Interprocessor Communication in Synchronous Multiprocessor Digital Signal Processing Chips

JAN DECALUWE, JAN M. RABAHEY, MEMBER, IEEE, JEF L. VAN MEERBERGEN, MEMBER, IEEE, AND HUGO J. DE MAN, FELLOW, IEEE

Abstract—This paper deals with interprocessor communication in synchronous multiprocessor DSP chips, the kind of systems that are synthesized by the Cathedral II silicon compiler. A model for the data flow between two processors is presented. A number of architectural possibilities are discussed. Key concepts are the “double-buffered” memory cell and an extended method of pointer addressing. This method leads to the definition of “Once In, Once Out” communication, as opposed to conventional FIFO buffering. The minimization of the buffer size by “skewing” the operation of the processors is worked out for specific important types of communication. The proposed techniques have been implemented in a synthesis tool which is part of the Cathedral II silicon compiler. The practical significance of the work is illustrated with several examples.

1. INTRODUCTION

The goal of the Cathedral II silicon compiler, as presented in [1]-[3], is to synthesize single chip solutions for a subset of DSP algorithms. The intended subset consists of complex algorithms in the 100 kbyte/s range, typically related to speech processing, digital audio, and telecommunication. Each processor is dedicated to the DSP subfunction to be realized. As a consequence of the “divide and conquer” philosophy used in Cathedral II, the processors operate relatively independently of each other. More specifically, this means that there is no direct information exchange between their controllers. This implies that the interprocessor communication happens fully synchronously. Moreover, there has to be a supervising controller that provides start pulses to the processors and possibly other signals regulating the global data traffic in the system. A global view on the target architecture is shown in Fig. 1.

The Cathedral II architecture is particularly suited to block oriented algorithms. Each processor takes as input a block of data words and passes result blocks to other processors. As a consequence, memory buffers have to be placed in between processors. The interprocessor communication area cost might therefore well become dominant over the cost of the processors themselves. The cost factors are not only the memory itself, but also the addressing logic and the routing between the blocks that are involved. Cathedral II systems will be another illustration of the VLSI “law” that communication is expensive while computation is cheap.

Fig. 1. Global view on the Cathedral II target architecture.

Thanks to the Cathedral II approach, the microcode of the processor can be scheduled independently [4]. The design of an optimal interprocessor communication system comes only after the scheduling task. On the other hand, it is perfectly possible and even very likely that an interprocessor communication synthesis tool would propose reiterations of the data path synthesis [5] and/or the scheduling steps. Obviously, such a tool should also take advantage of the degrees of freedom that do not cause such iterations.

In this paper a study will be made on how the interprocessor communication cost may be minimized. However, the problem will be simplified to unidirectional communication between two processors. The motivation for this is that all practical multiprocessor DSP systems that we have been considering were linear arrays of processors, with a unidirectional data flow through the array. In such systems, the optimal interprocessor communication solution may be found by optimizing the communication between each pair of communicating processors, individually, and simply combining the results. Although some work has been done with respect to more complex structures of communicating processors, we decided not to present it in this paper, thereby only emphasizing the methods which should be directly applicable in practice.

In Section II, a quite general mathematical model will be introduced for unidirectional communication between two processors. Throughout the paper, this model will provide the formal basis for definitions and lemmas. Section III deals with architectural considerations with regard to the communication problem. The main idea is to use less general but also less expensive architectures in case the data flow properties allow it. Key concepts are the double-buffered memory cell and an extended method of pointer addressing. In Section IV, this method leads to the definition of Once In, Once Out communication, as opposed to simple FIFO buffering. The main goal of this section is the minimization of the buffer size by “skew-
ing” the operation of the processors. For all fully pointer addressable buffers, which form a superset of the Once In, Once Out buffers, algorithms are presented that determine the minimal necessary buffer size. Section V describes a prototype program for the synthesis of the interprocessor communication system, as part of the Cathedral II silicon compilation environment. In Section VI, several examples are shown that illustrate the proposed solutions. Finally, in Section VII, the paper is summarized and conclusions are drawn.

II. A MODEL FOR UNIDIRECTIONAL COMMUNICATION BETWEEN TWO PROCESSORS

Suppose that a processor $prc_P$ “produces” values of variables and that these values are “consumed” by processor $prc_C$. We define a set $\mathcal{V}$ that contains all those variables:

$$
\mathcal{V} = \{ v \mid \text{a value for } v \text{ is produced by } prc_P \\
and consumed by } prc_C \}.
$$

We assume that all values are produced (consumed) over one bus, one at a time. In general, the production (consumption) bus consists of the following parts:

- a data bus that contains the value of a variable;
- an address bus that specifies the variable;
- one wire for the signal that specifies the cycle in which the variable is produced (consumed).

In DSP algorithms, the programs in the processors are repetitively executed in time. One program execution is called a frame. We assume that the frame length (expressed as a number of clock cycles) is a system parameter $F$:

$$
F \text{ is the number of clock cycles in one frame.}
$$

The system’s operation can be viewed as an infinite loop over frames, with an incrementing loop index $f$:

$$
f \text{ is the frame number.}
$$

The production and consumption of values occurs on frame basis: a variable gets a unique value during each frame and the values that are produced during frame $f$ in $prc_P$ are consumed during frame $f$ in $prc_C$. Usually, frame $f$ in $prc_P$ starts at a different moment in time than frame $f$ in $prc_C$, e.g., because the value of a variable cannot be consumed before it has been produced. It is not allowed to produce and consume a value in the same cycle because of timing considerations. We assume that a value is only produced once during a frame, but we do not exclude that it is consumed more than once. However, the number of consumptions is assumed to be fixed, i.e., independent of the frame. These assumptions are general enough to cover the vast majority of the practical cases, although other cases are easily conceivable: e.g., a global initialization of all variables followed by partial (conditional) updating would be an example of a nonfixed number of productions. There is no fundamental reason why such a kind of data flow could not be included in the model, but we will not do so because it would considerably obscure the notations for a doubtful profit.

The global data flow in the system can now be modeled as follows.

For all $f$ associate with every $v \in \mathcal{V}$ a production time $P_v(f)$ that denotes the cycle, relative to the start of frame $f$ in $prc_P$, on which the value for variable $v$ is produced, and a set of consumption times $\mathcal{C}_v(f) = \{ C_{v,k}(f) \}$ where $C_{v,k}(f)$ denotes the cycle, relative to the start of frame $f$ in $prc_C$, on which the value for variable $v$ is consumed for the $k$-th time; $k = 1, 2, \ldots, m_v; m_v \in \mathbb{N}_0$.

The interprocessor communication synthesis problem is the following.

Design a communication system that “transforms” the set of production times $\Pi(f) = \{ P_v(f) \mid v \in \mathcal{V} \}$ into the set of sets of consumption times $\Gamma(f) = \{ \mathcal{C}_v(f) \mid v \in \mathcal{V} \}$, for all $f$, resulting in a minimal silicon area.

This communication system has parts that are directly controlled by $prc_C$ or $prc_P$, and that therefore are considered to be part of the processors, but also an additional part, located “in between” the production bus and the consumption bus. In the processors there may be ACU’s and bus drivers, in between them there will be memory and decoders or pointer shift registers.

From the formulation and the modeling of the problem, we can already identify the degrees of freedom that will play a role in the search toward an optimal solution. First, good architectures have to be chosen. Second, note that there is no relation specified between the starts of the two processors. Therefore, we have to investigate what the optimal “skew” is between the operation of processors.

A further step could be trying to find more optimal sets $\Gamma(f)$ and $\Pi(f)$. This would involve explicit actions into the scheduling of the program that is executed in the processor. Such techniques will not be studied in this paper.

The data flow in DSP systems has often interesting properties. We will conclude this section by defining two of these properties formally, using the notations of the model. The definitions will be given for the consumption, for the case of production they are analogous.

**Definition 1**—Deterministic Consumption: The consumption of the values of the variables is deterministic if there exists an ordering relationship between all elements $C_{v,k}(f)$ of the subsets of $\Gamma(f)$ that is independent of $f$.

This is a very practical definition: the nondeterministic case occurs only when it depends on conditions which variable is accessed next.

**Definition 2**—Invariant Consumption: The consumption of the values of the variables is invariant if $\Gamma(f) = \Gamma$, for all $f$.

Invariant consumption (production) is a property of all unconditional programs. Obviously, if consumption (production) is invariant, it is also deterministic.
III. Architectures for Interprocessor Communication

A. A General Solution: Switched RAM’s

The most general solution for frame based (or block oriented) communication between two processors is the Switched RAM’s. Quite naturally, this was the first proposal at the definition step of the target architecture of Cathedral II [6]. Two conventional RAM’s are used, and during a frame each of the processors communicates with a different memory (Fig. 2). Before starting the next frame, the connections between RAM’s and processors are switched. This configuration allows for the most general kind of data flow one can think of. Bidirectional communication can be handled, and there are no restrictions on the way in which variables are written and read.

This solution is very powerful, but also very expensive. It gives rise to a complex floor-planning problem because besides the two processors, 6 strongly interconnected blocks have to be placed and routed: 2 RAM’s and 4 multiplexers. A floor-planning exercise has been done for the case in which the data block consists of 32 words of 16 bits. The result is shown in Fig. 2. A lot of area is occupied by routing. This has several more important disadvantages besides the area cost. The capacitance associated with long metal tracks may be dominant with respect to timing delays and the power consumption of the system. As these tracks are only generated during the last step of the chip synthesis, undesirable design iteration might be necessary.

B. Double-Buffered Memory

Although one would like to avoid the high cost of the Switched RAM’s, it seems impossible to find a cheaper solution that has the same generality. To put it another way: cheaper solutions will have to give in on generality. But as long as such architectures can support data flow mechanisms that are likely to occur in multiprocessor DSP systems, there is nothing wrong with that. In this paper, two such architectural concepts will be proposed: double-buffered memory in this subsection and pointer addressing in the following subsection.

The idea for double-buffered memory, as well as its name, has been taken over from [7]. In this reference, a double-buffered memory is used in an FIFO configuration at the input of a DSP chip. This memory architecture is indeed very well suited for interfacing an (asynchronous) host with a block processing slave chip. Although in this paper our primary interest will be the use of double-buffered memory for on-chip interprocessor communication, it will also be the ideal basis for the I/O interface of Cathedral II chips.

The principle of double-buffered memory is entirely reflected in its basic cell. Actually, this cell contains two memory locations in a master–slave configuration. Writing in the master part can occur completely independently from reading the slave part. Before a new frame starts, information will be transferred from master to slave.

The resulting structure (Fig. 3) should be compared to the Switched RAM’s (Fig. 2). In both alternatives, the actions of the processors are kept independent by complete data buffering. But the final transfer of information from one processor to the other occurs by very different mechanisms: in the former case by internally updating the slave part of each memory cell, in the latter case by switching the external connections between the RAM’s and processors. In other words, the desired functionality is obtained by routing and “glue logic” in the Switched RAM’s case while it is an inherent property of the structure of double-buffered memory.

The implications on floor-planning are quite dramatic (Fig. 3). Now only one block has to be placed in between the processors. The floor plan does not contain the many long metal tracks that were previously observed in the floor plan of the Switched RAM’s (Fig. 2). The important disadvantages of these long routing tracks, as discussed earlier, are therefore largely avoided. Note that the area occupied by memory is almost the same in both cases, but that the overhead due to the “glue logic” and the routing in the Switched RAM’s case is almost as important as the memory area itself. Although the shown floor plans are not optimized (they have been generated with an experimental floor planner), it is clear that it is much more difficult to find an elegant floor-planning solution for the Switched RAM’s. Consequently, the Switched RAM’s alternative will waste much more silicon area than the double-buffered memory solution. The floor-planning advantages of double-buffered memory over Switched RAM’s come from the fact that data and address busses are now customized to one task: input or output, and read address or write address. At the same time this leads to the following restriction.
Restriction 1—Restriction on the Use of Double-Buffered Memory: Communication between two processors can be accomplished by a double-buffered memory iff the data flow is unidirectional.

As mentioned earlier, unidirectionality is a property of many DSP systems and therefore it was a basic assumption in the model of Section II. On the other hand, even if the "global" data flow is unidirectional, a double-buffered memory cannot be used as a "background memory" for a processor, in contrast with a Switched RAM's architecture, because this implies "local" bidirectional communication. As a last remark, note that "global" bidirectional data flow may always be split into two unidirectional data flows.

C. Pointer Addressing

While the previous subsection proposed a nonconventional type of memory, this subsection will introduce a nonconventional addressing method. Typically, memories are addressed by calculating an address in an address computation unit (ACU) and transferring it over an address bus to a decoder which then raises the appropriate word select line.

For the special case of FIFO memories, another addressing method is in common use. For both the read and the write select sequence, a shift register is placed on top of the memory array. A "running one" points at a certain memory location and is shifted further after each memory access. This method, further on referred to as "pointer addressing," has the following advantages:

- no addresses have to be stored and calculated in a separate ACU module,
- there is no address bus, thereby avoiding routing and simplifying the floor-planning task,
- the decoder, which is inherently an irregular structure (and which sometimes, depending on the design style, dissipates a lot of power), is replaced by a regular shift register row.

Although FIFO data flow is very important, one observes also many more complex kinds of data exchange in DSP systems. Pointer addressing would therefore seem to have only a very specific and limited use. We will, however, show that this conclusion is too pessimistic. The shift register cells may just be regarded as storage units. Instead of only shifting the pointer through adjacent cells, it can be moved from one location to any other over a routing track. By allowing this, the possibility of using pointer addressing for a read or write sequence depends solely on the sequence itself, and not on both sequences. Fig. 4 illustrates the idea of using routing to determine the sequence. Note that this routing is added at the module generation level, and not during the final floor-planning step. Therefore, possible timing problems with long tracks can be anticipated and taken care of at the module-generator level, instead of causing floor-planning and design iterations.

The final restriction on the use of pointer addressing in this broader sense is the following.

Restriction 2—Restriction on the Use of Pointer Addressing: A necessary and sufficient condition on the use of pointer addressing for the production of variables is that the production is deterministic. Necessary and sufficient conditions on the use of pointer addressing for the consumption of variables are that the consumption is deterministic, that \( n_v = M \) for all \( v \), and that the consumption sequence can be divided into \( M \) equal subsequences.

Because of the determinism, the address bus controlled by the processors can be replaced by a local control unit next to the interprocessor buffer. Thanks to the additional conditions, an elegant implementation with shift registers and a routing network is possible. In essence, these conditions make sure that, during system's operation, addressing occurs as a repetition of a fixed sequence in which all variables are accessed exactly once.

D. Combining the Concepts

In this subsection, the possibilities of combining several architectural alternatives will be investigated. At this point, it is convenient to think about the interprocessor communication system consisting of a memory part and of an addressing part. The basic memory cells under consideration are the conventional RAM cell, the double-buffered memory cell (introduced in Subsection B), and yet another cell that is typically used in FIFO's. This latter cell has strong similarities with the double-buffered cell, because it has a separate read and write port. The difference is that there is only one memory location per cell. For these reasons, it will be called "single-buffered memory cell." As far as addressing is concerned, we distinguish the ACU/address bus/decoder option and pointer addressing.

The possibilities of combining these concepts can be summarized in the following points, assuming unidirectional data flow:

- Switched RAM's cannot be combined with pointer addressing, unless a considerable hardware overhead is accepted. The reason is that the consumption and production busses are alternatively used by different processors. So when pointer addressing would be used for one or both sequences, switches would have to be placed on the word select lines. Essentially, the decoded address would be switched instead of the coded one, and therefore the corresponding area cost would be much larger. (To be precise, this overhead could be avoided if the production and the consumption sequences would be identical, but in that case one would rather use a specific FIFO structure.)
- The combination of double-buffered memory and pointer addressing is straightforward because the production and the consumption busses are in this case nicely separated. As soon as Restriction 2 holds for any of the two sequences, pointer addressing may be used for it.
- The previous conclusion is also true for single-buffered memory.
E. Electrical and Layout Issues

In this subsection we will give an overview of the electrical and layout characteristics of the basic memory cells that have been mentioned in this paper. These aspects are important because they may influence architectural decisions. The designs have been made in a 3 μm nwell CMOS technology with double metallization.

The transistor schemes of the RAM cell (abbreviated by sRAM), the single-buffered memory cell (SiBu) and the double-buffered cell (DoBu) are shown in Fig. 5. All designs are static. The SiBu cell has the same transistor schematic as the sRAM cell, but other functions have been assigned to transistors and wires. The cell has a separate read and write port, and one of the cross-coupled inverters has become a weak ‘bleeder’ inverter. The DoBu cell can be viewed as a concatenation of two SiBu cells. The SiBu and DoBu cells have a single-ended output, whereas the sRAM provides a differential output. But although a differential output is preferable from an electrical point of view, the use of a simple well-sized inverter as sense amplifier is often the best compromise between speed, area, and power consumption, for relatively small RAM sizes (e.g., up to 256 words). In that case, even the sRAM cell is used in a single-ended configuration.

The layouts of the memory cells are shown in Fig. 6. The SiBu cell is somewhat bigger than the sRAM cell because of the asymmetry between the cross-coupled inverters. Not surprisingly, the DoBu cell is approximately two times as big as the sRAM cell. Note also the unfavorable shape of the DoBu cell. Methods that change the memory plane shape, like postdecoding or ‘skewing,’ will even sooner be necessary than in the case of conventional RAM’s.

Table I summarizes the electrical and layout characteristics of the memory cells.

<table>
<thead>
<tr>
<th></th>
<th>sRAM</th>
<th>SiBu</th>
<th>DoBu</th>
</tr>
</thead>
<tbody>
<tr>
<td>%or type area</td>
<td>static</td>
<td>static</td>
<td>static-static</td>
</tr>
<tr>
<td></td>
<td>40.5 ± 53 μm²</td>
<td>42 ± 60 μm²</td>
<td>42 ± 105 μm²</td>
</tr>
<tr>
<td>output I/O bus</td>
<td>differential</td>
<td>common</td>
<td>single-ended</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>separate</td>
</tr>
</tbody>
</table>

F. Concluding Remarks on Architectures

In this subsection, architectures for interprocessor communication in Cathedral II have been presented. It has been shown that for the synthesis of single chip multiprocessor DSP systems it may be advantageous to change our ‘traditional’ way of thinking about memories and data access.

Traditionally, memory chips have a common I/O bus in order to minimize pin count. On the other hand, on-chip memories for communication between DSP processors might better have a separate input and output bus because such a structure is closer to the behavior of the system. This has been exemplified in the discussion on double-buffered memories.

Second, data access is classically divided into sequential and random access. In a silicon compilation environment, however, it is probably better to make the distinction between a data access sequence that is, to a certain extent, predictable at (silicon) compile time, and a data access sequence that is unpredictable at compile time. For when it is predictable, less expensive ways of accessing the data might be possible than by calculating and decoding an address. The extended method of pointer addressing, that has been introduced in this section, is a good example.

IV. Minimizing Buffer Size by ‘Skewing’

A large part of the interprocessor communication system will be memory. Therefore, an important method of optimization is to make the memory requirements as small as possible. In this section, we are still only considering unidirectional communication between two processors.

A. The Memory Size

It is almost trivial to derive an upper bound for the memory size.

Lemma 1—Upper Bound on the Memory Requirements: The smallest general upper bound for the number of memory locations in the interprocessor communication memory is 2N where N is the cardinality of V, or N = #V.

Proof: First we prove that 2N is an upper bound. Therefore, we construct the system in a way it is guar-
anted to work. Let frame \( f + 1 \) start in \( prc_p \) at the same moment as frame \( f \) in \( prc_C \). Therefore, all values of the variables \( v \in \mathbb{N} \) that \( prc_C \) has to consume in frame \( f \) have certainly been produced, and \( N \) memory places are needed to store them. While \( prc_C \) is consuming these values, \( prc_p \) is already producing the values for the next frame \( f + 1 \), and these are stored into \( N \) more places. When frame \( f + 1 \) starts in \( prc_C \) and frame \( f + 2 \) in \( prc_p \), the \( N \) values of frame \( f \) are no longer needed by \( prc_C \), so for the storage of the values of frame \( f + 2 \) no additional memory is required. We still have to prove that there is no smaller general upper bound. This can be done very easily by giving an example in which \( 2N \) memory places are really necessary. We leave this to the reader.

Architectures like Switched RAM’s and double-buffered memory always use \( 2N \) memory locations and are oriented toward a system’s operation as described in the proof of the Lemma. As a consequence, other architectures have to be used when an attempt is made to reduce the memory size. The most appropriate architecture is the so-called “single-buffered” memory consisting of memory cells with a separate read and write port, identical to the ones that are typically used in FIFO’s.

The question is now: for which kind of data flow exchanges and under which timing conditions can the memory size be reduced below \( 2N \), and how can we determine the size of the buffer? We start the discussion with an example. Consider the FIFO communication scheme in Fig. 7. The number of variables that is exchanged is \( 3 \). The difference between frames is made by quotes added to the variable name. We say that a variable is “bounded” if its value has been produced but not yet consumed. Obviously, the buffer size has to be equal to the maximum number of variables that may be bound at the same time. In the upper part of Fig. 7, where the system works according to Lemma 1, this number is 5. This is already less than 6, the theoretical upper bound. But now look at the lower part of Fig. 7. Here, the processors operation has been “skewed,” so that \( prc_C \) starts earlier with consuming the values that are being produced during frame \( f \) in \( prc_p \). The buffer size is now reduced to 2.

This example gives some hints for further investigations. Skewing is clearly a very interesting technique to reduce the memory size. It will be treated in a general way in the following section. Furthermore, the big gain in the example is obviously related to the fact that we were considering an FIFO data flow. However, in the sequel we will not restrict ourselves to this case, but we will consider all interprocessor communication memories that are fully pointer addressable.

B. The Optimal Skew Between Processors

The skew between two processors is defined as the number of cycles between the start of frame \( f \) in processor \( prc_p \) and the start of frame \( f \) in processor \( prc_C \). We assume that this value remains fixed during systems operation, i.e., that it is the same for each frame.

First, we give the following more formal definitions of the concepts “bounded” and “unbounded.”

A variable gets bound in frame \( f \) at the moment that its value is produced during frame \( f \) in \( prc_p \), i.e., at cycle \( P_r(f) \).

A variable gets unbound in frame \( f \) at the moment its value is consumed for the last time during frame \( f \) in \( prc_C \), i.e., at cycle \( C_{r,m}(f) \).

**Lemma 2—Optimal Skew Between Two Processors:** The optimal skew \( S_{opt} \) between two processors is given by

\[
S_{opt} = \max_{r,f} |P_r(f) - C_{r,m}(f)| + 1.
\]

**Proof:** The intention is to get the interprocessor memory as small as possible. The sooner a variable gets unbound after it has been bound, the better, because then the corresponding memory location gets free and may be used to store other values. Therefore, the smallest possible skew is also the optimal skew. The word “possible” refers to the obvious fact that a variable must have been bound before it may be consumed. Formula (1) gives the smallest number that strictly guarantees this condition for all variables and all frames. Therefore, it corresponds to the optimal skew.

Not surprisingly, reducing the size of the interprocessor buffer is the same as reducing the latency in the system. Note also that the skew may very well be negative.

C. Once In, Once Out Communication

In this subsection, we will introduce interprocessor buffers that are more general than FIFO’s but that can be built with almost identical hardware, the motivation being that these are largely encountered in practice. We will develop a general method to determine the minimum necessary size of that kind of buffer.

**Definition 3—Once In, Once Out Communication:** We call Once In, Once Out communication the kind of communication for which both production and consumption are deterministic and for which \( m_r = 1 \), for all \( v \).

Unfortunately, the name Once In, Once Out does not cover the condition of determinism, but is chosen in order to stress the similarity with FIFO’s. As in FIFO’s, both the production and the consumption sequence are pointer addressable, and every value is consumed only once, but the constraint that the two sequences have to be equal is removed. The ordering difference will be generated by a small routing network on top of the memory array.

If some more conditions are added, an interesting lemma on the reduction of the memory size can be proven.
Lemma 3: Consider invariant Once In, Once Out communication. The maximum in Formula (1), with which the optimal skew is calculated, is reached for a variable \( v_{m} \). (If more than one variable would give rise to the same maximum, one is free to choose the most favorable.) If now for all \( v \) for which \( C_{v} > C_{v_{m}} \), also holds \( P_{v} > P_{v_{m}} \), then, by optimally skewing, the buffer size can be made smaller than or equal to \( N \).

Proof: Apply the optimal skew between processors. This skew \( S \) may in this case be written as follows:

\[
S = \text{MAX} \left[ \alpha \epsilon \left( P_{v} - C_{v} \right) + 1 = P_{v_{m}} - C_{v_{m}} + 1 \right]. \tag{2}
\]

We will prove that the values of frame \( f \) are all consumed before the values of frame \( f + 1 \) are produced. That means that the same storage place may be used for those values, and as there are \( N \) variables, the number of necessary memory locations will at most be equal to \( N \). Arbitrarily, we let the production start at cycle 1, and the consumption will start \( S \) cycles later. Remember that \( F \) denotes the frame length, so for invariant communication, the variable's value of frame \( f + 1 \) will be produced exactly \( F \) cycles later than that of frame \( f \). So we have to prove that

\[ \forall v: C_{v} + S < P_{v} + F. \tag{3} \]

With Formula (2) we rewrite this inequality as

\[ \forall v: P_{v_{m}} - C_{v_{m}} < P_{v} - C_{v} + F - 1. \tag{4} \]

The way to prove this is to replace the term \( P_{v} - C_{v} \) by something that is certainly smaller or at most equal, and to verify that the resulting inequality always holds. The inequality will be verified in three steps: a) for the variables that are consumed before \( v_{m} \), b) for the variables that are consumed after \( v_{m} \), c) for the variable \( v_{m} \) itself.

Making use of the conditions of the lemma, we can state the following.

a) \( C_{v} < C_{v_{m}} \Rightarrow P_{v} - C_{v} \geq 1 - (C_{v_{m}} - 1) \geq 2 - C_{v_{m}}. \)

Substitution in Formula (4) yields

\[
P_{v_{m}} - C_{v_{m}} < 2 - C_{v_{m}} + F - 1 \]

\[ \Rightarrow \ P_{v_{m}} < F + 1. \quad \text{Q.E.D.} \]

b) \( C_{v} > C_{v_{m}} \Rightarrow P_{v} > P_{v_{m}} = P_{v} - C_{v} \geq P_{v_{m}} + 1 - F. \)

Substitution in Formula (4) yields

\[
P_{v_{m}} - C_{v_{m}} < P_{v_{m}} + 1 - F + F - 1 \]

\[ \Rightarrow \ C_{v_{m}} > 0. \quad \text{Q.E.D.} \]

c) \( v = v_{m} \Rightarrow F > 1. \)

This last result merely indicates that the lemma only holds for nontrivial frame lengths. The case for \( F = 1 \) would correspond to a communication system with one variable \( (N - 1) \) in which a value would be produced and consumed in every clock cycle. The system would therefore be word-oriented rather than block-oriented. This kind of communication indeed requires a common register with two memory locations (instead of one) so that the previous value is consumed while the new value is produced. (Note that a direct, memoryless connection of production and consumption bus would violate our initial assumption that a value may not be produced and consumed in the same cycle. Of course, whether this would be a possible solution or not merely depends on the timing characteristics of the system.)

Corollary: For the case of invariant FIFO communication, the buffer size can be made smaller than or equal to \( N \).

Proof: FIFO communication can be defined as follows. For all \( v_{1}, v_{2} \) for which \( C_{v_{1}} > C_{v_{2}}, P_{v_{1}} > P_{v_{2}} \) also holds. Therefore, the timing condition of the lemma is fulfilled, whatever \( v_{m} \) is.

What is interesting about this lemma is that it shows the importance of certain timing and ordering conditions for the reduction of the memory size. It also provides a formal support of our intuitive feeling that the gain is especially expected to be high in the FIFO case. However, it does not give an answer on what we really want to know, i.e., the minimum necessary value of the buffer size and this for all kinds of data flow exchanges.

Let us assume that we indeed use pointer addressing. What we in fact want to do is to let some of the \( 2N \) values of two consecutive frames share memory locations. But as the addressing sequence is fixed in the connections between the pointer cells, two values can only share the same memory location if the addressing pattern of the following variables, related with those values, is the same. To study this further, it is handy to define a concept \( R \), called repetitiveness in the ordering difference. We will first illustrate this with an example, and then give a small algorithm to find it.

Consider Fig. 8. With respect to a certain write (or production) sequence, the repetitiveness in the ordering difference is defined for three possible read (or consumption) sequences. In order to understand what \( R \) exactly means, first look at the second read sequence. By comparing to the write sequence, one sees that the same addressing pattern starts after 3 variables. In other words, the routing network that generates the correct read sequence for the first 3 variables will be identical with the one for the last 3 variables. Therefore, we say that \( R \) is 3 in this case. Similarly, we find that \( R = 1 \) for the FIFO case, while in the third case, where the read sequence is completely scrambled, \( R = 6 \) or in general \( R = N \).

The algorithm to find \( R \) is self-explanatory.

Algorithm 1—Repetitiveness in the Ordering Difference

1) Define the write and the read sequence in one frame.
2) Make a new sequence of integers, the integer on a certain position being found in the following way: define the variable on the corresponding position in the read sequence; define the position of that variable in the write sequence; subtract the two positions from each other.
3) \( R \) is the position in the new sequence on which it starts being repetitive. (The first position in a sequence is labeled 0.)

Lemma 4: The buffer size of a pointer addressed Once In, Once Out buffer is a multiple of \( R \).
Proof: The ordering difference between the write and the read sequence is generated by a routing network. After each R memory locations, an identical routing pattern will appear. This is, of course, only possible when the number of memory locations is a multiple of R.

Combining this Lemma with Lemma 1, we see that for the FIFO case, for which R = 1, the resulting size may be any number between 1 and 2N. While for the other extreme of R = N, there are only two possibilities: either memory reduction is possible and the buffer size is N, or not, and then the buffer size is 2N.

With all this in hand, we are ready to give the algorithm that defines the minimal buffer size for all pointer addressed Once In, Once Out Buffers.

Algorithm 2—The Minimum Necessary Buffer Size of Pointer Addressed Once In, Once Out Buffers:
1) Define the production and the consumption sequences, the sequences of minimal and maximal production times (taken over all frames), and the sequences of minimal and maximal consumption times.
2) Define the optimal skew S. (Maximal production times and minimal consumption times are needed in this step.)
3) Define R.
4) Consider now the sequences of minimal production times, and of maximal consumption times. Add the optimal skew S to each item of the consumption sequence. Divide both sequences in groups of R. Extend the production times sequence with the first group of R production times, all augmented with F, then with the second group of production times, all augmented with F, and so on, until all of the R added production times are larger than the consumption time of the “corresponding” values in the last group of consumption times. The “corresponding” value is the one that has the same relative position in the current group of consumption times as the relative position of the original value in the group in which it is consumed.
5) Define a variable #Reserved Locations and initialize it on 0. Consider now the first group of production times and the first group of consumption times and proceed as follows: if at least one of the production times in the group is smaller than the consumption time of the corresponding value, add R to #Reserved Locations and repeat for the next group of production times and the same group of consumption times. Else subtract R from #Reserved Locations and repeat for the same group of production times and the next group of consumption times. Stop when the consumption times sequence has been traversed completely.
6) The minimal buffer size is the maximum that #Reserved Locations has reached during the previous step.

Proof: The algorithm finds the maximal number of memory locations that have to be reserved under worst case conditions. Therefore, it has to consider minimal production times and maximal consumption times, taken over all frames. The consumption times have to be skewed. The values that are already produced in the following frame also have to be taken care of; therefore, the production times sequence is extended beyond the consumption times sequence. From Lemma 4, it follows that these times have to be taken in groups of R. The notion “corresponding” value is introduced for the following reason: when a group of R values would share memory locations with another group of R values, corresponding values will occupy the same location. Therefore, when comparing production and consumption times, corresponding values have to be considered.

Obviously, the buffer size must (at least) be equal to the maximum of the number of memory locations that have to be reserved at the same time. The body of the algorithm calculates that changing number throughout the system’s operation, using the variable #Reserved Locations. When R production times are compared to the R consumption times, two situations are possible: either at least one of the production times is smaller than or equal to the corresponding consumption time, or not. In the first case, R additional storage places have to be reserved, and the following step is to compare the next R production times to the current R consumption times. In the other case, R storage places may be released and the following step is to compare the next R consumption times to the current production times. The algorithm is initialized by considering the first R consumption and production times, and because of the skew, this step will always result in a reservation of R places. This concludes the explanation of the initialization and the body of the algorithm.

It is somewhat more difficult to prove that the algorithm stops at a valid place. Indeed, #Reserved Locations is initialized on 0, even though during real system’s operation it might well be that there are still values stored in the buffer at the beginning of a new production frame. We have to make sure that the algorithm is executed “long enough,” until the effect of this “incorrect” initialization has disappeared. To see that this is so, note that we did not only neglect the previous production frame, but also the previous consumption frame: in the real system, all values that were still in the buffer at the beginning of the production frame will certainly be consumed before the next consumption frame starts. Therefore, during the execution of the algorithm, #Reserved Locations will get the same, correct value at the begin of the consumption frame as at the end. Beyond this point, the input to the algorithm would be the same time sequences, shifted over one frame, and consequently the sequence of values for #Reserved Locations would start being repetitive. Therefore, the algorithm can be stopped after one consumption frame.

Obviously, the buffer size has to allow for the maximum number of reserved locations, and this explains the last step.
To illustrate the algorithm, consider the example in Fig. 9. In this case, \( N = 8 \) and \( R = 2 \). According to earlier results, we already know that the buffer size will be smaller than or equal to \( 16(2N) \) and a multiple of \( 2(R) \). The skew was calculated with maximal production times and minimal consumption times, taken over all frames. The figure shows minimal production times and the skewed sequence of maximal consumption times.

The variable `Reserved_Locations` has been initialized on 0 at the start of the production frame. The algorithm begins by comparing the production times for the first group of 2 values \( \{a \ b\} \) to the consumption times for the first group of 2 values \( \{b \ a\} \). The production times are of course smaller than the consumption times, thanks to the skew, and therefore `Reserved_Locations` is incremented with \( R = 2 \), and becomes 2. The following step is to compare the production group \( \{c \ d\} \) to the consumption group \( \{b \ a\} \). Here `c` "corresponds" to `a` and `d` to `b` because the ordering differences imply that `c` could only share a memory location with `a`, and `d` with `b`. As `c` is produced earlier than `a` is consumed, 2 more memory locations have to be reserved, and therefore `Reserved_Locations` goes to 4. Now the production group \( \{e \ f\} \) is compared to the consumption group \( \{b \ a\} \). This case, all consumption times are smaller, and thus `Reserved_Locations` is decremented with 2 and becomes 2. Next, the production group \( \{e \ f\} \) is compared to the consumption group \( \{d \ c\} \), and so on.

In this way, the algorithm continues and a sequence of values for `Reserved_Locations` is generated. Note that the new values `a'` and `b'` of the following production frame are taken into account. Note also that `Reserved_Locations` reaches the same value (2) at the start of the consumption frame as at the end, as was explained in the proof of the algorithm. Therefore, the algorithm can be stopped after one consumption frame. The final buffer size is the maximum of the sequence of values for `Reserved_Locations`, which is in the case equal to 4. Thus, the concept of skewing allows us to save a factor 4 in memory size, in comparison to a straightforward implementation without skewing, which would use a buffer of size 16.

### D. Extension Toward All Fully Pointer Addressable Buffers

There is a class of very practical interprocessor buffers that we did not consider until now, but that is easy to include in the previous theory. These are the buffers that are fully pointer addressable but that are not of the Once In, Once Out type. Looking at the condition for pointer addressing and at the definition of Once In, Once Out communication, we see that for these buffers \( m_c = M \neq 1 \). In effect, the same consumption sequence, that contains each variable exactly once, will be read \( M \) times. This implies that all the \( N \) values belonging to one frame will have to be stored and that the read pointer has to be fed back after \( N \) positions. Here are the extensions to Algorithm 2, needed to solve this problem.

**Algorithm 2—Extensions—Extensions to Find the Minimal Buffer Size of All Fully Pointer Addressable Buffers:**

1. Define also \( M \).
2. Identical.
3. If \( M \neq 1 \), set \( R \) formally equal to \( N \).
4. Identical.
5. Identical.
6. If the maximal value that `Reserved_Locations` reached equals \( N \), use a single-buffered memory of \( N \) places, else the maximum will be equal to \( 2N \) and a double-buffered memory of \( N \) cells columns (and \( 2N \) storage places) has to be used.

We have now a method to find the minimal memory size for all buffers that are fully pointer addressable. Further research could extend the algorithms toward even more general types of communication, where an ACU, an address bus, and a decoder are necessary to generate the addressing sequence. However, all practical examples that we have been studying until now may be solved with pointer addressing.

### V. Automatic Synthesis of the Interprocessor Communication System

The goal of a real Silicon Compiler, like Cathedral II, is to go from a behavioral algorithmic description to layout in an, as much as possible, automated way. A typical feature of Cathedral II is that it tries to exploit parallelism at the processor level with a multiprocessor architecture. In order to cope with the complexity of this synthesis problem, the algorithm is first divided into a number of independent subprocesses of individual processors. The synthesis of the interprocessor communication system is thus `a posteriori` at the moment that all processors are, or may be, synthesized completely.

In the Cathedral II environment, a program has been developed that automates the interprocessor communication system synthesis. Being a prototype program, it has been implemented in LISP. The program operates in two different modes. In the first mode, information is extracted that may influence the processor synthesis. For example, when pointer addressing is used, the ACU does not have to generate addresses for the interprocessor buffer. However, the processor synthesis tools will in a first phase always use the most general solution with the ACU. Therefore, the interprocessor synthesis tool will investigate if pointer addressing is possible and, if it is, it will propose this option under the form of a so-called "pragma." The pragma concept is a unique feature of Cathedral II, being the way in which the designer may give structural hints to the compiler. It should be stressed...
that the designer is not obliged to use the interprocessor (or any other) pragmas; for example, if he feels that the ACU/address bus/decoder option would yield a better result, he is free to leave out the corresponding pointer addressing pragma.

In the second mode, the processor synthesis is considered to be completed with all pragmas taken into account (whether they have been provided manually by the designer, or automatically by the program in the first mode), and the program proceeds by optimization steps that have no influence on the processor synthesis. These consist of architectural decisions and the minimization of the buffer sizes. The program uses the algorithms that have been presented in this paper. After defining the skew between communicating processors, the buffer sizes are calculated and the corresponding module generator calls are defined. Furthermore, the supervising controller is synthesized. This controller provides the start signals to the processors, the reset signals to pointer shift registers, and the transfer signals to double-buffered memories.

The interprocessor communication synthesis tool has been verified on several examples. Three of them are presented in the next section.

VI. EXAMPLES

A. A Pitch Extractor for Speech

As a first example, we will consider a pitch extractor system for speech [8]. This was also the initial test vehicle for Cathedral II.

The data flow between the four processors of the system is depicted in Fig. 10. In this system, with pipelined processors, there is interprocessor communication at three places. The data blocks consist mainly of arrays of variables, which are read in the same order as they are produced. However, none of the three cases corresponds completely to simple FIFO communication. With respect to the minimization of the buffer sizes, it is important to know that the third processor is the "bottleneck" with respect to the number of cycles; the cycle counts of the programs in the other processors are 2-10 times smaller.

In the first processor, an array of amplitudes is computed, but so also is the maximum of the array, which is scaled and used as a threshold in the next processor. As could be expected, the second processor first needs the threshold and then the array. The communication being the Once In, Once Out type, pointer addressing may be used for both the read and the write sequence. The data "scrambling" may be accomplished by a correct initialization of the read pointer and by one feedback track. The situation looks quite common: an array of similar variables is computed, together with some global information about the array, which necessarily comes last. That global information is needed in the next processor before further computations on the array can start. In this case, \( R = N = 65 \), and that is also the final buffer size.

The second processor computes an array of 8 frequency values. The third processor needs this array in the same order, but reads it over and over again. Every time, the array is compared to a new predefined pattern in order to find the best matching pattern. The only global variable that is produced is the fundamental frequency of the optimal pattern. Something similar happens in a DFT processor where, for the calculation of each single component of the spectrum, all time samples have to be read. This type of communication allows the use of pointer addressing but it is not of the Once In, Once Out type. As shown in Subsection IV-D, the buffer will be either an 8-word double-buffered or an 8-word single-buffered memory. Whether the latter (and smaller) solution may be used depends on the timing information. It turns out that in this example a single-buffered memory is sufficient.

The third processor computes global information about the frequency array. The fourth processor first reads the optimal fundamental frequency, and then the array, in order to compute the final pitch. The situation is therefore similar as between the first and the second processor except that a separate processor computes the global information about the frequency array. Note that the third processor also has to pass the frequency values. Hence, \( R = N = 9 \), and a 9-word single-buffered memory is needed for the data buffering.

Although none of the data flow was really of the FIFO type, it has been shown that the interprocessor communication in this system may be solved completely by devices with the same conceptual simplicity as FIFO's. For the processors, or from the floor-planning point of view, there is not a single difference.

B. The Pipelined FFT

As a second example, we will consider one of the most important DSP algorithms, i.e., the Fast Fourier Transform, and more precisely the pipelined implementation of it.

A schematic illustration of the FFT algorithm is depicted in Fig. 11. It consists of a number of so-called "butterfly" stages, equal to the logarithm (with base 2) of the number of input samples. The vertices in the graph correspond to the partially transformed data, the edges illustrate the data dependencies during the computations. The data are complex values, although we will assume that there is one complex memory location per value, as normally is done in explanations about the FFT algorithm, two memory locations per complex value would be necessary in a real implementation.

The idea of the pipelined FFT algorithm is to assign each butterfly stage to one processor. In that way, we end up with a linear array of processors, with the partially transformed data flowing from each processor to the next
one. This explains why this implementation is said to be "pipelined." The algorithm is ideally suited for this kind of parallelism because the processors are equally loaded. Indeed, they all perform a similar butterfly operation, the difference being only the multiplier coefficients and the "span" of the butterflies.

As the vertices in the graph correspond to partially transformed data, they also correspond to values that will have to be stored for some time. Note the "repetitiveness" in the data dependencies of the computation of one butterfly stage. After a certain number of inputs have been treated, a similar dependency graph appears. Therefore, we can hope that, by optimally skewing, memory locations corresponding to "equivalent" data values may be shared. Note further that the position where the repetitiveness starts doubles after each butterfly stage.

The example has further been worked out for the case of 16 input samples, but the conclusions are general. There are four processors in the pipeline, called BFL1 to BFL4. In Fig. 12, all the buffers that perform the communication between processors (including I/O) are shown. It is easy to verify that all communication is of the Once In, Once Out type. It is assumed that this observation is exploited by using pointer addressing.

The first processor has to access the input samples in a typical FFT manner, called bit-reversed addressing. This name comes from the way in which the consecutive addresses are generated: a sequence of incremental addresses is transformed into the "bit-reversed" sequence by reading the binary representation of each original address from right to left. This is a hard sequence: not only is \( R = N \), but all of the \( 2N \) memory locations, corresponding to the theoretical upper bound, are necessary for the storage of the input samples. For the implementation, we chose a double-buffered memory containing a separate read and write location per cell; a single-buffered solution is also possible, and the difference between the two is not essential at this point. Schematic ally, the memory has been split into two planes and a routing network in between the planes generates the bit-reversed sequence. The connections between pointer cells are depicted by straight lines. An identical buffer at the output generates the correct output sequence.

After this, things get better. By optimally skewing, the interprocessor buffer size can be reduced to the lowest possible value, i.e., the corresponding \( R \). This \( R \) is closely related to the repetitiveness observed in the data dependencies at a certain butterfly stage in Fig. 11 and it doubles after each stage (or processor). The largest buffer, the one between BFL3 and BFL4, has also been split into two parts: besides a better shape, this has the advantage of a very small routing network. At the bottom of the figure, there is a schematic representation of the skew between processors: note that the skew also doubles after each stage. The total number of memory locations in the system is \( 6N = 4 \), instead of \( 2N(1 + \log_2 N) \) in a system without memory reduction. No less than \( 4N \) is due to the bit-reversing I/O operations.

It should be noted that the results on skewing and memory reduction in the pipelined FFT are not new: similar results can, e.g., be found in [9]. What is new is the general (and automizable!) method with which we have found these results: the same method may be used to optimize all kinds of other systems. But maybe the most important novelty is the implementation of the communication buffers: this is oriented toward single chip VLSI solutions and uses the freedom that VLSI is offering. Typically, systems like the pipelined FFT are implemented on board level where each processor and each data buffer is a separate chip or several chips. The data buffers may be constructed out of shift registers, multiplexers, and a non-trivial control structure to get the right value out at the right moment [10]. In the implementation that we propose, that problem is solved in a routing network, added at the module generation level, and therefore from a system's point of view that difference with simple FIFO's disappears. For an FFT implementation, in which the addressing part is typically the main problem, this is a most interesting result.

C. Control System of an Optical Pickup Element

The following example is taken from a system that controls the position of an optical pickup element in a commercial application. The actual position of the optical pickup is sensed by four diode signals. These signals are
digitized by sigma delta modulators and are the input to a
first processor, which performs a second-order decimat-
ing filtering operation. In each frame, this processor cal-
culates two times four filtered diode signals. These are
passed to a second signal processor, which performs fur-
ther calculations on the four pairs of corresponding diode
signals. Obviously, the communication is not of the FIFO
type, but it is Once In, Once Out. Actually, the appro-
priate buffer is identical to the 8-word buffer in the pre-
vious example. A module-generated layout of this buffer
is shown in Fig. 13. Note the routing network on top of
the read pointer shift registers performing the reordering
of the sequence.

VII. CONCLUSIONS

In this paper, architectures and algorithms have been
presented that may help in finding an optimal interpro-
cessor communication solution in synchronous multipro-
cessor digital signal processing chips. In practice, a lot of
systems are easily partitioned into a linear array of con-
secutive processors. For these systems, the whole prob-
lem may be reduced to unidirectional communication be-
tween two processors. A general model for this case has
been introduced. This model allows us to define important
properties of DSP data flow in a mathematical way, to set
up formal rules for the possibility to use certain architec-
tures, and to study the effect of skewing the operation of
the processors on the buffer sizes. A number of architec-
tural possibilities have been discussed. In this context,
double-buffered memory and an extended method of
pointer addressing, leading to the concept of Once In,
Once Out communication, have been introduced. Algo-
rithms that determine the minimal buffer size for all fully
pointer addressed buffers have been presented. A synthesis
tool for interprocessor communication, which is part of
the Cathedral II silicon compiler, has been described.
The approach has been illustrated with a number of ex-
amples.

REFERENCES

[1] H. DeMan, J. Rabaeu, P. Six, and L. Claesen, “Cathedral II: A sil-
icon compiler for digital signal processing multiprocessor VLSI sys-
module generation system for multiprocessor systems on a chip,” in
Proc. NATO-ASI on Logic Synthesis and Silicon Compilation for VLSI
“Cathedral II: Computer aided synthesis of digital signal processing
cient microcode compiler for custom DSP—Processors,” in Proc.
multiprocessor digital signal processing chips,” in Proc. VLSI’87,
H. De Man, and J. Vandewalle, “General datapath, controller and
inter-communication architectures for creation of a dedicated mul-
tiprocessor environment,” in Proc. IEEE ICAS’86, San Jose, May
pitch extraction from speech and a hardware model applicable to
implementation,” in Proc. IEEE Int. Solid-State Circuits Conf., San

Jan Decaluwe was born in Deinze, Belgium, on
July 22, 1962. He received the electrical engi-
neering degree from the Katholieke Universiteit
Leuven (KUL), Belgium, in 1985. From
October 1985 until February 1988 he was
with the VLSI Systems Design Methodologies
Group at the Inter-University Microelectronics
Center (IMEC), Heverlee, Belgium, where he
worked on circuit design techniques, VLSI archi-
tectures, and architectural synthesis. Since March
1988 he has been a Visiting Researcher at the La-
boratório de Sistemas Integrados at the University of São Paulo, Brazil,
where he is working on hardware description languages, logic simulation,
and architectural synthesis.

Jan M. Rabaeu (S’80–M’83) received the E.E.
and Ph.D. degrees in applied sciences from the
Katholieke Universiteit Leuven, Belgium, in 1979
and 1983, respectively, on the topic of computer
aided analysis of switched capacitor circuits.
From 1983 till 1985 he was connected to the
University of California, Berkeley, as a Visiting
Research Engineer, where he developed an auto-
mated synthesis system for multiprocessor DSP
architectures. From 1985 till 1987 he was heading the Architectural and Algorithmic Strategies
Group in the VSDM (VLSI System Design Methodologies) section of the IMEC Laboratory, Leuven, Belgium. In 1987 he joined the Faculty of the University of California, Berkeley, where he is currently an Assistant Professor. His main interests are in the study of signal processing architectures and in the computer aided analysis, synthesis, and design of signal processing algorithms. He has authored or co-authored more than 40 publications.

Dr. Rabaei received the IEEE Journal on Computer Aided Design best paper award in 1986.

Jef L. Van Meerbergen (M'87) was born in Aarschot, Belgium, in 1951. He received the engineering degree from the University of Leuven in 1975 and the Ph.D. degree from the same university in 1980 in bandgap narrowing in silicon solar cells.

In 1979 he joined the Philips Research Laboratories, Eindhoven, The Netherlands. He was involved in the design of MOS digital circuits, general-purpose signal processors, and architectures for DSP. He is presently engaged in the design of silicon compilers for DSP applications in general, and architectural level synthesis in particular.

Hugo J. De Man (M'81-SM'81-F'88) was born in Boom, Belgium, on September 19, 1940. He received the electrical engineering degree and the Ph.D. degree in applied sciences from 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, 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. His actual field of research is the design of integrated circuits and computer-aided design. Since 1984 he has been Vice-President of the VLSI Systems Design Group of IMEC, Leuven, Belgium.

Dr. De Man 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 Circuits and Devices from 1982 to 1985. He 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.