Embedded Protocol Processor for Fast and Efficient Packet Reception

Tomas Henriksson, Ulf Nordqvist and Dake Liu
Dept. of Electrical Engineering, Linköpings universitet
SE-581 83 Linköping, Sweden
E-mail: {tomhe, ulfnor, dake}@isy.liu.se, Phone: +46-13-28{8956, 2903, 1256}

Abstract

Computer networks equipment present a bottleneck for further increase of the capacity in the networks. The terminals have problems to keep up with the network speed when using general purpose processors for the protocol processing. We present a novel processor architecture, that works in-line with the data flow and does not use a traditional von Neuman architecture. The program is contained in three lookup tables within the processor core, which allows for one cycle if-then-else and switch-case-case... execution. The processor is estimated to be able to handle a 10 Gb/s Ethernet connection when implemented in a 0.18 micron technology.

1. Introduction

During the last years, the capacity of computer networks has exploded and today it is no longer the physical transmission media that limits the performance. Instead computer networks equipment has become the bottleneck [2]. Many research efforts have been made in order to make fast switches and routers [3] [4], but less effort has been put into the network terminals. However, it has been stated that a traditional computer, that is connected to a Gigabit Ethernet uses 20%-60% of its processing power for protocol handling and connecting a traditionally designed computer to a 10 Gigabit Ethernet is totally in vain [5].

The solution to the processing problem is host processor off-loading. Basically the protocol processing functionality should be executed by another device, working in parallel with the host processor. This requires not only new network interface card (NIC) design, but also operating system (OS) rewriting. Although major redesign efforts are necessary, this is seen as the only way of increasing the actual network capacity to the end user.

To modern computer networks not only traditional computers are attached, but also various kinds of embedded systems and IP telephones. The IP telephones show an interesting feature of having a very low application bitrate and being attached to a high-speed network. This asks for an architecture, which splits the processing into two parts. One, which works at network speed and handles the network protocols and another, which works at the application speed and handles the application processing.

In this paper we present a protocol processor architecture consisting of a core and accelerators, that is dedicated for processing network protocols at network speed as a packet is received. Although this paper deals with Ethernet and IP, the same architecture can be programmed for other protocol stacks. The architecture is protected by a US patent [1], which is pending at the time of writing. The rest of the paper is organized as follows. In section 2, we describe the network terminal system and how the protocol processor fits into it. In section 3, the internal architecture of the core protocol processor is described and in section 4 the accelerators are explained. Section 5 provides a discussion on the efficiency of the architecture and compares it to other existing processors. Finally, in section 6, we draw some conclusions and outline future work in this area.

2. System perspective

In a network terminal, there are many bottlenecks, that have to be considered. It is important to understand the fundamental functionality and performance limits before designing any new processor architecture.

2.1. Traditional implementation

Traditionally a computer has received packets through a NIC, the Ethernet packet layer is processed on the NIC and the packet is buffered on the NIC, before it is transferred to the main memory. Then the IP header and the TCP or UDP header is processed by the OS. Since this implies calculating the checksum over the whole packet, the whole packet has to be read from memory. The OS can also have the memory area divided into one kernel part, for the OS, and one user part, for the applications. This implies a consecutive write operation of the whole packet as well. Figure 1 shows how this works.

2.2. Fundamental functionality

The above described process is not necessary in order

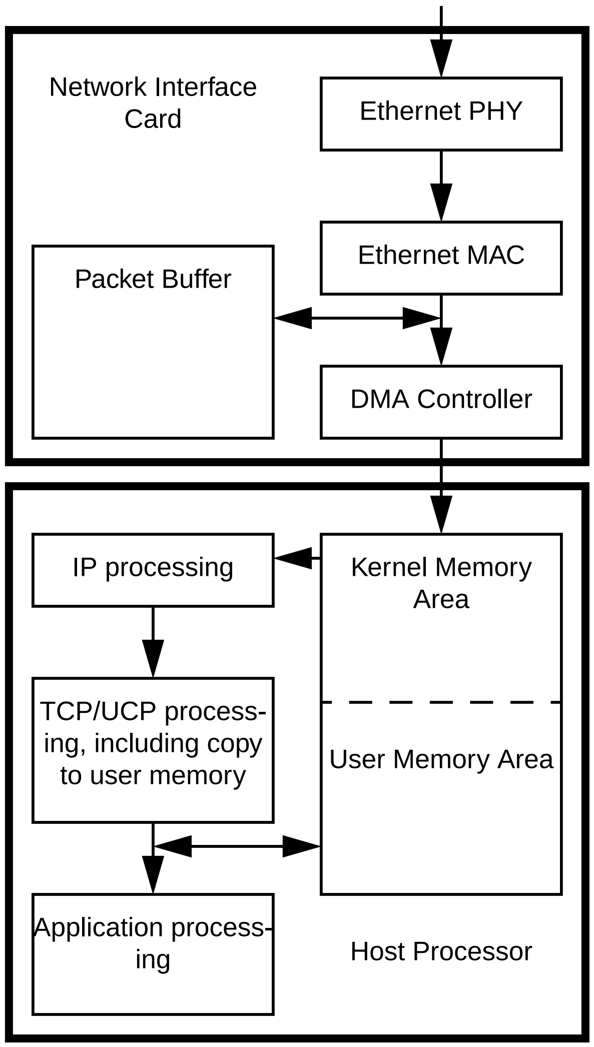

Figure 1: Traditional packet reception

to fulfill the fundamental functionality of packet reception, which is to check that the packet is received correctly, destined for the correct terminal and then deliver the payload to the application. That means, that the packet payload only has to be buffered in memory once, when changing from network speed to application speed. To do so however, requires that the network protocol processing is done at network speed and therefore a real-time system is required. It also requires the memory to allow instant writes from the NIC at any time.

2.3. Protocol processor environment

After having realized this, it is clear that as much as possible of the protocol processing must be off-loaded from the host processor. This can be done in different ways and what is described here is just one possibility. The protocol processor that we present operates on one frame at a time and therefore needs support in order to update connection state variables and trigger the sending of packets, for example acknowledgments. Since those tasks can be done concurrently with application processing and reception of the next packet, we assume that they are handled by a processor of traditional design. We refer to that processor as the supporting microcontroller or simply, the µC. The protocol processor is most suitable for protocols that do not include processing over several packets, e.g Ethernet, IP and UDP, if we assume that the IP reassembly can be handled by the µC. The suggested packet reception architecture is depicted in figure 2.

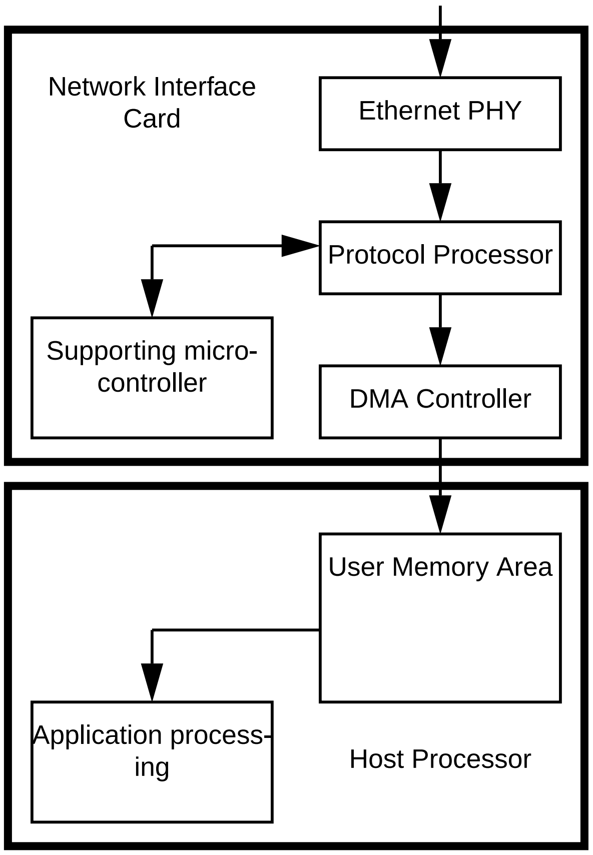

Figure 2: Suggested new packet reception

3. Protocol Processor Architecture

The previous section described the environment for the protocol processor and thereby put some constraints on the architecture. Instead of having the data in a memory, the data arrives at constant network speed on an input port. This makes the protocol processor a data stream processor for in-line processing. Since the data does not have to be loaded from memory, traditional registers are not necessary. Therefore the processor has very few internal states, the most important is the program counter. Others are just some status bits and a dynamic input buffer, which will be described later. The architecture is radically different from a von Neuman architecture and is referred to as the Linkoping architecture in table 1,which lists the main features.

Topic	von Neuman	Linkoping	Advantage
Data	RF and memory	input port	load operation not needed, less memory needed
Program	In memory	Distributed	load operation not needed, less memory needed
If control flow	pipeline penalty	no penalty	real time behaviour, faster
Case control flow	many instructions	single instruction	real time behaviour, faster
Regular tasks	ALU	Accelerators	Higher degree of parallelism

Table 1. Architectural features

3.1. Important instructions

The fundamental functionality translates into checksum calculations, field matching and memory management when looking at it at a lower level of abstraction. The checksum calculations are of two types, CRC and 1's complement addition. The field matching operations, when described in a sequential programming language, translates into compare instructions and if-then-else and switch-case-case... control flows. The key is to implement an instruction set architecture, which supports real-time fast execution of such programs.

3.2. Core architecture

Thus we have designed the architecture which is outlined in figure 3. The protocol processor also contains accelerators for checksum calculations, which are not shown in the figure.

The packet is received through the dynamic buffer, which normally holds only one word of data. The processor architecture is general and the word length, *l*, can be chosen at design time. In our prototype we have used *l=32* bits as the word length. The dynamic buffer can hold several words of data if that is required. It is controlled from the instruction. Attached to the dynamic buffer is also a

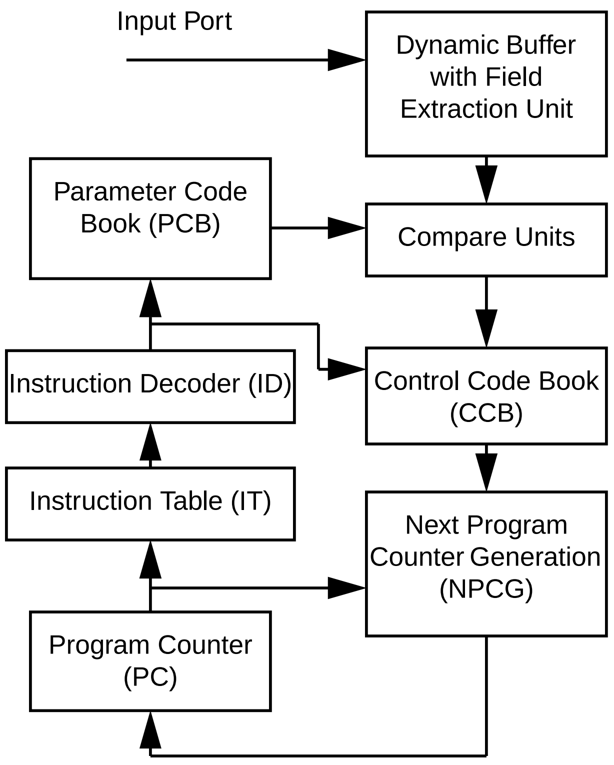

Figure 3: Protocol processor core overview

field extraction unit, which selects a field from the buffer content. That field is forwarded to the compare units (CU), an array of *n* comparators. The reference values for these comparators come from the parameter code book (PCB). The PCB has an output of *n* words. Internally it is a lookup table, with *k* lines of each *n* words. A pointer from the instruction decoder (ID) selects which line to forward to the output.

The output from the compare units is a vector of *n* bits, in which each bit represents a match or a non-match. These *n* bits are used to select an output from the control code book (CCB). The CCB is another lookup table, that contains relative jump addresses. It consists of *k* lines of each *n* addresses. The same pointer that is used for the PCB is also used for the CCB in order to select one of the *k* lines. The *n* output bits from the compare units select which address to forward to the next program counter generation (NPCG). The NPCG calculates the next program counter value, which is used by the program counter (PC). The PC is a simple register, which is updated every clock cycle. The output is used to select an instruction from the instruction table (IT). The IT is a lookup table, which contains the instructions for the protocol processor.

Since the only register in the chain is the PC, a complete switch-case-case... statement can be executed in one clock cycle as long as the branches are not more than *n*. An additional default branch can also be handled.

The program is split up into three parts, which are all contained in the processor core. The IT contains the core instruction, the PCB contains the reference values, and the CCB contains the relative jump addresses.

3.3. Example

An example is used to illustrate how this works. Let us look at the C like code in figure 4.

```
switch (ethType) {
  case 0x0800:
    processIP();
    break;
  case 0x0806:
    processARP();
    break;
  case 0x8035:
    processRARP();
    break;
  default:
    handleException();
    break;
}
```

Figure 4: Code example

That code will be executed in one clock cycle in the protocol processor and it works like this. The instruction specifies a line in the PCB and CCB, say line 3. On line 3 in the PCB the reference values 0x0800, 0x0806, and 0x8035 are stored. In the CCB on line 3 the corresponding relative jump addresses are stored, so let us assume that this instruction is on line *j* in the IT. And let us further assume that the routine processIP() starts on line *j+14*, processARP() starts on line *j+23* and processRARP() starts on line *j+32*. Then the content of line *i* in the CCB is 14, 23, and 32. In this example *n=3* is thus sufficient. When deciding on the value of *n*, one has to consider the switch-case-case... statement with the most branches in all the programs one wants to be able to execute in the protocol processor. Figure 5 shows the example operation, when ethType = 0x0806 and *n*=4.

3.4. Programming interface

The protocol processor core is programmed through a special configuration interface before the operation starts. During operation the configuration can be changed at any time by writing new values to a certain position in a lookup table. For example if a new connection is opened and a new UDP port will accept packets, this port number is stored on the adequate position in the PCB. The µC is

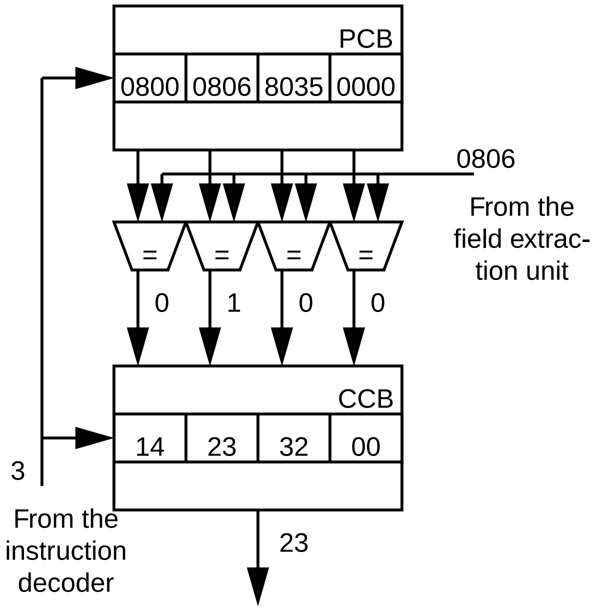

Figure 5: Example operation of PCB and CCB

responsible for the configuration and the incremental updates. It therefore has a copy of the configuration in its memory.

3.5. Implementation

The protocol processor core instruction set has been implemented in a cycle true C++ simulator, that simulates the structure of the architecture. It uses binary configuration data for the lookup tables. The assembly instruction set is finished and some key components have been implemented in VHDL.

4. The accelerators

The protocol processor core, that has been described so far only handles the field matching and decision making part of the packet reception processing. The other parts, checksum calculation and memory management, are handled by accelerators. The main reason for dividing the processing like this is that the field matching and decision making uses only the packet headers, but the checksum calculation and the memory management uses the whole packet. Since the core and the accelerators use the same data at the same time, the protocol processor is best described as a MISD architecture in the processor classification scheme of Flynn [6]. The program flow of the different parts is also completely different. The field matching part consists of almost only if-then-else and switch-case-case... statements, but the checksum calculation consists of bitwise operations, which are the same for each word of data, much like signal processing algorithms. Memory management implies storing the payload in the correct memory location based on the result of the field matching operations. So that later on the application can access the payload directly.

4.1. Communication

The accelerators have to communicate with the protocol processor core in order to synchronize the operation. This is done through synchronous control signals and flags. The protocol processor core gives start signals and other control signals to the accelerators. When they have finished their operation they set flags, which the protocol processor core can evaluate.

4.2. Implementations

Implementations for a CRC-32 accelerator [7] and a TCP/UDP checksum calculation unit [8] have already been done as a part of our first generation protocol processor [9]. These show that the accelerators can operate at high throughput rate, approximately 10 Gb/s and do not require much silicon area.

5. Evaluation and discussion

In this section we discuss the performance of the protocol processor and provide an estimation. We also compare our solution to other work.

5.1. Critical path analysis

The protocol processor does not use the pipelining technique, since that would not allow the single cycle execution of if-then-else and switch-case-case... statements. This implies that the program memory has to be kept small in order to be able to store it inside the processor core, in the three lookup tables IT, PCB, and CCB. The critical path is the longest circular path in figure 3. That is, from the PC, through the IT, the ID, the PCB, the CU, the CCB and the NPCG then back to the input of the PC. The ID does not contribute to the critical path, since the assembly instruction format is chosen so that the pointer for the PCB is directly stored in the instruction word. The delay through the IT is that of the multiplexer, that selects the correct instruction. Similarly, the delays through the PCB and the CCB also are multiplexer delays. All of these from the control signal input to the output. Some of the control signals have high fanout, which increases the delay. Except from the multiplexer delays there is also the delay in the compare unit, which is a comparison and in the NPCG, which is an addition.

The maximum program size influences the delay. If we define m as the number of instructions in IT and p as the number of bits in the instruction word in addition to k, l, and n as defined earlier, we can describe the delay easily. First there is an m-to-1 multiplexer of width p. Then a k-to-1 multiplexer of width $n*l$. Then a comparison of l bits. Then an n-to-1 multiplexer of width $\log_2(m)$. Finally there is the addition of $\log_2(m)$ bits.

We have a test program that checks the Ethernet destination address, demultiplexes IP and ARP packets, checks the IP destination address, demultiplexes UDP and TCP packets and checks the UDP port. For that test program, we use 32 bit words and 32 bit instructions. Further we have 6 lines in the PCB and the CCB, we use 27 instructions and we use maximally 3 branches in a case statement. This leads to the following configuration, k=6, l=32, m=27, n=3, and p=32. If we round up to the nearest potential of two we get k=8, l=32, m=32, n=4, and p=32, which leaves some headroom to execute more complex programs.

With the above configurations, the delay estimations are shown in table 2. When scaling to a 0.18 micron technology an expected performance gain of a factor higher than 2 indicates that the protocol processor core can handle 10 Gb/s data streams.

Unit	Delay contribution	In 0.35 micron technology
IT	32-to-1 multiplexer	2.1 ns
ID		0 ns
PCB	8-to-1 multiplexer	1.7 ns
CU	32-bit comparison	1.0 ns
CCB	4-to-1 multiplexer	0.6 ns
NPCG	5-bit addition	1.1 ns
Total		6.5 ns

Table 2. Delay estimations

5.2. Area estimations

The area of the protocol processor core is to large extent used by the lookup tables. With the parameters from the previous subsection, we get 1k bits in the IT, 1k bits in the PCB, and 160 bits in the CCB. This occupies less than 1 mm^2 in a 0.35 micron technology, or approximately 0.2 mm^2 in a 0.18 micron technology.

The rest of the circuitry demands less area and doubling the above values gives an upper limit, which still assures a small processor core. Therefore the limit on the sizes of the lookup tables do not come from the area usage, but rather from the size-dependent delay in the critical path. Small area is requested since the protocol processor is integrated with other components on the same silicon die.

5.3. Related work

As mentioned earlier, there is not much work done for network terminals, however some of the work on routers, that deal with port processing has some similarities with a network terminal.

Field-programmable port extenders (FPXs) [10] provide flexible port processing through FPGA technology. The processing functionality is described in VHDL. FPXs are aimed for ATM processing and it is hard to compare the results to our estimations. It can however be concluded that they use a totally different approach to solve a related problem.

In [11] an architecture is presented together with the design methodology. The specification lacks however some information and there are no performance figures, thus it is hard to compare it to our processor.

The storage area network (SAN) community has realized the same problem as have we [12] and efforts are going on to design TCP-Offload Engines (TOEs) [13]. Still no architectures have been presented. A TOE is more complex than our protocol processor, which actually could be used as a building block for TOEs. In [14] TCP offloading for transmission at Gigabit speed is described, but no hardware architecture is presented, also, the requirements for transmission are quite different from those for the reception. So again it is hard to make a fair comparison to our processor.

6. Conclusions and future work

The network terminals soon will become the bottlenecks in computer networks. New efficient solutions are needed. We have designed a protocol processor for packet reception that works in-line with the data flow and has a novel architecture, which is needed for real-time operation of field matching and decision making. The processor is estimated to be able to handle 10 Gb/s data streams, when implemented in a modern CMOS technology.

Ongoing work finalizes the instruction set coverage verification and starts implementing the architecture on register transfer level. We have also started to integrate the protocol processor in a system, in order to demonstrate its functionality in a real environment.

There is a need for a good programming interface, consisting of an assembler, compiler, and debugger, but non of this has been initiated yet.

References

[1] US Patent application no. 09/934372

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

[3] 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

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

[5] R. Merritt, “iWarp interface spec could bust IP bottlenecks”, *EETimes*, on the www: http://www.eetimes.com/story/OEG20011207S0091

[6] M. J. Flynn, “Very High-Speed Computing Systems”, *Proceedings of the IEEE*, Vol. 54, No. 12, December 1966, pp. 1901-1909

[7] T. Henriksson, H. Eriksson, U. Nordqvist, P. Larsson-Edefors, D. Liu, “VLSI IMPLEMENTATION OF CRC-32 FOR 10 GIGABIT ETHERNET", in Proceedings of *ICECS 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", to appear in Proceedings of *Design and Diganostics of Electronics, Cricuits and Systems*, April 17-19, 2002, Brno, Czeck Republic

[9] T. Henriksson, U. Nordqvist, D. Liu, "Specification of a configurable General-Purpose Protocol Processor", in Proceedings of *CSNDSP 2000*, pp. 284-289, July 18-20, 2000, Bournemouth, UK

[10] F. Braun, J. Lockwood, and M. Waldvogel, “Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Hardware”, *IEEE Micro*, Vol. 22, No. 1, Jan/Feb 2002, pp. 66-74

[11] M. Attia and I. Verbauwhede, “Programmable Gigabit Ethernet Packet Processor Design Methodology”, *European Conference on Circuit Theory and Design*, Vol. III, pp. 177-180, August 28-31, 2001, Espoo, Finland.

[12] K. Voruganti and P. Sarkar, “An Analysis of Three Gigabit Netowrking Protocols for Storage Area Networks”, *International Conference on Performance, Computing, and Communication*s, 2001, pp. 259-265

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

[14] H. Bilic, Y. Birk, I. Chirashnya, and Z. Machulusky, “Deferred Segmentation for Wire-Speed Transmission of Large TCP Frames over Standard GbE Networks”, *Hot Interconnects 9*, 2001, pp. 81-85