

# Architectural Strategies for an Application-Specific Synchronous Multiprocessor Environment

FRANCKY CATHOOR, MEMBER, IEEE, JAN RABAEY, MEMBER, IEEE, GERT GOOSSENS,  
JEF L. VAN MEERBERGEN, RAJEEV JAIN, MEMBER, IEEE, HUGO J. DE MAN, FELLOW, IEEE,  
AND JOOS VANDEWALLE, SENIOR MEMBER, IEEE

**Abstract**—In this paper, a multiprocessor architecture is presented which is suited for the customized and automated VLSI realization of complex low- to medium-speed DSP applications. The proposed architecture is constructed from a set of flexible and parameterizable data paths, a selection of powerful control units (for decision-making tasks), and a number of protocols for fast interprocessor communication. The flexible nature of this system allows for an efficient hardware realization, exploiting the inherent parallelism of a particular application. The effectiveness of the approach is substantiated with the synthesis of several test vehicles, such as a pitch-extraction algorithm for speech, in terms of the defined architecture.

## I. INTRODUCTION AND GOAL OF THE MULTIPROCESSOR SYSTEM

**N**OWADAYS, an increasing number of DSP applications are emerging, with low to medium throughput requirements, where the data rates are situated between 1 kHz and 1 MHz. Hereby, a rapidly increasing arithmetic complexity has to be combined with a need for flexible and powerful decision-making operations. These applications belong to the class of the so-called “second and third generation” signal processing algorithms, which are not only addressing word-oriented applications but also employ advanced vector and matrix operations [1].

Commodity (general-purpose) DSP processors are usually not powerful enough to handle these complex algorithms. Moreover, they cannot be parameterized according to the designer's needs such as memory and word-length requirements. In contrast, the necessary computation rate and customization can be achieved by semicustom gate-array or standard-cell approaches, but these require too much chip area. So, for both of these alternatives, too many chips would have to be provided leading to an unacceptable system cost.

On the other hand, the speed achievable with a full-

custom hard-wired bit-parallel bit-slice approach (see, e.g., [4]) cannot be fully exploited, as the area-time tradeoff becomes only favorable for the type of architectures in the range of high-speed applications (around 10 MHz). Moreover, for the applications envisioned, the involved cost overhead due to the excessive design time of a full-custom design is seldom justified.

Hence, it seems appropriate to look into a *more customized type of (multi)processor architecture* [3], [4], where *both speed and area efficiency are improved over the conventional programmable DSP processors*. This can be achieved by exploiting the specific properties of the implemented algorithms, such as, for instance, the inherent parallelism, and by matching the hardware to the algorithmic complexity. The new architecture methodology which will be proposed also features the additional flexibility to largely avoid the traditional bottlenecks of the general-purpose signal processors, especially with respect to the available data transfers and controller constructs. In this tradeoff, some additional hardware will have to be sacrificed, but this overhead can be reduced by selecting the best-suited realization alternatives from the ones provided in this paper.

As the architectures can become fairly complex, the process of mapping a particular DSP algorithm onto them has to be supported by a set of synthesis, optimization, and verification CAD tools, collected in the CATHEDRAL-II environment, which are currently being developed at our laboratory [8], [9], [16], [32]. So far, a large number of other “silicon compilation” environments employing (multi)processors to implement application-specific IC's tuned for digital signal processing have been published. Examples are found in SILC [12], Apollon [14], ALGIC [15], MIMOLA [21], S(P)LICER [25], HAL [27], CMAD [28], LAGER [30], [36], MacPitts [40], and CMU-DA [41]. But the authors believe that the efficiency in terms of density and achievable throughput can be substantially elevated by means of the concepts applied in their “CATHEDRAL-II” CAD-system [9].

The ultimate goal is to provide an *automated strategy* whereby the system engineer has to specify a *high-level behavioral (applicative) description* of the algorithm, which is then gradually refined and transformed *into a low-level functional description* in terms of a number of

Manuscript received June 17, 1986; revised June 15, 1987. This work was supported by the joint ESPRIT 97 project of the European Communities, in cooperation with several industrial and university partners.

F. Cathoor, G. Goossens, and H. J. De Man are with the Interuniversity Micro-Electronics Centre (IMEC), Kapeldreef 75, B-3030 Heverlee, Belgium.

J. Rabaey and R. Jain are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

J. Van Meerbergen is with Philips National Laboratories, Eindhoven, The Netherlands.

J. Vandewalle is with the ESAT Laboratory, Katholieke University Leuven, Kard. Mercierlaan 94, B-3030, Heverlee, Belgium.

IEEE Log Number 8718218.

prespecified primitive building blocks. The latter are then composed out of the parameterizable basic cells of a library (meet-in-the-middle strategy [7]). Finally, the floor plan of the entire chip is assembled and a last verification step (on the logical and the timing level) ensures that all connections operate correctly. The masks can then be fabricated, sent to the silicon foundry, and tested.

It should be noted that if desired, the designer will also be able to influence the final results in a positive way by feeding his/her experience into the "open" system at some specific entry points [16], [32] in order to arrive at a more optimal implementation.

In order to set up an automated CAD system which supports such a methodology, it is necessary to fully define the architectural strategy first. Otherwise, when the system approach is too general, the tools in the *silicon compilation environment* cannot handle unlimited flexibility in an efficient way. Therefore, a set of "constraints" has been incorporated, such as the fact that all modules have to operate in a synchronous way. Also, the application of the dedicated processor concept imposes constraints. However, these limitations will not significantly reduce the intended application domain in the DSP area; they will only influence the achievable throughput and the area efficiency, and the latter are still far better than the general-purpose approach.

The architecture which will be presented in this paper forms the core around which the high-level synthesis tools [17], [32] have been developed. Obviously, its definition will have to be based on manual design experience. For our purposes, a monolithic pitch-extraction system for speech coding [39] has served as a primary test vehicle [8], [16] during the definition phase (see also Section V). The set of typical operations which have to be performed in this algorithm indicates the *need for a limited number of "execution-units"* (EXU's) such as some powerful arithmetic units with a quite general behavior, as discussed in Section III. Experiments [4] have shown that these EXU's can be used in a straightforward way to realize other speech- and audio-domain systems as well. Moreover, due to the inherent speed advantage of our customized approach, the operation rates which can be maintained should be high enough to tackle automation, telecommunication, and back-end (100 kHz–1 MHz) image-processing applications too, such as, for instance, intelligent controllers for robots, high-speed modems, and pattern recognition or classification algorithms.

In order to minimize the area cost during the actual implementation phase, a semicustom silicon integration will be envisioned, making use of a set of fully characterized, predefined primitive cells, stored in a technology independent way in a library. These cells can then be assembled into a limited set of parameterizable "functional building blocks." This approach fits into the "third generation custom-design methodology" as advocated in [7]. The selection of this set of essential FBB's, such as an adder/subtractor, a register-file, and so on, in the EXU's has to be carefully investigated, with the common prop-

erties of the target applications in mind. Indeed, for the realizability in a VLSI design process, it is necessary to increase the modularity and hence to keep the number of primitive cells as low as possible.

In this paper, only the architectural methodologies for the DSP synthesis environment will be covered, and not the CAD aspects which constitute the topic of other publications [9], [16], [32]. The paper is organized as follows. In Section II, a general view and the hierarchy of the strategy will be presented. The main modules, namely, the processors, will be decomposed into parameterizable "execution unit"-data paths (Section III) and controllers (Section IV). For both of them, several alternatives will be compared from which a few prototypes will be withheld in the first version of the silicon compiler. Also, the necessary communication between the building blocks will be handled (topic of Subsections II-C and III-E). Finally, in Section V, some selected test vehicles will demonstrate the power of the outlined strategy.

## II. SURVEY OF THE ARCHITECTURAL STRATEGY

In this section, the general concepts of our architectural approach will be introduced. The class of DSP algorithms which is addressed (see Section I above) is typically constructed from a combined set of complex arithmetic and decision-making subtasks. In these target applications, the considered sample rates also remain at least an order of magnitude below the achievable clock frequencies (of about 10 MHz in a 3  $\mu\text{m}$  CMOS technology). For these two reasons, a flexible data path in combination with a controller is to be preferred over a presumably underemployed fully hard-wired design. Such a "processor" approach is particularly suited for the conditional type of operations as required in the ubiquitous decision making which is part of most complete DSP systems.

### A. System Hierarchy

In order to cope with the complexity of present-day VLSI realizations, it is essential to construct the architecture following a modular, hierarchical approach. Such a partitioning will not only reduce the implementation time in a substantial way, but it will also simplify the task of the synthesis tools which have to translate the algorithmic description into a suited architecture.

A decomposition of the system also requires the definition of a variety of adequate communication and interfacing protocols between the different modules at each hierarchical level. The appropriate choice will then depend on the amount and the type of the data to be transferred.

It has to be stressed too that it is vital to make the correct choice for the boundaries between the levels. Otherwise, an inefficient and inflexible methodology will result. A wrong decision would also have a devastating effect on the load balancing between the respective operational modules: some of them will become occupied only very infrequently because of mismatches in the achievable data throughput which does not meet the require-

ments, and this cannot be tolerated due to the waste of valuable chip area.

For these reasons, the following hierarchy is adopted in our system (Fig. 1): the chip itself is considered at the root, containing a number of *processor modules* which are operating relatively independently of each other. In fact, only global variables are interchanged by means of buffer units containing enough memory to handle the data exchanges (Subsection II-C). This will allow us to decouple the design task so that, in principle, only the programs on the individual processors have to be tackled. Afterwards, a separate tool will then take care of the required data flow between the processors. On the next lower level, each processor contains a controller (Section IV) which steers a cluster of “*strongly connected execution units*” (Section III). The latter are communicating with each other over a restricted number of dedicated busses. In this way, contention as occurring in the case of a dual-bus architecture is largely avoided. Each EXU is assembled out of a data path and possibly a local controller. These powerful modules will handle the arithmetic complexity and the decision making required by the algorithm.

Each EXU data path consists of a set of predominantly abutting “*functional building blocks*” (FBB’s), with customized connections. The FBB’s are groups of primitive cells performing an elementary arithmetic or logical operation. Examples are an adder/subtractor, a multiplier, a comparator, and so on. These parameterizable building blocks have to be provided by a set of module-generator programs, using the predefined (bit-level) cells stored in a library [9]. The compacted stick diagrams employed in this symbolic layout approach will allow a smooth update when the technology has to be scaled. Afterwards, the module generators for the complete EXU’s are constructed from the appropriate FBB descriptions. A similar assembly strategy can be employed for some of the control structures which are investigated, and which are modular enough to be constructed from more basic primitives, such as, for example, in a PLA which is generated from OR, AND, load, buffer, and glue cells [2].

In order to communicate with systems on other chips, *external I/O devices* have to be provided in the periphery. For this purpose, the following strategy will be applied: data are exchanged at the external interface by means of an asynchronous protocol which employs FIFO’s to buffer the information. Because of this buffering, it is possible to allow a small skew in the time references between the on-chip and off-chip circuitry without causing conflicts.

The control of this entire I/O operation is distributed over 2 levels. A fully synchronous protocol with “ready” and “acknowledge” signals is used for steering the communication between the FIFO’s and the external devices. Also, “empty” and “full” flags have to be provided in the buffers in order to exchange the status. On chip, data can be transferred from the FIFO’s to the processors and vice versa in a synchronous way with read- or write-enable signals, similar to the interprocessor communication described in Subsection II-C.

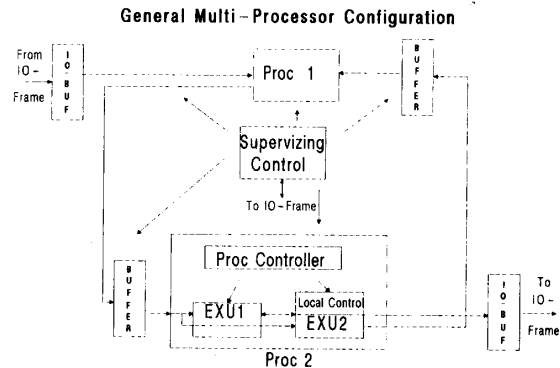


Fig. 1. Definition of the system hierarchy. Illustration of the levels which have been identified in the architecture definition.

### B. Hierarchical Processor Control

As already mentioned in Subsection II-A above, in order to decrease the complexity of the overall design, a modular approach has to be adopted, which allows us to identify several levels in the system hierarchy (Fig. 1). The same partitioning will now be applied for the design of the controllers.

First of all, some of the FBB’s themselves require a reasonably large set of control signals which can be grouped and coded with a reduced amount of instruction bits. This approach requires a *local decoder* to be stacked on top of the bit slices in the data path. This hardware tradeoff will usually be favorable due to the decrease in microinstruction word length, and more importantly even, the reduced number of wires carrying the control signals to the data path.

Internally, the processors contain a set of EXU’s and a single controller. The latter restriction has been made to simplify the synthesis task. At this level, a *microcode-ROM*, as the one of Subsection IV-A, could turn out to be the most attractive option. Sometimes a particular EXU can contain a local controller which acts as a slave to the processor controller. The response of such a local sequencer is faster than for the pipelined microcode controller. This will allow us to feed the decisions immediately back to the EXU without losing cycles due to pipeline delays. An example can be found in the divider of Subsection III-C-3).

Finally, entire processors which are performing related tasks can be located at the same level in the hierarchy. The intercommunication then has to be synchronized by a *common supervising control unit*. Since the number of processors is limited due to chip-area considerations, usually a single top level suffices. Essentially, from this point of view, the processors can be regarded as black boxes. After transmission of the START pulses, only the specification for the relative timing of the exchanged input and output signal(s) has to be taken into account.

### C. Interprocessor Communication

In this subsection, a number of efficient interprocessor communication strategies will be discussed [4]. The

choice between the options for a particular case will be determined by the properties of the required transfers: the number of sources and destinations, the required transfer capacity, synchronism, matching of data rates, the contention characteristics in the communication network, and whether or not to include handshaking.

The routing and placement task for the processor modules has to be performed by a floor-planner. The connections will be routed in channels in between the function blocks. Due to the usually very long communication tracks between the processors, the capacitive load of the buffers can become large (several pF). Therefore, these *transfers are latched* (with a full register) and consume at least one clock cycle.

Throughout this subsection, contention of the communication networks is, in general, considered as unacceptable. Indeed, in that case, part of the synchronism is lost and the concepts of self-timed and asynchronous systems (handshaking protocols) would have to be introduced (which will be avoided for the time being). In most cases, it appears to be more efficient to avoid the contention by extending the number of independent communication paths up to the number which is required in the algorithm to be mapped.

A restriction also applies for the scheduling of conditional jumps. If the actual times for the data transfers between the communicating processors are already known at compile time [17], [32], they are available to compute the skewing of the time reference for the processors in advance. However, if data-dependent transfers between the processors would be allowed too, the interprocessor communication would become much more complicated as, in that case, a full 2-way handshake protocol cannot be avoided. Therefore, the latter option is not considered for the time being.

However, the extension for allowing WHILE constructs or FOR loops with variable indexes is not too complex and is supported.

In the subsequent Subsections II-C-1)-3), several types of intercommunication networks are discussed for multiprocessor architectures. They are classified based on the rates of the two processors which have to be matched and on the compatibility of the order of the data transfer.

In Subsection II-C-4), the hardware involved for the components of the communication system (link, switches, storage) is discussed more elaborately.

1) *Processors with Matched Rates:* In this simple case, both processors can be directly connected to each other without spending additional hardware, but in view of the buffering latch, a separate clock cycle is necessary for the transfer. In the most general case, multiple sources and/or destinations have to be considered. So for establishing a link, the introduction of a multiplexer is required to select one of the sources and a demultiplexer to distribute the data to the correct destination(s) (Fig. 2). Both switches can be directly driven with signals generated in the master controller at the processor level.

In order to avoid the loss of information, an appropriate

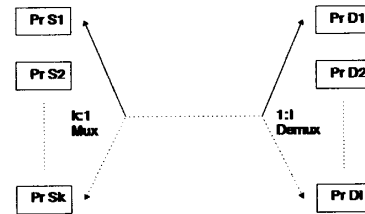


Fig. 2. Most simple interprocessor communication. Hardware for matched rates.

skewing of the data generation in the sources is necessary [17]. Unfortunately, this approach usually implies a correction between the instruction sequences of the respective processors. Independent design of the processors then becomes impossible. Hence, this option should be avoided, unless it is verified that no collisions are introduced. This could, for instance, occur if the global communication happens very infrequently. In that case, the tolerance on the absolute cycle time of the transfer will be relatively large so that it can be matched between the processors.

2) *Processor Rates not Matched, But in Sequence:* If the source and the destination processor(s) are communicating the data in the same order, the only addition compared to the previous subsection consists of a *dual-port FIFO-buffer* which has to be introduced in between the processors (Fig. 3). The number of registers has to exceed the maximal skew (in terms of the number of interleaving data) which is expected between reading and writing a specific value. In principle, the destination processor can read the data only once but, if necessary, this restriction can be relaxed if the same sequence has to be repeated a number of times. For this purpose, a cyclic FIFO should be provided which keeps on generating the entire block of stored data until its content is overwritten with new data from the source. This action is obtained by means of a two-way switch (see, e.g., [39]).

An easy way to solve the control requires only two modulo counters, with one of them always indicating the last item which has been entered, and the other pointing at the oldest value stored in the buffer. So no complicated address calculation unit (ACU) is necessary. The implementation as a feedback shift register with a unique "traveling 1" is even more efficient when special hardware can be employed.

3) *Processors with an Incompatible Sequence of Data Transfers:* In order to avoid access conflicts in this most general case, a double buffering of the source and destination data is necessary. This can be achieved by providing two separate memory sections which are alternately addressed during two successive sample or frame periods. One of the sections has to be assigned for the write operation from the source processor. Simultaneously, the destination processor can read the data from the other section, updated there during the previous period. For this purpose, a dual-port RAM is more generally suited than a single port.

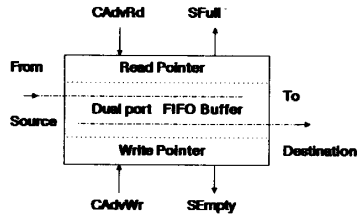


Fig. 3. FIFO-based interprocessor communication. A dual-port first-in first-out buffer is most appropriate if the data sent by the source are read into the destination processor(s) in the same order. The control signals CAdvWr and CAdvRd are used to advance, respectively, the read and write pointers. The status signals SEmpty and SFull notify whether the buffer is empty or full.

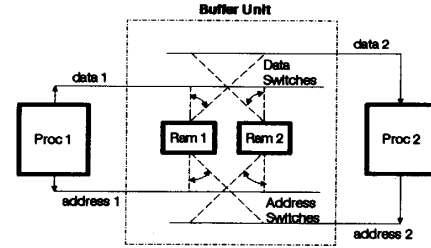


Fig. 4. Most general synchronous interprocessor communication. Two switched single-port RAM's with separate data and address busses are alternately read and written over successive sample periods.

Such an organization, with two data ports, is required to ensure that bus-collisions are avoided under all circumstances. However, as each of the variables which have to be exchanged has to be stored twice anyway, the dual-port memory can be more efficiently “modeled” by means of *two switched single-port memories* (Fig. 4), occupying a smaller area per cell. The switching of the data and the addresses happens by means of a two-way multiplexer and a demultiplexer controlling the link between the two communicating processors and the I/O and address ports of both RAM's.

In general, two separate address calculation units (ACU's) have to be provided. However, as the data are not overwritten during one “block period,” write addresses can be obtained with a modulo counter (i.e., in a linear order). Hence, only the read addresses have to be generated with a full ACU. Alternatively, if the size of the RAM is not very large, the required address bits can be coded directly into the ROM of the supervising controller. In cases with lower data-transfer rates, it is also possible that a single memory with only one port and one ACU will suffice.

4) *Implementation Considerations:* It is important to note that the proposed alternatives can be tuned further to the algorithm under consideration by stripping any FBB's which are not strictly required. Furthermore, this communication hardware can be repeated as often as desired when a lot of data have to be transferred, but this will rarely occur for the medium-speed applications.

The transfers themselves can happen in a parallel, a serial, or an intermediate (hybrid) way. So they can be optimally matched with the required frequency of exchange.

### III. DEFINITION OF THE EXECUTION UNITS

According to our architectural approach, which has been defined in the previous sections, each processor consists of a controller steering a set of “strongly connected” execution units (EXU's) [4]. In its turn, *each EXU is composed of a data path, which is possibly extended with a local controller, and communicates with the others over a network of customized busses.* In this way, a very powerful customized data path is constructed which is fully tuned to the task to be performed.

#### A. Introduction of the Five Prototype Data Paths

With the experience collected from a number of typical design exercises, it has been found that a limited but well-chosen set of six prototype data path configurations suffices to span most of the target applications. This selection consists of: a static RAM/ROM with the corresponding address calculation unit (ACU), a hard-wired multiplier-accumulator, an iterative software-controlled divider (cf. long-hand division), a comparator unit with a MIN/MAX block for decision making (comparison of two operands), a normalizer supporting the use of floating-point operations, and finally, an add-shift-based AU. In the latter, most general EXU, the adder can be possibly exchanged with a more general ALU (as described in [22] and already applied in [43]) which allows us to execute several logical operations and some more advanced add-subtract variants. These data paths have emerged as important building blocks during the architecture synthesis phase in the design of several test vehicles (Section V).

Their general usefulness and applicability is evident. Data memory is always required; the multiplier, the divider, the comparator, and the ALU perform basic arithmetic operations common to many DSP algorithms, whereas the last two EXU's are also suitable for decision-making tasks (such as the evaluation of conditional expressions in branches). Finally, the normalizer performs scaling operations and supports the conversion between fixed- and floating-point formats, as required in applications with a very large dynamic range.

It should be noted that this choice has been made with the intention to make a tradeoff between flexibility plus efficiency, and the need to simplify the automated synthesis of the processor architectures [32].

The main advantage of these EXU's is situated in their *flexibility*. If, for instance, a multiplication has to be performed, a variety of realizations are available: either the hard-wired multiplier is selected, or a shift-add-based iterative algorithm is executed on the AU or on the divider data path. This type of flexibility helps to support the following strategy during the mapping of an algorithm to the architecture: operations which occur very frequently, and are correspondingly also time-critical, obtain a “reserved” and “more optimally suited” EXU. The criterion for the optimality depends on the algorithm specifi-

cations. So for low-speed audio and speech systems, an attempt will be made to save area by selecting the most flexible EXU's (such as the ACU or the add-shift AU with an adder/subtractor or with the general ALU) which can cover a broad range of different operations. In this way, a restricted number of EXU's can handle the entire algorithm. At the upper edge of the addressed spectrum, however, the speed rather than the area becomes the dominating factor. So then also the "accelerators," namely, the hard-wired multiplier-accumulator, the comparator, and the fast iterative divider will be required occasionally.

Next, an attempt is made to map also the other, less frequently occurring types of operations on the EXU's which have already been selected. Sometimes an additional EXU will have to be included, namely, when the specified seed requirement cannot be met by the ones already available. In this way, the instructions can be evenly distributed over the EXU's, and the number of idle cycles introduced during the microcode sequencing [17], [32] can be minimized. Consequently, the inherent parallelism available in the algorithm is fully exploited by means of the concurrently operating EXU's.

In order to further increase the area efficiency, the data paths are kept *parameterizable*: both *structural* (word-lengths, reg-file size, shift-factors) and *electrical* parameters (load capacitance of the buffer-drivers) can be assigned in discrete steps. Also, the basic circuits as used in some of the FBB's can be adapted to match the speed requirements: an adder/subtractor can operate linear in terms of the signal word-length (Manchester carry-chain, carry bypass, etc.) or sublinear (carry select or look-ahead). Finally, also the degree of pipelining can be a parameter, such as in the multiplier/accumulator unit.

Along with the modifications already mentioned, also the construction of the EXU's themselves can be slightly varied according to the specifications, in the sense that *idle operators are skipped*. If, for instance, the final accumulation is not required in the multiplier-accumulator, it could be left out of the prototype-EXU configuration in order to save area. Furthermore, also another cell type can be selected: not all latches have to be externally controlled or statically store the information; and sometimes it is better to exchange a register file with a single dynamic register.

### B. Timing Considerations

Throughout this paper, a clock period of 100 ns will be assumed for the synchronous operations, evenly distributed over two nonoverlapping clocks  $\Phi_1$  and  $\Phi_2$ . All input(s) to the EXU's are conditionally read into a master-slave type of register (which combines two latches controlled by  $\Phi_1$ , respectively,  $\Phi_2$ ). A *full register file* with up to eight words is *located at both EXU inputs, to be used as "foreground memory,"* which can be addressed in the same cycle as the other operations (this in contrast with a RAM). The use of these distributed register files helps to solve the data communication and

memory access bottlenecks by keeping temporary variables local. After the computation is finished, a tristate buffer conditionally drives the output bus of the EXU.

For all the EXU prototypes, only the minimal level of pipelining has been introduced so far, namely, one stage in each EXU except for the multiplier-accumulator where a second stage has to be foreseen if the word length of the multiplicand exceeds 10 bits.

### C. Detailed Discussion of the Six Data Paths

In the next subsections, each of the identified EXU's will be discussed in more detail. The nomenclature for the signals in the figures has been standardized in the following way: names of signal-nodes begin with the letter "N," control signals start with "C," and "status signals" or flags (e.g., sign of adder/subtractor result) are preceded with "S." The latter are routed to the control unit (see Section IV) in order to evaluate the decisions.

1) *Static RAM*: The static RAM will be used as "background memory" in the processors, but can also be located in between the processors to control the intercommunication (Subsection II-C). As already mentioned, a dual-port type is not strictly required for this task. The addresses are computed in an *address calculation unit (ACU) which supports indexing (addition of base-address and offset) and modulo-counting facilities*. Indeed, when reading, for instance, a table over and over again, it is interesting to use a programmable modulo-block for determining whether the offset within a table reaches a specific maximum after which it is reset to zero. The proposed ACU-design of Fig. 5 (similar to the one in [43]) is both compact and powerful. In addition to address computations, this ACU will be used for evaluating the index counter of FOR loops by means of the dedicated comparator blocks.

2) *Multiplier-Accumulator*: Most currently available DSP's (e.g., [10] and [20]) contain a parallel multiplier (Fig. 6) combined with a slightly wider accumulator on chip. When many variable-by-variable multiplications have to be implemented, this approach is the most attractive. Both inputs in Fig. 6 can be read from the same (when squaring is necessary) or different busses. Moreover, the data width at the accumulator is larger than the word length at the inputs. This extended signal range is only reduced again when the product has to be written onto the output bus. A special format block (programmable shifter) is introduced for this purpose. Another structural parameter is available in the accumulator where either an adder or an adder/subtractor is placed.

3) *Iterative Divider*: In Fig. 7, a very effective, iterative divider EXU is shown. The detailed operation of this software-controlled unit is discussed in the appendix. This EXU allows to compute efficiently the *quotient of two positive or negative two's-complement numbers, generating 1 bit each cycle*. The numerator ( $N$ ) and the denominator ( $D$ ) are initially read into Reg1 and Reg3, respectively. Next, the quotient ( $Q$ ) is bitwise assembled in

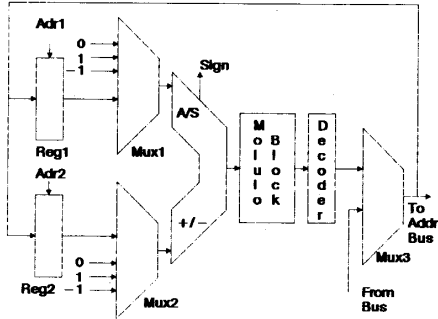


Fig. 5. Address calculation unit for the storage units between the processors. In the ACU, essentially two contributions for the address can be supported which are read out from the two register files at the inputs. These terms can, for instance, be the offset or base address and the index. Also, provisions for increment/decrement operations are made by means of the multiplexers. Finally, modulo facilities for table addressing and decoders for fixed tests are incorporated in this powerful EXU.

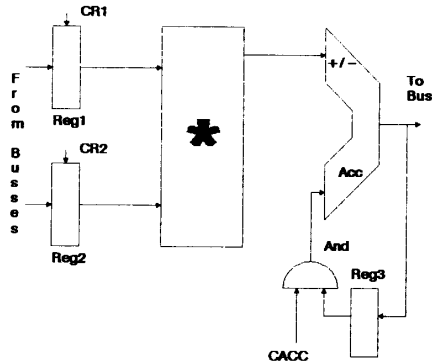


Fig. 6. Hard-wired multiplier-accumulator. At both inputs of this EXU, a register file with appropriate read and write addresses is available. The accumulator has both add and subtract capabilities. The feedback input can be disabled by the CACC signal for initialization purposes.

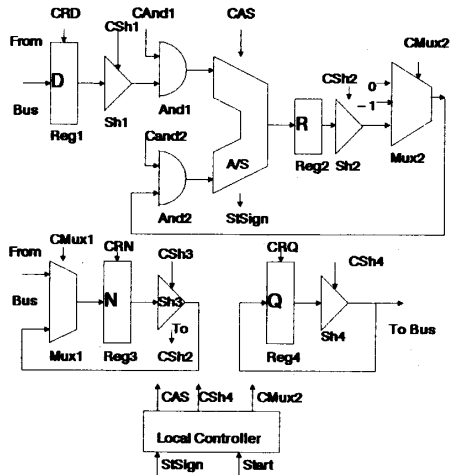


Fig. 7. Divider EXU. Flexible and powerful iterative software-controlled divider unit. Essentially, three parts can be identified in the data path. They are steered by means of a local controller (see the appendix).

Reg4 which is essentially a serial shift register when combined with the up-shifter Sh4. When the desired accuracy has been obtained,  $Q$  can be read out from Reg4 to another module over a dedicated bus. As opposed to conventional designs, *the correct alignment between  $N$  and  $D$  is taken into account automatically*, so no scaling has to precede this division.

During its operations, this EXU should be steered by a "slave controller," which is initiated with a START pulse from the supervising controller at the moment when the division is scheduled. In this way, the divider becomes a self-contained EXU module. The wide range of purposes for this module can be illustrated by the fact that the introduction of another simple slave controller also allows to perform, for instance, a parallel-serial multiplication in as many cycles as the coefficient word length.

4) *Comparator Unit*: In the fourth EXU, decision operations (comparison) are supported (Fig. 8). The status information available in the STH and STD signals indicates which of the two operands is the largest and whether they differ or not. This unit is suited to determine the *minimum or maximum of a (long) sequence of data*, or for the efficient evaluation of expressions of the type  $A = B$  or  $A > B$  (where  $A$  and  $B$  are variables) as occurring in *conditional branches*.

5) *Normalizer Unit for Floating-Point Conversion*: When *floating-point* operations have to be supported, they usually can be restricted to a critical but small part of the algorithm. Therefore, it is interesting to allow *mixing them with conventional fixed-point operations*. For this purpose, a normalizer unit for the conversion in both directions is required. Such an EXU is proposed in Fig. 9. For the conversion of a fixed-point number into a floating-point format, a unit to extract the most significant nonzero bit can steer a variable shifter. This results in a correctly scaled mantissa and an exponent. In the other direction, the floating-point number can control the variable shifter to produce a fixed-point equivalent. It should be remarked that this unit is also useful as a data programmable shifter, for instance, in block floating-point applications.

6) *General Add/Shift Arithmetic and Logic Unit*: The majority of the applications require a variety of logical and arithmetic operations. Most of the time, the number of required cycles per instruction is of less importance, and it would be unreasonable to include special-purpose FBB's for all the conceivable operations. A more interesting strategy consists of providing a *programmable multipurpose type of ALU* (Fig. 10). In this last EXU, both arithmetic operations (add/subtract-shift) and logic operations (AND-OR-EXOR) are combined. The main part of this EXU consists either of a fast adder/subtractor or a general ALU of the type described by Mead and Conway [22]. The latter module contains a carry block in combination with three fully programmable general function blocks (GFB's), namely, Kill, Propagate, and Result (Fig. 11) which require each four control signals. Most common special DSP functions can then be "modeled"

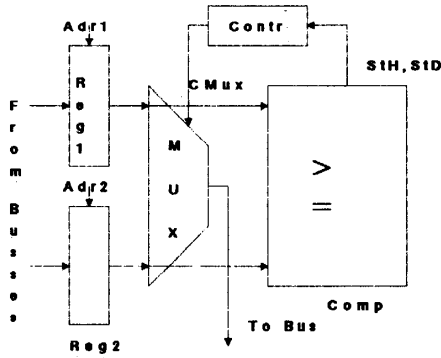


Fig. 8. Comparator EXU containing a MIN/MAX block suited for the evaluation of expressions during decision making or the min/max over a sequence of data. Again, a local controller is taking care of the internal operations.

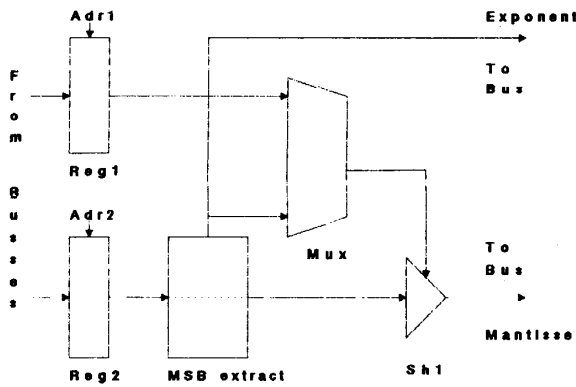


Fig. 9. Normalizer EXU. The MSB extraction unit can steer a variable shifter to convert a fixed-point number into a floating-point format, and vice versa.

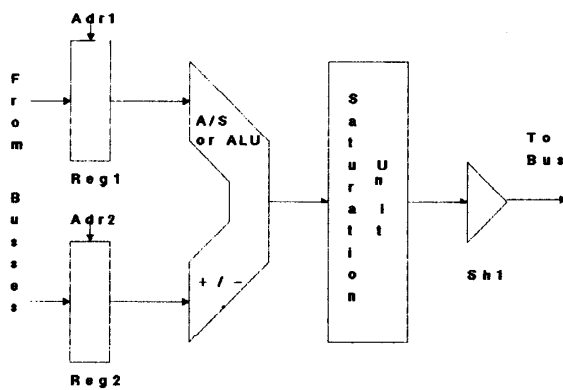


Fig. 10. Add-shift unit. General-purpose shift/add-based EXU containing two register files, an addition type of FBB, a saturation unit, and a logarithmic shifter.

by means of a concatenation of operations selected from the 16 basic ones in each GFB.

This FBB is potentially followed by a saturation unit which is very useful for overflow protection purposes. Af-

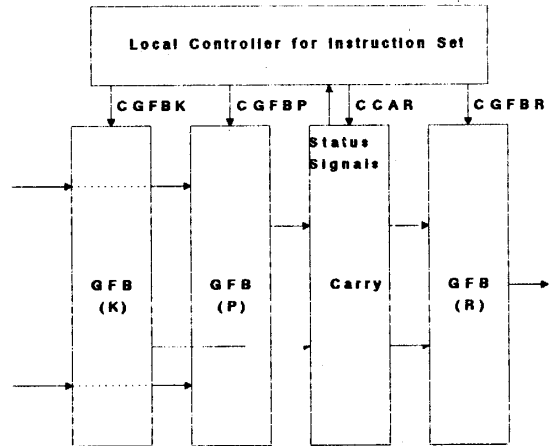


Fig. 11. Mead-Conway ALU. General arithmetic and logic unit with three general function blocks [22] each steered by a set of four control signals.

terwards, the result can be scaled by means of a logarithmic shifter. The arithmetic part of the data path can also be applied for a *shift-add-based multiplication* with a fixed or variable coefficient in binary or canonical-signed-digit (CSD) code [37]. An *iterative division* of two positive numbers with two cycles per quotient bit can be scheduled too. Apparently, this EXU is very powerful and all arithmetic operations which can be expanded into a combination of adds, shifts, and logical operations can be accommodated on it, even though it can cost a lot of cycles. In this way, it will become the "work horse" of the architecture, and only when the specific operations are performed too slowly with regard to the throughput specification, the other accelerator EXU's will be added.

It should be noted that variable shifters could be included in both of the branches before the ALU [4]. This would allow for a cycle reduction in many applications. However, the increased flexibility complicates the automated data path synthesis task considerably, and this option has therefore been omitted at present.

#### D. Layout Generation

As already pointed out in Section III-A, the described EXU's are parameterizable and, moreover, it should be possible to skip idle FBB's. Therefore, the module generators which provide the layout view of the FBB's in conjunction with the functional, circuit, and other level models, have to be powerful enough to support this flexibility. In order to reduce the area dedicated to interconnect, a following floor-planning technique has been adopted where the FBB's are connected to each other almost entirely by *abutment* (or fitting) [4]. The inter-FBB and even some of the inter-EXU connections will be accommodated in free tracks running over the cells (Fig. 12).

The list of the required data path FBB's for the four prototype EXU's (excluding the RAM) is *very limited*: conditionally clocked and "normal" static or dynamic

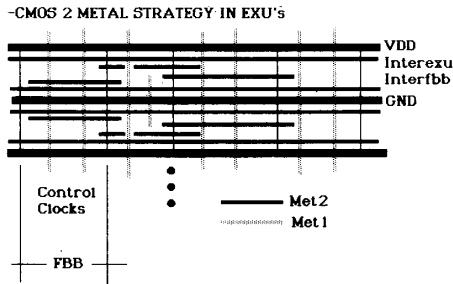


Fig. 12. General FBB connection strategy. Mainly, abutment is applied between the FBB's within the EXU's. In total, five predefined tracks are available to support routing over the FBB's.

registers, variable up and down shifters, AND gates, de(multiplexers), adder/subtractors, a logarithmic comparator, a general ALU, a parallel multiplier, and maybe an accumulator, but the latter could also be expanded as in Fig. 6. So the effort to maintain the library when the technology is updated can be kept within reasonable limits, especially if cells are stored in a procedural/symbolic form [8]–[9]. A library of cells has been developed in a double metal 3  $\mu\text{m}$  *n*-well CMOS technology.

#### E. Processor Assembly

For the inter-EXU communication, basically two options are available. In the *dual-bus strategy*, the EXU's are connected to at most two busses along conditionally controlled I/O ports. In this case, a general scheduling mechanism will be required in order to avoid data conflicts. This approach applies for most members of the new generation of general-purpose single-chip DSP's [10] where the possible communication channels are not known in advance. But this scheme suffers from the disadvantage of a *decreased time efficiency*, due to the bus-congestion problems which can cause the introduction of NOP's. Moreover, a more complex scheduling control is required, which will increase the constraints for the micro-code scheduler in the compiler, resulting in both a software- and a hardware-overhead.

In contrast, we have *chosen for an alternative where the EXU's are communicating over a number of dedicated busses*. With this strategy, the clocked transfers are scheduled at the end of the Phi1-phase and do not take a separate cycle, as opposed to the processor communication of Section II-C. The multibus fixed interconnection strategy, which is advocated here, requires somewhat more area for the busses than the dual-bus scheme. But this problem can be alleviated in a double metal technology by allowing routing over the cells, since enough routing channels (i.e., tracks) are available as already mentioned in Section III-D. Moreover, still at most a few physical tracks will be required over the entire range of the data paths because they will be split up into several local "busses." In this way, congestion and data conflicts will be completely avoided, again resulting in an increased efficiency compared to the general DSP's.

For the assembly of the EXU's into a processor, again several strategies can be applied [4]. In general, a fully automated or user-driven floor-planner should be responsible for assembling these blocks into a more or less rectangular frame.

#### IV. APPROPRIATE CONTROLLER ARCHITECTURES

In the present multiprocessor design methodology, the following phases can be identified in the synthesis of an algorithm onto a processor [4]. In a first step, the algorithm is partitioned into a number of subtasks which can be mapped independently [16], [32]. Then for each subtask, a number of EXU prototypes (Section III) are selected to produce a suitable data path. This selection can be based on the user's experience or can be performed by an easy to extend, rule-driven synthesis tool tuned to the target applications [32], [42]. Finally, the primitive operations in the algorithm can be scheduled and mapped into a control section [17].

The controller generation task heavily depends upon the nature of the selected architecture. A variety of suitable organizations is available in the literature [6]. The optimality of a particular architecture depends on the algorithm under consideration. We consider microcoded controllers, sequential state machines, glue logic, or a feedback shift register. All these alternatives will be discussed below. A novel "multiple-branch" architecture, totally suited for DSP applications containing a considerable amount of decision making, will be proposed first.

##### A. Microcoded Controller for Branching Purposes

1) *Motivation*: In many applications where decision making has to be handled, the microcode can become much more optimal when a multiple-branch instruction is available (for examples, see Section V). In the conventional architectures [10], [20], these multiple branches have to be split up over a number of consecutive conditional jumps (see also [6]). In this way, both instruction cycles (resulting in a decreased throughput specification) and ROM area are sacrificed in order to keep the control-sequence generation as simple as possible.

In this paper we present a controller architecture (Fig. 13) where this situation can be *avoided*. Instead of providing a special branch-instruction format such as the "JMP to address" in the conventional program ROM [6], *jump addresses are now separately stored in a second memory called ADRMEM*. During the normal program execution, the program counter PC is incremented with the INC operator. However, when some type of jump is required, the multiplexer MUX selects the output of ADRMEM instead to become the next PC value. In the case of a data-dependent branch, the status signals generated in the EXU-data path(s) have to be collected and interpreted first in a finite state machine (FSM) in order to decide on the appropriate next-state address. In either case, contrary to the solution applied in conventional microprocessors, *no separate cycle has to be foreseen to ex-*

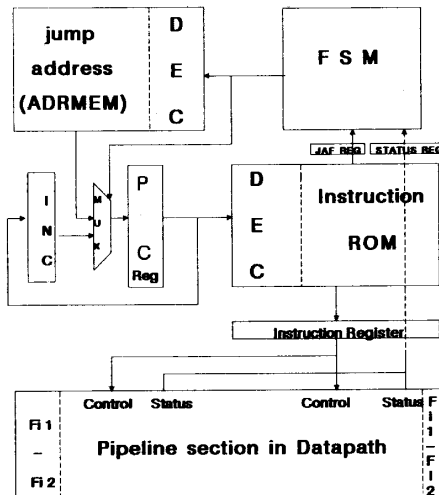


Fig. 13. Microcoded control architecture especially suited for multiple-branch instructions.

ecute the jump instruction as the normal arithmetic operations can proceed simultaneously, in parallel.

It should be stressed that, in this paper, especially the architectural aspects are investigated. Other researchers have already proposed to "compact" the microcode by means of advanced scheduling techniques on conventional microprocessors (e.g., [11]). But, in that case, the architecture of the control units is not suited to fully exploit the available parallelism. Moreover, multiple-branch instructions cannot be accommodated either. These constructs are only handled in the new design of Fig. 13.

2) *Algorithmic Aspects (Programmer's View)*: If a branch is executed unconditionally, the finite state machine (FSM in Fig. 13) can generate the appropriate address based on the value of a restricted number of bits grouped together in a jump-address field (JAF) which is reserved in the instruction-register IR. This information is required by the FSM to distinguish between the different program steps where jumps do occur. The JAF field can remain all zero in case the PC just has to be incremented. Otherwise, these bits represent the newly considered location in the program ROM (in a coded form). On the other hand, *conditional jumps* will depend on the status bits available from the data paths, from the address calculation unit (ACU) for the RAM's, or other information sources (see Section III for examples). Hence, the appropriate status bits have to be included as additional inputs to the FSM. In this way, the desired location of the ROM word, which should be evaluated during the next instruction cycle, can be derived. When the Boolean expression implemented in the FSM remains "false," a normal PC increment will be activated. A "true" condition, however, results in a jump to the desired address stored in ADRMEM. As already mentioned for the unconditional case, the branch address will also depend on the actual ROM location and thus on the contents of the corresponding JAF jump-address field.

The same strategy applies for a *multiple-branch instruction* too. But the appropriate jump address now has to be selected from a number of alternatives stored in ADRMEM, depending on the actual set of status-bit values. In this case, the JAF field is not associated with the jump address any longer, but only with the instruction where the multiple branch is located.

*Subroutines* can be realized too, in a restricted way at least, making use of special status bits which are cleared or set prior to the jump to the subroutine body. However, relatively more hardware has to be spent than in the conventional architectures where a dedicated stack is available. Notice that the restrictions mentioned here do not apply for the type of "subroutine" encountered inside a FOR loop.

3) *Floor-Plan and Implementation Considerations*: A generally applicable floor-plan strategy is difficult to be presented, as the constraints imposed by the corresponding data path(s) and the relative values of the structural parameters such as the aspect ratio's will influence the "optimal" solution in a major way. Therefore, the *module generator in the controller environment should select an appropriate controller floor-plan from a set of stored ad hoc approaches* for each particular application.

The FSM can be split up in two parts as illustrated in Fig. 14. The output of the "status-FSM" (FSM1) can be thought of as a set of condition-code bits (register CC). When necessary, these CC bits can be fed back to the input of FSM1 (dashed line) and they could be set or cleared by means of a dedicated instruction field in the IR. This solution becomes more attractive when the information stored in previously generated CC-bits can be reused in the Boolean expressions for other jump conditions. However, the tradeoff between the two alternatives (with or without CC-register feedback) cannot be solved without taking the actually involved parameters into account, and thus depends on the application.

It should be noted that when the number of (multiple)-branches is relatively large (50 percent or more) in comparison to the "normal" arithmetic instructions (and in practice this situation appears to occur frequently), it turns out to become more efficient to implement the triplet or the *jump-address memory, the program counter, and the incrementer by means of a full "sequencer" FSM* (e.g., a PLA). So the entire controller would then consist of the program ROM (including the jump-address field) steered by the "address" information generated in a partitioned FSM (or PLA) as illustrated in Fig. 15.

It can be concluded that as a crude strategy, the program ROM with an incrementer and a single FSM is very well suited for situations with relatively few branches (e.g., 20 percent or less). When more than 50 percent of the instructions contain a branch, the option with two FSM's and a ROM is more appropriate. In between these two extremes, the tradeoffs between all the different solutions have to be studied in more detail.

4) *Timing and Pipelining*: In the controller of Fig. 13, three pipeline stages (each covering one clock period) can

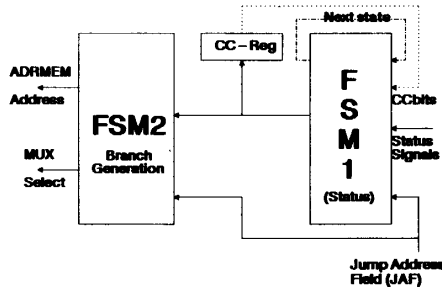


Fig. 14. Block diagram for the finite state machine. This partitioned FSM fits into the black box of Fig. 13.

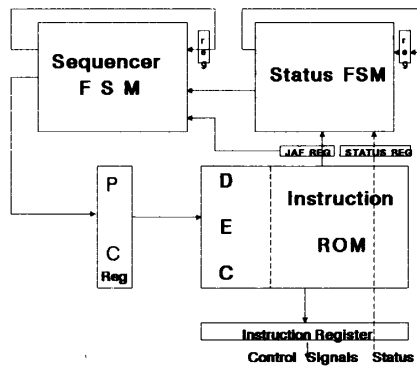


Fig. 15. Prototype controller implementation. A program ROM for generating the control signals in the instruction register is steered by a sequencer FSM, which is in turn controlled by a status FSM.

be identified in the critical control path: the first one is situated between the IR and the location where the status signals emerge from the data path. A second stage extends up to the evaluation of the ADRMEM. The last section is situated between the PC (output of MUX) and the IR which closes the loop again.

One important consequence resulting from this strategy constitutes the required *delay of two cycles between the instruction where status bits are generated and the program step where the scheduled branch address is validated* depending on the conditional expression coded in the FSM. So, in between, two “independent” instructions should be introduced. This situation is similar to the delayed branch option available in RISC architectures [26]. If these operations are not available, NOP’s have to be included. As a result, strategies have to be developed in the microcode scheduler [17], [32] to avoid these idle cycles as much as possible.

Anyway, if the multiple-branch instruction would be implemented with a number of successive conditional jumps, the same problems would occur and the number of required execution cycles would even grow considerably larger [6].

5) *Evaluation in Our Multiprocessor Environment:* In Table I, a comparison with regard to the required cycle count is made for three different program constructs re-

TABLE I  
COMPARISON BETWEEN THREE CONTROLLER ARCHITECTURES EVALUATED FOR THREE IMPORTANT PROGRAM CONSTRUCTS. THE INSTR ENTRIES ARE INSTRUCTIONS INDEPENDENT OF THE CONDITIONAL OPERATIONS. THE DEC, JMP, AND CJMP INSTRUCTIONS ARE DECREMENT, JUMP, RESPECTIVELY, CONDITIONAL JUMP OPERATIONS

Program construct	Classic Controller	RISC-based Controller	Solution of Fig.4.1
FOR-loop	DEC CJMP NOP	DEC CJMP INSTR	INSTR+DEC INSTR INSTR+CJMP
Overhead	3	2	1
IF THEN ELSE	CJMP NOP IF block JMP NOP ELSE block	CJMP INSTR IF block JMP INSTR ELSE block	INSTR+CJMP IF block INSTR+CJMP ELSE block
Overhead	4	2	0
Multiple Branch (with N options)	CJMP NOP Case 1 JMP NOP ... CJMP NOP Case N	CJMP INSTR Case 1 JMP INSTR ... CJMP INSTR Case N	INSTR+Jumps Combined Case 1 Case 2 ... Case N
Overhead	4N-2	2N-2	0

lated to decision making: a FOR loop, a single (un)conditional jump, and the multiple branch. Three different controller architectures are considered: a classical microprocessor controller with a ROM, PC, and INC; an RISC-based processor [26] including the “delayed branch” option; and finally, the proposal of Fig. 13. For each of them, the required number of “overhead” instructions is calculated, i.e., program steps in which no arithmetic operations directly contributing to the execution of the algorithm are performed. In all cases, *our architecture outperforms the other solutions*, and this with a relatively small hardware overhead, if any. In fact, the test on the FOR-loop index could for instance be shared with normal RAM-address calculations on a single ACU-EXU.

One of the primary reasons for these results stems from the fact that the jump address mechanism in our architecture, i.e., the status FSM and the sequencer, is completely separated from the control signal generation by means of the program ROM and the PC. This approach is entirely different from conventional controllers [6] where both control words and jump addresses are combined into a single “microcode” program ROM. Obviously, such an organization is very attractive in general-purpose applications where the entire program has to be loadable. However, when this restriction is removed, the partitioned architecture, as presented here, becomes much more suitable due to its flexibility and hardware efficiency.

It should be stressed, however, that the figures in Table I apply to the best cases, i.e., when enough “independent” instructions can be introduced between tests and the actual jump execution [see Section IV-A-4)].

It can be concluded that the controller architecture presented here allows us to perform most of the program constructs required in the decision-making part of an algorithm in a close to optimal way. Features such as (nested) subroutines can still be realized in a suboptimal way. Depending on the application, the architecture can be "stripped," and only the essential building blocks have to be retained. For instance, in some cases, no CC-register or even an FSM is required. Therefore, because of its generality and flexibility, this type of microcoded controller and its many "stripped" versions will be selected with the largest priority in the first version of the silicon compiler which is being developed at IMEC. It covers indeed the entire range from rather straightforward programs with few jumps to very complicated algorithms containing multiple branches.

### B. Other Control Structures

The microcode ROM approach is very general, but for particular cases other solutions can become more attractive, especially in terms of area. It should be stressed, however, that the burden for the microcode compiler increases considerably if such a choice has to be made. In the future, the CAD tools [9] will be gradually extended to allow for some of the other alternatives which are discussed now.

1) *Sequential State Machines*: If some part of the algorithm can be completely described by means of *conditional transitions between a limited set of "states,"* the sequential state machine as, for instance, discussed in [18] constitutes a very efficient implementation. In this case, the status bits generated in the data path sections, can be directly used to steer a single FSM which computes all required control signals. The FSM can, for instance, be realized as a Weinberger array or as a minimized and folded PLA with feedbacks (by means of an appropriate "module generator" [2]).

2) *Random Logic*: In the absence of (complex) decision making, the length of the periodically applied "program" usually remains very small, and this allows for the use of a combinatorial block with "random logic," realized as a two-level or multilevel logic array [38] which can be partitioned or folded, or by means of logic gates available in a standard-cell library. In some special cases, this type of controller *could even be reduced to a single line*, as it is the case for the STSIGN signal in the division-EXU of Fig. 7, or in the MIN/MAX block of Fig. 8 when the minimum of a data sequence has to be determined. During the implementation phase, the combinatorial or sequential logic can also be partitioned over several almost independent and smaller units, each taking care of a small subset of control signals.

3) *A Feedback Shift Register with Additional Logic Circuitry*: A simple 1-bit shift-register with a "running 1" which is fed back (Fig. 16), suffices when cyclic signals with a small period are necessary. When the period is limited to 3, all the required sequences can be generated

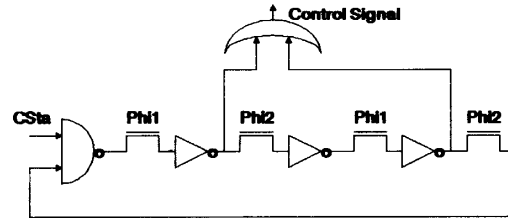


Fig. 16. Block diagram for a 1-bit feedback shift register. All possible sequences of control signals can be generated for a period of three cycles when three sections are cascaded.

[4]. Otherwise, a tool would be required to assign the don't cares in an "optimal" way to avoid the forbidden combinations. It should be noted that this architecture *can also execute simple branches* if the running 1 is controlled by means of additional multiplexers located in between the registers. Moreover, *more complex control sequences*, for instance, extended over several cycles or with more than a single 1, can be composed of several basic shift-register outputs by providing additional hardware, i.e., logic gates based on simple AND's and OR's (Fig. 16).

### V. TEST VEHICLES FOR THE MULTIPROCESSOR ARCHITECTURE

In order to evaluate the proposed multiprocessor architecture and to demonstrate the efficiency and power of the approach, a number of test vehicles will be discussed. Most of them have been *chosen* in the application domain of *speech processing*, namely, coding and recognition, where the desired throughput and sample rates in the subsystems range from reasonably low, i.e., 8 kHz at the DFT end, to medium in the dynamic time warping (DTW) unit. *Arithmetic* (in the DFT and DTW) as well as *complex decision-making* operations (in the pitch estimation and the endpoint detection) are involved. Also, a distinction has to be made between *block-oriented* algorithms such as the DFT-FFT and pitch extraction modules, and *scalar* operations (filtering) or *matrix-type* operations as for the dynamic programming in the DTW.

The architecture is further being evaluated with applications such as an adaptive echo canceler, a high-speed modem, an interpolator for use in a compact-disk unit, and a singular-value decomposition module.

Every subsystem of these different types of DSP algorithms can be accommodated on the different types of execution units presented in Section III. For almost all of them, a very efficient realization in terms of the cycle count, comparable to full custom approaches, can be derived by balancing the required operations as much as possible over the allocated hardware resources. If, for instance, a 1024-point Fourier transform has to be evaluated, the FFT program on *general-purpose processors* such as the ones described in [10] *takes at least 3 times more cycles* than the realization of Section V-A. The *flexibility of the architectural methodology is essential* in order to keep all execution units occupied during the many

instruction cycles which are available at the nominal 10 MHz clock rate.

Concerning the area costs, our architecture will in general save a lot of space compared to general-purpose processors because of the application-specific nature (Section V-D). So no overhead is implemented on chip. Of course, some area will be lost compared to the full-custom approach due to the use of the relatively small FBB-library and the fixed connection strategies. But this figure will be less than 20 percent in most of the cases, and the reduced design time offered by the automated synthesis environment CATHEDRAL-II [9], [32] will more than compensate this minor disadvantage. The area figures which will be mentioned in this section apply for an FBB library designed in a double metal 3  $\mu\text{m}$   $n$ -well CMOS process. The typical clock rates which can be achieved amount to 10 MHz. These sizes should be scaled if more advanced technologies (as for the off-the-shelf processors) become available.

In the next three subsections, a few of the applications will be analyzed in detail. First of all, the basic DFT and FFT operators will be discussed. This DFT/FFT operator can serve as the front end for a high-quality pitch extractor for speech, which is the topic of Section V-B. Some other typical blocks in a global speech analysis system, such as the one described in [23], will be handled in Section V-C. Finally, a decision feedback equalizer for ISDN will be discussed in Section V-D. For the latter application, a more detailed comparison to other approaches will be made.

#### A. Efficient DFT and FFT Implementations

The discrete (DFT) and the fast Fourier transform (FFT) [34] constitute very important subtasks in many DSP applications. Therefore, a study has been made of the performance of the architecture for both DFT and FFT. If the DFT algorithm is coded straightforwardly on a multiplier/accumulator execution unit (EXU) [see Section III-C-2]), the sine and cosine coefficients only have to be available for the  $N$  possible discrete steps of the argument, due to the periodic nature. So they have to be stored in a large coefficient ROM. In [13] an *alternative method* is suggested which *avoids this storage overhead at the cost of an increased number of computations* and a somewhat reduced accuracy. In that case, only the basic coefficient values for the primitive angles with indexes  $i = k = 1$  are required, so the ROM can be eliminated entirely. The DFT sums are now accumulated in an iterative way, and the influence of the sine and cosine coefficients is gradually built up for every complex partial sum  $P$ . In [4] an efficient architecture has been designed based on the proposed strategies in Sections II-IV. After the scheduling, the resulting code does not include any "idle" cycles without arithmetic operations. In this way, the DFT program requires only  $6 + 5 \cdot (N - 1)$  cycles per DFT value. Alternative signal processing architectures [10] will need about twice that amount.

If the DFT is used for the pitch extractor application in Subsection V-B, a cycle count of 82K cycles and an active area of about 8.4  $\text{mm}^2$  is needed.

Also, for the *radix-2 decimation-in-time FFT* scheme, an architecture has been derived [4]. The FFT algorithm can be executed *in situ* which means that the intermediate values  $X$  computed in every stage are overwriting the previous variables on that row. The *major problems* in implementing the FFT are located in the *address generation* for the many required operands. But the flexibility of our ACU design (Fig. 5) avoids the loss of idle cycles. Hence, similar results as in a full-custom approach can be obtained. In this way, an  $N$ -point FFT requires a total amount of only  $(2N + 2) \cdot \log(N) + 3$  cycles. The area-time tradeoff will thus be in favor of the FFT for large values of  $N$ . If, for instance, a 1024-point Fourier transform has to be evaluated, the DFT requires 2621K cycles whereas the FFT takes merely 20.5K cycles. Alternative *general-purpose processors* such as the ones described in [10] *take much more time*, namely, between 60K and 130K cycles.

It must be mentioned that even higher throughput rates can be obtained by partitioning the computation tasks for the FFT over multiple (pipelined) processors.

#### B. Pitch Estimation for Speech Analysis

An accurate algorithm for the *extraction of the "pitch" from a speech signal* has been proposed in [39]. The estimation is performed by studying the matching between the maxima of the frequency domain spectrum of the signal (obtained from a DFT on the windowed signal) and a set of predefined patterns (40 in total.) The system can be *divided into five subtasks* as illustrated in Fig. 17. After the DFT, the amplitude spectrum and the absolute maximum (threshold) are computed. The first 8 maxima above the threshold are derived in the third unit and compared to the 40 predefined patterns in module 4. In a last step, the precise value of the pitch is computed. This *block-oriented* algorithm exhibits both purely arithmetic (DFT-stage) and decision-making (harmonic sieving) operations.

In many ways, this application is very typical for a large part of the DSP class. Therefore, the *selection of the execution units* (Section III) *has been partly matched toward the functions which are required*, i.e., memory access (FIFO or RAM including the address calculation unit), fast multiplication/accumulation, iterative division, and add/subtract operations including multiple-branch decision-making dependent on the data path results (Sh/Add/Comp EXU and multiple-branch controller).

The algorithm has been manually, and partially automatically, mapped into the defined target architecture. The results are collected in Table II. It should be noted that with the obtained cycle counts, the achievable frame rate amounts to more than 3 kHz. In this sense, this exercise shows the potential performance of our architecture in terms of the achievable data rates. For an efficient reali-

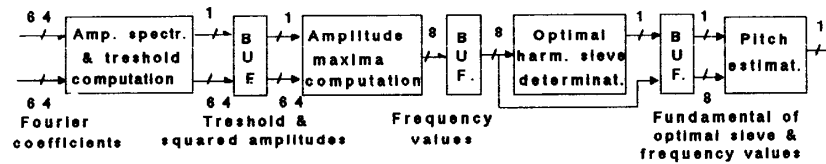


Fig. 17. Subtasks in the pitch extraction algorithm. Illustration of the partitioning into processors. In front of these 4 modules also, a DFT unit has to be included.

TABLE II  
SYNTHESIS EXERCISE FOR PITCH EXTRACTOR. RESULTS OF THE ACTIVE AREA AND CYCLE TIME ARE SHOWN FOR EACH OF THE PROCESSOR MODULES

Modules	Ampl	Max	Sieves	Pitch	Total
Data path (mm <sup>2</sup> )	3.0	4.3	8.3	6.1	29.5
Controller (mm <sup>2</sup> )	0.6	1.6	2.6	4.1	9.5
Buffers (mm <sup>2</sup> )	3.3	3.7	1.0	0.5	24.7
Total Area (mm <sup>2</sup> )	6.9	9.6	11.9	10.7	63.7
Cycle count	1225	661	2845	240	-

zation of the pitch extraction at the specified speed [39], the entire algorithm has to be scheduled on a single general-purpose data path. For this reason, also the flexible, general ALU unit of Section III-F has been included in the EXU list.

The active area needed for the total system including the DFT and the buffers equals 64 mm<sup>2</sup> in a 3  $\mu$ m CMOS process. Additionally, about 20 mm<sup>2</sup> will be needed for the interconnect. This will still fit onto a single chip as opposed to the 3 chip solution in [44]. As an example, the architecture for the Sieves module is illustrated in Fig. 18. It should be noted that an alternative realization for this module results in an area of 5.9 mm<sup>2</sup> (- 50 percent) and a cycle count of 4284 (+ 50 percent). A similar argument applies for the DFT which could be substituted with a much faster but also considerably larger FFT. These examples clearly demonstrate the *area-time trade-offs* which are feasible with our approach.

### C. Typical Architecture Modules for a Speech Analysis System

*Speech synthesis/recognition* [35] is becoming a very important topic in current DSP systems, especially for office and telephone applications. The kernel of such a speech recognition system consists of an assembly of dedicated functional units [33]. First, the feature extraction for the speech model has to be performed, based on short-time spectrum analysis with filter banks, or on linear predictive coding (LPC). This stage is followed by the normalization and segmentation algorithms where, for instance, the end-point detection for the words is executed. Next, the patterns have to be classified with respect to some reference patterns or templates. Finally, the syntactic and semantic analysis resulting in the actual speech understanding is performed.

The system which will be considered in this paper is the one described by Murveit *et al.* [23], [24]. It is in-

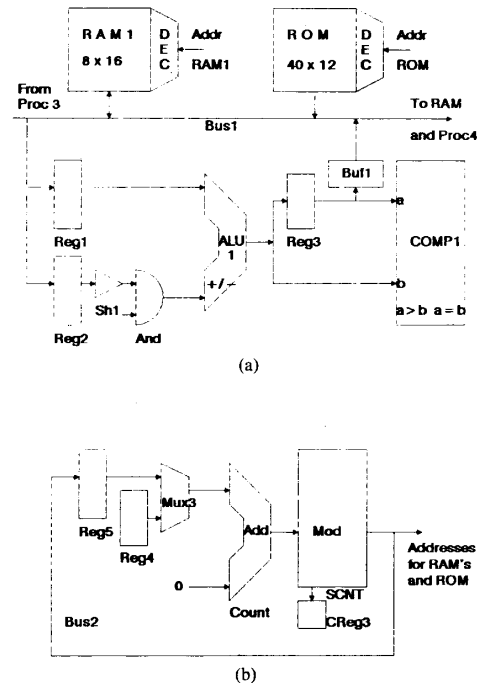


Fig. 18. Architecture for the Sieves module. For the data operations, a stripped Sh/Add/Comp AU, a local data RAM, and a constant ROM are provided (a). For the address computations and the control of the FOR constructs, an appropriately stripped ACU is available (b). A multiple-branch control unit (Section IV-A) has to steer the entire processor.

tended for *real-time speaker-independent, isolated word recognition*, although the extension to connected speech is possible. The vocabulary size is restricted to 1000 words. The system (Fig. 19) incorporates a front end providing the essential word features and a recognizer which is based on a dynamic time warping (DTW) scheme.

In [4], an efficient architecture has been proposed for each of the *front-end* tasks which meets the throughput specification. It is concluded that a single EXU suffices, with an estimated area of about 3.8 mm<sup>2</sup>. The data RAM which has to be included takes another 3.5 mm<sup>2</sup>. To these figures, the contribution of the I/O modules, the control unit, and the routing have to be added. The complete system will *occupy about 10 mm<sup>2</sup>*. For the *dynamic time warp module*, the total area for the EXU's can be estimated as 6.1 mm<sup>2</sup>. In addition to this, the row memory and the corresponding ACU contribute 6.7 mm<sup>2</sup>, and the input buffer plus the ACU for the template memory, take an-

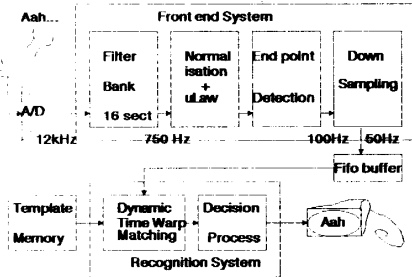


Fig. 19. Global Speech Recognition Scheme [23]: A front-end with 4 basic blocks is followed by a dynamic time warp module with the decision process. The speech is entered by means of a microphone and the recognition results are shown on a terminal.

TABLE III  
RESULTS FOR THE DECISION FEEDBACK EQUALIZER. A COMPARISON OF THE AREA COST AND THE TOTAL CYCLE COUNT IS MADE WITH THE ARCHITECTURES PROPOSED IN [20] AND [29]

Architectures	[20]	[29]	Sc2-4
Total area(mm <sup>2</sup> )	3 DSP's	20	6.6
Total cycle cnt	80	34	25

other 4.2 mm<sup>2</sup>. Hence, the *total active area* without control unit or routing amounts to 17 mm<sup>2</sup>. It should be noted that this pipelined architecture meets the throughput specifications with a hardware parallelism of only 2 [4]. This leads to the conclusion that the rigorous 2-D systolic approaches, such as the one advocated in [5], are not necessary in this type of application.

The total area estimations indicate the *feasibility of a single-chip solution* for the entire system, i.e., both the front end and the pattern matching DTW module. Only the 1.2 Mbit template memory has to reside off chip [4]. These results *compare favorably to the dedicated silicon implementation in [19] and substantiate the power of our approach.*

#### D. Decision Feedback Equalizer

As a last example, an application in the telecommunications domain will be discussed, namely, a decision feedback equalizer (DFE) plus timing recovery for ISDN purposes [29], [31]. The data rate amounts to 144 kbits/s which is very high. The *comparison of the performance of our architecture in terms of the cycle count, both with a general-purpose signal processor [20], and the dedicated multiprocessor architecture proposed in [29], is illustrated in Table III.* In [29], a 2 processor solution has been described. The die area for the DFE chip is 20 mm<sup>2</sup> in a 4  $\mu$ m nMOS technology. The DFE realization on the TMS320 [20] requires 80 cycles because of the less powerful controller architecture [31]. Hence, 3 DSP chips are needed due to the stringent throughput requirements. On our architecture, only one processor is required with a single add/shift ALU and a multibranch controller. The latter will allow us to evaluate the entire DFE without idle instructions in only 25 cycles. It can thus be concluded

that our architecture clearly *outperforms the other alternatives.*

## VI. EVALUATION OF THE MULTIPROCESSOR ARCHITECTURE

In this paper, a novel multiprocessor architecture has been introduced which can be used for the customized VLSI realization of complex low-to-medium-speed DSP applications (Section I). The alternatives and the decisions which have been made for the construction of the data paths (Section III), the controllers (Section IV), and the communication hardware (Section II) have been elaborately explored.

The advantages of this processor architecture stem especially from the flexibility and the efficient applicability in a broad domain. As already indicated in Section I, concerning the type of the operations, almost no restrictions exist: both block-oriented, scalar, vector- and matrix-based DSP algorithms can be accommodated. The efficiency can be guaranteed by providing several types of execution units, controllers, and interprocessor communication for the most commonly occurring data manipulation primitives (shift/add, multiplication, data transfer, decision making). In this way, *as opposed to general-purpose signal processors, the required data flow can be distributed evenly over the available hardware resources (i.e., EXU's).* Moreover, the proposed *control units contain enough concurrency to avoid the idle cycles* which are usually due to conditional (multiple) branches. Finally, the dedicated *communication in the processing elements can proceed without bottlenecks* at any level of the hierarchy at a very reasonable hardware cost.

Of course, these powerful resources will not always be required for all subalgorithms in a specific application. So wherever appropriate, a more general but slower type of EXU such as an ALU, or a less efficient communication unit, such as a bus protocol instead of switched RAM's or a smaller number of dedicated busses or multiplexers inside a processor, can be inserted. It is also possible to "strip" any of the EXU's and control units of "functional building blocks" which are of no use. In this way, the larger number of cycles will be traded off with a decreased area.

The combination of all these facilities has resulted in an *improved performance compared to existing approaches.* The difference with both general-purpose [20] and application-specific [29] architectures has been illustrated in Section V where several important demonstrator applications have been worked out. These include basic DFT-FFT modules, a pitch-extraction algorithm, speech recognition, and a decision feedback equalizer.

We believe that the proposed choice of the EXU's is practically independent of the available technology. It is in fact solely based on the properties of the signal flow graphs and the types of operations in the DSP application domain. Only the achievable clock frequency of the entire architecture, and thus the application range, would be increased in a more advanced technology.

The limitations of our methodology are obviously located in the achievable throughputs. At a 1 MHz sample rate, only 10 clock cycles remain available per data word. This is hardly enough to justify the processor approach where the necessary control units will begin to present an undesirable overhead. Therefore, at higher rates (above a few megahertz), the hard-wired bit-sliced type of architectures which are, for instance, discussed in [4] will be more appropriate.

At low throughputs (between 8–32 kHz depending on the complexity), another tradeoff will become apparent: the dedicated approach will be too powerful for the application at hand. In that case, off-the-shelf signal processors (such as in [10] and [20]) are less expensive. The lower cost will result both from the reduced design time, even compared to a “true” silicon compiler [7], and from the production size advantage (more samples will be sold). Moreover, also a better technology can be afforded due to the wider applicability. So as long as the DSP application can be programmed onto a single signal processor, there is no real interest in a semicustom (or full-custom) design. This situation will only be changed if the power consumption would present a problem or when the production size is extremely large (a few million samples).

Currently, a subset of the proposed architectural approaches has been selected which is “simple” enough to be realizable on a short-term basis. On the other hand, it is also powerful enough, as demonstrated with the benchmarks in Section V. The manual design of these test vehicles has allowed us to set up the specifications for an integrated CAD-tool box CATHEDRAL-II which should be able to *automate most of the “silicon compilation” tasks* [9], which is essential to reduce the design time for flexible architectures as the one presented above. These tools include architectural synthesis from a high-level behavioral description, microcode scheduling, and the final layout generation by means of dedicated module generators. Both synthesis and optimization tasks are taken into account in this transformation process [16], [32]. It should again be stressed that the *constraints* of the synchronicity, the DSP application domain, and the limited number of prototype EXU’s which we have imposed have contributed in a major way to the success of our automated synthesis approach. However, the restriction of equally long branches in the program flow after conditional jumps, as proposed in Section II-C, is not mandatory. Hence, it could be removed in the future in order to save instruction cycles. It should be noted, however, that due to the availability of the WHILE and the FOR statements, even applications such as variable-rate coding are already feasible with our methodology.

Concerning the *design for testability*, a consistent strategy covering all aspects of the test process has been developed [4]. In order to partition the modules and to make the FBB’s both controllable and observable, an incomplete scan-path approach in combination with the existing communication paths is adopted. This will restrict the area overhead to less than 10 percent. The dedicated test pat-

tern generation for most of the parameterizable FBB’s is based on a functional C-test approach. Only for the larger memory modules, a built-in self-test strategy is preferred.

## APPENDIX

### AN EFFICIENT, ITERATIVE DIVIDER IMPLEMENTATION

In this appendix, the detailed operation of the iterative divider introduced in Section III-C-3) will be elaborated. The ideas will be developed gradually, starting from the basic solution which is similar to the division algorithms performed in several DSP processors such as in [20] and [43], up to an extended and very powerful building block which can *handle positive as well as negative numbers, and which incorporates the correct initial alignment of the 2 operands.*

#### A. Basic Solution

In the data path of Fig. 20, two clusters are present: the one on top performs the long-hand division algorithm in an iterative way; the quotient itself is assembled bitwise in the smaller loop below. The numerator  $N$  (integer value) is read-in first, and stored in Reg2 (CAND2=0). During the next cycle, the denominator  $D$  is latched into Reg1 and stays there. Then, in the case of a restoring algorithm [34],  $D$  is subtracted from the partial result in Reg2 (which initially contains  $N$ ), for a number of consecutive iterations. At the end of each cycle, the sign information of the difference is available in STSIGN, allowing us to decide on the corresponding bit of the quotient: 1 if it is positive, and 0 otherwise. So the inverted STSIGN can be directly routed to the LSB input of a shifter Sh4, located in the small feedback loop. As the first bit constitutes the final MSB of the quotient, an up shifter is necessary. Furthermore, in the next iteration, either the previous content of Reg2 is restored (if STSIGN=1), or the new partial result obtained in the adder/subtractor (A/S). One of the options is then selected with the multiplexer MUX2 (which is directly controlled by STSIGN), shifted up over 1 position (in Sh2), and fed back to Reg2. If  $D$  and  $N$  contain  $W_D$ , respectively,  $W_N$  bits, in general,  $W_N - W_D + 1$  steps are necessary to perform this division algorithm, including the read-in phase. The data path has to be at least  $W_N$  bits wide.

#### B. Extensions for Including the Proper Alignment

Up to this point, it has been assumed that the numerator and the denominator have been previously aligned by some other part of the microcode program. However, this situation is not very realistic in practice where either  $D$  or  $N$  are not known in advance. So it would take a reasonably large amount of cycles or special-purpose hardware to perform the alignment. Moreover, a special EXU plus controller would have to be provided.

Therefore, it appears to be more appropriate to combine the alignment and the division into a single algorithm, as described below. At first it will be assumed that both  $D$  and  $N$  remain strictly positive.



```

000 0000 0100 0000 | 0000 1000
+111 1100 0 (= -8) | -----
-----
111 1100 0
+00 0010 00
-----
11 1110 01
+0 0001 000
-----
1 1111 010
+ 0000 1000
-----
1111 1100
+000 0100 0
-----
000 0000 0
+11 1110 00
-----
11 1110 00
+0 0001 000
-----
1 1111 000
+ 0000 1000
-----
1111 1000
(a)

000 0000 0100 0000 | 1111 1000
+111 1100 0          | -----
-----
111 1100 0
+00 0010 00
-----
11 1110 01
+0 0001 000
-----
1 1111 010
+ 0000 1000
-----
1111 1100
+000 0100 0
-----
000 0000 0
+11 1110 00
-----
11 1110 00
+0 0001 000
-----
1 1111 000
+ 0000 1000
-----
1111 1000
(b)

```

Fig. 22. Examples for the division algorithm. (a) Two positive operands are divided with each other. Nonrestoring division of positive numerator (64) by positive denominator (8). (b) A positive numerator is divided by a negative denominator to produce a negative quotient. Nonrestoring division of positive numerator (64) by negative denominator (-8).

add/subtract cycle, an addition has to be performed instead of a subtraction.

When the denominator becomes negative, some complications arise, as can be verified in Fig. 22(b) where an example is worked out. Either an additional initialization cycle has to be added during which the numerator is inverted (2-complement), or an additional quantization error of 1 LSB should be tolerated. In most of the envisioned applications, the latter can be allowed, so no further modifications are needed.

### E. Controller Requirements

The required steps can be summarized as follows. In the case of a nonrestoring algorithm, the adder/subtractor has to realize an addition if  $\text{Sgn}(R) \neq \text{Sgn}(D)$  (with  $R = N$  initially); the quotient bit will be  $\text{Sgn}(D)$ . When  $\text{Sgn}(R) = \text{Sgn}(D)$ , a subtraction is scheduled and the quotient is extended with the complement of  $\text{Sgn}(D)$ .

For the restoring algorithm, on the other hand, the  $A/S$  always (in every step) has to add if  $\text{Sgn}(n) \neq \text{Sgn}(D)$ , and subtract otherwise. In both cases, the quotient has to be extended with a 0 (1) if the partial result is negative (positive). The restoring operation (feeding back the previous rest instead of the new difference) has to be applied when  $\text{Sgn}(R) = \text{Sgn}(D)$ .

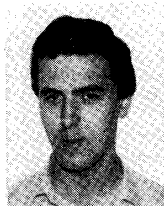
### ACKNOWLEDGMENT

The authors wish to express their gratitude to the colleagues in the project and at IMEC for the stimulating discussions and the useful comments.

### REFERENCES

- [1] J. Allen, "Computer architecture for digital signal processing," *Proc. IEEE*, vol. 73, pp. 854-873, May 1985.
- [2] M. Bartholomeus, L. Reynders, M. Pauwels, and H. De Man, "PLASCO: A procedural silicon compiler for PLA-based systems," in *Proc. IEEE Custom Integrated Circuits Conf.*, Portland, OR, May 1985, pp. 226-229.
- [3] F. Catthoor, J. Rabaey, G. Goossens, J. Van Meerbergen, R. Jain, H. De Man, and J. Vandewalle, "General data path, controller and inter-communication architectures for creation of a dedicated multi-processor environment," in *Proc. IEEE Int. Symp. Circuits and Syst.*, San Jose, CA, Apr. 1986, pp. 730-731.
- [4] F. Catthoor, "Architectural design strategies for complex DSP-systems in an automated synthesis environment," Ph.D. dissertation, ESAT, K. U. Leuven, Belgium, May 1987.
- [5] H. D. Cheng and K.-S. Fung, "VLSI architecture for dynamic time-warp recognition of handwritten signals," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 603-613, June 1986.
- [6] M. Davio, J.-P. Deschamps, and A. Thayse, *Digital Systems with Algorithm Implementation*. New York: Wiley, 1983.
- [7] H. De Man, "Evolution of CAD-tools towards third generation custom VLSI-design," in *Dig. Euro. Conf. Solid-State Circuits, ESCIRC*, Toulouse, France, Sept. 1985, pp. 256-256c.
- [8] H. De Man, R. Jain, G. Goossens, F. Catthoor, I. Vandeweerd, M. Pauwels, J. Vanhoof, and P. Six, "Development of a computer-aided design methodology for VLSI signal processing devices using multi-processing architectures," in *Dig. 2nd ESPRIT Tech. Week*, Brussels, Belgium, Sept. 1985.
- [9] H. De Man, J. Rabaey, and P. Six, "Cathedral II: A synthesis and module generation system for multiprocessor systems on a chip," presented at the Workshop NATO Advanced Study Inst. Logic Synthesis and Silicon Compilation for VLSI, L'Aquila, Italy, July 1986.
- [10] —, "Advanced chips usher in a new era for digital signal processing," *Electron. Design*, pp. 129-174, Feb. 1986.
- [11] J. A. Fischer, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, pp. 478-490, July 1981.
- [12] J. R. Fox and J. A. Fried, "Telecommunications circuit design using the SILC silicon compiler," in *Proc. IEEE Int. Conf. Comput. Design*, Port Chester, NY, Oct. 1984, pp. 213-219.
- [13] R. Holm and R. Jones, "Use discrete Fourier transforms to simplify signal processing," *EDN J.*, pp. 203-216, Apr. 1983.
- [14] R. Jamier, N. Bekkara, and A. Jerraya, "The automatic synthesis of data processing systems," in *Proc. IEEE Int. Conf. Comput. Design*, Port Chester, NY, 1986, pp. 64-67.
- [15] H. Joepen and M. Glesner, "Architecture construction for a general silicon compiler system," in *Proc. IEEE Int. Conf. Comput. Design*, Port Chester, NY, 1985, pp. 312-316.
- [16] G. Goossens, J. Rabaey, F. Catthoor, J. Vanhoof, R. Jain, H. De Man, and J. Vandewalle, "A computer-aided design methodology for mapping DSP-algorithms onto custom multi-processor architectures," in *Proc. IEEE Int. Symp. Circuits and Syst.*, San Jose, CA, May 1986, pp. 924-925.
- [17] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man, "An efficient microcode-compiler for custom DSP-processors," submitted for publication.
- [18] F. Hill and D. Peterson, *Introduction to Switching Theory and Logical Design*. New York: Wiley, 1968, pp. 267-331.
- [19] R. Kavalier, "The design and evaluation of a speech recognition sys-

- tem for engineering workstations," Ph.D. dissertation, U.C. Berkeley, 1986.
- [20] S. S. Magar, "Architecture and applications of a programmable monolithic digital signal processor—A tutorial overview," in *Proc. IEEE Int. Symp. Circuits and Syst.*, Montreal, Canada, May 1984, pp. 944–950.
- [21] P. Marwedel, "A new synthesis algorithm for the MIMOLA software system," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, Las Vegas, NV, June 1986, pp. 271–277.
- [22] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
- [23] H. Murveit, "An integrated circuit based speech-recognition system," Ph.D. dissertation, U.C. Berkeley, 1983.
- [24] H. Murveit and R. Broderson, "An integrated-circuit-based speech recognition system," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 1465–1472, Dec. 1986.
- [25] B. M. Pangrle and D. Gajski, "State synthesis and connectivity binding for microarchitecture compilation," in *Proc. IEEE Int. Conf. Comput. Aided Design*, Santa Clara, CA, Nov. 1986, pp. 210–213.
- [26] D. A. Patterson and C. H. Sequin, "RISC I: A reduced instruction set VLSI computer," in *Proc. 8th Int. Symp. Comput. Arch.*, Minneapolis, MN, May 1981, pp. 443–457.
- [27] P. G. Paulin, J. P. Knight, and E. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, Las Vegas, NV, June 1986, pp. 263–270.
- [28] Z. Peng, "Synthesis of VLSI systems with the CAMAD design aid," in *Proc. 23rd ACM/IEEE Design Automation Conf.*, Las Vegas, NV, June 1986, pp. 278–284.
- [29] S. Pope, "Automated generation of signal processing circuits," Ph.D. dissertation, U.C. Berkeley, Feb. 1985.
- [30] J. Rabaey, S. Pope, and R. Broderson, "An integrated automated layout generation system for DSP circuits," *IEEE Trans. Comput.-Aided Design*, vol. CAD-4, pp. 285–296, July 1985.
- [31] J. Rabaey and R. Broderson, "Experiences with automatic generation of audio band digital signal processing circuits," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, Tokyo, Japan, Apr. 1986., pp. 29.7.1–4.
- [32] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor, "CATHEDRAL II: A synthesis system for multi-processor DSP systems," in *Silicon Compilation*, D. Gajski, Ed. Reading, MA: Addison-Wesley, 1987.
- [33] L. Rabiner, "Pattern recognition algorithm for isolated and connected word recognition," in *Proc. IEEE Int. Conf. Comput. Design*, Port Chester, NY, Oct. 1985, pp. 745–748.
- [34] L. Rabiner and B. Gold, *Theory and Applications of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
- [35] L. Rabiner and R. Schafer, *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
- [36] V. K. Raj, "Another automated data path designer," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, Santa Clara, CA, Nov. 1986, pp. 214–217.
- [37] S. W. Reitwiesner, "Binary arithmetic," in *Advances in Computers*, vol. 1. New York: Academic, 1966, pp. 231–308.
- [38] A. Sangiovanni-Vincentelli, "An overview of synthesis systems," in *Proc. IEEE Custom Integrated Circuits Conf.*, Portland, OR, May 1985, pp. 221–225.
- [39] R. J. Sluyter, H. J. Kotmans, and A. Van Leeuwen, "A novel method for pitch extraction from speech and a hardware model applicable to vocoder systems," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, Denver, CO, Apr. 1980, pp. 45–48.
- [40] J. Southard, "Mac Pitts: An approach to silicon compilation," *IEEE Comput. Mag.*, 1983.
- [41] C.-J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Comput.-Aided Design*, vol. CAD-5, pp. 379–395, July 1986.
- [42] J. Vanhoof, J. Rabaey, and H. De Man, "A knowledge based CAD system for synthesis of multi-processor digital signal processing chips," submitted to VLSI'87, Int. Conf. VLSI, Vancouver, Canada, Aug. 1987.
- [43] J. Van Meerbergen, F. Welten, F. Van Wijk, J. Stoter, J. Huisken, A. Delaruelle, and K. Van Eerdewijk, "An 8 MIPS CMOS digital signal processor," in *Proc. IEEE Int. Solid-State Circuits Conf.*, Anaheim, CA, Feb. 1986, pp. 84–85.
- [44] P. Zuidweg, J. van Meerbergen, and M. van der Meulen, "Custom LSI chip-set for speech analysis," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, Apr. 1982.



**Francky Catthoor** (S'86–M'88) was born in Temse, Belgium, in October 1959. He received the engineering degree and the Ph.D. degree in electrical engineering from the Katholieke Universiteit Leuven (KUL), Heverlee, Belgium, in 1982 and 1987, respectively.

From September 1983 until June 1987 he has been a researcher in the group involved in VLSI design methodologies for digital signal processing, with Prof. H. De Man and Prof. J. Vandewalle as thesis advisors—first, until December 1984, at ESAT, KUL; and from January 1985 at the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium. In the summer of 1987 he spent a 2-month postdoctoral NFWO research fellowship at the University of Berkeley, CA. Currently he heads the Applications and Architectural Strategies group in the VSDM Division at IMEC. His research activities mainly belong to the field of architecture design for application-specific IC's intended for DSP algorithms, including design for testability; but he is also involved in the development of computer-aided design tools for the high-level synthesis and optimization of DSP systems. In these fields he has authored or coauthored about 18 papers.

Dr. Catthoor received the Young Scientist Award from the Marconi International Fellowship in 1986.

**Jan Rabaey** (S'80–M'83), photograph and biography not available at time of publication.



**Gert Goossens** was born in Genk, Belgium, on October 10, 1960. He received the electrical engineering degree from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1984.

He joined the VLSI Systems Design Group of the Inter-University Microelectronics Center (IMEC), Leuven, Belgium, where he is currently working towards the Ph.D. degree. His interests and activities are mainly in the field of computer-aided synthesis of architectures for DSP systems and in the application of analytical optimization techniques in VLSI design.



**Jef L. Van Meerbergen** 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.

**Rajeev Jain** (S'83–M'84), photograph and biography not available at time of publication.



**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, 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 ON SOLID-STATE CIRCUITS from 1975-1980, and was European Associate Editor for

that TRANSACTIONS from 1982 to 1985. He received a Best Paper Award at the ISSCC of 1973 on Bipolar Device Simulation, and at the 1981 ESCIRC Conference for work on an integrated CAD system.



**Joos Vandewalle** (S'71-M'79-SM'82) was born in Kortrijk, Belgium, on August 31, 1948. He received the engineering degree and the doctorate degree in applied sciences, both from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1971 and 1976, respectively.

From 1972 to 1976 he was Assistant at the E.S.A.T. Laboratory, Laboratory of the Katholieke Universiteit Leuven, Belgium. From 1976 to 1978 he was Research Associate, and from July 1978 to July 1979 he was Visiting Assistant Professor, at the University of California, Berkeley. Since July 1979 he has been back at the E.S.A.T. Laboratory of the Katholieke Universiteit Leuven, Belgium, where he is currently a Full Professor. His research interests are mainly in mathematical system theory and its applications in circuit theory, control, signal processing, and cryptography. He has authored or coauthored about 70 papers in these areas.