

AN EXPANDABLE PROTOCOL PROCESSOR ARCHITECTURE

Seppo Virtanen¹ and Johan Lilius²

¹ Turku Centre for Computer Science (TUCS), University of Turku (Lab. of Electronics and Information Technology)
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland, E-mail seppo.virtanen@utu.fi

² Turku Centre for Computer Science (TUCS), Åbo Akademi University (Department of Computer Science)

BACKGROUND

The design of modern data processing hardware for deeply embedded systems like cellular telephones is facing great demands because of the decreasing times-to-market and the increasing demands for performance. This is especially true in third generation cellular phones in which the convergence of traditional telephony and modern multimedia applications makes the emergence of small and efficient hand-held communication devices possible. In order to meet increased performance requirements and to achieve shorter development times new system design technologies like System-on-Chip and ASIP have arisen.

In **System-on-Chip (SoC)** design the objective is to reduce the number of microchips needed to build a certain system. As the name suggests, the ultimate goal is to integrate an entire system to one microchip. An SoC is designed from pre-designed and reusable intellectual property (IP) blocks. An IP block could be eg. a silicon layout of a multiplier unit. The SoC system designer obtains IP blocks from in-house IP block designers or possibly from a third-party IP block provider, and then combines and possibly alters the blocks to reach a System-on-Chip that matches the original specification. An SoC device can be any kind of static or programmable microchip.

The hardware architecture and the instruction set of an **application-specific instruction-set processor (ASIP)** are designed to perform certain specific tasks as efficiently as possible. Because ASIP's are targeted mainly at embedded applications, processor simplicity is a major design goal. In typical ASIP design flow the application software is profiled at assembler language level to detect instruction sequences that occur often and that could be implemented in hardware to improve performance. The typical size of such a detected instruction sequence is 2-3 instructions [5, 6].

In specific application areas (eg. protocol processing) the ability to implement larger functional blocks in hardware might be advantageous. It is not clear how the existing ASIP design approaches scale up to such functional blocks.

We have found that in control oriented protocol processing there are certain recurring protocol processing operations that are in practice similar in all communications protocols [1, 2, 3]. By utilizing the knowledge of such recurring operations it is possible to form programmable functional units that are able to perform the required processing tasks regardless of the protocol at hand. These functional units, or IP blocks, can then be taken advantage of in designing an ASIP specification and design environment.

The objective in our research project, **TACO** (Tools for Application-Specific HW/SW Codesign), is to design and implement an ASIP design environment that is optimized for the specification and synthesis of communications protocol processors. By concentrating on protocol processing we expect to incorporate the usage of IP blocks in our ASIP design environment. This approach generates ASIP assembler code where large code blocks have

optimized execution in hardware (eg. a CRC calculation block), and no application-specific code or code block identification is necessary during the synthesis of the processor.

The TACO environment is intended to provide the necessary tools and functionality for generating an ASIP, its instruction set and its application program code from a high level protocol description using our protocol processing IP blocks.

ARCHITECTURE

After studying different microprocessor architectures we have come to the conclusion that TTA (Transport Triggered Architecture) [4] could serve as the base architecture from which the ASIP's are generated. TTA is a modified VLIW architecture, which does not feature logic for execution optimization (run-time instruction reordering to improve concurrent use of functional units etc.). Instead, TTA processors rely on the program compiler to perform instruction scheduling. As the name implies, in TTA processors the data transports are programmed and they trigger operations (traditionally operations are programmed and they trigger transports).

A TTA processor is formed of functional units that each perform particular tasks, eg. one unit takes care of logical operations (AND, OR, ...) and another one handles comparing data (less than, equal to, ...). Each functional unit has one or more operand registers, trigger registers and result registers. An operation is triggered when data is transported to a trigger register. The functional units are connected via an interconnection network of buses, controlled by an interconnection network control unit. The maximum number of instructions (ie. data transports) that can be carried out in one clock cycle is equivalent to the number of buses in the interconnection network. The advantage TTA's provide to our research is their modularity and scalability in hardware configuration. Functional units can be added to an architecture or they can be refined and changed as long as they provide the same interface to the interconnection network. The same holds naturally for the interconnection network: it can be tuned as long as the interface remains the same. According to [4], this modularity allows the hardware design to be automated. In the following discussion we will call the functional units IP (intellectual property) blocks.

Figure 1 shows a base architecture for a protocol processing TTA processor with some of the IP blocks identified in [2], and some generic logical units. All of these blocks are fairly simple and well-known solutions for reasonably fast algorithms, gate-level schematics or even silicon layouts exist and can be either obtained without charge or purchased from a provider. Finding and optimizing algorithms or gate-level implementations to perform these functions is beyond the scope of this document that focuses on system-level specification and simulation. Analyses of the requirements and implementations of some of the IP blocks as well as the motivation to use exactly these IP blocks can be found in eg. [1], [2], [3], [4] and [7].

Because of the relative simplicity of the IP blocks, this architecture is very scalable: with quite little design effort and effect on physical processor size more IP blocks can be added into the architecture in the design flow. If for example the targeted application needs several counters/timers, they can be incorporated into the design. As another example, an application may require such high speed bitstring matching that two matcher units are needed. When there are more than one of the same kind of IP blocks in the processor, the blocks are identified with an ordinal number (eg. Matcher1 has registers OM1, TM1 and RM1 while Matcher 2 has registers OM2, TM2 and RM2, see figure 1).

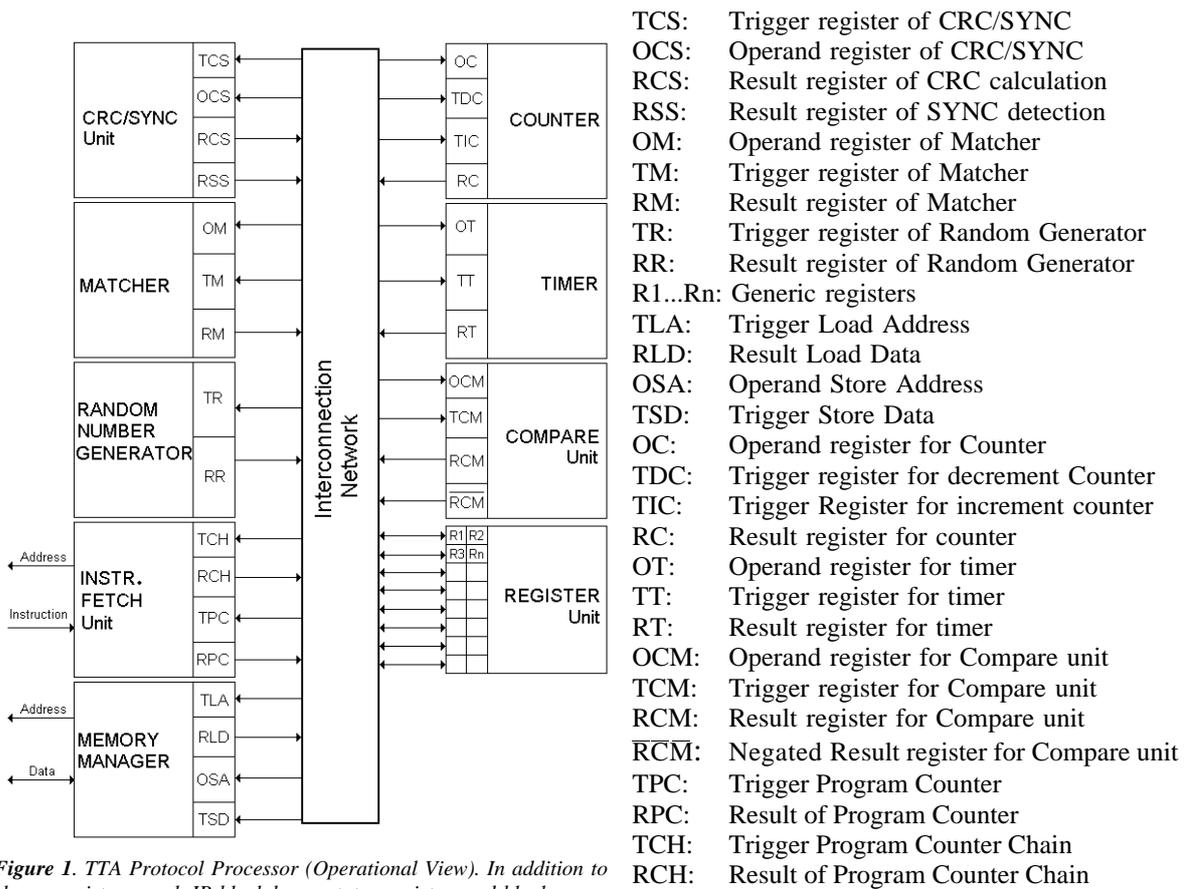


Figure 1. TTA Protocol Processor (Operational View). In addition to shown registers, each IP block has a status register, and blocks communicating with external memory have control registers.

IP BLOCKS

For all the IP blocks of figure 1 we know that there exist solutions that are able to perform the required task in minimal clock cycles. For example, from [1] we know that a solution for comparing two bitstrings and getting a result in one clock cycle can be found, and from [7] we know that we can calculate a 32-bit CRC for a 32-bit input data word in one cycle. As mentioned earlier, depending on the application requirements, there can be more than one of any kind of these IP blocks in our designs to improve overall system performance and reduce bottlenecks.

SIMULATING THE PROCESSOR

The first version of a simulator for the protocol processor presented in Figure 1 was made using GNU C++ tools and the SystemC [8] simulation extensions to them. SystemC is intended for creating executable specifications and allows the use of system clocks, signals etc. in C++ code. The first version of our simulator features the IP blocks and their respective registers shown in figure 1, as well as a controller unit which manages the data transports between the IP blocks. In this version of the simulator the controller unit also holds the application to be run, coded in C++.

As the algorithms that perform the IP block functions are well known and hardware (gate-level or schematic level) specifications with excellent performance characteristics exist for them, the emphasis in the simulator is on determining the control structure and the required number of buses in the interconnection network to ensure maximal protocol processing throughput for incoming data.

As the first simulated application we implemented an algorithm that examines incoming ATM cells to find out if a cell is a regular user cell, an empty cell or an AIS operation and maintenance (OAM) cell. The application is executed in our simulator as 32-bit data transports between the IP blocks. The AIS algorithm was chosen as an example since another approach to its processing was presented by Jantsch et al in [1]. In this example, AIS cells cause the system to enter AIS state, in which cells are sent from the outbuffer and an AIS cell is inserted after every 1024 regular cells. If a regular cell appears in the inbuffer when the system is in AIS mode, the system returns to normal processing. If an empty cell appears in the inbuffer, it is discarded. If there are no cells to be sent, empty cells are transmitted.

An ATM cell consists of a 5 byte (40 bit) header and a 48 byte payload. The last 8 bits of the header contain HEC (Header Error Check), which is a CRC-8 checksum of the header. First the HEC is checked to determine whether the incoming data is the cell header of a new cell. If it is, then the cell type must be determined. The information of the cell type is encoded into the PTI (Payload Type Identifier) field located in bits 29..31 of the header. To determine the type of an incoming ATM cell we need to compare these three bits of the incoming cell to bit patterns in memory representing different types of ATM cells. If the match result indicates that an incoming cell is an OAM cell, then the first 8 bits of the cell payload must be analyzed to determine if the cell is an AIS cell (requires another matching operation). We constructed our simulator to simulate a synchronous 32-bit processor with one matcher, which means that two 32-bit data words need to be matched against certain bit patterns to determine if the cell is an AIS cell. This takes at least two clock cycles. The HEC analysis also requires two clock cycles, since the HEC and the rest of the header do not fit into a single 32-bit data word and for this reason the HEC unit needs to wait for two data words before determining the result.

If the last matching reveals that the cell is an AIS cell, then an AIS cell is sent, the counter unit is set to 1024 and the system remains in AIS state until the header and HEC of a regular ATM cell arrive at the input.

Our simulations show that the required clock cycles for eg. AIS operation can be greatly reduced by using the IP blocks of the processor in parallel. For example, while datawords are analyzed to determine if they form a header, the same data can already be matched against the bitstrings that determine if the header is of type OAM. If the HEC result indicates that the data words do not form a valid header, then the match results can be discarded. On the other hand, if the data forms a valid header, then the header type is already known.

CONCLUSIONS AND FUTURE WORK

We presented a modular and scalable microprocessor architecture for communications protocol processing applications. Our IP block based design specifies a microprocessor based on the Transport Triggered Architecture (TTA). We have successfully simulated the ATM AIS processing on our architecture with a simulator written in C++/SystemC. SystemC proved to be a worthy design tool, since it took about two weeks to get familiar with it, write the code for the IP blocks and then write a controller and the first protocol processing application for the simulator.

The next task in our research is to build a more exact simulator, which will have, among other things, a more detailed specification of the interconnection network. In the next version of the simulator, the interconnection network controller will no longer contain the application code, but the application code will be imported as pure binary data or possibly assembler.

There are VHDL models for the IP blocks in the works, and we aim to co-use them and SystemC for automatic VHDL generation of processors for particular applications.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the discussions concerning protocol processors with professor **Jouni Isoaho** (laboratory of electronics and information technology, University of Turku) and the discussions concerning VHDL models of the IP blocks with research associate **Tomi Westerlund** (laboratory of electronics and information technology, University of Turku).

REFERENCES

- [1] A. Jantsch, J. Öberg and A. Hemani, "Is there a Niche for a General Protocol Processor Core?", Proceedings of the 16th IEEE NORCHIP Conference, pp. 93-100, Lund, Sweden, November 1998.
- [2] Virtanen, Seppo, "On Communications Protocols and their Characteristics Relevant to Designing Protocol Processing Hardware", TUCS Technical Report 305, Turku Centre for Computer Science, Turku, Finland, 1999.
- [3] Virtanen, Seppo, Isoaho, Jouni, Westerlund, Tomi and Lilius, Johan, "A Programmable General Protocol Processor - a Proposal for an Expandable Architecture", in URSI/IEEE XXIV Convention on Radio Science, Turku, Finland, October 1999. Abstract in Informo No. 181, Tuorla Observatory Reports, pp. 112-113, Turku, Finland, October 1999.
- [4] Corporaal, Henk, "Microprocessor Architectures - from VLIW to TTA". John Wiley & Sons publishing, Inc, USA, 1997.
- [5] J. Van Praet, G. Goossens, D. Lanneer, H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs", Proceedings of the Seventh International Symposium on High-Level Synthesis, pp. 11-16, Niagara-on-the-lake, Canada, May 1994.
- [6] A. Both, B. Biermann, R. Lerch, Y. Manoli, and K. Sievert, "Hardware-Software-Codesign of Application Specific Microcontrollers with the ASM Environment", Proceedings of the Conference on European Design Automation Conference, pp. 72-76, Grenoble, France, September 1994.
- [7] R. Hobson and K. Cheung, "A High-Performance CMOS 32-Bit Parallel CRC Engine", IEEE Journal of Solid-State Circuits, Vol. 34, No. 2, February 1999.
- [8] SystemC: <http://www.systemc.org>