Nico Moser, Stefan Hauser, Carsten Gremzow

A hybrid transport/control operation triggered architecture

A Hybrid Transport/Control Operation Triggered Architecture

Nico Moser, TU Berlin, Department of Computer Engineering and Microelectronics, 10587 Berlin, Germany
Stefan Hauser, TU Berlin, Department of Computer Engineering and Microelectronics, 10587 Berlin, Germany
Carsten Gremzow, TU Berlin, Department of Computer Engineering and Microelectronics, 10587 Berlin, Germany

Abstract

We present an approach to a scalable and extensible processor architecture with inherent parallelism named *synZEN*. One aim was to create a synthesizable application specific processor which can be mapped to an FPGA. Besides architectural features like the interconnection network for flexible data transport and *synZEN* units with communication managing interface we give an overview of the programming model, show basic operation design and depict assembler notations to program these architecture. The paper closes with a brief toolchain overview and some synthesis results that support our design decisions.

1. Introduction

1.1. Motivation

One major challenge in electronic design automation is to generate hardware descriptions from software descriptions. Most commonly it is justified with hardware software codesign or with complexity of given specifications. Many researchers try to cope with this challenge with derived software languages [1] which often implies subsets of them or extensions (e.g. in form of libraries) to avoid widely known problems (e.g. pointer analysis) of hardware synthesis from software language descriptions. But those compromises lead to scattered developer landscape and loss of implementations from software engineers.

Therefore we develop a structure aware framework [2] based on runtime analysis which automatically partitions given software implementation and maps it to different targets.

We searched for an application specific processor as one of those targets that fulfills the following constraints: It should be flexible, extensible, scalable, and support for instruction level parallelism. Beside the architectural features it should be synthesizable to FPGAs. Our investigations lead us to the *synZEN* architecture described in this paper.

In the second part of this section we will give an overview of configurable architectures and a brief introduction of the presented architecture *synZEN*. In the following sections 2 and 3 main components of the architecture are presented in more detail. In section 4 a complete overview of synthesis and programming toolchain is given. A brief programming model overview can be found in section 5. In section 6 first results are presented. Finally the paper closes with a conclusion.

1.2. Background

There are several commercial soft IP processors for FPGAs on the market: ARM Cortex M1 [3], Altera NIOS [5] and Xilinx MicroBlaze [4]. Even though there were different intentions to offer them - for the first to make the ARM architecture available on even more development channels, the latter to offer a complete digital design system to developers using respective FPGAs - they have in common that they are RISC based and only have limited configuration capabilities (e.g. optional FPU) per core. Although ρ-VEX [8] is a very practical approach to realize a reconfigurable and extensible softcore VLIW processor, parallelism is limited by instruction width. Extensible parallelism is offered by [7]. This co-processor provides complex vectorization capabilities but cannot handle non SIMD-like parallelism. Transport triggered architectures (TTA) [6] are extensible and scale in parallelism because of their dataflow character. Drawbacks of this attribute are a lack of possibility to use more complex function units as well as inherent storing capabilities to relax scheduling issues.

1.3. Architecture

In figure 1 the top level structure of an exemplary *synZEN* implementation is shown. The central and most significant component is the interconnection network (ICN) which resembles of TTA networks. In fact there are a lot of similarities. Horizontal lines indicate busses. The number of busses are in accordance with the maximal number of concurrent data movements respective parallelism. Data transports are triggered by transport operations which select source and destination *synZEN* units. Per bus one transport operation is needed and current transport operations are stored in instruction register (IR) indicated at the left side of figure 1.

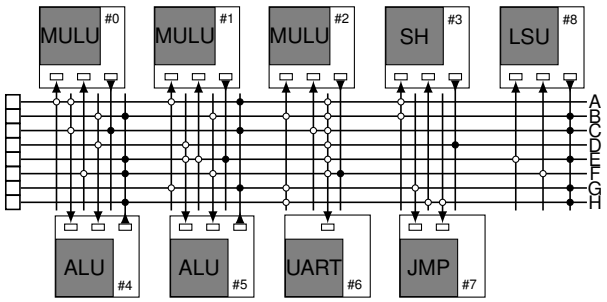


Figure 1: *synZEN* overview: There are three main components: on the left the instruction register, in the middle the connection network, above and below the network *synZEN* units for data operations.

Functionality of a particular implementation depends on functionality of *synZEN* units attached to ICN. Beside binary operations and branch units, load and store units are also possible as well as dedicated registerfiles. To extend functionality one only has to connect another *synZEN* unit to the ICN. As depicted in figure 1 sparse connection matrices are possible and should be preferred, because there is no necessity to use fully connected networks with prohibitive costs.
By now the datapath consists of a two stage pipeline with registers at input ports and output ports. Pipeline conflicts have to be resolved by program code with e.g. delay slots. Hardware solutions like interlocking and bypassing are not practicable. Especially the latter would lead to exploding hardware costs because of the vast number of connection paths through inter connection network. In spite of the cutbacks of pipelining it is necessary for control logic.

2. synZEN Unit

2.1. Structure and Datapath

As shown in figure 2 *synZEN* units can be divided into two main components.
First and emphasized in grey in the figure is the function unit part. For every operation in the application there has to be at least one *synZEN* unit with function unit part that can perform this operation. Function units which can perform several instructions are possible because operations contain control bits reserved for operation codes. Besides the need for interoperation with the interface part of the *synZEN* unit there are no other constraints for these function units.

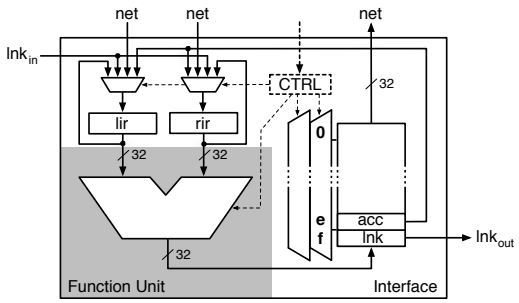


Figure 2: *synZEN* unit structure and datapath: A simplified version of synZEN unit datapath is depicted. Both main components are shown: function unit and interface. The latter manage communication with interconnection network (*net* ports) and dataflow control. Additionally it contains result registerfile with special purpose registers (*acc,lnk*).

Second there is the *synZEN* unit interface part. This component has to deal with several different tasks. On the communication side all datapaths of a *synZEN* unit from or to interconnection network (*net*) or other *synZEN* units (lnk_{in}, lnk_{out}) has to pass through the interface. Furthermore it contains the 16-entry output registerfile with one write port for the function unit results and one read port for access from interconnection network. From top level view the registerfiles can be used as distributed memory that scales with the number of connected *synZEN* units.
Two of the entries have a special meaning, but can be used as the other 14 in general manner as well:
The labeled entries *acc*/***Re*** and *lnk*/***Rf*** can be accessed separately from the general read port. While *acc* is directly connected to entries of same *synZEN* unit *lnk* can be connected to the entries of another *synZEN* unit.
At last the interface gets control data from operations and distributes them to the different components (highlighted by dashed lines):

- select signals for datasources at entry register of the function unit (*Data Transfer Mode*)
- pass on operation code to the function unit
- address read and write ports of the output registerfile

2.2. Data Transfer Mode

The easiest way to reduce the interconnection network complexity is to reduce data traffic and substitute a network with a sparse version. To use alternative data paths different modes were implemented.

2.2.1. Net Mode

It is the standard data transportation method. If there is no other optimized possibility to transport data from one unit to another a transport operation has to pass this data through the interconnection network.

2.2.2. ACC Mode

The behavior of a *synZEN* unit is switched to that of a 1-address-machine (respectively accumulator). Therefor the control logic in the *synZEN* unit interface has to set the entry of one of the inputs to the accumulator shadow register (*acc*).

2.2.3. Link Mode

It is possible to connect the link shadow register (*lnk*) of a unit to another at configuration time. With the help of that link data can bypass the interconnection network and consequently relax data traffic in the interconnection network. These links enable virtual grouping (or chaining) *synZEN* units and realize polyadic operation units.

2.2.4. Hold Mode

This mode is implemented for sequential operations with constant operands. The current content of the input register is held. It is only allowed to activate *hold mode* on one of the inputs.

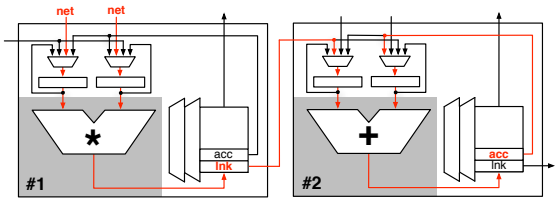


Figure 3: In this figure two chained *synZEN* units of different functionality are shown. The marked datapathes show possibilities given by the use of different data transfer modes.

In figure 3 the use of different modes is depicted. In this example data flow of a multiply accumulate operation

$$A = A + B * C$$

which is often used in signal processing applications is highlighted red. There are two *synZEN* units. The left performs a multiplication and the right one an addition/accumulation. Both inputs of the left *synZEN* unit are in *net mode*, while the left entry of unit #2 is in *link mode* and the right in *accumulate mode*. Instead of using four transport operations stressing the interconnection network only two operations are necessary.

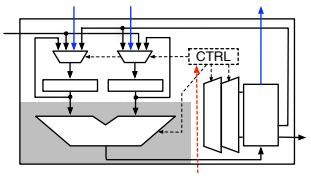
3. Network and Operations

With FPGAs as primary target architecture for *synZEN* processors the use of multiplexer based networks is inevitable. We investigate two different approaches for top level network architecture. First a unified approach similar to TTA where control and transport communications are combined in one network. Second we investigate a split approach where data transport and communication infrastructure are separated. A comparison of both approaches exceeds the scope of this paper. We present the split approach which is superior in hardware cost and programming model.

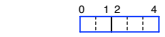
An empirical proven general cost function for data transport networks is shown below:

$$\begin{aligned} C_{net} &= \sum_{i=1}^{bus} \left(c_{bus} + n_{bus} \cdot 2^{\lceil \log_2 m_{bus}(i)+1 \rceil} \right) \\ &+ \sum_{i=1}^{in} \left(c_{in} + n_{in} \cdot 2^{\lceil \log_2 m_{in}(i)+1 \rceil} \right) \end{aligned}$$

The cost function C_{net} mainly depends on the three parameters of the *synZEN* processor: number of busses (bus) and unit inputs (in). Each partial cost function is structured very similar. To reduce costs you can either reduce the number of connection nodes (m_x) or constant respective variable costs (c_x, n_x). The former leads to sparse networks - one goal of reducing network traffic by use of alternative data transfer modes - the latter can be achieved by reducing the number of signals per port.



(a) *synZEN* unit with separate control path



(b) transport operation

(c) control operation

Figure 4: Figures show the structure of *synZEN* unit (4a) in split networks, transport operation (4b) that routes data through ICN and control operation (4c) which is needed by every *synZEN* unit.

Figure 4a depicts the control handling scheme in *synZEN* units using an infrastructure separated from ICN. There is no dependency between control signals and data transports.

Bit width of transport operation shown in figure 4b depends on address space of sources and destinations connected by busses in ICN. One transport operation is composed of source address and destination address. In the shown example first two bits TO[0:1] are source and last three bits TO[2:4] destination address. Most *synZEN* units represent dyadic functions. Therefore there are more destinations than sources which can be addressed by transport operation which leads to asymmetric segmentation of these operations.

The control operation (fig. 4c) conjuncts all control vectors needed by *synZEN* units. The different control bits encode from least to most significant bit: data transfer mode for left and right port (CO[0:1] and CO[2:3]), result register-file write address (CO[4:7]) and read address (CO[8:11]), and function unit operation code (CO[12:15]).

Beside obvious advantages like much smaller networks contrary to unified networks (control operations can be transferred peer to peer) other enhancements (like an ubiquitous *MOVE-immediate* instruction) can be realized.

4. Toolchain

The initial interconnection network (ICN) description can be generated in two independent ways. As shown in figure 5 you can either use java classes to program net architecture and generate XML descriptions (*JavaGen*) or use a tool that can visualize given net descriptions and even create or modify net descriptions with graphical user interface (*NetView*, *NetEdit*).

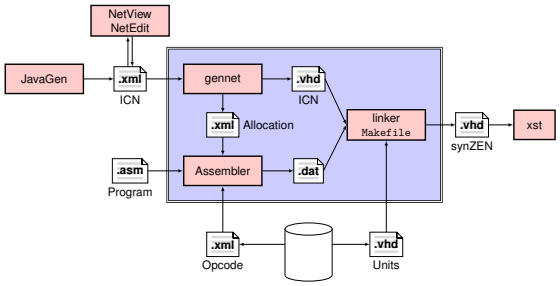


Figure 5: synZEN toolchain is shown: In the upper left you can see different tools to generate net architecture specifications. Together with assembler code and function units and operation codes belonging to it the net specifications are consumed by core tools (highlighted by blue background) to generate VHDL descriptions of synZEN instances.

The created XML-description is consumed by the Java-based tool *gennet*. It parses XML files and creates Java data structures to perform consistency checks and computes network features like control signal width or multiplexer and encoder amount. Control information represented as hash tables are handed out as XML file (*Allocation*). Net structure information is generated as VHDL descriptions. The VHDL descriptions contain the whole ICN and instances of every connected *synZEN* unit.
The *Assembler* obtains information, to generate machine code from assembly descriptions from two different sources. It gets control information from *gennet* as mentioned above. Operation codes for used function units are delivered by a database. It depends on net organization form (split vs. unified) whether one bit file is generated for each bus or additional one for each function unit.
The *linker* binds bit files, VHDL net description and function unit components (semi automatically), and creates VHDL description of whole *synZEN* architecture which is consumed by third party tool (in figure 5 xst from Xilinx). The depicted database in bottom of figure 5 contains function units and corresponding operation code. The only criterion function units have to satisfy is that they can be wrapped by the above-mentioned *synZEN* unit interface.

5. Programming Model

With every new *synZEN* instance resource allocation is changing, too. In addition to that there are several transport and control operations per cycle because of the massive parallelism of the architecture. Caused by these conditions there is an imperative need for an assembly language. We developed a context free assembly language and an assembler with a parser frontend generated by JavaCC.

Listing 1 shows a short extract from an assembly program computing fibonacci numbers. Transport operations are shown as well as control operations of *synZEN* units connected by the transport operations. The assembly code was generated for the *synZEN* instance shown in figure 1 which was generated especially for computation of fibonacci numbers.

```
1  { // transport operations
2    BUSA:SU5_O0,SU0_I0;
3    BUSG:SU5_O0,SU1_I0;
4    ...
5    // control operations
6    SU0:NET–HLD,R0,R0.MUL;
7    SU5:NOP,R1;
8    SU1:NET–HLD,R0,R0.MUL;
9    ...
10 }
```

Listing 1: Extract from an assembly program computing fibonacci numbers.

Curly braces in Line 1 and 10 mark scope for all parallel operations started at the same time step. Every operation is noted in a single line ended by semicolon. Transport and control operations are not strictly separated but differ by discriminative labels in front of the line (`BUSx`, `SUx`). These labels which are separated by colon from the rest of the operation contain a unique identifier to exactly select component. For every single component is only one operation during one period allowed.

The structure of transport operations is plain simple as the respective machine code. Beside the mentioned label there are descriptors for source address as well as destination address. The code in line 2 can be read as: a date is sent through bus A from first output port of *synZEN* unit #5 to first input port of *synZEN* unit #0.

Control operations are more complex in structure and with variable number of elements. The operations in line 6 and line 8 show exactly the same operation setting for *synZEN* unit #0 as well as #1: The first input register gets its data from ICN while value in the second input register is held. The output port reads its data from register 0 (`R0`) and the result of performed operation (`MUL`) is written to register 0 (`R0`).

The control operation in line 7 shows a special case: `No Operation (NOP)` has a parameter. Even if a unit (in this case #5) should not perform an operation there has to be the ability for transport operations to access data from result register to profit from distributed registerfiles - one main feature of this architecture. In the example shown in listing 1 even two transport operations access the output port of *synZEN* unit #5 which is synonymous to a multicast access.

6. Results

Synthesis is performed by the commercial tool Xilinx XST version 9.2i for Xilinx Spartan 3A DSP 3400 (xc3sd3400a-4-fg676) FPGAs. Tests in this section were run on a Intel Xeon 3.0 GHz with 4 MB Cache and 32 GB main memory running ubuntu linux with kernel version 2.6.24.

#	#unit	#bus	dst.	src.	slices	longest path	synth. time
Unified Network							
1	8	16	4	2	6%	13,4ns	0m37.684s
2			8	4	12%	15,0ns	1m34.687s
3			11	6	19%	18,8ns	3m40.005s
4	12	24	6	3	14%	12,6ns	2m12.813s
5			8	4	17%	15,3ns	2m10.118s
6			12	6	29%	19,2ns	8m53.159s
Split Network							
1	8	16	4	2	5%	11,4ns	0m26.821s
2			8	4	9%	12,8ns	0m54.085s
3			11	6	16%	18,9ns	2m23.957s
4	12	24	6	3	11%	12,4ns	1m54.780s
5			8	4	14%	12,8ns	1m20.329s
6			12	6	24%	20,1ns	5m05.724s

Table 1: Synthesis results of generic communication network variants with different number of connected *synZEN* units, busses, as well as connected sources and destinations.

We created a set of generic test cases with slightly different parameters. Results are shown in table 1. The parameters we changed were numbers of *synZEN* units (*#unit*) and busses (*#bus*) to vary parallelism combined with different numbers of destination ports (*dst.*) and source ports (*src.*) to vary network connectivity. These six test cases were investigated as unified network implementation (top half) as well as split network implementation (bottom half).
At first we can observe that connectivity costs more than parallelism. Comparing results *3* and *5* of both types from the table shows clearly that the test case with less function units and busses but more addressable destinations and sources costs more area, synthesis time and frequency than the compared one. And these results are independent from the kind of network.

7. Conclusion

We presented in this paper a new processor architecture which combines features of several established architectures like transport triggered architectures (flexible interconnection network) and RISC (registerfile). We made progress in network design and could show improvements with regard to costs. Furthermore we could introduce *synZEN* unit techniques to relieve data traffic load on ICN. Additionally we showed the *synZEN* generation toolchain. With help of [2] we will analyze applications to specify *synZEN* features in future work.

References

[1] S. Edwards, *The challenges of hardware synthesis from C-like languages*,Design, Automation and Test in Europe, 2005. Proceedings, pp. 66-67, vol. 1, 2005

[2] C. Gremzow, *Quantitative Global Dataflow Analysis on Virtual Instruction Set Simulators for Hardware/-Software Co-Design*, 26th IEEE International Conference on Computer Design, Lake Tahoe, Kalifornien, USA, 2008

[3] ARM Cortex-M1 - ARM Processor,
http://www.arm.com/products/CPUs/ARM_Cortex-M1.html,
last visit: August 2009

[4] MicroBlaze Soft Processor Core,
http://www.xilinx.com/tools/microblaze.htm,
last visit: August 2009

[5] Embedded Processor,
http://www.altera.com/products/ip/processors/nios2/ni2-index.html,
last visit: August 2009

[6] H. Corporaal, *Microprocessor Architectures from VLIW to TTA*, John Wiley & Sons, Ltd., 1998.

[7] C. Kozyrakis, D. Judd, J. Gebis, S. Wlliams, D. Patterson, and K. Yelick, *HardwareCompiler Codevelopment for an Embedded Media Processor*, Proceedings of the IEEE, pp. 1694 - 1709, vol. 89 (11), 2001

[8] S. Wong,T. van As, and G. Brown, *ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor*, Field-Programmable Technology, 2008. FPT 2008. International Conference on, pp. 369 - 372, 2008

[9] I. Janssen, *Enhancing the Move Framework. Endianness Port and Immediates Handling.*, Master Thesis, May 2001.