Linköping Studies in Science and Technology

Dissertation No. 813

Intra-Packet Data-Flow Protocol Processor

Tomas Henriksson

INSTITUTE OF TECHNOLOGY

LINKÖPINGS UNIVERSITET

Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden

Linköping 2003

Linköping Studies in Science and Technology

Dissertation No. 813

ISBN: 91-7373-624-4

ISSN: 0345-7524

Printed by UniTryck, Sweden, 2003

Abstract

Protocol processing is the bottleneck in high-speed computer networks. Many network processors have been suggested for switches and routers. Protocol processing in terminals has other characteristics than the processing in switches and routers. Therefore a new type of processor is desirable for terminals.

I define that the protocol processing tasks in a terminal can be partitioned into intra-packet tasks and inter-packet tasks. It is suitable to use two processors that work in a coarse-grained pipeline to implement this partition. The partition crosses over the protocol layers, which have previously been used for partitioning the protocol processing implementations.

The inter-packet tasks are irregular and can be efficiently executed by a traditional von Neuman style processor. The intra-packet tasks can be sub-partitioned into regular tasks and irregular tasks. A novel processor architecture for intra-packet tasks has been developed that makes use of the sub-partitioning by executing the regular tasks in accelerator units and the irregular tasks in a data-flow core unit.

Our core architecture is different from traditional processors because it does not operate on data stored in a memory. Instead it operates in a data-flow fashion directly on the data that is received on the network interface. Thereby no load and store operations are necessary. So the packets are already processed to a large extent when the payload is written into memory. This saves both data memory bandwidth, program memory size, processing time and power consumption. Most of the packets that should be discarded never have to be stored in memory at all.

The data-flow processing also creates some problems since the program flow has to be perfectly synchronized with the incoming data stream, which prevents the use of a pipelined processor architecture. The performance requirements are fulfilled by splitting up the program into three parts and by using a dedicated program memory storage architecture. A standard cell implementation of the processor indicates support of a data flow of more than 10 Gigabit/s. The imple-

mentation can be used for network terminals as well as for port acceleration in switches and routers.

The total silicon area of the processor including the program memory is small (0.4 mm^2 in 0.18 micron standard cells) and accommodates for an increased number of ports on a real time Ethernet switching chip or the integration of the protocol processing off-loading onto the host processor chip in terminal equipment such as desktop computers.

The accelerator for cyclic redundancy check (CRC) has been implemented with standard cells and manufactured in a 0.35 micron process technology. The chip has measured performance of more than 5.76 Gb/s.

The most significant contribution of my research is the new data-flow processor architecture, which has been proven in a fully functional demonstrator system. The demonstrator system is based on my novel partition of protocol processing tasks into intra-packet tasks and inter-packet tasks. A dual processor architecture, implemented in an FPGA, receives, synchronizes and plays back an audio stream which is sent in UDP/IP packets over fast Ethernet.

The processor architecture will be very important for any processor operating on a data flow. There are possible improvements to the processor, for example a detailed analysis between data width and flexibility will support trading off the internal width and program memory size. Future work also includes investigating in what other areas, besides networks, the processor architecture can be successfully used.

My most important contributions are:

- The partition of protocol processing
- The data-flow processor architecture

Preface

The research project that is the basis for this thesis was started in 1998. In August 1999, I joined the project and continued to work on the for the time being proposed architecture. The work on that architecture was presented in my licentiate thesis [1.3] in December 2001. In the beginning of year 2001, I started simultaneously to work on a new processor architecture. That work is the focus of this thesis. The thesis is based on the results presented in the following papers and manuscripts:

- Tomas Henriksson, "In-Line CRC Calculation and Scheduling for 10 Gigabit Ethernet Transmission", in Proceedings of Swedish System-on-Chip Conference 2002, March 18-19, 2002, Falkenberg, Sweden
- Tomas Henriksson and Ingrid Verbauwhede, "Fast IP Address Lookup Engine for SoC Integration", in Proceedings of Design and Diganostics of Electronics, Circuits and Systems, pp. 200-210, April 17-19, 2002, Brno, Czeck Republic
- Tomas Henriksson, Niklas Persson and Dake Liu, "VLSI Implementation of Internet Checksum Calculation for 10 Gigiabit Ethernet", in Proceedings of Design and Diagnostics of Electronics, Circuits and Systems, pp. 114-121, April 17-19, 2002, Brno, Czeck Republic
- Tomas Henriksson, Ulf Nordqvist and Dake Liu, "Embedded Protocol Processor for Fast and Efficient Packet Reception", in Proceedings of International Conference on Computer Design, pp. 414-419, September 16-18, 2002, Freiburg, Germany
- Tomas Henriksson and Dake Liu, "Novel ASIP and Processor Architecture for Packet Decoding", in Workshop of Application Specific Processors Digest, pp. 25-31, November 19, 2002, Istanbul, Turkey
- Tomas Henriksson and Dake Liu, "Implementation of Fast CRC Calculation", in Proceedings of Asia South Pacific Design Automation Conference, pp. 563-564, January 21-24, 2003, Kitakyushu, Japan

- Tomas Henriksson, Dake Liu and Harald Bergh, "Method and apparatus for general-purpose packet reception processing", United States Patent Application 20030039247, February 27, 2003
- Tomas Henriksson and Dake Liu, "100 Gb/s CRC Generation Circuit in 0.13 Micron Standard Cells", manuscript, submitted
- Tomas Henriksson and Dake Liu, "Off-loading the Off-loading processor: A new way to separate protocol processing", manuscript

During the period from December 2001 to May 2003 other work has been done that was not mentioned in the licentiate thesis and is not covered in this thesis. Some of that work was presented in the following publications:

- Henrik Eriksson, Tomas Henriksson, and Per Larsson-Edefors, "Full Custom vs. Standard Cell Based Design - an Adder Comparison", in Proceedings of Swedish System-on-Chip Conference 2002, March 18-19, 2002, Falkenberg, Sweden
- Ulf Nordqvist, Tomas Henriksson and Dake Liu, "Configurable CRC Generator", in Proceedings of Design and Diagnostics of Electronics, Circuits and Systems, pp. 192-199, April 17-19, 2002, Brno, Czeck Republic
- Henrik Eriksson, Tomas Henriksson, Per Larsson-Edefors and Christer Svensson, "Full-Custom vs. Standard-Cell Design Flow - An Adder Case Study", in Proceedings of Asia South Pacific Design Automation Conference, pp. 507-510, January 21-24, 2003, Kitakyushu, Japan
- Tomas Henriksson and Dake Liu, "Fully programmable Video Stream Decoding", to appear in Proceedings of Swedish System-on-Chip Conference 2003, April 8-9, 2003, Eskilstuna, Sweden
- Tomas Henriksson, Daniel Wiklund and Dake Liu, "VLSI Implementation of a Switch for On-Chip Networks", to appear in Proceedings of Design and Diagnostics of Electronics, Circuits and Systems, April 14-16, 2003, Poznan, Poland

Acknowledgments

This thesis would not have been completed without the support from several persons. I especially want to thank the following:

Professor Dake Liu for being a good advisor, for long and interesting discussions and for giving me the opportunity to work in this interesting area.

Fellow Ph.D. student Ulf Nordqvist for almost 4 years of shared research and interesting discussions.

Professor Ingrid Verbauwhede at UCLA for letting me work in her group for 5 months during 2001 and for the cooperation on IP address lookup architectures.

Professor Christer Svensson for being my formal advisor in the beginning of my studies and for many interesting discussions.

Professor Per Larsson-Edefors and Ph.D. student Henrik Eriksson, now at Chalmers University of Technology, for cooperations and interesting discussions.

Niklas Persson and Kristoffer Martinsson for their hard work as master's thesis students in projects closely related to my protocol processor architecture.

Fellow Ph.D. students Daniel Wiklund, Mikael Olausson, Eric Tell and Sumant Sathe for good cooperations and valuable knowledge sharing.

Further more I would like to acknowledge past and present members of the Electronic Devices Division and the Computer Engineering Division at Linköpings universitet and of the IVGroup at UCLA for creating a nice an stimulating working environment.

Finally I would like to thank my industry cooperators, SwitchCore Corp. (Kenny Ranerup and Peter Tufvesson), Ericsson AB (Per Sundin and Per Holmberg), Ericsson Research (George Liu), and VIA Sweden (Harald Bergh).

The thesis work was sponsored by the Swedish Foundation for Strategic Research (SSF) through the Integrated Electronic Systems Programme (INTELECT) and through the STRINGENT center.

Contents

1

Introduction

In the beginning of the 21st century computer networks are used extensively throughout the whole society, at companies as well as in private homes. There is a constant urge to get higher capacity and better service. This thesis discusses the processing requirements and resources in computer networks and presents a new processor architecture that can be used to provide efficient network terminals as well as high capacity switches and routers. A list of acronyms and a glossary of technical terms can be found at the end of the thesis in appendices A and B.

1.1 Background

General purpose processors can execute any kind of applications. This comes to the cost of high overhead, such as complex instruction decoding and inefficient data movement. In network terminals, for example desktop computers, the main central processing unit (CPU) is used to process the packets for the network. This has been a viable solution for 100 Mb/s Ethernet, but limits the network performance when higher capacity networks, such as Gigabit Ethernet and 10 Gigabit Ethernet are used.

By examining a set of applications it is possible to design a processor that is better suited for the task of executing similar applications than a general purpose processor. However the specially designed processor may not be able to

execute all kinds of applications and if it is the performance for other types of applications can be very poor. Flexibility and overall performance is traded for better performance for a certain type of applications.

The processing of network protocols has proven to consist of a limited number of processing task types and therefore it makes sense to design specialized processors for them. The reason for doing that is that general purpose processors do not fulfill the requirements on performance and power consumption that are put on the systems in which protocol processing is necessary. An alternative could be to design fixed function application specific integrated circuits (ASICs), but those are specific for only one protocol stack and cannot accommodate for future protocol or service changes. That is a problem since redesign and manufacturing of new circuits is both time-consuming and very costly.

The aim of my thesis work was to develop a new and efficient software programmable processor architecture for processing network protocols of received packets in a network terminal.

1.2 Computer Networks

Computers are interconnected by computer networks. The languages that the computers use to exchange information are called protocols. The communication is packet-based, that means that all information is exchanged in units of limited size which are called packets. A computer program, normally referred to as an application, that has some information to send to another computer (or other equipment attached to the network) compiles a packet with the information, by adding headers and sometimes trailers, according to the protocol that is used. The header and trailer contain for example address information and checksums. Figure 1.1 shows the general structure of a packet.

Each protocol has its own specific header and trailer structure. Some are of fixed length and others are of variable length. The communication between two computers uses several protocols on top of each other, so called protocol layers. In the ISO-OSI description there are 7 protocol layers [1.1], but computers that use the TCP/IP protocol stack normally use only 4 of them, [1.2].

Whenever a packet is received from the network, the receiving computer must check that the packet has the correct destination address and if so which appli-

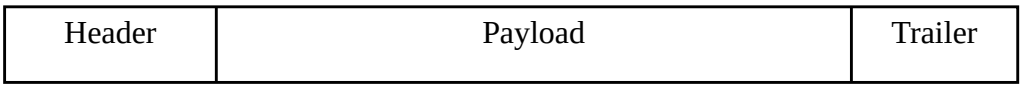


Figure 1.1: General structure of a packet in a computer networks

cation it should be delivered to. To make sure that no bit errors have been introduced, the checksums must also be calculated and compared to the received checksums. In the routers and switches that the packet passes through on its way from source to destination, a forwarding decision must be made and the packet header must be modified, which implies checksum recalculation. An example of a network structure is shown in figure 1.2. The term terminal refers to any kind of equipment that is attached to a computer network and is not a switch or a router. It may be for example a desktop or laptop computer, a network attached printer, an advanced mobile telephone, a set-top-box for network distributed TV, a backup device or an image IP telephone.

A more detailed review on computer networks is available in my licentiate thesis [1.3]. For complete coverage on the topic [1.1] and [1.2] are suggested. It is of importance to understand how a computer network operates for the further reading of this thesis, but it is not further described in this thesis.

1.3 Processors

Processing of data can be handled in many ways. One way to describe the processing elements is to categorize them dependent on their flexibility. As already mentioned there are fixed function circuits, ASICs, which cannot change the

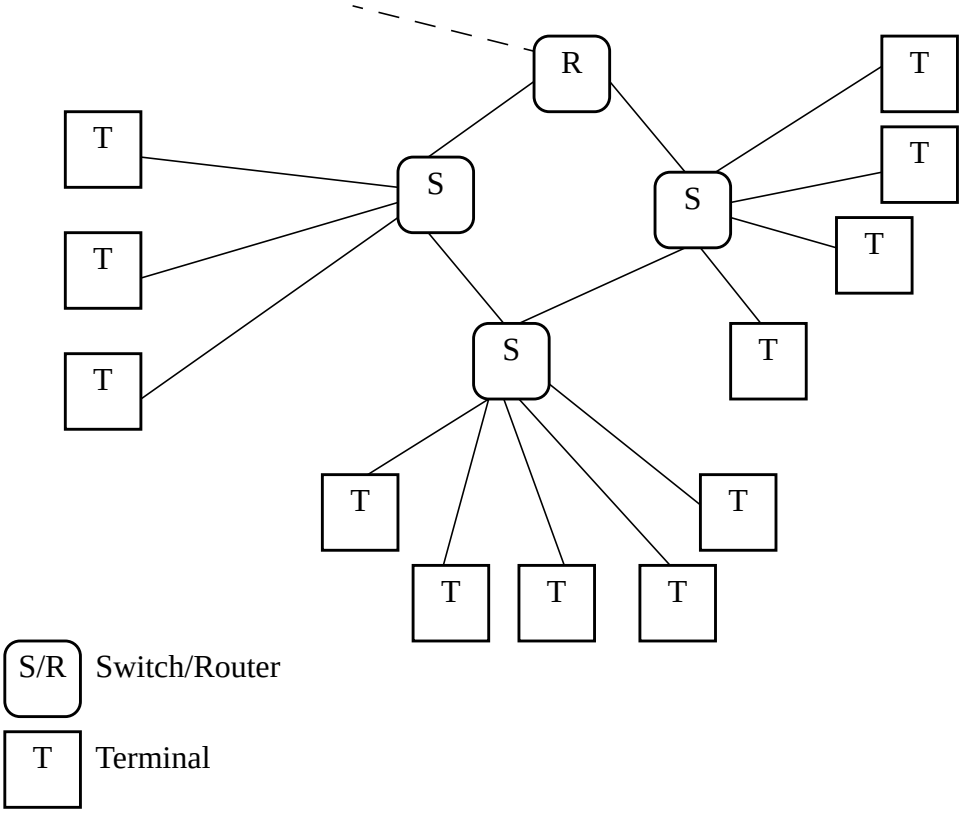


Figure 1.2: Example structure of a computer network

functionality after manufacturing. For the circuits that have flexible functionality, there are two types of flexibility that are common, configurability and programmability. A circuit is called configurable if the functionality can be changed after manufacturing and more than one clock cycle is needed to reconfigure the functionality. Common types of configurable circuits are field programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs).

A circuit is called programmable if the functionality can be changed every clock cycle. This is what we normally refer to as a processor. The processor is defined by the instruction set architecture (ISA) and the register file (RF). This is what is called the programmer's view of a processor and that is the interface between the hardware that constitutes the processor and the software that can be executed on the processor, [1.4].

The development of processors has led to pipelined superscalar and very long instruction word (VLIW) processors, that are capable of executing several instructions every clock cycle. The number of instructions per cycle (IPC) is limited by data dependencies and stalls due to cache misses and branch miss-predictions. The level of abstraction that describes the clock cycle based timing of a processor is called the microarchitecture.

General purpose processors can execute all kinds of applications. Application specific processors on the other hand can execute a limited domain of applications. In application specific processors flexibility is traded for less complexity or higher performance, see figure 1.3. A more detailed discussion on application specific processors is available in chapter 2.

Traditionally processors execute programs stored in memory on data which is also stored in memory. This is however not necessarily the only way that a processor can work, as is described in chapter 6.

1.4 Thesis Organization

The rest of this thesis describes in more detail how processors work and how to chose the best type of processor for a certain type of applications. A more detailed study on common network protocols and how they are processed is also provided. Then the new processor architecture that I have developed is described along with prototype implementations and a system demonstrator.

In chapter 2, application specific processors are described and the impact of accelerator units is described and discussed. Chapter 3 deals with network processors (NPs) and describes the particular requirements that are placed on those. In chapter 4 a detour to implementation of IP route lookup is done. In

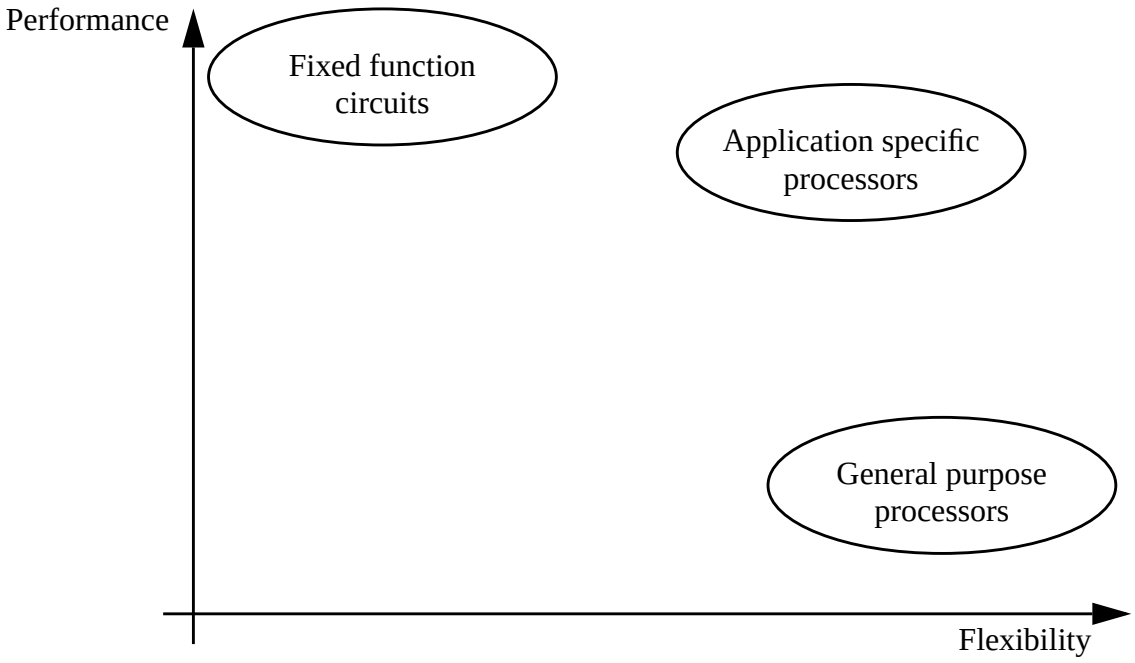


Figure 1.3: Comparison of different types of processing elements for a certain application domain

chapter 5 my novel way of partitioning protocol processing in a terminal is described. Chapter 6 contains the description of the unique intra-packet processor (intra-PP) architecture and discusses the design methodology as well as the ISA and the microarchitecture, which includes several accelerator units. In chapter 7, a VLSI implementation of the intra-PP is discussed and estimations on performance and silicon area are presented. The accelerator units are further described in chapter 8 and implementation estimations or measured results are provided for the checksum accelerators. Chapter 9 presents the demonstrator system that has been built around the intra-PP. Finally in chapter 10, I conclude the findings during this thesis work and suggest directions for future work in this area.

1.5 Contributions

The most important contributions of the work that led to this thesis are:

- The novel and efficient protocol processor architecture, that is applicable to all types of data flow processing
- The novel partition of protocol processing in terminals

References

[1.1] Andrew S. Tannenbaum, “Computer Networks”, Prentice-Hall, Inc., ISBN 0-13-349945-6, 1996.

[1.2] W. Richard Stevens, “TCP/IP Illustrated, Volume 1: The Protocols”, Addison Wesley Longman, Inc., ISBN 0-201-63346-9, 1994.

[1.3] Tomas Henriksson, “Hardware Architecture for Protocol Processing”, Licentiate Degree Thesis, Linköping Studies in Science and Technology, Thesis No. 911, December 2001, ISBN: 91-7373-209-5.

[1.4] John L. Henessy and David A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufman Publishers, Inc., ISBN 1-55860-329-8, Second Edition 1996.

2

Application Specific Processors

Application specific processors are normally characterized by a special instruction set and are then referred to as application specific instruction set processors (ASIPs). There are however other ways to specialize a processor than merely changing the instruction set.

2.1 Instruction Sets

The instruction set architecture (ISA) of a processor constitutes the interface between hardware and software. An application is mapped to a sequence of instructions from the instruction set. The sequence may contain jumps, loops and other program flow control mechanisms.

2.1.1 Representations

There are two common ways to describe such a sequence, which is called a program. The binary format consists of only 1s and 0s. This is the format of the instructions when they are stored in memory and the format that the processor can decode and execute.

In the assembly format each type of instruction has a short operation code assigned to it, for example addition can be assigned the code ADD. The assembly format provides better readability for humans than the binary format. Together with the operation code also the operands are specified, which can be registers, memory locations or constants carried in the instruction word. Normally not all types of instructions can use all types of operands. In assembly format the code segments can be assigned labels, which can be used to express jumps. Constants, registers and memory addresses can also be assign labels, which can be used to make the code more readable. There is a one to one correspondence between the assembly format of a program and the binary format of the same program. A piece of software called assembler can convert between the two formats.

2.1.2 Compiling

When an application is mapped to the instruction set, there are several things to consider if high performance is requested. Normally the application is described in a high level programming language such as C or C++. Then the mapping to the instruction set is the task of a compiler.

An application can be modelled with a data flow graph (DFG), which describes the computation that must take place. In a DFG, the inputs, intermediate results and outputs are described only by variables and it is not specified where they are stored, just like in high level programming languages. Many processors can only compute on values that are stored in the register file. Therefore register allocation is an important task when mapping an application to an instruction set. If there are more intermediate results than registers available at any time some values must be stored in memory. That is called spilling values to memory and special spill code is inserted in the program. This requires additional instructions and thereby decreases performance. So it is important to have an adequate number of registers in the register file.

Another thing that is important when mapping the application to the instruction set is the instruction selection. A part of an application can be mapped to more than one sequence of instructions. Normally as few instructions as possible are desirable, since that minimizes the code size. Small code size is important especially if cache memories are used. Then small memory footprint of the code reduces the number of cache misses and thereby increases overall performance. Small code size can however have bad impacts on the execution time. This is due to the fact that processors execute instructions in a pipelined fashion. So several instructions are in the pipe at the same time, some executing and others are fetched and decoded. If the code contains a small loop many

instances of the same instruction can be in the pipe at the same time, but this leads to data hazards which cause pipeline stalls and thereby decreased performance. By using techniques such as software pipelining and loop unrolling the performance is increased to the cost of increased code size and larger memory footprint.

The third thing that is important when mapping an application to an instruction set is the instruction scheduling. For simple one-way in-order processors and for very long instruction word (VLIW) processors the scheduling determines the final execution order of the instructions. For more advanced out-of-order superscalar processors the final execution order is determined in runtime, but the instruction scheduling still has a large impact on the execution order. The instruction scheduling should try to avoid data dependencies, structural and control flow hazards.

The compilers normally deal with the three tasks of register allocation, instruction selection and instruction scheduling. Traditionally they are optimized based on heuristics one at a time, but new approaches include integrating all three tasks into an integer linear programming problem. The mapping from a high-level programming language to an assembly description is non-trivial and heuristics are used to find a good mapping. Optimality is however almost never achieved. The performance is dependent not only on the ISA and the register file of the processor, but also on the details of the interior of the processor, the microarchitecture.

2.2 Microarchitecture

The ISA describes the interface between the software and the hardware, but it does not describe how the instructions are executed in hardware. That is the task of the microarchitecture.

2.2.1 Instructions Per Clock

In a high performance processor the execution of several instructions per clock (IPC) is supported. This is managed by having several (normally 4 or 8) parallel execution units. Although the theoretically possible IPC is as high as the number of execution units, it is not possible to achieve that IPC for a real application in a general purpose processor. For an application specific processor where the applications are known it is possible to design a processor that can maintain a relatively higher IPC than a general purpose processor.

There are two common ways to achieve parallelism in a processor. Very long instruction word (VLIW) processors let the compiler determine the final execution order and bundle operations together that shall be executed in parallel.

Superscalar processors handle the bundling of instructions at runtime in hardware.

2.2.2 Pipeline

The execution of an instruction includes the steps of fetching the instruction from memory, decoding the instruction, fetching the operands, executing the instruction and writing back the result. This whole procedure takes long time and limits the maximum clock frequency for the processor. To be able to run the processor at a higher clock frequency, a technique called pipelining is used. That means that each step of the instruction is executed in one or more clock cycles. The intermediate results are stored in pipeline registers and therefore the execution of a new instruction can be started every clock cycle.

In every clock cycle only a step or a part of step has to be executed, the clock frequency can be increased dramatically compared to that of a non-pipelined processor. Almost all processors make use of the pipelining technique. Most processors have only a few pipeline stages, but high performance processors use up to 20 pipeline stages [2.1].

If a conditional branch occurs, there will be instructions that have started to execute that really should not be executed. Then the execution of those must be reversed. It is called flushing the pipe because all intermediate results in the pipeline registers are simply discarded.

2.2.3 Cache Memories

By the use of pipelining, the clock frequency of the processors has increased dramatically, but the memories that store the program code and the data have not kept up with that performance increase. Cache memories are used to increase the speed of the memories. Cache memories are smaller and faster than the main memory. They contain parts of the program code and parts of the data at a time. Whenever the processor needs program code or data that is not stored in the cache, that will be fetched from the main memory. The cache memories fetch a line from the main memory at a time, that implies that instructions or data close to the requested item will also be fetched.

Cache memories are feasible because of spatial and temporal locality of a program and data. Temporal locality of a program means that a program will use the same instructions many times, for example in a loop. Temporal locality of data means that the same variables will be accessed several times within a short time frame. Spatial locality of a program means that instructions close to each other will be used after each other. This is the case for a sequential part of a program. Spatial locality of data means that data items close to each other will be

used in the same short time frame, for example when operating on a vector or a matrix of data.

The spatial and temporal locality of programs and data combined with the fact that cache memories work with lines makes sure that most requested instructions and data will be in the cache memory when they are needed. The processor pipeline assumes that this is the case and the number of pipeline stages for a fetch is adjusted for the fetch time from the cache memory. If the requested instruction or data is not in the cache there will be a cache miss and the processor has to wait until the instruction or data has been fetched from the main memory. A cache miss can cause up to several hundreds of clock cycles delay.

Cache memories are very complex and it is not the aim to cover the design of them here. More information on cache memory organization and replacement strategies can be found for example in [2.2].

2.2.4 Conditional Branches

As already mentioned, conditional branches can cause the pipeline to be flushed. This has a negative impact on performance. A branch predictor is normally used to avoid having to flush the pipeline. For example in a loop it is highly likely that the conditional branch will jump back to the beginning of the loop. This will happen in every case except when the loop is done. By using a branch predictor the most probable continuation of a program can be selected and the pipeline does only rarely have to be flushed.

As with cache memories, branch predictors are complex and many alternative implementations exist. It is also not the intention to cover them in detail in this thesis. It is merely concluded that no matter how good a branch predictor is, it will make occasional miss-predictions, with lost execution cycles as a consequence. This is an important observation, which is used in chapter 6.

2.3 Profiling and Benchmarking

Obviously there are many different ways to design processors for a group of applications and each design team would come up with different solutions if they could start with the same design goal and the same domain of applications. A way to measure how good a processor is at a certain application domain is benchmarking.

2.3.1 Benchmarking Suites

There exist benchmarking suites for various types of application domains, for example general purpose programs, digital signal processing programs, multi-

media programs, and several more for other types of embedded systems. A benchmarking suite consists of applications and input data. The most important results of a benchmark simulation is the number of clock cycles that are used by the simulated processor to execute the benchmarking suite. The execution time is calculated as number of clock cycles divided by the actual clock frequency that the processor is able to run at. A simple way to compare the performance of two processors is to run the same benchmarking suite on the two processors. Compute the execution times and then compare the execution times and determine which processor has the highest performance.

If the ISA has been fixed and various microarchitectures are evaluated, the IPC count alone describes the relative performance of the processors, assuming that they can run at the same clock frequency.

There are some things that have to be considered when running the benchmarking simulation. For example if the same application is run twice the cache memory will already contain most of the instructions and data in the second run, since benchmark applications are normally small. This can influence the performance significantly. It is also important to consider an operating system running in the background and other applications that compete about the processing resources. For some application specific processors only one application runs at a time without any operating system, but most processors actually run both operating systems and other applications simultaneously. This influences the outcome of the benchmarking.

2.3.2 Instruction Profiling

The simulators that are used for running the benchmarks normally can provide much more information than the number of execution cycles. The instruction profiling is one of the most interesting things when designing an ASIP. The profile captures how frequently each type of instruction is executed. It can also give information on how frequently combinations of instructions are executed. If some combination is very frequent it is advisable to create a new instruction that executes the combination in only one instruction if possible. This will of course only help if the intermediate results are not consumed anywhere else in the application.

Other interesting information includes the rate of pipeline stalls, cache miss rate, the occupancy for each execution unit and branch miss-prediction rate. All of these help tuning the microarchitecture. That means that the performance can be increased without changing the instruction set.

2.4 Optimization

When the performance of a processor is too low, optimizations of the processor are necessary. Even if the performance is adequate optimizations regarding silicon area, power consumption or program code size may be necessary.

General purpose processors have total programmability and power consumption as constraints and performance as the design goal. ASIPs normally have performance and power consumption as constraints and programmability as the design goal.

2.4.1 ISA Optimization

Changing the ISA can have huge impact on the performance of a processor. If specialized instructions for operations that occur frequently in the program are included in the instruction set these operations will need less code, possibly create less intermediate results and often those operations can execute in one clock cycle instead of needing several clock cycles. One typical example of this technique is the use of multiply and accumulate (MAC) instructions in digital signal processors (DSPs).

Processors that have very wide data paths, for example 32 or 64 bit wide can make use of single instruction multiple data (SIMD) instructions to increase performance. SIMD instructions here mean that the data path is split into several units of 8 or 16 bit width. These sub-units execute the same instruction on one piece of the data word each. SIMD instructions are useful for example in image and video applications where the pixels are normally specified with only 8 bits.

When the ISA of a processor is changed, both the hardware and the software, that is the compiler, must be updated.

2.4.2 Register File Optimization

As mentioned earlier on, increasing the size of the register file can improve the performance of a processor because more values can be stored in the register file and less interaction with the memory is needed. If the number of registers is increased more bits in the register address in the binary representation of the instruction word are needed to specify the operands. If not enough bits are available the instruction word must be prolonged, which causes the program to need a larger memory.

Adding more registers has however bad impacts on the silicon area, power consumption and can also increase the cycle time, which means that the maximum clock frequency is decreased. So a performance trade-off can be made on

how many registers optimally to have in the register file for a set of applications.

If only some instructions have the problem that the additional bits for specifying all new registers cannot fit in the instruction word, those instructions can be restricted to use only some part of the register file. When all instructions can use all registers the instruction set is called orthogonal. Compilers have problems with non-orthogonal instruction sets.

Another reason for making the instruction set non-orthogonal can be that there is a limitation of the number of ports to the register file. Principally each execution unit needs two read ports and one write port from and to the register file. Having many ports on a register file causes the same problems as having many register in the register file, that is increased area, power consumption and cycle time. Therefore it is important to limit the number of ports. One way to do that is to make use of clustered register file. The execution units are assigned to one of the clusters and can only operate on the registers in that cluster. There are specific move instructions available to exchange data between the clusters. Using a clustered register file puts more constraints on the compiler.

2.4.3 Microarchitectural Optimization

The microarchitectural optimizations aim at increasing the IPC and the clock frequency of the processor. These are the two factors that contribute to the performance once the ISA and register file size have been fixed.

The IPC is increased by additional execution units, more advanced scheduler, better branch prediction and larger cache memories. All of these changes increase the complexity of the processor. Increased complexity almost always means increased design time, increased silicon area and increased power consumption. So there is a price to pay for the increased performance. A factor that also has to be considered is the cycle time, which determines the maximum clock frequency.

The clock frequency is normally increased by adding more pipeline stages, so that the cycle time can be reduced. The clock frequency can also be increased when moving to more modern manufacturing technologies with smaller dimensions, but that is outside the scope of this study. When more pipeline stages are added, a branch miss-prediction will have a higher cost and therefore the IPC may be reduced. So high IPC and high clock frequency are contradictory in some sense and a detailed trade-off analysis is necessary to reach the optimum.

The optimum is dependent on the characteristics of the applications. DSPs, for example, execute signal processing algorithms which contain few branches.

The DSPs therefore normally have deep pipelines and run at high clock frequency. A branch miss-prediction is costly but occurs rarely.

Microcontrollers is another group of application specific processors. They are intended and optimized for control applications, which do not require much computation, but have a complex program flow and depend on many external signals. Therefore often special registers are available, which are accessible on the pins for inputs and outputs to and from the processor. The pipeline of this type of processors is normally shallow since the many conditional branches in the applications are hard to predict and therefore would cause significant cost in pipeline stalls in a processor with a deep pipeline.

2.5 Processor Acceleration

Sometimes enough performance can not be achieved by the techniques described in the previous sections. Then there exist a number of alternatives on how to modify the processors.

2.5.1 Multi Processor Systems

If the application contains independent parts or parts with very little interaction, a simple solution can be to split the application into two or more programs that execute in parallel in a multi processor system. Each processor can be optimized for the tasks that it has to execute.

The problem with multi processor systems is that the tasks on the processors are not totally independent and information must be exchanged between the processors. This can be done by message passing or shared memories. It has proven hard to design compilers for multi processor systems.

2.5.2 Accelerator Units

Another way to increase performance is to use accelerator units. Accelerator units can be used independently of if the program can be split into several parts or not. Accelerator units can be tightly or loosely coupled with the processor core. A tightly coupled accelerator unit typically has a short and predictable execution time and it is accessed by special purpose register for inputs and outputs. For example the accelerator unit can be triggered by the processor loading a value into its input register and 3 clock cycles later a result is available in its output register, which the processor can read.

A loosely coupled accelerator unit can have data dependent execution time and signal completion via an interrupt or let the processor poll its status. The

inputs and outputs can be interchanged via memory mapped registers between the processor core and the accelerator unit.

The accelerator units off-load regular tasks from the processor. One common type of accelerator units are direct memory access (DMA) controllers, that move blocks of data between different memories in a system.

I define a regular task as a task where the data path does not need to change every clock cycle. For applications with only irregular tasks, accelerator units cannot be used efficiently.

References

[2.1] Doug Carmean, “Pipeline Depth Tradeoffs and the Intel® Pentium® 4 Processor”, presented at Hotchips 13, 19-21 August, 2001, Palo Alto.

[2.2] Harald S. Stone, “High-Performance Computer Architecture”, Addison-Wesley Publishing Company Inc. 1993, ISBN 0-201-52688-3

3

Network Processors and TCP Off-load Engines

One special type of processors are the network processors (NPs). Many NPs are actually highly integrated computing systems consisting of several processing units and memory blocks. TCP off-load engines (TOEs) is another type of processing elements for computer networks. To be able to understand the design decisions for these kinds of processors the characteristics of protocol processing must be well understood.

3.1 Characteristics of Protocol Processing

Protocol processing is the broad name for any type of processing that is involved with a computer network. For example packet forwarding in routers, packet generation in terminals, packet filtering in fire walls, packet encryption in security equipment and so on. To understand the characteristics of the processing it is therefore necessary to divide the protocol processing into narrower categories. However, there is one characteristic that is common to all protocol processing and that is that there is little or no data locality. Similarly as for signal processing each piece of information is processed once and then never used

again. Therefore data caches have little or nothing to contribute to protocol processing systems.

In the rest of this section and in sections 3.2 and 3.3 the focus will be on protocol processing in terminals. More specifically the processing that occurs when a packet is received from the network. Some of the processing in routers will be further discussed in section 3.4 and a case study on IP route lookup is presented in chapter 4.

3.1.1 Protocols in Local Area Networks

The dominating local area network (LAN) protocol is Ethernet. Ethernet exists in many different shapes and is constantly developing. Ethernet is a layer 1 and 2 protocol, meaning that it specifies the physical layer and the data link layer. On top of Ethernet, Internet protocol (IP) is the dominating network layer protocol. The only competitor to IP is its own newer version 6, IPv6. It has been discussed for many years when IPv6 will take over the dominating position of IP, but so far all predictions have been wrong and IP continues to be the most used protocol.

Layer 1, 2 and 3 consisting of Ethernet and IP is the common foundation for most LANs. A LAN is connected to the rest of the Internet via one or more routers. In the wide area networks other layer 1 and 2 protocols than Ethernet and used, but that is not the intention of this study to cover them. On layer 3 IP is used in all networks, IP is the language that facilitate communication between various types of networks and systems.

Layer 1 and 2 provide point to point communication, although in early versions of Ethernet all terminals could overhear all communication. Broadcasts and multicasts are also possible and frequently used for example by the address resolution protocol (ARP). IP provides the ability to route packets over multiple point to point connections and thereby provides communication between terminals which are not physically directly connected.

Layers 1, 2 and 3 specify the behavior per packet. The processing of one packet is not dependent on other packets, except from some special cases. All information that is needed for the processing is self-contained in the headers of each packet. Although options for segmentation and reassembly are provided in IP they are almost always avoided. Flow control exists in Ethernet, but is optional in implementations.

3.1.2 Processing Tasks in Ethernet and IP

The processing of Ethernet and IP upon reception of a packet consists of the principal tasks of calculating checksums, checking addresses and demultiplex-

ing the packet stream based on identifiers in the packet header. The checksums are of two types, cyclic redundancy check (CRC) and two's complement addition, also called the Internet checksum.

Addresses must be checked in both Ethernet and IP protocol headers. The Ethernet addresses are 48 bits long and the IP addresses are 32 bits long. Special addresses are reserved for broadcasts and multicasts. Those addresses must be treated in all terminals.

Packet demultiplexing on the Ethernet layer consists of investigating an 16 bit type field. The type field specifies what protocol type is used by the packet carried in the Ethernet frame. Packet demultiplexing on the IP layer consists of checking the 8 bit protocol field. The protocol field specifies which protocol is carried in the IP packet.

3.1.3 Processing Tasks in Transport Layer Protocols

On top of layer 3, the network layer, there is the transport layer. There are two protocols commonly used on the transport layer, transmission control protocol (TCP) and user datagram protocol (UDP). The processing at the transport layer is similar to the one at layers 1 to 3, except that TCP requires state handling since it offers reliable connections which are maintained with sequence numbers and acknowledgments.

The processing is thus of two main kinds, the checksums which require regular computation intensive processing and the rest which requires control dominated data dependent processing.

3.1.4 Protocol Processing Implementations

The traditional way of implementing the protocol processing in a computer system is to have a network interface card (NIC) that buffers the incoming packets and processes the Ethernet protocol. Then a direct memory access (DMA) mechanism transfers the packet into main memory and the host processor can then handle all the other protocols. In many operating systems the network and transport layer processing is done in the OS kernel and the applications have to manage layers 5 to 7 if they exist.

In traditional implementations of protocol processing the operating system, task handling and data movement also influence the protocol processing performance significantly [3.1]. That is not because of fundamental properties of the protocols, but rather it is a consequence of the implementation and can be circumvented.

3.2 Parallelization of Protocol Processing

It was realized many years ago that the protocol processing could be a bottleneck in a system and various approaches have been applied to speed up the protocol processing. The main idea has been to parallelize the processing on several processors.

In [3.2] four ways to parallelize the processing are described and that is a good survey of all approaches that have been suggested in the literature. A farm of identical processors was used to execute the protocol processing. The key point is how to partition the processing among the processors. Table 3.1 summarizes the characteristics of the four partition schemes.

3.2.1 Processor-per-Message

The first partition, processor-per-message, suggests having one processor to take care of each packet that arrives. This has the advantages of coarse-grain parallelism and good load balancing. It also gives flexibility in the number of processors allocated for protocol processing. If the network is not heavily used for some period of time some processors can be allocated for other purposes during that time.

Drawbacks are that the connection states must be shared among several processors. Especially for protocols with a complex connection state, like TCP, this makes the processor-per-message partition costly.

3.2.2 Processor-per-Connection

The processor-per-connection partition means that each connection is assigned to one processor and that that processor has to handle all the tasks for all packets that belong to that connection. The advantage over the previous

Partition scheme	Shared data	Drawback	Advantage
Processor-per-Message	Connection state	Shared connection state	Flexibility, good load balancing
Processor-per-Connection	None	Bad utilization and saturation	No shared data
Processor-per-Protocol	Packet	Shared packet data	Specialization is possible
Processor-per-Task	Packet and connection state	Shared packet data and connection state	Possible latency reduction

Table 3.1: Characteristics of partition schemes

approach is that the connection state does not need to be shared among several processors. The disadvantage on the other hand is that bursty connections will saturate one processor while other processors might be idle. They cannot off-load the saturated processor because they cannot access the connection state. It often occurs that connections are bursty, because that is be behavior of file transfer.

3.2.3 Processor-per-Protocol

Processor-per-protocol is a partition where each processor is assigned to one protocol. Advantages are that the processors could be specialized and the code size can be small for each processor. The problem is that the packets must move from processor to processor and the granularity is too small for protocols with small processing needs.

3.2.4 Processor-per-Task

Finally, processor-per-task implies that each processor executes a single task of one or more protocols. This partition has finer granularity than the previous one and requires that both the packets and the connection states can be shared by several processors. That turns out to be too costly in a practical implementation. The advantages come from that several tasks can be executed in parallel and a theoretical reduction in processing latency for one packet is possible.

All approaches except processor-per-connection have overhead and lock contention since the processors must use some shared resources. The processor-per-message approach was found to be the best in the environment used in [3.2]. Also in [3.3] a processor-per-message architecture was the favored approach.

In chapter 5 a novel way to parallelize and partition protocol processing is presented.

3.3 TCP Off-load Engines for Network Terminals

In [3.4] a way of reducing the overhead associated with protocol processing was suggested. By using packet accelerators, that modify the packet headers, the headers are reduced in size and less complex to process. The processing is partitioned into two phases, pre-processing, which does not modify the connection state and post-processing, that updates the connection state. This partition improves the packet header processing latency. For example headers can be predicted in the pre-processing phase and some processing can thereby occur even before the packet is received.

Because of the total dominance of IP, TCP and UDP during the last decade, it is no longer possible to add protocol accelerators, that modify the packet headers. Instead it has been realized that processing the existing protocols at efficiently as possible is what matters. The partitioning into phases is applicable, but not as efficient, also for protocol processing implementations that do not modify the packet headers. That concept is further developed in this thesis in chapter 5.

The main approach to do so is to off-load the protocol processing from the host processor to the NIC. Soon the NIC will also be integrated closer with the main memory and host processor since the peripheral component interconnect (PCI) bus is not keeping up with the increases in network bandwidth. The two standards that are of interest are Gigabit Ethernet and 10 Gigabit Ethernet. It was estimated that file transfer with Gigabit Ethernet requires 20%-60% of the processing power of a top of the line processor [3.5]. For 10 Gigabit Ethernet the processor will be overloaded and limit the performance of the network.

This is a known problem and there exist approaches to solve it. In [3.6] a method is shown on how to modify the logical interface between the host processor and the NIC. The socket layer in the host processor is replaced by a simple pair of memory buffers. All the protocol processing is thereby moved to the NIC. This allows for much less processing in the host processor. It requires however major changes in the OS and a modified version of the Linux OS was presented.

More pragmatic approaches have been taken by companies such as Alacritech, Trebia and iReady, which all have presented NICs for Gigabit Ethernet with protocol processing capabilities. These devices are referred to as TCP Off-load Engines (TOEs).

3.4 Network Processors for Switches and Routers

In the last section of this chapter a detour from the terminals is taken and NPs for switches and routers are described. It leads to the presentation of the novel architecture for IP route lookup in the next chapter.

3.4.1 Basic Router Functionality

The difference between a switch and a router is that a switch handles only one layer 2 protocol, for example Ethernet. A router on the other hand can have ports which use different layer 2 protocols. The switch uses the layer 2 addresses (hardware addresses, MAC addresses) to do the packet forwarding. A router on the other hand uses layer 3 addresses, that is the IP addresses. In the rest of this section routers will be considered.

The requirements on routers are quite different from the ones on terminals. In switches and routers as little interaction with the packets as possible is normally desired. The fundamental task is only to forward each packet on the correct output port. To do so there is some processing that is needed.

There is one change that must be made to all packets. The IP header must be modified, since the time to live (TTL) field should be decreased by one at every router. This causes the IP header checksum to be updated as well.

The other major task that a router does with all packets is to extract the destination address and look up in a table on which port the packet must be forwarded. The next chapter deals with the problems of IP route lookup in detail and presents a novel architecture that executes IP route lookup extremely fast.

3.4.2 Router Architectures

Router architectures have evolved over the years and two types are still existing. Most routers are constructed around a switching backplane, which is a cross connection for moving data packets. On this backplane the line cards are attached. Each line card can have one or more ports, for example Ethernet, ATM, and ADSL. The processing is done on the line cards for incoming packets and when the output port is determined, by an IP address lookup, the packet is sent over the backplane to the corresponding line card, which buffers the packet until the output port is available. The other way to build switches or routers is to use a shared high-speed memory instead of having a switching backplane and local buffers at the output ports. This latter type of architecture can be integrated onto one single chip.

The fundamental functionality of a switch or router is simple, but the high requirements on performance make the designs complex. Another thing that contributes to the complexity is the increasing demand for quality of service (QoS). That means that different types of packets should be treated differently. For example packet streams with real-time constraints, such as streaming audio and video should get higher priority than file transfer and electronic mail packets, which do not suffer from small delays.

QoS requirements increases the processing load per packet significantly, since the content of the packets must be inspected in order to determine which type of packet it is. This is referred to as packet classification and normally 5 fields of the packet headers are used to do this classification. Once the packets have been classified they have to be treated differently, a task performed by a queue management system.

Another problem for routers and switches is the increasing demands on security in the computer networks. This has lead to many packets being encrypted.

When the routers need to do some inspection in the packets they much first decrypt the contents and then encrypt the packets again before forwarding them.

Totally this multitude of tasks for switches and routers has lead to the development of NPs, which are dedicated processors for handling the processing in switches and routers. Most NP systems consist of one main processor with several co-processors, for example for encryption, packet classification and queue management. There exist standardized interfaces for how to connect the co-processors to the main processor, [3.7]. The interfaces make the co-processors look like memory blocks to the main processor.

The programmability of the processors is varying. The constraints on the deigns is normally throughput and power consumption. Programmability is a design goal, that can be sacrificed so that the constraints are fulfilled. The switches and routers are real-time systems, where the packets must be processed as they arrive.

References

[3.1] Peter Steenkiste, “Analyzing Communication Latency using the Netcar Communication Processor”, ACM SIGCOMM Computer Communication Review, vol. 22, No. 4, pp. 199-209, October 1992

[3.2] Mats Björkman and Per Gunningberg, “Performance Modeling of Multiprocessor Implementations of Protocols”, IEEE/ACM Transactions on Networking, vol. 6, No. 3, pp. 262-273, June 1998

[3.3] Niraj Jain, Mischa Schwartz and Theodore R. Bashkow, “Transport Protocol Processing at Gpbs Rates”, ACM SIGCOMM Computer Communications Review, vol. 20, No. 4, pp. 188-199, September 1990

[3.4] Robbert van Renesse, “Masking the Overhead of Protocol Layering”, ACM SIGCOMM Computer Communication Review, vol. 26, No. 4, pp. 96-104, August 1996

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

[3.6] Philip Buonadonna and David Culler, “Queue Pair IP: A Hybrid Architecture for System Area Networks”, International Symposium on Computer Architecture 2002, pp. 247-256, June 2002, Anchorage, Alaska

[3.7] Harmeet Bhugra, “LA-1: Standardizing the Look-Aside Processor Interface”, CommsDesign, on the www: http:www.commsdesign.com/story/OEG20020917S0022

4

IP Route Lookup Implementation

In this chapter the focus is IP route lookup and the architecture that was presented in [4.1]. First the task of IP route lookup is described. Then implementation alternatives are discussed and finally the novel architecture is presented.

4.1 Route Lookup

IP route lookup is defined as the process of finding an action pointer associated with an IP destination address. The action pointers are stored in a routing table.

An IP address consists of 32 bits. These are organized as two parts, network identifier and host identifier. All computers on a LAN have the same network identifier. When Internet was designed three types of networks were allowed, class A, class B and class C networks. Class A networks had 8 bit network identifiers, class B networks had 16 bit network identifiers and class C networks had 24 bit network identifiers. Therefore class C networks can only have a maximum of 256 hosts (actually only 254 because 2 addresses are reserved for broadcast). This led to problems as companies and organizations grew. They

then received another class C network, with a network identifier independent of the previous they had, because the similar ones were already taken.

In a router only the network identifier part of the destination address is considered, unless the packet has reached its final LAN router. For each network identifier a new entry is needed in the lookup table. Since companies and organizations now had several network identifiers, several entries were needed in all routers. The class-based addressing was saturated and a new approach was taken. The classes were removed from the Internet and instead each entry in the routing tables has a mask that specifies how many bits are constituting the network identifier, called classless interdomain routing (CIDR). This has led to a decreased growth rate of entries in the routing tables, but still the routing tables in core routers can contain several hundreds of thousands of entries.

So each entry in a routing table consists of three interesting fields, network identifier, mask and action pointer. The action pointer specifies what to do with the packets that match the corresponding entry. A simplified way of looking at it is as the output port identifier although more information is normally contained, such as next hop IP address. The lookup task is to compare the IP destination address of the incoming packet with all the network identifiers in the routing table and find the best match. The best match is defined as the match with the highest number of bits in the mask. For example if entry *n* is the tuple <130.236.54.0, 24, action 3> and entry *m* is <130.236.0.0, 16, action 4> and these are the only two matches for a packet, with destination address 130.236.54.3, then entry *n* is the best match since 24 is greater than 16.

A mathematical way to describe the lookup problem is to view the address space as a continuos line, where each entry specifies an interval, see figure 4.1. All intervals are either disjoint or one is contained in the other. Intervals that

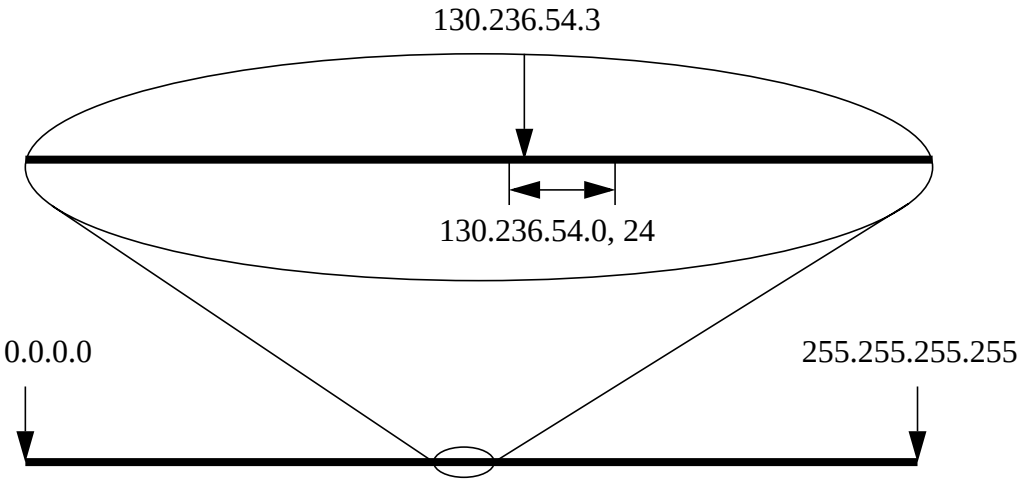


Figure 4.1: Mathematical view of the IP address space

partly overlap do not exist because of the structure of the network identifiers. The IP destination address of a packet corresponds to a single point on the continuos line. The point can be contained in one or more intervals. There is a default entry, which has the whole line as the interval. If several intervals contain the point, the shortest interval corresponds to the best match.

When looking at the processing requirements there is only one operation which is needed and that is the compare operation. Based on the results of the comparison, decisions must be made. Another aspect that must be considered is that the routing tables may change rapidly, in some cases table updates as often as every 100 ms have been reported.

4.2 Implementation Alternatives

There exist three main categories of IP route lookup implementations, software, special hardware and ternary content addressable memories (TCAMs). The TCAMs offer massive parallelism and execute all comparisons in parallel. Then prioritize logic selects the best match. This is done by storing the entries in order of mask length in the TCAM and simply selecting the match with the lowest address. The TCAMs can be pipelined and support high performance. One problem arises when new entries must be added to the TCAMs, but efficient ways handle that have been presented [4.2]. The key idea is to sort the entries in order of mask length and save some unused space between every block of entries with a certain mask length. Thereby new entries independent of mask length can easily be added in the TCAM. If there is no available space between two blocks where a new entry must be added, one entry from the neighboring block can be moved to the other side of that block to create an empty space for the new entry.

The downside with TCAMs is that a TCAM cell is about twice as expensive as an ordinary SRAM cell in terms of silicon area, power consumption and access time.

For the software and special hardware implementations it is impossible to do a sequential search through the table and achieve reasonable performance. It is also not possible to have large table enough that can be directly addressed with the destination address. The size of the table would be 2^{32} entries. Each entry would only need to contain the action pointer, since the network identifier and the mask are included in the address. The action pointer could be 1 byte and thus a memory of 4 GB would be sufficient. Although large this is certainly a reasonable size for an off-chip memory. The problem that arises is that of table updates. An update of an entry with a mask of 8 could lead to a maximum of 2^{24} entries (=16 Mentries) that need to be updated. Instead a mixed compare

and directly addressed table architecture is used. For software implementations it is important to have a small enough table so that it can fit in the cache memory. Several papers have been presented for fast and efficient IP address lookup, in [4.3] the various algorithms are surveyed.

4.3 Novel Architecture

The novel architecture, which I developed together with Professor Verbauwhede at UCLA during the spring and summer of year 2001, is an attempt to use an ASIP to perform the IP address lookup. Since there is only one operation, compare, there is no need for an instruction word because the operation can be built into the architecture. The architecture consists of several processing stages. Each stage consists of an SRAM and a processing element (PE), see figure 4.2. The processing stages work together in a pipelined fashion, see figure 4.3. Each processing stage contains a part of the routing table. *W* is the number of bits in the address, 32 for IP. The number of bits that are processed in every stage is *k*. Therefore there are totally *W*/*k* stages.

Stage 0 gets the whole address (n=32) as input. In stage 0 the *k* most significant bits (msbs) of the address are used to address the SRAM and the output is a pointer that combined with the next *k* bits of the address are used to address the SRAM in stage 1. The following stages work the same way until the result is found in an SRAM. Then the output of that SRAM is the result, that is the action pointer, and the *is_pointer* is set to 0. In the remaining stages the result is simply forwarded from the input to the output.

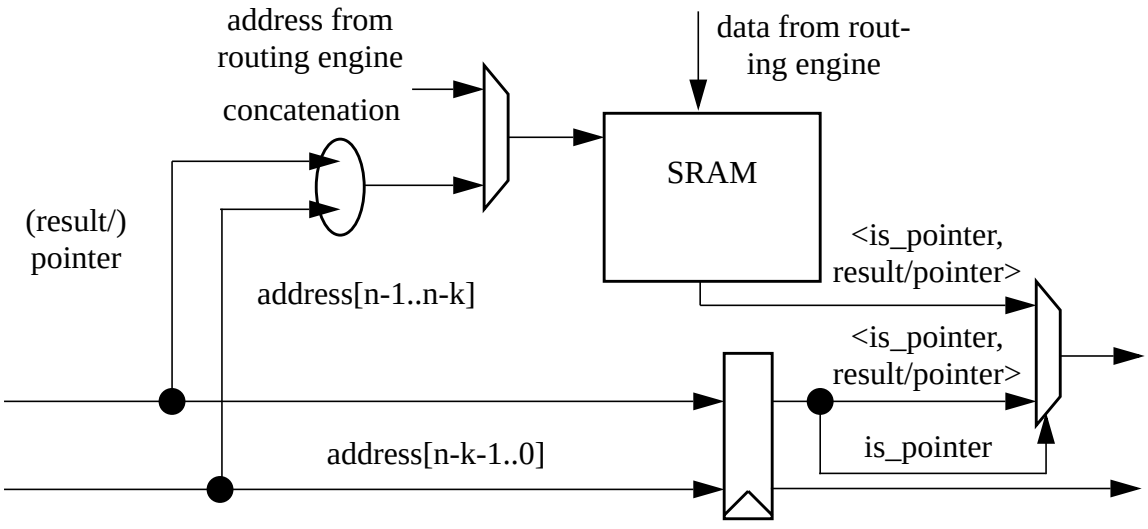


Figure 4.2: Processing element and SRAM in the novel architecture

4.4 Performance Evaluation

Since the processing stages are pipelined, the maximum performance is very high. The critical delay consists of one SRAM read access in series with two 2-input multiplexer delays.

The more interesting aspect is how much memory that is required. This was investigated by using publicly available routing tables from IPMA of May 23rd 2001 [4.4]. Five different routing tables are available, in table 4.1 the results for the smallest table, mae-east, with 16416 routing table entries are presented. The unit of the values are number of memory entries. A memory entry is a tuple <*is_pointer, result/pointer*>. As expected the memory requirements increase when k increases, since the memory is used in chunks of 2^k. It can also be seen that when $k=2^x$ the memory requirements are relatively smaller than otherwise. This is because the bits in the address exactly matches the number of stages. For all other values of k the last stage will have less than k bits to process. This increases the amount of required memory. The most promising values of k are 2 and 4, since they give good trade-offs between number of stages and memory requirements. Similar analysis were made for the other tables and they show similar results.

Unfortunately the number of entries at the different stages are varying with several orders of magnitude. It will be the stage with the largest memory requirement that will decide the minimum clock cycle period. The size of the largest memory, e_{max}, will also decide how many bits, m, that are necessary for the *result/pointer* field in each entry, assuming that the result can be expressed with less bits than the pointer. The memory address consists of $m+k$ bits and thus $2^{m+k} >= e_{max}$. For example, if $k=4$, $e_{max}=298320$ for mae-west (the largest table available), and $m>=15$. Setting $m=15$ would allow for an e_{max} of up to

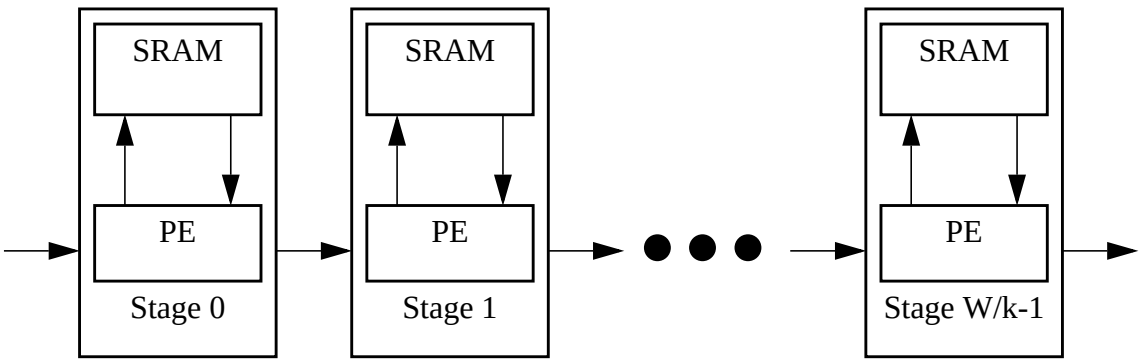


Figure 4.3: The PEs work together in a pipelined fashion

2^{15+4}=524288 and therefore *m*=15 could handle routing tables with more entries than mae-west. Exactly how large routing tables can be handled is dependent of the exact distribution of entries in the routing table.

Having *m*=15 is certainly reasonable, since each entry in the memory would consist of 16 bits, 1 bit for the *is_pointer* field and 15 bits for the *result/pointer* field. With 2^{15+4} entries, the largest memory requires 1 mega byte (MB). It should also be noticed, that the memory with the most entries normally does not have to have that many bits for the pointer since the next stage has a lot fewer entries. However, for the sake of regularity it is reasonable to make each stage

Stage	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10
0	2	4	8	16	32	64	128	256	512	1024
1	4	16	48	160	608	1728	6272	21248	76288	274432
2	8	40	216	1328	8576	52736	258048	713728	2384896	5517312
3	12	108	1192	13184	91264	298112	794496	4608	5632	1024
4	20	332	6592	44608	172416	1152	640			
5	38	1072	22816	86208	320	64				
6	54	3296	37264	288	32					
7	98	8064	49656	80						
8	166	11152	144							
9	298	18632	88							
10	536	21552	8							
11	978	26788								
12	1648	72								
13	2676	36								
14	4032	20								
15	5704	4								
16	5576									
17	7440									
18	9316									
19	9478									
20	10776									
21	12414									
22	13394									
23	14100									
24	36									
25	20									
26	18									
27	22									
28	10									
29	4									
30	2									
31	2									
Total	98882	91188	118032	145872	273248	353856	1059584	739840	2467328	5793792

Table 4.1: Memory Requirements for Mae-East, 16416 routing table entries.

identical. According to [4.5], a 2 MB 4-transistor SRAM can operate at 500 MHz. In [4.6] a 4K word (16 bit) 6-transistor SRAM has an access time of 1.21 ns, a larger memory will have slightly longer access time. It is also known that 2 input multiplexers have a propagation delay of about 0.15 ns. All these numbers are for the 0.18 µm generation of technology. It is probable that our architecture has a cycle delay of about 2 ns and since we handle one routing lookup request every clock cycle the maximum throughput is approximately 500MPackets/s. This seems to be an upper limit for the performance of IP address lookup, but no formal proof has been derived. With access times of 2 ns and minimum sized packets of 40 bytes, 160 Gb/s of worst case traffic can be supported.

4.5 Hardware Multiplexing

In order to decrease the differences between the memory requirements of the stages a trade-off with the throughput can be made. The idea of hardware multiplexing is to use each physical stage for two or more logical stages of the k-multibit trie search algorithm. This can be done in different ways, e.g. mirroring, serial stage reuse, and full loops. For the example of mirroring, see figure 4.4. Each physical stage will need an extra multiplexer in order to chose the correct input. There is also a need for a control part, which supports the select sig-

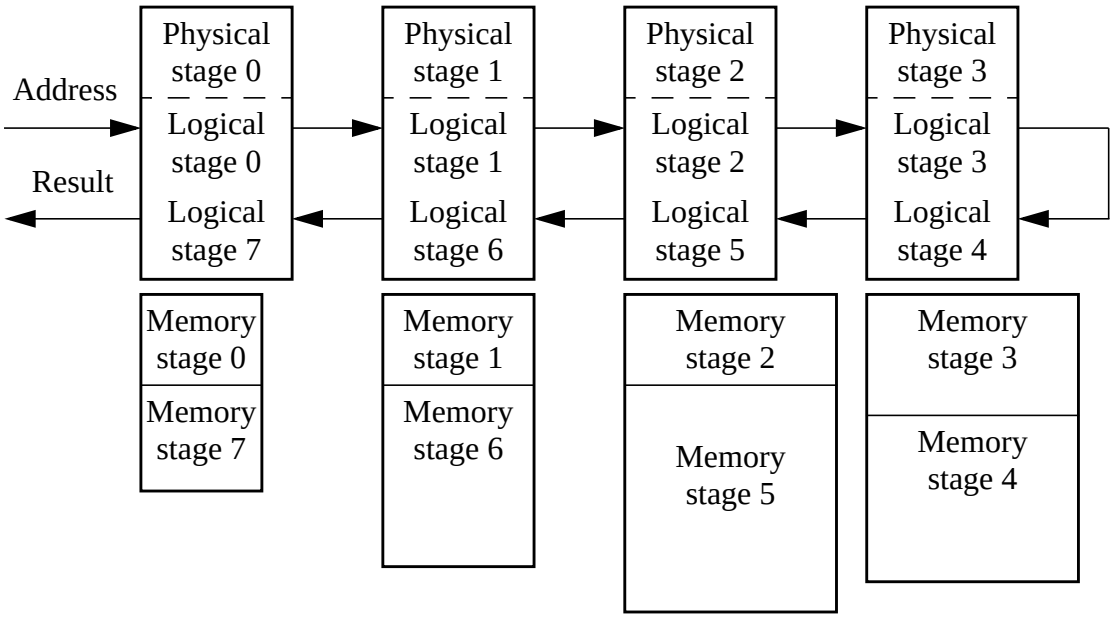


Figure 4.4: Hardware multiplexing by mirroring for k=4. The sizes of the boxes are not proportional to the required memory size, since the exact size depends of the content of the routing table, they only provide an overview of how the memory requirements add up on each physical stage, when hardware multiplexing is used.

nals to all multiplexers according to the multiplexing scheme selected. It is also roughly shown how the memory requirements add up on each physical stage.

To describe the three previously mentioned multiplexing schemes we define p_i as the i:th physical stage and l_i as the i:th logical stage. The tuple $<p_i, l_j>$ represents that logical stage j is executed on physical stage i. In tables 4.2-4.7 it can be seen how the scheduling for the three schemes work for *k*=4 with 2 and 4 logical stages being executed on each physical stage respectively. Each row is representing a task and each column represents a clock cycle.

It can easily be seen that all three schemes provide full utilization of the provided physical stages, both for double and quadruple multiplexing. The exact scheduling determines when results will be ready. This differs dependent on

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1, l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6	p_0,l_7							
2			p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6	p_0,l_7					
3					p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6	p_0,l_7			
4							p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	$p_2,l0$	p_1,l_6	p_0,l_7	
5									p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6
6											p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4
7													p_0,l_0	p_1,l_1	p_2,l_2
8															p_0,l_0

Table 4.2: Scheduling for Mirroring, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1, l_1	p_1,l_2	p_0,l_3	p_0,l_4	p_1,l_5	p_1,l_6	p_0,l_7							
2			p_0,l_0	p_1,l_1	p_1,l_2	p_0,l_3	p_0,l_4	p_1,l_5	p_1,l_6	p_0,l_7					
3									p_0,l_0	p_1,l_1	p_1,l_2	p_0,l_3	p_0,l_4	p_1,l_5	p_1,l_6
4											p_0,l_0	p_1,l_1	p_1,l_2	p_0,l_3	p_0,l_4

Table 4.3: Scheduling for Double Mirroring, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7							
2			p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7					
3					p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7			
4							p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7	
5									p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6
6											p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4
7													p_0,l_0	p_0,l_1	p_1,l_2
8															p_0,l_0

Table 4.4: Scheduling for 2 Serial Reuses, k=4

which scheme that is used and the results can be seen in the tables 4.2-4.7. Although necessary for the exact implementation it is of less general interest.

The fact of how the memory requirements add up, on the other hand, is most interesting, since now we have to consider the total memory on one physical stage as the performance limiting factor. As mentioned above, an extra multiplexer for the input selection and some control logic for selecting the correct input is also needed, so now the critical path will consist of 3 multiplexer delays + the memory access time. Table 4.8 shows the memory requirements per physical stage when the various multiplexing schemes are used on the mae-east routing table. For this routing table full loops are best when each stage is used twice, but double mirroring is better when using each stage 4 times. The purpose is obviously to minimize the maximum memory requirement per physical stage, the e_{max} when hardware multiplexing is used.

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_0,l_1	p_0,l_2	p_0,l_3	p_1,l_4	p_1,l_5	p_1,l_6	p_1,l_7							
2					p_0,l_0	p_0,l_1	p_0,l_2	p_0,l_3	p_1,l_4	p_1,l_5	p_1,l_6	p_1,l_7			
3									p_0,l_0	p_0,l_1	p_0,l_2	p_0,l_3	p_1,l_4	p_1,l_5	p_1,l_6
4													p_0,l_0	p_0,l_1	p_0,l_2

Table 4.5: Scheduling for 4 Serial Reuses, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7							
2		p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7						
3			p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7					
4				p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7				
5									p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6
6										p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5
7											p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4
8												p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3

Table 4.6: Scheduling for 2 Full Loops, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5	p_0,l_6	p_1,l_7							
2		p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5	p_0,l_6	p_1,l_7						
3									p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5	p_0,l_6
4										p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5

Table 4.7: Scheduling for 4 Full Loops, k=4

Using the right multiplexing scheme makes it possible to only slightly increase the maximum memory requirement per stage, e.g. for double mirroring, *k*=2 it increases from 26788 to 27300. We can approximate that the memory access time will remain the same and the addition of an extra multiplexer will only minorly influence the minimum clock cycle period. There will also be extra wiring needed and the output of some stages will have a higher fan-out. Totally, however, the multiplexed solution should be able to run almost as fast as the original configuration. It must be noticed that since each physical stage is used 2 or 4 times for each lookup request, the throughput will be decreased with the same factor.

Interestingly one can use the same configuration of, for example, 8 physical stages to implement a scheme with *k*=2 with multiplexing (mirroring and 2 full loops are almost equally good) and also a scheme with *k*=4 with no multiplexing. Since the memory sizes must be determined at the time of manufacturing this observation increases the flexibility of the architecture. A small routing table will fit in the memory with *k*=4 and the throughput can be kept maximal. When the routing table grows too large, *k* can be changed to 2 and multiplexing can be used so the same hardware can support increasingly larger routing tables, with a throughput decrease. Even larger routing tables will fit in with *k*=1 and double mirroring.

Building the memory in blocks can also allow post-manufacturing memory allocation to the stages by using configurable interconnects. This however introduces more hardware overhead and increases the access time.

Stage	Mirroring		Dbl. mirror		2 serially		4 serially		2 full loops		4 full loops	
	k = 2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4
0	8	96	19224	57888	20	176	168	14688	11156	44624	11560	46240
1	36	448	21964	87984	148	14512	12764	131184	18648	86368	19756	99632
2	76	87536	22700		1404	130816	78124		21592	1616	24908	
3	180	57792	27300		11360	368	132		26896	13264	34964	
4	27120				29784				404			
5	22624				48340				1108			
6	21928				108				3316			
7	19216				24				8068			
Total	91188	145872	91188	145872	91188	145872	91188	145872	91188	145872	91188	145872

Table 4.8: Memory requirements when multiplexing is used for mae-east, 16416 routing table entries

4.6 Scaling

When talking about the scalability of an IP address lookup implementation, two different views can be considered. First it is the scaling to more entries in the routing table and second the scaling to longer addresses, e.g. 128 bit IPv6 addresses.

To investigate the scalability of the routing table, all five routing tables from IPMA were used. The simulations were performed for k=1, 2, and 4. As already noticed $k=2^x$ gives good values and k should be kept fairly small in order to limit the size of the memories. The results can be seen in figure 4.5. The interesting fact is that for growing routing tables the e_{max} grows less than linearly for k=4. For k=2 the growth is approximately linear for the used routing tables. This is because many routing table entries share the same entry in the k-multibit trie. With an increasing number of routing table entries, this aggregation helps keeping the e_{max} limited. For larger k, e_{max} grows, but one must keep in mind that the number of stages decreases with a factor of k, so the total memory requirement does not grow as rapidly as e_{max}.

The second scalability issue is harder to investigate, since there is a lack of accessible IPv6 routing tables of sufficient size. IPMA provides us with one, which was used as of December 31st, 2000 23:14. It only contains 219 distinct prefixes and the simulation gives the results of table 4.9. It is hard to tell how well our architecture will handle large IPv6 tables. What is clear, however, is that the same principle is applicable to IPv6 and that the delay until the result is available will increase since more stages are needed. How the throughput is affected is dependent of how much e_{max} grows.

4.7 Updating and Further Extensions

4.7.1 Updating Issues

A routing table is constantly changing its content. The implementation must allow instant incremental changes to the routing table. The k-multibit trie implementation in software has an updating complexity of $O(W/k + 2^k)$ [4.3]. Our hardware implementation has the same complexity and more precisely, an update requires at most 2^k writes in the memory at each stage. By scheduling

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=16
e_{max}	4	16	40	128	384	1280	4480	14336	7864320
Total	510	812	1424	3168	7904	20160	47488	100352	20316160

Table 4.9: Memory requirements for IPv6 routing table

these writes so that they are pipelined in the same fashion as the lookup requests, only 2^k clock cycles will be lost for an update since W/k stages can be accessed in parallel. This all requires that the routing update engine has a copy of the memory structure, but since it is normally implemented on a general purpose CPU this is not a problem. The C-program used for simulation is capable of generating the necessary memory structures for configuration and updating of the forwarding engine.

Another tempting way of managing the updating is to make use of dual-port memories. These memories are almost as fast as single port, but require more area and totally smaller storage capacity is supported. However, the updating could be performed at the same time as a lookup request is serviced, which would allow zero time overhead for updates. An issue that has to be addressed

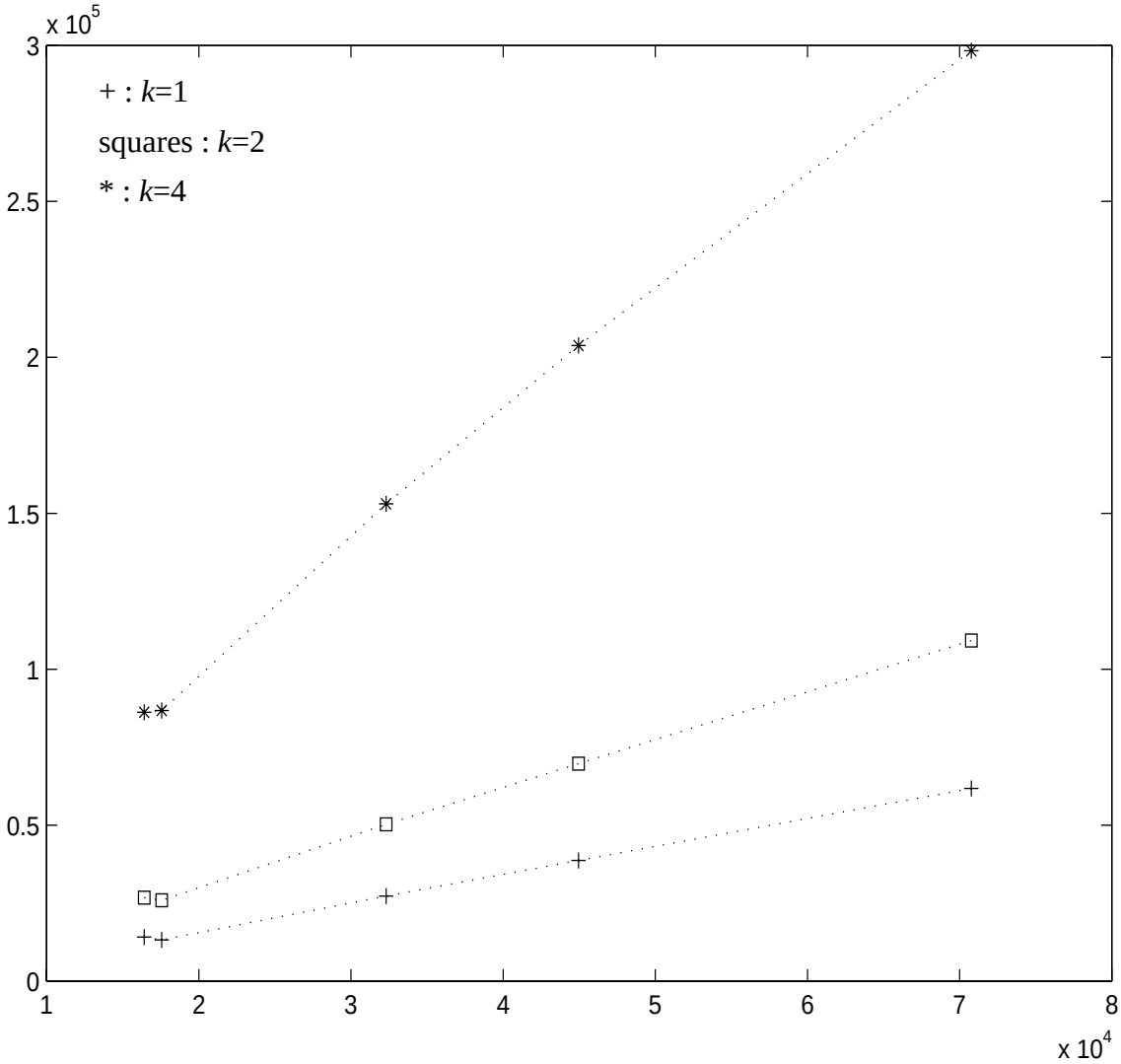


Figure 4.5: Maximum memory entries per stage, e_{max}, as a function of number of routing table entries, n

is that of the consistency. When a new rule is added this can require a maximum of 2^k writes in each stage as previously mentioned, therefore an update may be partial at the time of a lookup, which could lead to an incorrect forwarding decision.

4.7.2 2-Dimensional Search

IP packet forwarding based only on the destination address limits the router performance to best effort, since no differentiation of packets can be done. To improve this more fields from the packet header, such as source address, transport layer protocol, TCP/UDP source port and destination port are used. Each field can be seen as a search dimension. Normally the different dimensions have different search criteria, only IP destination address and source address can be assumed to have longest prefix match search. Transport layer protocol typically requires exact match and TCP/UDP ports often require range matching.

Thanks to the work done by Srinivasan et al. [4.7], it is known that the *k*-multibit trie search algorithm can be used also for 2-dimensional longest prefix match. Since our architecture is a direct implementation of this algorithm, it can also handle the search on source address as well as destination address. Further investigations, on how this affects the number of stages and the e_{max} have not been performed, since there is also a lack of such routing tables.

That concludes the study on routers. In the next chapter the focus is changed back to terminals.

References

[4.1] Tomas Henriksson and Ingrid Verbauwhede, “Fast IP Address Lookup Engine for SoC Integration”, Design and Diagnostics on Electronics, Circuits ans Systems 2001, pp. 200-210, April 17-19, 2002, Brno, Czeck Republic

[4.2] Devavrat Shah and Pankaj Gupta, “Fast Updating Algorithms for TCAMs”, IEEE Micro, vol. 21, No. 1, pp. 36-47, January/February 2001

[4.3] Miguel Á. Ruis-Sanchez, Ernst W. Biersack and Walid Dabbous, “Survey and Taxonomy of IP Address Lookup Algorithms”, IEEE Network, vol. 15, No. 2, pp. 8-23, March/April 2001

[4.4] “Internet Performance Measurement and Analysis Project”, on the www: http://www.merit.edu/ipma/

[4.5] Noda, K., Takeda, K., Matsui, K., Ito, S., Masuoka, S., Kawamoto, H., Ikezawa, N., Aimoto, Y., Nakamura, N., Iwasaki, T., Toyoshima, H., Horiuchi, T., “An Ultrahigh-Density High-Speed Loadless Four-Transis-

tor SRAM Macro with Twisted Bitline Architecture and Triple-Well Shield", IEEE JSSC, Vol. 36, No. 3, March 2001, pp. 510-515

[4.6] "Embedded Memories", http://www.umc.com/english/design/e.asp

[4.7] Srinivasan, V., Varghese, G., Suri, S., Waldvogel, M., "Fast and Scalable Layer Four Switching", Proceedings of ACM SIGCOMM 98, pp. 191-202

5

Partition of Protocol Processing

In chapter 3 some ways to parallelize the protocol processing were described. It is important to partition the protocol processing judiciously between the processing resources. My novel way to do that is to separate intra-packet tasks from inter-packet tasks.

5.1 Motivation for a New Partition Scheme

All the four partition schemes for protocol processing presented in chapter 3 suffer from either unbalanced workload among the processors or problems with shared data. The fundamental property of a good partition is little shared data. Shared data that is accessed through shared memory or communicated via message passing can create stall in the processing and thereby unpredictable processing performance.

The novel partition presented here aims at predictable performance, which accommodates for in-line processing. Thereby the packets can be at least partly processed before they are stored in memory. This saves memory bandwidth and power consumption.

The problem with unbalanced workload among the processors is harder to solve. To be able to process worst case packets at real-time there must be headroom over the average workload of the processors. One way to decrease the unbalancing in the workload is to specialize each processor for the tasks that are allocated to it.

5.2 Intra-Packet Tasks

Intra-packet tasks are the tasks that can be performed on one packet without any information about the packet flow. The intra-packet tasks do not have any side effects. They do not update any connection state variables, nor do they trigger any transmission of acknowledgments or other types of packets.

The purpose of separating the intra-packet tasks from the inter-packet tasks is that the intra-packet tasks can be executed without the need of any shared resources in a multiprocessor system. The previously suggested ways to split up the protocol processing are based on the concept of protocol layers. The layers are good for describing the logical behavior of protocols in computer networks, but limit the flexibility and thereby the performance of an implementation. The intra-packet tasks have no correspondence to the protocol layers and these tasks may reside in any of the protocol layers that are used.

Since there are no side-effect of the intra-packet tasks, these can be executed speculatively and out of order and easily be reversed if they should not have been executed. For example the header fields can be inspected before the checksum was proven correct. If later on the checksum proves to be incorrect the intermediate results of the header field inspection can simply be discarded. A way to look at it is that speculative execution that was miss-predicted does not need to be rewinded, but can simply be discarded instead.

The intra-packet tasks consist of regular checksum calculations and irregular comparisons. They are further highly predictable on a coarse task level. Principally they consist of calculating the checksum on the lowest layer. Then executing a couple of comparisons, for example checking an address and a higher layer protocol identifier. Then calculating the checksum on the next layer and keep doing this until all layers have been processed. Most packets have most likely no bit errors introduced during transmission and therefore all checksums will be correct. This means that all the comparisons can be executed in a speculative manner before the checksum calculations have been completed with a low risk of misprediction.

5.3 Inter-Packet Tasks

Inter-packet tasks are the tasks of protocol processing that are not intra-packet. That means tasks that involve updating connection state variables, setting timers and triggering transmission of packets. This also includes delivering the payload to the application.

Inter-packet tasks have side effects and only some of them are reversible. For example if a local connection state has been updated, it is possible to reverse that change if the execution was speculative and proved to have been miss-predicted. Other tasks are not reversible, such as transmitting an acknowledgment or delivering payload to the application. These irreversible tasks cannot be executed speculatively and specifically important for protocol processing, the checksums must have been proven correct before they are executed.

Inter-packet tasks are irregular. They consist of for example executing state machines, computing congestion windows, setting timers and triggering packets to be transmitted. They are somewhat predictable, which has been proven for TCP where most packets contain data or acknowledgments, but do not change the state of the connection [5.1].

5.4 Common Protocol Tasks

Although each network protocol has its own set of unique features, many mechanisms are common to most protocol stacks. In this section several of these mechanism are described and discussed from the perspective of reception of packets in a terminal. It is necessary to analyze and classify these tasks so that an appropriate assembly instruction set can be developed.

The first three tasks, multiplexing, checksums, and addressing belong to the intra-packet tasks and the last two tasks, flow control and congestion control belong to the inter-packet tasks. Table 5.1 summarizes the kernel operations in every task.

5.4.1 Multiplexing

In the layered protocol architecture we are used to think about, each protocol layer specifies which protocol is used on the layer above. This is done by a certain field in the header, which contains a value specific to the protocol used on the layer above. The processing is of course dependent on this value and we say that the packet stream is demultiplexed to the correct upper-layer protocol processing routine based on this value. The demultiplexing must take place for every packet that is received from the network.

Muliplexing is used for example by IP to decide if the payload is using TCP, UDP or another format. The kernel operations are compare and conditional branches.

5.4.2 Checksums

In computer networks the transmission media is not perfect. Optical fibers have a very small bit error rate, but it can still happen that a bit is flipped during the transmission. Electrical cables have a little bit worse bit error rate and wireless channels suffer from time-varying fading that can cause very high bit error rates. The common way to deal with the problem of erroneous bits in the received packets is to use checksums.

The checksums are computed over the whole packet or parts of the packet at transmission time and put in a certain field in the header or as a tailer to the packet. The receiver has to compute the same checksum and compare to the precomputed value. If a mismatch occurs the whole packet must be discarded and the sender will have to retransmit the packet. For one single packet several checksums on different layers may be present, for example a TCP packet sent by Ethernet will have both the TCP checksum and the Ethernet frame check sequence.

The checksum computations must take place for every packet that is received from the network. Based on the demultiplexing it is known which protocols that are used and thereby which checksums to compute. Various algorithms for checksum computations exist. The most common ones are cyclic redundancy check (CRC) and one's complement addition.

CRC is for example used in Ethernet and normally implemented in an hardware accelerator. In a software implementation shifts and logical operations are the kernel operations. One's complement addition is often implemented in software, for example in IP, TCP, and UDP. The kernel operation is addition.

Task	Kernel operations
Multiplexing	compare, conditional branch
Checksums	shift, logical operations, addition
Addressing	compare, conditional branch
Flow control	compare, conditional branch, assignments
Congestion control	addition, subtraction, shift, compare, conditional branch

Table 5.1: Kernel operations for common protocol processing tasks

5.4.3 Addressing

Addresses are a common property to computer network protocols. They are used for two purposes, the first is to route or switch the packet to the right destination terminal. This part is not discussed further here since I am concentrating at the processing in a terminal. The second purpose of the address is identify the destination. Each protocol layer, at least at layer 2, 3, and 4, has its own addressing scheme. In a terminal addresses on all layers have to be checked. Layer 2 and layer 3 addresses typically specify the terminal (e.g. Ethernet destination address and IP destination address) and layer 4 addresses typically specify the application (e.g. UDP and TCP ports). The addresses have to be compared to certain acceptable values to make sure that the packet is destined for an application running in the terminal that has received the packet. The address matching has to take place for every packet that is received from the network.

The kernel operations are compare and conditional branches.

5.4.4 Flow Control

Protocols can be either conectionless and connection based. The connection based protocols, for example TCP, have to handle flow control. That means that a connection has to be built up by the use of special packets before any actual payload data can be transported. This implies that the terminals that communicate must keep track of the connection state of each connection and act based on that state. The transitions between states occur when packets are received, packets are sent, timers time out or on commands from the application. This means that the state machine handling is done only on some of the received packets and involves a lot of interaction with other parts of the system. Flow control also involves keeping track of sent, acknowledged and received data on an open connection.

The kernel operations are compare, conditional branch, and assignments for finite state machine handling. Timer implementation, interrupt handling and procedure calls are also important to accelerate in a software implementation.

5.4.5 Congestion Control

In connection based protocols, a technique called sliding window is often used for controlling how much data has been sent and how much has been acknowledged. If packets are sent but do not get acknowledged, the congestion control should decrease the size of the sliding window to reduce the congestion on the network. The algorithms used for congestion control involve a variety of

instructions and is triggered by the timers timing out rather than by received packets.

Kernel operations are addition, subtraction, shift, compare, and conditional branches.

5.5 Dual Processor Architecture

The partition into intra-packet and inter-packet tasks gets really beneficial when looking at a dual processor architecture, where one processor is dedicated for each kind of tasks. They are referred to as intra-packet processor (intra-PP) and inter-packet processor (inter-PP).

In the next three chapters the intra-PP is thoroughly described and analyzed. The system architecture is based on the two processors communicating through a shared memory and with some special purpose signals, see figure 5.1.

In embedded systems, the inter-PP and host can be the same processor, then the inter-PP to host interface is a software application programming interface (API).

In general the inter-PP is assumed to be a traditional microcontroller. It has an irregular workload and therefore no specializations are very beneficial. An example of the dual processor architecture is presented in chapter 9.

5.6 Partition of Common Protocols

In this section a closer look is taken on the reception part of common computer network protocols and how they map to the intra-PP and the inter-PP. The protocols that are examined are Ethernet MAC, IP, IPv6, UDP and TCP. Table 5.2 summarizes the partitions.

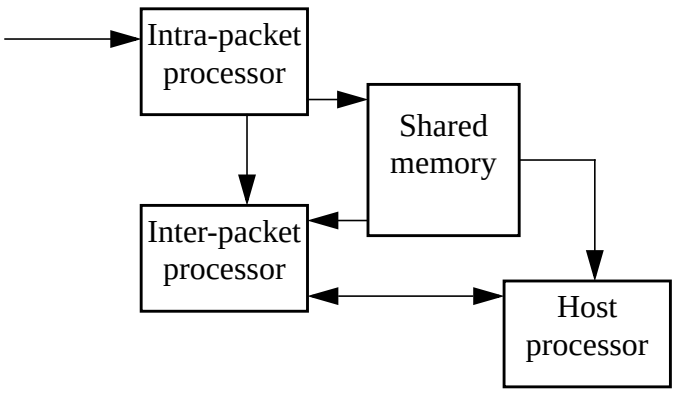


Figure 5.1: The dual processor architecture

5.6.1 Ethernet MAC

The Ethernet MAC protocol is intra-packet task heavy. The CRC checksum calculation, the destination address check and protocol demultiplexing are all intra-packet tasks. New features such as flow control based on pause frames are mapped onto the inter-PP. Figure 5.2 explains the control flow of Ethernet MAC.

5.6.2 IP

The Internet protocol (IP) is also an intra-packet task heavy protocol. Header checksum calculation, destination and source address matching and protocol

Protocol	Intra-packet tasks	Inter-packet tasks
Ethernet MAC	CRC, address check, demultiplexing	Flow control with pause frames
IP	header checksum, address check, demultiplexing	fragmentation and reassembly
IPv6	header checksum, address check, demultiplexing	fragmentation extension header
UDP	checksum, port check, demultiplexing	
TCP	checksum, port check, demultiplexing	flow control, congestion control

Table 5.2: Task partition for common protocols

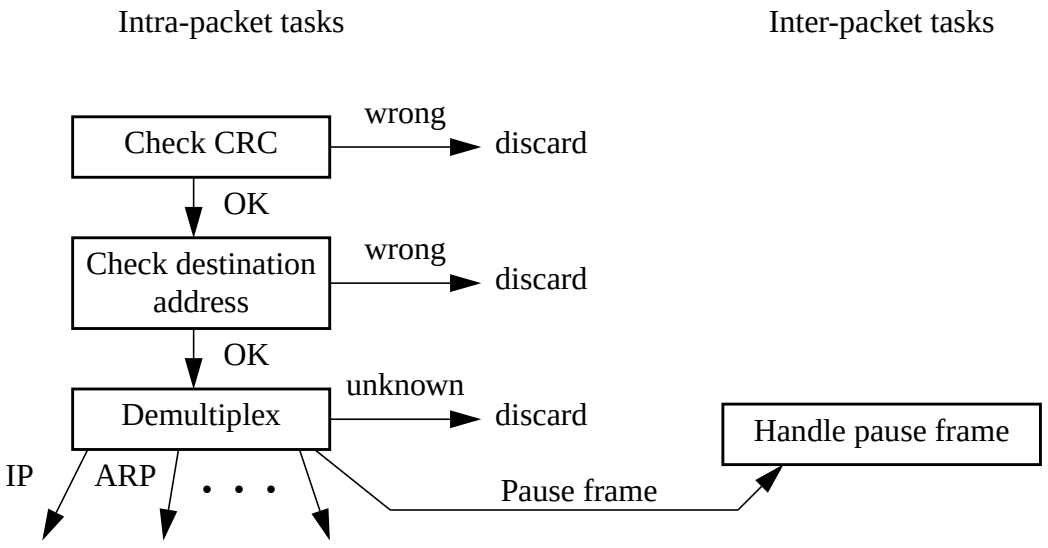


Figure 5.2: Ethernet MAC partition

demultiplexing are mapped onto the intra-PP. The only mechanism in IP that is mapped onto the inter-PP is fragmentation and reassembly, see figure 5.3.

5.6.3 IPv6

The new version of IP, IPv6 works similarly to IP and all tasks except from the fragmentation extension header processing is handled by the intra-PP, see figure 5.4.

5.6.4 UDP

User datagram protocol (UDP) is a conectionless transport layer protocol. All tasks are mapped onto the intra-PP since in UDP there is never any relation

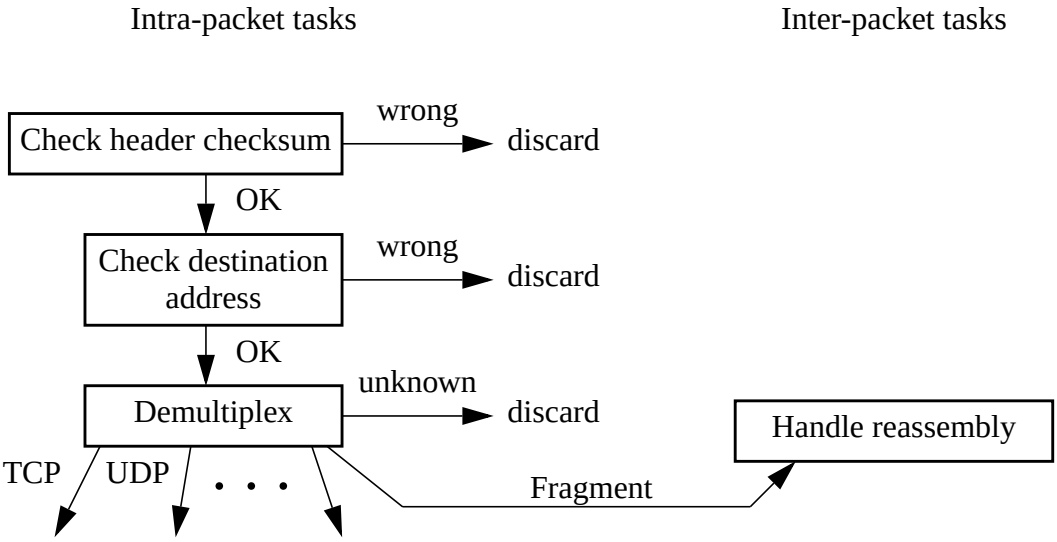


Figure 5.3: IP partition

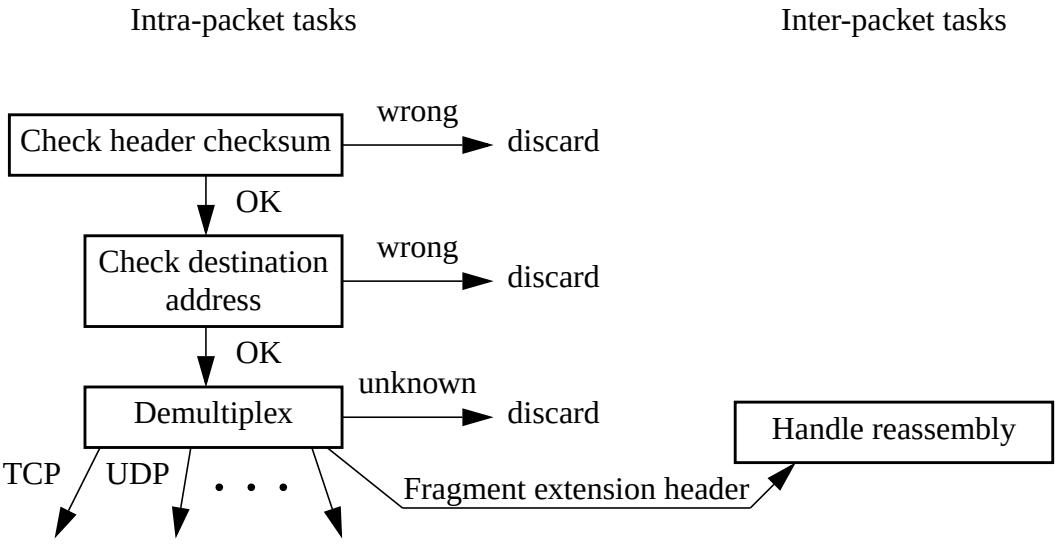


Figure 5.4: IPv6 partition

between two packets. The tasks are checksum calculation, destination and source port matching and payload demultiplexing, see figure 5.5.

5.6.5 TCP

Transport control protocol (TCP) is a connection-based transport layer protocol. For each connection that is built up the terminal keeps information on the state, the window size, the sent, the received and the acknowledged data. The checksum calculation, the destination and source port matching and the payload demultiplexing are handled by the intra-PP. All others tasks are mapped onto the inter-PP, see figure 5.6.

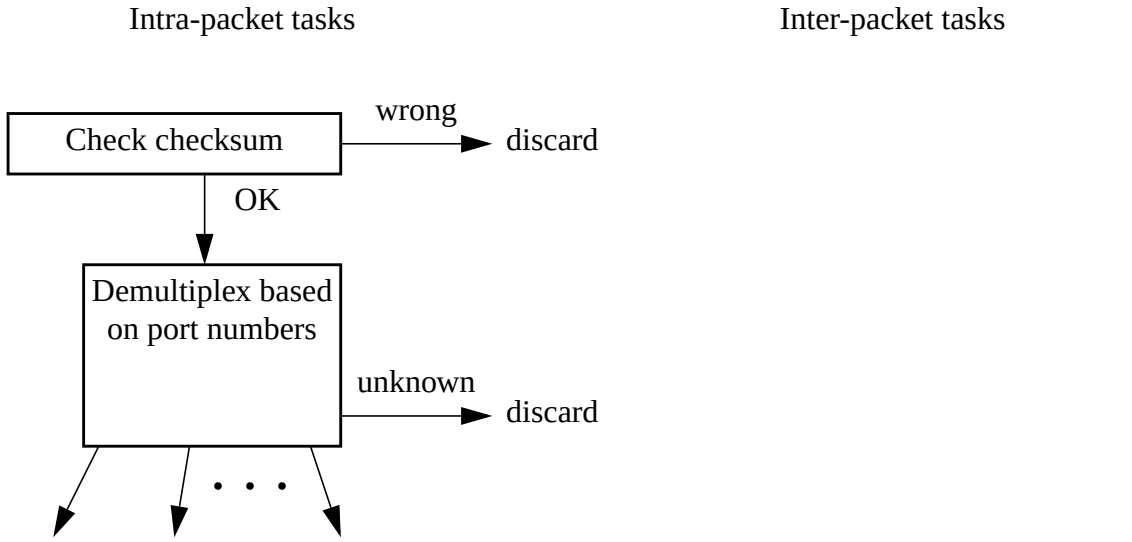


Figure 5.5: UDP partition

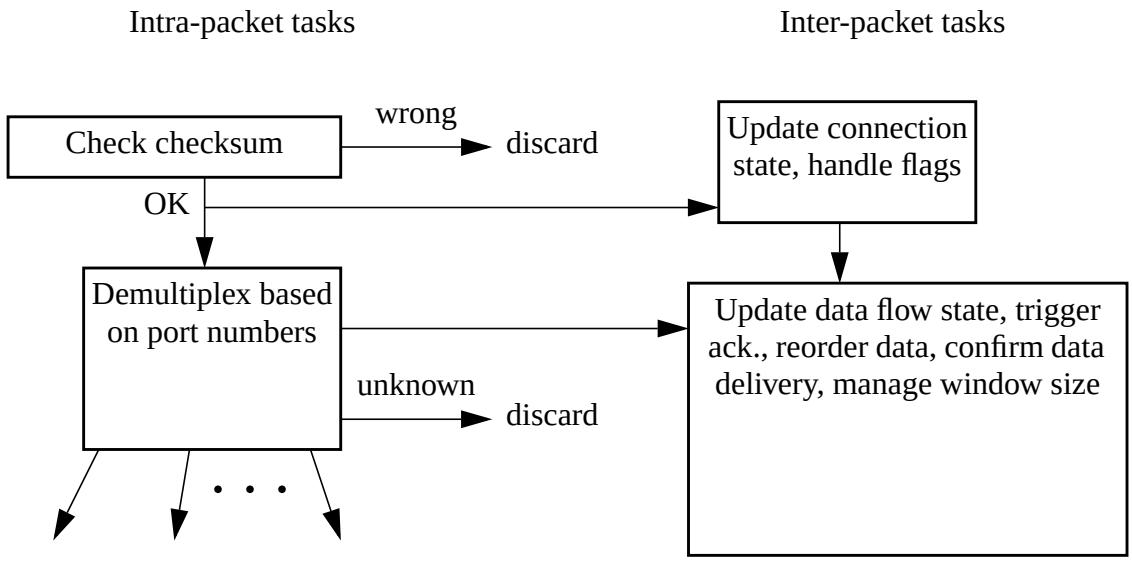


Figure 5.6: TCP partition

References

[5.1] D.D. Clark, V. Jacobson, J. Romkey, H. Salwen, ”An analysis of TCP processing overhead”, IEEE Communications Magazine , vol. 27 No. 6 , pp. 23 -29, June 1989

6

Linkoping Architecture

The intra-PP is based on a novel architecture, called the Linkoping architecture. It has several differences from traditional von Neuman and Harvard architectures. In the Linkoping architecture there does not exist any general purpose register file or any data memory. The architecture is specialized for operating at streaming data. It is a programmable data-flow processor.

6.1 Overview

The Linkoping architecture is aimed at applications operating on a data stream, for example the data from the network. The input data is implicitly always loaded into the dynamic input buffer, so no load operations are necessary. This puts hard real-time requirements on the execution, since a new word of data will be available every clock cycle. This is different from traditional input memory buffers, which stores all the data from the network in a memory and then the processor can access the data randomly and without hard real-time constraints from that memory. However in such a system, the processor needs several instructions per data word, for example for explicit loads and stores, so the processor must run at a frequency much higher than the frequency of the data words.

In the Linkoping architecture, the program is synchronized with the input data clock cycle by clock cycle. Therefore all instructions need to have fully predict-

able execution times and since conditional branches are common in the intended applications, no pipeline is allowed. A pipeline would cause penalty for some conditional branches in some cases and thereby unpredictable execution time for the conditional branch instructions, as discussed in chapter 2.

A processor with no pipeline that still should be able to use a high frequency clock must minimize the time spent for instruction fetch and instruction decode. In the Linkoping architecture this is done by storing the program in three lookup tables inside the processor core. The lookup tables have short access times and the decoding is kept minimal for the parts of the instruction word that is used in the critical path. The program can fit into the three small lookup tables because the instruction set is optimized for a small set of applications and therefore the code size is very small.

The Linkoping architecture does not allow for storing any general intermediate results since no general purpose register file is used. Computations that need intermediate results are off-loaded to accelerators. The accelerators communicate with the core via synchronous control and status signals. An overview of the core and accelerators is shown in figure 6.1. One accelerator is specialized for the task of creating the output data stream. The output data stream can have different format than the input data stream, for example if it should be stored in memory it will contain not only the data but also address information. It can

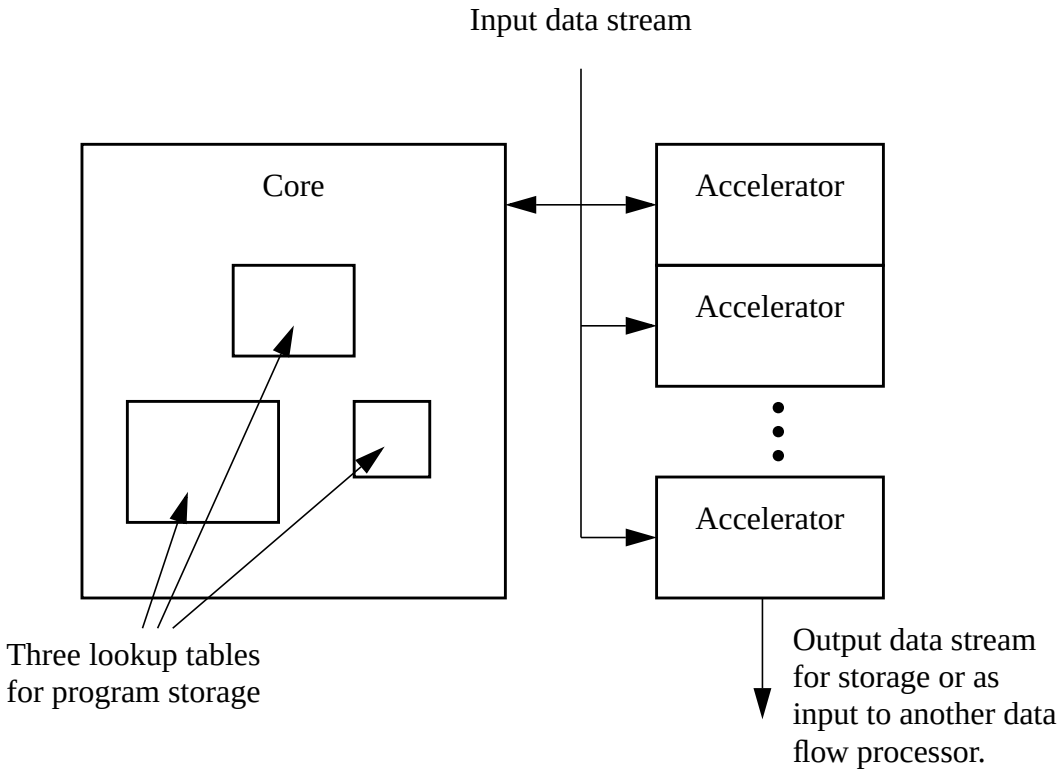


Figure 6.1: Linkoping architecture overview

also have different width of the data and some modifications on the data itself can of course have occurred in the processor. In a network application the most common modification is that some headers are taken away because they have already been processed.

Because the core and the accelerator operate in parallel on the same data word, the Linkoping architecture can be classified as a multiple instruction single data (MISD) architecture. In the core, however, a single instruction is used by several parallel execution paths, so the core itself can be considered to be a single instruction multiple data (SIMD) architecture. Altogether this means that the Linkoping architecture constitutes a multiple instruction multiple data (MIMD) machine. There is one master instruction flow in the core, that can start the execution of the accelerators dynamically dependent on the input data. The accelerators thereafter execute independently of the core instruction flow until a merge occurs when the accelerators have completed execution. This type of machine is sometimes referred to as a loosely coupled co-processor (LCC) machine.

An accelerator has four general communication lines, two inputs start and stop and two outputs, ready and OK, see figure 6.2. Not all accelerators must have all communication lines implemented, for example some accelerators do not need an external signal to control when to stop the execution. The communication lines are connected to the core for most accelerators, but cascading accelerators is also possible, so that the output of one accelerator is the input of another.

6.2 Design Methodology

The Linkoping architecture was designed with protocol processing applications in mind. The main factor for designing a new processor architecture was

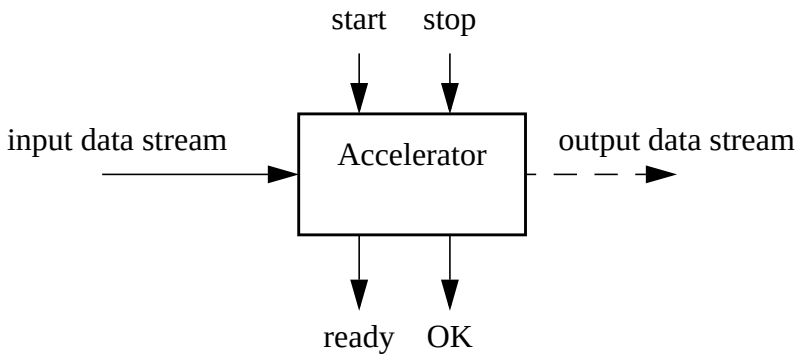


Figure 6.2: Accelerator overview

the desire to process the incoming packets before storing them in a memory. This is beneficial for several reasons. If it is discovered when processing the packet header, that the packet is not interesting for the terminal, then the packet can be discarded before it is fully stored in memory and the further processing can be cancelled. The processor can go into sleep mode and wake up when the next packet arrives on the input port. This saves energy as well as memory bandwidth. Since the processing is already taken care of when the packet is stored in the memory it can be directly delivered to the application just by passing a pointer.

The starting point for the intra-PP architecture was a small set of computer network protocols. An architecture was designed that could execute them and later on minor changes were made to accommodate for other protocols. No complete investigation of all interesting protocols has been performed.

The intra-PP is an instance of the Linkoping architecture, but it was designed before the more general Linkoping architecture specification was developed. The Linkoping architecture was derived by extracting the key properties of the intra-PP. The most important design goal was to be able to execute if-then-else and switch-case-case... statements in one clock cycle, no matter which branch that was taken. These are used for example in protocol demultiplexing and address resolution protocol (ARP) handling, see figure 6.3.

6.3 Intra-PP Architecture

The intra-PP is an instance of the Linkoping architecture, that is used for Ethernet, IP, ARP and UDP processing. A word length of 32 bits is used in the intra-PP.

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

```
if (ARP_targetAddr == MyAddr) {
   if (merge_flag == false) {
      add_to_table();
   }
   if (op_code == request) {
      generate_reply();
   }
}
else {
   discardPacket();
}
```

Figure 6.3: C code examples of typical protocol processing tasks

There are eight important blocks in the intra-PP core, as can be seen in figure 6.4. These are the blocks used for executing the switch-case-case... statements and the if-then-else statements. There are also other blocks in the core, used for communication with the accelerators and external inputs and outputs.

The intra-PP has five accelerators, for CRC calculation, IP header checksum calculation, UDP checksum calculation, packet length counting and memory management.

6.3.1 Dynamic Input Buffer

The dynamic input buffer with field extraction unit captures the new data on the input port every clock cycle. There are two control signals from the instruction decoder (ID). One for controlling how many words to keep in the buffer and one for controlling the extraction. In the intra-PP they are only one bit each, meaning that the buffer can keep 1 or 2 words of data and that the extraction can start from the latest bit or from bit number 16. The structure of the dynamic input buffer can be seen in figure 6.5.

6.3.2 Compare Units

The dynamic buffer and field extraction unit provides input data to the compare units. The reference input comes from the parameter code book (PCB). In

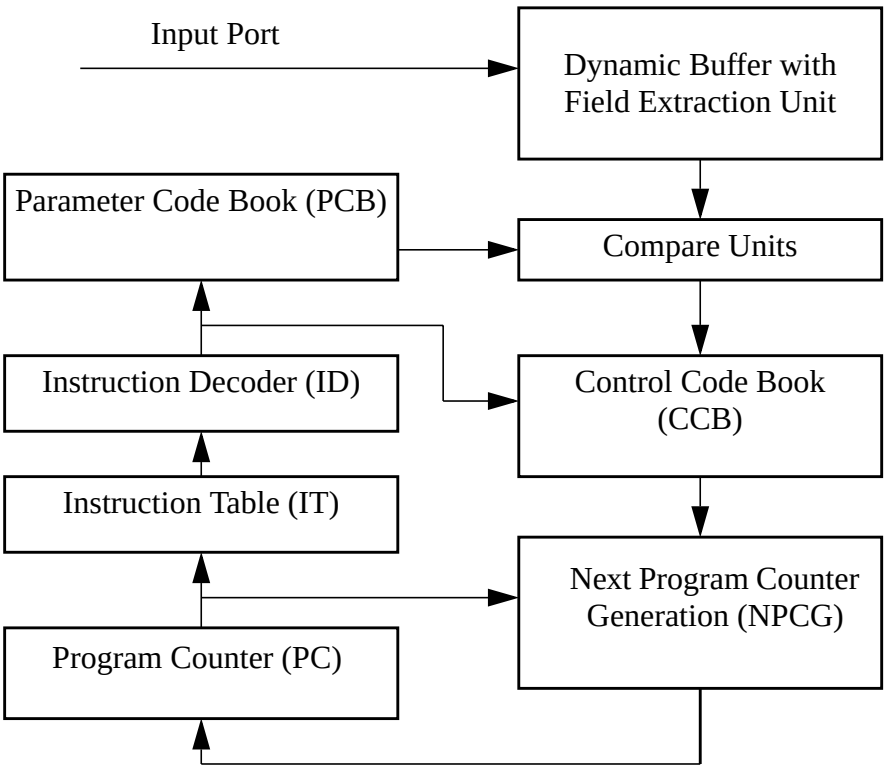


Figure 6.4: Intra-PP architecture

the intra-PP there are 4 parallel compare units. This means that switch-case-case... statements with up to 4 cases and one additional default case are possible. The control signals from the ID are threefold. First there is the mask control signal, which controls the width of the comparisons. There are 4 different widths possible, 32 bit, 16 bit, 8 bit and 4 bit. The second control signal controls whether to use the result of the comparison immediately or only to store it for future use. The third control signal decides if the new result should be combined with the previously stored result or if it should start a new comparison. This feature is important for very wide comparison, for example for Ethernet MAC addresses which are 48 bits wide the intra-PP compares first 32 bits and then 16 additional bit the next clock cycle. Comparisons of any length are possible, so IPv6 addresses of 128 bits can also be handled by the intra-PP. Figure 6.6 shows the compare units.

6.3.3 Parameter Code Book

The parameter code book (PCB) is a lookup table that contains the parameters for the program. These parameters are used for comparisons with data from fields in the packets. The PCB has 16 entries, which contain 4 parameters of 32 bits each. Totally there are 2 kbits of memory required for the PCB. The input is a pointer to one of the lines and the output is the 4 parameters on that line in parallel. By using the PCB the instruction word can be short and the values for the comparisons can be flexible and still carried by the instructions. The PCB is one of the special features of the Linkoping architecture.

6.3.4 Instruction Decoder

The instruction decoder (ID) decodes the instruction word and creates all the control signals for the other blocks in the intra-PP. The pointer to the PCB is in

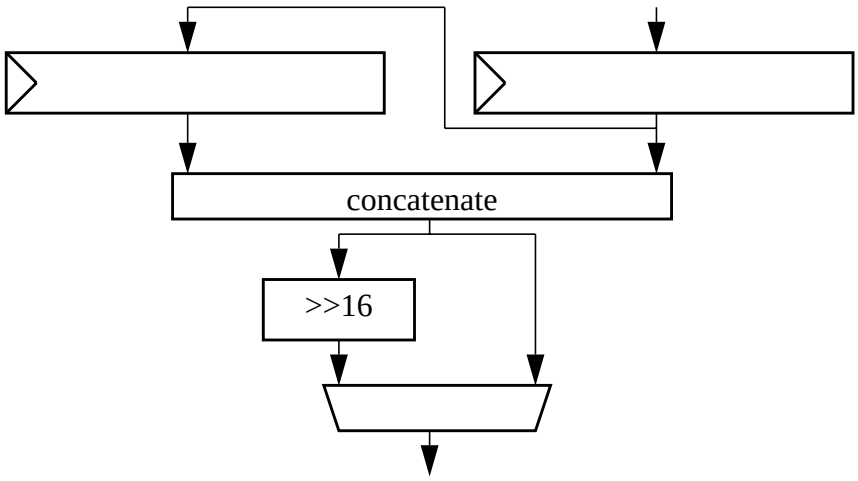


Figure 6.5: The dynamic buffer and field extraction unit

the critical path of the intra-PP and therefore stored directly in the instruction word, so the ID does not influence the maximum clock frequency of the intra-PP.

6.3.5 Instruction Table

The instruction table (IT) stores the main program. Each instruction is 24 bits wide and the IT can hold up to 256 instructions. So the IT requires 6 kbits of memory. The input to the IT is the address, which is the value of the program counter. The output is the instruction stored at that address.

6.3.6 Control Code Book

The control code book (CCB) stores relative jump addresses. The CCB consists of 8 lines, which correspond to the 8 first lines in the PCB. Each line in the CCB contains 4 relative jump addresses of each 8 bits. So the CCB requires 256 bit of memory. The inputs to the CCB are the 3 least significant bits of the PCB pointer and the four outputs from the compare units. If any of the outputs from the compare units is 1, the value stored at the corresponding position of the line to which the pointer points will be the output. If all outputs from the compare units are 0, the output of the CCB will be 0.

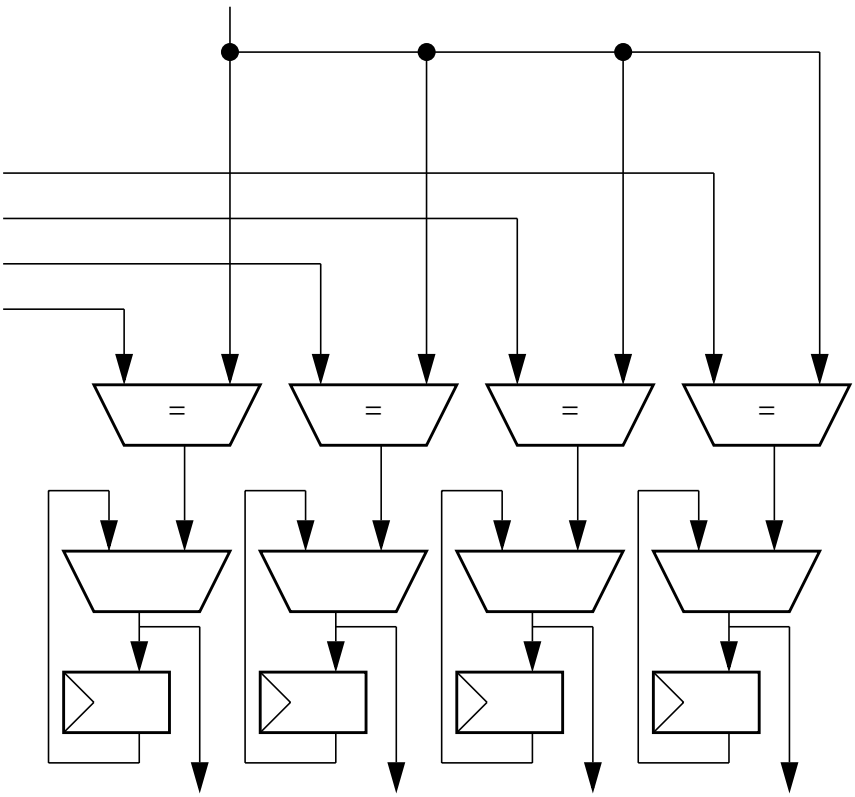


Figure 6.6: The compare units

The CCB allows for the execution of multiple conditional branches in one instruction. This is one of the unique features in the Linkoping architecture. In combination, the PCB and the CCB accommodate the instruction extensions.

6.3.7 Program Counter

The program counter (PC) stores the current address to the IT. The value is updated every clock cycle.

6.3.8 Next Program Counter Generation

The next program counter generation (NPCG) unit calculates the next value for the PC. The inputs are the current PC, the CCB output and control signals from the ID based on which type of instruction is being executed.

6.3.9 Inputs and Outputs

The intra-PP has 19 general-purpose inputs and 10 general-purpose outputs. The outputs are set by special instructions and only valid for one clock cycle, after that they return to “0”. Three of the outputs have special hold circuits, so that they keep their value until it is explicitly reset. The inputs are used for two purposes, jumps and waits. The program execution can be conditionally halted until a certain input pattern occurs. Likewise a jump can be conditionally executed dependent on the inputs. For the conditional jumps only 9 of the 19 inputs can be used.

6.4 Instruction Set

The intra-PP instance of the Linkoping architecture only has 6 instructions. These instructions are optimized for processing of Ethernet, IP, UDP and TCP. For other or more general protocol stacks small modifications in the instruction set are to be expected. The instruction set has not been optimized for minimal code size. We have focused on demonstrating the functionality and architecture of the processor architecture instead of compressed code size. Because of limited design time all optimizations could not be carried out.

6.4.1 Instruction Format

The instructions are 24 bits wide, with possible extensions. The base instructions are stored in the IT and the extensions are stored in the PCB and in the CCB. There are four bits to specify the instruction code, giving a total of 16 possible instruction codes. Only 6 of these are used so the instruction set is easily extendable. The basic format of the instructions is shown in figure 6.7. The bit next to the instruction code is common for all instructions. It is the

Buffer control, that specifies the function of the dynamic input buffer. A “1” in position 19 implies that two words of data will be stored in the dynamic input buffer. A “0” on the other hand implies that only one word of data is stored.

6.4.2 Compare Instruction (CMP)

The compare instruction (CMP) is the instruction that uses the most advanced feature of the intra-PP. The CMP makes it possible to execute a complete switch-case-case... statement in one clock cycle. Since there are 4 comparators in the compare unit, the statement can have up to 4 cases and one additional default case. The CMP instruction makes use of the values in the PCB and in the CCB.

The format of CMP is shown in figure 6.8. Position 18, *New*, specifies if the intermediate results in the compare should be used or not. A “1” in position 18 implies that the intermediate result is discarded. A “0” implies that the intermediate result is used as a precondition in addition to the current comparison.

Position 17, *Jump*, specifies if a jump should be taken. A “1” in position 17 implies that a jump will be taken based on the results of the comparisons and the relative jump addresses in the CCB. A “0” implies that the results of the comparisons will only be stored in the compare unit.

Position 16 through 13, *Pointer*, specifies the pointer to the PCB. The positions 15 through 13 are used as the pointer to the CCB.

Positions 12 and 11, *Width*, specifies the width of the comparisons. The possible values are shown in table 6.1. Position 10, *Offset*, specifies the offset for the extraction from the dynamic input buffer. A “1” in position 10 implies that the extraction starts from the 16th latest arrived bit. A “0” implies that the extraction starts from the latest arrived bit.

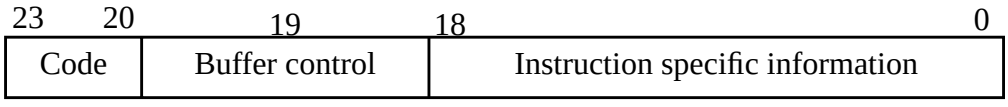


Figure 6.7: Basic instruction format

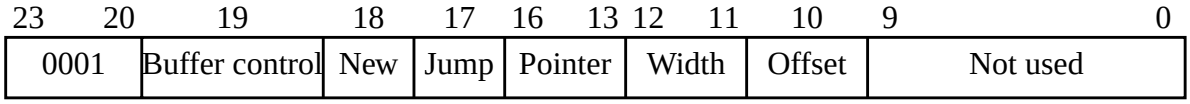


Figure 6.8: Compare instruction format

6.4.3 Jump Instruction (JMP)

There are three types of jumps possible in the intra-PP, all covered by the jump instruction, JMP. The general format of JMP can be seen in figure 6.9. Positions 18 through 17, *Type*, specify the type of the jump. The three available types are shown in table 6.2. The positions 7 through 0, *Jump offset*, specify the relative jump offset from the current instruction. The jump offset is coded in offset code with an offset of 128. So the next instruction after a jump is calculated as *<current instruction> + <jump offset> - 128*. That implies that jumps up to 127 instruction forward and up to 128 instructions backward in the program are possible.

For the unconditional jump positions 16 through 8 are not used. For the jump type *conditional dependent on the inputs*, positions 16 through 8 specify the required input bit pattern. For every “1” in the instruction word a corresponding “1” on the inputs is required for taking the jump. Inputs 8 through 0 are used by the jump instruction.

Width code (Positions 12 through 11)	Width of comparison
00	4
01	8
10	16
11	32

Table 6.1: Possible widths in the comparisons and their corresponding codes

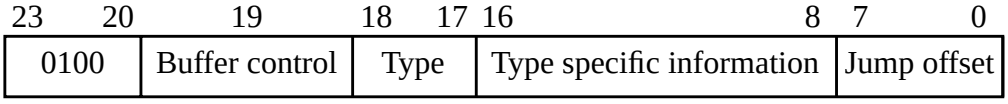


Figure 6.9: Jump instruction format

Type code (positions 18 through 17)	Jump type
00	Unconditional
01	Conditional, dependent on the inputs
10	Conditional, dependent on the comparison results

Table 6.2: Possible types of the JMP

For the last type, *jump conditional dependent on the results of the comparison*, figure 6.10 shows the format of the type specific bit positions. Positions 16 through 13, *Pointer*, positions 12 through 11, *Width*, and position 10, *Offset*, have the same functionality as for the CMP instruction. Although the CCB is not used in case of a JMP instruction. The jump is taken if any of the parameters in the PCB match the extracted field from the dynamic input buffer.

Position 9, *New*, has the same functionality as position 17, *New*, in the CMP instruction. The three major differences between a CMP and a JMP dependent on the comparison results is that the JMP cannot be used only to set the intermediate results in the compare units, that the JMP jumps if any of the parameters match the extracted field and that the relative jump address is contained in the instruction word instead of being taken from the CCB.

6.4.4 Wait Instruction (WAT)

The wait instruction (WAT) is used for synchronizing the program flow with external events, for example the arrival of a packet or the completion of an accelerator task. The format of WAT is shown in figure 6.11.

The wait instruction works in the way that it will defer the updating of the program counter until the inputs match positions 18 through 0, *Input bitmap*, in the instruction word. For every “1” in the *Input bitmap* of the instruction word the corresponding inputs must also be “1” until the program counter is advanced to the next instruction.

6.4.5 Set Instruction (SET)

The set instruction (SET) is used to set the outputs of the intra-PP core. The outputs are used for triggering accelerators or communicating events to other

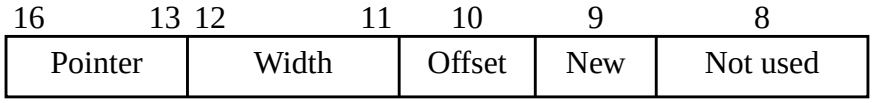


Figure 6.10: Jump conditional dependent on the results of the comparisons

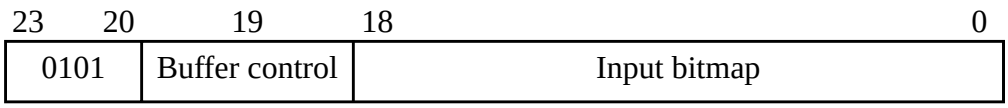


Figure 6.11: Wait instruction format

units in the system. The format of SET is shown in figure 6.12. Positions 9 through 0, *Output bitmap*, specify the outputs that should be set to “1”. As mentioned earlier, the set instruction only holds the outputs for one clock cycle, but on outputs 7, 8, and 9 there are hold circuits which hold their value until they are explicitly reset by external events.

6.4.6 Compare and Set Instruction (CPS)

For efficient program execution, it is necessary to execute compare and set at the same time. This is done by the compare and set instruction (CPS). The format of CPS is shown in figure 6.13.

6.4.7 No Operation Instruction (NOP)

The program flow always has to stay synchronized with the incoming data. If some fields in the packet headers are not included in the processing, the program should ignore that data. This is managed by the no operation instruction (NOP). The format of the NOP is shown in figure 6.14.

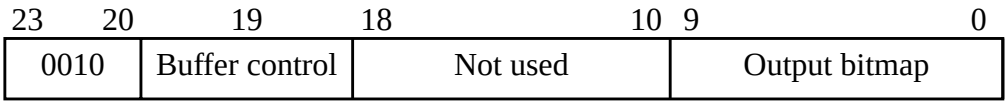


Figure 6.12: Set instruction format

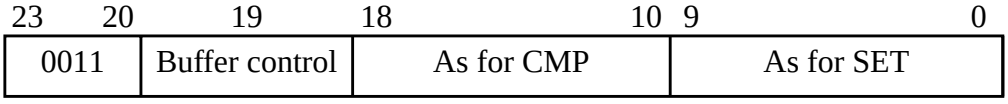


Figure 6.13: Compare and set instruction format

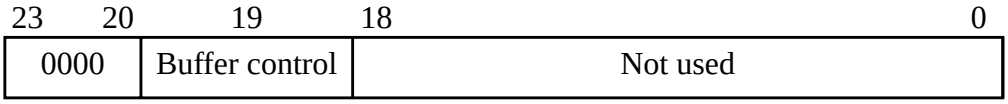


Figure 6.14: No operation instruction format

6.5 Example Program

The best way of describing the intra-PP operation is through an example program. Here I 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 intra-PP is operating in this example, is 130.236.55.5 and the hardware address is 0c:5a:80:ac:4a:b7.

6.5.1 Program Code

The IT contents are shown in figure 6.15, where it is also shown in assembly representation. The PCB contents are shown in figure 6.16. Each line in the PCB consists of 4 parameters of 32 bits each. 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 6.17. 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 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.

6.5.2 Program Execution

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 arriving 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

```
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 6.15: Example program for Ethernet II, ARP and IP/UDP decoding

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 computa-

```
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 6.16: PCB contents for example program

tions. 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.

6.5.3 Comparison to RISC Implementation

The above described execution on the intra-PP required 18 instructions to be executed and decoded. Similar functionality on a traditional RISC machine, where it is assumed that the packet header is stored in memory already, for example by a DMA engine, requires many more instructions. The intra-PP benefits from having the program code split up into three parts. The program for the intra-PP covers more than the program code of a RISC machine. The content of the PCB and partly the content of the CCB are stored in complex data structures in a RISC implementation. Therefore the access to those values is much more efficient in the intra-PP.

For a comparison to the traditional implementation it is assumed that the Ethernet address has already been checked. This is normally performed by a separate ASIC, that takes care of the Ethernet processing. The implementation style

```
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 6.17: CCB contents for example program

in the comparison is taken from [6.1]. The RISC implementation starts with demultiplexing the packet dependent on the type field in the Ethernet header. Then a software interrupt is scheduled, again assuming an IP packet, the interrupt is scheduled for a IP interrupt handler.

The interrupt handler takes care of a lot of things, including gathering statistics. Here only the processing that is also performed by the intra-PP is accounted for. Out of this the first thing that happens is a header checksum computation. Then, the IP destination address is checked. This is done by sequentially checking the acceptable addresses, which are stored in a linked list.

Then the packet is demultiplexed and the appropriate transport layer function is called. This is done by finding a pointer to the function in a lookup table. The lookup is based on a conversion of the payload identifier in the IP header.

Assuming that the packet is of UDP type, the UDP input function is called. It starts with taking away the IP options from the packet and then calculates the UDP checksum. Finally, the UDP port is looked up in a sequential list containing all applications that accept data on any ports and if a matching application is found, the payload is delivered to that application.

The first observation is that the execution time is dependent on how the sequential lists are organized and thereby also unpredictable. Even the minimum execution time is hard to predict, since it is dependent on the length of the IP header as well as the length of the UDP payload, because of the checksum calculations. In the intra-PP, the checksums are calculated in the accelerators, so the instructions for the checksum calculations are not included in the comparison from here on. The interrupt overhead is also not included, because in a separate RISC processor for protocol processing, a jump to the correct function can be used instead of software interrupts. It must, however, be noticed that if the RISC processor is also used for applications, or if accelerators for Ethernet and for the IP and UDP checksums are not available then the instruction count for the RISC processor will increase dramatically.

A detailed look at the remaining tasks yield the instruction counts shown in table 6.3. The instructions that are used are generic RISC instructions and the data path of the processor is assumed to be 32 bits wide. The packet header is assumed to already be stored in memory. The instruction counts are best case, that is, a match is always assumed on the first comparison in a linked list. The average case will be worse and the worst case will be considerably much worse. Totally 54 instructions are used in the best case. This must be compared to 14 instructions in the intra-PP, on the same functionality, the count for the Ethernet destination address match has been deducted from the previously mentioned 18 instructions. So even in the best case, the RISC processor must run more than

Task	Executed instructions (best case)	Instructions (code size)
Ethernet type demultiplexing	10	14
IP address check	18	25
IP demultiplexing	11	11
UDP demultiplexing	15	22
Total	54	72

Table 6.3: Instruction count for reception tasks

three times as fast as the intra-PP to keep up with the line rate. The penalty for one missed iteration in the UDP port search is 17 instructions. The UDP demultiplexing is likely to require several iterations for most of the packets, so in practice the RISC processor has to run a lot more than three times as fast as the intra-PP.

References

[6.1] Gary R. Wright and W. Richard Stevens, “TCP/IP Illustrated, Volume 2: The Implementation”, Addison Wesley Longman, Inc., ISBN 0-201-63354-X, 1995

7

Protocol Processor Implementation

The intra-PP core was synthesized to a 0.18 micron library. The estimated performance after synthesis and placement of the standard cells is 281 MHz. A complete chip layout for the whole intra-PP including all accelerators has also been made. That design is based on a 0.35 micron library.

7.1 Specification

The intra-PP described in the previous chapter has been implemented in a standard cell design flow. The first three sections describe the implementation of the core and then the last section presents the complete chip layout of the core and the accelerators.

7.1.1 Implementation Purpose

The purpose of the implementation was to derive the maximum clock frequency for the intra-PP as well as the silicon area consumption. The maximum clock frequency in combination with the word length gives the absolute performance of the intra-PP. Benchmarking and instruction profiling are not applicable to the intra-PP since the architecture operates in a real-time environ-

ment and the only interesting performance targets are the standard network transmission speeds. The intra-PP is aimed at 10 Gigabit Ethernet, so supporting 10 Gb/s is the design goal. If the performance is lower than that the intra-PP can only support Gigabit Ethernet, which runs 10 times slower. Likewise if the maximum performance is higher than 10 Gb/s there is no benefit from that, except that the processor does not have to run at the maximum of its performance and therefore the supply voltage could be decreased in order to save energy consumption.

7.1.2 Intra-PP Parameters

The intra-PP in the implementation has an IT of 64 entries, a PCB of 16 entries and a CCB of 8 entries. The word length is 32 bits and the parallelism of the processor is 3, meaning that there are 3 comparators in the compare unit. The dynamic input buffer consists of maximally 2 words.

All of these parameters can be modified and optimized for a certain application. For the example program that was described in the previous chapter, a much smaller architecture would have been sufficient.

There are 5 accelerators in the intra-PP. Those are not included in the first standard cell implementation. The accelerators have been implemented individually. Those implementations are described in the next chapter. The complete chip implementation, which is described in section 7.4, includes the accelerators.

7.1.3 Configuration Interface

The intra-PP has a configuration interface, which can be used by the inter-PP or the host processor to configure the IT, the PCB and the CCB. The configuration interface works in the same way as the interface to a synchronous SRAM. That is, each location in the IT, the PCB and the CCB is associated with an address and the content of a location can be changed by writing data to the corresponding address. The configuration interface data bus is 16 bits wide, so the instructions in the IT have two address associated with them, one for the high 8 bits and one for the low 16 bits. Also the parameters in the PCB have two addresses for each parameter in a similar way.

7.2 Design Flow

The design of the intra-PP was based on a C++ high level behavioral description. This description does not contain any information about the intra-PP architecture. Instead it is a generic algorithm for processing packets.

7.2.1 Structural Model

The behavioral model was used to create reference behaviors of packets. Packets were generated with a small piece of software and sent to the behavioral model, which processed the packets and created reference outputs.

A structural model was developed in C++. The structural model is cycle true and bit true. The structural model operates on the packets in the same fashion as the intra-PP itself. It uses the same program code for the IT, the PCB and the CCB and stores the same information. The purpose of making a C++ structural model was that it was easy to make architectural changes and fast to check the functional correctness. The outputs of the structural model were compared to those of the behavioral model.

The next step on the way to implementation was to transfer the structural model from the C++ description to a VHDL description. This was done manually and was a straight forward process, since almost all architectural details were contained in the C++ structural model.

7.2.2 Verification of the VHDL Code

The functional verification of the VHDL code, was divided into three kinds of test cases, single instruction based, formal functions and error injections. In the verification the accelerators were included for full functionality.

Single Instruction Based Verification

The single instruction based verification for the intra-PP differs from that of a traditional processor. Since there are no target registers for the instructions, there is only the program counter (PC) and the outputs, that can prove the correctness of the implementation. The NOP, 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 intra-PP was extended with a non-synthesizable 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.

Formal Functions Verification

For the verification of the formal functions, the example program from chapter 6 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 testbench for the VHDL model use the same stimuli files as the C++ structural model and thereby the VHDL code was verified efficiently.

Error Injection

To make sure that the intra-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 graphical user interface (GUI) of the simulator.

7.2.3 Synthesis and Placement

The VHDL model described in the previous subsection was used for the implementation. The model contains the part of the intra-PP that executes the instructions, the configuration interface and 5 accelerators. 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 accelerators were not included in the synthesis and placement.

The intra-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.

The memory blocks, IT, PCB, and CCB, were implemented with regular flip-flops from the standard cell library. This may not be optimal, but still gives a reasonably good estimation of the performance and the area consumption.

7.3 Implementation Results

All results are estimations after synthesis and placement. The intra-PP uses an area of 0.4 mm^2 without the accelerators. The accelerators have been implemented separately and are described in the next chapter. The ILT and the PCB, that together contain 3072 flip-flops use more than half of the total intra-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 intra-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, there are strong indications that the intra-PP will support a data stream of more than 10 Gb/s.

The critical path is from the PC, through the IT, the PCB, the compare units, the CCB and an adder in the NPCG back to the PC. The delay from the parts can be seen in table 7.1.

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.

7.4 Complete Chip Layout

A complete chip layout of a test chip has also been produced. The chip has not yet been sent for manufacturing, but the layout is ready to send in its current state. All the steps in back-end design have been performed.

7.4.1 Chip Purpose

The chip was designed so that the functionality and performance of the intra-PP can be tested in a real environment. The chip is designed for being tested in a lab environment, but can also be built into a system. There are, however, no concrete plans to do that.

The chip was implemented with a 0.35 micron standard cell library from AMS because that is a proven process and because I was already familiar with the design flow for that process. Since a 0.35 micron process was chosen, 10 Gb/s performance cannot be expected. The chip was optimized for as high performance as possible to be able to extrapolate a predicted performance for the same design in a modern manufacturing process.

7.4.2 Chip Specification

The core in this implementation was incorporating larger lookup tables than the core implementation described in the previous sections. The ILT has 256 entries, so more complex programs can be used. The parallelism is 4, so the PCB and the CCB have four values in each line and there are 4 comparators. The accelerators that were implemented, were CRC accelerator, UDP checksum accelerator, IP header checksum accelerator, packet length counter accelerator, and memory management unit accelerator.

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

Table 7.1: Delays in the critical path

Further more the interfaces have been modified so that the number of pins on the chip could be decreased. The configuration interface is 8 bits wide, so the configuration requires 3 accesses for each instruction in the ILT and 4 accesses for every parameters in the PCB.

There is a special test mode implemented on the chip. By asserting a test input, the two bit test mode input will select what internal state to make observable on the outputs. There are 4 test modes and 48 physical outputs, so totally 192 internal nodes can be observed. The nodes that were chosen to be observable were the core program counter, comparison results in the core, the state machines in the accelerators, the accelerator intermediate results, and the inputs and outputs of the core.

7.4.3 Implementation and Results

The layout of the chip can be seen in figure 7.1. The chip was synthesized and placed by Cadence Envisia PKS, floorplanning, pad placement and power routing were done in Cadence Silicon Ensemble. The final verification was done in Cadence design framework II.

The chip area is pad limited and is 21.9 mm^2. The standard cells use a total cell area of 5.3 mm^2. The performance is estimated to 90 MHz by static timing analysis.

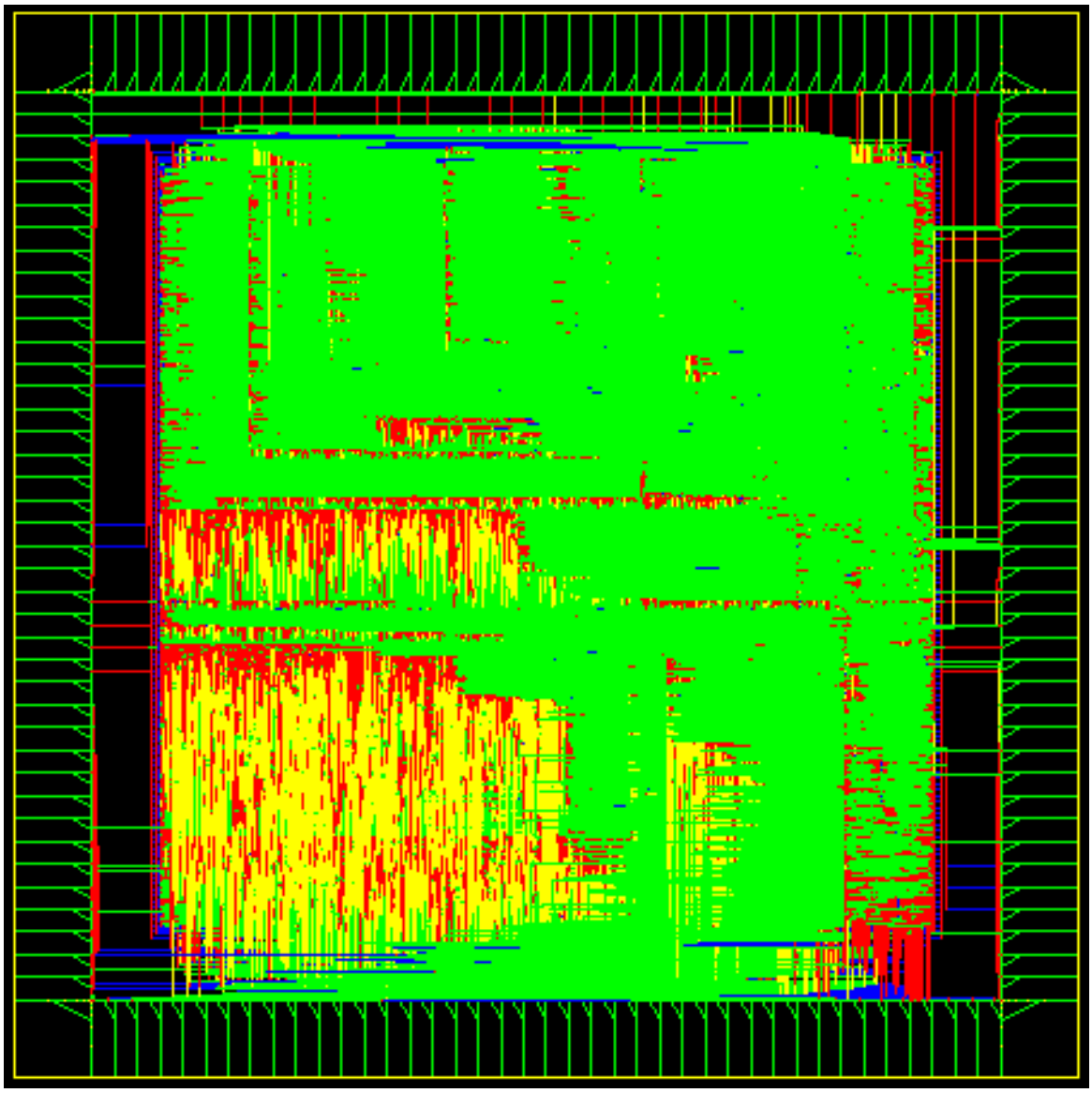


Figure 7.1: Chip layout

8

Checksum Accelerator Implementations

The accelerators for CRC calculation and Internet checksum have been implemented as standalone units for performance estimations. This is important because the checksums are the most computationally intensive parts of the protocol processing. One version of the CRC calculation accelerator has also been manufactured and measured results are available.

8.1 Internet Checksum Accelerator

The Internet checksum is used for error detection in IP headers, in TCP and in UDP. It is therefore a very common checksum that must be calculated for almost all packets in modern computer networks.

8.1.1 Internet Checksum Specification

In the intra-PP two separate accelerators for Internet checksums are used. One is used for UDP and TCP checksums and the other for IP header checksums. The reason for having two accelerators is that they have to operate simultaneously and therefore a single accelerator would not have been sufficient. In this section only the accelerator for UDP a TCP is discussed.

The Internet checksum is calculated as the one's complement sum [8.1]. For the IP header checksum, all the fields in the header are included in the computations. For UDP and TCP some fields from the IP header are included in the checksums, together with the whole UDP or TCP packet. This is a violation against the layered protocol stack definition and complicates a truly layered implementation.

The collection of fields from the IP header that are included in the UDP and TCP checksums is called the pseudoheader. The pseudoheaders for IP and IPv6 can be seen in figures 8.1 and 8.2 respectively.

The checksums are 16 bits wide and are calculated by splitting up the data into 16 bit words. If the packet size is not a multiple of 16 bits, it is padded with zeros for the checksum calculation. The 16 bit words are added by one's complement addition. The one's complement addition is associative, so it does not matter in which order the words are added. More important, the words can be

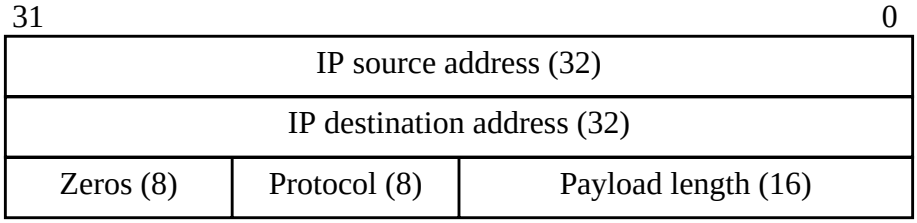


Figure 8.1: IP pseudoheader

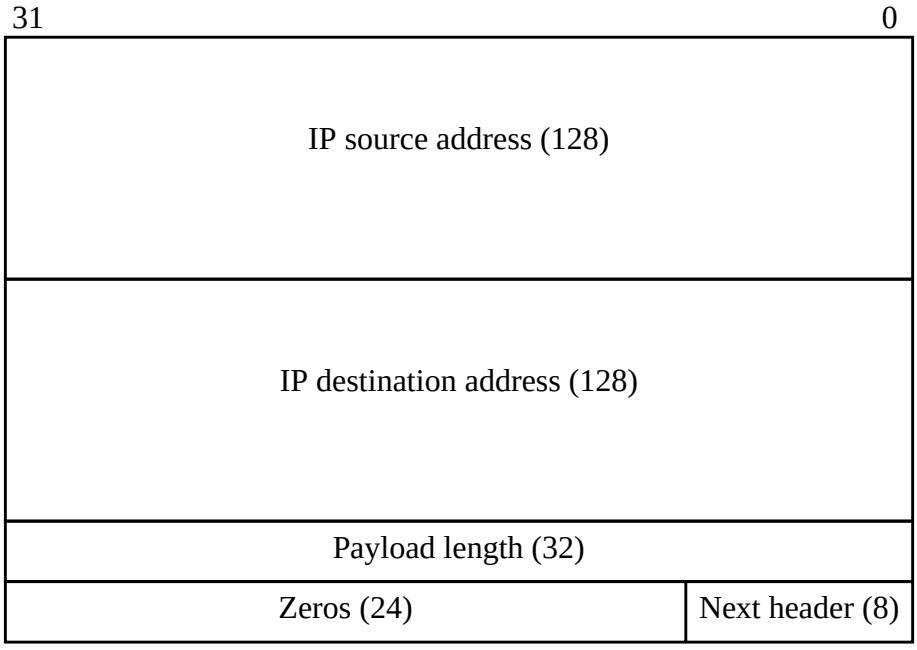


Figure 8.2: IPv6 pseudoheader

added in parallel for efficient implementation. This technique is commonly used in software implementations.

The sender initially sets the checksum value in the header to zero and then pads the packet with zeros to a length which is a multiple of 16 bits. Thereafter the packet is split into 16 bit words, which are added by one's complement addition. The result is inverted and placed in the header field for the checksum value.

The receiver also pads the packet with zeros to a length which is a multiple of 16 bits. Then it splits the packet into 16 bit words and adds all the words with one's complement addition. The resulting checksum value must be zero if no transmission errors have occurred.

The accelerator described here handles UDP and TCP checksums. The accelerator for IP header checksums is simpler and therefore not discussed in detail. For the UDP and TCP checksum calculation a problem arises if the IP packet has been fragmented. The problem is due to the processing paradigm, where the checksums are processed in the intra-PP and the reassembly takes place later on in the inter-PP. The way to solve that problem is to let the accelerator communicate with the inter-PP. The accelerator stores the intermediate checksum calculation results and recognizes when another fragment of the same packet arrives. The inter-PP can order the accelerator to discard intermediate results if the reassembly timer expires.

The Internet checksum accelerator is general and can handle both UDP and TCP and both IP and IPv6. It includes a Length calculation unit because the TCP header does not specify the packet length and it must be calculated from the IP packet total length and the IP packet header length. The implementation that is described here is based on the results from [8.2] and has some differences from the ones that are used in the intra-PP and the demonstrator.

8.1.2 Hardware Implementation

The accelerator consists of four units. A calculation unit, a memory unit, the length calculation unit and a control unit, see figure 8.3. For the calculation unit, three different one's complement adder structures were evaluated. Two of these are based on combinations of two's complement adders. Figure 8.4 shows adder structure *A* and figure 8.5 shows adder structure *B*, which is a pipelined version of structure *A*. The last structure, *C*, is based on the logic equations for directly describing one's complement addition. In [8.3] a 4-bit version is presented, that adder was converted into a 16-bit adder, which is adder structure *C*.

The calculation unit contains two one's complement adders because three 16-bit numbers must be added every clock cycle.

The length calculation unit handles the calculation of the packet length. It is also used for calculating the total length of fragmented packets and calculating

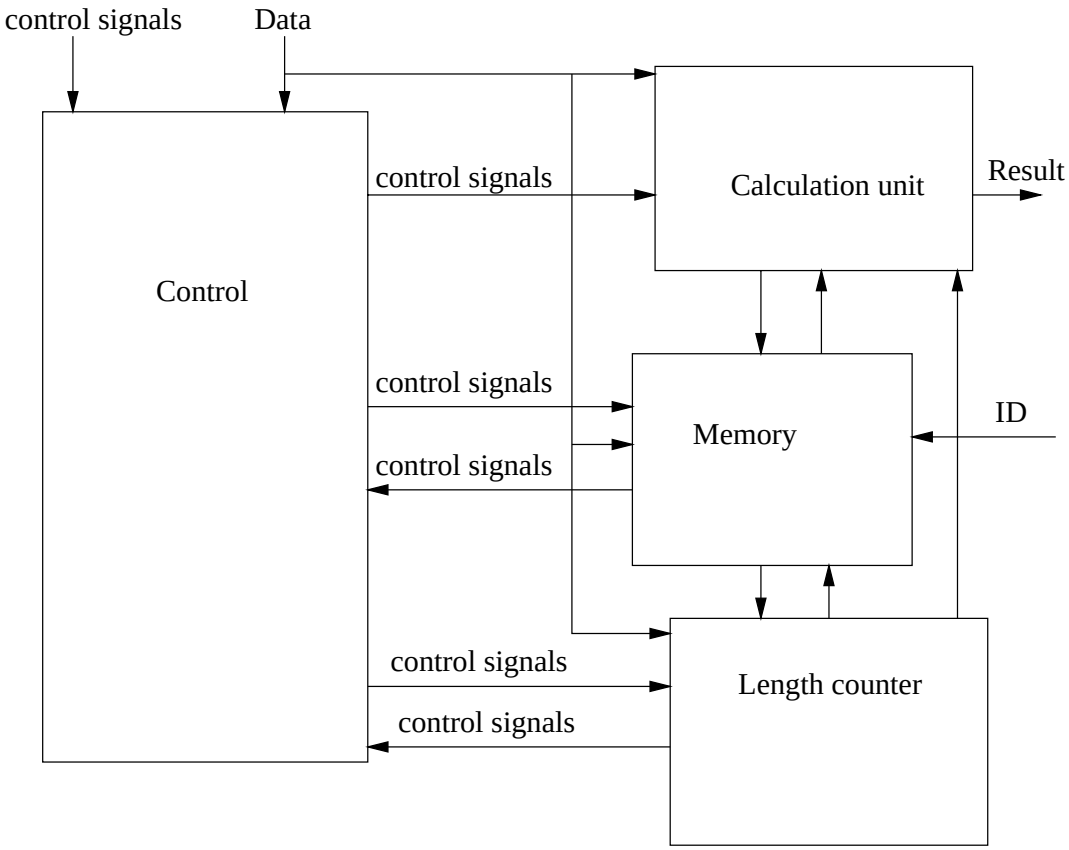


Figure 8.3: Overview of the Internet checksum accelerator

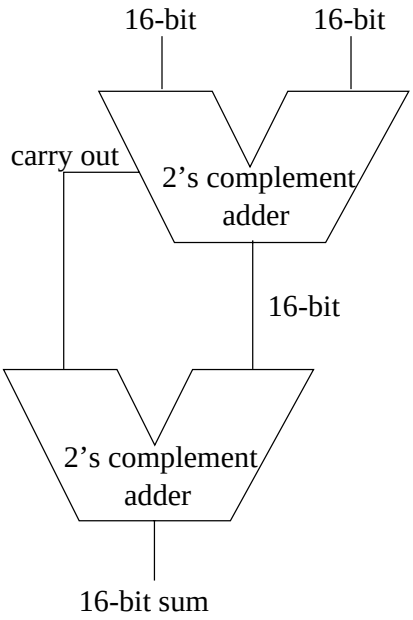


Figure 8.4: Adder structure A

the accumulated number of bytes that have arrived from one packet that has been fragmented.

The memory unit stores the intermediate results for fragmented packets along with the identification data that is needed to recognize to which packet a fragment belongs. The memory unit also implements the communication interface with the inter-PP for discarding intermediate results to packets which have timed out.

Finally the control unit generates all control signals that are needed for selecting the correct input to the calculation unit and controlling all the other activity in the accelerator. The control unit also handles the communication with the intra-PP core inputs and outputs.

8.1.3 Implementation Results

The Internet checksum accelerator has been synthesized to a 0.18 micron 6 metal layer standard cell library from UMC. The synthesis was done by cadence Ambit Buildgates and placement and routing was done by Cadence Silicon Ensemble. Static timing analysis has been used to derive the performance estimations.

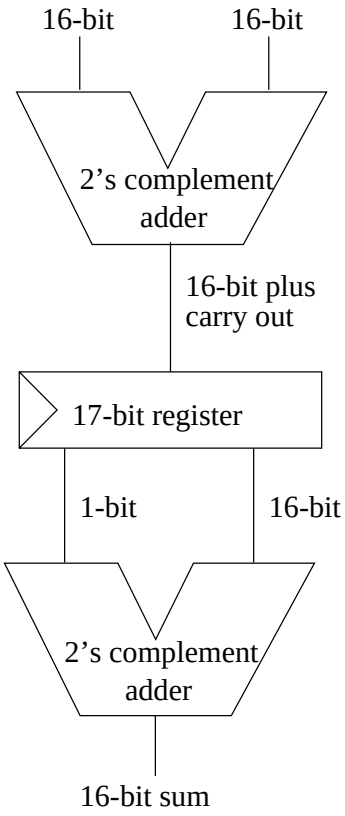


Figure 8.5: Adder structure B

The results of the three adder structure implementations are shown in table 8.1. Structure C was chosen for the final implementation, because all three structures are sufficiently fast and structure C occupies the smallest silicon area.

The performance of the whole Internet checksum accelerator was not satisfactory as can be seen in table 8.2. However the critical path was found to be in the control unit which had not been optimized for high speed. So the result for only the data path of the accelerator was extracted and that is sufficiently fast, as also is shown in table 8.2.

An interesting observation is that the memory unit occupies 61% of the total silicon area. All results are obtained from typical process and typical operating conditions.

8.2 CRC Accelerator Chip

Cyclic redundancy check (CRC) codes are used in many communication protocols. The CRC codes are used for error detection. The CRC codes come in many fashions, which are described by the length and a generator polynomial. The most common is CRC-32 which is a 32 bit CRC described by a specific polynomial. CRC-32 is used in for example Ethernet and asynchronous transfer mode (ATM) adaption layer 5 (AAL5).

Structure	Propagation delay	Silicon area
A	1.34 ns	0.0041 mm^2
B	0.93 ns	0.0105 mm^2
C	1.19 ns	0.0025 mm^2

Table 8.1: Adder implementation results

Unit	Delay	Max frequency	Throughput	Silicon area
Data path	2.77 ns	361 MHz	11.55 Gb/s	0.199 mm^2
Accelerator	3.59 ns	278 MHz	8.91 Gb/s	0.232 mm^2

Table 8.2: Implementation results

8.2.1 CRC Specification

The CRC calculation is specified as the calculation of the remainder of a polynomial division. The dividend is built up from the data packet and the divisor is the generator polynomial. The generator polynomial for CRC-32 is specified as:

$$g(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$$

The data packets can be up to 1500 bytes in Ethernet (or 9000 bytes if jumbograms are allowed) so the dividend is a polynomial of order up to 1500 * 8 = 12000. The division of such a polynomial with the generator polynomial certainly does not look simple.

The dividend is not only the data packet, to the data packet 32 zeros must be added and then the polynomial division can be performed. These 32 zeros correspond to a frame check sequence (FCS) that is zero.

In Ethernet, the sender inverts the 32 first bits of the packet and appends 32 zeros to the end of the data packet and performs the polynomial division. Then the remainder of the division is inverted and appended to the packet as the frame check sequence (FCS).

The receiver performs the same polynomial division, including the received FCS instead of 32 zeros at the end of the packet. After the division, the remainder must be zero if the packet is correctly transferred.

The computation of the remainder of the polynomial division can be performed by a linear feedback shift register (LFSR). The LFSR processes one bit of data every clock cycle.

8.2.2 CRC Parallelization

The performance of a CRC implementation can be improved by processing several bits every clock cycle. This requires the logic to change from an LFSR to more complex combinational logic. The combinational logic is based on XOR-gates, just as the LFSR, but contains several XOR-gates in series in the critical path. The total speedup is therefore limited by the complexity in the logic and processing *n* bits every clock cycle gives a speedup of roughly *n*/2 over the LFSR implementation [8.4]. The principle for parallelization is shown in figure 8.6.

The packet data is U[n], where U[0] is the bit which is transmitted last and also least significant in the dividend. The generator polynomial is described in a vector $[g_k\ g_{k-1}\ g_{k-2}\ ...\ g_0]$, where g_k and g_0 are always 1. The index *k* is the order of the polynomial, so for the CRC-32, *k*=32. The LFSR (a) can be split up into a combinational part (CL1) and a state register (b). Furthermore, two of these combinational blocks, CL1, can be connected in series (c) and thereby two

input bits can be processed every clock cycle. Finally the two CL1 blocks can be merged into one CL2 block. By using this technique repeatedly parallelizations of arbitrary width can be constructed.

One problem with implementing the CRC calculation in parallel is that the packet size is not necessarily a multiple of *n* bits. For Ethernet the packet size is a multiple of 8 bits, because the size is specified in number of bytes, but for *n* greater than 8 special considerations must be made. Parallelization with *n*=32 or *n*=64 are desirable to reduce the clock frequency and achieve high performance.

There are several ways to cope with the last part of the packet, which is not a multiple of *n* bits. One way is to have a special unit that takes care of the last bytes one by one. It starts after the main unit has completed the calculation on the greater part of the packet and uses a calculation unit with an 8 bit wide input. Another approach was chosen in a chip implementation, described in the next sub-section.

The standards specify that the implementation must start by resetting the state register and then perform the polynomial division. That requires complex combinational logic in the critical loop and in [8.5] it was shown that the implementation can be made simpler and thus faster by setting the state register to a modified start value and then use simpler combinational logic in the critical loop. This way also requires that 32 zeros are fed to the calculator after the

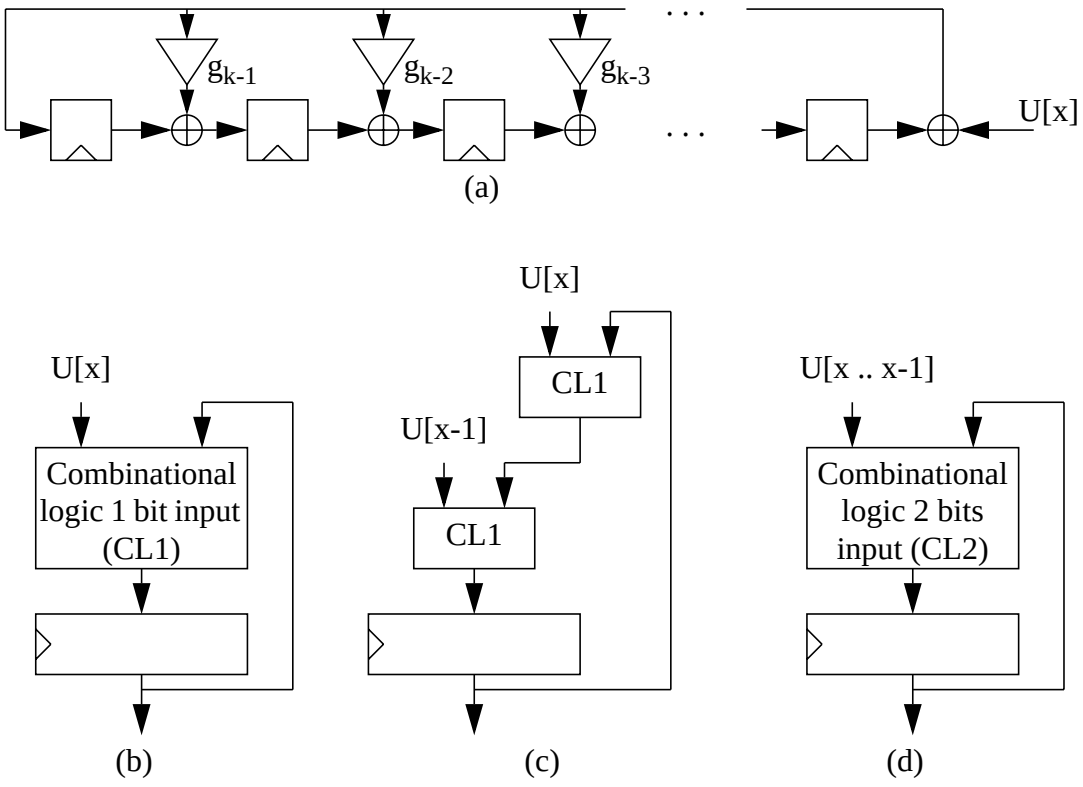


Figure 8.6: Principle for CRC parallelization

actual data in order to get the correct value. The major change that is achieved compared to the parallelizations described in figure 8.6 is that the inputs are only used once in the equations.

8.2.3 Chip Implementation

In this sub-section an implementation of the algorithm in [8.5] using 32 bit and 64 bit parallel input is described. Chips have been manufactured and measured results are presented in the next sub-section.

The circuit was implemented by using RTL descriptions in VHDL and synthesis by Cadence Ambit Buildgates. Place and Route was done in Cadence Silicon Ensemble and physical verification was performed by the Cadence DIVA tool. The manufacturing process was AMS 0.35 micron process with 3 metal layers. The standard cell library was supplied by Europractice.

The VHDL code consists of 5 parts, as shown in figure 8.7. The input registers capture the 36 input signals every positive clock edge. There are 4 lanes each consisting of one control bit and 8 data bits, so the interface is similar to the XGMII (extended gigabit media independent interface) of the 10 Gigabit Ethernet. The implementation can handle any number of bytes in the packet.

The input8_2 unit splits the data to 64 data bits in parallel and generates an enable signal for the crc8 unit. The crc4 unit calculates the CRC by adding 4 new bytes every clock cycle and the crc8 unit calculates the same CRC by adding 8 new bytes every second clock cycle. The purpose of making two implementations on the same chip is just to be able to compare them.

The output multiplexer selects the output from crc4 or crc8, so only one of them can be observed at a time.

Both calculation units can handle a last word of data that contains any number of bytes. This is managed by actually having several computation units in paral-

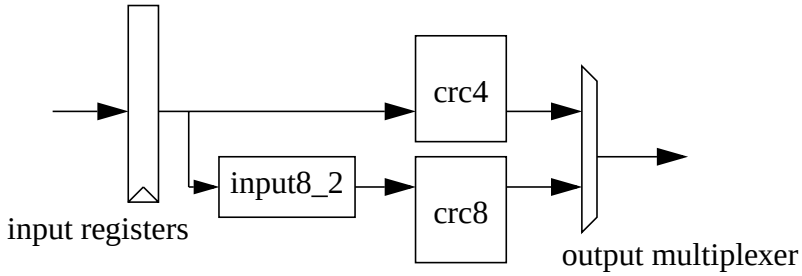


Figure 8.7: Block diagram that shows the structure of the implementation.

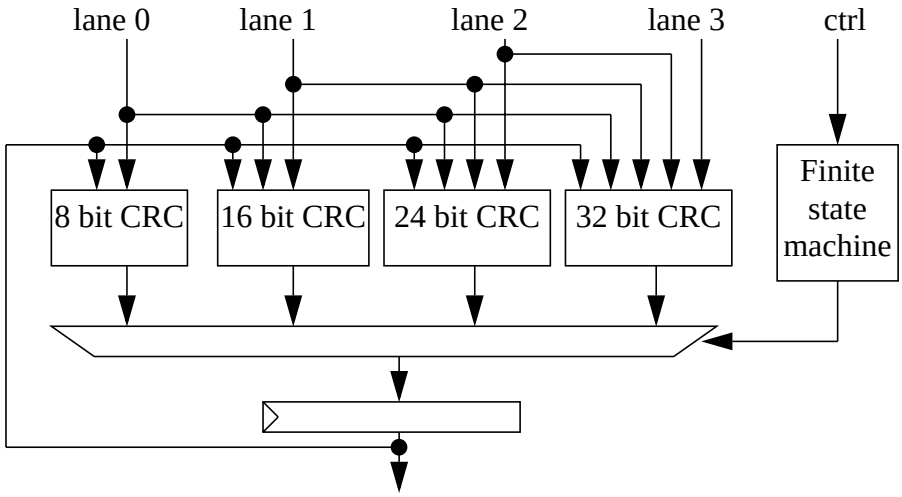


Figure 8.8: The architecture selected for implementation

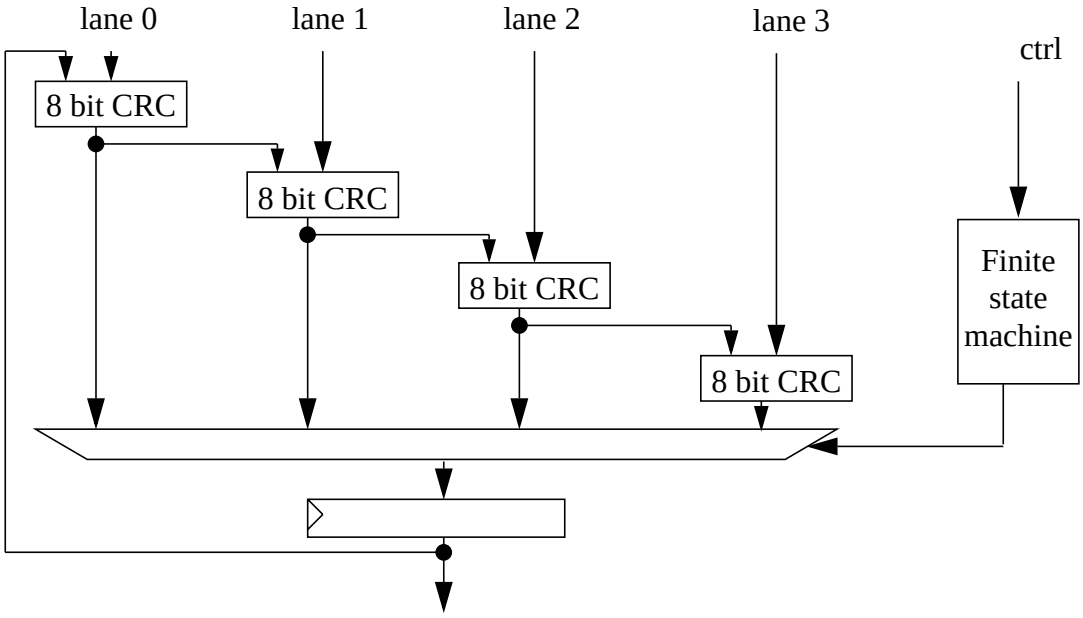


Figure 8.9: The alternative architecture, not selected for implementation

lel and selecting the correct value based on the input control signals and by having a finite state machine, that controls the computations.

There is another possible solution to this problem, which is to use several 8 bit input computation units that are connected in series. Both alternatives were considered and synthesis has been made for both of them, but only the prior one was selected for chip fabrication. The alternatives for crc4 are explained in figure 8.8 and figure 8.9. For crc8 the same principle is used. The second alternative (figure 8.9) has longer critical path than the prior one and therefore cannot handle as high throughput, on the other hand it consumes smaller silicon area. For this chip throughput was more important than silicon area and therefore the architecture in figure 8.8 was selected.

The equations for the CRC calculation with different input widths were generated by a C-program, which takes only a polynomial as the input. This was necessary since there are 128 equations required for the crc8 unit. Actually units with up to 20 bytes input in parallel were also generated, but these were too complex to be implemented efficiently.

In order to improve the throughput of the circuit, some techniques were used. First, flip-flops in the state register that have high fanout were cloned. Second, the most critical paths were identified and logic was pushed over the register boundary to relax the critical paths. This, of course, had to be compensated for at the output.

The observability of the circuit is further improved by having the possibility to select internal states with the output multiplexer. This is not shown in figure 8.7, but the output multiplexer has actually 4 inputs of 32 bits each, 2 of those are the CRC values from crc4 and crc8 respectively. The other 2 are built up by internal states which can be used for testing purposes.

8.2.4 Implementation Results

The total chip implemented in 0.35 micron technology uses 7.73 mm^2 silicon area, see figure 8.10. It has 84 pads and is pad limited. The area of the core is 2.87 mm^2.

3 chips have been mounted on specially designed test boards, see figure 8.11. Measurements were conducted with Agilent 16700 logic analysis system. The limitation in the testing was the testing equipment, which can only supply 180 MVectors/s. The results at supply voltage of 2.5 V are shown in table 8.3. Because 32 bits are processed every clock cycle a throughput of 5.76 Gb/s is managed. All three chips operated at 180 MHz for supply voltage from 2.2 V to 2.7 V. The crc8 circuit on chip #1 did, however, not work correctly.

Why the chips did not work as fast at supply voltages above 2.7 V is hard to explain. The crc4 on chip #1 operated correctly at 130 MVectors/s at 3.3 V supply voltage, which is the nominal supply voltage for the process.

Chip #	crc4 throughput	crc8 throughput
1	>5.76 Gb/s	Failure
2	>5.76 Gb/s	>5.76 Gb/s
3	>5.76 Gb/s	>5.76 Gb/s

Table 8.3: Measurement results

Overall the manufactured chips worked better than expected. Static timing analysis on post layout data estimated a maximum throughput of 5.5 Gb/s for crc4 and 5.2 Gb/s for crc8, using typical process parameters in typical operating conditions.

The reason why crc8 is estimated to support less throughput than crc4 is that it uses an enable signal to control the operation and therefore has to use more complex flip-flops.

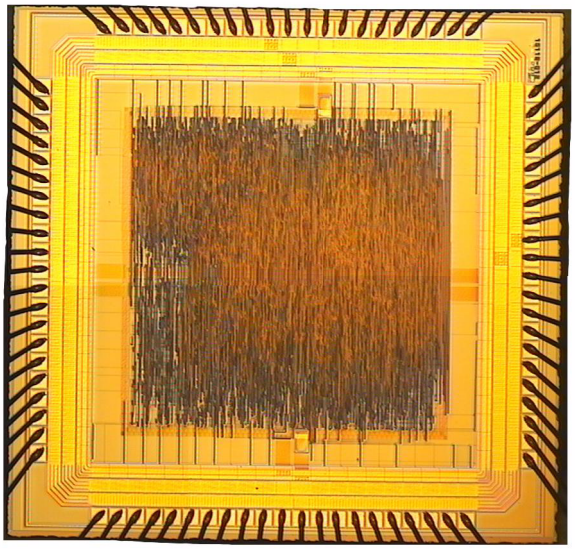


Figure 8.10: Chip photo. The chip area is 7.73 mm^2.

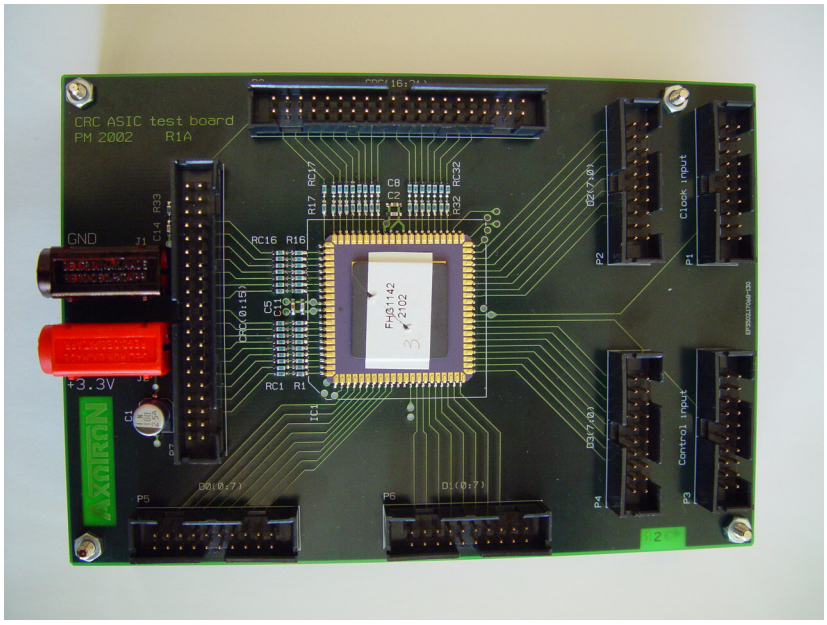


Figure 8.11: Photo of test board for chip measurements.

8.3 Modified CRC Accelerator

By the use of state-space transformations, the logic in the critical loop of the CRC accelerator implementation can be significantly reduced [8.6]. The trade-off is that pre-processing and post-processing logic of considerable complexity is needed. These blocks can, however, be pipelined unlike the critical loop and very high performance is possible.

8.3.1 State-Space Transformations

By using state-space transformations it is possible to reduce the logic depth in the critical loop of the CRC computation to just 1 level of 3-input EXOR gates. That is the same complexity as an LFSR has. This comes to the cost of required pre-processing of the input and post-processing of the state to calculate the CRC value. The complexity of the logic in the critical loop is not dependent of how many bits that are processed in parallel.

The pre- and post-processing can be pipelined since it is not part of the loop. In the loop, pipelining cannot be used because of data dependencies. For each CRC length and polynomial, there exist several different state-space transformations, that result in these nice properties. The complexity of the pre- and post-processing logic is dependent of the transformation that is used. The transformation is described by a matrix **T** in Galois Field 2 (GF2).

The CRC calculation before any state-space transformation has taken place, is described as:

$\mathbf{x}(m+1)=\mathbf{A}^M\mathbf{x}(m)+\mathbf{B}_M\mathbf{u}_M(m);$

$\mathbf{y}(m)=\mathbf{x}(m)$

Here, $\mathbf{x}(m)$ is the current state, $\mathbf{x}(m+1)$ is the next state and M is the number of bits processed per clock cycle. Here the CRC value $\mathbf{y}(m)$ is used as the internal state in the computations. This is what introduces the complexity in the critical loop. The matrix **A** is derived from the CRC polynomial and so is the vector **b**. $\mathbf{B}_M$ is derived from **b** and **A** as $\mathbf{B}_M = [\mathbf{b}\ \mathbf{Ab}\ \mathbf{A}^2\mathbf{b}\ ...\ \mathbf{A}^M\mathbf{b}]$. The vector $\mathbf{u}_M$ contains the M input bits to be processed, and $\mathbf{y}(m)$ is the output.

After the transformation the calculation is described as:

$\mathbf{x}_t(m+1)=\mathbf{A}_{Mt}\mathbf{x}_t(m)+\mathbf{B}_{Mt}\mathbf{u}_M(m);$

$\mathbf{y}(m)=\mathbf{C}_{Mt}\mathbf{x}_t(m)$

Here

$\mathbf{A}_{Mt}=\mathbf{T}^{-1}\mathbf{A}^M\mathbf{T};$

$\mathbf{B}_{Mt}=\mathbf{T}^{-1}\mathbf{B}^M;$

$\mathbf{C}_{Mt}=\mathbf{T}$

The transformation matrix **T** can be derived from a vector $\mathbf{b}_1$. The matrix **T** has M number of rows and as many columns as the degree of the generator polynomial. In this study the CRC-32 polynomial is used, thus **T** has 32 columns and is calculated as:

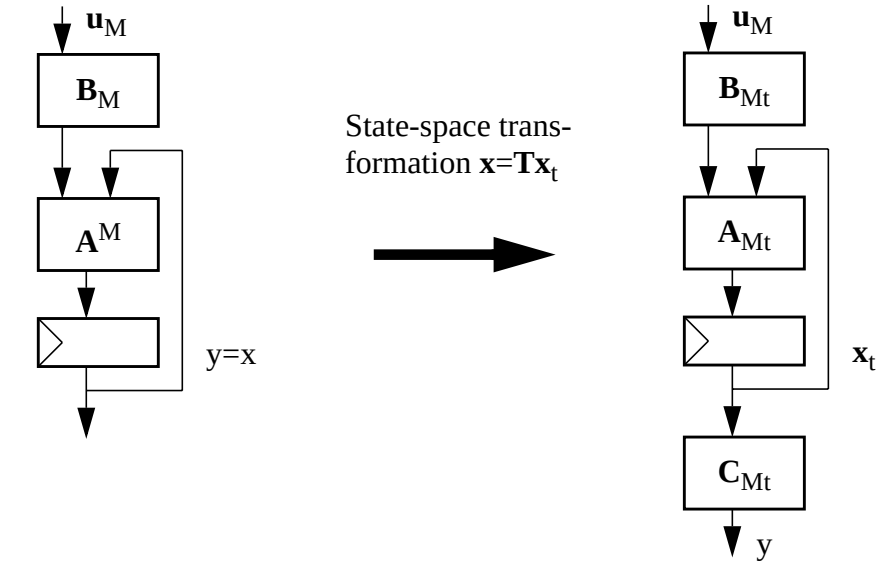
$\mathbf{T} = [\mathbf{b}_1\ \mathbf{A}^M\mathbf{b}_1\ \mathbf{A}^{2M}\mathbf{b}_1\ ...\ \mathbf{A}^{31M}\mathbf{b}_1]$

The $\mathbf{b}_1$ has to chosen with the criteria that all the columns in **T** are linearly independent.

8.3.2 Automatic VHDL Code Generation

A C-program was constructed in order to rapidly be able to create many different implementations of the CRC circuit. The input to the C-program is the matrix, **A**, and the vector, **b**, describing the M bit parallel CRC calculation before any state-space transformation has taken place and the vector, $\mathbf{b}_1$, describing the state-space transformation, see figure 8.12.

The program starts by generating the transformation matrix, **T**, which is then used to create the matrices describing the combinational logic, $\mathbf{A}_{Mt}$, $\mathbf{B}_{Mt}$ and



Circuit before transformation. $\mathbf{A}^M$ requires complex combinational logic, which cannot be pipelined because it is in a loop.

Circuit after transformation. $\mathbf{A}_{Mt}$ requires only one logic level of 3-input EXOR gates. $\mathbf{B}_{Mt}$ and $\mathbf{C}_{Mt}$ require complex combinational logic, which can be pipelined.

Figure 8.12: Description of the state-space transformation

$\mathbf{C}_{Mt}$. In this step **T** has to be inverted as well. The inversion of matrices in GF2 is straight forward and implemented by solving the equation system **TX**=**1**.

After having performed the transformation, the C-program generates VHDL code for the critical loop based on $\mathbf{A}_{Mt}$ and VHDL code with pipelining flip-flops for the pre- and post-processing based on $\mathbf{B}_{Mt}$ and $\mathbf{C}_{Mt}$. The program flow is shown in figure 8.13.

8.3.3 Hardware Implementation

The automatically generated VHDL code describes an architecture as the one shown in figure 8.14. I have studied implementations with M=32 and M=64 bits input width respectively. For M=32, the pre- and post-processing logic blocks each have been pipelined into 3 stages. Because the same length of the critical path in all parts of the circuit is desired, each stage consists of either a 3-input EXOR gate or a 2-input EXOR gate. This implies that maximally 27 terms (input bits) can be used in a transformation to compute each modified input bit.

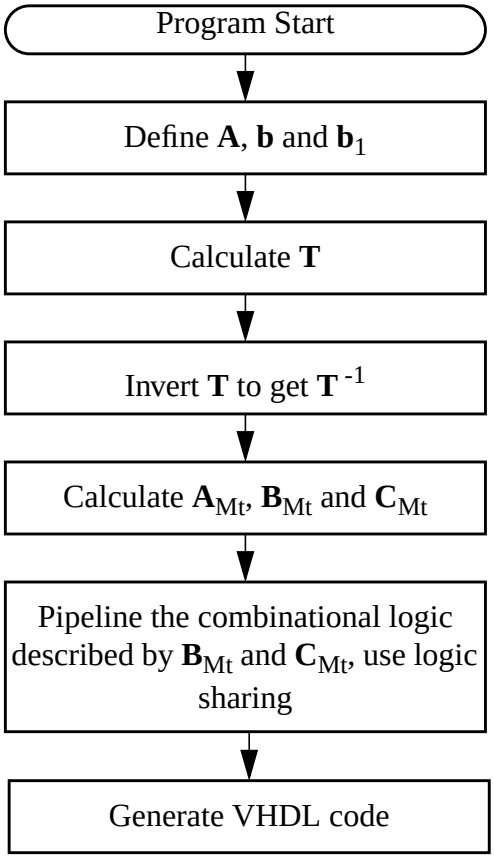


Figure 8.13: Program flow for the C-program that generates the VHDL descrip·tions of the architectures

In the transformations that we have considered, no transformation requires more than 21 terms. If a transformation, that requires more than 27 terms is used, another pipeline stage has to be introduced. In reality as few terms as possible are desired, because that reduces the silicon area and the power consumption. So the maximum of 27 terms is not limiting the interesting design space exploration.

For M=64, the pre-processing was pipelined into 4 stages, since more than 27 terms are used. 4 stages of 3-input EXOR gates allow 81 terms and since M=64 is less than 81 all transformations are possible by using 4 stages. The post-processing logic on the other hand has similar complexity independent of the input width M.

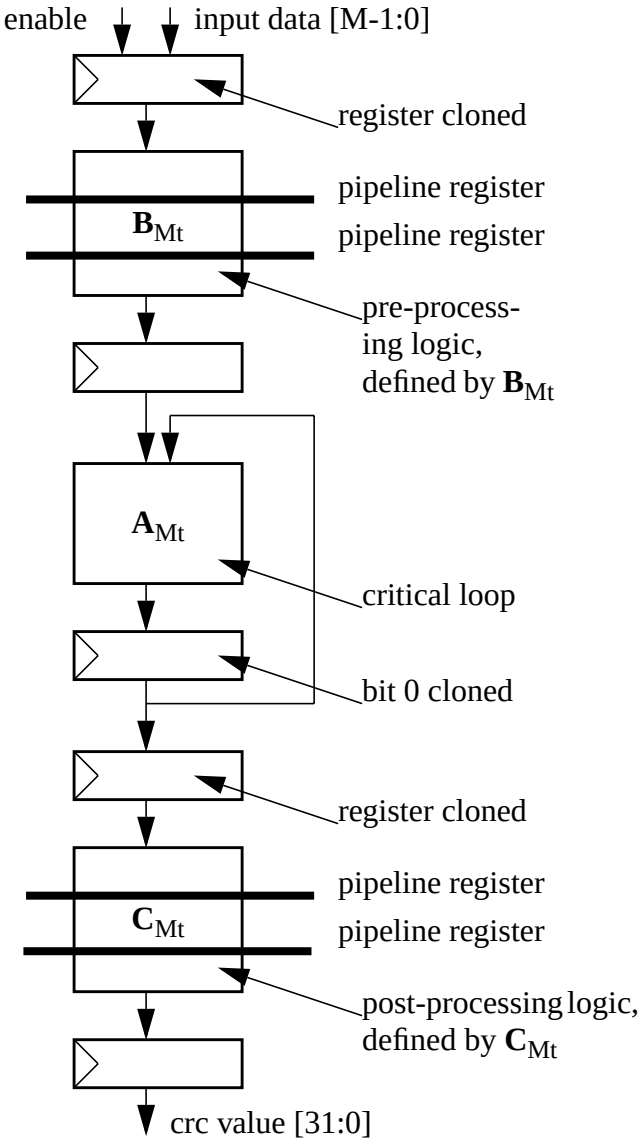


Figure 8.14: General architecture of the CRC circuit after state-space transformation and pipelining

The register between the state register and the post-processing logic block was cloned in order to avoid large fanouts on those flip-flops. The input register was also cloned, because the first stage of the pre-processing similarly created a large fanout otherwise. Without these clones the critical path was situated in the pre- or post-processing logic instead of in the loop. The increased load on the state register was negligible. The least significant bit in the state register was also cloned, because it is used in the computation of several bits in the loop logic and therefore has high fanout. All other bits in the state register are only used in the computation of one bit in the loop logic. This is a property that the selected state-space transformations have. This is also the reason why the loop logic for each bit maximally consists of a 3-input EXOR gate (and a 2-input multiplexer for enable control). All of the implementation details were managed by the C-program, described in the previous section.

The most promising designs, those with the least number of pipeline flip-flops, were chosen for implementation. The VHDL code was simulated in Modelsim for functional verification. The synthesis and placement of standard cells was done in Cadence Envisia PKS for a 0.13 micron standard cell library from UMC. The row utilization was initially set to 0.8.

8.3.4 Implementation Results

32 implementations were generated for both M=32 and M=64. The designs with the least number of pipeline flip-flops were selected for implementation. The number of pipeline flip-flops is a measure of the complexity of the pre- and post-processing logic. The critical path is logically the same in all implementations, that is a 3-input EXOR gate. Therefore it is only interesting to minimize the overall complexity. By doing that it is probable that also the maximum fanout is limited, which is interesting because that will influence the delay in the critical path and thereby the throughput.

A reference design, like the one presented in [8.5] but with 32 bit parallel input, has also been implemented in the same process technology for comparison. Table 8.4 shows the silicon area, the critical path delay, the throughput and the latency for the 3 implemented designs. All values are post-placement values.

After pipelining the pre- and post-processing logic, totally 6 pipeline stages have been introduced in the M=32 case and 7 in the M=64 case. This means that the latency of the circuit is 6 or 7 clock cycles longer than that of a traditional CRC implementation. The reference architecture has one additional clock cycle latency compared to a traditional implementation because of the required 32

zero bits that have to be appended to the input data compared to a traditional design.

The state-space transformation technique improves the performance significantly. This comes to the cost of larger silicon area. The scaling properties are excellent, since the performance scales almost linearly with the input width and the area consumption increases less than linearly. This is because only the preprocessing logic becomes more complex when the input width is increased, the other parts of the design have the same complexity independent of the input width.

8.3.5 Further Discussion

All the implementations discussed above do not consider the fact that an Ethernet frame can consist of any number of bytes. CRC calculation has the property that adding extra bytes of for example zeros to a packet (to fill out the input word of the CRC calculation unit) would change the CRC value. This means that the computation must be capable of handling variable number of bytes. In [8.7] we made an implementation that takes one byte as input every clock cycle in order to avoid that problem. If future network protocols allow packet sizes that are not a multiple of 4 or 8 bytes, a combination of the two architectures can be used where the major part of the packet is managed by the architecture presented here and the last bytes are managed by an architecture that handles one byte at every clock cycle. A small controller is needed to conduct the calculations. That controller has to be designed carefully so that the critical path of the circuit does not appear in the control path.

The circuit that I have designed can handle CRC calculation for the next generation of network protocols at more than 100 Gb/s. This satisfies the needs for near-term future computer networks.

The pre- and post-processing logic consume most of the silicon area. In a full-duplex system two CRC calculation circuits are necessary, one for transmission

Design	Area [mm^2]	Critical path delay [ns]	Throughput [Gb/s]	Latency [clock cycles]
32-bit	0.030	0.45	71	8
64-bit	0.047	0.47	136	9
Reference	0.017	0.93	34	3

Table 8.4: Results of the implementations

and one for reception. The circuit for reception can be simplified by removing the post-processing logic and instead using a comparator to compare the state value directly to $\mathbf{T}^{-1}\mathbf{y}_{correct}$. This would also reduce the latency with 3 clock cycles.

However, low latency is more important in the transmission part, where the CRC value must be appended at the end of the Ethernet frame. If the CRC calculation is done in-line at the same time as the data is transmitted, the data must be delayed by the same number of clock cycles as the CRC calculation requires to complete. This can be managed by pipeline flip-flops or a small FIFO.

The power consumption of the CRC calculation can be reduced by creating an enable signal for the register after the state register. Disabling that register until the state computation is finished will avoid unnecessary toggling in the post-processing logic and thereby save power.

References

[8.1] R. Braden, D. Borman and C. Patridge, “Computing the Internet Checksum”, RFC 1071, September 1998

[8.2] N. Persson, “Specification and Implementation of a Functional Page for Internet Checksum Calculation”, master’s thesis at Linköpings universitet, LiTH-IFM-EX-959, March 2001

[8.3] J. Touch and J. Postel, “Assigned Numbers”, RFC 1700, October 1994

[8.4] T.-B. Pei and C. Zukowski, “High-speed parallel CRC circuits in VLSI”, IEEE Transactions on Communications, Vol. 40, pp. 653-657, April 1992

[8.5] R. J. Glaise and X. Jacquart, “Fast CRC Calculation”, Proceedings of IEEE International Conference on Computer Design: VLSI in Computers, pp. 602-605, 1993

[8.6] J. H. Derby, “High-Speed CRC Computation Using State-Space Transformations”, pages 166-170, Global Telecommunications Conference, 2001.

[8.7] T. Henriksson, H. Eriksson, U. Nordqvist, P. Larsson-Edefors, D. Liu, “VLSI Implementation of CRC-32 for 10 Gigabit Ethernet”, Proceedings of ICECS 2001, vol III, pages 1215-1218, Malta, Sep 2001.

9

Protocol Processor Demonstrator

A demonstrator system was constructed around the intra-PP from chapter 6. The protocol processing partition principle and dual processor architecture described in chapter 5 was used. The demonstrator system receives an audio stream over the university research network and replays the audio on a set of stereo speakers.

9.1 Demonstrator Overview

The demonstrator is implemented on a XSV-300 experimental board from XESS Corp. Except from the board also a piece of software on a Sun workstation, the university fast Ethernet network and a pair of loudspeakers are used for the demonstrator, see figure 9.1.

9.1.1 Demonstrator Purpose

The purpose of the demonstrator is to prove the applicability of the intra-PP in a real system. The intended use for the intra-PP is as a network accelerator in file servers and back-up equipment connected to high-speed networks. Because of the complexity of such a system, the demonstrator was built around audio

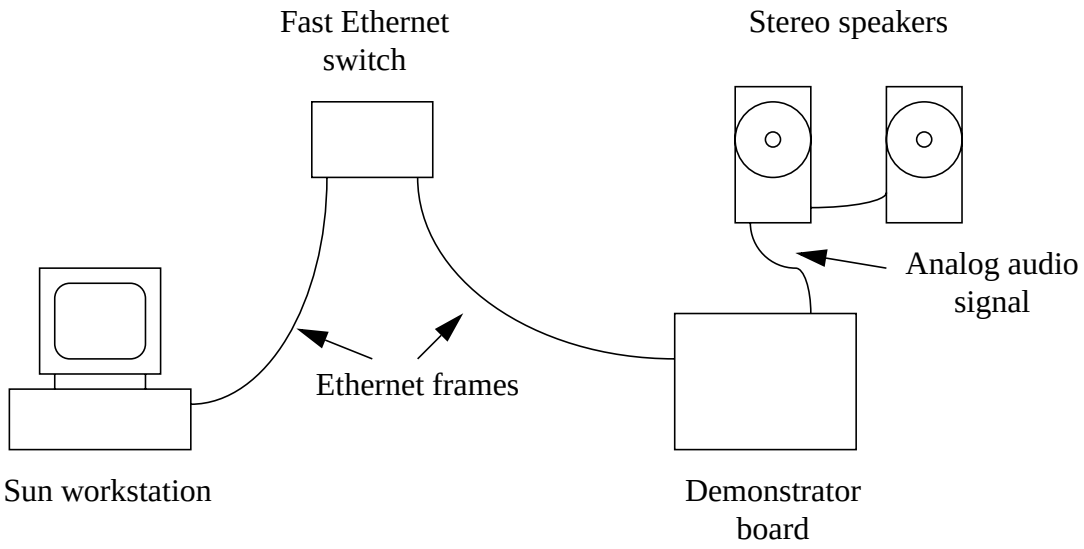


Figure 9.1: Demonstrator system overview

reception instead. The functionality of the intra-PP and its capability to operate in a real system environment is proven also by this simpler demonstrator implementation and a lot of design time is saved.

It would be interesting to build a another demonstrator, where the intra-PP is integrated in a network interface card (NIC) of a file server, for example in combination with the modified host CPU-NIC communication scheme which was suggested in [9.1]. Unfortunately time has not allowed for such an implementation.

9.1.2 Packet Flow

Three types of packets exist in the demonstrator system. An ARP request packet is sent from the Sun workstation to the demonstrator board to find out the hardware (MAC) address of the demonstrator. The demonstrator replies with an ARP reply packet, which tells the Sun workstation how to address the demonstrator. During audio transmission, audio packets are sent from the Sun workstation to the demonstrator. The packet flow is shown in figure 9.2. Audio packets are transmitted in chunks of 256 packets, with only small delay between each of them. The software in the Sun workstation then synchronizes with its real time clock before it transmits the next chunk of 256 packets.

The demonstrator has capability to buffer a little more than 900 audio packets and starts playing back the audio stream when 600 packets have arrived. Each packet contains 9.999424 ms of audio. The audio is sent in raw 16 bit stereo format.

9.1.3 Packet Format

The format of the ARP request packet is shown in figure 9.3. The destination Ethernet address is the broadcast address FF:FF:FF:FF:FF:FF. The source address and the sender addresses are those of the Sun workstation. The target IP address is the IP address of the demonstrator, 130.236.55.05. The operation is 0x1, which specifies request in ARP.

The format of the ARP reply is the same as that of the request, but most of the parameters are different. In the reply, the destination Ethernet address is the hardware address of the Sun workstation and the source address and the sender addresses are those of the demonstrator. The demonstrator has the hardware address 12:23:45:67:89:AB on the university network at Linköpings universitet. This is a local hardware address, so it could be assigned by the network administrator.

The audio packets are transmitted with the user datagram protocol (UDP) on top of IP and Ethernet. The format is shown in figure 9.4. Each packet carries 244 stereo samples, consisting of 16 bits for the left channel and 16 bits for the right channel. The 16 bit sequence number is incremented with one for each consecutive packet and is used in the demonstrator to synchronize the play back of the audio.

The audio packets are created by the Sun workstation by reading a raw audio file. The sample frequency is 24.414 kHz. Coding techniques for the audio have

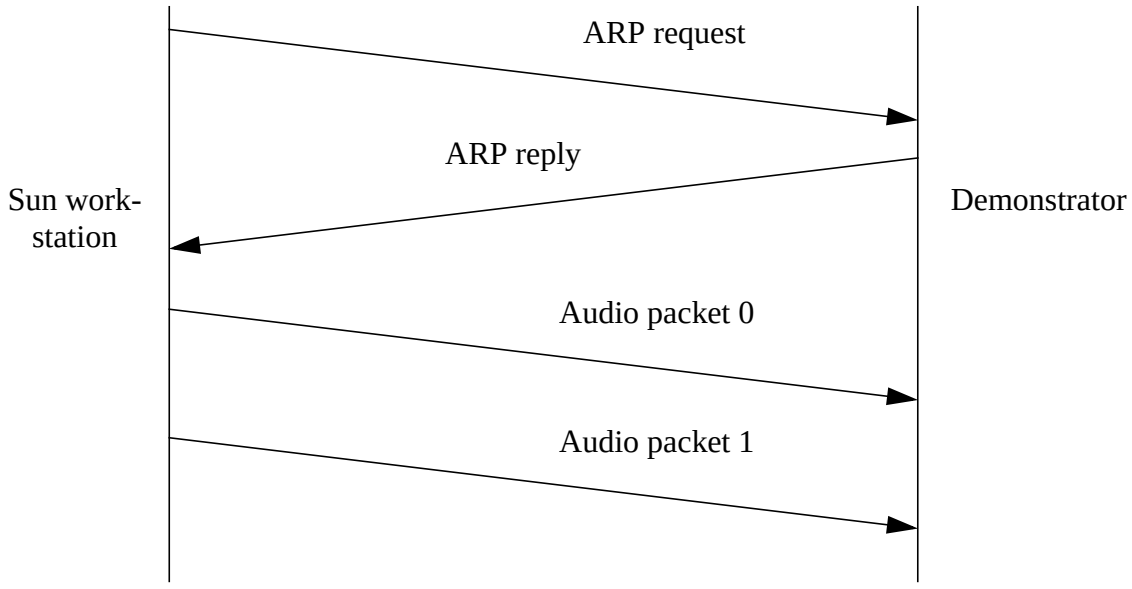


Figure 9.2: General Packet Flow

not been introduced since that would complicate the demonstrator implementation and not benefit the demonstration of the intra-PP functionality.

9.2 Hardware Organization

Figure 9.5 gives an overview of the demonstrator. The Ethernet PHY, the memories, the Stereo codec, the amplifier and the connectors are available on the experimental board as separate components. Everything else (inside the dashed line) is integrated in the Xilinx Virtex 300 FPGA. I have designed and implemented all the virtual components in the FPGA. The Fast Ethernet media independent interface (MII) delivers data 4 bits at 25 MHz. That is converted to 32 bits at 3.125 MHz for the intra-PP.

The main operation of the demonstrator is that the intra-PP decodes the packets that are received on the network and if an ARP packet or an audio packet is received the payload is written in the payload memory in a large circular buffer. The microcontroller handles the replies to ARP requests and the sorting and synchronization of the audio packets. It also sends the audio samples to the stereo codec.

Ethernet Destination Address 47-16		
Ethernet Destination Address 15-0	Ethernet Source Address 47-32	
Ethernet Source Address 31-0		
Ethernet Type (ARP)	Hardware Type (Ethernet)	
Protocol Type (IP)	Hardware Size	Protocol Size
Operation	Sender Ethernet Address 47-32	
Sender Ethernet Address 31-0		
Sender IP Address		
Target Ethernet Address 47-16		
Target Ethernet Address 15-0	Target IP Address 31-16	
Target IP Address 15-0	Padding	
Padding		
Padding		
Padding		
Padding		
CRC		

Figure 9.3: Demonstrator ARP Packet Format

9.2.1 Intra Packet Processor

The purpose of the demonstrator is to prove the functionality of the intra-PP architecture in a real environment. Therefore it could be said that the intra-PP is the core of the demonstrator. The intra-PP in the demonstrator is a scaled down version of the intra-PP described in chapter 7. The reason for scaling it down is that the FPGA has limited capacity and a usage of more than 80% of the slices of the FPGA is not desirable because that creates routing congestion and possibly timing issues.

The intra-PP has 3-way parallelism, that means that there are three comparators, each line in the PCB has three parameters and each line in the CCB has three values. The ILT has been reduced to 64 words. The demonstrator program

Ethernet Destination Address 47-16			
Ethernet Destination Address 15-0	Ethernet Source Address 47-32		
Ethernet Source Address 31-0			
Ethernet Type (IP)	IP ver	IP HL	IP ToS
IP Length	IP Identification		
IP Fragmentation Information	IP TTL		IP Protocol (UDP)
IP Header Checksum	IP Source Address 31-16		
IP Source Address 15-0	IP Destination Address 31-16		
IP Destination Address 15-0	UDP Source Port		
UDP Destination Port	UDP Length		
UDP Checksum	Sequence Number		
Sample 0 Left	Sample 0 Right		
Sample 1 Left	Sample 1 Right		
Sample 2 Left	Sample 2 Right		
Sample 3 Left	Sample 3 Right		
...	...		
Sample 242 Left	Sample 242 Right		
Sample 243 Left	Sample 243 Right		
Ethernet Frame Check Sequence (CRC)			

Figure 9.4: Demonstrator Audio Packet Format

requires only 39 instructions, so that is no limitation for the implementation. The program counter is reduced to 7 bits and so are the values in the CCB. Actually only 6 bits would have been necessary to address the 64 instructions in the ILT. The PCB has 8 lines and the CCB has two lines. That is exactly what is required for the demonstrator program.

The ILT, PCB and CCB are configured via the FPGA configuration instead of by the inter-PP. By doing so they can be mapped to ROM structures in the FPGA. There is also a version of the demonstrator with the ILT, the PCB and the CCB as RAMs, using the configuration interface from the inter-PP. There, the ILT is mapped to a synchronous RAM so that the PC value (the ILT address) is stored internally in the RAM. The PCB and CCB are mapped to asynchronous RAMs. The benefit of mapping the ILT to a synchronous RAM is that the block RAM structure of the FPGA can be utilized. The PCB and CCB have to use distributed RAM structures, that consume logic resources of the FPGA. That is the reason to keep them at minimum size. However, in the final imple-

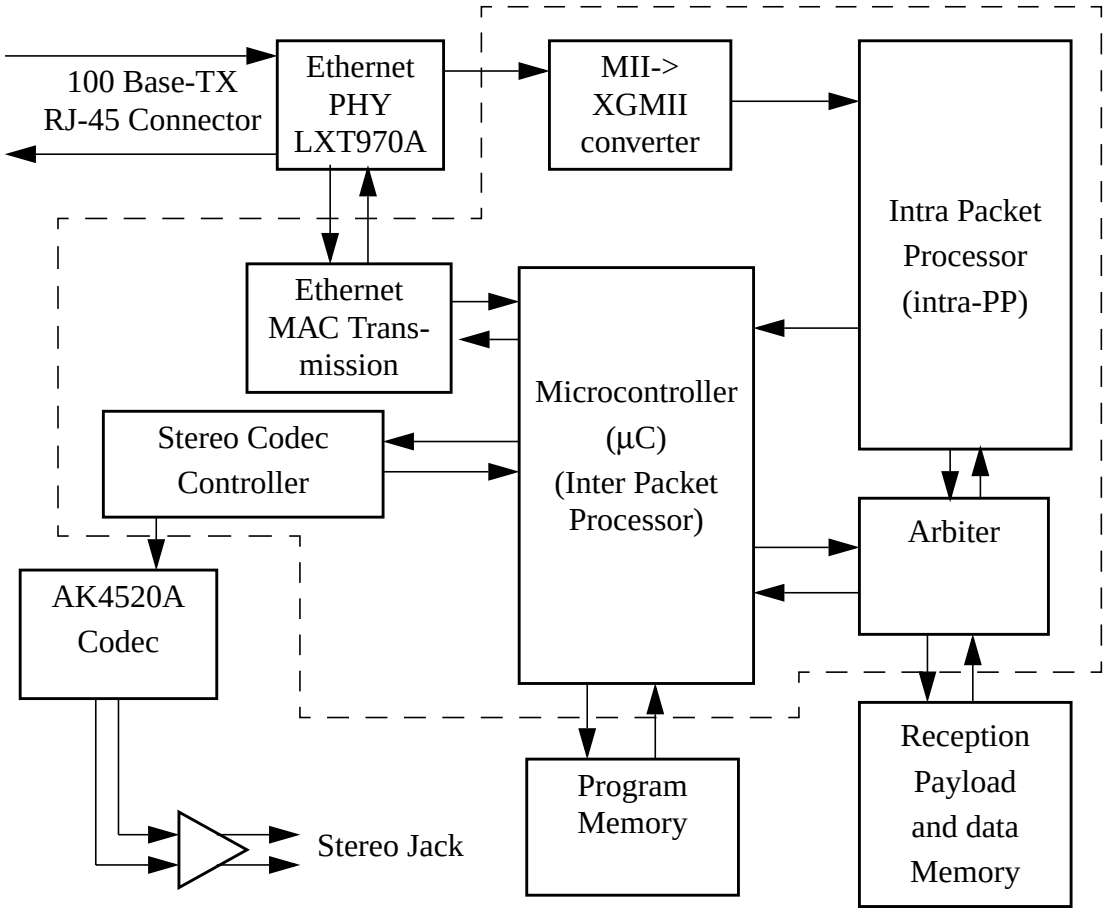


Figure 9.5: Demonstrator Overview

mentation, where the ILT, the PCB and the CCB are mapped to ROM structures, the FPGA allows for larger sizes of the lookup tables than the ones that were actually used.

The accelerators in the intra-PP also have some differences from the ones described in chapter 8. This is because the demonstrator system only uses packets with sizes that are multiples of 32 bits. Therefore the CRC accelerator can be substantially simplified. The UDP checksum accelerator also benefits from this restriction in packet sizes. For the IP header checksum and UDP checksum accelerators, the major difference is the implementation of the one's complement adders. The logic equations, that describe the direct implementation of an ones complement adder, that where used in the implementation in chapter 8, do not synthesize well onto the FPGA. Instead much better implementation results where obtained by using two serially connected two's complement adders.

The memory management unit (MMU) accelerator also had to be modified to fit the demonstrator experimental board. The internal data width of the intra-PP is 32 bits, but the memory interfaces available on the board are only 16 bits. There are two such memory modules available, but only one of them can be used for the circular data buffer, the other is required for the inter-PP program. So the MMU accelerator has to convert from 32 bit data to 16 bit data. This is possible because there is the faster clock available from the MII. So the MMU accelerator runs at 25 MHz and for each 32 bit data word it writes two 16 bit subwords to the payload memory on consecutive positive clock edges.

All the accelerators are controlled from the intra-PP core. The core inputs and outputs are shown in table 9.1 and table 9.2. In some cases direct communication between the accelerators is also allowed, for example for the end of packet signalling.

9.2.2 Microcontroller

A specially designed microcontroller is used as the inter-PP in the demonstrator. The microcontroller is an enhanced version of the microcontroller described in [9.2], which I designed in cooperation with one of my master's students. I decided to design the microcontroller from scratch because it needs some special functionality. The interfaces of the microcontroller are shown in figure 9.6.

The microcontroller has 16 bit wide instructions, because the program memory has a 16 bit wide interface. The data memory is also 16 bits wide, but has a 20 bit wide address, so internally the microcontroller can work on 16 or 20 bit wide data. The 20 bit wide registers are loaded and stored by separate instruction for the 16 least significant bits (lsbs) and the 4 most significant bits (msbs).

The microcontroller runs on the same 25 MHz clock as the MMU accelerator of the intra-PP.

The microcontroller makes use of a Harvard architecture, where the program and data are stored in different memories and use different buses. The microcontroller has a general 2 stage pipeline, with the first stage for instruction fetch

Flag	Purpose	From	To
Input 0	Start	MII->XGMII converter	PP core, CRC accelerator
Input 1	CRC ready	CRC accelerator	PP core
Input 2	CRC OK	CRC accelerator	PP core
Input 3	IP header checksum ready	IP header checksum accelerator	PP core
Input 4	IP header checksum OK	IP header checksum accelerator	PP core
Input 5	UDP checksum ready	UDP checksum accelerator	PP core
Input 6	UDP checksum OK	UDP checksum accelerator	PP core
Input 7	End of Packet	Length Counter Accelerator	PP core, CRC accelerator, UDP checksum accelerator, MMU
Input 8	not used		PP core
Input 9	not used		PP core
Input 10	not used		PP core
Input 11	not used		PP core
Input 12	not used		PP core
Input 13	not used		PP core
Input 14	not used		PP core
Input 15	not used		PP core
Input 16	not used		PP core
Input 17	not used		PP core
Input 18	not used		PP core

Table 9.1: PP core input signals

and decode and the second stage for operand fetch, execution and write-back. The load instructions have a variable latency dependent on the memory arbitration, described in the next subsection. Therefore the microcontroller has a halt

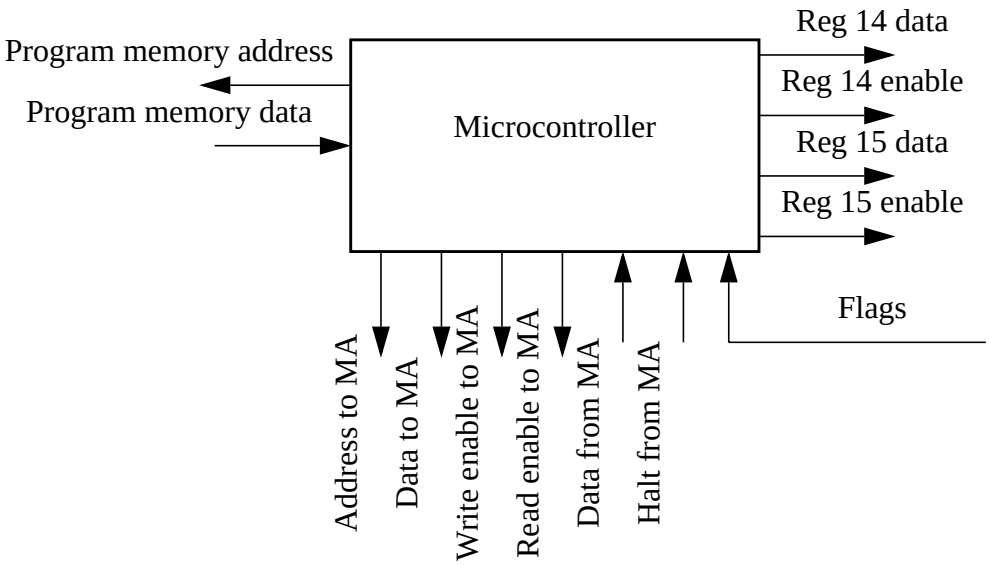


Figure 9.6: Microcontroller interfaces

Flag	Purpose	From	To
Output 0	IP header checksum start	PP core	IP header checksum accelerator
Output 1	UDP checksum start	PP core	UDP checksum accelerator
Output 2	Start IP	PP core	Length counter accelerator
Output 3	Start ARP	PP core	Length counter accelerator
Output 4	Start payload	PP core	MMU
Output 5	Confirm payload	PP core	MMU
Output 6	Discard payload	PP core	MMU
Output 7	ARP packet ready	PP core	External output unit (to µC)
Output 8	Audio packet ready	PP core	External output unit (to µC)
Output 9	not used	PP core	External output unit

Table 9.2: Core output signals

input, that forces the microcontroller to stall when a halt is asserted. The store instructions also have variable latency, which is managed in the same way, by the halt signal. However, the store instructions normally finish in one clock cycle since there is a small write buffer that interfaces between the microcontroller and the memory. Only when the write buffer is full, the microcontroller is forced to stall.

One special feature with the microcontroller is that it has 5 general purpose input flags, that can be used for conditional branches. Two of those are used for the indications on arrived packets from the intra-PP. One of the others is used to indicate that the stereo codec controller needs a new sample.

The data memory address space of the microcontroller is shown in figure 9.7. As can be seen, there are addresses assigned to the intra-PP ILT, PCB and CCB although those are configured as ROMs in the FPGA. So those addresses are not used in the final implementation of the demonstrator. Likewise, only addresses 80010-80011 in the intra-PP are used for parameters. They contain the pointer in the circular buffer to the latest received packet. The SRAM part of the data memory is split up into two areas, the main part is used for the circular buffer and the small upper part is used for the microcontroller to store variables and data structures.

The microcontroller has two special purpose register in the register file of totally 16 registers. The two special purpose registers are used for interfacing to the stereo codec interface and to the Ethernet transmission unit.

9.2.3 Memory Arbitration

The intra-PP and the microcontroller access the same data memory. Therefore there is a need for arbitration. This is handled by the memory arbiter. Because the intra-PP is operating in a real-time environment the memory arbiter gives static priority to it. The microcontroller on the other hand has no hard time limits and does not suffer from a few clock cycles latency.

When there are concurrent memory requests from the intra-PP and the microcontroller, the microcontroller is stalled by asserting the halt signal, discussed in the previous subsection. The halt signal is actually asserted for at least two clock cycles for every load instruction, since a memory access is time consuming and both the address and data have to be pipelined.

A small finite state machine in the memory arbiter conducts the actual arbitration and asserts the halt signal whenever necessary. The memory arbiter runs on the same clock as the microcontroller and the MMU accelerator of the intra-PP so it operates in a fully synchronous environment.

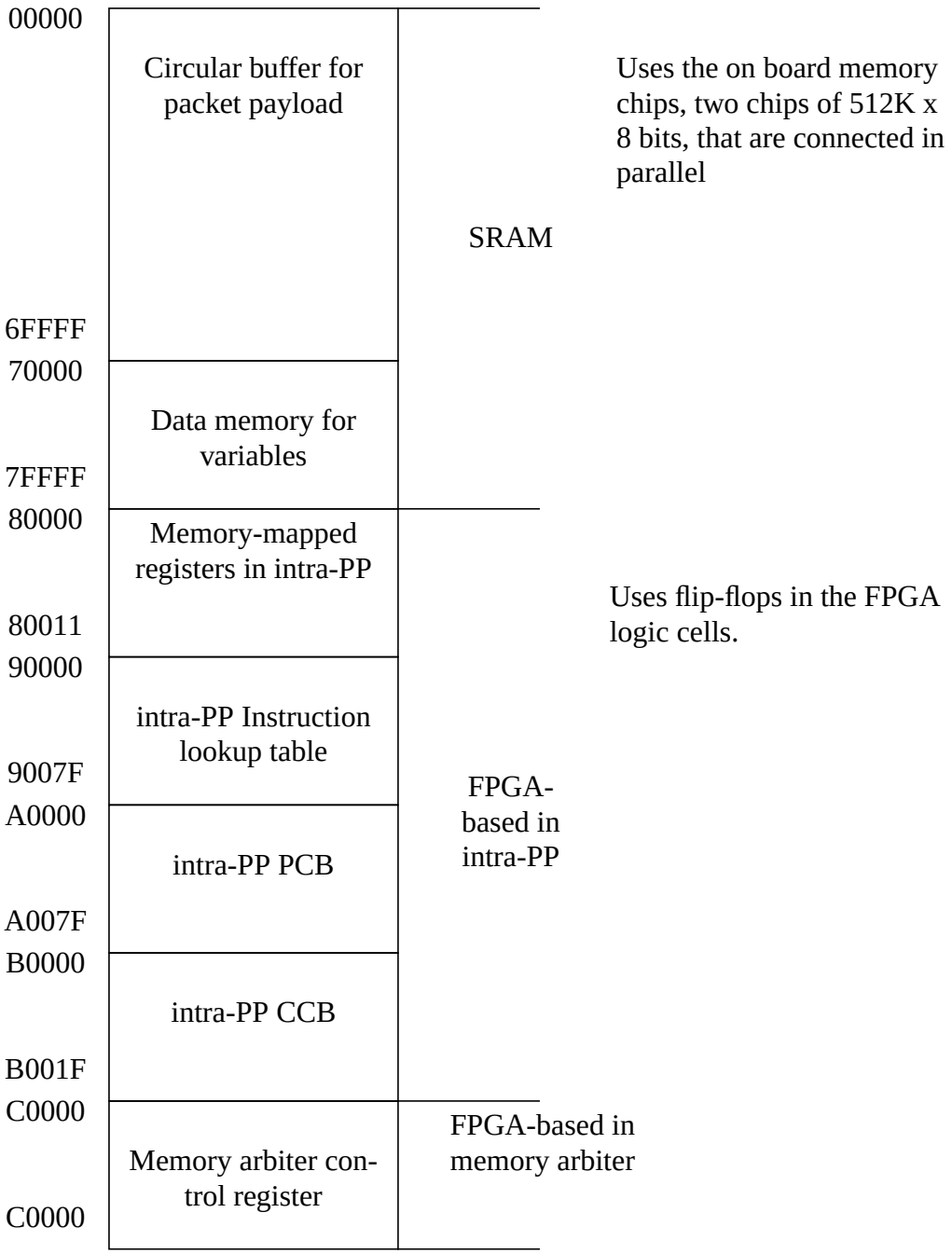


Figure 9.7: Microcontroller data memory organization

9.2.4 Stereo Codec Interface

The stereo codec controller converts the 16 bit stereo samples to serial 32 bit sample format that is required by the stereo codec. The sample frequency is 24.414 kHz. The play back frequency is derived from the 25 MHz Ethernet clock, that is used for the microcontroller. Since there are two samples (left channel and right channel) that must be supplied to the stereo codec with

24.414 kHz, there are 512 clock cycles in the microcontroller between every sample.

To reduce the real-time requirements on the microcontroller, a 2-register buffer is used in the stereo codec interface. This implies that the microcontroller has a window of 1024 clock cycles to input a new sample whenever the stereo codec interface indicates that the buffer is not full.

The stereo codec is used in a mode where it takes 20 bit samples as inputs. In the demonstrator, the stereo codec interface inputs zeros on the lsbs for every sample.

9.2.5 Ethernet Transmission

The Ethernet transmission is asynchronous to the rest of the demonstrator. That is inevitable because the transmission is based on the local oscillator that generates the transmission clock tx_clk and the reception is based on the received clock rx_clk. Therefore the Ethernet transmission part is split into two parts and interconnected with an asynchronous FIFO (first in first out) memory. The first part runs on rx_clk and writes data from one of the special purpose registers in the microcontroller to the FIFO. When a complete packet (60 bytes) has been written into the FIFO an asynchronous strobe signal is asserted.

The second part is triggered by the strobe signal. It runs on the tx_clk and sends an acknowledgment signal back to the first part, which can de-assert the strobe signal. The second part reads data from the FIFO and sends it through a CRC calculation unit to the Ethernet PHY. After the 60 bytes of data, the 4 bytes of CRC are appended to the packet.

The Ethernet transmission is limited to 64 byte Ethernet frames, but that is all that is required for the ARP reply. The interface is easily extendable to other packet sizes, but that was not necessary to implement for the demonstrator.

9.3 Implementation

For the demonstrator, three major parts are necessary. The FPGA configuration, the intra-PP software and the microcontroller software.

9.3.1 FPGA Configuration

The whole FPGA-part of the demonstrator is described in VHDL with appropriate instantiations of Virtex specific units, such as ROMs, FIFO and buffers. These unit where also modelled in VHDL for the purpose of system simulation in a general event-driven VHDL simulator.

The VHDL-code was synthesized and mapped onto the FPGA by Xilinx tools, which also generated the configuration file. The demonstrator utilizes 71% of the available logic resources in the Virtex 300 FPGA.

The configuration file is transferred to the FPGA via the parallel cable from a PC.

9.3.2 Intra-PP Software

As described earlier, the intra-PP software is part of the FPGA configuration file, but nevertheless a separate implementation flow was needed to create that software. There does not exist any compiler for the intra-PP, so the software must be coded in assembly language. Then a manual conversion to binary code, described in hexadecimal format in a text file is necessary.

The text file is automatically converted to binary format by a special piece of software on the Sun workstation. Finally that binary description is incorporated in the VHDL description for synthesis into a ROM structure in the FPGA.

9.3.3 Microcontroller Software

The microcontroller software resides in the program memory, which is of the same type as the data memory. It is transferred to the memory via the parallel cable from a PC. Then the software is in binary format.

There is no compiler available for the microcontroller, so its software must be written in assembly language. The assembly program is automatically converted into binary format by an assembler.

There is an instruction set simulator, that can simulate the execution of the microcontroller software and produce detailed register content reports for every clock cycle. That simulator is important for the software verification as well as for the verification of the VHDL description of the microcontroller.

9.4 Results

The demonstrator that was designed around the intra-PP works well. Several audio files have been sent over the network and have been correctly received, decoded and played back by the demonstrator.

The successful implementation of the demonstrator has proven the applicability of the Linkoping architecture in the shape of the intra-PP in a real system environment.

References

[9.1] Philip Buonadonna and David Culler, “Queue Pair IP: A Hybrid Architecture for System Area Networks”, International Symposium on Computer Architecture 2002, pp. 247-256, June 2002, Anchorage, Alaska

[9.2] K. Martinsson, “Design of Application Specific Microcontroller”, master’s thesis at Linköpings universitet, LiTH-ISY-EX-3283-2002, November 2002

10

Conclusions

There are very many conclusions drawn during the work with this thesis. Many of them can be found in the publications that are the basis for this thesis. Only the major achievements are mentioned here.

10.1 Achievements

Partitioning protocol processing into two groups of tasks, intra-packet tasks and inter-packet tasks, allows for an efficient dual processor implementation. A protocol processor that operates directly on the received data stream has been implemented and proved to be able to execute the intra-packet tasks.

The intra-packet tasks of protocol processing consists of regular and irregular tasks. The regular tasks can be efficiently executed on accelerator units, that operate in parallel with the core of the protocol processor.

10.2 Suggestions for Future Work

The work on programmable data stream processors is in its infancy. There is still much to be done. The protocol processor should be investigated with more protocol stacks, varied parallelism, varied data word width and dynamic input buffer size for example. The possibility to use similar architectures for other data stream application than networking should be thoroughly investigated.

A

Acronyms

The following generally accepted acronyms are used in the thesis and provided here to make the reading of the thesis easier.

AAL	ATM Adaption Layer
ADSL	Asynchronous Digital Subscriber Line
ATM	Asynchronous Transfer Mode
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
CIDR	Classless Interdomain Routing
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DFG	Data Flow Graph
DMA	Direct Memory Access
DSP	Digital Signal Processor
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GB	Gigabyte

Gb/s	Gigabit/second
GF2	Galois Field 2
ID	Instruction Decoder
IP	Internet Protocol
IPv6	Internet Protocol version 6
IPC	Instructions Per Clock
IPMA	Internet Performance Measurement and Analysis Project
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
kbits	kilobits
LAN	Local Area Network
LFSR	Linear Feedback Shift Register
lsb	least significant bit
MAC	Medium Access Control
MAC	Multiply and Accumulate
MB	Megabyte
MHz	Mega hertz
MII	Media Independent Interface
µm	micro meter
MMU	Memory Management Unit
MPackets/s	mega packets/second
ms	milisecond
msb	most significant bit
NIC	Network Interface Card
NP	Network Processor
ns	nanosecond
OS	Operating System
OSI	Open Systems Interconnect
PC	Program Counter
PHY	Physcial Layer
QoS	Quality of Service

PE	Processing Element
RF	Register File
ROM	Read Only Memory
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TOE	TCP Offload Engine
TTL	Time To Live
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLIW	Very Long Instruction Word
XGMII	eXtended Gigabit Media Independent Interface

B

Glossary

The following technical terms and expressions from the areas of communication, computer networks, digital circuit design and computer architecture are used in the thesis and explanations are provided here to make the reading of the thesis easier. Some of the explanations are inspired from [B.1] and [B.2].

AAL **ATM Adaption Layer**

ATM formats that specify constant or variable bit rate and connection oriented or connection-less mode.

ATM **Asynchronous Transfer Mode**

A protocol for transmitting flows with variable bandwidth requirements. The data is divided into 48 byte cells, which are combined with a header to form the 53 byte cells which are transferred by ATM.

ARP **Address Resolution Protocol**

A protocol that is used to derive the hardware (MAC) address starting from the IP address. ARP uses broadcast packets.

ASIC **Application Specific Integrated Circuit**

Integrated Circuits (ICs) customized to perform a specific task - as opposed to general purpose microprocessors or DSPs.

ASIP **Application Specific Instruction set Processor**

A processor that has an instruction set which is optimized for a certian domain of applications.

BIT

Binary character consisting of one of two possible values, 0 or 1.

BROADCAST

A service with one transmitter and many receivers, where all receivers connected to the network receive the message, often by use of a broadcast address. (Compare to Multicast, where a subset of the receivers are addressed.)

BYTE

A group of eight bits that is processed as a single logical unit.

CIDR Classless Interdomain Routing

The routing scheme that is used in the Internet. It took over after the class based routing scheme failed to supply addresses to all medium-sized companies and organization in an efficient way.

CPU Central Processing Unit

The core part of a computer that executes the programs. Also referred to as processor.

CRC Cyclic Redundacy Check

A type of block check character that is very effective in detecting communications errors. CRC characters are usually 12, 16, 24 or 32 bits long.

DMA Direct Memory Access

A way to offload block memory transfers from the CPU to an accelerator.

DSP Digital Signal Processor

A type of processors specifically targeted to signal processing applications.

GB Gigabyte

1 GB=1024 MB (megabyte)=1024^2 kB (kilobyte)=1024^3 byte. A unit for measuring data storage capacity.

Gb/s Gigabit/second

10^9 bit/second, a unit for measuring network capacity.

ID Instruction Decoder

The part of the processor that decodes the instructions into control signals. The control signals are used to control the operation in detail.

IP Internet Protocol

The by far most important protocol on the network layer.

IPv6 **Internet Protocol version 6**

The successor of IP. IPv6 provides more addresses when all IP addresses are taken.

IPC **Instructions Per Clock**

A measure on how efficient a microarchitecture can execute a certain instruction stream.

ISA **Instruction Set Architecture**

The programmer's view of a processor. The ISA is the interface between the hardware that constitutes the processor and the software that can be executed on it.

FIFO **First In First Out**

A type of storage element (memory) where the address is implicit in a way that the oldest data will be read next.

LAN **Local Area Network**

As defined by IEEE Committee 802.6: A non-public data network in which serial transmission is used without store and forward techniques for direct communication among data stations on a user's premises. Examples are ethernet (802.3) and token ring (802.5).

lsb **least significant bit**

Lowest order bit in the binary representation of a numerical value. The lsb has the least impact on the value of the number which is digitally represented.

MAC **Medium Access Control**

Part of the data link link layer (layer 2) that interface to the physical layer.

MAC **Multiply and Accumulate**

A common operation in digital signal processing, which has its own instruction in many DSPs and also its own execution unit, all with the same name.

MB **Megabyte**

1 MB= 1024 kB(kilobyte)=1024^2 byte. A unit for measuring data storage capacity.

MHz **Mega hertz**

1 MHz = 10^6 Hz. A unit for measuring frequency.

MII **Media Independent Interface**

The interface between the PHY and the MAC in Ethernet and fast Ethernet.

μm **micro meter**

1 μm = 10^{-6} m. A unit for measuring distance.

msb **most significant bit**

Highest order bit in the binary representation of a numerical value. The msb has the most impact on the value of the number which is digitally represented.

MULTICAST

A service with one transmitter and more than one addressed receiver. (Compare to broadcast - With broadcast, all receivers on the network are addressed. With multicast, a subset of the receivers are addressed.)

NETWORK

A set of terminals, the communications links that joint them, and the protocols that allow them to function together and communicate with each other.

NETWORK LAYER

Layer 3 of the OSI model. It defines how data packets are switched and routed through the network.

NIC **Network Interface Card (Controller)**

An interface that is usually located within a terminal and which connects a LAN to the terminals address, data and control buses.

NP **Network Processors**

A type of processors specifically targeted to network applications.

ns **nano second**

1 ns = 10^{-9} s (seconds). A unit for measuring time.

OS **Operating System**

The program that manages the execution of the applications in a computer.

OSI **Open Systems Interconnect**

International Standards Organization (ISO) model of how data communications systems can be interconnected. Communication is partitioned into seven functional layers. Each layer builds on the service provided by those under it.

PACKET

A grouping of data, typically from 1 to 1500 characters in size, which usually represents on transaction. A packet is always associated with an address header and control information.

PHY **Physcial Layer**

The lowest layer (layer 1) of the OSI Model that defines the physical medium for data communications.

PROTOCOL

In general, any agreement that facilitates communications. In data communications, a public (standard) or private (proprietary) specification for communications between peer layers in a layered architecture.

QoS **Quality of Service**

Quality of service in computer networks means that different connections or flows can get different priority. For example QoS can be used to guarantee minimum bandwidth for real-time traffic.

PC **Program Counter**

A storage element that stores the address to the current instruction in the program memory.

RF **Register File**

The register file is the primary computing buffer of a processor. Normally operations can only be conducted on values which are stored in the register file.

ROM **Read Only Memory**

A storage element which cannot change content.

ROUTER

A device that connects two or more LANs to each other and that operates at OSI Model layers one through three. A router is able to select among multiple paths to route a data packet through the network based on an address sent with the data.

ROUTING TABLE

A table of the addresses of the various nodes on the LANs served by a bridge or other internet working device. The routing table allows frames to be forwarded to the LAN where their destination node is located.

RTL **Register Transfer Level**

An abstraction level for describing the functionality of digital circuits. RTL describes the registers in the design and how data can move between them.

SIMD **Single Instruction Multiple Data**

A classification of computers. In a SIMD system one instruction stream applies to many data streams.

SRAM **Static Random Access Memory**

The fastest type of computer storage for information.

Superscalar

A type of processors that can execute more than one instruction per clock cycle by having multiple execution units. The scheduling of instructions is done in hardware at runtime.

TCP **Transmission Control Protocol**

A transport protocol that delivers reliable end-to-end communication by the use of connections. TCP is often used with IP, TCP/IP.

TERMINAL

The device on a network that sends or receives data. A terminal is often a computer.

TOE **TCP Offload Engine**

A device that is used to fully or partly offload the TCP processing from the host CPU.

TTL **Time To Live**

A header field in the IP header that was initially meant as the time to live for a packet in seconds, but due to the implementation problem is used as a counter of number of hops to live.

UDP **User Datagram Protocol**

A transport protocol that delivers unreliable best-effort communication.

VHDL **VHSIC Hardware Description Language**

A description language for digital circuits. VHDL can be used to write RTL descriptions of circuits.

VLIW **Very Long Instruction Word**

A type of processors that can execute multiple operations per clock cycle. The scheduling of operations is done in the compiler.

XGMII **eXtended Gigabit Media Independent Interface**

The interface between the PHY and the MAC in 10 Gigabit Ethernet.

References

[B.1] Longman Comprehensive Dictionary of Computing, Longman Asia Limited, ISBN 962 359 417 8.

[B.2] “Glossary of Communication Terms”, Intersil Application Note AN9640.1, December 1996.