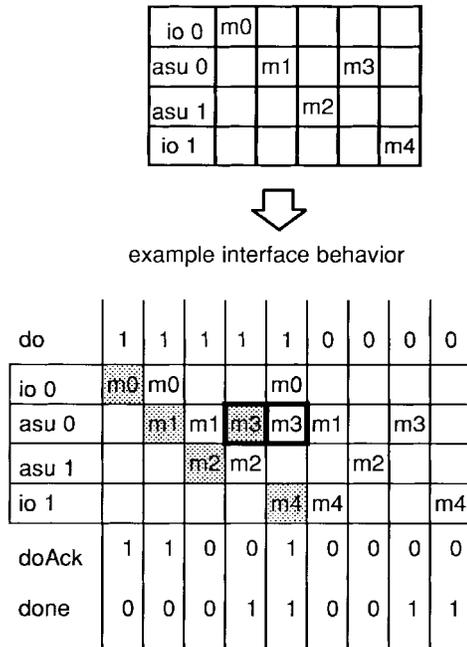


Fig. 5. The Processor Interface Behavior.

on top. Below, the processing of three SFG frames is shown in terms of the processor interface pins. Time runs from left to right, one clock cycle at a time.

The *processor interface* makes use of three signals *do*, *doAck* and *done*. The *do* pin is used to initiate the processing of one SFG frame, represented in the accelerator controller by one reservation table instance. When a *do* command is accepted, it means that hardware will be available to execute the schedule in reservation table during the next few cycles.

In the example the *do* pin is held high during five consecutive clock cycles. Acceptance of the *do* command is acknowledged through the *doAck* output. At the second clock cycle, a new reservation table instance can be interleaved with the first one. For the third and fourth cycle however, this interleaving fails and the *do* is not acknowledged. This failure originates in the hardware sharing from *asu 0* and is called a *pipeline conflict*. Thus, the accelerator controller takes care of two key functions:

- run-time scheduling of ASU micro-instructions and detection of conflicts;
- interleaving of accelerator-level instructions.

This leads to the accelerator controller hardware presented in Fig. 6. Three parts are discerned:

- the micro-instruction shifter;
- the conflict controller;
- the processor interface.

The *micro-instruction shifter* is used to store reservation table initiations. Each ASU micro-instruction bus or input/output strobe has a proper shift register corresponding to one row in the reservation table. A *start* signal loads one instance of the reservation table into the shift registers, in order to obtain the interleaving shown earlier.

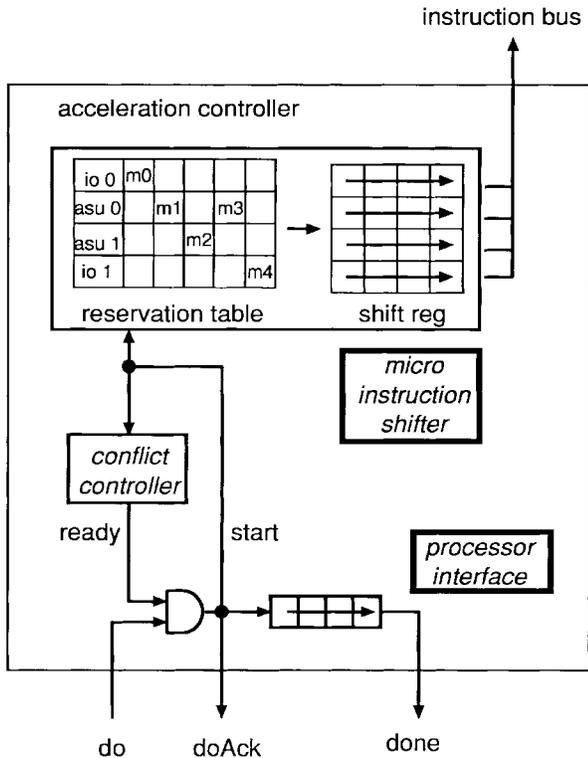


Fig. 6. The accelerator controller.

The start signal is also fed into the *conflict controller*. This is a hardware conflict model that signals occurring pipeline conflicts through the *ready* output. Whenever this output is high, a new reservation table can be interleaved in the micro-instruction shifter. When this output is low, interleaving is not possible. The start signal depends on two conditions:

- the user requests accelerator execution through the *do* pin of the processor interface;
- the conflict controller indicates the shift register controller is ready to accept a new initiation.

Therefore, the start can be derived out of the *do* and *ready* signals by means of an AND gate.

The two remaining processor interface signals are easily derived out of the start signal. The *done* output models the latency of the accelerator, and is obtained out of start through simple delay.

The processor interface makes both stand-alone and slave operation possible. For stand-alone operation, the *do*-pin is tied to a logical high. In this case, the processing rate is fixed by the conflict controller, and the processing of a frame will be initiated whenever there is no pipeline conflict occurring.

E. The Controller at Work

The conflict controller contains the core of the dynamic scheduling properties of the accelerator. A simple control strategy and architecture, which can be used as conflict controller, is due to Davidson [20]. His approach is based on dynamic modeling of data path resource conflicts.

The instances at which a conflict occurs after the *initiation* of a reservation table are called *forbidden latencies*. In the

example reservation table of Fig. 7, an initiation will introduce a pipeline conflict, due to *asu 0*, two cycles after this initiation.

An *initiation latency* is defined as the delay, in clock cycles, between two successive initiations. In order to satisfy the resource constraints, an initiation latency cannot equal a forbidden latency.

To achieve dynamic modeling, the pipeline conflicts are marked as indices in a bit vector, which is shifted right as time proceeds. This bit vector, which is called *collision vector*, is numbered right to left starting from one. Bit position *i* of this vector indicates whether a pipeline conflict occurs within *i* clock cycles. Hence, bit position one indicates whether a conflict occurs the next cycle, and thus whether a new initiation is possible at the next cycle.

The initial marking of the forbidden latencies found out of the reservation table results in an initial collision vector. Upon each new initiation, the initial collision vector is marked into the current collision vector.

Taking the initial collision vector, a state diagram can be constructed, with the states indicating initiation instances and edges carrying the initiation latencies. This state diagram is discussed in literature [21], [22]. The states are marked with a collision vector. The initial state carries the initial collision vector, and represents the moment just after initiation of the empty pipeline.

Out of the positions within a collision vector with zero bits, the initiation latencies can be derived. Out of the initial state in the example (state 10), a new initiation is possible already the next cycle. At that moment, the initial collision vector is shifted one position, corresponding to a one cycle delay. At the same time, the pipeline conflicts introduced by this new initiation are annotated in the collision vector by OR-ing the initial collision vector into the current one. This results in a new collision vector (state 11). Out of this state, an initiation latency of at least three cycles is required, as bit positions 1 and 2 are nonzero in the collision vector. This next initiation returns us to the initial state.

The state diagram models every valid pipeline state, and therefore any cycle within this state diagram is a valid schedule.

Using the initial collision vector, a simple hardware structure that generates the state diagram can be constructed. The collision vector is modeled by means of a shift register. Upon initiation, a new version of the initial collision vector is OR-ed into the current collision vector. This structure is used as the *conflict controller*.

The advantages of using this controller architecture are as follows.

- Static and dynamic schedules are available within the same controller architecture (corresponding to stand-alone and slave-mode operation). By using runtime conflict modeling, all possible schedules are supported.
- The controller has a regular structure, and is small and fast. It can be shown that careful design reduces the critical path to one gate delay.
- It allows parallel, pipelined execution of several SFG frames.

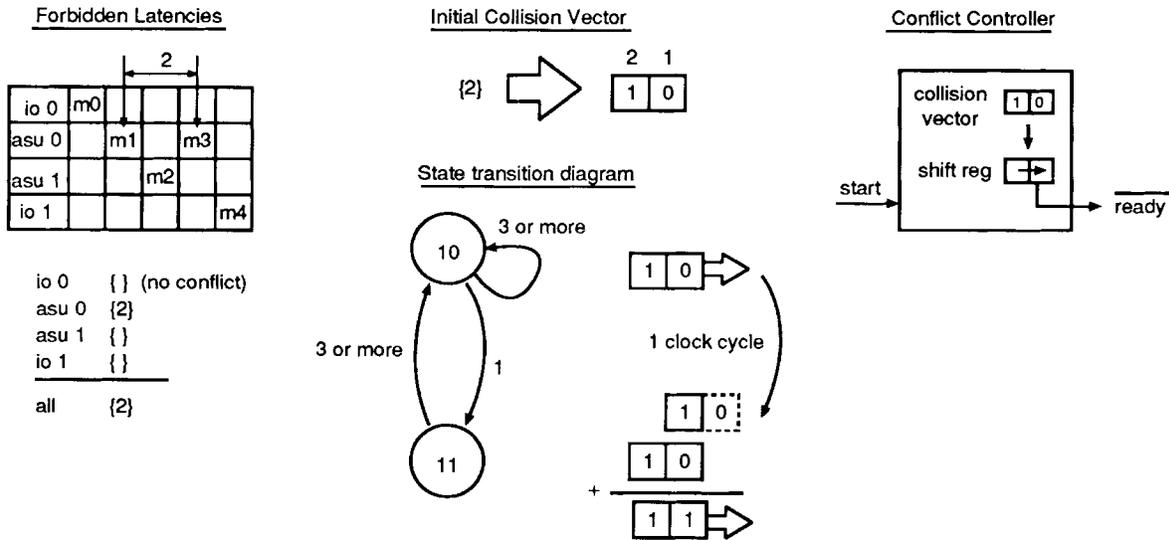


Fig. 7. Construction of the conflict controller.

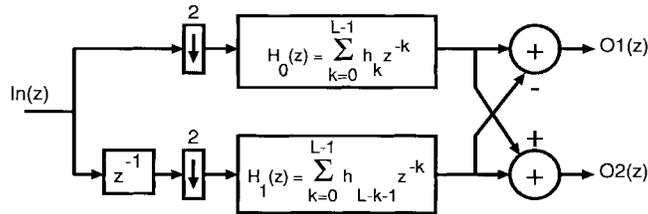


Fig. 8. A two-channel QMF bank.

IV. DESIGN EXAMPLE

Next, we present a synthesis environment called DOLPHIN that supports the synthesis of the accelerator processors. We will do this in course of the synthesis of an example accelerator.

The example concerns an accelerator typically used in image encoding algorithms: the two-channel *quadrature mirror filter* (QMF) bank. This filter decomposes a full-rate signal into two half-rate subsignals, such that the ensemble of subsignals can be used to reconstruct the original signal. The basic theory on QMF banks is exposed in [23], while [24] and [25] present more recent work.

Fig. 8 sketches the system architecture of a 2 channel QMF bank. The signal $In(z)$ is decimated into two streams, consisting of the even and odd $In(z)$ samples. These are fed into two digital filters $H_0(z)$ and $H_1(z)$. Two decimated signals $O_1(z)$ and $O_2(z)$ are produced. The filter impulse responses of $H_0(z)$ and $H_1(z)$ are mirror-symmetrical and related such that the reconstruction property holds on the output signals $O_1(z)$ and $O_2(z)$.

We wish to generate an accelerator processor for this filter. One input port is allocated to feed the input signal $In(z)$, and one output port extracts both $O_1(z)$ and $O_2(z)$. This way, the data rates on input and output port are balanced. We also want the accelerator processor to have minimal latency and maximal data throughput.

A behavioral description of the QMF filterbank in the SILAGE behavioral language [26] is shown in Listing 1 at the bottom of the next page. In this design example, we use the filter coefficients for the 16 tap FIR case described in [24], quantized to 14 b.

The accelerator is now synthesized according to the design flow shown in Fig. 3: Clustering of the SFG, definition of the ASU datapath, evaluation of the cluster schedule, and controller synthesis. The target implementation is a CMOS $0\mu 7$ standard cell technology.

For the purpose of clustering, the SILAGE description is converted to a graph format, upon which the user can interactively indicate the desired clustering.

The clustering is performed as follows and as indicated in Fig. 9.

- The input operation of $In(z)$ is assigned to I/O tuple (io 0, m0).
- The splitting of $In(z)$ into odd and even streams requires one delay operation, which is assigned to asu 0, m0.
- The filtering of the decimated streams has been grouped into a single cluster, assigned to the asu-microinstruction tuple (asu 1, m0). No sharing is introduced because of the maximal throughput requirement and irregular structure of the constant tap multiplications.
- The two output operations have been assigned to I/O tuple (io 1, m0) and (io 1, m1). This assignment

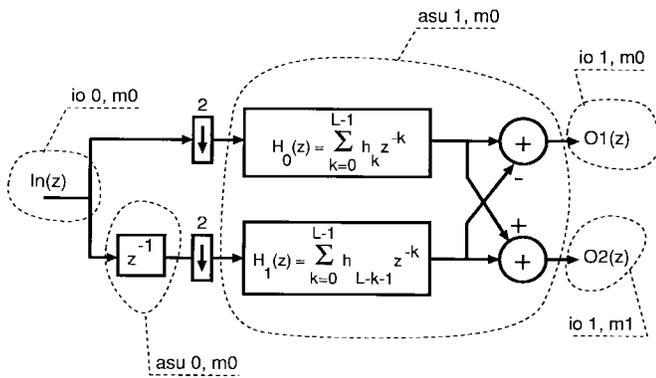


Fig. 9. Clustering of the the QMF bank.

expresses the sharing of the output signals $O_1(z)$ and $O_2(z)$ on the output port.

Following clustering, the ASU definition is done. The following steps are performed.

- Operator selection and operator netlist generation. This includes expansion of operations such as the expansion of the constant filter tap multiplications into canonical signed digit (CSD) add-shift operations.
- Mapping of the operator netlist to an abstract standard cell library.
- Standard cell netlist optimization including redundancy removal, retiming and buffering. The retiming tool allows to specify a desired target clock. For our example, a target clock of 27 MHz was chosen.

We next evaluate the cluster schedule using the timing view obtained from the ASU definition. The conflict model is shown on the left of Fig. 10. The presence of *decimate* operations in the input description is visible as multiplexing switches. The tool copes with this by an expansion step which enumerates

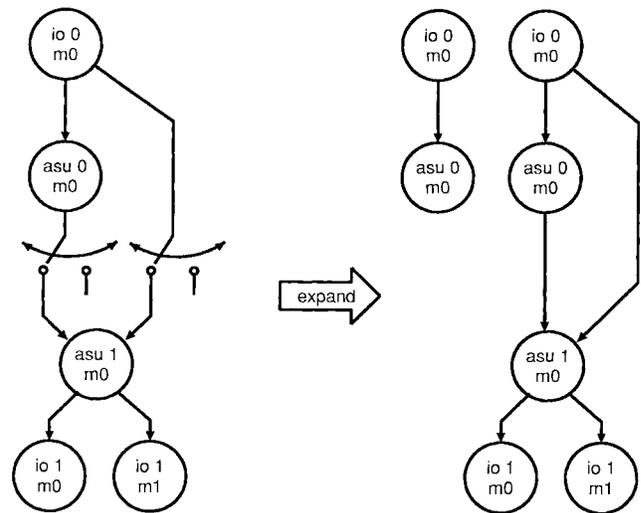


Fig. 10. The QMF multirate and expanded conflict model.

reservation table

io 0	m0	m0			
asu 0		m0	m0		
asu 1			m0		
io 1				m0	m1

collision vector = 11

state transition diagram

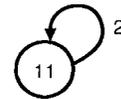


Fig. 11. The QMF reservation table.

all paths in the clustered graph. Since the graph is acyclic, this is a trivial operation. The resulting conflict model is shown on the right of Fig. 10. Using the schedule obtained by the list scheduler, a reservation table as shown in Fig. 11 is constructed. This data structure fixes the collision vector, and defines the state transition diagram. The shift register

Listing 1:

```
#define W fix<14,12>
#define W2 fix<18,16>

func main(in : W) outL, outH : W =
begin
  (phi1, phi1_discard) = decimate(in);
  (phi2, phi2_discard) = decimate(in@1);

  (h01, h11) = ( W2(W(11.111111101011B) * phi1) , W2(W(00.000000000110B) * phi2) );
  (h02, h12) = (h01 + W2(W(00.000001110010B) * phi1@1), h11 + W2(W(11.11111110100B) * phi2@1));
  (h03, h13) = (h02 + W2(W(00.000101110010B) * phi1@2), h12 + W2(W(11.111111010111B) * phi2@2));
  (h04, h14) = (h03 + W2(W(00.011110110001B) * phi1@3), h13 + W2(W(00.000110010011B) * phi2@3));
  (h05, h15) = (h04 + W2(W(00.000110010011B) * phi1@4), h14 + W2(W(00.011110110001B) * phi2@4));
  (h06, h16) = (h05 + W2(W(11.111111010111B) * phi1@5), h15 + W2(W(00.000101110010B) * phi2@5));
  (h07, h17) = (h06 + W2(W(11.11111110100B) * phi1@6), h16 + W2(W(00.000001110010B) * phi2@6));
  (h08, h18) = (h07 + W2(W(00.000000000110B) * phi1@7), h17 + W2(W(11.111111101011B) * phi2@7));

  outL = W(h08 + h18);
  outH = W(h08 - h18);
end;
```

TABLE III
CIRCUIT PROPERTIES OF THE QMF BANK ACCELERATOR

Component	Cells	Active Area (mm ²)	Critical Path (ns)
Data Path	1894	2.88	26.3
Controller + InterConnect	408	0.46	5.75

Technology: CMOS 0μ7 standard cells

controller as well as the conflict controller is synthesized out of it. Finally, the controller and datapath are interconnected.

The resulting circuit properties are summarized in Table III. The complete design flow is supported by a software script allowing a short edit-compile-test cycle from behavioral description to optimized netlist. For the example given, this cycle takes less than 15 minutes on a HP700.

V. CONCLUSION

In this paper, a strategy and a tool was presented to integrate data path synthesis and retiming tools into a system component design environment.

- The proposed strategy allows to generate small and efficient control for pipelined systems.
- In addition, an implementation of a system level data model is offered through a processor interface.
- Different schedules are available within one controller architecture, allowing nonmanifest data rates.

An integrated environment for the design of these accelerators, called DOLPHIN, was developed. Currently, it is being used for the design of accelerator parts in systems for videotelephony and compression, and advanced CATV modem parts.

ACKNOWLEDGMENT

The authors wish to thank K. Van Rompaey and S. Vernalde from IMEC for the constructive remarks during the writing of this paper. The work is also founded on the netlist optimization tools developed by L. Rijnders and Z. Sahraoui, on the library work of V. Derudder, and on the test work of M. Wouters, all from IMEC, Leuven, Belgium.

REFERENCES

- [1] P. Pirsch, N. Demassieux, W. Gehrke, "VLSI architectures for video compression," *Proc. IEEE*, vol. 83, Feb. 1995.
- [2] K. S. Hwang, A. E. Casavant, "Scheduling and Hardware Sharing in Pipelined Data Paths," in *Proc ICCAD '89*, Nov 1989, pp. 24–27.
- [3] H. S. Jun, S. Y. Hwang, "Design of a pipelined datapath synthesis system for digital signal processing," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 292–303, Sept. 1994.
- [4] V. G. Oklobdzija, "Issues in CPU coprocessor communication and synchronization," *North-Holland Microprocessing and Microprogramming*, vol. 24, pp. 695–700, 1988.
- [5] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 356–370, Mar. 1988.
- [6] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications," in *Proc. DAC91*, San Francisco, CA, 1991, pp. 597–602.
- [7] S. Note, F. Catthoor, G. Goossens, and H. De Man, "Combined hardware selection and pipelining in high-performance data-path design," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 413–423, Apr. 1992.

- [8] S. Vernalde, P. Schaumont, I. Bolsens, H. De Man, and J. Frehel, "Synthesis of high throughput DSP ASIC's using Application specific data paths," *DSP & Multimedia Technol.*, vol. 3, pp. 13–21, June 1994.
- [9] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, pp. 24–35, Jan. 1987.
- [10] Z. Sahraoui, L. Rijnders, J. Vanhoof, I. Bolsens, and H. De Man, "Area optimization of bit-parallel custom data paths," in *IFIP Workshop on Logic and Architecture Synthesis*, Grenoble, France, Dec. 1993.
- [11] L. Rijnders, Z. Sahraoui, P. Six, and H. De Man, "Timing Optimization by bit-level arithmetic transformations," in *European Design Automation Conf.*, Brighton, England, Sept. 1995.
- [12] C. T. Hwang, Y. C. Hsu, and Y. L. Lin, "PLS: A scheduler for pipeline synthesis," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 12, pp. 1279–1286, Sept. 1993.
- [13] P. G. Paulin and J. P. Knight, "Force directed scheduling in automatic data path synthesis," *Proc. 24th Design Automation Conf.*, Miami Beach, FL, July 1987, pp. 195–202.
- [14] K. Van Rompaey, "Functional Requirements for an extended scheduling tool," IMEC ESA-SCADES Rep. P50259-IMRP-0009.
- [15] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. Computer-Aided Design*, vol. 39, pp. 464–475, Apr. 1995.
- [16] A. El-Amawy and Y. C. Tseng, "Maximum performance pipelines with switchable reservation tables," *IEEE Trans. Comput.*, vol. 44, pp. 1066–1069, Aug. 1995.
- [17] X. Hu, S. C. Bass, and R. G. Harber, "Minimizing the number of delay buffers in the synchronization of pipelined systems," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 13, no. 12, pp. 1441–1449, Dec. 1994.
- [18] W. Geurts, "Synthesis of accelerator data-paths for high-throughput signal processing applications," Doctoral dissertation, ESAT/EE Dep., K.U. Leuven, Belgium, Mar. 1995.
- [19] R. F. Sproull, I. E. Sutherland, and C. E. Molnar, "The counterflow pipeline processor architecture," *IEEE Design Test Comput.*, Fall 1994 issue, pp. 48–59.
- [20] E. Davidson, L. Shar, A. Thomas, and J. Patel, "Effective control for pipelined computers," in *Proc. IEEE COMPCON 75*, New York, NY, 1975, pp. 181–184.
- [21] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: Hemisphere, 1981.
- [22] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *ACM Comput. Surveys*, vol. 9, no. 1, pp. 61–101, Mar. 1977.
- [23] P. P. Vaidyanathan, "Quadrature mirror filter banks, M-band extensions and perfect-reconstruction techniques," *IEEE ASSP Mag.*, pp. 4–20, July 1987.
- [24] A. Divakaran and W. A. Pearlman, "A closed form expression for an efficient class of quadrature mirror filters and its FIR implementation," *IEEE Trans. Circuits Syst. II*, vol. 43, no. 3, pp. 207–219, Mar. 1996.
- [25] J.-F. Yang, D.-Y. Chan, Y.-B. Chen, "Fast and Low Roundoff Implementation of Quadrature Mirror Filters for Subband Coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, no. 6, pp. 524–532, Dec 1995.
- [26] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man, "DSP specification using the SILAGE language," in *Proc. IEEE. Int. Conf. Acoust., Speech, Signal Processing*, 1990, pp. 1057–1060.



Patrick Schaumont was born in Gent, Belgium, on October 28, 1966. In 1988, he received the electronics and telecommunications engineering degree from the Industriële Hogeschool van het Rijk, Gent, Belgium. In 1990, he received the M.S. degree in informatics from the Rijksuniversiteit Gent, Belgium.

Since 1992, he has been with the VLSI Systems Design Group of IMEC, Leuven, Belgium, working on design automation for medium and high throughput digital VLSI systems. Currently, he is member of the High Speed Modem group working on the design of a broadband access network modem. His main research interests are in digital signal processing for telecommunications and high throughput VLSI implementations.



Bart Vanthournout was born in Ostend, Belgium, on September 30, 1967. He received the electrical engineering degree from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1990.

Subsequently, he joined the VLSI Systems Design Group of IMEC, Leuven, Belgium, and was involved in developing scheduling and assignment techniques for VLIW (Very Large Instruction Word) controllers aimed to ASIC implementations. In 1994, he worked on design methodologies for DSP accelerators. Since 1995, he has been a member

of the Hardware and Software Systems group, developing a synthesis and simulation environment for Hardware and Software codesign. His main interests are complex DSP applications and their requirements on design automation.



Ivo Bolsens was born in Wilrijk, Belgium, on November 1958. He received the electrical engineering and Ph.D. degree from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1981.

In 1984, he joined the IMEC Laboratory, where he started doing research on the development of knowledge-based verification for VLSI circuits, exploiting methods in the domain of artificial intelligence. In this context, he introduced functional programming, using Lisp, and object oriented programming, using Smalltalk. In 1989, he became

responsible for the application and development of the Cathedral-2, and later the Cathedral-3, architectural synthesis environment. He was also heading the application projects that produced the first silicon generated by these software environments. In 1993, he became Head of the Applications and Design Technology group, focusing on the development and application of new design technology for mobile communication terminals. In this context, he was responsible for the implementation of a programmable spread-spectrum transceiver for satellite communications. Since 1994, he has been heading a European Network on "VLSI design technology for high speed and mobile communication systems". In 1995, he became Director of IMEC's VLSI Systems and Design Methods division.

Dr. Bolsens was the recipient of the Darlington Award of the IEEE Circuits and Systems Society with the citation: "Best Paper published by the IEEE CAS Society that bridges the gap between theory and practice," in 1986. At the 1991 International Conference on CAD, he received a Distinguished Paper Citation. In 1993, he received a Best Circuit Award from the EUROASIC-EDAC Conference. Since 1981, he was a member of the CAD group at the ESAT laboratory of the Katholieke Universiteit Leuven, Leuven, Belgium, where he was working on the development of an electrical verification program for VLSI circuits and on mixed mode simulation.



Hugo J. De Man (M'81-SM'81-F'86) was born in Boom, Belgium, on September 19, 1940. He received the electrical engineering degree and the Ph.D. degree in applied sciences from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1964 and 1968, respectively.

In 1968, he became a Member of the Staff of the Laboratory for Physics and Electronics of Semiconductors at the University of Leuven, Leuven, Belgium, working on device physics and integrated circuit technology. From 1969 to 1971, he

was at the Electronic Research Laboratory, University of California, Berkeley, as an ESRO-NASA Postdoctoral Research Fellow, working on Computer-Aided Device and Circuit Design. In 1971, he returned to the University of Leuven as a Research Associate of the NFWO (Belgian National Science Foundation). In 1974, he became a Professor at the University of Leuven. During the winter quarter of 1974-1975, he was a Visiting Associate Professor at the University of California, Berkeley. He was an Associate Editor for the IEEE JOURNAL OF SOLID-STATE CIRCUITS from 1975 to 1980 and was European Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN from 1982 to 1985. From 1984 to 1995, he was Vice-President of the VLSI systems design group of IMEC (Leuven, Belgium), where the actual field of this research division is design methodologies for integrated systems for telecommunication. Examples are spread spectrum and ATM components design. Research of this group has been at the basis of the EDC-Mentor Graphics DSP-Station. Since 1995, he has been a Senior Research Fellow of IMEC responsible for research in system design technologies.

Dr. De Man received a Best Paper Award at the ISSCC of 1973 on Bipolar Device Simulation and at the 1981 ESSCIRC Conference for work on an integrated CAD system. In 1986, he received (together with L. Claesen) the Best paper Award in CAD from the ICCD-86 Conference and in 1987 for best publication in 1987 in the *International Journal of Circuit Theory and Applications*. In 1989, he received the Best Paper Award at the Design Automation Conference. He is a corresponding member of the Royal Academy of Sciences, Belgium, and a member of the Royal Flemish Engineering Society (KVIV).