xDSPcore: A Compiler-Based Configurable Digital Signal Processor

xDSPcore is a digital signal processor (DSP) architecture that enables time- and space-efficient execution of typical digital signal applications written in high-level languages. Our evaluation shows that the corresponding compiler can use all the DSP features as efficiently as a programmer coding in assembly language.

Andreas Krall
Ivan Pryanishnikov
Ulrich Hirnschrott
Vienna University of Technology

Christian Panis
Carinthian Tech Institute

DSPs find wide use in systems for real time processing of audio, video, and communication tasks. Typically, such systems must be inexpensive and consume little power, but provide high performance. These constraints, along with a relatively narrow application domain, have led designers to create special architectural features, as found in the Harvard architecture, VLIW (very long instruction word) architectures, and special addressing modes. Traditionally, software developers have programmed DSPs in assembly language for efficiency. This implies time-consuming programming, extensive debugging, and little or no code portability. With more complex embedded systems and higher costs for software development, programming such systems without the support of high-level programming languages becomes impractical. However, code generation for DSPs has high quality standards.

Despite extensive work on code generation for DSPs and embedded processors,[1] existing compilers do not generate target code that has acceptable efficiency, for two major reasons. First, identifying special functional units brings architectural information into the compiler, thus impacting compiler retargetability. The second problem that hardware specialization exposes is a lack of compilation algorithms for such architectures, and the computational hardness of known algorithms. In other words, applications that demand high computational performance at low cost usually require specialized architectures, which in turn frequently affects programmability. The tradeoff between performance and programmability spans solutions ranging from a complete hardware solution using an application-specific IC (ASIC) to a general-purpose processor.

Our new DSP architecture, xDSPcore, is a codesign of a C compiler and a DSP processor. Our approach is to introduce a hardware feature only if the compiler can support it. The goal of this technology is to make DSP applications programmable entirely in a high-level programming language like C, instead of assembly language.

Compiler-friendly DSP architectures

DSP application development in high-level programming languages is efficient only if the architecture fulfills the requirements of the

0272-1732/04/$20.00 © 2004 IEEE Published by the IEEE computer Society

language and the corresponding compiler. Before, instruction sets increased complexity to close the semantic gap between hardware and software. Now, with advances in chip technology, RISC design principles are in common use; returning to some of them would simplify an architecture and make it more compiler friendly. For example, instruction set architectures should be simple and regular for efficient use by compilers and for easy exploitation of hardware resources. All arithmetic operations should only operate on registers. Instructions should perform memory access via separate load and store operations. However, indirect addressing with offsets is necessary to support access to values held in a stack frame and to members of structures. Hardware pipelining should increase parallelism, but it also lengthens branch delays. With compiler support, instructions can move into delay slots, making the static prediction of the branch direction exploitable. Dynamic branch prediction would be very accurate, but needs more hardware resources.

To satisfy the computational demand of DSP applications, any architecture must exploit massive amounts of parallelism. Superscalar processor implementations and VLIW or vector architectures can use parallel functional units. Superscalar processor implementations, like the Intel Pentium IV or the AMD Opteron, execute instructions out of order and achieve the highest performance, but have a high implementation cost. VLIW architectures, like the Multiflow Trace or the Cydra 5, use compiler techniques for instruction reordering to reduce the hardware cost but result in bloated code size. Current DSPs, like the C60 series from Texas Instruments, the Blackfin from Analog Devices, or the SC140 from Starcore, use variable-length VLIW instruction sets. Short-vector parallelism is exploitable using single instruction multiple data (SIMD) instructions and the use of long registers to hold several short data values. The PowerPC's Altivec extension and the x86 architecture's SSE2 extension use this technique.

Many DSPs rely on SIMD extensions. The Intel Itanium is one of the fastest architectures, but a high-quality compiler is necessary to achieve peak performance. The compiler reorders instructions using predicated execution for exploiting the full range of parallelism in VLIW instructions. Itanium uses a very large rotating register file to keep all data for large software pipelined loops in registers.

DSPs typically have extensions for application-specific requirements. For example, audio and video data is often shorter than the common data size of 32 bit, and special-purpose instructions support such data. Additionally, 16- and 32-bit fixed-point arithmetic needs fewer resources than floating-point arithmetic, so DSPs are more likely to provide fixed-point than the floating-point arithmetic. Streaming access to data buffers is also very important for digital signal applications. So DSPs usually support special addressing modes with automatic updating to provide efficient access to circular buffers. Bit-reversed addressing speeds up fast Fourier transformations (FFTs), while parallel comparisons support Viterbi decoders that save the index.

State of the art

Current architectural approaches to DSPs fall into three groups: traditional architectures for DSP cores, scaleable core architectures, and architecture description languages. Each approach lacks certain characteristics for success in terms of their support for easy application-specific programming.

Traditional DSP core architectures

The Starcore SC1200 and SC14000 are the latest products based on traditional DSP core architectures. These products are also notable because they result from the cooperative work of several typically competitive companies—Motorola, Agere, and Infineone Technologies—that builds upon the Blackfin DSP, which is, in turn, the outcome of a cooperation between Analog Devices and Intel. Both Starcore concepts are RISC-based load-store architectures, claiming to be efficiently programmable in high-level languages like C or C++.

In such traditional designs, the instruction set architecture (ISA) and the microarchitecture are fixed. This prevents the application-specific modifications necessary to closing the gap between hard-wired implementations and software-based solutions.

Scalable core architectures

The best known examples of architectures with scaleable cores are from Tensilica and

ARC. However, both have their basis in traditional microcontroller architectures. Therefore, efficient implementation of traditional DSP algorithms is difficult and issues such as minimizing the worst-case execution time do not receive attention. Software support for using DSP-specific features is inadequate. The philosophy of using "just an additional multiply-accumulate unit" directs the focus toward increasing theoretical performance, instead of an analysis of the overall system performance.

Architecture description languages

The Language for Instruction Set Architecture (LISA) from Coware, mainly a development of Rheinisch-Westfälische Technische Hochschule-Aachen, is the best known architecture description language. More recent projects include ArchC in Brazil. The concept of defining your own specific core architecture to fulfill the requirements of your application code sounds unbeatable at first glance. However, automatic generation of a core microarchitecture from only a behavioral-level description results in poorly used silicon.

In the very large solution space provided by an architectural description language, very many core architectures are definable, but only a few are compatible with the development of an optimizing high-level language compiler. In addition, using an architecture description language like LISA to generate efficient solutions requires a deep knowledge of the processor architecture.

Design space exploration is necessary when supporting scalability or configuring a core architecture; it has to have its basis in a high-level language compiler. Unfortunately, automatic generation of high-level language compilers is still infeasible. Even with approaches like Compiler Development System, a software product from Associated Computer Experts, the quality of the code that an automatically generated compiler produces is poor. The poor quality of the generated code can mislead designers into making poor decisions regarding architectural modifications.

Our approach

In summary, the problem is that understanding how the application requirements affect the core architecture requires an efficient high-level language compiler that produces high-quality code. Such a compiler is impractical to generate for each core architecture; automatically generating such a compiler is also not possible using current methods.

Our xDSPcore approach attempts to solve these problems by providing a general-purpose DSP core architecture that has its basis in a RISC load-store model and enables efficient execution of traditional DSP algorithms. We include system aspects like the possibility of minimizing the worst-case execution time. To close the gap between hard-wired ASIC implementations and software-based solutions, the core concept enables scaling of the main architectural features, while the microarchitectural model remains unchanged. We defined the microarchitecture to satisfy the requirements for developing an optimizing C compiler, which would support design space exploration of specific application code. To keep the validation and verification effort low, we use a unique configuration file based on the Extensible Markup Language (XML). This lets us scale core features while ensuring the effects of all changes propagate to the hardware, tools, and documentation.

xDSPcore architecture

Requirements

Four assumptions influence the design requirements for the xDSPcore architecture:

- Application developers will program the processor only in a high-level programming language like C, C++, or Java.
- Implementations can have a different number of registers or pipeline stages.
- Mobile, embedded DSP applications will use the architecture. Therefore, the processor has to be area efficient—that is, have a minimal area for program memory, the DSP core, and data memory. It also must be energy efficient and have available the computational resources for executing DSP applications.
- Additionally, control-intensive applications must have efficient support.

These requirements led us to choose the following architectural features:

- dual Harvard load-store architecture,
- variable-length VLIW architecture,

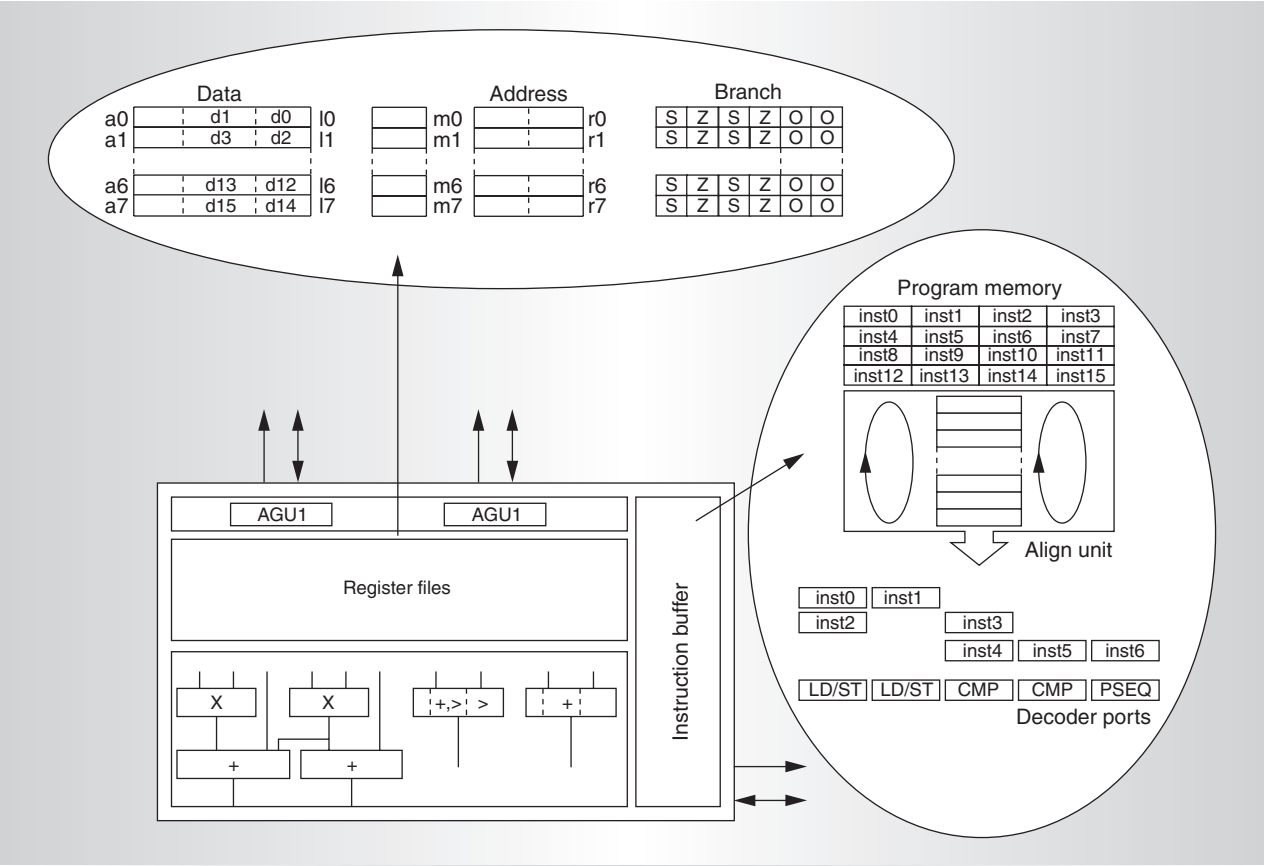

Figure 1. Architecture overview.

- predicated execution,
- orthogonal instruction set,
- separate register files,
- register file access in the corresponding pipeline stage,
- SIMD instructions, and
- DSP data types and extensions.

Architecture description

We based the xDSPcore on a modified, dual Harvard load-store architecture. Figure 1 gives a brief top-level overview of the xDSPcore architecture. VLIW is the programming model. Static scheduling allows shifting dependency resolution into the C compiler and therefore reduces the complexity of the core architecture. However VLIW produces poor code density, a problem that leads us to introduce xLIW.[2] We base xLIW on a variable-length execution set (VLES), which enables decoupling of fetch and execution bundles. Compared to VLES, xLIW permits reducing the size of the program memory port (and therefore reduces the wiring effort) without limiting the core architecture's peak performance. To speed up the execution of the inner loops of DSP algorithms, we introduced an instruction buffer that overcomes the possible bandwidth mismatch resulting from the reduced size of the program memory port.[3] A typical program memory port size would be four instruction words, whereas an xLIW instruction can use up to 10 instruction words. The buffer also reduces transition activity at the program memory port during execution of hardware and software loops. If the loop body is fetched, no further program memory access is necessary, reducing dynamic power dissipation.

Two independent data buses connect the data memories with xDSPcore. Performance reasons dictate the use of small physical memory blocks, and interleaved addressing reduces the likelihood of hazards where two address pointers simultaneously access the same physical memory block. If hazards do take place,

xDSPcore identifies this problem at run-time and serializes the two data memory accesses. The hardware hazard detection gives the compiler the opportunity to place two memory access operations, which usually access different memory blocks but in rare cases access the same block, in the same instruction. The widths of the data buses are scaleable for adapting memory bandwidth to application-specific requirements.

For load-store architectures, the register file is a central part of the core architecture. Separate instructions help move data between a register file and data memory; all arithmetic instructions use operands stored in the register file. The register file of xDSPcore splits into three parts: data register file; address register file, including modifier registers; and a separate branch file (which is not fully visible to the instruction set architecture).

The data register file supports three types of register sizes. Data registers are 16 bits wide, whereas long registers are 32 bits wide and accumulator registers are either 40 or 64 bits wide, depending on the core variant. Two consecutive data registers are accessible as one long register. The long register and the additional guard bits for an increased range of fixed-point values form the accumulator register. We use 64-bit-wide accumulator registers for xDSPcore variants that execute the four operations of a SIMD instruction in parallel.

The structure of the register file is orthogonal. There are no restrictions on the usage of registers for special instructions. The same is true for the address register. Each 32-bit-wide address register has a modifier register for modulo addressing and bit-reversal address mode (necessary for efficient FFT implementation).

The third part of the register file, the branch file, contains status information about the register contents (like a sign and zero flag for each register of the data register file). It also contains dynamic information updated by the data flow (like overflow flags or flags indicating loop status). The core static information is updated each time the register content changes. Therefore during interrupt handling or task switching, it is unnecessary to update static information. However the user must handle and sometimes modify dynamic information. We chose a separate branch file to relax the number of read or write ports associated with the register files, a resource already stressed by the orthogonality requirements.

Status information in the branch file is for conditional branch instructions and controlling predicated execution. The xDSPcore supports a rich set of predicated or conditional execution instructions, thus reducing the frequency of branch instructions and consequently avoiding branch delays.[4] When executing control code, predicated execution can significantly reduce the number of unused branch delay slots.

The xDSPcore features a RISC-like pipeline with three phases: instruction fetch, decode, and execute. Each phase can be split into several clock cycles, which results in higher clock frequencies. However, splitting the instruction fetch into several clock cycles increases the number of branch delays.

Spending several clock cycles on the execution phase increases load- and define-use dependencies. Compensation methods for the arising drawbacks (bypasses and branch prediction) are available but increase core complexity and silicon area.

Instructions access register operands in the pipeline stages where the core uses or computes them. The core can save register values internal to a pipeline stage during an interrupt. This reduces register pressure and makes software pipelining possible without the need for rotating register files and modulo variable renaming, as we explain later.

Code density measures how efficiently an algorithm can use the chosen core architecture. To increase code density and exploit available parallelism, the xDSPcore supports SIMD instructions. Executing two multiply-accumulate instructions in parallel (adding the results of the two multiplications into one accumulator) can speed up filter operations. In many cases, this eliminates the need for accumulator splitting and reduces the pressure on registers. The arithmetic logic units' (ALUs') data paths can execute two instructions in parallel on the same data path (like two 16-bit additions taking place on the same adder structure). SIMD on the ALU paths both increases code density and reduces the number of clock cycles necessary to execute an algorithm. To reduce the number of move operations between registers in the register file, the instruction set supports

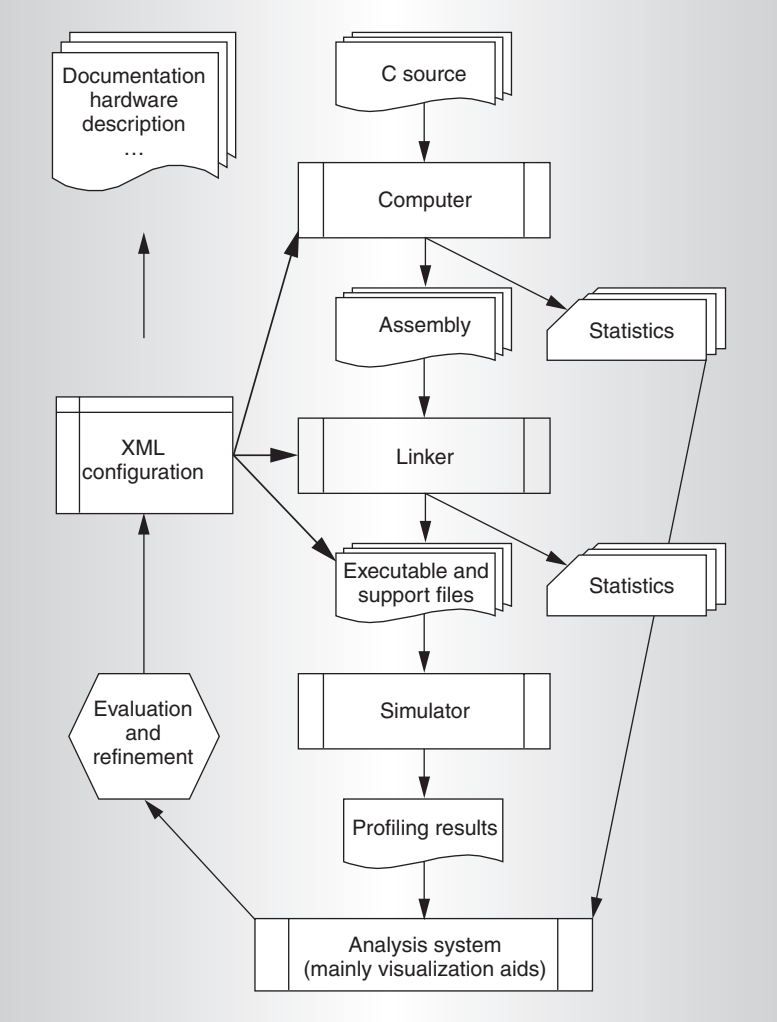

Figure 2. DSPxPlore overview.

SIMD cross operation. The high and low words of long registers are usable for SIMD operations, combining one high operand with one low operand in a crosswise manner, and vice versa.

For higher code density, it is possible to adapt the instruction set architecture to application-specific requirements. We can optimize the mapping of the chosen instruction set to binary coding in order to reduce switching activity at the program memory port. The size of the native instruction word is scaleable. The native instruction word of xDSPcore contains three bits to indicate operation class and containing alignment information. The remaining bits are for instruction coding. An additional parallel word is available for coding long immediate values or offsets.

Exploration methodology

The xDSPcore allows parameterization of core features to meet application-specific requirements more efficiently (in terms of area consumption and power dissipation). Along with the architectural design of xDSPcore, we also developed a methodology for design space exploration called DSPxPlore.[5] Figure 2 gives a top level view of the relevant tools and the flow of information between them.

A central part of DSPxPlore is an XML-based configuration file. This file defines an architectural instantiation within the design space. An XML schema checks validity of the definition during the edit of the XML file. The various tools in the tool chain (compiler, assembler, linker, and interpreting and compiling simulator) all obtain their configuration parameters from this file, guaranteeing consistency. The different tools automatically generate statistical and profiling information; an analysis system subsequently processes the information to render it more intelligible to the system designer (allowing the visualization of code size, the display of resource utilization and histograms of, for example, immediate values and the identification of hotspots).

At this time, the system designer reasons about possible improvements and makes the necessary architectural changes. However, because of the defined work flow and the available XML tool sets, an automatic reasoner is possible at this stage of the exploration. Whenever the system designer is satisfied, he can fix the configuration and generate the main parts of the hardware description as well as system documentation.

Our experiences with an implemented prototype show that when the source code is stable and we've properly configured the tool chain workflow, the exploration process can iterate very quickly. We spend the most time in the reasoning process, because setting up a new architectural instance is straightforward when using a validating XML editor. Nevertheless, DSPxPlore is still just an expert system; the system designer still needs a profound knowledge of the architecture.

Compiler optimizations

Part of the xDSPcore project is the implementation of hardware-dependent compiler passes and optimizations. For the hardware

independent parts, we use the *open compiler environment* from Atair, which provides a C compiler front end. It builds an intermediate representation based on an *abstract syntax tree*, which serves as the basis for our optimizations.

Our optimizations (which include instruction selection, instruction scheduling, and register allocation) mainly target code quality, a crucial requirement for the compiler. If the generated code is not good enough, embedded-system designers would neither accept nor use the compiler, and revert to manual assembly coding. Our goal is to generate code that has about a 5 to 10 percent overhead (in terms of size and cycles) as compared to handwritten and manually optimized assembly code.

To evaluate the performance of our compilation techniques, we assembled a test suite of DSP benchmarks and typical DSP applications as C programs for compilation. Our benchmarks include typical DSP routines used in the real world such as dot product, matrix operations, and filtering. C implementations of typical DSP applications in our experiments include those for a Global System for Mobile communication codec, fast Fourier transform, and Blowfish.

Register allocation

Register allocation is an important optimization during code generation by an optimizing compiler. The state-of-the-art algorithms for solving this problem have their basis in graph coloring. This approach best suits highly regular, general-purpose RISC architectures. In embedded systems, however, we often deal with irregular architectures. One common irregularity occurs with shared registers. This means that a register is addressable either as one 32-bit register or as two half registers of 16 bits each. Without proper modeling, traditional register allocators introduce unnecessary spill code for shared registers. Therefore, we need to extend the model and the available algorithms to consider these architectural constraints. Proposals range from a multigraph approach to a weighted graph, and to a more generalized approach.[6]

Another approach for solving the coloring problem and minimizing spill code uses a mathematical model called partitioned boolean quadratic programming (PBQP).[7] Cost vectors and cost functions encode constraints (architectural and interference) on symbolic registers, and a solver delivers an optimal register assignment with optimal spilling decisions.

Table 1. Comparison of spill costs for graph coloring and PBQP dynamic spill instructions.

	No. of dynamic spill instructions executed	
No. of registers	**Graph coloring**	**PBQP**
4	2,550	1,760
8	1,210	1,040
16	94	57

In a previous work, we compared the results of an extended graph-coloring register allocator against the results of a PBQP-based register allocator for different register file sizes. The results indicate that PBQP clearly outperforms graph coloring; the latter is acceptable only for large register files because few spills occur. Table 1 shows the accumulated spilling costs (number of executed spill instructions) for more than 200 different C functions from our benchmark suite.

The main problem with the PBQP approach is its excessive runtime for some irreducible interference graphs, sometimes even resulting in degeneration to an nonpolynomial-complete enumeration. Therefore we perform a prepass recognition of those graphs, based on heuristics in our previous work.[8] As a result, the compiler usually performs PBQP register allocation, in which case the register assignments are optimal, but falls back to graph coloring if the allocator identifies an irreducible problem. The heuristics and experiments are still at an early stage of development.

A key point of the xDSPcore architecture is its scalability. This affects the size and the layout of the register file, which has a major impact on the implementation of the register allocators. Maintaining several different source code versions of the register allocators for different architecture variants is infeasible. Therefore, we decouple the algorithms from architectural issues. Architectural information for graph coloring is encapsulated into a generic method interface, whereas the allocator dynamically creates the cost models for PBQP. Both of these approaches use the same information from the XML configuration file. The runtime overhead of the configuration step is negligible compared to that of the total runtime.

```
a) C code
for (i=0;i<32;i++) sum += a[i]*b[i];

b) base assembly code (executed in 97 cycles)
clr A0 || bkrep 31,loopend
    ld (R0)+,D2 || ld (R1)+,D3
    nop
    fmac D2,D3,A0
loopend:

c) SIMD optimized code (executed in 49 cycles)
clr A0 || bkrep 15,loopend
    ld (R0)+,L1 || ld (R1)+,L2
    nop
    fdmac L1,L2,A0
loopend:

d) software pipelining code (executed in 18 cycles)
ld (R0)+,L1 || ld (R1)+,L2 || clr A0
ld (R0)+,L1 || ld (R1)+,L2 || rep 13
    ld (R0)+,L1 || ld (R1)+,L2 || fdmac L1,L2,A0
fdmac L1,L2,A0
fdmac L1,L2,A0
```

Figure 3. Loop that computes the dot product for two arrays with 32 short values to a long accumulator.

SIMD generation

Embedded processors for media applications usually have SIMD instructions, which improve program execution performance by performing the same type of computation on different data items in parallel.

In load-store architectures, load and store are the only instructions that interact with the main memory, while arithmetic and logical instructions can only access registers and constants. This leaves two principal directions for optimizing SIMD code: first, we can use arithmetic and logical SIMD instructions to parallelize independent computations, and second, we can use load and store operations to efficiently access data vectors in memory.

To create SIMD instructions, our compiler framework has a special optimization based on algorithms in our earlier work.[9] The algorithm operates on individual C functions, using the machine-independent intermediate representation. It detects innermost loops, in which SIMD instructions will be generated, and unrolls these loops K times, where K depends on the data types. For the unrolled loop, we build an acyclic data dependence graph. The nodes of the graph are s nodes, *statements* executed in each loop iteration, and b nodes, *basic blocks* that are not executable in each loop iteration. The algorithm then collects the def or use information for every node; it adds an edge between two nodes if there is a def-use, use-def, or def-def dependency between them. The algorithm schedules the nodes of the graph in a top-down manner. It also combines structurally equivalent s nodes into one SIMD instruction if there are no true or output dependencies between them.

Figure 3c shows the use of SIMD instructions to compute the dot product for two short arrays into a long accumulator. (To understand Figure 3, it helps to know that the bkrep instruction stands for a zero-overhead hardware loop, and fmac and fdmac are multiply-accumulate and double multiply-accumulate instructions.) The SIMD optimized code, in Figure 3c, is two times faster than the compilation without the SIMD optimization in Figure 3b, because of the long load and the use of the double multiply-accumulate of the xDSPcore architecture.

If number of iterations N in the original loop is not a multiple of K, the unrolled loop is preceded or followed (depending on the alignment information about pointers within the original loop) by a copy of the original loop executed $N \bmod K$ times. If a statement in the original code contains a scalar expression, the SIMD optimization computes a suitable K-dimensional nonscalar expression to replace it. The elements of an expanded scalar equal the value of the scalar from which the expression was expanded.

Another common operation is computing the sum of an array's elements. This operation is not directly vectorizable, and a K element auxiliary array is necessary. Its elements accumulate partial sums in parallel, from which another addition can compute the final sum.

To use a block of data for SIMD load and store instructions, it must be verifiable either statically or dynamically if the block is aligned, that is, if it starts at an address that is the zero modulo of the block size. For example, to use SIMD load for the code in Figure 3a, pointers a and b require alignment. (We must assume this requirement for the code in Figure 3c.) Our alignment verification mechanism takes care of the alignment requirements, integrating both a static alignment analysis[9] and dynamic check. The static analysis is a context-sensitive interprocedural analysis, having intra- and interprocedural parts.

The intraprocedural alignment algorithm traverses the procedure's control flow graph and calculates for all pointer definitions a set of the possible values of the least-significant bits (called *may* analysis). If the set is the singleton set then this produces exact alignment information. The interprocedural analysis follows the sequence of functions in the program, for each function storing call alignment information for global pointers, actual pointer arguments, and function return pointers. We use intraprocedural analysis to compute the intraprocedural information sets for the function until we reach a function call. We then use the information sets to update the corresponding interprocedural sets. Our algorithm handles addition, subtraction, and multiplication operations (like pointer arithmetic) common in address calculation. Experimental results show that this alignment algorithm can use static analysis to determine about 50 percent of the pointers.

To use the alignment information for SIMD optimization within a function body, the algorithm merges all possible calling contexts for that function. If the context merging leads to an excessive loss of precision in the alignment information for a function, it is possible to introduce copies of the function with different calling contexts. In addition, the alignment analysis enables further code optimizing transformations, which adjust unaligned buffers to fit the alignment requirements.

To evaluate the SIMD performance, we use the suite of benchmarks mentioned earlier. Four different compilations in Table 2 present averages of the evaluation results. In this table, the result labeled "base" does not use any SIMD instructions; "short load" uses only arithmetic and logical SIMD instructions, but not load and store; "dynamic" generates any possible SIMD instructions using dynamic checks; and "static" generates any possible SIMD instructions using static alignment information. We normalize the figures with respect to the results of the non-SIMD compilation (base).

The experiments indicate that our method for SIMD instruction generation can significantly improve code quality, and that increasing the degree of SIMD parallelism can reduce the number of cycles. Removing the dynamic checks can reduce the code size by up to a factor of 4.5.

Table 2. Evaluation of the SIMD performance.

Compilation	Cycles (percentage of base)	Code size (percentage of base)
Base	100	100
Short load	83	132
Dynamic	61	324
Static	58	120

Software pipelining

Software pipelining is an important optimization for pipelined VLIW architectures. The xDSPcore performs a register operand read at the start of the pipeline stage and a register operand write at the end of the pipeline stage, whenever this operand is necessary. A multiply-accumulate instruction reads its multiply operands at the start of the multiply stage, and reads the accumulator at the start and writes to it at the end of the accumulate stage. We exploit this feature to generate software-pipelined loops with minimum register requirements. Without this feature, it would be necessary to make copies of the register and perform modulo variable renaming, thus increasing the code size and register requirements.

With software pipelining, the loop body in Figure 3d can execute in one cycle resulting in 18 cycles overall. Two cycles are necessary to set up the loop and load the operands for the multiply (prolog). There is a load delay of one cycle. The values that are loaded into registers L1 and L2 in the first line of the prolog are available in the first iteration of the loop body. The second iteration of the loop body uses the values that are loaded in the second line of the prolog. The multiplication in iteration $n + 2$ uses the values that were loaded in the loop in iteration n. Two cycles are necessary to execute the remaining multiplications after the loop has exited (epilog). The value in accumulator A0 is usable with a one-cycle delay after the last fmac operation in the epilog.

Our software pipelining algorithm is based on iterative modulo scheduling,[10] which estimates the minimum initiation interval (the number of cycles necessary to execute the loop kernel). The instruction scheduler tries to find a valid schedule for this initiation interval. If it does not, it increments the initiation interval until it finds a solution or the number of cycles exceeds the cycle count for the original

Table 3. Reduction of switching on the program memory port.

$VLIW_{max}$	Switching reduction (percentage)
3	-7.8
4	-10.0
5	-10.4
6	-11.1

loop. As we mentioned, the xDSPcore features help avoid the modulo variable renaming.

Power saving

Energy consumption and power dissipation are also important issues in embedded-systems design. Attacking these factors from a hardware-only perspective is insufficient because the software that executes on the system has a major impact. There is significant potential for reducing energy consumption with software methods. Compiler optimizations can help reduce dynamic power dissipation.

Previous work has shown that dynamic switching activity is the main cause of power dissipation in current microprocessors.[11] Additionally, memories and buses consume a bigger fraction of the energy needs than the functional units inside the data path. Therefore, we can expect a considerable reduction in power dissipation from reducing memory accesses and switching on buses.

Traditional compiler optimizations focus mainly on minimizing data memory accesses. The Harvard architecture of xDSPcore allows the separate optimization of data and program memory accesses. To optimize program memory accesses, the scalable instruction buffer and the align unit are the important architectural features that span the optimization space.

The instruction buffer is a cache structure that facilitates the efficient execution of inner loops. The core fetches the instructions of the loop body only once from program memory. Afterwards, fetching stalls until the loop finishes. Thus the size of the instruction buffer is a parameter that affects loop optimizations. In the reverse direction, the code size of inner loops affects the choice of instruction buffer size. These two parameters form a feedback loop for architectural exploration that requires support by the tool chain. Our latest experiments show that embedded applications typically spend up to 90 percent of their cycles in inner loops that fit into buffers of 16 to 32 entries (quite a reasonable size).

The align unit is part of the VLIW code compression method used in the xDSPcore architecture. It decouples the program counter and fetch counter, reassembles the individual execution bundles, and performs instruction predecoding. This allows the arbitrary ordering of the instructions within a VLIW.

We implemented a compiler optimization that finds the best global ordering (for one function) of instructions, so that switching on the program memory bus is minimal. The algorithm runs in two phases; the first one searches for local optimization candidates (per basic block), and the second chooses those local candidates that yield the global minimum. The advantage of our approach over others is that unaligned bundles, as well as operand swapping, can occur during optimization. Our solution is therefore a generalization of the previous work mentioned. Table 3 shows the mean switching reduction for over 200 different C functions from our benchmark suite. The first column gives the maximum VLIW width, in number of functional units. We offer an exhaustive description and evaluation in another work.[12]

Instruction encoding

Code size is important in embedded systems because of its effect on chip area and therefore on chip costs. In addition to the quality of the compiled code (the mapping of the program to a specific instruction set), the binary coding of the instruction set also provides crucial opportunities for optimization.

There are several factors that influence the instruction encoding, for example, number of different instructions, number of registers, width of offset fields, and type of code compression. The xDSPcore features an instruction encoding based on two different instruction word lengths. A native word encodes the most frequently used operations and offsets. A long word that has twice the size of a native word is for the remaining instructions. In addition, there are two special cases for instruction encoding, a 16- and a 20-bit native word. Which encoding is best depends on the type of application. Table 4 shows a

total code size comparison for an enhanced full rate coder (EFR).

Besides total code size, the binary encoding also affects switching activity on the program memory bus. Smaller native words reduce the width of the bus and thus the wiring effort. Bit transitions are reducible if there is a minimum Hamming distance between instructions in pairs that are frequently fetched in succession.

Our digital signal processing architecture's hardware design is based on up-to-date DSP features. The joint development of the C compiler and the architecture helps efficiently exploit the features in a high-level programming language. New register allocation techniques use multiple banked register files. Register accesses at the corresponding pipeline stages makes software pipelining simpler and more efficient. Because the hardware handles conflicting parallel memory accesses, the compiler can apply optimistic placement of parallel memory access instructions. A three-input adder helps avoid accumulator splitting. Architecture exploration supports customizing the architecture to suit the application.

In another project, we are working on an automatic encoding generator. An abstract description of the instructions and their necessary coding fields as well as statistical code information (ranges of offsets and instruction histograms) serve as input to the generator. Using this input, the generator creates a concrete instance of a binary coding that the simulator can evaluate. MICRO

Table 4. Comparison of code size by using different encodings: encoding, native, long, and size (in bits).

Encoding type	Native (no. of words)	Long (no. of words)	Size (no. of bits)
16 bit, 32 bit	3,780	1,892	121,024
20 bit, 40 bit	4,079	1,592	145,260

Acknowledgments

This work is supported by Infineon Technologies Austria, the European Commission under the project SoCMobinet (IST-2000-30094) and the Christian Doppler Gesellschaft.

References

1. R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems*, Kluwer Academic Publishers, 2001.
2. C. Panis, et al., "xLIW—A Scalable Long Instruction Word," *Proc. Int'l Symp. Circuits and Systems* (ISCAS 03), IEEE CS Press, 2003, pp. 69-72.
3. C. Panis, et al., "A Scalable Instruction Buffer for a Configurable DSP Core, *Proc. 29th Int'l European Solid State Circuits Conf.* (ESSCIRC 03), IEEE Press, 2003, pp. 49-52.
4. C. Panis, et al., "FSEL—Selective Predicated Execution for a Configurable DSP Core," *Proc. Symp. VLSI Emerging Trends in VLSI Systems Design* (ISVLSI 04), IEEE CS Press, 2004, pp. 317-320.
5. C. Panis, et al., "DSPxPlore: Design Space Exploration Methodology for an Embedded DSP Core," *Proc. Symp. Applied Computing* (SAC 04), ACM Press, 2004, pp. 876-883.
6. J. Runeson and S.O. Nystrom, "Retargetable Graph-Coloring Register Allocation for Irregular Architectures," *Proc. 7th Int'l Workshop Software and Compilers for Embedded Systems* (SCOPES 03), http://user.it.uu.se/~svenolof/wpo/AllocSCOPES2003.20030626b.pdf, Sept. 2003.
7. B. Scholz and E. Eckstein, "Register Allocation for Irregular Architectures," *Proc. Joint-Conf. Languages, Compilers, and Tools for Embedded Systems: Software and Compilers for Embedded Systems* (LCTES/SCOPES 02), ACM Press, 2002, pp. 139-148.
8. U. Hirnschrott, A. Krall, and B. Scholz, "Graph-Coloring Versus Optimal Register Allocation for Optimizing Compilers, *Proc. Joint Modular Language Conf.* (JMLC 03), Lecture Notes in Computer Science, Springer Press, 2003, pp. 202-213.
9. Pryanishnikov, et al., "Pointer Alignment Analysis for Processors with SIMD Instruction," *Proc. 5th Workshop on Media and Streaming Processors* (MICRO 36), Christian Doppler Laboratory Publications, Dec. 2003, http://www.complang.tuwien.ac.at/cd/publications.html.
10. B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann. Int'l Symp. Microarchitecture* (MICRO 27), ACM Press, 1994, pp. 63-74.
11. A.P. Chandrakasan, et al., "Low-Power CMOS Digital Design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, Apr. 1992, pp. 473-484.

12. U. Hirnschrott and A. Krall, "VLIW Operation Refinement for Reducing Energy Consumption," *Proc. Int'l Symp. System-On-Chip* (SOC 03), 2003, http://www.complang.tuwien.ac.at/cd/soc2003.pdf.

Andreas Krall is a professor of computer science at the Vienna University of Technology, Austria. His research interests include compilers, programming languages and computer architectures. Krall has a PhD in computer science from the University of Technolgy in Vienna. He is a member of the IEEE and ACM.

Ulrich Hirnschrott is a computer scientist and a PhD student at the Vienna University of Technology, Austria. His research interests include power-saving compilation techniques and parameterizable compiler design. Hirnschrott has an MS in computer science from Vienna University of Technology.

Christian Panis is pursuing a PhD in "Scalable DSP Core Architectures Addressing Compiler Requirements" at Tampere University of Technology, Finland. His research interests include digital signal processor architectures. Panis has an MS in industrial electronics and control theory from Vienna University of Technology.

Ivan Pryanishnikov is a computer scientist and a PhD student at the Vienna University of Technology, Austria. His research interests include computer architectures, compilation, and parallel computations. Pryanishnikov has an MS in computer science from the Technical University of Tashkent.

Direct questions and comments about this article to Andreas Krall, TU Wien, E185-1, Argentinierstr. 8, A-1040 Wien, Austria; andi@complang.tuwien.ac.at.