CoolFlux DSP

The embedded ultra low power C-programmable DSP core

Hans Roeven	Jeroen Coninx	Marleen Ade
Philips PDSL	Philips PDSL	Philips PDSL
1101 McKay Drive	Interleuvenlaan 74-82	Interleuvenlaan 74-82
San Jose, CA 95131, USA	B-3001 Leuven, Belgium	B-3001 Leuven, Belgium
+1 (408) 474 9005	+32-16-390842	+32-16-390616
hans.roeven@philips.com	jeroen.coninx@philips.com	marleen.ade@philips.com

ABSTRACT

In this paper we present CoolFlux DSP, a new licensable embedded DSP core from Philips designed for audio applications. We discuss the hardware architecture as well as the software-programming environment, which was developed in parallel with the core. Also, we discuss optimizing techniques for power and code density and a design example showing the capabilities of CoolFlux DSP.

Design goals for the core were ultra low power consumption, a small core size and a small memory footprint. Furthermore the CoolFlux DSP is programmable in ANSI-C with a highly optimizing and very efficient C compiler. Based on many years of experience in the design of ultra low power electronics, the core was developed for portable audio applications, including audio encoding and decoding, sound enhancement algorithms and noise suppression. Targeted products include headsets, hearing aid devices and portable audio players.

The hardware architecture of CoolFlux DSP comprises a dual Harvard memory architecture; full 24/56 bits data paths, two 24x24 bit multipliers and 56 bit accumulators. Extensive addressing modes ensure efficient memory access without cycle penalties. This includes modulo protection and bit-reversed addressing for multiple dedicated address registers. The gate count of the core is 43k. In a 0.18u CMOS process the performance is 135MHz (worst case commercial conditions) which yields >1000 MOPS.

Full DMA capabilities for all memory spaces, a 64k addressable I/O expansion bus and maskable interrupts ensure easy integration in a system environment. Extensive start/stop instructions are used for reducing power consumption.

The programming tools for CoolFlux DSP have been developed by Target Compiler Technologies in close cooperation with Philips in order to guarantee highly efficient usage of the parallelism in the core. Included in the programmer's suite are: C-compiler, assembler, disassembler, linker and graphical debugger. For bit- and cycle true simulation an instruction set simulator (ISS) is available as well as a C/C++ simulation model. Compaction techniques are used for reducing the program memory code size.

Optimizing ANSI-C source code for memory usage and power consumption can be done interactively with the Target tool-suite. Various techniques are used to optimize code at ANSI-C source level. The ANSI-C compiler results are comparable to hand crafted assembly code, both in code size and in required DSP cycles.

An efficient implementation of an MP3 decoder serves as a design example. Highlights of this implementation are: 14.5MHz cycle requirement, 3.9 Kwords program memory and 9.8 Kwords data memory. These results have been obtained without assembly level optimizations. Core power consumption is less than 1mW in 0.18µ CMOS.

Various software applications are available for CoolFlux DSP.

© 2004 Global Technology Conferences, Inc.

DESIGN FOR C COMPILATION

In many modern DSP cores, C compilation has been an afterthought rather than a design goal. Often the hardware design will include orthogonal register sets and a regular data path to benefit C compilation, but still the compiler has to comply with the designed hardware. CoolFlux DSP was designed from a different approach. Starting point for the design, and one of the key initial goals of CoolFlux DSP is the C compiler. In fact the compiler was designed before the hardware of the core. A retargetable C compiler tool-suite from *Target Compiler Technologies* formed the basis for this development.

Tool Overview

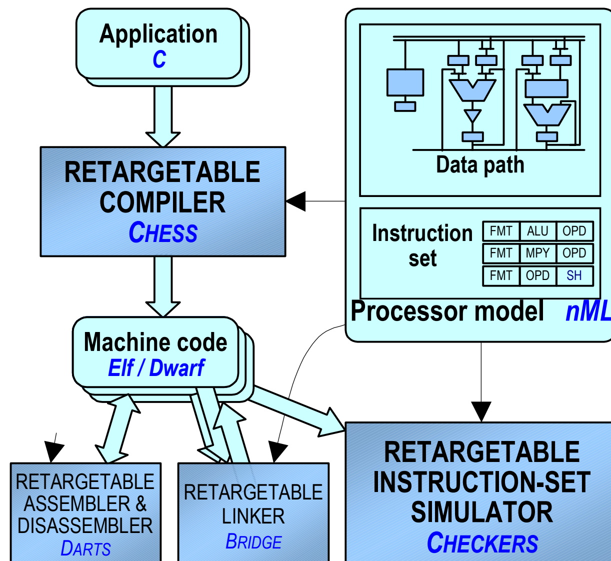

Figure 1 Target design environment

Processor Design Cycle

Using the *Target* tool-suite, a processor model and instruction set can be described using the *nML* language. This description is fed to the retargetable compiler (*Chess*), which uses it to translate C source code into machine code for the target processor. The resulting machine code can be simulated using the retargetable instruction set simulator (*Checkers*). Based on profiling results (code size, cycle count) the processor model and/or the application code are then adapted to improve performance for the next iteration. In the case of CoolFlux DSP the application code contained a set of algorithms typically used in audio decoding. After several iterations an optimal machine description had been defined, the process was frozen, a final tool chain was generated (CoolFlux DSP specific) and the processor was implemented. The processor implementation was then verified against thousands of test cases automatically generated by another *Target* tool: the *Risk* test program generator.

Instruction Set Design

In designing the instruction set for CoolFlux DSP a very good balance has been sought between Risc-like instructions and instructions supporting maximum parallelism. There are two major groups of instructions.

The *long instructions* use the entire 32-bit program word. These are, for instance instructions, that use a long immediate (direct memory accessing) or control instructions (mode setting, flow control instructions). The *long* group also contains some instructions that do pointer arithmetic including operations on the stack pointer, all using a long immediate.

The second major group constitutes the *arithmetic/move instructions*. These instructions actually combine 2 separate operations. The first part (*arithmetic)* could be one multiply-accumulate operation or one ALU operation, or two such operations in parallel. The *arithmetic* part is completely separated from the *move* part of the machine. The *move* part can perform a single move between registers or between a register and a memory, or a parallel move where it is possible to access both X and Y memory in a single cycle.

There are many instructions that influence the flow control in the CoolFlux DSP; both conditional and unconditional branching is possible. Many branching instructions are available in two forms, one with and one without delay slots. The compiler can thus optimally choose between both possibilities based on whether it is possible to schedule a useful instruction or not. Conditional arithmetic operations are used to avoid using conditional branching too often, thereby reducing cycle count.

The CHESS compiler is able to exploit all available parallelism in the design. This means that it is possible to perform two memory accesses with two pointer updates in parallel with two multiply-accumulate operations.

A very important feature of CoolFlux DSP is code compression. Code compression has been applied in order to reduce even further the already very efficient code generated by the CHESS compiler. This has been leveraged by the use of scheduling techniques within the compiler, which favor the generation of instruction sequences, which allow maximal compression to be used. The way these instructions are executed is exactly the same as before compression in number of cycles or execution time, but the code size will be smaller. In typical application code, program memory savings of 25-30% have been reached.

© 2004 Global Technology Conferences, Inc.

HARDWARE ARCHITECTURE

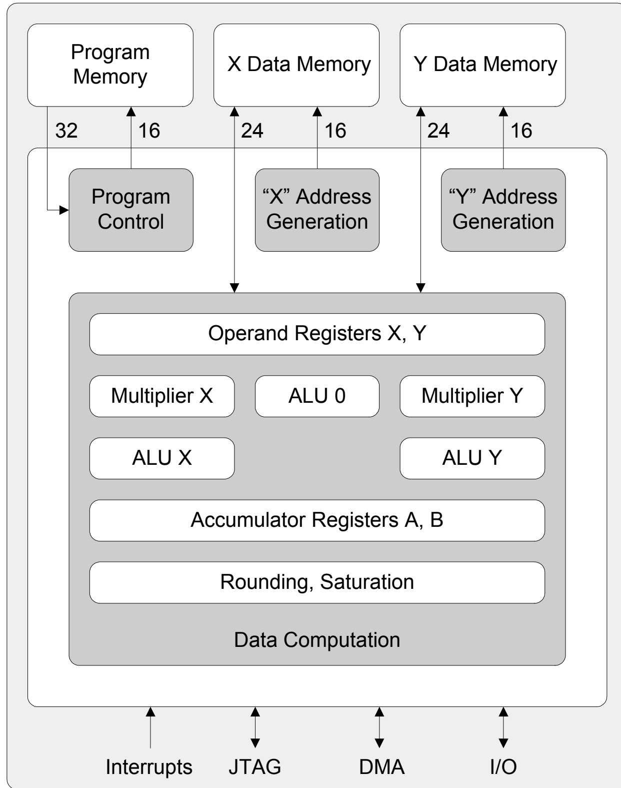

Figure 2 CoolFlux DSP architecture

Memory Architecture

CoolFlux DSP has a dual Harvard memory scheme. The program memory is 64K words of 32 bits while the X and Y memories, also 64K words, have 24 bits. All memories are synchronous. The X and Y memories can be combined to form a double precision 48-bit word memory. It is possible to read or write two data values to or from X and Y memories in one cycle while at the same time fetching another instruction from the program memory.

Program Control and Instruction Pipeline

The program control unit fetches the instructions, decodes them and schedules the pipelines by issuing control signals to the entire machine. Interrupts are also handled here. CoolFlux DSP guarantees an interrupt response time of two cycles. Very important is the loop control unit, which provides zero-overhead looping. A maximum of four nested loops is supported. Hardware loops are fully interruptible, even if they contain a single instruction.

CoolFlux DSP has a very simple and well-balanced pipeline structure. It has three major pipeline stages:

- Instruction fetch (one cycle)
- Instruction decode (one cycle)
- Execute (one or two cycles)

During the fetch stage, a new program word is read from the program memory and some initial decoding is done. In the subsequent stages, the instruction is then fully decoded and executed. The basic pipeline scheme of the CoolFlux DSP is kept very simple for several reasons. A deeper pipeline would gain some performance in terms of maximum operating frequency, but at the cost of additional power consumption and area. Since one of the main design goals of CoolFlux DSP was ultra low power consumption, a careful balance had to be kept between performance, power consumption and area. As a result CoolFlux DSP has a very well balanced pipeline with regards to timing.

Another reason for keeping the pipeline simple is the fact that the compiler must have a complete representation of the pipeline to be able to schedule all instructions in the most optimal way. A simple pipeline structure helps to get the most out of the compiler.

There is one exception where this simple pipeline scheme has been slightly modified. A dedicated multiplier, combined with an ALU, implements the multiply-accumulate operation (MAC). In order to balance timing, the MAC operation has been implemented with one extra pipeline stage. Two MAC operations can be executed per clock cycle. The implication for the compiler is that there will be certain restrictions in which operation can be scheduled in the cycle after a multiplication. These restrictions apply to data path units (ALU) or registers.

Address Generation

The entire X/Y memory space can be addressed via a long instruction using a 16 bit immediate constant. Indirect addressing is possible using any of the available pointer registers.

Many different post-modifications can be used for all pointer registers. Modulo protected addressing is supported as well as bit reversed addressing. The latter is used intensively in butterfly calculation used for instance in FFT algorithms. For every pointer register there is a modulo-register and a step-register. The pointer, step and modulo registers are always used with the same index and considered as groups of 3 corresponding register sets.

To save area, power and opcode space, the number of Y pointers is limited (X memory is used more intensively).

Next to the X and Y memories, CoolFlux DSP also has an IO memory space and a double precision XY memory mode with the same addressing features.

 © 2004 Global Technology Conferences, Inc.

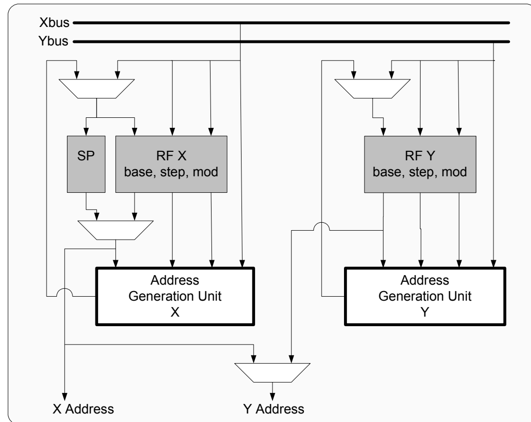

Figure 3 CoolFlux DSP Address generation

Data Path

The data path in the CoolFlux DSP consists of two multipliers, two ALUs and two round saturate and select units (RSS). There are four register files, respectively called X, Y (24 bits), A and B (56 bits).

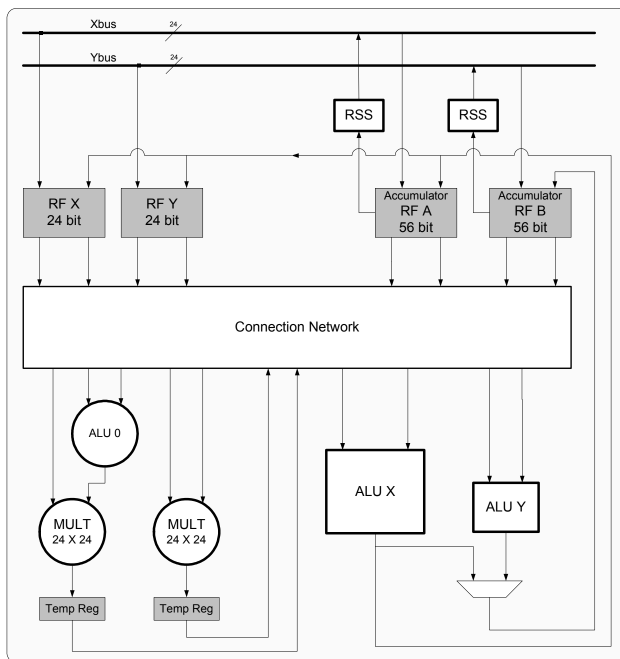

Figure 4 CoolFlux DSP data path

The data path can be seen as two separate sides, the X side (X multiplier, X ALU, X RSS, X register file and A register file) and a Y side (Y multiplier, Y ALU, Y RSS, Y register file and B register file). Both sides are highly asymmetric where the X side is used more intensively than the Y side. The Y side is smaller and mostly used for parallel arithmetic operations in parallel with the X side. This is done in order to save power and area. It also makes the job of the compiler easier. The X multiplier is actually a pre add/subtract multiplier. Two X or Y register values can first be added (subtracted) before being multiplied with a third X or Y register value. This multiplication can either be a signed-signed, signed-unsigned or unsigned-unsigned multiplication to support different forms of arithmetic calculations. The Y multiplier can only do a simple signed-signed multiplication and has no pre-add unit. The 48 bit outputs of both multipliers are stored in an intermediate register.

The X and Y ALUs can be used for a stand-alone ALU operation or for the accumulation in a multiply-accumulate operation. The X ALU is more extensively used than the Y ALU. The X ALU can perform several shift operations, bit wise operations, addition, subtraction, many compares, single operand operations or conditional operations. The operands can be two data path register values or one data path register value and a small immediate value.

Two RSS units, attached to accumulator register files A and B, reduce the size of the accumulators from 56 bits to 24 bits. First rounding is applied. After rounding, saturation is applied. For double precision arithmetic, it is possible to reduce the accumulator value to 48 bits. All arithmetic blocks explained here are operand isolated, making sure they only toggle when really necessary, in order to save power.

System Integration

CoolFlux DSP can be used in stand-alone mode or it as a coprocessor in a larger system. System integration is particularly important in an application where a host processor is required to load the CoolFlux DSP with program code or blocks of data to be processed. For this purpose two high-speed *DMA interface* ports were included. Program and data memory can be independently accessed by a DMA controller, each at one word per clock cycle. The *I/O bus* provides a simple but effective way to connect peripherals to the core, which can be addressed directly from the software.

 © 2004 Global Technology Conferences, Inc.

DESIGN FOR ULTRA LOW POWER

Holistic Approach

A total solution for ultra low power can only be obtained if the problem is approached at all levels throughout the design cycle. Every step in the entire design flow is important when designing for ultra low power, going from application software and architectural design, through implementation down to the layout of the DSP. Right from the start, when performing the iterations in the *Target* design environment the complete implementation flow was in place, allowing the designers to quickly study the effect of any architectural change in implementation and layout.

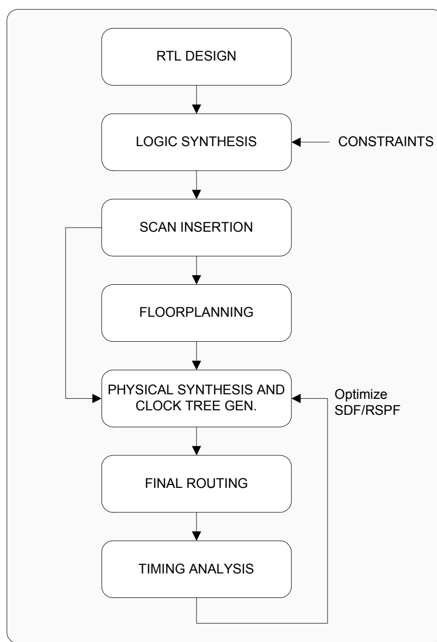

Figure 5 CoolFlux DSP implementation flow

Technology and Library Choice

An important prerequisite when designing for ultra-low power consumption is the choice of the right technology library. The process must be capable of aggressive voltage scaling. Logic as well as memory libraries must remain reliably functional at these low supply voltages.

Circuit design

In order to minimize power consumption at the circuit level, different design techniques were applied for CoolFlux DSP. *Clock gating* was applied throughout the design to reduce the switching activity as much as possible. The clock gating is applied on flip-flop level rather than on module level. Besides clock gating, also *operand isolation* is applied as much as possible. This means that whenever the result of an operation is not used (redundant operation), toggling of the circuit is inhibited. Memories are activated only when needed by use of a *memory select signal*. The effect of this measure depends on the type of memory used.

Compact Design and Layout

Having a small core area and keeping the amount of flip-flops to an absolute minimum greatly contributes to lower power consumption. To a large extent a compact design is an automatic consequence of having a simple, small pipeline structure. A well-designed streamlined data path also helps. One of the most important issues for layout is the row utilization. This factor should be as high as possible, since this influences directly timing, area and power. High row utilization can be achieved by reducing the number of hardware interlocks and write back loops in the design and hence reducing the amount of wiring. The hardware design flow used for CoolFlux DSP is shown in Figure 5.

An implementation with 16K P-memory, 16K X-memory and 16K Y-memory yields a maximum clock frequency of 135MHz in a standard 0.18μ CMOS process. Libraries for the process are characterized down to 0.9V supply voltage allowing aggressive voltage scaling. The core layout is shown in Figure 6 demonstrating the excellent grouping of functions.

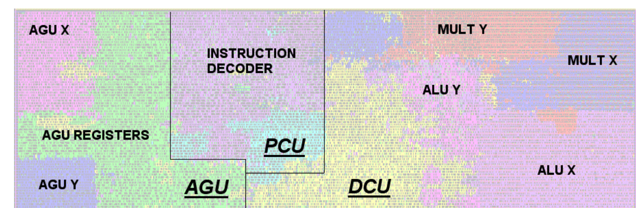

Figure 6 CoolFlux DSP layout

DSP Architecture

Important aspects of DSP architecture for low power are the pipeline structure, decoder implementation, register file structure and parallelism.

The well-balanced *pipeline structure* uses a single edge clock and allows a full clock cycle for the memory access. Having a good timing balance in the pipeline yields a maximum operating frequency, thereby giving the most room for voltage scaling which in turn gives quadratic power reduction. The pipeline structure is further optimized with the rest of the micro-architecture design resulting in simple interlocks and minimal bypassing.

Another way to save power is the use of a *segmented decoder*. Instead of having one large

 © 2004 Global Technology Conferences, Inc.

decoder, there are several small decoders that are only active when needed. This reduces the amount of toggle energy in the entire decoder, making the CoolFlux DSP instruction decoder extremely power friendly. Use of this technique relies heavily on the modular, orthogonal structure of the ISA.

Register file design can have a great impact on power consumption. CoolFlux DSP uses distributed register files, local to their respective computational resources. Together with the structure of the pipeline this gives significant advantages over a single multi-ported central register file. A single central register file would lead to many bypasses over different pipeline stages. The most stringent problem with a distributed register file is the efficient register allocation and scheduling. The CHESS C compiler handles this excellently. This problem is also intercepted by intelligent connections with the functional units.

System Design

CoolFlux DSP has different ways to start and stop the core. This is very important for low power as well as for the ease of external interfaces. This is a system level issue; CoolFlux DSP provides the right ‘handles’ to ease power saving at system level. Firstly, the core can be stopped via the software by using the end-of-program instruction. During this time, the clocks can be switched off until the core is restarted again. A dedicated input restart pin or an external hardware interrupt can be used to restart the core. CoolFlux DSP can also be halted via external pins. The clock can then be switched off.

Application Level Optimizations

Optimization for low power at the software development level translates into two design goals. *Minimizing cycle count* will allow the DSP in the final application to run at a lower frequency, which in turn allows further voltage scaling. *Minimizing memory usage* will reduce the memory size and access cycles, both yielding lower power consumption. Guiding information for the programmer includes memory mapping from the compiler and profiling information from the instruction set simulator. An efficient C compiler that can effectively use all the instruction level parallelism in the core is essential.

APPLICATION EXAMPLE

MP3 Decoder

An MP3 decoder supporting all defined fixed and free-format bit rates was developed as a test application. It supports MPEG1, MPEG2 and MPEG2.5. MP3 started as a synonym for the audio MPEG-1 standard, implementing its most complex layer 3 compression methodology. In the past few years, it was extended to lower sampling frequencies in MPEG-2 LSF, which gives better performance with lower sample frequencies at lower bit rates. MPEG-2.5 has decreased the sample rate even further. Our decoder is an ISO/IEC compliant full layer3 audio decoder.

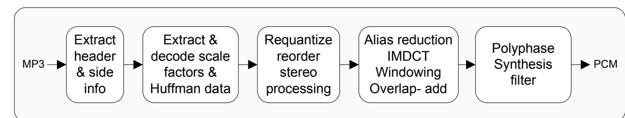

Figure 7 MP3 decoder block diagram

Software optimization strategies

Processor independent optimizations

This includes *reducing computational complexity and memory accesses* by introducing alternative algorithms. Memory accesses could be significantly reduced by exploiting symmetry in tables for instance, or by switching to block processing, allowing *reuse* of already loaded coefficients or data. Of course data copying should be avoided as much as possible. Circular buffers can often help in this case.

Another interesting strategy is to *move conditions to as high a level as possible*, extracting them from inner loops and functions, or by simply removing conditions when possible. Derivatives of a function could be made where a particular argument is replaced with a constant value, if this is a frequently used value. Functions should only be called if they need to perform a significant action. Otherwise the calling will cause too much overhead. Small functions called often should be in-lined. In general, *the more compile-time knowledge is exploited, the better*.

Processor dependent optimizations

The code should be *structured to best match the architectural features* of the processor, and to fully exploit its parallelism. One should bear in mind the number of available accumulator- and pointer-registers and limit their simultaneous use inside loops or code blocks. Otherwise costly spilling of

 © 2004 Global Technology Conferences, Inc.

these registers will be inserted, to save and restore them from memory.

Regarding pointer handling, one should select memory access strategies that favor pointer updates with step size 1 or 2, or with a constant value, as this matches best the hardware provisions. Variable step sizes are of course possible, but will cost more cycles.

The application layer also provides intrinsic functions for some operations, indicating to the compiler that an efficient hardware resource exists for its implementation. One example is *modulo addressing*. Another is a *maximum function*. The programmer should adapt the applicable parts of the code to use these functions.

The compiler provides a number of *pragmas* to help with this optimization process. With a pragma on a data array, one can *direct the data explicitly to X or Y memory.* E.g. for computing filters, the coefficients could be redirected to Y memory, and with the input data in X memory, parallel loads can occur. Other pragmas allow to specify the number of *nested hardware do-loop* levels in a function, to specify bounds on non-manifest loop counts to avoid tests and to enable *software pipelining*. Or to reduce data dependencies between different pointers for instance.

Compiler co-optimizations

During the development the compiler has been optimized such that it can detect structures in the code that can easily be supported by the hardware of the processor. E.g. the compiler could select the hardware support for modulo buffer addressing through the use of a special function in the C-code for updating the pointers. It has also been improved to make better use of the hardware loop provisions, which resulted in gains of 20% in functions with large loop counts.

Results

The resulting MP3 decoder implementation requires only *14.5MHz*, uses *3.9 Kwords program memory* and *9.8 Kwords data memory*. These results have been obtained without assembly level optimizations. Core power consumption is *less than 1mW* in 0.18µ CMOS at 0.9V supply voltage.

REFERENCES

[1] *Design Of Low Power Processor Cores using a Retargetable Tool Flow*
Gert Goossens, Dirk Lanneer, Peter Dytrych.
(to be published)

[2] *The design of a very low power MP3 decoder accelerator*;
Peter Dytrych, Marleen Ade, Jeroen Coninx, Johan David, Patrick Vandebroek.
DSP Valley, October 8, 2002

 © 2004 Global Technology Conferences, Inc.