Novel ASIP and Processor Architecture for Packet Decoding

Tomas Henriksson
Dept. of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden
Phone: +46-13-288956
E-mail: tomhe@isy.liu.se

Dake Liu
Dept. of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden
Phone: +46-13-281256
E-mail: dake@isy.liu.se

ABSTRACT

Packet decoding is important in terminals as well as in switches and routers. A new instruction set architecture and a new processor microarchitecture is needed since traditional processors cannot keep up with the network speed. Our protocol processor works in-line with the data flow and can execute conditional jumps in one clock cycle since it uses a novel way to divide the program into three parts. The processor has been synthesized and static timing analysis based on the synthesized and placed netlist indicates that it can support a data stream of more than 10 Gb/s.

1. INTRODUCTION

Computer networks keep increasing the transmission bandwidth at a pace much higher than what microprocessors can keep up with [11], [10]. The bottleneck is no longer the transmission media, but rather the networking equipment, such as network interface cards (NICs), switches and routers [3]. At the same time as faster throughput is required from these devices, the protocols that are used for the communication keep changing. This calls for programmable solutions, which can support both high throughput and enough flexibility.

The terminals have not received as much attention as needed, but suffer from heavy processing, which must be off-loaded from the host processor [4], [2] and [1]. An important part off that processing is the packet decoding and checksum calculation.

In this paper we present a new instruction set architecture (ISA), which is optimized for the packet decoding task. Along with the ISA we also present a processor architecture, the protocol processor (PP), that can execute the instruction set efficiently. The PP architecture was outlined in [5]. In this paper the PP ISA is described along with performance figures for the synthesized and placed standard cell netlist. Based on the ISA a functional coverage analysis is done.

The basic principle for the PP is that it must be able to execute several conditional branch instructions at once with predictable execution time no matter if a jump is taken or not. This is solved by never letting the PP miss any clock cycles which implies that no pipeline can be used. Since no pipeline is used, the program code has to be self-contained within the processor core in order to allow fast enough cycle time. This in turn implies that the program code has to be kept very small, which actually is possible because of the application specific nature of the PP and the way the program is split up into extendable instructions and special lookup tables. The architecture is protected by a pending US patent [6].

The rest of this paper is organized as follows. In section 2 an overview of the PP is given, then the instruction set is described in section 3, and an example program is explained in section 4. Section 5 deals with the functional coverage of the instruction set and the functional verification. Section 6 describes the implementation and in section 7 the performance figures are given and discussed. In section 8 we compare our PP to related work and finally, section 9 concludes our findings.

2. PROTOCOL PROCESSOR OVERVIEW

The PP decodes packets as they are received. It works in-line with the data flow, that means that it processes the packet header before it is stored in memory. The PP is directly connected to the interface to the physical layer. For example in an Ethernet environment the PP replaces the MAC part of the Ethernet circuit, but incorporates more functionality, such as IP, TCP and UDP processing. Based on the program and the content of the received packet header the PP stores the packet payload on a predefined location in the payload memory. Figure 1 shows how the PP fits in a system. The PP only performs single packet operations. Other tasks, such as connection state handling have to be supported by a simple microcontroller. The application accesses the data directly from the payload memory and sends directives to the microcontroller when it wants to set up or tear down connections. The microcontroller then configures the program in the PP accordingly. The microcontroller and the application processor can be the same processor unit, in that case the interface is a software API.

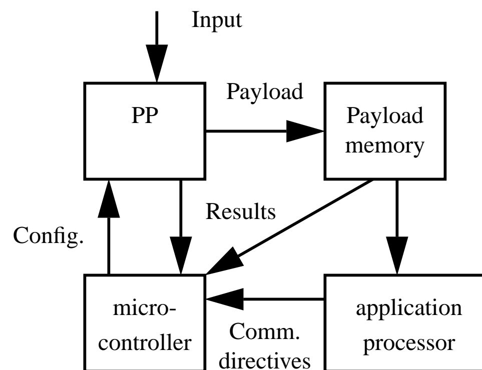

Figure 1: PP system overview

The PP architecture is quite different from traditional processor architectures. It supports one cycle compare operations and conditional jumps without any penalty. These instructions occur frequently in packet decoding. It is necessary to execute all instructions in the PP in one clock cycle because the instruction flow is aligned with the data on the input port clock cycle by clock cycle. A penalty would imply that the PP would lose its synchronization with the data.

There is only one register, the input buffer, which can be used as an input. Data is written implicitly to this input buffer from the input port, e.g. XGMII (eXtended Gigabit Media Independent Interface). There is no in-buffer data memory, so load and store instructions do not exist. The result of the operations are stored in specific output registers, which can be accessed by the microcontroller as memory-mapped registers. The PP communicates with its accelerators (inside the PP) by the use of internal inputs and outputs. The outputs can be set in order to trigger an accelerator to start and the inputs can be used for conditional jumps in the PP. Similar external inputs and outputs are used for communication with the rest of the system, for example the microcontroller.

3. INSTRUCTION SET ARCHITECTURE

The instruction set for the PP is different from a traditional instruction set. It consists of only six instructions which can have extensions in order to allow several parameters. Three of the six instructions can have pointers, which specify extensions to the instructions. This can be thought of as variable length instructions, but physically the instruction storages are also split up. Thereby even the longest instruction can be fetched and executed in one clock cycle.

Since the instructions can be extended, the assembly description is fairly complex. To make it easier to understand, it is specified in accordance with the physical storage separation. That means that the program is specified in three parts, the instruction lookup table (ILT), the parameter codebook (PCB) and the control codebook (CCB). The ILT contains the main instruction flow and the PCB and the CCB contain the extensions.

3.1. Instruction Lookup Table

The general instruction format of the 24 bit instructions in the ILT is shown in figure 2. Four bits are used to specify the instruction code. Since only six instructions are used, the instruction set is sparse and future changes are accommodated for. The six instructions with their various formats are shown in figure 3. The buffer control bit (bit 19) is used to control the dynamic input buffer, the FIFO, which can hold either one or two words of data.

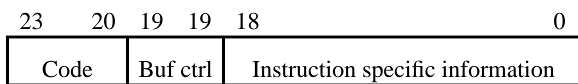

Figure 2: Instruction format

3.2. Parameter Codebook

The PCB contains the parameters, which are used for the compare and jump operations. The parameters are organized in lines. A maximum of 16 lines can be used, since the pointer in the ILT is four bits wide. Each line contains four 32 bit parameters. In a compare operation, the four selected parameters are fed to a an array of four comparators. The extracted field from the input buffer is fed to all four comparators. The width bits carried by an instruction are used to mask the data to the desired width (4, 8, 16, or 32 bits). The offset bit decides which part of the input buffer to extract into the extracted field. The result of the comparators is a four bit array, which is used as an input to the CCB.

The comparison is controlled by the new bit, which specifies if the comparison should start from scratch or if it is a continuation of a previous comparison. The resulting array of a comparison is always implicitly stored within the comparator array for this purpose. For example for IPv6 addresses a 128 bit comparison can easily be accommodated for in this way.

3.3. Control Codebook

The CCB contains relative jump addresses for the compare instructions. The CCB has only 8 lines of each four values. The reason for having more lines in the PCB than in the CCB is that the CCB is only used for CMP and CPS instructions, with the jump bit set to 1. The PCB is also used for CMP and CPS instructions with the jump bit set to 0 and for JMP instructions with type 10.

Inputs to the CCB are the 3 least significant bits of the pointer and the resulting array from the comparators. The pointer selects a line and the array from the comparators selects a value within that line. The value, just as the relative jump offsets in the JMP instruction, is 128 + the actual relative jump address. Thereby no negative values are needed.

4. EXAMPLE: PROGRAM FOR ETHERNET/IP/UDP DECODING

The execution in the PP is best explained by an example program. Here we have chosen a program, which decodes Ethernet II frames containing IP/UDP packets or ARP packets. All other packets are discarded. Only UDP packets to port 2025 (0x07e9) are accepted. The IP address of the node where the PP is operating, is in this example 130.236.55.5 and the hardware address is 0x0c5a80ac4ab7. The ILT contents are shown in figure 4, where it is also shown in assembly representation. The PCB contents are shown in figure 6. Each line in the PCB consists of 4 32 bit parameters. The line number is specified to the right of the first parameter. The other three parameters follow on the three rows below it in the figure. It can be seen that line 0 contains the Ethernet codes for IP and ARP, line 1 contains the protocol value for UDP, line 8 contains the first part of the hardware address and line 9 the second. Line 10 contains the acceptable IP destination addresses and line 11 the acceptable UDP destination ports. The other lines are not used by the example program. The corresponding CCB content is shown in figure 5. Line 0 contains the corresponding relative jump addresses for the Ethernet codes and line 1 contains the corresponding relative jump address to the protocol value. The protocol

NOP No operation, mostly used for aligning the data flow processing

23 20	19	18 0
0000	Buffer ctrl	000000000000000000

WAT Wait for inputs

23 20	19	18 0
0101	Buffer ctrl	Input bitmap

Waits until the general purpose inputs matches the input bitmap. All 19 inputs can be used to trigger the continuation of the program execution.

SET Set outputs

23 20	19	18 10	9 0
0010	Buffer ctrl	Not used	Output bitmap

Sets the general purpose outputs for one clock cycle. All 10 outputs can be set by using the bitmap.

JMP Jump

23 20	19	18 17	16 13	12 11	10	9	8	7 0
0100	Buffer ctrl	00	Not used					Relative offset
0100	Buffer ctrl	01	Input bitmap					Relative offset
0100	Buffer ctrl	10	Pointer	Width	Offset	New	Not used	Relative offset

Jumps to the relative offset specified in bits 7-0. Bits 18-17 decides the type of jump. Type 00 means unconditional. Type 01 means conditional on the general purpose inputs. The bitmap in bits 16-8 corresponds to the inputs 8-0. Type 10 means conditional on match. Match is further described in the CMP instruction. The jump is taken if any of the parameters match the extracted field from the input buffer.

CMP Compare

23 20	19	18	17	16 13	12 11	10	9 0
0001	Buffer ctrl	New	Jump	Pointer	Width	Offset	Not used

Compares the extracted field with the parameters. The parameters are stored in the PCB and the pointer points out the parameters to use. The field New indicates if the comparison is the continuation of the previous comparison or not. The field Jump indicates if a jump should be conducted at a match. The field Width determines the width of the comparison, 4, 8, 16, or 32 bits. The field Offset indicates how to extract the field from the input buffer.

CPS Compare and set outputs

23 20	19	18 10	9 0
0011	Buffer ctrl	As for CMP	As for SET

Sets the outputs and performs a comparison.

Figure 3: Detailed instruction format for all instructions

```
0  500001  WAT 0, input(0)
1  151800  CMP 0, new=1, jump=0, pointer=8, width=32, offset=0
2  453483  JMP 0, type=10, pointer=9, width=16, offset=16, new=0, jump=0x83(5)
3  200040  SET 0, output(6)
4  40007c  JMP 0, type=00, jump=0x7c(0)
5  000000  NOP 0
6  361413  CPS 0, new=1, jump=1, pointer=0, width=16, offset=16, output(4, 1, 0)
7  40007c  JMP 0, type=00, jump=0x7c(3)
8  200044  SET 0, output(2,6)
9  162800  CMP 0, new=1, jump=1, pointer=1, width=8, offset=0
10 400079  JMP 0, type=00, jump=0x79(3)
11 000000  NOP 0
12 080000  NOP 1
13 455e82  JMP 0, type=10, pointer=10, width=32, offset=16, new=1, jump=0x82(15)
14 400075  JMP 0, type=00, jump=0x75(3)
15 457682  JMP 0, type=10, pointer=11, width=16, offset=16, new=0, jump=0x82(17)
16 400073  JMP 0, type=00, jump=0x73(3)
17 200010  SET 0, output(4)
18 50002a  WAT 0, input(5, 3, 1)
19 425482  JMP 0, type=01, input(6, 4, 2), jump=0x82(21)
20 40006f  JMP 0, type=00, jump=0x6f(3)
21 200120  SET 0, output(8, 5)
22 40006a  JMP 0, type=00, jump=0x6a(0)
23 200008  SET 0, output(3)
24 500002  WAT 0, input(1)
25 420482  JMP 0, type=01, input(2), jump=0x82(27)
26 400069  JMP 0, type=00, jump=0x69(3)
27 2000a0  SET 0, output(7, 5)
28 400064  JMP 0, type=00, jump=0x64(0)
```

Figure 4: Example program for Ethernet II, ARP and IP/UDP decoding

```
82 0                00 4
91                  00
00                  00
00                  00
82 1                00 5
00                  00
00                  00
00                  00
00 2                00 6
00                  00
00                  00
00                  00
00 3                00 7
00                  00
00                  00
00                  00
```

Figure 5: CCB contents for example program

check could have been implemented with a JMP instead of a CMP (instruction 9 in the ILT) but this would have made it harder to extend the program to also handle other layer 4 protocols, TCP for example.

From the beginning, instruction 0 waits for input 0, which indicates packet start. Instruction 1 then compares the first 32 bits of the Ethernet destination address with the acceptable parameters from PCB line 8. The result of the comparison is only stored locally in the comparator array since the jump bit is set to 0. Instruction 2 continues the comparison, since the new bit is set to 0. Here only 16 bits are used and compared to PCB line 1, since the width code is 10. If any match occurs, i.e. the Ethernet frame is destined for the host, a jump is done to instruction 5. Instruction 5 is NOP to align the data flow processing (in this example we do not care about the Ethernet source address). Instruction 6 compares the Ethernet type field with PCB line 0 and uses the jump addresses from CCB line 0. So if the type field is 0x0800 a jump is done to instruction 8, otherwise, if it is 0x0806 a jump is done to instruction 23. If there is no match the execution continues with instruction 7. At the same time outputs 4, 1, and 0 are set. These are used to trigger the start of accelerators for payload storage, IP header checksum calculation, and UDP checksum calculation. The Ethernet CRC accelerator was already triggered by input 0.

Continuing the execution at instruction 8 (assuming that the arriv-

```
00000800 0      ffffffff 8
00000806        0c5a80ac
00000000        00000000
00000000        00000000
00000011 1      0000ffff 9
00000000        00004ab7
00000000        00000000
00000000        00000000
00000000 2      82ec3705 10
00000000        82ecffff
00000000        ffffffff
00000000        00000000
00000000 3      000007e9 11
00000000        00000000
00000000        00000000
00000000        00000000
00000000 4      00000000 12
00000000        00000000
00000000        00000000
00000000        00000000
00000000 5      00000000 13
00000000        00000000
00000000        00000000
00000000        00000000
00000000 6      00000000 14
00000000        00000000
00000000        00000000
00000000        00000000
00000000 7      00000000 15
00000000        00000000
00000000        00000000
00000000        00000000
```

Figure 6: PCB contents for example program

ing packet is IP) outputs 2 and 6 are set. Output 2 triggers the length counter accelerator for IP and output 6 stops the payload storage. For an IP/UDP packet only the UDP payload should be stored. For an ARP packet on the other hand, the whole Ethernet payload is stored, since the data is needed by the microcontroller in order to compile the ARP reply. Instruction 9 checks the protocol field in the IP header and if it is 0x11 (UDP) a jump is done to instruction 11. Instruction 11 is NOP and so is 12 (data flow aligning), but instruction 12 is the first (and only in this example) to use the second word in the input buffer. This means that the last 64 bits from the input will be available for instruction 13. This is also needed, since instruction 13 is JMP with a compare of 32 bits with an offset of 16 bits, meaning that bits 47 down to 16 are extracted from the input buffer. In instruction 13 that is the IP destination address, which is compared with PCB line 10. For a correct packet, then the UDP port is checked by instruction 15 and instruction 17 triggers the payload storage to start again. After that, the header has been processed and the PP waits for inputs 5, 3, and 1 in instruction 18. These three inputs indicate that the IP header checksum accelerator, the UDP checksum accelerator and the CRC accelerator have completed their computations. In instruction 19 a conditional jump is done on inputs 6, 4, and 2. These are all 1 if the just mentioned accelerators have received correct checksums. Then finally, the reception of a valid IP packet is acknowledged through outputs 8 and 5 in instruction 21 and instruction 22 jumps back to instruction 0 in order to wait for the next packet.

If the Ethernet code was ARP, instructions 23 to 28 would have executed in a similar manner. Whenever the received packet does not match the requirements the packet is discarded and the PP waits for the next packet. This is done by a jump to instruction 3, which set output 6, discard payload, and then instruction 4 jumps back to instruction 0.

5. FUNCTIONAL COVERAGE

As mentioned earlier, the PP operates on a frame in-line, as it is received on the input port. Therefore the PP operates, as planned, on one frame only and does not handle any inter packet operations. The PP can handle layer 2 protocols, like various Ethernet packet formats, layer 3 protocols, like IP and layer 4 protocols, like UDP. For connection based protocols, like TCP and some wireless layer 2 protocols, the PP supports the microcontroller by decoding the incoming packets. The microcontroller handles the updating of the connection state variables and the sending of packets. The PP can also be used to decode any other packet stream, for example decoding MPEG control layer. The PP can be used in terminals as explained in figure 1, but can also be used as a port accelerator in a switch or a router. In these devices it is specially important with programmability, since system companies want to use their own proprietary protocols to communicate between the switches and routers within a system. Although we have not been able to receive a specification for such a protocol, we cannot see any obstacle for using the PP to decode it.

The PP has four parameters in each line of the PCB and CCB. This limits the number of ports etc. that can be handled. If there is a need for more parameters, the PCB and CCB can be made wider following the trade off of performance and silicon area.

The functional verification of the PP, was divided into three kinds of test cases, single instruction based, formal functions and error injections.

5.1. Single Instruction Based Verification

The single instruction based verification for the PP differs from that of a traditional processor. Since there are no target registers for the instructions, there is only the program counter and the outputs, that can prove the correctness of the implementation. The WAT, JMP, CMP, and CPS instructions all influence the program counter. The CPS and SET instructions influence the outputs. All instructions can influence the buffer content.

The coverage of the verification is dependent on two parts, the control signals and the data pattern. The control signals are decided by the instruction in the ILT and the CCB content. The data pattern is decided by the content in the PCB and the received packet. For each instruction all possible correlated control signal combinations were listed and for them where the data pattern influences the outcome of the execution corner cases were selected.

The VHDL model of the PP was extended with a non-synthesiz-

able part which writes the PC and the outputs and the input buffer to a file every clock cycle for verification purposes only. For all input combinations, corresponding reference files were manually created in order to simplify the verification task.

This covers most of the RTL code, but of course the coverage is not 100%. Control signals that do not interfere were not tested in all combinations and all data patterns were not used, since that would have required too much time.

5.2. Formal Functions Verification

For the verification of the formal functions, the example program from section 4 was used. It covers the following functions as basic and kernel functions for a general purpose protocol processor:

- Synchronize the processing based on information from the physical interface
- Match packet header field to several acceptable values
- Demultiplex packet processing based on upper layer protocol
- Use checksums to assure that no transmission errors have occurred
- Hand over a correct packet payload to the application processing

The reception processing was first modelled in C++ at a behavioral level. Then a structural C++ model was developed, which executes the instruction set in a cycle true manner. The simulation needs four inputs, the ILT content, the PCB content, the CCB content and the received packet. The structural model was then manually transferred into a VHDL model, which is used for implementation. The testbench for the VHDL model use the same stimuli files as the C++ model and thereby the VHDL was verified efficiently.

5.3. Error Injection

To make sure that the PP executes all program branches correctly, packets with various errors were injected into the simulation. These faulty packets cover the following errors:

- Ethernet destination address that is not for the host
- IP destination address that is not for the host
- Ethernet code which is not IP or ARP
- IP protocol which is not UDP
- UDP port which is not 2025
- Wrong IP header checksum
- Wrong UDP checksum
- Wrong Ethernet CRC

The resulting operation was checked in the GUI of the simulator.

6. IMPLEMENTATION

The VHDL model described in the previous section was used for the implementation. The model contains the part of the PP that executes the instructions, the configuration interface and 5 accelerators. The microcontroller can access the ILT, the PCB, and the CCB via an SRAM interface. The lookup tables are implemented by flip-flops. The microcontroller finishes the configuration of the PP by writing data to the first position in the ILT, which triggers the start of program execution in the PP. Figure 7 provides a block diagram of the PP. PC is the program counter, ID is the instruction decoder, NextPC is the unit that calculates the next PC value based on the control signals from the ID, the output from the CCB and the inputs to the PP. The configuration interface and the accelerators are not shown in the figure.

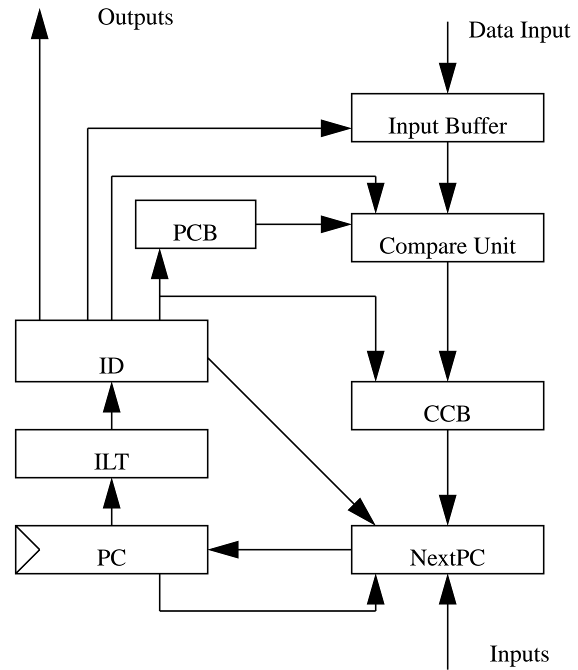

Figure 7: PP block diagram

The accelerators that were used were an Ethernet CRC accelerator, an IP header checksum accelerator, a UDP checksum accelerator, a packet length counter and a memory interface unit.

The PP was synthesized to a 6 metal layer 0.18 micron library from UMC in order to get an accurate estimate of the performance. Cadence Envisia PKS was used for the synthesis and placement of the standard cells.

7. RESULTS AND DISCUSSION

All results are estimations after synthesis and placement. The PP uses an area of 0.4 mm^2 without the accelerators. The accelerators have previously been implemented [7], [8], [9]. The three lookup tables, that totally contain 3072 flip-flops use more than half of the total PP area. The lookup tables can be implemented with memory macro blocks instead of standard cell flip-flops, which would decrease the area consumption even further.

The delay in the critical path is 3.43 ns and with a setup time of 0.12 ns, a minimum cycle time of 3.55 ns can be used. This corresponds to a maximum clock frequency of 281 MHz and the PP can thus support a data stream at more than 9 Gb/s, since 32 bits are processed every clock cycle. Therefore, when implementing the PP in a 0.13 micron process we have clear indications that the PP will support a data stream of more than 10 Gb/s.

The critical path is from the PC, through the ILT, the PCB, the

compare units, the CCB and an adder back to the PC. The delay from the parts can be seen in table 1.

Techniques in order to reduce the delay of the critical path, such as flip-flop cloning, have not been used since there is only a need to support standard network speeds such as 10 Gb/s and that will be managed by using a 0.13 micron technology.

8. RELATED WORK

Several publications exist which address the topic of off-loading the protocol processing from the host processor, for example [4], [2] and [1]. There are also many companies working in this area, iReady, Alacritech, Intel, Agilent, Silverback, Trebia are some examples. From the companies limited information on the detailed implementation is available.

Typical for all published work is that it uses traditional RISC-like processors for the protocol processing off-loading. For example, in [2] a 133 MHz general purpose RISC processor was used. We have designed a completely new processor architecture and ISA, which is dedicated for protocol decoding and therefore can reach performance as high as 10 Gigabit/s. The other solutions aim at Gigabit Ethernet and with the RISC-like processor it will be hard to increase the performance 10 times, although some of the start-ups promise scalable designs. On the other hand some other solutions manage the complete TCP protocol for example, our PP handles only the decoding of received packets. Therefore our processor can be seen as a part of a TCP off-loading engine. Combined with other specialized processors it can constitute a complete off-loading engine for 10 Gigabit Ethernet.

The RISC processor based implementations assume that the packet is stored in a memory. Our processor on the other hand operates at the packet before it is stored in memory. If it can be decoded from the packet header that the packet should not be processed further, the packet payload is never stored in the memory at all. This saves memory bandwidth and power consumption.

9. CONCLUSIONS

A protocol processor for in-line packet decoding has been designed and implemented. The instruction set consists of only 6 instructions, but they support many variations which allows enough flexibility. The processor has been implemented in a 0.18 micron technology and static timing analysis performance estimation indicates that the processor can support 9 Gb/s data streams.

Table 1. Delays

Part	Delay [ns]
PC (Clk to Q)	0.21
ILT	0.74
PCB	0.86
compare and CCB	1.16
Adder (NextPC)	0.46
Setup	0.12
Total cycle time	3.55

Future work includes the integration of the processor into an FPGA-based demonstrator for audio reception and implementation in a 0.13 micron technology for more accurate performance analysis. Evaluations and optimizations of processor word length, program memory size and processor internal parallelism in the comparator array are also planned. Further on a compiler will be developed.

10. REFERENCES

[1] Alacritech, Delivering High-Performance Storage Networking, Alacritech whitepaper, on the www: http://www.alacritech.com/

[2] P. Buonadonna, D. Culler, Queue Pair IP: A Hybrid Architecture for System Area Networks, *International Symposium on Computer Architecture 2002*, pp. 247-256, June 2002, Anchorage, Alaska

[3] W. Bux, W. E. Denzel, T. Engbersen, A. Herkersdorf, and R. P. Luijten, Technologies and Building Blocks for Fast Packet Forwarding, *IEEE Communications Magazine*, Vol. 31, No. 1, Jan 2001, pp. 70-77

[4] L. Gwennap, Count on TCP offload engine, EETimes, on the www: http://www.eetimes.com/semi/c/ip/OEG20010917S0051

[5] T. Henriksson, U. Nordqvist, and D. Liu, Embedded Protocol Processor for Fast and Efficient Packet Reception, in *International Conference on Computer Design* 2002, pp. 414-419, September 16-18, 2002, Freiburg, Germany

[6] T. Henriksson, D. Liu and H. Bergh, METHOD AND APPARATUS FOR GENRAL-PURPOSE PACKET RECEPTION PROCESSING, US Patent application no. 09/934372

[7] T. Henriksson, H. Eriksson, U. Nordqvist, P. Larsson-Edefors, D. Liu, VLSI IMPLEMENTATION OF CRC-32 FOR 10 GIGABIT ETHERNET, in Proceedings of *International Conference on Electronics, Circuits and Systems 2001*, vol III, pp. 1215-1218, September 2-5, 2001, Malta

[8] T. Henriksson, N. Persson, D. Liu, VLSI IMPLEMENTATION OF INTERNET CHECKSUM CALCULATION FOR 10 GIGABIT ETHERNET, in Proceedings of *Design and Diganostics of Electronics, Cricuits and Systems*, pp. 114-121, April 17-19, 2002, Brno, Czeck Republic

[9] T. Henriksson, U. Nordqvist, D. Liu, Specification of a configurable General-Purpose Protocol Processor, *IEE Proceedings on Circuits, Devices and Systems*, Vol 149, No. 3, 2002, pp. 198-202

[10] J. Williams, Architectures for Network Processing, *International Symposium on VLSI Technology, Systems, and Applications 2001*, pp. 61-64

[11] T. Wolf and J. S. Turner, Design Issues for High-Performance Active Routers, *IEEE Journal on Selected Areas in Communications*, Vol. 19, No. 3, March 2001, pp. 404-409