Interconnection Optimization for Dataflow-Architectures

Nico Moser, Carsten Gremzow, Matthias Menge
Technical University of Berlin, Faculty IV - Computer Science and Electrical Engineering,
Institute of Computer Engineering and Microelectronic

Abstract— **In this paper we present a dataflow processor architecture based on [1], which is driven by controlflow generated tokens. We will show the special properties of this architecture with regard to scalability, extensibility, and parallelism. In this context we outline the application scope and compare our approach with related work. Advantages and disadvantages will be discussed and we suggest solutions to solve the disadvantages. Finally an example of the implementation of this architecture will be given and we have a look at further developments.**
We believe the features of this basic approach predestines the architecture especially for embedded systems and system on chips.

I. Introduction

Dataflow processors boomed for the first time in the 1980s when they were analyzed for use in the supercomputing community (e.g. [4]). During that time the pure dataflow computing model with asynchronous instruction scheduling was the preferred approach. One problem of dataflow processors is the memory connection. Frequent memory access and the lack of registers and caches (compared to von Neumann architectures) make pure dataflow archtitectures vulnerable to bad memory performance. So the increasing processor-memory gap hits dataflow architectures harder then von Neumann based architectures. Even though this architecture got out of focus the original reason for this research (data parallelism) and the analyzed techniques didn't.
In today's state of the art microprocessors adapted techniques can be found primordially developed for dataflow architectures. The most important is multithreading whose origin can be found in dataflow processors [5]. Principal aim of this technique is to increase parallelism in processors based on von Neumann architectures.
A new application field arised for processor architectures in the 1990s, when embedded systems gained importance. As the first embedded processors were very slim designs (like ARM RISC-based processors) because of the special constraints for embedded systems, the present development of embedded processors points to more powerful architectures. Changed application scenarios (e.g. audio or video decoding [8]) with data intensive tasks require processors that can manage a lot more parallelism than early embedded processors.
Another trend in the processor industry is the integration of components on chiplevel previously on boardlevel. These system on chip design should be supported by the processor architecture.
In addition to that a more flexible architecture is necessary if an application specific instructionset processor should be realized. The functionality of these special embedded processors are adapted to the application running on them.
The last trend this architecure is influenced by is the realization of embedded processors using field programming gate arrays (soft intellectual property). The reconfiguration capabilities of FPGAs opened a wide spectrum of applications for a processor architecture.

In following section we will outline the overall register transfer level architecture of the synZEN processor. Section three will focus on a number of critical aspects during datapath / interconnect design and the respective realization for FPGAs as the target technology. After describing the current prototype implementation and its essential properties, we will draw conclusions from the current state of work and outline the main project objectives for the immediate future.

II. Architecture

In figure 1 the simplified registertransfer level structure of a controldriven dataflow processor is shown. The program counter, instruction memory, instruction register, and the branch processing unit are depicted on the left (some simplification were made [no memory managment unit, cache] for a better overview). The rest of the picture shows the function units, load and store units, and the registerfile resp. constant unit. All of these units are connected with the crossbar switch which is

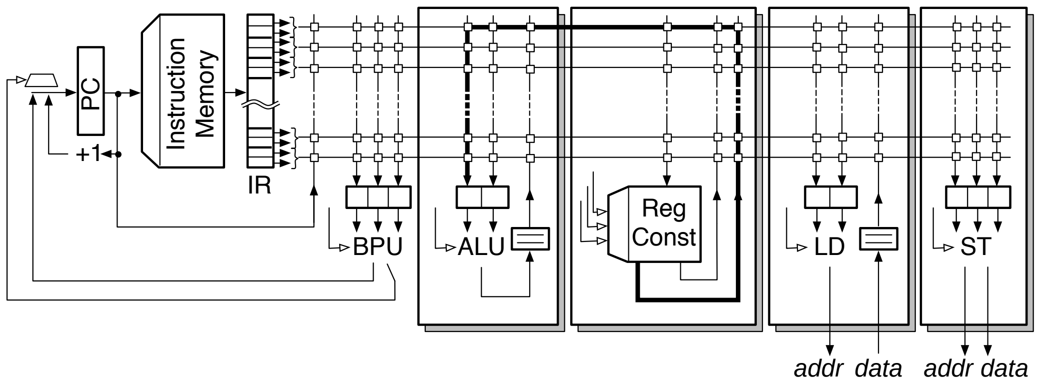


Fig. 1. Base architecture of the controlflow driven dataflow processor

shown above them and exchange there consumed and produced data across it. It is possible to use several instances of the units which is indicated by the shadow units. One particular feature of this architecture is the ring buffers on the data output of function units. They facilitate to distribute data temporarily with the positive side effect of shrink the registerfile and especially its port count. Common superscalar RISC and VLIW architectures tend to need much more ports than those planned for this controlflow driven dataflow architecture.
In one instruction several, parallel transport operations are encoded. These operations connect two units to make a dataexchange possible. Each of them contains one source address as well as one destination address for connecting the datapath between them and control information for the function units that belong to both addresses. Every port of every unit has a unique address. Processing one instruction begins with loading the instructon from the instruction memory into the instruction register. The transport operations are read and the adresses are decoded so that the datapath can be set. A bold line in figure 1 shows one such datamovement. The source address encoded in the considered transport operation is the output of the register file the destination address is the left input port of the arithmetical logical unit (ALU). The control information decoded in the transport operation in this case could be the operation of the ALU for the destination address and the register number for the source address.

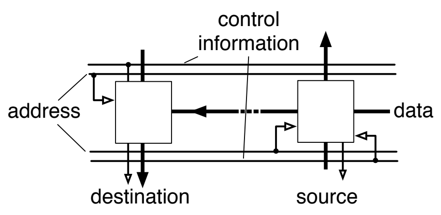
As soon as all operands are transfered to the destinations the function units generate the results. If there is one operand missing, the function unit stalls and waits until the operand appears in the operand register of that function unit. After generating the result it is written to the ring buffer, from where it can be used for further processing.

The transport triggered architecture [6] is based on a similar approach but uses different techniques. It requires special trigger register. It doesn't support chaining (section 3.1). In addition there are no ring buffers forcing the function units to stall in some cases.

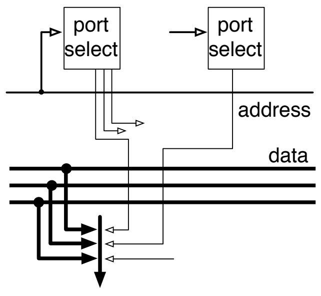
III. Interconnection Network

The interconnect network shown in figure 1 is a basic approach which guarantees maximum scalability and flexibility. It is very similar to a crossbar switch. In order to link an interconnection network like a crossbar switch a lot of hardware resources are necessary. The lack of tristate buffer in modern FPGA (the last FPGA series with tristate buffer from XILINX were the Virtex II Pro for which XILINX offered a crossbar switch IP) let this resourceproblem escalate.
Instead of a true crossbar switch two alternatives are presented. In figure 2a the simplified scheme of a peripheral approach is shown and in figure 2b centralized approach is depicted. Both have in common the extensive use of logic structures on an FPGA because the implementation of the busstructure is based on multiplexer.
The source and destination addresses are decoded directly on the connection switches if the peripheral approach is used. A multiplexer net[1] has to evaluate every databit and every port. In addition to that the same has to be done for the control information, too. For every transportoperation two units that select the port are needed. One for the readaddress and one for the writeaddress. There are some possibilities to reduce the

[1]there are more efficient bus infrastructures [7] but FPGA solutions are restricted to multiplexers



(a) peripheral



(b) centralized

Fig. 2. Network managment

cost of the interconnection network. We will point them out in the next sections. There are implicit ones that reduces the traffic and allow a more efficient use of the interconnection network and there are explicit ones that reduce the complexity by design decisions.

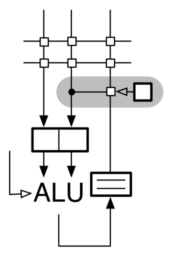
A. Chaining

One of the most important features of the in this paper specified processor architecture is the arbitrary connection between two function units. It is possible to chain function units in any order of choice of the user. This control driven chaining is also called *soft chaining*. It is initialized by an transport operation and uses one explicit bus of the interconnection network. After the operation this connection is closed and the bus is released to realise an other connection.
Beside this very common use there are other possibilities to connect function units. Figure 3a depicts *operand backcoupling*. This connection allows to lead back the result operand directly to one input operand register. Such functionality could be used by the accumulation operation. Another possibility is shown in figure 3b: *direct coupling*. This works similar to the previous mentioned method. Instead of directing the result to the function unit it is routed to an other function unit. In the shown example the result from a multiplier is routed to operand register of an ALU. It can be used to map a multiply accumulate (MAC) operation, which is often used in data intensive tasks like signal processing, on it. Both methods use connections with limited flexibility bypassing the network (grey highlighted ares in the figures). These connections allow only one destination for the operands. But they realize very common operation chains and they release the network, because they are not driven by transport operations directly but by side effects of previous operations. If an operand is forwarded to a function unit, control information is sent with it. This information can be used to activate these bypass connections by setting a status register or a sticky bit (suggested by the squares on the grey background). Because these connections can be established for several cycles, these methods are called *hard chaining*.
The third *hard chaining* technique we present is *result sharing* 3c.

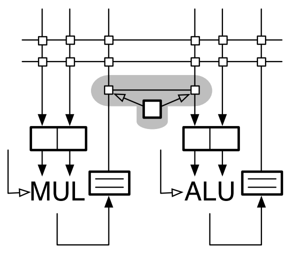
It allows the explicit reuse of a generated operand by several transition operations. Therefore a sticky bit evaluated by the ring buffer has to be set from the operation that generates the operand. Without these functionality a multiple reuse is only possible with an intermediate transition of the operand to the register file. That means a higher latency for direct following data operations and more transition operations on the network.
The last shown *hard chaining* method *constant storing* (figure 3d) uses no bypass connection but a sticky bit. For a continuous use of a constant in the same function unit the transport operation has to forward the constant to on operand register of the function unit and has to set the sticky bit (similar to the other methods).
Once set *hard chaining* connections can be annulled explicitly by a transport operation or implicitly by a side effect of one, if this connection is not required.

B. Reducing port count and connectivity

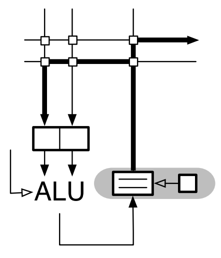
The number of ports per unit is often given by the functionality of the unit. Arithmetical and logical units realize dyadic operation: they process to operands and generate one operand. For the register file that means in data intensive applications more read than write accesses. In addition to that a lot of operands are directly consumed from ringbuffer by function units. That means in summary a minimum access to the register file, so that the number of ports can reduced compared to other architectures.
The network does not have to be fully connected. Reducing the connectivity means loosing flexibility in code generation but with alternating distribution of connections between busses there should be minimal trade offs in connectivity therefore large economization of hardware costs.



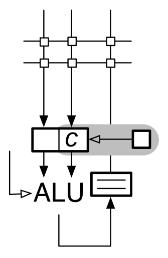
(a) Operand backcoupling



(b) Direct coupling

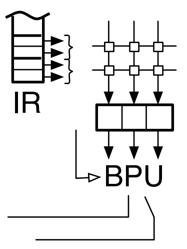


(c) Result sharing

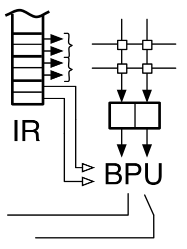


(d) Constant storing

Fig. 3. Hard chaining methods



(a) Transport operation controlled



(b) Instruction register controlled

Fig. 4. Alternative branch processing unit implementations

C. Branch processing unit

The functioning of the branch processing unit in figure 4a is very similar to the other function units. It consumes two operands for comparison with each other and one branch address. The BPU is controlled by the information sent with the transport operation. One branch operation needs three transport operations respectively three ports to the network.
An alternative approach is depicted in figure 4b. The BPU is controlled directly by an operation in the instruction register. Hence the capacity of the instruction register has to be extended beyond the demand for the transport operations, but the need for ports is reduced down to two. For statically known addresses this BPU needs one cycle like the initially presented BPU. For dynamically processed branches addresses it takes two cycles: the first to store the address and the second to process the branch.

D. Serialization

Another possibility of complexity reduction is datapath serialization of the interconnection network [3]. The structure of modern FPGA series allows to implement the necessary shift registers with up to 64 Bit in elementary structures (Configuration Logic Blocks) [9]. In case of serialization the network must be clocked higher than the function units.

IV. Protoype implementation

We started to design a prototype, where functionality is reduced to a coprocessor system. It is shown in figure 5.
Data access is memory management unit driven (not shown in figure 5) and instruction memory is controlled by the main processor.
Six busses are provided for data transportation. There are three function units (two ALUs[2] and one multiplier), one register file with 16 entries, one *instruction register controlled* BPU and one store unit. Because the three function units altogether consume six operands, we decided to implement three load units. For streaming tasks they are planned with burst mode.
All function units provide *constant storing* and *result sharing*. There are also two *direct coupling* connections planned: One between the multiplier and ALU1 and one between ALU1 and ALU2. The network isn't fully connected as you can see by the missing squares in the network matrix in figure 5. Grey filled squares belong to

[2]multiple function unit instances are depicted as grey shadows behind the first instance of function units with the same functionality

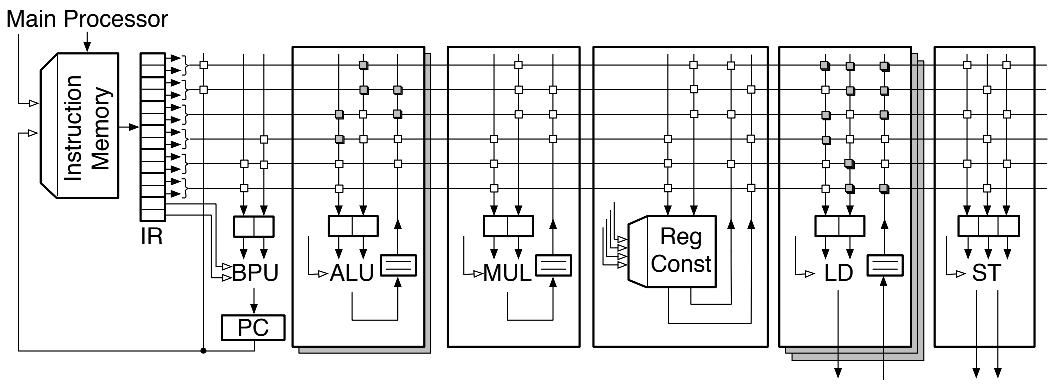


Fig. 5. Prototype design of synZEN

different instances. The number of connections between ports and busses conforms to frequency of use and the number instances of function units. For instance the BPU is less frequently used than the multiplier in dataflow intensive tasks and there are two instances of ALU compared to one of multiplier. Therefore the ports of BPU and ALU exhibit fewer connections than the ports belonging to the multiplier.

Altogether there are 19 destination and 8 source ports. To address all ports in one transport operation five bits for destination and three bits for source are nedded as shown in figure 6. Four additional bits per endpoint are provided for control information (e.g. 16 registers in the register file have to be addressed). Altogether 16 bit are needed for one transport operation, when nine bits are used for the destination and seven bits for the source part.

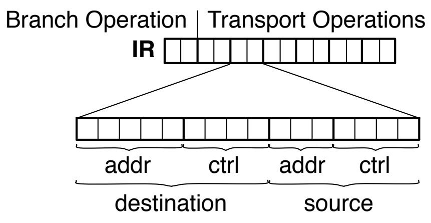


Fig. 6. Transport operation

V. Architectural Planning

Obviously the availability of at least one common compiler tool chain such as the GNU Compiler Collection (GCC) is mandatory for efficient, practical evaluation and testing of a prototype processor architecture. The LLVM infrastructure eases the task of GCC compiler backend generation (and also code optimization) a great deal. Using the LLVM bytecode as an intermediate program representation serves the purpose of constructing and programming the controlflow driven dataflow architecture. The corresponding tools are not directly integrated into the LLVM tool flow as e.g. compiler optimization passes or as a code generating backend.

Since the synZEN represents a scaleable processor architecture with variable amount of computation and interconnect resources, a code generating backend for the LLVM would need to know about the actual processor outline (be created each specific resource configuration) and be created on a dynamic basis. Instead, we have chosen to implement an independent, integrated code analysis and synthesis tool called LLILA-AS (Low Level Intermediate Language Analysis and Architecture Synthesis). It parses LLVM bytecode as well as assembly source and is responsible for processor layout planning and code generation.

a) Architecture Planning: Simple basic block level dataflow analysis and subsequent code scheduling (without resource constraints) is used to get a first impression of average and peak code parallelism. In turn, this information is used to guide unit and interconnect allocation. LLVM instruction scheduling is repeated under resource constraints to compute the set of all possible parallel operation constellations. As a first measure, the applied scheduling algorithms (based on a modified list scheduling algorithm) try to make best usage of the processor's capabilities of soft and hard chaining as well as result sharing. Aside from the above measures, cross basic block optimizations and speculative code execution have been implemented to enhance execution performance.

Applying ILP techniques, an optimal layout of each individual ALU instance is achieved.

VI. Conclusion

In this paper we presented a controlflow driven dataflow architecture. In our work we focus to adjust the datapathes between function units to FPGA structures and showed the possibilities of (hard) chaining methods in that context.
An register transfer level description of the synZEN in VHDL is soon to be completed. Target architecture is a XILINX Virtex 4 SX 35. We expect first test results within several weeks and believe that an implementation with more units than synZEN is achievable.
An other important aspect of further investigation is the automated code generation. With LLILA-AS we have a powerful framework for automated instruction coding and code emitting.

There are also some possibilities to refine or extend the basic concept. One thinkable extension is conditional transport operation (similar to predication in Intel Itanium). Also datapath serialization appears to a very promising task for the immediate future.

References

[1] M. Menge: *Zen 1 - Prozessor mit kontrollflussgesteuertem Datenfluss, Technical Report 21*, TU Berlin , 2003.

[2] H. Corporaal: *Microprocessor Architectures: From VLIW to TTA, Wiley, J*,1997.

[3] H. Sasaki, H. Maruyama, H. Kobayashi, T. Nakamura, H. Tsukioka, N. Shoji:*Reconfigurable Synchronized Dataflow Processor*, In: Design Automation Confer- ence, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific. (January 2000) 27–28

[4] K.-Q. Luc and K.B. Irani: *A High Performance Dataflow Processor for Multiprocessor Systems*, In: Workshop on the Future Trends of Distributed Computing Systems in the 1990s, 1988. Proceedings. (September 1988) 438–444

[5] Jurij Silc, Borut Robic and Theo Ungerer: *Asynchrony in parallel computing: From dataflow to multithreading, Technical Report*, Jozef Stefan Institute, 1997.

[6] H. Corporaal: *Design of Transport Triggered Architectures*, In: VLSI, 1994. 'Design Automation of High Performance VLSI Systems'. GLSV '94, Proceedings. (March 1994) 130–135

[7] R. Mäkelä, J. Takala and O. Vainio: *Analysis of Different bus Architectures for Transport Triggered Architectures*, In: Proc. 21st Norchip Conference. (November 2003) 56–59

[8] S. Sair and Youngsoo Kim: *Designing Real-Time H.264 Decoders with Dataflow Architectures*, In: Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. (September 2005) 291–296

[9] Xilinx: *Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs*, http://www.xilinx.com/bvdocs/appnotes/xapp465.pdf, Edition 1.1, (May 2005)