J Sign Process Syst
DOI 10.1007/s11265-008-0172-z

FlexCore: Utilizing Exposed Datapath Control for Efficient Computing

Martin Thuresson · Magnus Själander · Magnus Björk · Lars Svensson · Per Larsson–Edefors · Per Stenstrom

Received: 12 October 2007 / Revised: 4 January 2008 / Accepted: 4 March 2008

Abstract We introduce FlexCore, the first exemplar of an architecture based on the FlexSoC framework. Comprising the same datapath units found in a conventional five-stage pipeline, the FlexCore has an exposed datapath control and a flexible interconnect to allow the datapath to be dynamically reconfigured as a consequence of code generation. Additionally, the FlexCore allows specialized datapath units to be inserted and utilized within the same architecture and compilation framework. This study shows that, in comparison to a conventional five-stage general-purpose processor, the FlexCore is up to 40% more efficient in terms of cycle count on a set of benchmarks from the embedded application domain. We show that both the fine-grained control and the flexible interconnect contribute to the speedup. Furthermore, according to our VLSI implementation study, the FlexCore architecture offers both time and energy savings. The exposed FlexCore datapath requires a wide control word. The conducted evaluation confirms that this increases the instruction bandwidth and memory footprint. This calls for efficient instruction decoding as proposed in the FlexSoC framework.

Keywords Flexible · Interconnect · Computer architecture · Reconfigurable

M. Thuresson (✉) · M. Själander · M. Björk · L. Svensson · P. Larsson-Edefors · P. Stenstrom
Chalmers University of Technology, Gothenburg, Sweden
e-mail: martin@ce.chalmers.se

1 Introduction

Cost- and performance-sensitive application areas, such as cellular phones and other battery-powered multimedia devices, are not well served by present-day general-purpose computing platforms. To meet user expectations of features and battery capacity, designers instead resort to highly heterogeneous systems where a collection of specialized hardware accelerators (built for encryption, image and video coding, audio playback, etc.) are controlled by an embedded microprocessor, such as an ARM core. For cost reasons, several accelerators will typically be collocated with the microprocessor on a single system-on-chip (SoC).

The present practice has drawbacks. Tasks outside the set originally intended may not benefit from the computing capacity available: computing resources hidden inside an accelerator may be difficult or impossible to use in ways other than those considered by the accelerator designer. Even when possible, the software constructs necessary to access a "hidden" hardware block bear little resemblance to ordinary code.

For ease of software development and maintainability, a uniform hardware/software interface, similar to those offered by general-purpose processors (GPPs), would be highly desirable; but present-day GPPs cannot compete with heterogeneous SoCs in terms of performance at a given power level. Merging the accelerator datapath elements into the GPP infrastructure would be possible in principle, but a very wide instruction word would be required to make fine-grained control possible. A wide instruction word potentially increases the memory footprint of an application. For typical embedded systems, memory is expensive in terms of money and power. Because of the tight power and cost constraints together with the high volumes used, it is important to keep the memory usage and I/O activity down.

In our approach, called FlexSoC [1], we address these problems by moving away from the conventional GPP instruction set architecture (ISA) and use a wide control word. This allows us to control the units in the datapath at a much more fine-grained level. Other important differences between our approach and the standard GPP-like ISA is that each control word, or native-ISA (N-ISA), controls the datapath in the current cycle only and that forwarding needs to be done statically. Previous work on exposed control has shown significant performance improvements [2] when used within a hardware/software co-design system. In contrast, the FlexSoC approach includes a (extendable) pipeline together with a flexible instruction decoder.

To better understand the requirements of the instruction decoder, our study investigates how the instruction bandwidth as well as the static code size is affected by the exposed control. In particular, we note that even though the wide instructions allow for more efficient control, the number of instructions needed is still large, thus increasing the static code size considerably compared to a conventional GPP-version. In fact, all benchmarks are larger and require three times as much bandwidth as a GPP-version.

Consequently, in the FlexSoC framework we propose to use a programmable instruction decoder. This decoder allows the compiler to use a compressed Application-Specific ISA (AS-ISA) for each application, or set of applications, to be executed. Applications are stored as AS-ISA instructions that are expanded on-the-fly to N-ISA instructions when fetched from the program memory. Figure 1 illustrates this scheme. As will be shown, the findings in this article clearly motivate such a scheme.

In this paper, we make the following main contributions. (1) We introduce the FlexCore datapath, a datapath with exposed control based on the FlexSoC framework. (2) We evaluate the microarchitecture in detail and show that it enables better performance in cycle-count and execution time as well as energy efficiency. A detailed study of the interconnect shows the usage patterns of our applications and layout implementations of the schemes allow us to do qualitative comparisons. (3) In this framework, we also show that the exposed pipeline comes at a cost of both static code-size and instruction bandwidth.

This paper is organized as follows: The FlexCore architecture is introduced in Sections 2 and 3, followed by compilation techniques used, Section 4. Section 5 presents how the FlexCore architecture was evaluated, and the results are presented in Section 6. Related work is discussed in Section 7; the paper is concluded in Section 8.

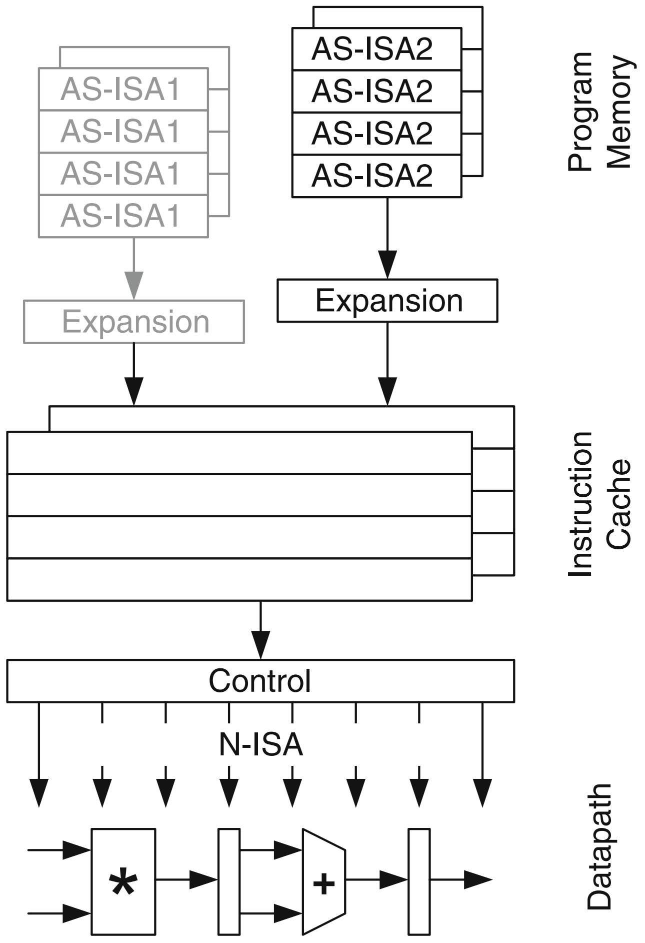


Figure 1 AS-ISA instruction decoding into N-ISA instructions.

2 The Baseline FlexCore Architecture

Application studies in our field of interest, in particular comparisons [3] of two audio compression standards (MP3 and Ogg Vorbis), have convinced us that full GPP functionality is necessary for flexibility. To offer the full programmability of a GPP, we have therefore decided to include all the datapath units necessary to emulate a full-featured processor in our baseline architecture. We have opted to use a conventional, single-issue, five-stage pipeline similar to the Hennessy–Patterson 32-bit DLX and MIPS R2000 as a template [4]. This is not a high-performance processor design; in FlexSoC, however, our ambition is to provide application performance mainly through the use of specialized accelerators and fine-grained control rather than through conventional methods. Thus, our core is designed to be flexible and gradually extensible, with accelerators according to application requirements. Moreover, recent research have shown that simple cores can be very energy efficient and successfully used in high-performance systems [5, 6].

The datapath is fully exposed and is controlled through a 91-bit wide N-ISA control word. The baseline FlexCore consists of four datapath units (see Fig. 2): register file, arithmetic logic unit (ALU), load/store unit (LS Unit), and a program counter unit (PC Unit), with all units connected to a flexible, fully connected interconnect. To allow for GPP functionality, each datapath-unit output port is connected to a data register. The data registers act as pipeline registers in the various pipeline configurations that can be assembled using the flexible interconnect. Thus, it is easy to create different pipelines by routing a result from one datapath unit to the next.

To allow instructions to be scheduled on the FlexCore in the same way as on a conventional five-stage pipeline, two data registers (RegA and RegB) have been included. The two data registers are used in the execute and load/store stage of a conventional GPP and allow data to bypass the ALU and LS unit.

The baseline FlexCore can act as a GPP since it can emulate a conventional pipeline and run instructions in the same way. The FlexCore architecture also allows for high resource utilization due to the flexibility in scheduling and the fine-grained N-ISA control word.

2.1 N-ISA: Exposed Datapath

As described in Section 1, the datapath of a FlexCore can be precisely controlled through the N-ISA control word. The N-ISA format depends heavily on the architecture and its datapath units. Figure 3 shows the N-ISA control word for the baseline FlexCore architecture. Starting from the least significant bit, the control word

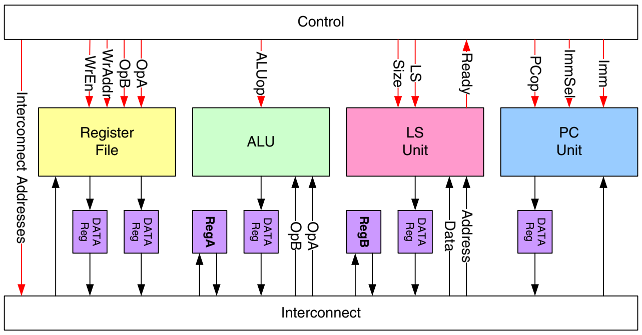


Figure 2 Illustration of a baseline FlexCore. Note that each DATA Reg also has a stall signal not shown in the figure.

Interconnect	PC	D	LS	A	Register

Figure 3 FlexCore N-ISA control word. The different fields are: *Interconnect* (24 bits), *PC* (37 bits, of which 32 bits are immediate), *D* (data buffers, 2 bits), *LS* (load/store, 5 bits), *A* (ALU, 5 bits), and *Register* (18 bits). The total length is 91 bits.

consists of bits that control the interconnect, the PC Unit (which also includes the 32-bit immediate value), the two data buffers, the Load/Store Unit, the ALU, and finally the Register File.

The expected functionality of the datapath units allows us to identify the role of most of the bits in a straightforward manner. For example, the bits controlling the register file contain the fields denoting which two registers to read and which register to write. These bits also contain a write enable signal and two stall signals for each read port data register. The PC Unit handles the immediate value, and the ImmSel signal selects if the value emitted from the PC Unit should be the current Immediate value, or the address of the next instruction (which is used in jump-and-link-like instructions).

The N-ISA includes the bits controlling the interconnect. Since each output can be associated to any input in every cycle, the number of bits, n, needed to control an N-input, M-output interconnect is $n = M \cdot \lceil \log_2(N) \rceil$.

In each cycle, an N-ISA word controls all units in the datapath as well as the interconnect. The exposed-datapath approach differs from the conventional pipelined control word, which is found in general-purpose processors and digital-signal processors, and in which one control word (corresponding to one instruction) contains information about all pipeline stages, for this *and* consecutive cycles.

As can be seen in Fig. 3, the N-ISA word consists of 91 bits. Compared to a conventional 32-bit GPP, the FlexCore requires an instruction bandwidth that is almost three times as large, in order to keep the datapath busy. The N-ISA is clearly not an efficient representation. Therefore, FlexSoC assumes a reconfigurable instruction decoder/expander in the hardware/software interface. It comes as no surprise that our results confirm that both the static code size and the instruction bandwidth must be addressed.

The goal of giving the compiler complete control of the hardware has the drawback that binary compatibility between processors with different datapath architectures is lost. With a reconfigurable instruction decoder, as proposed in the FlexSoC framework [1], it may however be possible to reuse the same AS-ISA for different hardware configurations, but that is a topic for future research and, thus, not addressed in this work.

In Section 6, we analyze performance gains from using an exposed datapath and a flexible interconnect with the same datapath units as a conventional five-stage pipeline.

3 Extensions to the Baseline FlexCore

An advantage of the FlexCore architecture is that it can be extended with application-specific accelerator units, simply by adding more ports to the flexible interconnect and extending the N-ISA control word to include control signals for the new units. Since different units are treated equally, we hope to avoid complex ad-hoc solutions usually found in irregular interconnects. For instance, for each unit added to a conventional pipeline, the forwarding network with control logic has to be modified. A conventional fixed pipeline depth also makes it cumbersome to add datapath units and utilize them efficiently: either the new unit is put in the execute stage and can thereby only be used if the ALU is not used; or a new pipeline stage is added, which changes the architecture considerably; or the unit can be added as a co-processor, which causes communication overheads.

A fully connected crossbar guarantees that the interconnect will not restrict the scheduling of operations on the datapath units. This motivates its use in the explorative phase of the design. As seen in Section 6, the full connectivity may not be needed for a given application domain; this provides an opportunity to reduce the area and power requirements, once a suitable collection of datapath units has been determined.

3.1 Multiplier Extension

For this study, the baseline FlexCore has been extended with a multiplier in order to be able to efficiently execute embedded application benchmarks, such as the fast Fourier transform (FFT). The 32-bit multiplier, which is pipelined into two stages to balance its critical path to the other datapath units, is connected to the interconnect that has been extended with two input and two output ports. The two output ports deliver operands to the multiplier, while the result is divided into two 32-bit values. Each value is connected to an input port of the interconnect. One of the ports carries the 32 least significant bits of the result, while the other port carries the 32 most significant bits of the result, as shown in Fig. 4.

An N-ISA instruction for the extended FlexCore consists of 108 bits. The multiplier has no control signals, since it is only capable of executing one operation.

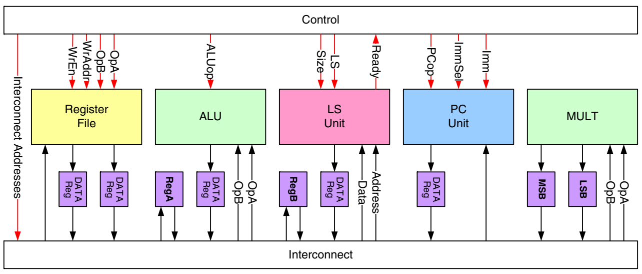


Figure 4 Illustration of a baseline FlexCore extended with a multiplier. Note that each DATA Reg has a stall signal, while RegA, RegB, LSB, and MSB have an enable signal which is not shown.

The extension to the N-ISA is due to extra bits for addressing the added ports in the interconnect, as well as two enable signals for the LSB and MSB register that is holding the result after a multiplication.

4 Compiling for FlexCore

The flexibility of architectures based on the FlexCore concept enables numerous compilation strategies. Given the ability of a FlexCore to emulate a conventional GPP, we chose as our initial approach to translate GPP-like assembly instructions into N-ISA code. As this work will show, the FlexCore can indeed emulate a conventional five-stage pipeline in real time.

The translation of single GPP instructions to N-ISA code is straightforward. We use the same datapath structure as a five-stage pipeline, but the instruction fetch stage is implicitly handled by the FlexCore control unit. In other words, each instruction spans four cycles. The first cycle uses the immediate port and the read ports of the Register File. The second cycle uses the ALU and RegA. The third cycle uses the Load/Store Unit and RegB. Finally, the fourth cycle uses the write port of the Register File. Sequences of such instructions are merged using static optimization techniques (see Section 4.1 below) to achieve pipelining and forwarding.

Obviously, making the FlexCore operate as a GPP is not the best way to exploit this architecture. However, we chose to execute GPP programs on the FlexCore to establish a performance baseline. Even though the FlexCore interconnect allows for communication between any two units, GPP instructions use only the paths corresponding to those found in the GPP. Therefore, we aim to compile high-level code down to N-ISA along the lines of other compilation methods for general datapaths [7]. This enables the pipeline length and structure to be changed as often as needed, and allows for programs to use the datapath units in any order. Currently, profiling allows the programmer to manually schedule performance-critical regions. The manually scheduled parts are written in an RTN[1]-like format that can be interleaved with GPP instructions.

4.1 Instruction-Level Static Code Optimization

Translating GPP assembly code into N-ISA code yields a number of N-ISA instruction sequences that should be scheduled as tightly as possible, overlapping each other as allowed by resource conflicts and data dependencies. Resource conflicts are not an issue in the case of GPP assembly instructions, due to their pipelined structure. Data dependencies are more important, since consecutive operations often use the same register.

When one operation uses the contents of a register that is updated by the previous operation, several cycles can often be saved by forwarding: taking the value directly from the datapath unit that produces it, rather than waiting for it to be written to the register file first.

Pipelined processors usually do these optimizations at run-time. On the FlexCore, however, they must be done statically due to the exposed control word. The basic operation is to compose two sequences of N-ISA instructions sequentially, with as much overlap as possible. This is done by annotating each instruction with information about what resources that are used, and the status of all registers. Each register can have status *available*, *unavailable*, or *rerouted(p)*, where p is the name of an output port of a datapath unit. Normally, registers are marked as *available*, which means that their value can be read from the register file. A register is *unavailable* when a new value for the register is currently being computed and is not yet available. When the value is available but not yet written to the register file, the *rerouted(p)*-annotation tells the compiler where the value can be found. In such a case, the register read is omitted, and the value is fetched from the port p instead of the register port.

Techniques such as these are not restricted to rescheduled conventional pipelined programs, but can be used for any N-ISA code. They help determine whether any two annotated N-ISA sequences are composable with a fixed overlap. To find out the maximal possible overlap, we begin by composing them without overlap, then with one cycle overlap, thereafter with two cycles overlap, and so on until we fail. It may be possible to continue even further, but then we must perform a more careful analysis to make sure that no write order conflicts occur. However, we do not expect such aggressive optimizations to have a significant efficiency impact.

Even though the compiler at times needs to make pessimistic choices not to violate any data dependencies when scheduling, our results show that the generated code is only marginally slower on GPP-code compared to a fixed five-stage pipeline with dynamic forwarding and pipelined control.

4.2 Scheduling Opportunities

To exemplify what type of optimizations are possible with the fully connected interconnect and the wide control word, we list four general optimizations that are possible on the FlexCore architecture, but not on a GPP.

4.2.1 Dynamic Pipeline Depth

A conventional processor typically has a very rigid pipeline, generally for a five-stage pipeline they are called IF/ID/EX/MEM/WB. In the DLX case, each has a fixed one cycle latency, though more advanced pipelines might have different latency for various operations in the execution unit and have hardware support for out-of-order execution.

In FlexCore, the interconnect allows us to change the structure of the pipeline during execution of the program. Consider the case of a tight loop which performs advanced arithmetic calculations, but does not have any memory accesses. During this time, the pipeline can be modified to bypass the MEM-stage completely and thus reduce the latency for each instruction.

4.2.2 Static Forwarding

Forwarding is a key principle in pipelines to increase performance. The time for a value to go from producer to consumer needs to be minimized, and in most architectures, a complex set of forwarding links and control logic are used when forwarding opportunities are dynamically identified. With the fully connected interconnect available in the FlexCore network and the fine-grained control available, it is possible for the compiler to find these opportunities and make the forwarding of operands statically, thus, eliminating the need for forwarding control logic.

4.2.3 Static Optimization

Another important feature of the exposed control is that less redundant operations need to be performed in hardware. If two consecutive arithmetic GPP-

[1]Register transfer notation.

instructions operate on the same register value, it is clear that the second instruction will overwrite the result of the first. If forwarding is used to pass the value from the first to the second instructions the value produced by the first instruction does not need to be written to the register file. With the granularity of GPP instructions, only dynamic hardware schemes can remove such redundant writes. With FlexSoC, each write is exposed and we can statically decided whether to write the result or not.

4.2.4 Instruction Parallelism

Even though the instruction stream in RISC code is sequential, the execution of the operations does not have to be linear as long as the result is *as if* the instruction were executed sequentially. The amount of instruction level parallelism (ILP) available in the program as well as the hardware resources available both limit the amount of parallelism that can be exploited. With the finer control possible with FlexSoC, we are able to find more opportunities to perform various operations in parallel. Also, if there is a bottleneck, the flexible interconnect will make it easy to add more resources which the compiler can utilize.

4.2.5 Example Schedule

To illustrate the different scheduling optimizations that can be done for the FlexCore datapaths we will consider the three GPP assembly instructions shown in Fig. 5.

In a conventional five-stage pipeline the instructions could be scheduled as shown in Fig. 6a. The instructions are scheduled one after the other, each with a latency of four cycles. The load-word (LW) instruction is independent of the other two instructions and could have been scheduled earlier. However, the total number of cycles to execute the three instructions would not be affected.

The schedule of the three instructions in Fig. 5 for the FlexCore datapath is shown in Fig. 6b. One can directly notice that the schedule for the FlexCore datapath is only four cycles, instead of six as for the GPP schedule. This is achieved by applying the scheduling optimizations described above: (1) Each instruction has a latency of three instead of the conventional four. (2) Forwarding is not explicitly shown in the schedules but are performed statically to transfer the result of the ADD in cycle 2 to the ADD in cycle 3. (3) Register $1 is only written once, thus the write-port of the register file can be used by another instruction. (4) Since the load-word instruction has an address-offset of zero, it is unnecessary to compute a new address from which to make the load from. This together with the available write-port of the register file allows the load-word instruction to be scheduled in parallel with the first ADD instruction.

Figure 5 Example of three consecutive GPP assembly instructions.

```
ADD    $1, $3, 16
ADD    $1, $5, $1
LW     $3, 0($9)
```

	ID	EX	MEM	WB
1:	Read $3 & IM16			
2:	Read $1 & $5	ADD		
3:	Read $0 & $9	ADD	NOP	
4:		ADD	NOP	Write $1
5:			LW	Write $1
6:				Write $3

a GPP Schedule

	ID	EX	MEM	WB
1:	$3, IM16, & $9			
2:	Read $5	ADD	LW	
3:		ADD		Write $3
4:				Write $1

b FlexCore Schedule

Figure 6 Instruction scheduling on a GPP and FlexCore datapath.

5 Experimental Framework

To evaluate the performance of FlexCore, four different benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC) were selected. The four benchmarks are the Fast Fourier Transform (FFT), Autocorrelation (Autocor) and Viterbi Decoder (Viterbi) from the Telecom benchmark suite and the High Pass Grey-Scale (RGBHPG) filter from the Consumer benchmark suite. All selected benchmarks were executed to completion and the result presented excludes code belonging to the test-harness.

To distinguish between the performance gains achieved by an exposed datapath and the flexible interconnect, a FlexCore with only the interconnects present in a conventional GPP pipeline has also been simulated; it is identified as “Exposed GPP” in the tables.

We chose MIPS as the ISA for the GPP pipeline, since it is supported by mature free-ware tools and cross-compilers. Each of the benchmarks was compiled using a cross-compiler to MIPS assembly with the default optimization flags for EEMBC (-O2). The assembly code was profiled to identify computational kernels, which subsequently were manually scheduled, using RTN notations, for the Exposed GPP and the FlexCore. All the benchmarks were manually scheduled without doing any algorithmic changes or optimizations, such as loop unrolling. Each assembly instruction was translated into RTN notations and scheduled as early as possible onto the given datapath. An instruction can therefore be scheduled earlier than given by the original assembly code, if there are no data dependencies and no resource conflicts. Thus, three versions of each benchmark were generated:

- **GPP** The output of the FlexCore compiler. As described in Section 4, the compiler schedules the instructions in the same way as if they would have been executed on a conventional GPP.
- **Exposed GPP** The most frequent instructions (inner loops) have been manually scheduled using only the communication paths of a conventional GPP. For example, RTN-operations make it possible to remove unnecessary register updates.
- **FlexCore** Manual scheduling of the same instructions as for the Exposed GPP but with the full interconnect of the FlexCore. For example, data can be moved directly from the ALU to the register file, without updating the buffer registers.

The code was executed on a cycle-accurate simulator that models a FlexCore connected to an ideal memory architecture, with single cycle latency. The simulator has been verified against a VHDL implementation (see Section 5.1 below). By using CRC-checks on the output of the benchmarks, we have verified that they execute correctly.

5.1 Hardware Implementation

To evaluate the performance in terms of delay, power/energy, and area, VHDL implementations were

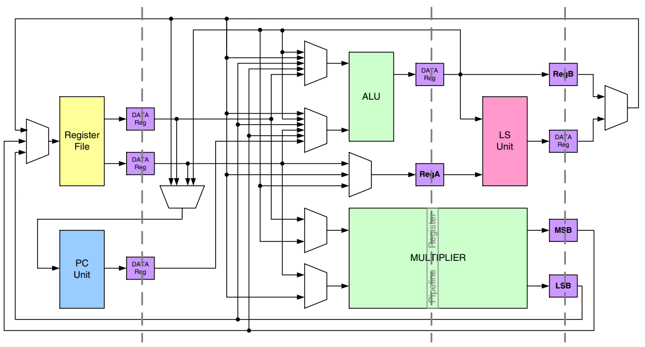


Figure 7 Schematic of the datapath for the GPP and exposed datapath. Only the communication paths for data are shown, without any control signals.

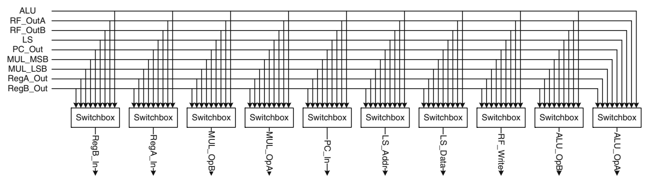


Figure 8 Illustration of the crossbar switch of the fully connected interconnect.

created for the different architectures. Note that no control logic has been implemented for the evaluated architectures in our comparison. The exposed GPP is a conventional GPP, where the control logic has been removed. Therefore, when disregarding control logic and instruction fetch, the exposed GPP and conventional GPP are equivalent. The influence of control logic and instruction fetch on the performance of the different architectures is an issue addressed within the FlexSoC project, but this is not a topic of this paper.

Figure 7 shows a detailed schematic of the communication paths that are available to the GPP and the exposed datapath. The datapath is inspired by the DLX and MIPS R2000 datapaths [4]. The multiplier is an intrinsic part of the datapath and takes its inputs from the register file. To improve performance, it is possible to forward results from the ALU and LS Unit directly. The output registers of the multiplier do not work as conventional pipeline registers, where a new value is clocked in for each new clock cycle. Instead these registers hold the result from the previous multiplication until a new multiplication is performed.

The baseline FlexCore datapath using the flexible interconnect was designed such that an input port to a datapath unit can be connected to any output port of any datapath unit. This creates a fully connected crossbar switch as interconnect.

Figure 8 illustrates how the crossbar switch is constructed from a number of switchboxes. Each input to a datapath unit (in Fig. 7) as well as to the two registers RegA and RegB is connected to its own switchbox. Each switchbox is in turn connected to all the outputs of the datapath units. The switchbox functions as a multiplexer, according to Fig. 9.

The VHDL descriptions were synthesized, placed, and routed (Cadence First Encounter [8]) using a

Figure 9 Illustration of a switchbox.

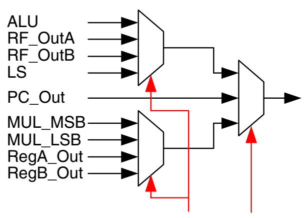


Table 1 Application cycle count.

Datapath	Autocor	Autocor (Loop)	FFT	FFT (Loop)	RGBHPG	RGBHPG (Loop)	Viterbi	Viterbi (Loop)
GPP	1.5k (100%)	1.3k (100%)	58k (100%)	43k (100%)	3.4M (100%)	3.3M (100%)	268k (100%)	218k (100%)
Exposed GPP	1.3k (88%)	1.1k (86%)	47k (80%)	31k (71%)	3.2M (93%)	3.2M (97%)	268k (100%)	222k (102%)
FlexCore	1.0k (67%)	0.8k (62%)	37k (64%)	22k (50%)	2.8M (82%)	2.8M (86%)	243k (90%)	198k (90%)

commercially available 65 nm low power technology, with standard threshold voltage. Delay and power estimations were done on the placed and routed netlists with extracted resistance and capacitance and for the worst case corner at 125 degrees Celsius. The power estimations are given for the maximum clock frequency for each of the datapaths.

6 Results

6.1 Cycle-Count Evaluation

In the first set of experiments we have executed the benchmarks and measured the dynamic cycle count. Table 1 shows the dynamic cycle count for the benchmarks. The columns labeled Loop show the number of cycles spent in the manually scheduled inner loops. Here we see that at least 75% of the executed cycles are spent in the inner loops. The percentage in the table allows us to compare the cycle count between the three architectures.

The FlexCore architecture is between 10% and 50% better in cycle count compared to the GPP for all the applications. The Exposed GPP improves only some of the applications, with FFT being the best with 20% improvement, while Viterbi could not be improved at all.

In order to understand the difference in performance among the benchmarks, the utilization of the resources in the datapath was also monitored. As with many embedded media benchmarks we saw that the inner loops performed many arithmetic operations on register values.

Figure 10 shows the relative number of cycles the ALU was used, for the various benchmarks. For all applications, the ALU utilization in the FlexCore is higher than that in the Exposed GPP, whose ALU utilization in turn is higher than in the GPP. Comparing the results with the dynamic cycle count, we see that the benchmarks with the smallest ALU utilization gained the most by the FlexCore architecture. Autocor and FFT, which have only 60% and 70% ALU utilization,

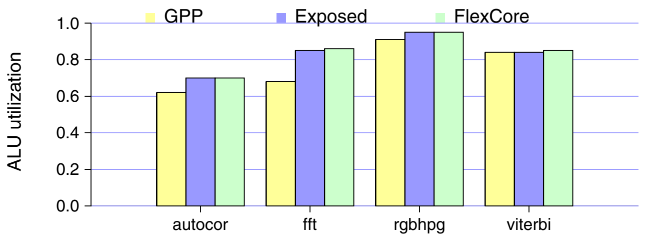
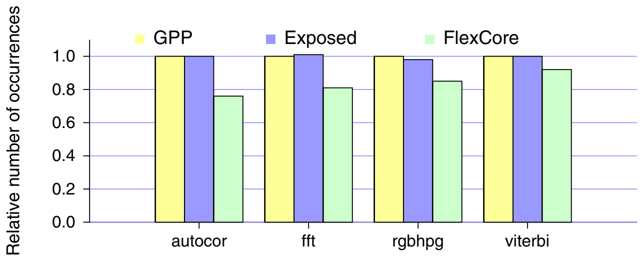
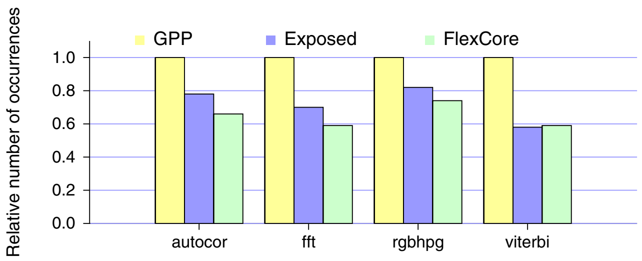


Figure 10 Percentage of cycles the ALU is used.



a ALU operations



b Register Writes

Figure 11 Number of operations for the benchmarks normalized to the GPP.

improved by 33% and 36% in cycle count with FlexCore, while RGBHPG and Viterbi with a 90% and 85% ALU utilization only gained 18% and 10%.

Furthermore, the number of ALU operations executed in total decreases in FlexCore. Figure 11a shows that between 5% and 22% of all ALU operations executed by the GPP are redundant and have been removed for FlexCore. An example of redundant ALU operations is the address calculation with a zero offset for load and store operations.

In a GPP, all calculated values are written to the register file. Nevertheless, all written values that are used in the next instruction will be routed through the forwarding network. If the value is never again read from the register file, the write is unnecessary. In the

Table 2 Delay, power, and area estimates.

Datapath	Timing (ns)	Power (mW)	Area (mm^2)
65 nm—baseline			
GPP	2.14 (100%)	7.36 (100%)	0.093 (100%)
Exposed GPP	2.14 (100%)	7.36 (100%)	0.093 (100%)
FlexCore	2.27 (106%)	8.69 (118%)	0.110 (118%)
Pruned	2.15 (100%)	7.33 (100%)	0.099 (106%)
130 nm			
GPP	2.89 (100%)	5.70 (100%)	0.260 (100%)
Exposed GPP	2.89 (100%)	5.70 (100%)	0.260 (100%)
FlexCore	3.21 (111%)	5.92 (104%)	0.293 (109%)
Pruned	3.02 (105%)	5.90 (104%)	0.267 (103%)

Table 3 Interconnect paths used by the benchmark on the GPP and on the FlexCore.

	PC In	RF Write	ALU OpA	ALU OpB	LS Address	LS Data	RegA In	RegB In	MULT OpA	MULT OpB
PC Out	1-	2-Flex	3-	4-GPP	5-	6-	7-	8-	9-	10-
RF ReadA	11-GPP	12-Flex	13-GPP	14-	15-	16-	17-	18-	19-GPP	20-
RF ReadB	21-GPP	22-	23-	24-GPP	25-Flex	26-Flex	27-GPP	28-Flex	29-	30-GPP
ALU Out	31-GPP	32-Flex	33-GPP	34-GPP	35-GPP	36-Flex	37-GPP	38-GPP	39-GPP	40-
LS Out	41-GPP	42-GPP	43-GPP	44-GPP	45-	46-	47-GPP	48-	49-Flex	50-GPP
RegA Out	51-	52-	53-Flex	54-Flex	55-Flex	56-GPP	57-	58-	59-	60-Flex
RegB Out	61-GPP	62-GPP	63-GPP	64-GPP	65-Flex	66-	67-GPP	68-	69-	70-
MULT LSB	71-	72-GPP	73-GPP	74-GPP	75-	76-	77-Flex	78-	79-	80-
MULT MSB	81-	82-GPP	83-GPP	84-GPP	85-	86-	87-	88-	89-	90-

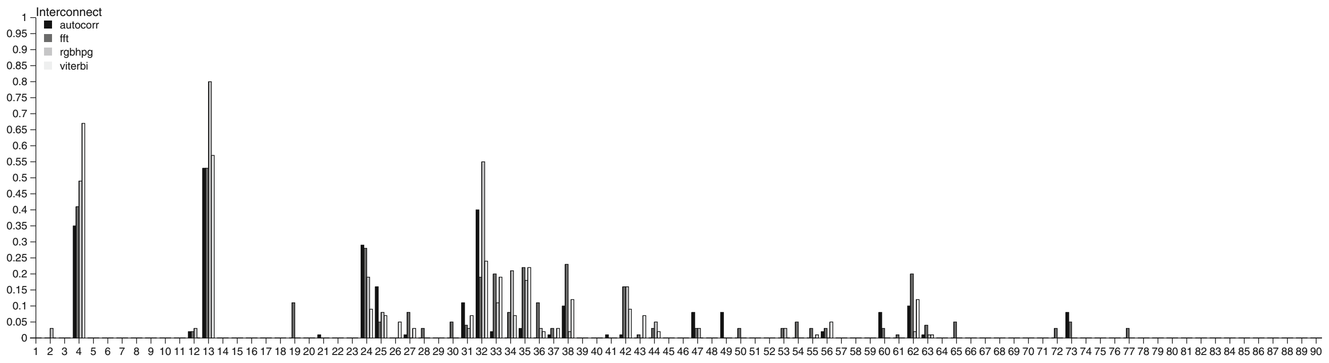


Figure 12 Details on utilization of interconnect links. Table 3 shows the link-ID used on the *x*-axis. The *y*-axis shows how often the link is used during the total execution of the benchmark.

FlexCore architecture, it is possible to skip the generation of such writes, as long as it is possible to statically find such superfluous register writes in the program. Figure 11b shows the relative number of register writes for the different architectures compared to the GPP version. On average, the exposed GPP allows us to remove 28% of all register writes and for FlexCore 36% are removed. This reduces the contention of the register file as well as saves power.[2]

6.2 Performance Evaluation

Cycle count alone is not sufficient to make a comparison of the different architectures. One also needs to consider implementation aspects. Table 2 shows the result of timing, power, and area estimations for the FlexCore, the Exposed GPP and a conventional GPP. The implementations are done in a 65 nm technology, but values for a 130 nm technology are also included for reference. As previously explained, we have modeled the microarchitecture without control and thus the implementation result for the GPP and the Exposed GPP are the same.

The layout implementations were performed for minimum delay, and as expected, the more modern 65 nm version has better delay values and worse power. For the remainder of the paper, we will use the values for 65 nm as our baseline. For the FlexCore architecture, the extra paths require 18% extra space. More importantly, the delay of the critical path is increased by 6% and the power dissipation is increased by 18%.

The high overhead in terms of delay and especially power makes a fully connected interconnect impractical. An earlier study [9] showed that the interconnect can be pruned without losing any performance in terms of cycle count. Here we conduct a more thorough investigation on which paths contribute to the improved performance for each application.

In the GPP architecture, there are 33 possible communication paths, while the fully connected interconnect offers 90 paths. Table 3 lists all these paths and highlights the ones that are used by our four benchmarks. Paths marked GPP are used when executing GPP code; additionally, when the full interconnect is used, paths marked Flex are also used. Here we see that out of the 90 paths available in the fully connected interconnect, 43 are *never* used in the benchmarks of this study.

Figure 12 illustrates the actual utilization of the paths in our fully connected interconnect. Each path listed in Table 3 is represented on the *x*-axis; for each benchmark, the bars show how often the link is used during the full execution. For all the benchmarks, the path between register output A and the ALU is very important (path number 13). Even though the trend is similar among all the four benchmarks, there are some variations. The total number of paths used in the benchmarks is very different between FFT (32) and the other three benchmarks (Autocor uses 18, FFT 32, RGBHPG 18, and Viterbi 19 paths). The distribution of used links also vary between FFT and the rest. The ten most used links in FFT cover 73% of all the transfers while the corresponding numbers are 87%, 93%, and 83% for Autocor, RGBHPG, and Viterbi. Note that

[2]It also complicates exception handling, which is however not the topic of this paper.

Table 4 Number of links needed for 95% of all transfers.

Benchmarks	#links
Autocor	13
FFT	26
RGBHPG	12
Viterbi	15
All	30

the links do not necessarily need to be the same in the different benchmarks.

Since all data movements are statically scheduled by the compiler, it is possible to prune idle communication paths from the flexible interconnect, without losing performance. Table 2 include the results of an architecture with the pruned interconnect as shown in Table 3. The results show that pruning the interconnect improves power dissipation as well as delay and area.

It is also enlightening to calculate the minimum number of links necessary to account for 95% of the transfers. This metric is listed in Table 4. Out of the used links, between 67% and 81% of the links use 95% of the transfers. This uneven distribution suggest that even more pruning is possible with small effect on performance. The combined statistics show that even though FFT uses a larger set of paths than the other benchmarks, it is not a true superset of the ones needed by Autocor, RGBHPG and Viterbi.

By combining the results from the placed and routed architectures with the cycle count, the true performance of the different approaches is shown. Table 5 shows the total time as well as the amount of energy (*power* · *clock period* · *cycle count*) needed to execute each benchmark.

These figures show that all four applications are executed faster on the FlexCore than on the GPP, when the cycle time is also considered. Compared to the Exposed GPP, we notice that the performance boost is attributed to both the exposed control and the flexible interconnect. With GPP as reference, the Exposed GPP gives up to 19% performance improvement, while the FlexCore delivers improvements up to 32% on the same benchmark (FFT).

The energy result on the other hand is not as promising. Comparing the FlexCore architecture to the GPP shows that only two of the benchmarks are executed with less energy (17% for Autocor and 20% for FFT) while two expend more energy (4% for RGBHPG and 14% for Viterbi). This shows that if energy is a first-level constraint, the improvement in cycle count is not large enough to outweigh the penalty associated with the fully connected interconnect.

Pruning the interconnect, as described earlier, reduces the delay and power significantly. This is reflected in the low execution time and energy dissipation when executing the four benchmarks. The FlexCore with the pruned interconnect is on average 19% faster and 14% less energy dissipating than the GPP.

6.3 Static Code Size

The total number of instructions for the benchmarks is shown in Table 6. The results are presented for both the full benchmark (excluding library code) and for the manually scheduled inner loop. First focusing on the full benchmarks, we see that the optimized versions are about the same or slightly smaller than the GPP versions.

Note that any change in the number of instruction is the result from the manually scheduled inner loops. Therefore we also present the number of instructions for these. Here we see that the FlexCore architecture has made it possible to decrease the number of instructions significantly for some benchmarks. For example, the inner loop of the FFT needs only half the number of cycles as on the conventional GPP. However, since an N-ISA instruction is about three times as large as a conventional GPP instruction, the total static code size for FlexCore is still larger than for a conventional GPP.

Table 5 Execution time and energy dissipation.

	Autocor		FFT		RGBHPG		Viterbi	
	Time (μs)	Energy (nJ)	Time (μs)	Energy (nJ)	Time (ms)	Energy (μJ)	Time (μs)	Energy (μJ)
GPP	3.2 (100%)	24 (100%)	124 (100%)	912 (100%)	7.3 (100%)	54 (100%)	574 (100%)	4.2 (100%)
Exposed GPP	2.8 (88%)	21 (88%)	101 (81%)	743 (81%)	6.8 (93%)	50 (93%)	574 (100%)	4.2 (100%)
FlexCore	2.3 (72%)	20 (83%)	84 (68%)	730 (80%)	6.4 (88%)	56 (104%)	552 (96%)	4.8 (114%)
Pruned	2.2 (69%)	16 (67%)	80 (65%)	586 (64%)	6.0 (82%)	44 (81%)	522 (91%)	3.8 (90%)

Table 6 Code size (number of instructions) for both application and the inner loop for each benchmark.

Datapath	Autocor	Autocor (Loop)	FFT	FFT (Loop)	RGBHPG	RGBHPG (Loop)	Viterbi	Viterbi (Loop)
GPP	60 (100%)	16 (100%)	432 (100%)	42 (100%)	197 (100%)	43 (100%)	366 (100%)	66 (100%)
Exposed GPP	61 (102%)	17 (106%)	424 (98%)	30 (71%)	198 (101%)	42 (98%)	367 (100%)	67 (102%)
FlexCore	56 (93%)	12 (75%)	414 (96%)	21 (50%)	191 (97%)	37 (86%)	360 (98%)	61 (92%)

The increase of instructions seen for the Exposed GPP originates from some of the manual scheduling, which added prefix code to handle the first iteration of loops.

7 Related Work

Reconfigurable architectures is an active area of research. Dedicated hardware is becoming less attractive because of huge initial costs, long time to market, and inability to adapt to new and changing standards. Reconfigurable hardware is a promising approach to address these problems, without forsaking the performance of dedicated hardware. Hartenstein has compiled a thorough survey of reconfigurable architectures [10]. Many modes of reconfigurability have been proposed: reconfigurable accelerators may be connected to a standard pipeline [11]; or reconfigurable tiles may be orchestrated to solve given problems [12, 13]. In contrast, the FlexSoC approach as presented here employs reconfigurability only in the instruction decoding hardware, leaving the actual data processing to highly efficient dedicated hardware.

The exposed datapath concept has recently been used in the No Instruction Set Computer (NISC) [2, 7, 14] project, where the control pipeline is removed and the controller emits a wide instruction word each cycle. Co-design refinement of hardware and software is used to reach the desired performance. The reported speedups are comparable to those we see for the FlexCore example. However, the static code size of a NISC program is claimed to be comparable to that of a GPP. While this might be true for a co-design approach where common complex operations can be implemented with few control bits, we have not seen the same results for the FlexCore architecture. In FlexSoC, we rely on compression and run-time expansion to solve the code size problem. Increased controllability of the datapath has been motivated by the reduction in hardware complexity, in a similar way as in the transport triggered architecture [15].

Liang et al. [16] propose an architecture based on a reconfigurable interconnect and show good performance for some domain-specific computations. It is, however, not clear how the results translate to a wider domain of applications.

A common way to accelerate multimedia applications is to add sub-word parallelism within the datapath units (SIMD). This technique is used both in modern general-purpose computers and specific media processors. For a five-stage DLX implementation, Nia and Fatemi report a speedup of more than a factor of three with only minor growth in chip area [17]. The approach is orthogonal to those proposed here and would seem to make a fruitful addition to a FlexSoC core.

Similarly to FlexSoC, the FITS project [18–20] also envisions the use of flexible instruction decoders. Application profiling allows the selection of a 16-bit application-specific ISA that gives the same performance as the 32-bit baseline case. FlexSoC combines a similar application-specific ISA approach with the performance gains offered by the exposed datapath and the flexible interconnect.

The translation envisioned in the FlexSoC project is somewhat similar to microcode processing, where a complex ISA is broken down into micro-operations that are executed on the pipeline. The main purpose of microcode is to separate the architecture from the implementation and the microcode is usually derived from the already given ISA. In FlexSoC, this constraint is relaxed and the AS-ISA can be created by the compiler to fit the needs of the applications.

8 Conclusion

The exposed datapath of FlexCore offers distinct performance benefits when compared to a GPP with corresponding datapath units. The flexible interconnect network further improves performance, and also allows special-purpose datapath units to be integrated while maintaining a uniform programming interface. With knowledge of the datapath structure, a compiler can realize these performance benefits. Additionally, it is always possible to execute programs as if they had been compiled for MIPS and at cycle counts comparable to a standard MIPS implementation.

We have analyzed four embedded applications and shown that for this set of applications, FlexCore is between 10% and 40% faster in terms of cycle count compared to a conventional five-stage GPP with the same datapath units. This speedup is attributed to both the exposed datapath and the flexible interconnect. Using cycle times obtained from placed and routed layouts, we show that this cycle count translates to a total execution-time improvement of up to 30%. We also show that it is feasible to prune the fully connected interconnect without losing any performance in cycle count. For our benchmarks, one third of the possible links account for about 95% of all the data transfers. This allows for a more energy-efficient implementation and also improves the cycle time. However, the boost in performance comes at a cost of both instruction bandwidth and static code size: the FlexCore instructions are about three times as wide as a 32-bit GPP instruction. The number of static instructions is reduced for the FlexCore, but not by a factor of three. Therefore, the static code size will be larger for an application compiled for FlexCore. A reconfigurable instruction decoder, as proposed in the FlexSoC framework, is clearly needed to reduce both the static code size and the instruction bandwidth.

Future work includes evaluating a reconfigurable instruction decoder together with FlexCore. Different compression schemes can be expected to be more or less suited to the ISA transformations needed, and to carry different implementation costs. Configuration of the instruction decoder could be a one-time event; but run-time, on-demand reconfiguration offers intriguing possibilities, where several tasks, each with a distinct AS-ISA, could share the same hardware.

Acknowledgments We thank our FlexSoC colleagues (Hughes, Jeppson, Sheeran) for support and helpful discussions, Marius Grannæs for valuable comments, and Jonas Ferry for his help during the project. The FlexSoC project is sponsored by the Swedish Foundation for Strategic Research.

References

1. Hughes, J., Jeppson, K., Larsson-Edefors, P., Sheeran, M., Stenstrom, P., & Svensson, L. J. (2003). FlexSoC: Combining flexibility and efficiency in SoC designs. In *Proceedings of the IEEE NorChip conference*.
2. Reshadi, M., Gorjiara, B., & Gajski, D. (2005). Utilizing horizontal and vertical parallelism with no-instruction-set compiler for custom datapaths. In *International conference on computer design* (*ICCD*), October.
3. Mårts, J., & Carlqvist, T. (2006) *A hardware audio decoder using flexible datapaths.* MSc Thesis, Chalmers University of Technology, March.
4. Patterson, D. A., & Hennessy, J. L. (1998). *Computer organization & design, the hardware/software interface* (2nd ed.). Morgan Kaufman.
5. Kongetira, P., Aingaran, K., & Olukotun, K. (2005). Niagara: A 32-way multithreaded sparc processor. *IEEE Micro, 25*(2), 21–29.
6. Balakrishnan, S., Rajwar, R., Upton, M., & Lai, K. (2005). The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Computer Architecture News, 33*(2), 506–517.
7. Reshadi, M., & Gajski, D. (2005). A cycle-accurate compilation algorithm for custom pipelined datapaths. In *International symposium on hardware/software codesign and system synthesis* (*CODES+ISSS*), September.
8. *Encounter User Guid Version 6.2*.
9. Själander, M., Larsson-Edefors, P., & Björk, M. (2007). A flexible datapath interconnect for embedded applications. In *IEEE Computer Society Annual Symposium on VLSI*, May.
10. Hartenstein, R. (2001). A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of design, automation and test in Europe, 2001* (pp. 642–649), March.
11. Ye, Z. A., Moshovos, A., Hauck, S., & Banerjee, P. (2000). CHIMAERA: A high-performance architecture with a

tightly-coupled reconfigurable functional unit. In *ISCA '00: Proceedings of the 27th annual international symposium on computer architecture* (pp. 225–235). New York, NY, USA: ACM Press.
12. M. B. T. et al. (2004). Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on computer architecture* (p. 2). Washington, DC, USA: IEEE Computer Society.
13. K. S. et al. (2004). TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Trans. Archit. Code Optim., 1*(1), 62–93.
14. Gorjiara, B., Reshadi, M., & Gajski, D. (2006). Designing a custom architecture for DCT using NISC design flow. In *ASP-DAC'06 Design contest*.
15. Corporaal, H. (1999). Ttas: Missing the ilp complexity wall. *Journal of Systems Architecture, 45*(12–13), 949–973.
16. Liang, X., Athalye, A., & Hong, S. (2005). Dynamic coarse grain dataflow reconfiguration technique for real-time systems design. In *The 2005 IEEE international symposium on circuits and systems* (pp. 3511–3514). IEEE Computer Society, May.
17. Nia, E., & Fatemi, O. (2003). Multimedia extensions for DLX processor. In *Proceedings of the 10th IEEE international conference on electronics, circuits and systems* (pp. 1010–1013), December.
18. Cheng, A., Tyson, G., & Mudge, T. (2004). FITS: Framework-based instruction-set tuning synthesis for embedded application specific processors. In *DAC '04: Proceedings of the 41st annual conference on design automation* (pp. 920–923). ACM Press.
19. Cheng, A., Tyson, G., & Mudge, T. (2005). PowerFITS: Reduce dynamic and static i-cache power using application specific instruction set synthesis. In *Performance analysis of systems and software, 2005. ISPASS 2005. IEEE International Symposium on* (pp. 32–41).
20. Cheng, A. C., & Tyson, G. S. (2006). High-quality ISA synthesis for low-power cache designs in embedded microprocessors. *IBM Journal of Research and Development, 50*(2), 299–309.

Martin Thuresson received the MSc degree from Chalmers University of Technology, Göteborg, Sweden, where he is currently a PhD candidate in the Computer Science and Engineering Department. His main research interests are low overhead compression techniques and embedded reconfigurable systems. He is a student member of the IEEE.

Magnus Själander received in 2003 the MSc degree from Luleå University of Technology, Luleå, Sweden. He is currently a PhD candidate in the Computer Science and Engineering Department at Chalmers University of Technology, Göteborg, Sweden. His main research interests are embedded processors and system design. He is a graduate student member of the IEEE.

Magnus Björk did his PhD in computing science at Chalmers University of Technology, and has held post doc positions at Chalmers University of Technology and University of Oxford. His main research interests include formal verification, automated theorem proving, and compilation.

Lars Svensson was awarded the PhD degree in Applied Electronics from Lund Universtiy, Sweden, in 1990. He has been with Chalmers University since 2002. His research interests include low-power circuit techniques, innovative architectures, and switching noise abatement. He is the author or co-author of several dozen papers and is listed as an inventor or co-inventor on more than 20 patents.

Per Larsson-Edefors holds the Chair of Computer Engineering at Chalmers University of Technology, Gothenburg, Sweden. He received the M.Sc. degree in Electrical Engineering and Engineering Physics in 1991 and the Ph.D. degree in Electronic Devices in 1995; both degrees from Linköping University, Sweden. He was a visiting scientist at National Microelectronics Research Center, Ireland, in 1996/1997 and a visiting professor at Intel Corporation's Circuit Research Lab in 2000. His research activities are focused on low-power high-performance digital circuits and microarchitectures, as well as circuit and interconnect macromodeling for efficient performance and power estimation.

Per Stenstrom is a professor of computer engineering at Chalmers University of Technology. His research interests are devoted to design principles for high-performance computer systems. He is an author of two textbooks and more than a hundred research publications. He is regularly serving program committees of major conferences in the computer architecture field. He has been an editor of IEEE Transaction on Computers, is an editor of Journal of Parallel and Distributed Computing, the IEEE TCCA Computer Architecture Letters, and a founding editor-in-chief of the Journal of High-Performance Embedded Architectures and Compilation Techniques. He has served as General as well as Program Chair of the ACM/IEEE Int. Symposium on Computer Architecture. He is a Fellow of the IEEE and member of ACM.