ASYNCHRONOUS INTEGRATION OF COARSE-GRAINED RECONFIGURABLE XPP-ARRAYS INTO PIPELINED RISC PROCESSOR DATAPATH

Jürgen Becker, Alexander Thomas and Maik Scheer
Institut für Technik der Informationsverarbeitung (ITIV), Fakultät für Elektrotechnik und Informationstechnik, Universität Karlsruhe (TH), Karlsruhe, Germany

Abstract: Nowadays, the datapaths of modern microprocessors reach their limits by using static instruction sets. A way out of these limitations is a dynamic reconfigurable processor datapath extension achieved by integrating traditional static datapaths with the coarse-grain dynamic reconfigurable XPP-architecture (eXtreme Processing Platform). Therefore, a loosely asynchronous coupling mechanism of the corresponding datapath units has been developed and integrated onto a CMOS 0.13 µm standard cell technology from UMC. Here the SPARC compatible LEON processor is used, whereas its static pipelined instruction datapath has been extended to be configured and personalized for specific applications. This allows a various and efficient use, e.g. in streaming application domains like MPEG-4, digital filters, mobile communication modulation, etc. The chosen coupling technique allows asynchronous concurrency of the additionally configured compound instructions, which are integrated into the programming and compilation environment of the LEON processor.

Key words: Reconfigurable Datapath, XPP Architecture, LEON Processor

1. INTRODUCTION

The limitations of conventional processors are becoming more and more evident. The growing importance of stream-based applications makes coarse-grained dynamically reconfigurable architectures an attractive alternative [3], [4], [6], [7]. They combine the performance of ASICs, which

Please use the following format when citing this chapter:
Becker, Jürgen, Thomas, Alexander, Scheer, Maik, 2006, in IFIP International Federation for Information Processing, Volume 200, VLSI-SOC: From Systems to Chips, eds. Glesner, M., Reis, R., Indrusiak, L., Mooney, V., Eveking, H., (Boston: Springer), pp. 263-279.

are very risky and expensive (development and mask costs), with the flexibility of traditional processors [5].

In spite of the possibilities we have today in VLSI development, the basic concepts of microprocessor architectures are the same as 20 years ago. The main processing unit of modern conventional microprocessors, the datapath, in its actual structure follows the same style guidelines as its predecessors. Although the development of pipelined architectures or superscalar concepts in combination with data and instruction caches increases the performance of a modern microprocessor and allows higher frequency rates, the main concept of a static datapath remains. Therefore, each operation is a composition of basic instructions that the used processor owns. The benefit of the processor concept lays in the ability of executing strong control dominant application. Data or stream oriented applications are not well suited for this environment. The sequential instruction execution isn't the right target for that kind of applications and needs high bandwidth because of permanent retransmitting of instruction/data from and to memory. This handicap is often eased by using of caches in various stages. A sequential interconnection of filters, which do the according data manipulating without writing back the intermediate results would get the right optimization and reduction of bandwidth. Practically, this kind of chain of filters should be constructed in a logical way and configured during runtime. Existing approach to extend instruction sets uses static modules, not modifiable during runtime.

Customized microprocessors or ASICs are optimized for one special application environment. It is nearly impossible to use the same microprocessor core for other applications without loosing the performance gain of this architecture.

A new approach of a flexible and high performance datapath concept is needed, which allows to reconfigure the functionality and make this core mainly application independent without losing the performance needed for stream-based applications.

This contribution introduces a new concept of loosely coupled implementation of the dynamic reconfigurable XPP architecture from PACT Corp. into a static datapath of the SPARC compatible LEON processor. Thus, this approach is different from those, where the XPP operates as a completely separate component within one Configurable System-on-Chip (CSoC), together with a processor core, global/local memory topologies and efficient multi-layer AMBA-bus interfaces [11]. Here, from the programmers' point of view the extended and adapted datapath seems like a dynamic configurable instruction set. It can be customized for a specific application and accelerate the execution enormously. Therefore, the programmer has to create a number of configurations, which can be

uploaded to the XPP-Array at run time, e.g. this configuration can be used like a filter to calculate stream-oriented data. It is also possible, to configure more than one function at the same time and use them simultaneously. This concept promises an enormously performance boost and the needed flexibility and power reduction to perform a series of applications very effectively.

2. LEON RISC MICROPROCESSOR

For implementation of this concept we chose the 32-bit SPARC V8 compatible microprocessor [1][2], LEON. This microprocessor is a synthesizable, free available VHDL model which has a load/store architecture and has a five stages pipeline implementation with separated instruction and data caches.

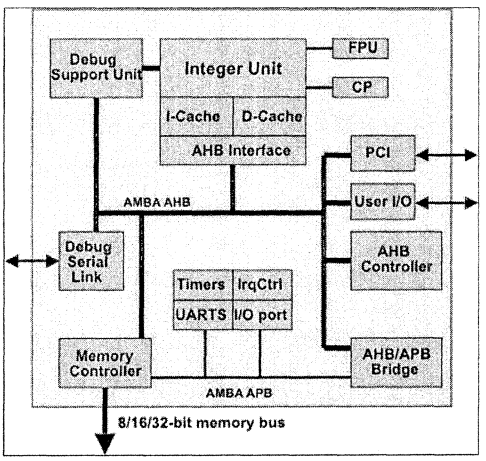

Figure 1. LEON Architecture Overview

As shown in Figure 1 the LEON is provided with a full implementation of AMBA 2.0 AHB and APB on-chip busses, a hardware multiplier and divider, programmable 8/16/32-bit memory controller for external PROM, static RAM and SDRAM and several on-chip peripherals such as timers, UARTs, interrupt controllers and a 16-bit I/O port. A simple power down mode is implemented as well.

LEON is developed by the European Space Agency (ESA) for future space missions. The performance of LEON is close to an ARM9 series but

don't have a memory management unit (MMU) implementation, which limits the use to single memory space applications. In Figure 2 the datapath of the LEON integer unit is shown.

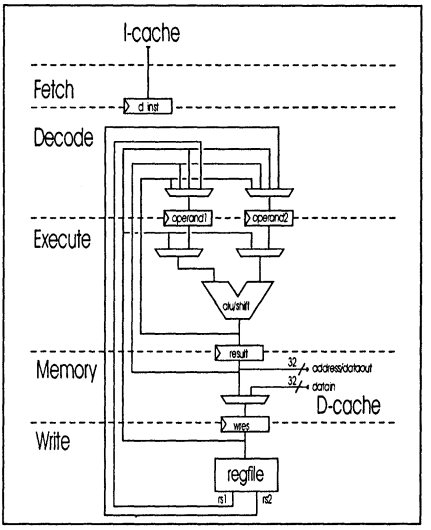

Figure 2. LEON Pipelined Datapath Structure

3. EXTREME PROCESSING PLATFORM – XPP

The XPP architecture [6], [7], [8] is based on a hierarchical array of coarse-grain, adaptive computing elements called *Processing Array Elements (PAEs)* and a *packet-oriented communication network.* The strength of the XPP technology originates from the combination of array processing with unique, powerful run-time reconfiguration mechanisms. Since configuration control is distributed over a *Configuration Manager (CM)* embedded in the array, PAEs can be configured rapidly in parallel while neighboring PAEs are processing data. Entire applications can be configured and run independently on different parts of the array. Reconfiguration is triggered externally or even by special event signals originating within the array, enabling self-reconfiguring designs. By utilizing protocols implemented in hardware, data and event packets are used to process, generate, decompose and merge streams of data.

The XPP has some similarities with other coarse-grain reconfigurable architectures like the Kress-Array [3] or Raw Machines [4], which are

specifically designed for stream-based applications. XPP's main distinguishing features are its automatic packet-handling mechanisms and its sophisticated hierarchical configuration protocols for runtime- and self-reconfiguration.

3.1 Array Structure

A CM consists of a state machine and internal RAM for configuration caching. The PAC itself (see top right-hand side of Figure 3) contains a configuration bus which connects the CM with PAEs and other configurable objects. Horizontal busses carry data and events. They can be segmented by configurable switch-objects, and connected to PAEs and special I/O objects at the periphery of the device.

A PAE is a collection of PAE objects. The typical PAE shown in Figure 3 (bottom) contains a BREG object (back registers) and an FREG object (forward registers) which are used for vertical routing, as well as an ALU object which performs the actual computations. The ALU performs common fixed-point arithmetical and logical operations as well as several special three input opcodes like multiply-add, sort, and counters. Events generated by ALU objects depend on ALU results or exceptions, very similar to the state flags of a classical microprocessor. A counter, e.g., generates a special event only after it has terminated. The next section explains how these events are used. Another PAE object implemented in the XPP is a memory object which can be used in FIFO mode or as RAM for lookup tables, intermediate results etc. However, any PAE object functionality can be included in the XPP architecture.

3.2 Packet Handling and Synchronization

PAE objects as defined above communicate via a packet-oriented network. Two types of packets are sent through the array: data packets and event packets. Data packets have a uniform bit width specific to the device type. In normal operation mode, PAE objects are self synchronizing. An operation is performed as soon as all necessary data input packets are available. The results are forwarded as soon as they are available, provided the previous results have been consumed. Thus it is possible to map a signal-flow graph directly to ALU objects. Event packets are one bit wide. They transmit state information which controls ALU execution and packet generation.

3.3 Configuration

Every PAE stores locally its current configuration state, i.e. if it is part of a configuration or not (states „configured" or „free"). Once a PAE is configured, it changes its state to „configured". This prevents the CM from reconfiguring a PAE which is still used by another application. The CM caches the configuration data in its internal RAM until the required PAEs become available.

While loading a configuration, all PAEs start to compute their part of the application as soon as they are in state „configured". Partially configured applications are able to process data without loss of packets. This concurrency of configuration and computation hides configuration latency.

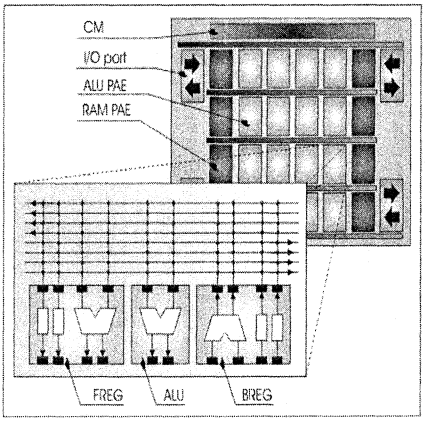

Figure 3. Structure of an XPP device

3.4 XPP Application Mapping

The Native Mapping Language (NML), a PACT proprietary structural language with reconfiguration primitives, was developed by PACT to map applications to the XPP array. It gives the programmer direct access to all hardware features.

In NML, configurations consist of modules which are specified as in a structural hardware description language, similar to, for instance, structural VHDL, PAE objects are explicitly allocated, optionally placed, and their connections specified. Hierarchical modules allow component reuse, especially for repetitive layouts. Additionally, NML includes statements to support configuration handling. A complete NML application program consists of one or more modules, a sequence of initially configured modules,

differential changes, and statements which map event signals to configuration and prefetch requests. Thus configuration handling is an explicit part of the application program.

A complete XPP Development Suite (XDS) is available from PACT. For more details on XPP-based architectures and development tools see [6].

4. LEON INSTRUCTION DATAPATH EXTENSION

The system is designed to offer a maximum of performance. LEON and XPP should be able to communicate with each other in a simple and high performance manner. While the XPP is a dataflow orientated device, the LEON is a general purpose processor, suitable for handling control flow [1], [2]. Therefore, LEON is used for system control. To do this, the XPP is integrated into the datapath of the LEON integer unit, which is able to control the XPP.

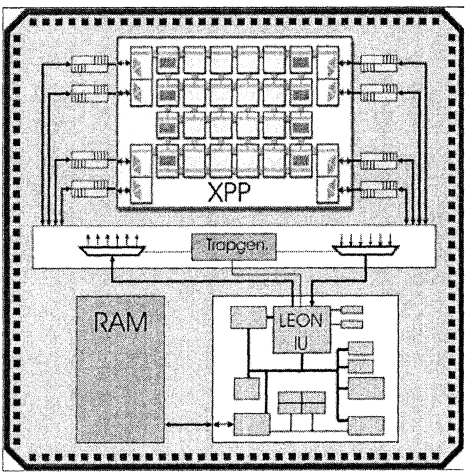

Figure 4. Extended Datapath Overview

Due to unpredictable operation time of the XPP algorithm, integration of XPP into LEON datapath is done in a loosely-coupled way (see Figure 4). Thus the XPP array can operate independent from the LEON processor, which is able to control and reconfigure the XPP during runtime. Since the configuration of XPP is handled by LEON, the CM of the XPP is not necessary and can be left out of the XPP array. The configuration codes are

stored in the LEON RAM. LEON transfers the needed configuration from its system RAM into the XPP and creates the needed algorithm on the array.

To enable a maximum of independence of XPP from LEON, all ports of the XPP – input ports as well as output ports – are buffered using dual clock FIFOs. Dual-clocked FIFOs are implemented into the IO-Ports between LEON and XPP. To transmit data to the extended XPP-based datapath the data are passed through an IO-Port as shown in Figure 5. In addition to the FIFO the IO-Ports contain logic to generate handshake signals and an interrupt request signal. The IO-Port for receiving data from XPP is similar to Figure 5 except that the reversed direction of the data signals. This enables that XPP can work completely independent from LEON as long as there are input data available in the input port FIFOs and free space for result data in the output port FIFOs. There are a number of additionally features implemented in the LEON pipeline to control the data transfer between LEON and XPP.

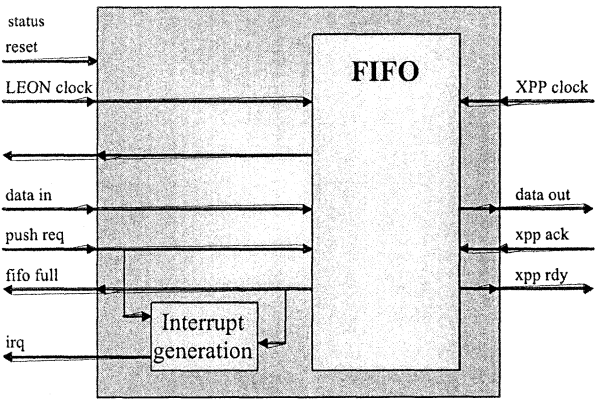

Figure 5. LEON-to-XPP dual-clock FIFO

When LEON tries to write to an IO-Port containing a full FIFO or read from an IO-Port containing an empty FIFO a trap is generated. This trap can be handled through a trap handler. There is a further mechanism - pipeline-holding - implemented, to allow LEON holding the pipeline and wait for free FIFO space during XPP write access respectively wait for a valid FIFO value during XPP read access. When using pipeline-holding the software developer has to avoid reading from an IO-Port with empty FIFO while the XPP, respectively the XPP input IO-Ports, contains no data to produce outputs. In this case a deadlock will occur and the complete system has to be reseted.

XPP can generate interrupts for the LEON when trying to read a value from an empty FIFO port or to write a value to a full FIFO port. The occurrence of interrupts indicates that the XPP array cannot process the next step because it has either no input values or it cannot output the result value. The interrupts generated by the XPP are maskable.

The interface provides information about the FIFOs. LEON can read the number of valid values the FIFO contains. The interface to the XPP appears to the LEON as a set of special registers (see Figure 6). These XPP registers can be categorized in communication registers and status registers.

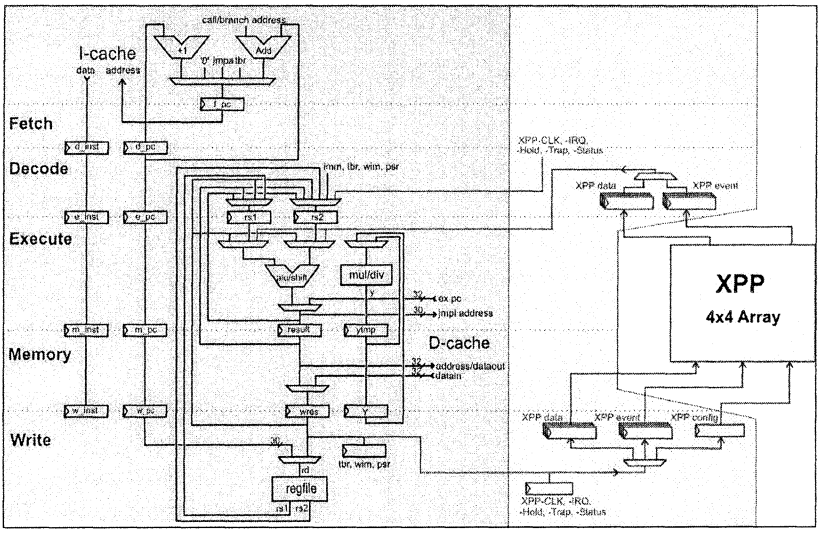

Figure 6. Extended LEON Instruction Pipeline

For data exchange the XPP communication registers are used. Since XPP provides three different types of communication ports, there are also three types of communication registers, whereas every type is splitted into an input part and an output part:

The data for the process are accessed through XPP data registers. The number of data input and data output ports as well as the data bit width depends on the implemented XPP array.

XPP can generate and consume events. Events are one bit signals. The number of input events and output events depends on the implemented XPP array again.

Configuration of the XPP is done through the XPP configuration register. LEON reads the required configuration value from a file – stored in his system RAM – and writes it to the XPP configuration register.

There are a number of XPP status register implemented to control the behavior and get status information of the interface. Switching between the usage of trap handling and pipeline holding can be done in the hold register. A XPP clock register contains a clock frequency ratio between LEON and XPP. By writing this register LEON software can set the XPP clock relative to LEON clock. This allows to adapt the XPP clock frequency to the required XPP performance and consequently to influence the power consumption of the system. Writing zero to the XPP clock register turns off the XPP. At last there is a status register for every FIFO containing the number of valid values actually available in the FIFO.

This status registers provides a maximum of flexibility in communication between LEON and XPP and enables different communication modes:

If there is only one application running on the system at the time, software may be developed in pipeline-hold mode. Here LEON initiates data read or write from respectively to XPP. If there is no value to read respectively no value to write, LEON pipeline will be stopped until read or write is possible. This can be used to reduce power consumption of the LEON part.

In interrupt mode, XPP can influence the LEON program flow. Thus, the IO-Ports generate an interrupt depending on the actual number of values available in the FIFO. The communication between LEON and XPP as done in interrupt service routines.

Polling mode is the classical way to access the XPP. Initiated by a timer-event LEON reads all XPP ports containing data and writes all XPP ports containing free FIFO space. Between those phases LEON can perform other calculations.

It is anytime possible to switch between those strategies within one application.

The XPP is delivered containing a configuration manager to handle configuration and reconfiguration of the array. In this concept the configuration manager is dispensable because the configuration as well as any reconfiguration is controlled by the LEON through the XPP configuration register. All XPP configurations used for an application are stored in the LEON's system RAM.

5. TOOL AND COMPILER INTEGRATION

The LEON's SPARC V8 instruction set [1] was extended by a new subset of instructions to make the new XPP registers accessible through software. These instructions based on the SPARC instruction format are not SPARC V8 standard conform. Corresponding to the SPARC V8 conventions of RISC (load/store) architectures the instruction subset can be split in two general types. Load/store instructions can exchange data between the LEON memory and the XPP communication registers. The number of cycles per instruction is similar to the standard load/store instructions of the LEON. Read/write instructions are used for transfers between LEON registers. Since the LEON register-set is extended by the XPP registers the read/write instructions are extended also to access XPP registers. Status registers can only be accessed with read/write instructions. Execution of arithmetic instructions or compare operations on XPP registers is not possible. Values have to be written to standard LEON registers before they can be used. The complete system is still SPARC V8 compatible. By doing this, the XPP part is completely unused.

The LEON is provided with the LECCS cross compiler system [9] standing under the terms of LGPL. This system consists of modified versions of the binary utilities 2.11 (binutils) and GNU cross compile 2.95.2 (GCC). To make the new instruction subset available to software developers, the assembler of the binutils has been extended by a number of instructions according to the implemented instruction subset. The new instructions have the same mnemonic as the regular SPARC V8 load, store, read and write instructions. Only the new XPP registers have to be used as source respectively target operands. Since the modifications of LECCS are straightforward extensions, the cross compiler system is backward compatible to the original version. The availability of the source code of LECCS has offered the option to extend the tools by the new XPP operations in the described manner.

The development of the XPP algorithms has to be done with separate tools, provided by PACT XPP Technologies.

5.1 Current Compiler Integration

Well, the actual status of the compiler integration allows the using of assembler macro calls within the C-code. Figure 7 shows the development flow in detail. The first step has to be done is the partitioning of your software tasks: Which part should be executed within the LEON datapath and which part should be executed by the XPP architecture. The resulted

partition has a big influence on the communication strategy, which strongly affects the reachable performance. The next step is to select the communication strategy. Which transfer mode is the best, depends on several facts of your environment. The most important reason is the time point of the availability. Are my data words already in memory and can be accessed every time I want or there is no way to predict when my data is available. Those things decide which mode is the best for my situation. Like described above you can choose between three modes: Trap Mode, Hold Mode and Polling Mode. The best way to implement the communication is by using two or three modes in parallel. Therefore it is possible to use each register in its own mode. This feature is not implemented yet, but will be available shortly.

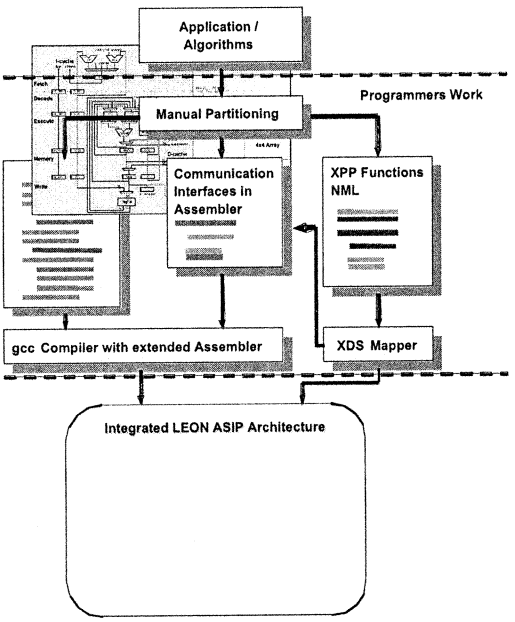

Figure 7. Application Development Flow

Now, the programmer has to take into account the exactly interface structure of his software which depends on his software specification demands. It is important to implement all needed synchronization mechanisms to guarantee full data consistency. The resulted communication structure has to be implemented in assembler and covered with macros. Of

course, the assembler parts can be used directly within C-code by enclosing with assembler statement blocks.

The development of the XPP configuration has to be done with in the XDS, like already described above. The parameters of the resulted configuration, like the used IO-ports, duration of a calculation cycle, the size of the configuration stream, and so on must be taken into account within the communication interface realization. The size of a configuration stream e.g. affects the time which is needed to configure the XPP. If you have strong real-time demands, this size can lead to a violation of your real-time behavior and to prevent this mistake has to be taken into account. The other parameter like to IO port number implicates which XPP register should be used for the IO. Those parameters bind both sides of our application tightly where the assembler communication routines describe the way this binding is realized.

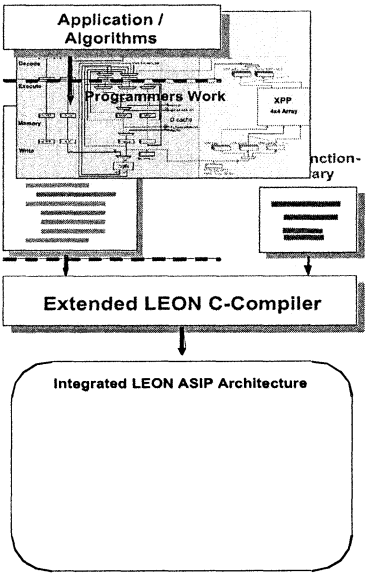

Figure 8. Future Application Development Flow

After the compilation of the resulted code we get an executable program with a data block which contains the implemented configuration for the XPP. The current way to develop an application for our architecture is surely not easy to use. Therefore we are working on the next generation compiler support which offers more transparency to users.

5.2 Future Compiler Integration

The future implementation of the software tools and compilers are shown in figure 8. The main advantage of this concept is the using of the XPP function library which contains a set of function templates with prepared interface descriptions. The programmer can decide at the development time which part of his application should be executed by LEON and which one by the XPP by using the XPP function library. This procedure is comparable to the partition in the current development flow with one difference: the user has just to decide the way how the XPP and LEON have to communicate but don't have to implement the communication routines. Within the C-Code he just inserts selected functions like he always do with standard C functions and the new compiler inserts the appropriate routines for the communication. Therefore the XPP function library contains beside the XPP function templates communication routine templates. The compiler uses those functional drafts for exactly environment parameterization and supports in this way the automatic implementation. The main gain of this concept is that the XPP unit seems nearly transparent to the programmer. He just has to take into account the exactly execution distribution and make for the XPP part appropriate function calls.

Table 1. Performance on IDCT (8x8)

	LEON alone	LEON with XPP in IRQ Mode	LEON with XPP in Polling Mode	LEON with XPP in Hold Mode
Configuration of XPP	———	71.308 ns 17.827 cycles	84.364 ns 21.091 cycles	77.976 ns 19.494 cycles
2D IDCT (8x8)	14.672 ns 3.668 cycles	3.272 ns 818 cycles	3.872 ns 968 cycles	3.568 ns 892 cycles

There is no guarantee for optimal application implementation by using the new development flow. This concept has no mechanisms for performance estimation so its programmers job to make an optimal application partitioning to get the best results.

6. APPLICATION RESULTS

As a first analysis application a inverse DCT applied to 8x8 pixel block was implemented. For all simulations we used 250 MHz clock frequency for LEON processor and 50 MHZ clock frequency for XPP. The usage of XPP accelerates the computation of the IDCT about factor four, depending on the communication mode. However XPP has to be configured before computing the IDCT on it. Table 1 also shows the configuration time for this algorithm.

As shown in figure 9, the benefit brought by XPP rises with the number of IDCT blocks computed by it before reconfiguration, so the number of reconfigurations during complex algorithms should be minimized.

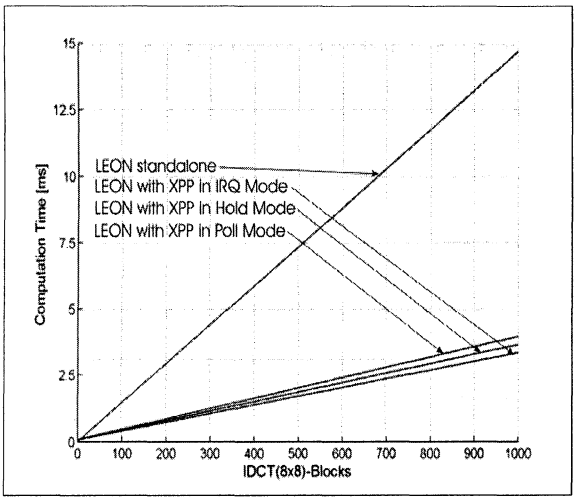

Figure 9. Computation Time of IDCT 8x8-Blocks

A first complex application implemented on the system is MPEG-4 decoding. The optimization of the algorithm partitioning on LEON and XPP is still under construction. In Figure 8 the block diagram of the MPEG-4 decoding algorithm is shown. Frames with 320 x 240 pixel was decoded. LEON by using SPARC V8 standard instructions decodes one frame in 23,46 seconds. In a first implementation of MPEG-4 using the XPP, only the IDCT is computed by XPP, the rest of the MPEG-4 decoding is still done with LEON. Now, with the help of XPP, one frame is decoded in 17,98 s. This is a performance boost of more then twenty percent. Since the XPP performance gain by accelerating the iDCT algorithm only is very low in the moment, we work on XPP implementations of Huffmann-decoding, dequantization and prediction-decoding. So the performance boost of this concept against the standalone LEON will be increased.

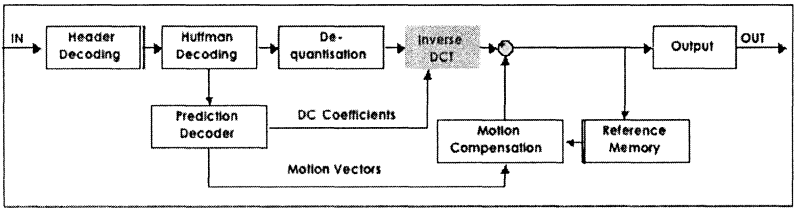

Figure 10. MPEG-4 Decoder Block Diagram

7. CONCLUSION

Today, the instruction datapaths of modern microprocessors reach their limits by using static instruction sets, driven by the traditional von Neumann or Harvard architectural principles. A way out of these limitations is a dynamic reconfigurable processor datapath extension achieved by integrating traditional static datapaths with the coarse-grain dynamic reconfigurable XPP-architecture (eXtreme Processing Platform). Therefore an asynchronously loosely-coupled mechanism for the given microprocessor data path has been developed and integrated onto the UMC CMOS 0.13 µm standard cell technology. The SPARC compatible LEON RISC processor has been used, whereas its static pipelined instruction data path has been extended to be configurable and personalize able for specific applications. This compiler-compatible instruction set extension allows a various and efficient use, e.g. in streaming application domains like MPEG-4, digital filters, mobile communication modulation, etc. The introduced coupling technique by flexible dual-clock FIFO interfaces allows asynchronous concurrency and adapting the frequency of the configured XPP datapath dependent on actual performance requirements, e.g. for avoiding unneeded cycles and reducing power consumption.

As presented above, the introduced concept combines the flexibility of a general purpose microprocessor with the performance and power consumption of coarse-grained reconfigurable datapath structures. Here, two programming and computing paradigms (control-driven von Neumann and transport-triggered XPP) are unified within one hybrid architecture with the advantages of two clock domains. The ability to reconfigure the transport-triggered XPP makes the system independent from standards or specific applications. This concept opens potential to develop multi-standard communication devices like software radios by using extended processor architectures with adapted programming and compilation tools. Thus, new standards can be easily implemented through software updates. The system is scalable during design time through the scalable array-structure of the

used XPP extension. This extends the range of suitable applications from products with less multimedia functions to complex high performance systems.

In spite of all the introduced features of the resulted architecture the processor core (LEON processor) is still SPARC compliant. This advantage is necessary in the migration of the reconfigurable hardware extensions into the processor domain. The important advantage is the compatibility of the prior developed compilers and debugging tools which can still be used for developing purposes and older software version are still executable on this architecture. It is also important to keep the effort for compiler tool extension small just by using the available compiler technologies.

REFERENCES

1. The SPARC Architecture Manual, Version 8, SPARC international INC., http://www.sparc.com
2. Jiri Gaisler: The LEON Processor User's Manual, http://www.gaisler.com
3. R. Hartenstein, R. Kress, and H. Reinig. A new FPGA architecture for word-oriented datapaths. In Proc. FPL'94, Prague, Czech Republic, September 1994. Springer LNCS 849
4. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, and P. Finch. Baring it all to software: Raw machines. IEEE Computer, pages 86-93, September 1997
5. J. Becker (Invited Tutorial): Configurable Systems-on-Chip (CSoC); in: Proc. of 9th Proc. of XV Brazilian Symposium on Integrated Circuit Design (SBCCI 2002), Porto Alegre, Brazil, September 5-9, 2002
6. PACT Corporation: http://www.pactcorp.com
7. The XPP Communication System, PACT Corporation, Technical Report 15, 2000
8. V. Baumgarte, F. Mayr, A. Nückel, M. Vorbach, M. Weinhardt: PACT XPP - A Self-Reconfigurable Data Processing Architecture; The 1st Int'l. Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA'01), Las Vegas, NV, June 2001
9. LEON/ERC32 Cross Compilation System (LECCS), http://www.gaisler.com/leccs.html
10. M. Vorbach, J. Becker: Reconfigurable Processor Architectures for Mobile Phones; Reconfigurable Architectures Workshop (RAW 2003), Nice, France, April, 200
11. J. Becker, M. Vorbach: Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC); IEEE Computer Society Annual Workshop on VLSI (WVLSI 2003), Tampa, Florida, USA, February, 2003