In Proceedings of the 18th IEEE NorChip conference, 6-7 November 2000, Turku, Finland

A Processor Architecture for the TACO Protocol Processor Development Framework

Seppo Virtanen[1,2], Johan Lilius[1,3] and Tomi Westerlund[2]

[1] Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland,
Phone +358-2-215 4204, Fax +358-2-241 0154
[2] University of Turku, Department of Applied Physics, Laboratory of Electronics and Information Technology
[3] Åbo Akademi University, Department of Computer Science
Email seppo.virtanen@utu.fi, johan.lilius@abo.fi, tomi.westerlund@utu.fi

Abstract *-- Increasing performance expectations and requirements for modern communications and networking devices call for novel solutions in hardware design. To achieve high performance without losing flexibility, several authors have identified the need for a special family of processors dedicated to protocol processing. In previous work we have identified a number of common protocol processing operations. In this paper we present a modular and scalable protocol processor architecture that has these operations as primitive instructions. We also discuss a simulator framework and VHDL models for the architecture. As a case study we present simulation results for ATM AIS cell processing.*

1. INTRODUCTION

The design of modern networking hardware is facing new challenges because of decreasing time to market and increasing demands on performance. This is especially true in third generation mobile networks, where the convergence of traditional telephony, modern multimedia and the internet will be used to provide totally new services to customers. In order to meet these 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. A SoC is designed from pre-designed and reusable intellectual property (IP) blocks. An IP block could be e.g. 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 SoC that matches the original specification. A SoC device can be any kind of static or programmable microchip. A SoC trades time-to-market and flexibility for performance and price.

An alternative approach is to try to take a general purpose processor (GPP) architecture and increase its performance by moving often executed instruction sequences into special hardware units. Such an **Application Specific Instruction-set Processor** (ASIP) is 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 a 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 [1, 9].

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 [5, 10, 11] and that these operations typically do not fit into 2-3 instructions. 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. It is not clear how the existing ASIP design approaches scale up to such functional units. There is therefore a need for a new approach to programmable protocol processor design.

The objective in our research project, **TACO**, is to design and implement an ASIP design environment that is optimized for the specification and synthesis of communications protocol processors. A core component of the methodology is to exploit functional blocks in design of the processor. The novelty of this approach is that no application-specific code or code block identification is necessary during the synthesis of the processor. This approach also generates very compact ASIP assembler code, because the functionality that would normally be implemented in software is now replaced by a single assembler instruction. 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.

In this paper we present a proposal for a Programmable Protocol Processor (PPP) architecture in the TACO project that has the following two properties:

1. It is flexible and scalable; it is easy to add new instructions to the processor and increase its performance.
2. Its architecture is focused on moving data; a crucial component in protocols.

2. THE TACO PROTOCOL PROCESSOR

The main function of a telecom protocol is to reliably transfer data from the sender to the receiver. Interleaved within this data transfer task are different signaling tasks like control flow, connection setup or teardown etc. However, in a well designed protocol these signaling activities should occur seldom enough not to incur any extra penalty on the performance of the data transfer. In selecting an architecture for the TACO processors an important criteria was therefore the ability to have efficient data transfers.

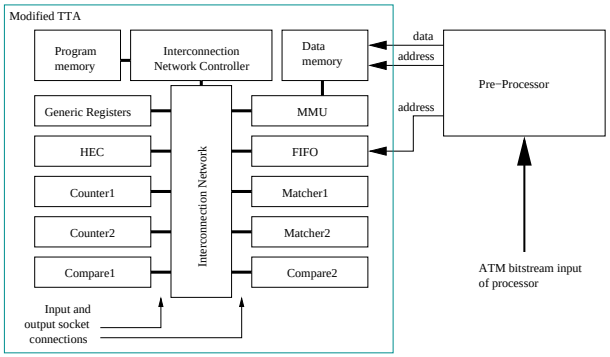


***Figure 1**. A TACO protocol processor for ATM.*

2.1. Architecture

The TACO processor architecture is a slightly modified transport triggered architecture (TTA), a novel processor architecture proposed by H. Corporaal [2]. In TTA processors data transports are programmed and they trigger operations (traditionally operations are programmed and they trigger transports). A TTA processor is formed of functional units (FU's) that communicate via an interconnection network of data buses, controlled by an interconnection network

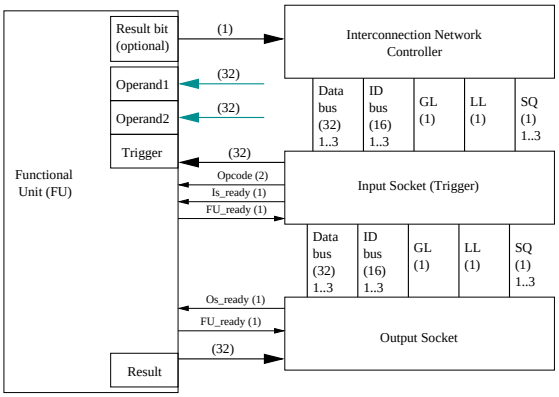


***Figure 2**. Connectivity between FU's, sockets and the interconnection network. GL = global lock, LL = local lock request, SQ = squash.*

controller unit, as seen in figure 1. The connection between a functional unit and the interconnection network is managed by input and output socket units as shown in figure 2. 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.

TTA's are in essence one instruction processors, the only instruction being move data. Thus, the instruction word of a TTA processor consists mostly of source and destination addresses of sockets called socket ID's. The socket ID's are transported on ID buses from the interconnection network controller. There are as many ID buses as there are data buses in the interconnection network. Upon finding its socket ID on one of the ID buses, a socket opens the connection between an FU and the corresponding bus on the interconnection network. The maximum number of instructions (i.e. data transports) that can be carried out in one clock cycle is equivalent to the number of data buses in the interconnection network.

The benefit of the TTA architecture is its modularity and scalability. Functional units can be added to the architecture or they can be refined and changed as long as they provide the same interface to the sockets connecting them to the interconnection network. The same holds naturally for the interconnection network. According to [2], this modularity allows the hardware design to be automated.

In previous work [10] we have analyzed a number of commonly used communications protocols and identified a number of typical protocol processing elements that are common to the protocols: bitstring matching, integer comparison, checksum calculation (especially CRC) and indexing (counters). Wireless and timing-critical protocols also need capabilities for maintaining timers and generating random values. All of these protocol processing tasks are distinct enough to be considered for implementation as FU's.

Some protocols also benefit from protocol data unit (PDU) pre-processing (the pre-processor in figure 1). The tasks performed in pre-processing are protocol dependent and may include synchronization to the incoming bitstream, data integrity verification (by means of performing a protocol-dependent error check on incoming data) and incoming PDU storage into the processor's data memory (using DMA). The memory addresses of first data words of PDU headers can be stored into a FIFO to provide quick access to the data that requires processing. The pre-processor unit in our architecture is optional and protocol dependent.

2.2. The Processor Simulator

To test the fundamental assumptions of the TACO framework we are at the moment prototyping a processor for processing ATM cells. It features three 32-bit buses in the interconnection network as shown in figure 2. This makes it possible to have three parallel data transports in one machine cycle. The control signals in figure 2 are typical for TTA and thus are not explained in detail in this paper. The reader is referred to [2] for a detailed discussion on this topic.

A functional view of the simulated processor is shown in figure 1. All of the functional units in the processor are fairly simple, and well-known solutions for reasonably fast algorithms, gate-level schematics or even silicon layouts for all of them exist. Analyses of the requirements and implementations of some of the FU's as well as the motivation to use exactly these units can be found in e.g. [2], [3], [5], [10] and [11].

The simulator is written in SystemC [7]. SystemC is a C++ [6] application framework for simulating hardware. It provides a set of classes for describing common entities in hardware design e.g. signals, clocks etc. SystemC is distributed under an open license and is

supported by several of the major EDA companies. SystemC tries to address the verification bottleneck in ASIC design. Instead of going from a textual specification to a VHDL design directly the system specification is expressed as an executable SystemC specification. Because SystemC is based on C++ it is possible to exploit all the powerful structuring capabilities of C++, like inheritance, in the designs. We have used inheritance and object oriented (OO) concepts extensively in the simulator. The class hierarchy of the simulator is given in Figure 3. OO techniques ensure that similar objects have compatible external interfaces (e.g. FU's have compatible registers). They also make easy addition of new objects of the same kind into the system possible (e.g. two matchers). Functional units that are not needed for a certain protocol processing application can be left out of the simulation (e.g. in the prototype processor simulation, objects from classes Timer and RandomGen are not needed (figure 3)).

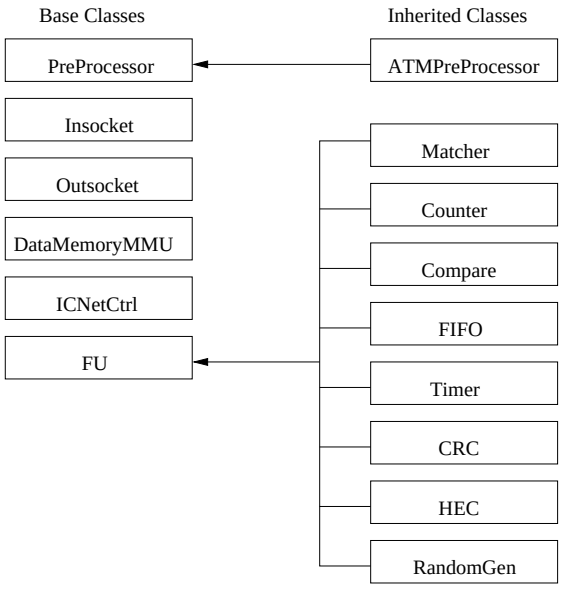


***Figure 3**. Class hierarchy of the TACO simulation framework.*

As the algorithms needed for internal FU functionality are well known and hardware (gate-level or schematic level) specifications with excellent performance characteristics exist for them, the emphasis in the simulator is to define the processor control structure, internal signaling and the number of FU's in a way that ensures maximal protocol processing throughput for a certain application.

2.3. VHDL Models of Processor Components

To make it possible to synthesize processors and obtain initial physical estimates we are also working on VHDL models of TACO processors. Presently we have a model for the processor shown in figure 1. The VHDL models for processor components are described by using both structural and behavioral VHDL, depending on the component. FU's are described as hybrid models: the common operations that are the same from one FU to another (e.g. interfaces to sockets) are modeled as structural VHDL, and the parts that are specific to each kind of FU (e.g. calculations) are modeled as behavioral VHDL. All processor specific functions, types and constants are located in a package.

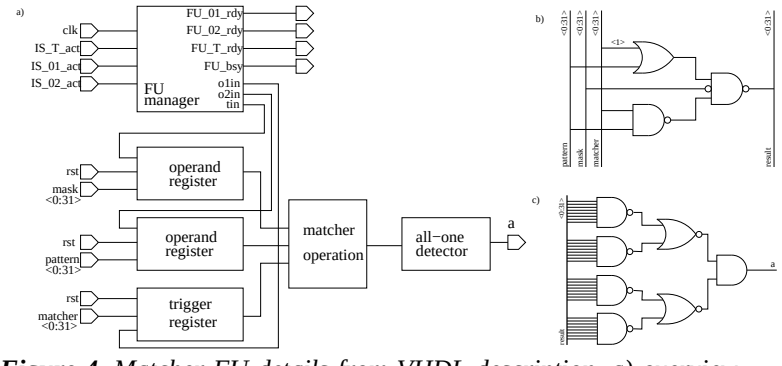


***Figure 4**. Matcher FU details from VHDL description. a) overview, b) one bit match operation, c) all-one detector.*

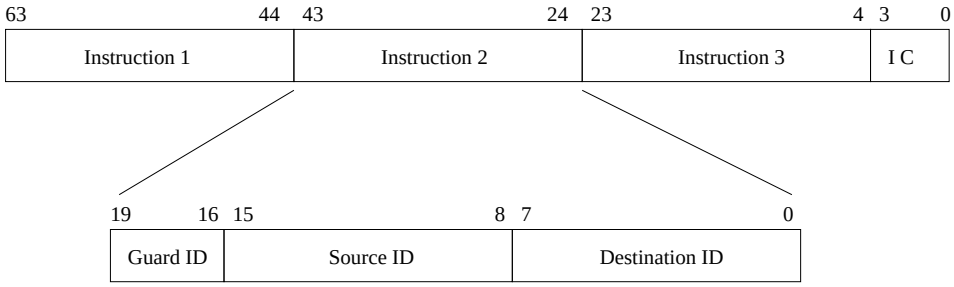


Figure 5*. TACO instruction format for a processor with three data buses. The 64 bit instruction is divided into three 20 bit bus instructions. The four IC bits are used for long/short immediate generation.*

Because of the nature of the TACO processors and the use of structural VHDL coding the reusability level of component and subcomponent models is high. A duplicate of an FU is created by copying the component instances related to the FU. Only the FU identification information (source and destination ID information) needs to be modified. Common operations inside FU models can be reused and some flexibility is achieved using parameterized constants. As an example, a matcher FU (figure 4a) consists of three registers, the FU manager and the match operation itself. Registers and the FU manager are generic subcomponents and are used by all FU's. These subcomponents are included in other FU models using component instances. The match operation is coded using behavioral VHDL because only the matcher FU needs this operation. In figure 4b is a schematic of the match operation for one bit and in 4c a schematic for a 32 bit all-one detector.

3. PROGRAMMING THE PROCESSOR

The prototype processor has three data buses and is programmed with 64-bit instructions as shown in figure 5. This leaves 20 bits of instructions for each bus. The last four bits are used for controlling the generation of short and long immediate values. TTA processors have in essence only one instruction, move data. For this reason, the 20 instruction bits for each bus only contain a source socket ID and a destination socket ID accompanied with four guard bits (i.e. Guard ID, see figure 5) used for conditional execution. The 64 bit instruction is decoded and the socket ID's distributed by the interconnection network controller.

As presented in figure 2, some FU's have a result bit connected directly to the interconnection network controller. With this structure it is possible to directly use the result from an FU in guard bit evaluation. This feature is especially useful in matchers, compare units and error check units. The interconnection network looks for a certain combination of

Guard ID	Boolean	Assembler	Guard ID	Boolean	Assembler
0000 (0)	a	a:src -> dst	1000 (8)	e	e:src -> dst
0001 (1)	¬a	!a:src -> dst	1001 (9)	f	f:src -> dst
0010 (2)	b	b:src -> dst	1010 (10)	a ∧ b	a.b:src -> dst
0011 (3)	¬b	!b:src -> dst	1011 (11)	a ∧ (¬b)	a.!b:src -> dst
0100 (4)	c	c:src -> dst	1100 (12)	(¬a) ∧ b	!a.b:src -> dst
0101 (5)	¬c	!c:src -> dst	1101 (13)	(¬a) ∧ (¬b)	!a.!b:src -> dst
0110 (6)	d	d:src -> dst	1110 (14)	(¬a) ∧ (¬c)	!a.!c:src -> dst
0111 (7)	¬d	!d:src -> dst	1111 (15)	TRUE	src -> dst

Table 1. *Assembler commands, Guard ID's and their corresponding Boolean expressions in the TACO processor being prototyped (two matchers, two compare units and two counters). a = result from Matcher 1, b = result from Matcher 2, c = result from Compare 1, d = result from Compare 2, e = Counter 1 has reached zero, f = Counter 2 has reached zero.*

the result bits specified by the Guard ID for each bus. An example of guard bits and corresponding result bit signal conditions is shown in table 1. If the result bit signals do not satisfy the condition specified by the Guard ID, a squash signal is sent to all sockets connected to the corresponding data bus to terminate the specified data move. If the Guard ID for a bus is 1111, then no evaluation is needed and the data move is carried out regardless of the result bit signal values. This technique is similar to branch predication used in advanced processor architectures like the IA-64 [8].

To exploit the increased parallelism offered by three concurrent data transports in TACO processors the assembler code of any application must be organized in an optimal manner. This optimization will be carried out by a compiler that takes in program code written in a high-level language and yields optimized assembler code. As an example we will consider the algorithm in figure 6. Clearly the processing of this algorithm would require four cycles to complete in traditional sequential programming (assuming each operation takes one cycle and there is no instruction pipeline).

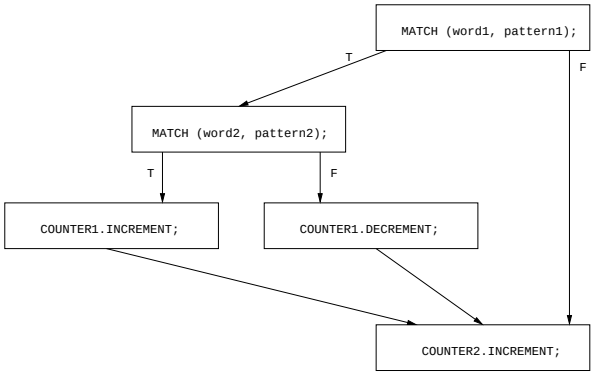


Figure 6*. Block diagram of the application used for a programming example.*

If we consider a TACO processor with two counter FU's and two matcher FU's (already holding the match patterns *pattern1* and *pattern2*), we have the following assembler code for this particular TACO processor (each line represents one 64 bit TACO instruction, i.e. is executed in one cycle):

```
word1->TM1; word2->TM2; incr->TC2;   // TM1, TM2 = trigger registers of matchers 1 and 2
a.b:incr->TC1; a.!b:decr->TC1;       // TC1, TC2 = trigger registers for counters 1 and 2
```

The assembler code for this algorithm requires five data moves, and can be carried out in two cycles. In the code, both of the match operations are carried out in parallel. The third move in the first cycle is used for updating counter2 (in the original algorithm counter2 is updated at the end of the algorithm, but updating it earlier does not have an effect on the outcome of the algorithm). Counter1 is incremented in the second cycle only if both match results are true, and decremented, if the result from matcher1 is true and the result from matcher 2 is false. If the result from matcher1 is false, counter1 is not updated. Note the utilization of conditional moves.

4. SIMULATIONS

As our first case study we created a simulation of the protocol processor presented in figure 1. As its application we used an algorithm for processing ATM AIS (Alarm Indication Signal) cells [4]. The algorithm is used for analyzing 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 FU's. The AIS algorithm was chosen as an example since another approach to its processing was presented by Jantsch et al. in [5]. The control flow of the example is shown in figure 7. AIS cells cause the system to enter AIS state, in which cells are sent from the buffer that holds outgoing cells, and an AIS cell is inserted after every 1024 regular or empty cells. If a

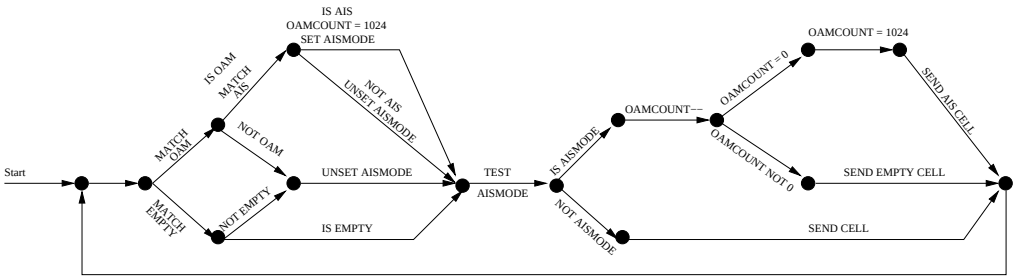
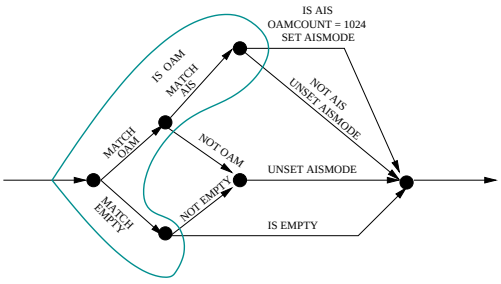


***Figure 7**. Control flow of the ATM AIS processing example.*

regular cell appears at the processor input when the system is in AIS mode, the system returns to normal processing. If an empty cell appears at the input, it is discarded. If there are no cells to be sent, empty cells are transmitted. A part of the assembler implementation is shown in figure 8, where each block of three instructions represents code executed in one clock cycle. Note in figure 8 the corresponding parts in the code (boldface) and the control flow diagram (dotted line): the three match operations can be executed in one clock cycle when a compare unit is used for matching an empty cell.




```
FIFO -> R1;       // cell header mem addr from FIFO to R1
FIFO -> OMM;      // cell header mem addr into MMU op reg
0 -> TRMM;        // trigger data mem read with offset 0

RMM -> R2;        // put mem read result into register R2
1 -> TRMM;        // trigger data mem read from address R1+1
0 -> OCM1;        // zero compare 1 to check if cell is empty

R2 -> TM1;        // perform OAM match operation
R2 -> TEQ1;       // perform empty cell compare operation
RMM -> TM2;       // perform AIS match operation

RMM -> R3;        // put data memory word to register R3
!a.!c:0 -> OCM2;  // cell not empty/not OAM -> unset AIS mode
a.!b: 0 -> OCM2;  // OAM but not AIS -> unset AIS mode

a.b:1 -> OCM2;    // if OAM and AIS -> set AIS mode
1 -> TEQ2;        // Test AIS mode
```

***Figure 8**. Part of the example's control flow and its corresponding assembler code.*

We also obtained results for a processor without the dual FU's, i.e. a processor with just one matcher, counter and compare unit, for comparison purposes. The results of our simulations are shown in figure 9 and table 2. Data bus utilizations presented in figure 9 show quite clearly that the processor with only one matcher, counter and compare unit is not able to take advantage of the third data bus: the third bus is used only during 19% of the cycles needed to process an ATM cell, whereas the processor with dual FU's has a 95% bus utilization when running this AIS processing algorithm.

Table 2 shows that the processor with dual FU's is able to process an ATM cell 30-40% faster than the processor without dual FU's (the increase in speed depends on the ATM cell type received). The theoretical minimum requirement for processor clock speed is calculated for 622 Mbps ATM, which transports approximately 1,47 million cells per second. In the dual FU processor cycle count depends on the type of cell to be sent.

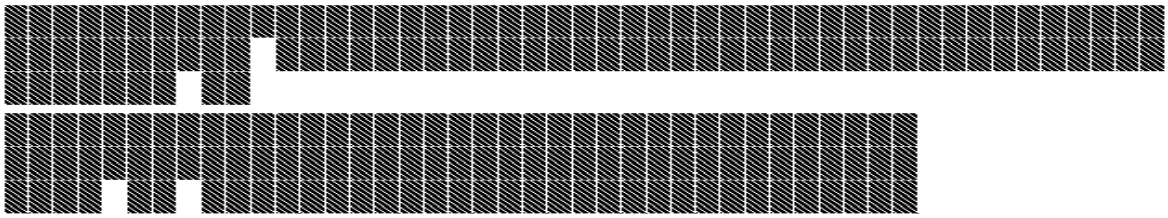


***Figure 9**. Data bus utilization and worst case clock cycles in prototyped TACO processors. Above: processor without dual FU's, below: processor with dual FU's. Rows represent data buses and columns represent clock cycles. Darkened box means bus activity.*

Processor	Lines of asm code	Bus instructions	Cycles	Min. clock [MHz]
TACO 1	26	63	47	69
TACO 2	19	52	33-37	55

Table 2. *TACO processor comparison in processing ATM AIS cells. TACO 1 is a processor with one matcher, counter and compare unit, TACO 2 has two of each.*

5. CONCLUSIONS AND FUTURE WORK

We have presented a modular and scalable microprocessor architecture for protocol processing applications. Our design is based on the Transport Triggered Architecture. We have implemented a prototype processor both in SystemC and VHDL and successfully simulated ATM AIS cell processing. The long term goal of the project is to develop a design environment for protocol processors that includes a compiler, simulators and a tool for physical estimation, as shown in figure 10. In this context we will e.g. explore how to automatically generate VHDL code from the simulators/executable specifications. Also an important line of work is to study the effects of different VLSI design methods to processor speed and power consumption.

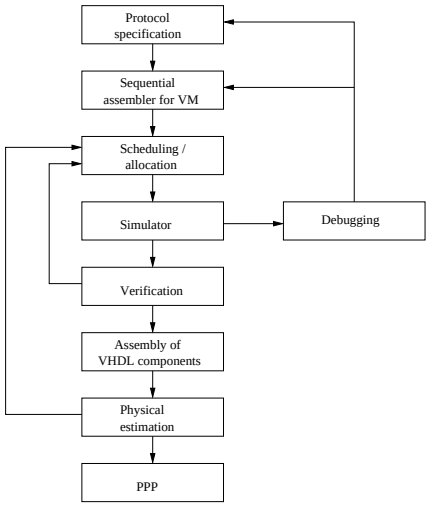


Figure 10*. Long term goal of the TACO project: a protocol processor design environment [12].*

6. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the discussions concerning protocol processors with professor **Jouni Isoaho** (Laboratory of Electronics and Information Technology, University of Turku).

7. REFERENCES

[1] 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.

[2] H. Corporaal, "Microprocessor Architectures - from VLIW to TTA". John Wiley & Sons Ltd, Chichester, West Sussex, England, 1998.

[3] 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.

[4] ITU-T Recommendation I.610, “B-ISDN Operation and Maintenance Principles and Functions”, International Telecommunication Union, Telecommunication Standardization Sector, 1993.

[5] 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.

[6] B. Stroustrup, “The C++ Programming Language - Third Edition”. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1997.

[7] SystemC: http://www.systemc.org

[8] The IA-64 processor architecture: http://developer.intel.com/design/ia-64/

[9] 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.

[10] S. Virtanen, "On Communications Protocols and their Characteristics Relevant to Designing Protocol Processing Hardware", TUCS Technical Report 305, Turku Centre for Computer Science, Turku, Finland, 1999.

[11] S. Virtanen, J. Isoaho, T. Westerlund and J. Lilius, "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.

[12] S. Virtanen, J. Lilius and T. Westerlund, “The TACO Protocol Processor Development Environment”, To appear in TUCS Technical Report series, Turku Centre for Computer Science, Turku, Finland, 2000.