Synthesis of Pipelined DSP Accelerators with Dynamic Scheduling

Patrick Schaumont Bart Vanthournout Ivo Bolsens Hugo De Man[†*]

IMEC/VSDM, Kapeldreef 75, B-3001 Leuven, Belgium
*Professor at the Katholieke Universiteit Leuven

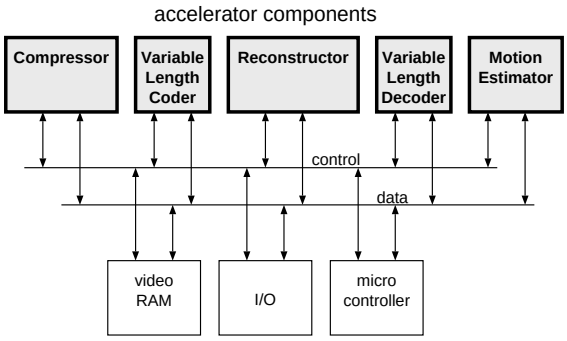

Figure 1: Architecture of a Video Phone

Abstract

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

1 Motivation

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

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

- High throughput requirements impose the usage of pipelined data paths.
- Area can be saved through the hardware sharing of different micro-instructions. The high complexity of the targeted functions make this sharing important.
- The accelerator processor has to be embedded in an overall system architecture.
- The accelerator functions can execute both at a manifest rate and a nonmanifest rate. An example of the former is the processing of a data stream out of a A/D converter. An example of the latter is the processing of data out of a processor core inside the system.

Automated synthesis systems for pipelined datapaths have been reported previously: SEHWA [8], PISYN [2], and SODAS [3]. All of these assume an infinite time loop and fix the runtime schedule at compile time. For an accelerator function, the infinite timeloop assumption does not hold and more flexibility is needed.

For this purpose, our work has concentrated on the following issues. The accelerator algorithm is defined by means of a signal flow graph (SFG). The accelerator datapath is defined as a set of *application specific units* (ASU) [7]. An ASU is a bit-parallel hardware operator, able to execute one or more micro-instructions. The micro-instructions are defined by subsets or *clusters* of the SFG. Each cluster corresponds to one micro-instruction, and the set of all clusters covers the complete SFG. This way, the accelerator algorithm corresponds to a *sequence* of micro-instructions.

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

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

Finally, a complete design flow, from algorithm specification to implementation is defined. For the synthesis, including pipelining and retiming of ASU components, we rely on existing data path synthesis tools and retiming tools [10, 6].

In the next section we will further detail the design

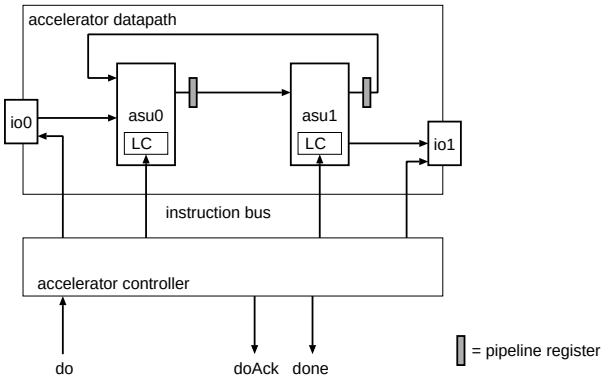

Figure 2: Architecture of the Accelerator Processor

steps to realise a data path architecture composed of ASU's. Section 2.2 elaborates on the architecture of the run-time schedule controller, and section 2.3 explains the operation of the controller. Finally, the described architecture and method will be demonstrated by synthesizing an FFT butterfly accelerator.

2 The Accelerator Architecture

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

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

The accelerator controller is steered by the system level controller through a processor interface. It also generates control signals for the accelerator datapath, as well as strobe signals for the input and output busses on that datapath.

2.1 Design of the Accelerator Data Path

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

a. Using the SFG specification of the accelerator function, the accelerator data path is defined by *clustering* the SFG. A cluster can contain functional operations (additions, shifts, ...), or else signal flowgraph inputs and outputs. SDF semantics [5] are assumed.

b. These clusters are assigned to hardware. In case of functional operations, they are assigned to ASU operators. Input and output node clusters of the SFG are assigned to input/output strobes, which are generated inside the acceleration controller. The SFG data precedences that cross the borders of the clusters define *interconnection busses* of the accelerator data path.

c. The set of clusters assigned to one ASU define the ASU micro-instruction set and composition. It consists of a local controller (LC) and a bitparallel data path. The local controller handles micro-instruction decoding and local decision making. As a consequence, there is no global decision making and thus no condition evaluation circuitry in the accelerator controller. Using state-of-the-art tools [10, 6], the ASU structure is obtained, and pipeline registers are inserted in the ASU data path by retiming software. The I/O timing behavior on the data and control ports of an ASU is known as the *timing view*. It is expressed as a number of clock cycles, representing latency between ASU input consumption and output production.

d. To find the number of interconnection pipeline registers we proceed as follows. The *cluster latency* is expressed as the number of clock cycles needed to evaluate that cluster. Using the cluster latencies as operation lengths, the clustered graph is scheduled. The cluster latency is equal to the ASU latency incremented by one. The increment of one ensures that at least one pipeline register will be present on an interconnection bus between two ASU clusters. The maximum combinatorial delay, or *critical path length*, of the accelerator datapath will therefore comply the timing specs that were used for the individual ASU's.

The ASU timing view is also used to model the micro-instruction sequence of the overall data path in a table, with one row per ASU and one column per clock cycle. Such a structure is called a *reservation table*, which is the basic data structure in the design of the accelerator controller.

2.2 The Accelerator Controller

The accelerator controller must perform ASU micro-instruction sequencing according to the cluster schedule, as represented in the reservation table.

In figure 4, the operation of the controller is illustrated by a an example. The reservation table that was derived in the previous example is on top. Below, the processing of three SFG frames is shown in terms of the processor interface pins. Time runs from left to right, one clock cycle at a time.

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

In the example the `do` pin is held high during 5 consecutive clock cycles. Acceptance of the `do` commands

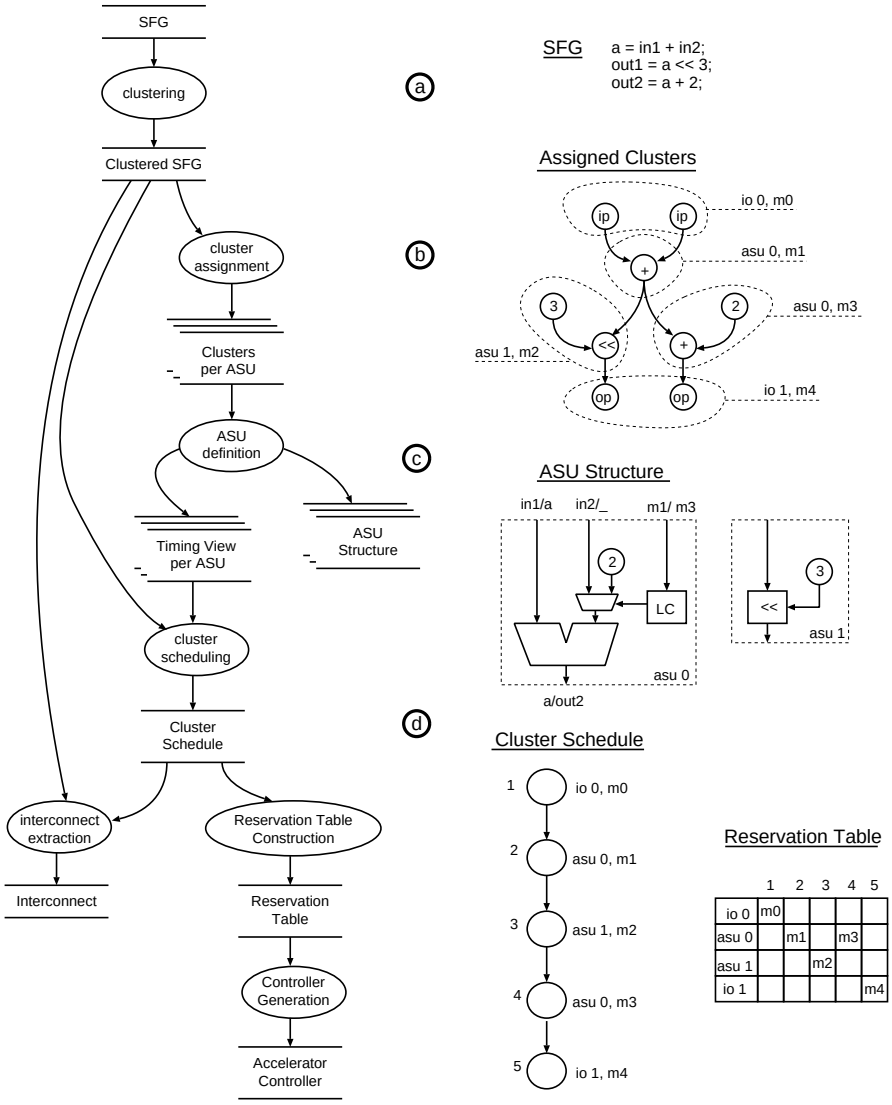

Figure 3: Design flow for the Accelerator Processor

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

- Run-time scheduling of ASU micro-instructions and detection of conflicts
- Interleaving of accelerator-level instructions

This leads to the accelerator controller hardware presented in figure 5. Three parts are discerned:

- The Micro-Instruction Shifter
- The Conflict Controller
- The Processor Interface

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

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

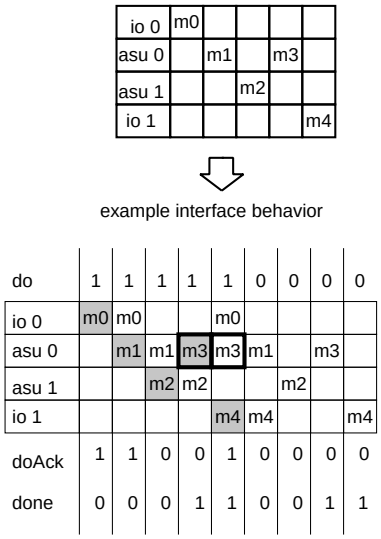

Figure 4: The Processor Interface Behavior

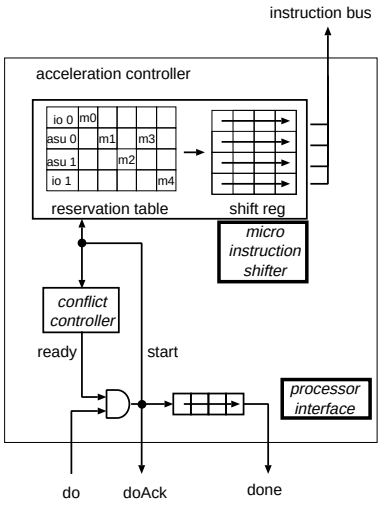

Figure 5: The Accelerator Controller

• The user requests accelerator execution through the `do` pin of the processor interface.

• The conflict controller indicates the shift register controller is ready to accept a new initiation.

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

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

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

2.3 The Controller at work

The conflict controller contains the core of the dynamic scheduling properties of the accelerator. A simple control strategy and architecture, which can be used as conflict controller, is due to Davidson [1]. His approach is based on dynamic modeling of data path resource conflicts.

The instances at which a conflict occurs after the *initiation* of a reservation table are called *forbidden latencies*. In the example reservation table, an initiation will introduce a pipeline conflict, due to `asu 0`, two cycles after this initiation.

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

To achieve dynamic modeling, the pipeline conflicts are marked as indices in a bit vector, which is shifted right as time proceeds. This bit vector, which is called *collision vector*, is numbered right to left starting from 1. Bit position `i` of this vector indicates wether a pipeline conflict occurs within `i` clock cycles. Hence, bit position one indicates wether a conflict occurs the next cycle, and thus wether a new initiation is possible at the next cycle.

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

Taking the initial collision vector, a state diagram can be constructed, with the states indicating initiation instances and edges carrying the initiation latencies. This state diagram is discussed in literature [9, 4]. The states are marked with a collision vector. The initial state carries the initial collision vector, and represents the moment just after initiation of the empty pipeline.

Out of the positions within a collision vector with zero bits, the initiation latencies can be derived. Out of the initial state in the example (state `10`), a new initiation is possible already the next cycle. At that moment, the initial collision vector is shifted one position, corresponding to a one cycle delay. At the same time, the pipeline conflicts introduced by this new initiation are annotated in the collision vector by OR-ing the initial collision vector into the current one. This results in a new collision vector (state `11`). Out of this state, an initiation latency of at least 3 cycles is required, as bit positions `1` and `2` are non-zero in the collision vector. This next initiation returns us to the

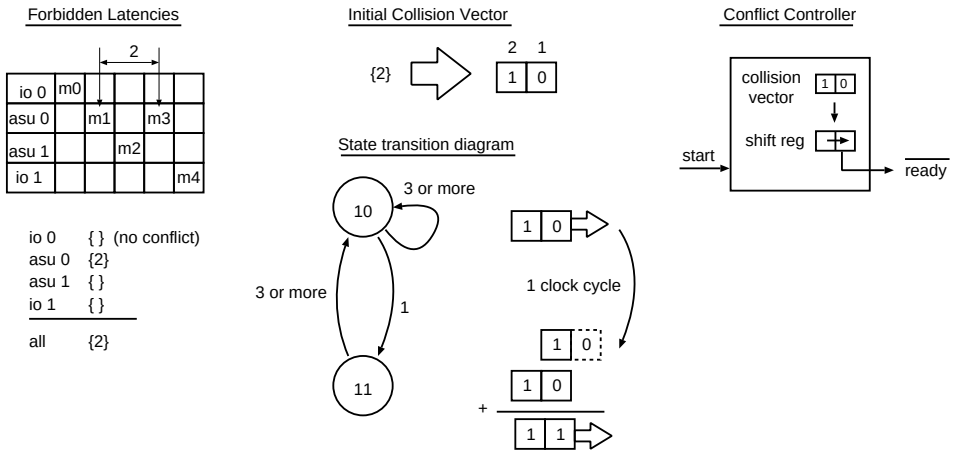

Figure 6: Construction of the Conflict Controller .

initial state.

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

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

The advantages of using this controller architecture are

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

3 Design example

The example concerns an accelerator to evaluate a Fast Fourier Transform *butterfly* operation [11], which is defined as follows

```
X = (A + B)
Y = (A - B) * W
```

All signals are complex, and will be denoted `Ar`, `Ai` and so on to indicate the real and imaginary parts respectively.

The upper left of figure 7 shows the accelerator integration. Complex input data `A` and B enters along a shared bus, while complex weight factors `W` use a second bus. The sources of this bus are for example RAM and ROM memories. The complex outputs `X` and `Y` are send to a shared output bus.

The operations composing the complex butterfly are shown on the upper right, together with an operation name, and operator assignment. This assignment fixes the clustering and is driven by two design issues:

- The boundary conditions imposed by the integration are modelled by `io` strobe assignments.
- Hardware sharing, which is obtained by assignment of similar operations to the same `asu` operator.

In the environment used to design the accelerator, the input SFG is represented graphically. The assignment of operations to clusters is indicated by drawing shapes and shades around the operator nodes. All nodes with the same shade and shape thus belong to the same cluster.

Next, the datapaths are designed by behavioral synthesis [10]. The implementation technology, 5.0 V_{dc} 0μ7 CMOS, fixes the timing views of the hardware. One pipeline section is added in the retiming of `asu 1` to balance the critical path to that of operator `asu 0`.

The timing views are backannotated to the clustered graph, which is scheduled. The resulting schedule is modelled in a reservation table, from which the collision vector and the conflict controller is obtained. Using the schedule, the interconnection network and accelerator controller is found. Finally, the overall architecture is obtained as a VHDL netlist, with a 16 ns critical path and 2.48 mm^2 active area. The design script is implemented in software to provide an integrated design traject from SFG to netlist.

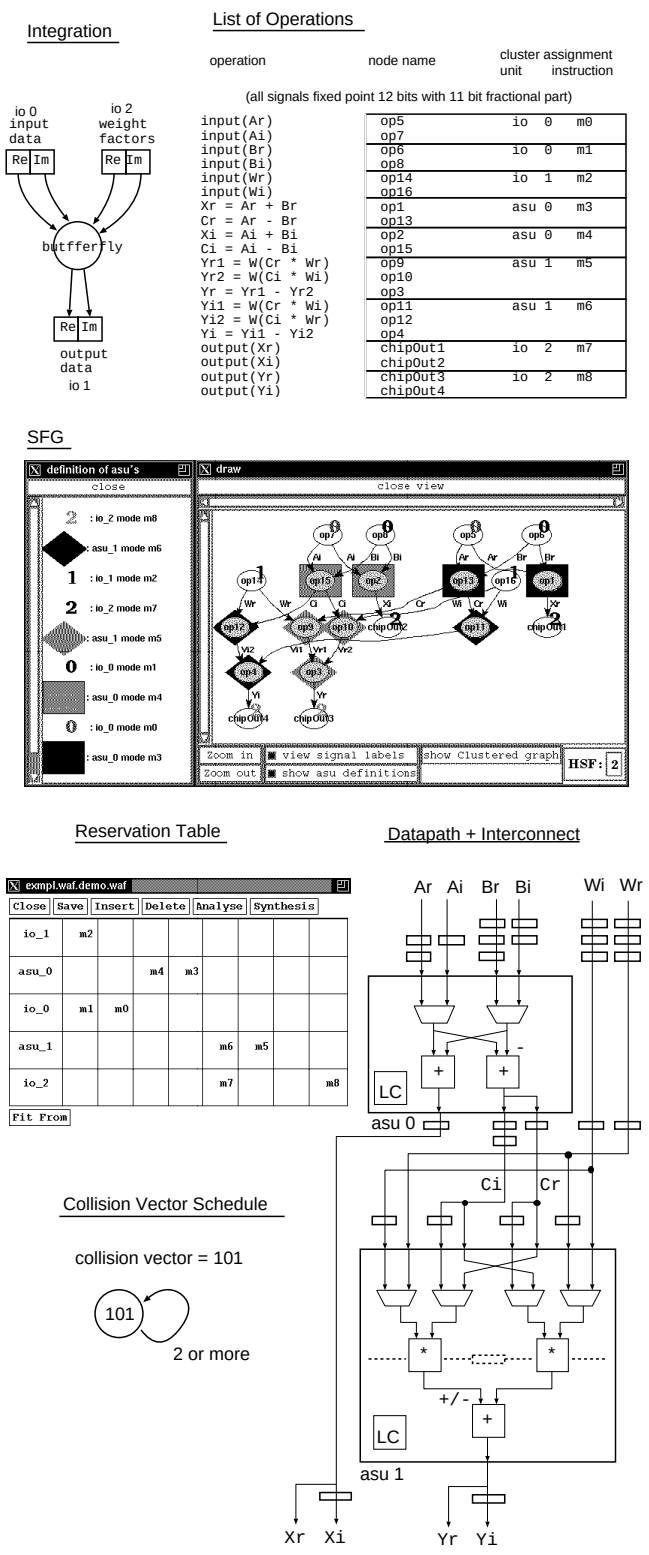

Figure 7: Design of a butterfly unit

4 Conclusion

A strategy was presented to integrate data path synthesis and retiming tools into a system component design environment.

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

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

Acknowledgements

The authors wish to thank Karl Van Rompaey and Serge Vernalde from IMEC for the constructive remarks during the writing of this paper.

References

[1] E. Davidson, L. Shar, A. Thomas, J. Patel, 'Effective Control for pipelined computers', *COMPCON 75*, IEEE, N.Y., 1975, pp. 181-184.

[2] K.S. Hwang, A.E. Casavant, 'Scheduling and Hardware Sharing in Pipelined Data Paths', *Proc. ICCAD*, Nov. 1989, pp. 24-27.

[3] H.S. Jun, S.Y. Hwang, 'Design of a Pipelined Datapath Synthesis System for Digital Signal Processing', *IEEE Trans. VLSI Syst.*, Vol 2, no. 3, Sep 1994, pp 292-303.

[4] P.M. Kogge, 'The Architecture of Pipelined Computers', *Hemisphere Publishing*, N.Y., 1981.

[5] E.A. Lee, D.G. Messerschmitt, 'Synchronous data flow', *IEEE Proceedings*, Sep 1987.

[6] P.E.R. Lippens, J.L. van Meerbergen, W.F.J. Verhaegh, D.M. Grant and A. van der Werf, 'Design of a 30 MHz, 32/16/8-points DCT processor with PHIDEO', VLSI Signal Processing VII, IEEE Catalog Number 94TH8008, 1994.

[7] S. Note, W. Geurts, F. Catthoor, H. De Man, 'Cathedral-III: Architecture-Driven High-level Synthesis for High Throughput DSP Applications', *Proc. DAC91*, San Francisco, Calif., 1991, pp. 597-602.

[8] N. Park, A.C. Parker, 'Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications', *IEEE Trans. Computer Aided Design*, Vol 7, no. 3, Mar 1988, pp 356-370.

[9] C.V. Ramamoorthy, H.F. Li, 'Pipeline Architecture', *ACM Computing Surveys*, Vol. 9, No 1, March 1977, pp 61-101.

[10] S. Vernalde, P. Schaumont, I. Bolsens, H. De Man, J. Frehel, 'Synthesis of high Throughput DSP ASICs using Application Specific Data paths', *DSP & Multimedia Technology*, June 1994.

[11] Vetterli M, and Nussbaumer H.J., 'Simple FFT and DCT algorithms with reduced number of operations', *Signal Processing*, 6, 1984, pp. 267-278.