TACO IPv6 Router - a Case Study in Protocol Processor Design

Virtanen, Seppo; Truscan, Dragos; Lilius, Johan

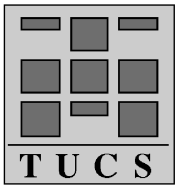
Published: 01/01/2003

Document Version
Final published version

TACO IPv6 Router - a Case Study in Protocol Processor Design

Seppo Virtanen
Dragos Truscan
Johan Lilius

Embedded Systems Laboratory



Turku Centre for Computer Science
TUCS Technical Report No 528
April 2003
ISBN 952-12-1166-0
ISSN 1239-1891

Abstract

In this report we present the TACO protocol processor platform and its use in application-specific processor design. We discuss a case study, in which we designed an IPv6 router processor on the TACO platform. IPv6 is the latest generation of the Internet Protocol (IP) introduced to overcome address restrictions of the current version of the Internet Protocol.

The TACO platform is based on transport trigger architectures (TTA), in which data transports trigger processor operations. In TACO processors, all the operations are protocol processing related tasks. A major advantage of using TTA as the base architecture for TACO is its support for design automation achieved through modularity.

Keywords: protocol processor, processor architecture, transport triggered architecture, application-specific instruction-set processor, internet protocol version 6, IPv6 router

TUCS Laboratory
Embedded Systems Laboratory

Acknowledgements

The authors wish to thank M.Sc. student *Jani Paakkulainen* (University of Turku) and PhD student *Tomi Westerlund* (TUCS) for their comments regarding some of the hardware solutions suggested in this report.

Seppo Virtanen gratefully acknowledges financial support from the *HPY research foundation* and from the *Nokia foundation*.

Contents

1 Introduction

Network hardware design is becoming increasingly challenging because more and more demands are put not only on network bandwidth and throughput requirements but also on a device's time-to-market. Using current standard techniques like general purpose microprocessors and ASICs, these goals are difficult to reach simultaneously. General purpose microprocessors are no longer an appealing alternative for networking hardware on account of their lack of optimized execution units for network processing. All the networking functionality must be implemented in software, which in turn leads to high CPU clock frequency requirements. A general purpose processor in the speed range is going to be expensive or may not even be available. Also, many general purpose processor features, like floating point units, can usually not be taken advantage of in networking applications. For these reasons among others, ASICs have been widely used for networking devices. ASICs can provide more processing speed with lower clock frequency than general purpose processors. However, ASIC design is difficult and expensive, and the time-to-market for an ASIC tends to be long. Also, ASICs are usually not programmable and thus need to be redesigned for updated or new network protocols, making them inflexible in dynamic market segments.

One solution to this problem that has recently attracted interest is the design of programmable processors with network-optimized hardware, that is, network or protocol processors. Such a processor is an attempt to harness the processing speed of ASICs and the programmability of general purpose processors for optimal protocol processing speed. The challenge in designing such protocol processors is finding an architecture that is a good compromise between a general purpose processor and a custom, protocol-specific ASIC. Ideally the hardware architecture should be optimized for protocol processing while it would still provide flexible programmability.

1.1 The TACO Project

In our research project TACO (Tools for Application-specific HW/SW Co-design) we are developing a framework for designing programmable protocol processors. Within this framework we suggested a protocol processor architecture platform in a conference paper in 1999 [25]. A proof-of-concept case study on the platform was later described in a master's thesis [27]. This technical report gives an updated and more detailed description of the TACO protocol processor platform. We also discuss a more demanding case study in configuring the platform to meet the requirements of a protocol processing application (IPv6 routing).

The underlying base microprocessor architecture for the TACO processors is TTA (Transport Triggered Architecture) [6, 22]. TTA processors reverse the traditional programming paradigm: in a TTA processor data transports are programmed and operations are performed on the data as a side effect of the transports (i.e. a data move to a certain register *triggers* an operation). Traditionally the operations would be programmed and data transports would be performed as a side effect. TTA processors are formed of functional units (FUs) that carry out the triggered operations, a set of buses (called the interconnection network) that transport the data between the functional units, and control structures.

In TACO processors the functional units are designed and optimized for protocol processing [23]. In this sense they can be considered as ASIPs (Application-Specific Instruction-Set Processors), although in TACO the code blocks that have optimized execution in hardware are larger than in traditional ASIPs (in which the size of such code blocks can be as low as 2-3 instructions [3, 19]).

This report starts with an introduction to TTA. We also discuss briefly the advantages gained from using it as the base architecture for TACO. Then we discuss hardware details of the TACO protocol processor platform. Finally, we give an overview of IPv6 and IPv6 routing followed by a description of the IPv6 router processor architecture.

2 TTA Architecture

This section gives an overview of transport triggered architectures (TTAs). In this and the following sections we will use the term *"base TTA"* when referring to the MOVE TTA framework presented in [6].

TTAs perform operations on the data as side effects of data transports. For this reason, TTAs can be seen as OISC (One-Instruction Set Computer) type processors: the programmed transport, also called the *move* operation, is the only programming construct available at the machine code level in a TTA processor.

A TTA processor is formed of functional units (FUs) that communicate via an interconnection network of data buses, controlled by an interconnection network controller unit. The FUs connect to the buses through modules called sockets. Each functional unit has input (operand and trigger) and output (result) registers, and each register has a corresponding socket. FU operations are executed every time data is moved to a specific kind of input register, the trigger register. The number of transport buses and the number and type of FUs depends on the target application and usually also

on design constraints for physical characteristics like clock frequency, power consumption, chip area etc.

Functional Units An FU has a set of addressable locations, registers. An addressable location is the source or destination of a move operation. Every location has either one physical source ID or one physical destination ID. However, several logical IDs can be mapped onto a single FU register. The logical IDs are used for operation selection: an FU may provide more than one operation, and a logical ID specifies which operation should be used. The main types of FU registers are *input* and *output*, used for inputting data to or outputting data from the functional unit. There are two subtypes of input registers, namely *operand (OP)* and *trigger (TR)* registers. Output registers are called *result (R)* registers.

The difference between operand and trigger registers is that a data transport to a trigger register triggers an FU operation. The FU operation uses the transported data word as an operand. The operand registers are used for inputting additional operands (for operations that need more than one value to compute, e.g. addition). For such operations, data needs to be transported to the operand register prior to triggering; data transports to operand registers do not trigger operations. The results of FU operations, if there are any, are stored in one or more result registers.

The MOVE framework [6] distinguishes between two classes of functional units: FUs and SFUs. FUs implement regular, commonly used operations like ALU functions (in fact, an ALU can be considered to be an FU). Typically the FU operations resemble operations performed by general purpose processors. In contrast SFUs, or Special Functional Units, perform operations that are application-domain specific and are not often executed in general purpose processing.

Sockets A socket is a gateway between the interconnection network and a functional unit. Each socket is connected to one or more buses and to one FU register. A socket can pass one data word per clock cycle to the FU it is connected to.

An input socket evaluates if a destination identifier on a bus connected to it matches its own identifier. If the identifiers match, the socket passes data from the bus on which the identifier was found to the FU (more precisely, to the register in the FU to which the socket is connected). The number of destination identifiers a single socket can recognize is not limited to one. If the socket has more than one identifier, usually an *opcode* is extracted from the identifier. This opcode is passed to the FU at the same time as the data

is, and the operation performed by the FU on the data is specified by the extracted opcode.

Trigger sockets are a special kind of input sockets. Trigger sockets function like input sockets, but upon passing the data from the bus to the FU the trigger socket also signals the FU to start executing its operation. Data transfers through regular input sockets do not cause FU operations to start executing.

An output socket is similar in implementation to an input socket. It compares the source identifier(s) on the connected source bus(es) to its own identifier(s) and if there is a match, the possible opcode is extracted and data is passed from the FU to the bus on which the identifier was found. Also the output sockets can have more than one identifiers.

Interconnection Network By changing the type and number of FUs and by changing the connectivity and capacity of the interconnection network, a wide range of processor architectures can be specified. Since the number of FUs and buses in the interconnection network is not restricted and the design of these elements is independent, TTA is quite a flexible platform in terms of hardware design. There are practically no constraints on designing the interconnection network and different kinds of FUs as long as both are in accordance with the socket interface specification.

In base TTA each bus on the interconnection network actually consists of data, address (source and destination) and control buses. Source and destination buses transport the *move* instructions to the sockets and data buses transport data from one FU to another. Control buses are used, among other things, for protecting unfinished execution and for conditional execution.

The interconnection network can be partly connected (each socket connects to only some of the buses), or fully connected (each socket is connected to every bus). If the interconnection network is fully connected, each register in each FU can move data on any of the buses, thus making code generation for the processor easier. The number of buses in a processor is not restricted but the size of the instruction word limits the reasonable amount of buses to less than ten [6]. Also, the power needed to drive the buses increases with the number of buses.

Programming TTAs TTA is a modified VLIW (Very Long Instruction Word) architecture that does not feature logic for execution optimization, i.e. run-time instruction reordering to improve concurrent use of functional units etc. Instead, TTA processors rely on the program compiler to perform instruction scheduling in an optimal way.

TTA instructions resemble VLIW instructions. They consist of several RISC type subinstructions that each define a data transport by specifying a source and a destination socket address for the data bus in question. The subinstructions also include a guard identifier for specifying conditional data moves; if the condition specified in the guard identifier is not met, the data move is cancelled.

2.1 TTAs and Design Automation

The TTA architecture provides modularity and scalability to processor design. Functional units can be added to the architecture or they can be refined and changed as long as they provide the same interface to the sockets connecting them to the interconnection network. The same holds naturally for the interconnection network. With TTA hardware architectures become simpler to design and implement since many traditional logical tasks (e.g. program code scheduling for optimal execution) are left to be taken care of by the program compiler instead of the processor. One of the most important contributors to overall algorithmic performance in TTA processors is a well designed program compiler.

The design of new functional units is straight-forward, since the general functionality and connectivity is very similar from one functional unit type to another. For this reason, it is possible to construct a library of components written in a hardware specification/description language, from which modules can be selected to be used in a particular processor architecture instance. This is in fact the case in our TACO framework: we have created component libraries in SystemC [17] and VHDL from which we select components to form architecture candidates for design space exploration.

3 TACO Protocol Processor Architecture

As mentioned earlier, the TACO platform is based on the *base TTA* architecture [6]. However, there are some fundamental differences between TACO and the base TTA architecture. Analyzing these is beyond the scope of this document, so we will limit ourselves to a brief description of the key differences. Many of the differences are simplifications in TACO when compared to base TTA; we believe that it is beneficial to reduce hardware complexity and to leave as much of the "executional intelligence" as possible into the program code compiler. This means among other things, that the compiler is responsible for scheduling the code in a way that eliminates hardware access conflicts and the need for run-time optimizations and checks.

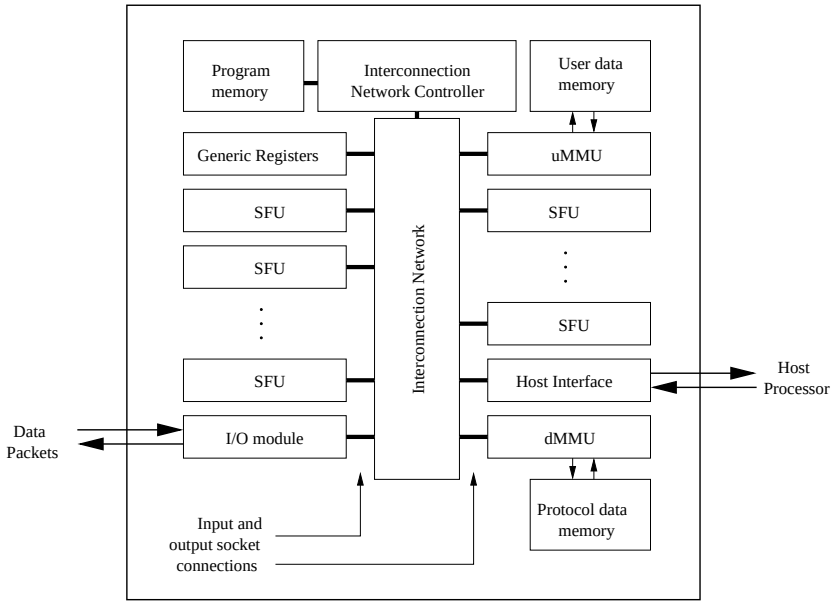


Figure 1: A generic TACO protocol processor (functional view).

A summary of the key architectural differences between TACO processors and the MOVE32INT processor (presented as an example TTA processor in [6]) is given below.

- TACO processors have only SFUs (Special FUs), no regular FUs. Each TACO FU performs a protocol processing task. For example, there is no ALU FU in a TACO processor. For simplicity, from here on we will use the terms "functional unit" and FU when referring to the TACO special functional units.

- TACO FUs execute their operations in one machine cycle. This limitation makes code scheduling much easier, but it may have to be lifted in the future to allow more complex operations to be performed.

- TACO processors have a four stage pipeline, the MOVE32INT has a three stage pipeline.

- TACO processors do not have control buses. This means that the signals Global Lock (GL), Local Lock Request (LL) and Squash (SQ) do not exist in TACO processors. The functionality provided by these signals is provided to TACO processors in part by the program compiler, in part by the four stage pipeline, and in part by the Interconnection Network Controller.

- In TACO processors the interconnection network is fully connected. This is not a strict requirement, but it eases code generation and scheduling.
- TACO processors have no general purpose registers. The MOVE32INT processor has 11 general purpose registers.
- TACO processors have three separate memories: Program memory (for the program code), Protocol data memory (for storing/retrieving protocol data units), and User data memory (for storing/retrieving user data). The MOVE32INT implements a traditional Harvard architecture with separate program and data memories.

Figure 1 shows a functional view of a generic TACO protocol processor architecture. The functional units marked with "SFU" are the special protocol processing units. Their type varies from one protocol and/or application to another, and therefore the types are not specified in this figure.

In TACO processors each functional unit implements a particular protocol processing task. The method for selecting tasks for each FU was initially to analyze commonly used communications protocols and certain protocol processing applications [23]. A recently introduced application analysis method [11] is expected to provide a more formal way of suggesting tasks to be implemented as FUs.

3.1 Interconnection Network

The interconnection network is formed of one or more data buses and the same number of SRC and DST buses. In TACO processors the interconnection network is fully connected, i.e. all buses have connections to all sockets. The number of possible data moves in one clock cycle equals the number of data buses in the interconnection network. Full connectivity of the interconnection network makes automated hardware and software generation less demanding and ensures maximal use of bus bandwidth (with partial connectivity, situations in which a bus is idle, but can not be used due to a lack of necessary connections, may arise).

3.2 TACO Instruction Word

The TACO processor architecture is not limited to specific bus configurations or data word lengths. However, for each architecture instance the processor designer has to make decisions on the number of buses to have in the interconnection network, and on the data word length of the processor. These

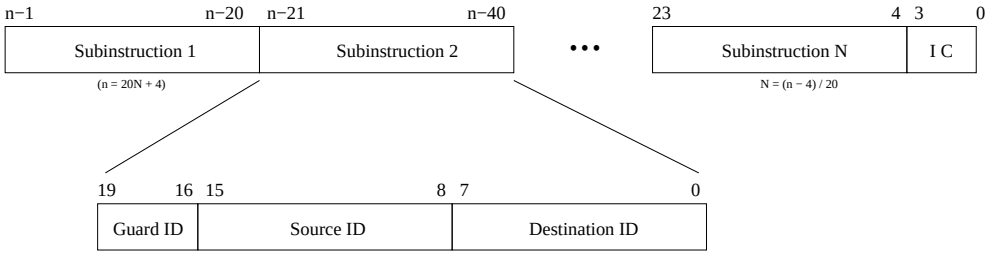


Figure 2: TACO protocol processor instruction word (N buses).

decisions then effect the instruction word length of the processor (the instruction word must be long enough to provide source and destination identifiers to all the buses).

As seen in Figure 2, a TACO instruction word consists of 20-bit subinstructions and a four-bit immediate control (IC) field. Hence, a processor with one bus in the interconnection network has the instruction word length of 24 bits, whereas a processor with 6 buses has the instruction word length of 124 bits.

Each subinstruction specifies a data move for its corresponding bus (Subinstruction 1 defines a data move for bus 1 and so on). Each subinstruction is constructed of a four-bit Guard ID (GID), an eight-bit Source ID (SRC) and an eight-bit Destination ID (DST). The source and destination IDs define the addresses from/to which data is moved. The addresses refer to registers in functional units.

Guard ID The guard ID is used in defining conditional execution: if a non-zero GID exists in a subinstruction, the data move specified by SRC and DST may be carried out only if the logical condition specified by the GID exists in the processor. Such a logical condition could be for example a boolean false result from two specified functional unit operations. The functional units report these logical conditions to the interconnection network controller by using special one-bit signals called "guard signals".

Immediate integers and IC bits TACO processors support eight-bit immediate integer generation. Immediate integers are specified in program code using the four IC bits in the TACO instruction word (see Figure 2). These bits specify the subinstruction (and hence the bus) that contains an immediate integer in place of an SRC identifier. Thus, generating and dispatching an eight-bit integer does not have an effect on the number of available data transports per cycle. The suggested way of using larger than eight-bit integer

data values in TACO processors is to initialize the required number of User data memory locations (see Figure 1) with needed values.

3.3 Interconnection Network Controller

The structure of the interconnection network controller is reasonably simple because it does not include any logic for execution optimization, e.g. dynamic scheduling. The instruction scheduling is done already at the assembler code level, since the assembler code itself is a list of data moves. The network controller has no support for operating system functions such as virtual memory and multitasking.

The key tasks the Interconnection Network Controller performs are:

- fetching instructions from the Program memory
- maintaining the Program Counter (PC)
- evaluating guard signals and guard IDs for conditional execution
- splitting long instruction words into subinstructions
- dispatching subinstructions onto the buses
- generating and dispatching immediate integers specified in program code

Figure 3 shows a functional view of the network controller. The Network Controller retrieves a long TTA instruction word from the program memory. Then, the long instruction word is divided into subinstructions for each bus. We recall from earlier that each subinstruction consists of a source address (SRC, 8 bits), a destination address (DST, 8 bits) and a guard expression (GID, 4 bits). If the guard expression is all zeros, a guard expression is not specified. If the guard expression has a non-zero value, the programmer has specified a conditional data move. In this case, the guard expression is compared to the values of the guard signals from the functional units (see Figure 3). If the guard expression is satisfied by the guard signals, or if there is no guard expression, the execution of the subinstruction is allowed. At this point, the SRC and DST values are written onto the SRC and DST address buses.

We recall from the earlier discussion regarding the TACO instruction word that TACO processors support eight-bit immediate integers. The values are provided for the processors by replacing an SRC address in the instruction

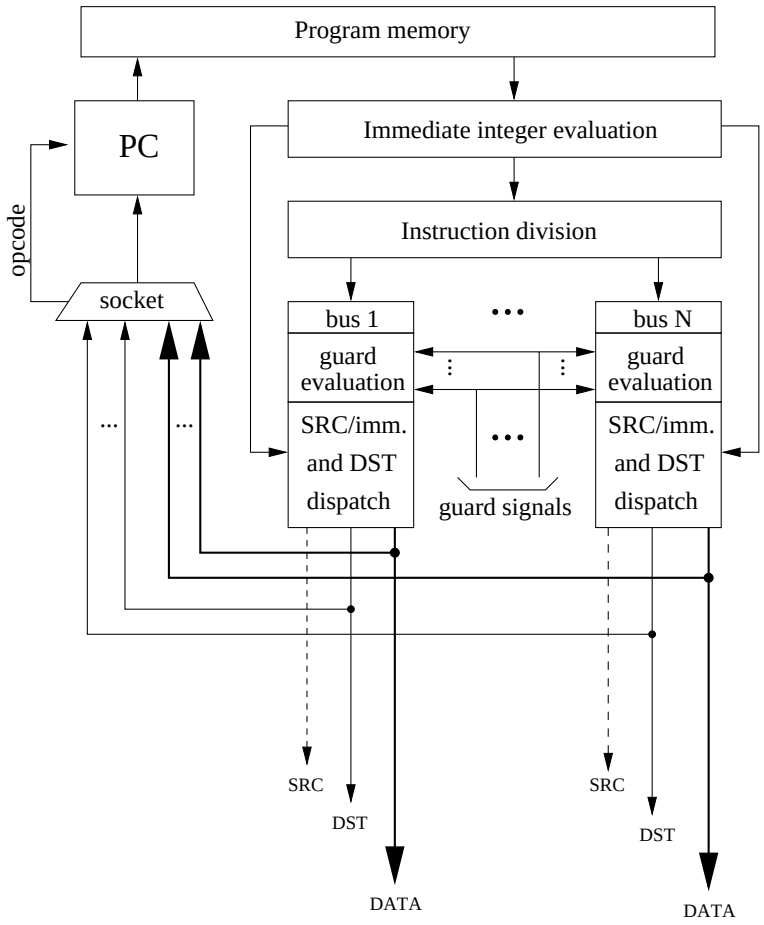


Figure 3: Functional view of the interconnection network controller in a processor with N buses.

word with an integer value, and declaring this change in the immediate control bits of the instruction word. If the Network Controller detects immediate control bits that specify an immediate integer, the SRC value of the specified subinstruction is treated as a data value instead of a socket address. This value is dispatched on the corresponding DATA bus, and a zero is dispatced on the corresponding SRC address bus.

Four stage pipeline Instruction execution in TACO processors is carried out in four pipeline stages. The first stage is instruction fetch (fetch) in which the next instruction is fetched from program memory. The second stage is instruction decode (decode) which has two steps: in the first step source and destination identifiers are put onto the instruction buses, and in

the second step sockets decode the identifiers locally. If there is a match between a hard-coded identifier and an identifier in the instruction bus, the socket stores the result of the decode process to be used on the next cycle (becomes enabled). In the third stage (move stage) FUs with enabled sockets write/store data to/from the buses. The last stage is the execute stage, in which the FU operations are carried out. The pipeline is shown in Figure 4.

Program counter The interconnection network controller is also responsible for maintaining, updating and loading the program counter (PC). For the loading functionality, the network controller has a built-in trigger socket. The PC can be loaded with a new value sent from a functional unit to make jumps in program code possible. The program counter socket has three logical triggers:

- TAPC: Program counter is loaded with the value specified as the trigger data (TR). The resulting action is an absolute jump to the specified program code line (PC = TR).
- TUPC: Program counter is incremented by the value specified as the trigger data (TR). The resulting action is a relative PC increment (PC = PC + TR).
- TDPC: Program counter is decremented by the value specified as the trigger data (TR). The resulting action is a relative PC decrement (PC = PC - TR).

The jumps also require some pipeline management. Figure 4 shows how the pipeline is emptied when programmed jumps occur. When the network controller detects that one of the subinstructions in the long TACO instruction

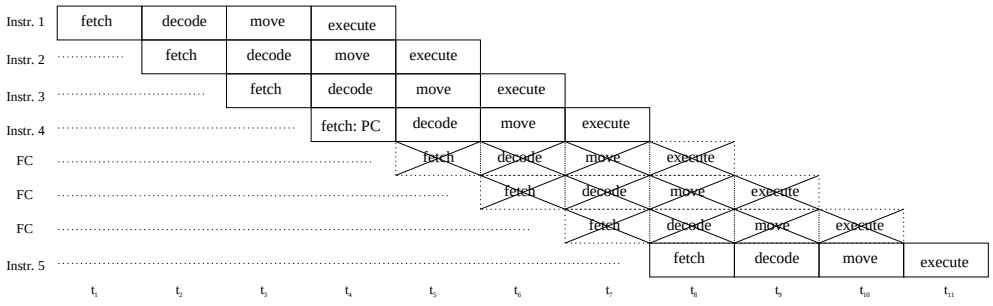


Figure 4: TACO protocol processor pipeline and its operation during programmed jumps. A programmed jump is detected in instruction 4 (labeled *fetch: PC*). *FC* = fetch cancelled.

word is a program counter load, it does not allow further instruction fetches for three machine cycles. This delay is needed for the already pipelined subinstructions (N active subinstructions when there are N buses in the interconnection network) to finish. The program counter load is performed in the execute stage, since program counter loads from the functional units are possible.

3.4 Sockets

Functional units are connected to the interconnection network through input, trigger and output sockets. In TACO processors the input and output sockets have one hardcoded logical identifiers (addresses) and the trigger sockets at least one. Multiple identifiers are used to specify opcodes for functional units that are able to perform more than one operation on the input data (e.g. boolean evaluation unit operations like "=", "$\geq$", "$\leq$", ...). Multiple logical identifiers belonging to a particular trigger socket have a consecutive integer identifier, so opcode extraction is done by subtracting the value of the first hard-coded logical identifier from the identifier read from the DST bus. This opcode is dispatched as a four-bit signal (integer value 0..15) to the functional unit, and the functional unit performs the operation corresponding to the opcode. Naturally, only one hard-coded identifier per socket can be addressed during one cycle.

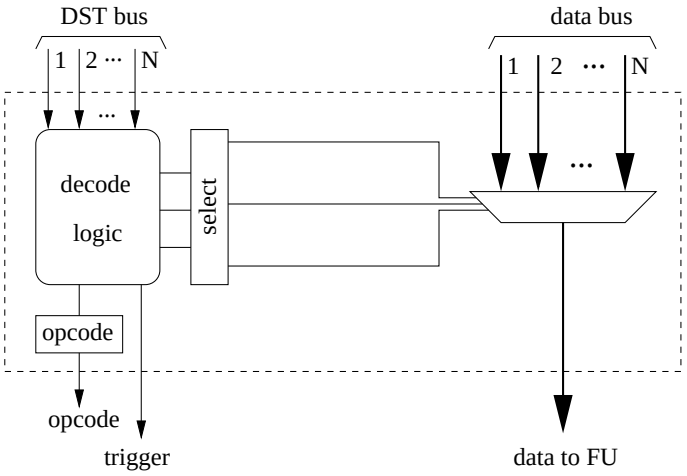


Figure 5: Implementation of a TACO trigger socket. The implementation of an Input socket lacks the opcode and trigger characteristics, but is otherwise exactly the same.

3.4.1 Input and Trigger Sockets

Input sockets do not include any logic for situations in which the same socket is addressed from multiple sources. The programmer and the program code compiler are trusted to prevent such situations. Thus only one bus can be connected to an input socket at a time.

Input sockets decode destination addresses from the DST buses. If a DST address on one of the buses matches one of the hard-coded logical identifiers, the corresponding bus ID is stored in a select register (see Figure 5). If there is no match, the value zero is stored. On the next machine cycle, if there is a non-zero value in the select register, a connection between the selected data bus and the receiving register in the functional unit is opened.

Trigger sockets (Figure 5) function like regular input sockets except for two additional pieces of functionality:

- A trigger socket always signals its host FU to start executing its operation when data is written through the socket into the corresponding FU register. This is implemented using a one bit trigger signal.
- A trigger socket always extracts an opcode from the DST IDs. The extracted opcode is passed to the FU when the FU is triggered.

3.4.2 Output Sockets

Output sockets decode source addresses (SRC) just as the input and trigger sockets decode DST addresses. A data connection is opened between the corresponding FU register and ALL the data buses for which the decode process found a match.

The output socket implementation is very similar to that of the input socket. The differences are that the direction of data flow is opposite, and an output socket can open a connection between the FU register and multiple data buses.

3.5 Functional Units

All TACO FUs are designed to perform a particular protocol processing task in one machine cycle. Future analyses of applications and protocols may reveal tasks that are too complex for this kind of execution. There is no restriction in designing functional units that execute their operations for longer than one cycle. It is up to the programmer and the program code compiler to schedule the code in a way that there are no access conflicts when such FUs are used.

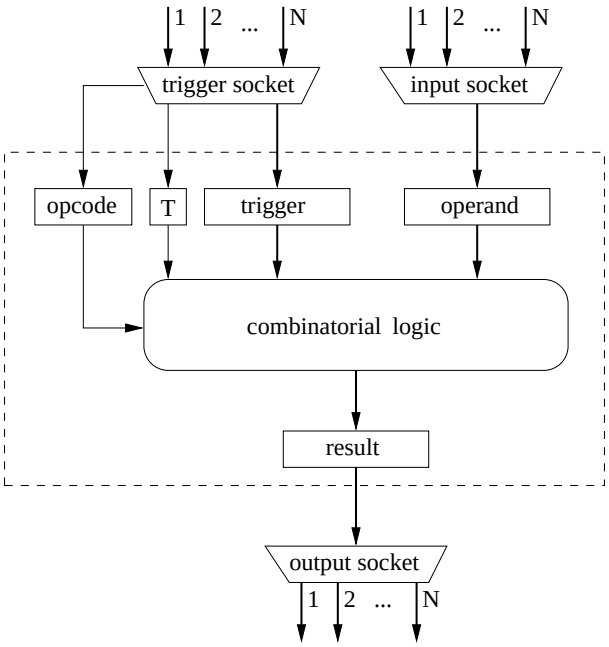


Figure 6: General structure of the functional units. Note that there can be (and often is) more than one operand inputs and result outputs. "trigger", "operand" and "result" are FU data registers, T is the one-bit trigger signal.

Since the current FUs provide their results in one clock cycle there are no pipeline structures inside the FUs. Such pipelines can be added to the FUs in the future, if necessary. There is no limit for the number of FUs of the same kind in a processor. If the application to be implemented requires the same operation frequently, improved performance can be achieved through FU parallelism: having two or more of the same kind of FUs in a processor. Our earlier experiments have shown that there are clear performance gains when an architecture with just one FU of each kind needed to perform the target application is compared with an architecture with two FUs of each needed kind [12, 25].

Some functional units have a guard signal (result bit signal) connected directly to the network controller. These bits are used when the network controller is evaluating logical conditions for conditional execution. Guard bits and guard signals were discussed earlier in sections 3.2 and 3.3.

Figure 6 shows the general structure of all FUs. For simplicity, there is only one input operand register and one output result register (and corresponding sockets) pictured. Many FUs actually have several operand inputs and result outputs. However, there is always only one physical trigger reg-

ister in an FU. The FU operation resides in the combinatorial logic part of Figure 6.

3.6 Memory

For TACO processors, we have chosen to support SRAM as the internal cache memory type. SRAMs provide excellent performance with some cost on power consumption. In most TACO designs the target clock speed is below the memory access speed of a modern SRAM cache memory block. Thus, one memory access per clock cycle can be executed.

Choosing SRAM for the memory type also makes it possible to use a third party processor for fast memory access and table lookup. One such processor is the iFlow address processor [16], designed to act as a co-processor for speeding up internet routing table look-ups. The host network processor sees the iFlow processor as standard SRAM, and reads from and writes to the iFlow processor using standard SRAM mechanisms.

On-chip memory is most often produced into a layout at the time of manufacturing the chip. The memory manufacturer provides information of the necessary signals for using the memory block, and a simulation model of the memory. The designer can choose the word sizes etc., but is not able to modify the actual memory implementation. Since the detailed memory interface is not known until the memory/chip manufacturer has been chosen, we can not design a memory interface unit that would be compatible with any on-chip memory IP block.

However, since the SRAM memory interface is quite simple, not much design effort is needed to connect the memory block onto a TACO processor. Figure 7 shows the connections of a typical SRAM block. It is to be noted that the number of address signals, data signals and control signals varies

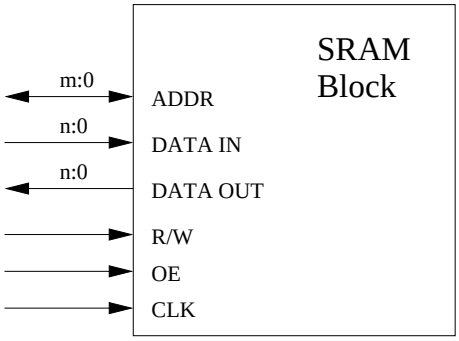


Figure 7: SRAM cache memory block, example of connection complexity.

from one manufacturer to another. Therefore, Figure 7 should be treated only as an example for estimating the complexity of the connections needed to connect an SRAM block to a TACO processor. Also, some applications may require dual-port SRAM, in which case a second set of address, data and control lines is required.

The wiring needed to connect an off-chip memory module into TACO processors is similar to that shown in Figure 7, but again depends on the type of memory and the overall memory configuration in the system.

3.7 I/O Structures

There are two kinds of I/O communication in TACO processors:

1. Reading data from and writing data to the network,
2. Communicating with a host processor (if there is one).

The mechanisms for these tasks can be designed individually to suit the needs and the functioning environment of a certain protocol processing application, or a generic (standard) solution can be used. Application-specific solutions usually provide better performance at the cost of interconnectibility. In the following sections we will discuss some possible solutions for both I/O tasks.

3.7.1 Connection to Network

The network interface of any device is defined by the type of the physical network medium. In copper-wired networks it is usually necessary to enhance, filter and convert the incoming signal before it can be interpreted as digital data words. In optical networks this task is simpler, since the incoming signal is already digital - it only needs to be converted from optical to electrical form. In any case, it is only after these conversions that the actual protocol data is ready for analysis. Figure 8 shows the tasks that need to be carried out in copper-wired and optical networks before the received data is ready for processing.

Standard Solutions For TACO processors, the generic standard solution for network I/O is to connect the I/O module FU (shown in Figure 1) directly to a send/receive buffer (*receive buffer* of Figure 8). The send/receive buffer may be e.g. a buffer on an ethernet chip where deframed data coming in from the network is stored.

The I/O module FU is connected to the interconnection network like any other FU. Thus the protocol data is easily accessed by means of standard

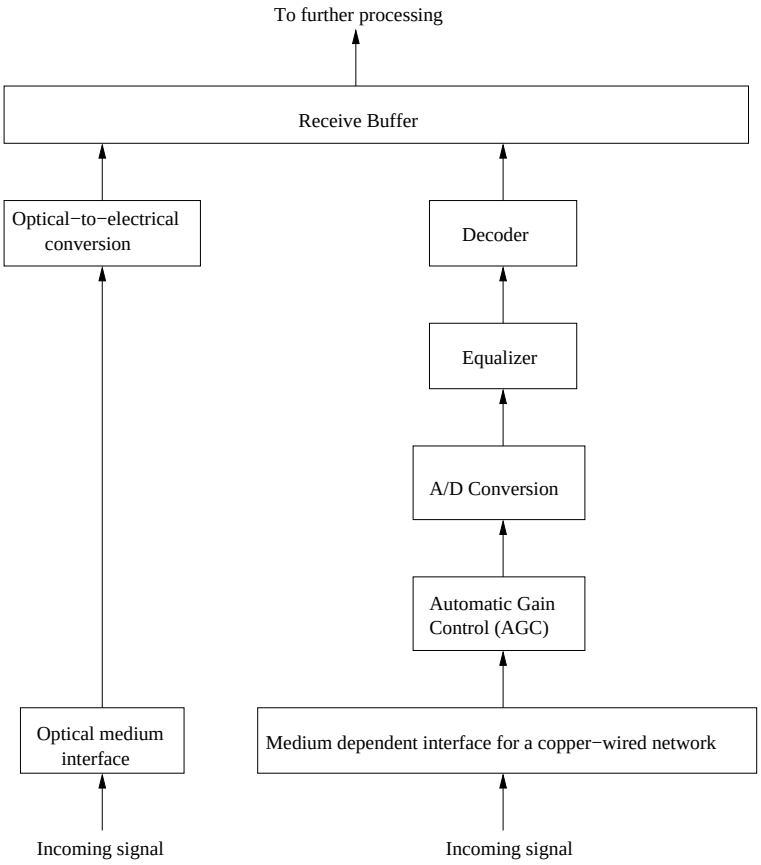


Figure 8: Block diagram of tasks in preparing a signal from the network for data processing.

TACO programming conventions. Since the tasks needed to be performed on signals from a copper-wired network vary from one type of physical medium to another and one type of protocol to another, placing the entire signal processing into one FU would require a separate FU for each kind of physical medium and communications protocol. For this reason, the standard I/O FU only accesses data from the physical/data link receive buffer and does not manage the actual physical communication. Of course an FU with all the necessary signal processing could also be constructed to avoid the additional circuitry required by an off-the-shelf standard interface.

For sending data to the network, the structure in Figure 8 is reversed.

Custom solutions Our first protocol processor implementation (The TACO ATM processor, see [25]) utilized a custom solution for cell I/O: the incoming

data cells are pre-processed by an ATM specific pre-processing unit, which synchronizes with the incoming data stream, verifies the header checksums of the cells and writes the cells into the internal cache. The cell header memory addresses are written into a FIFO FU.

Also the TACO IPv6 router processor described later in this report uses a custom solution for IPv6 datagram I/O. The functional principle is to use dual port memory for queuing incoming datagrams for processing, and to forward the datagrams to the next router/host from the same memory. The memory space taken up by the datagram is released as soon as the datagram has been processed (forwarded or discarded).

3.7.2 Connection to a Host Processor

A TACO processor can operate in a system in one of three alternative ways:

1. As a stand-alone processor,
2. As a stand-alone co-processor,
3. As a co-processor core in an SoC device.

For the latter two, a connection and communication mechanism of some sort is needed. As a stand-alone co-processor we face another point of decision: whether the TACO processor should support the co-processor interface of a specific family of host processors, or if it should provide a generic interface to basically any kind of host processors.

If the decision is to use a TACO protocol processor as a stand-alone co-processor for a specific host family, the interconnection can be designed in a more optimal way to support only the needed communication between the two processor families. This type of connection is used in e.g. early Intel x86 CPUs and their x87 math co-processors, and Texas Instruments DSPs and TI MSP430 series microcontrollers. A good choice for this kind of connection might be something similar to what is used in the TI processors - their HPI (Host Processor Interface) communication resembles the fast path - slow path approach needed in protocol processing (fast path: DSP calculations, slow path: system control by the microcontroller). Although this kind of an approach would be advantageous in terms of performance, by using such an interface the TACO processors would no longer be able to function as generic co-processors.

For a generic interface to a multitude of host processors an industry standard interface is needed. Again, the interface solutions are different for stand-alone co-processors and for SoC cores. For stand-alone processors, a generic

external interface is needed, whereas for SoCs, the structures inside the chip connecting the IP blocks depend on the designer.

PCI bus For a stand-alone co-processor, an industry-standard approach would be to implement the PCI (Peripheral Component Interconnect) bus. PCI is widely supported by most modern general purpose controllers and processors as well as special purpose processors like IXP1200, Motorola PowerQUICC and TI DSPs. PCI support requires a special FU into TACO protocol processors that converts the data from the processor (more precisely, from the Interface FU) into a format suitable for PCI, and that would manage the PCI communication independently.

OCP and AMBA In an SoC, one of the most important features of an IP core (like a TACO processor core) is reusability. A reusable IP core must remain unmodified as it is transferred from one SoC configuration to another.

A recent solution for these requirements is the freely available Open Core Protocol, OCP [21]. It defines a bus-independent communication interface between IP cores and other on-chip components.

Using OCP on a core only requires the SoC integrator to build a bus bridge between the bus and the IP core, which is far simpler a task than rebuilding the IP block every time it is transferred to another SoC configuration. According to [21], all on-chip cores using OCP can be easily reached by any bus structure through simple bridge structures, and the design work needed for building an OCP wrapper for a core is regular enough for automatic interface synthesis. OCP has been tested on, among others, SoCs that run AMBA buses [1] and SoCs that use IBM CoreConnect buses [9].

The AMBA bus is becoming almost a de-facto standard for on-chip SoC buses. It is an open standard for the interconnection and management of SoC IPs. Choosing the bus for an SoC is a task for the SoC integrator. If TACO processors are considered as SoC IPs that support OCP, the on-chip inter-module bus implementation is then not a part of TACO processor design.

4 IPv6

IPv6 (Internet Protocol version 6) [7, 14] is the latest version of the Internet Protocol, introduced to overcome the address restrictions of IPv4 by featuring 128-bit addresses (only 32-bit addresses for IPv4) and improved address hierarchy. The structure of the IPv6 packets has been simplified by introducing an extensible packet format. It contains a simplified fixed size IPv6

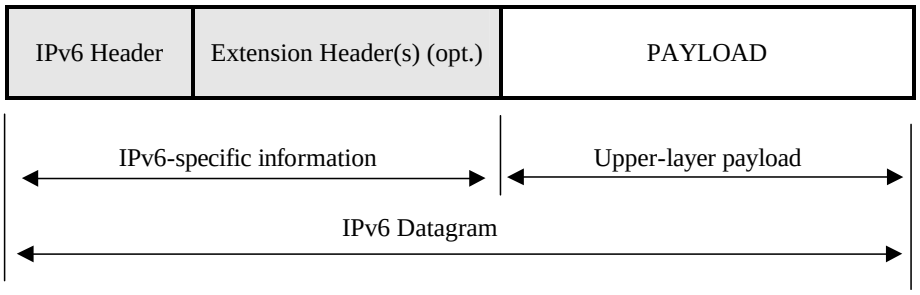


Figure 9: IPv6 Packet Format.

header and a number of optional extension headers that provide improved flexibility and support for options. Moreover, IPv6 offers now extensions to support authentication, data integrity and optional data confidentiality.

An IPv6 packet is basically composed of two parts: (1) the IPv6-specific information (headers) that is processed by the IPv6 layer of the Internet nodes and (2) the carried information (payload) to be used by the upper layer protocols of the nodes. The IPv6-specific information is composed of the IPv6 header and a number (can be zero) of optional extension headers (Figure 9).

The IPv6 header has a fixed size (40 octets) and is composed of a number of fields (Figure 10) as follows:

- Version - 4-bit field specifying IP version (6 for IPv6).

- Traffic Class - 8-bit field, intended for destination hosts or forwarding routers to distinguish among different classes or priorities of IPv6 packets. By default is set to all zero value.

- Flow Label - 20-bit field to request to a host that a packet is handled in a certain manner. If a host does not offer support for this field, its value is set to zero by the originator and ignored by receivers.

- Payload length - 16-bit unsigned integer that specify the length, given in octets, of the entire IPv6 packet except the IPv6 header.

- Next Header - 8-bit field that identifies the header immediately following the IPv6 header. The next header can be either an IPv6 extension header or an upper layer protocol.

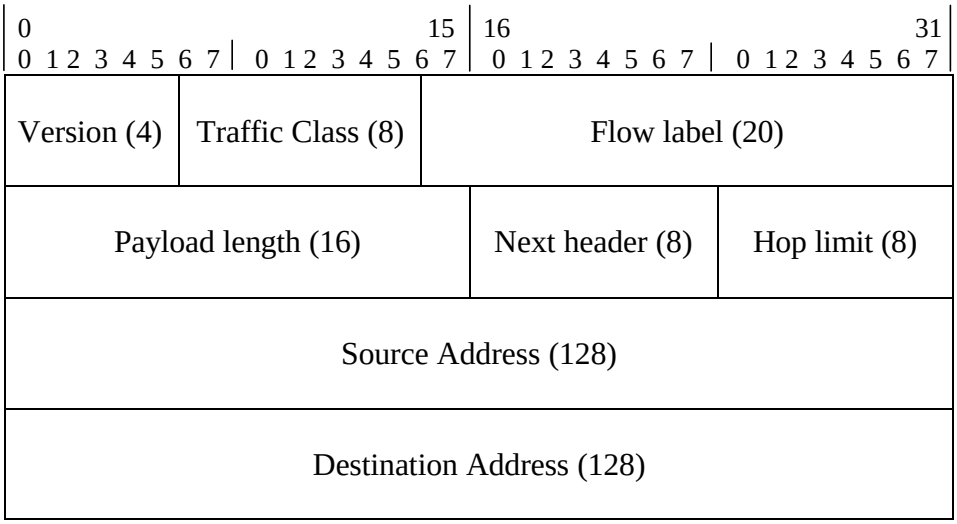


Figure 10: IPv6 Header Format.

- Hop Limit - 8-bit long unsigned integer value that shows the lifetime of the packet. It is set to 255 by the originating host and decremented by 1 by each host (router) on the way to destination. When value 0 is reached the packet is considered to be expired and a corresponding error message is returned to the originator.
- Source Address - 128-bit field that identifies the originator of the packet.
- Destination Address - 128-bit field that identifies the destination of the packet.

While IPv4 addresses are divided into network classes (class A, class B, etc), IPv6 addressing and routing is performed by using variable-length prefixes from the address. Hosts can legitimately treat IPv6 addresses as opaque 128-bit addresses, while routers need only store prefixes (ranging from 1 to 128 bits). The addresses are expressed in text as hexadecimal values, while the prefix lengths are expressed as a decimal value that specifies the leftmost bits of the address comprising the prefix.

IPv6 address structure provides three different types of addresses:

- Unicast - an identifier for a single interface of an Internet node. A packet sent to a unicast address is delivered to the interface identified

by that address only. A number of forms for the unicast addresses have been defined in IPv6, each providing a different level of hierarchy:

  - special (reserved) addresses: Unspecified Address (::0), Loopback Address (::1), testing addresses, etc
  - Aggregatable Gobal Unicast Address - global scope
  - Local Use Addresses (Link-Local and Site-Local addresses) - local scope

- Anycast - an identifier for a set of interfaces (usually belonging to different nodes). A packet sent to an anycast address is delivered to one of the interfaces identified by that address (the nearest one according to the routing protocol's measure of distance).
- Multicast - an identifier for a set of interfaces (usually belonging to different nodes). A packet sent to a multicast address is delivered to all interfaces identified by that address.

There are no broadcast addresses in IPv6, their function being superseded by multicast addresses.

The Extension Headers of an IPv6 packet contain information (options) to be processed either by the final destinations or by hosts (routers) on the way. The extension headers are processed in the order they are present.

- Hop-By-Hop options header - used to specify the delivery parameters at each hop on the path to destination. For optimization purposes it has to be placed first after the IPv6 header. This header is either used for padding or for specifying payload sizes greater than 65,535 octets.
- Destination Options header - used to specify packet delivery options either for intermediate destinations (when the Routing Header is present) or for the final destination.
- Routing header - specifies a list of intermediate destinations for the packet to travel on its path to the final destination.
- Fragment Header - used for IPv6 fragmentation and reassembly service. In IPv6 protocol, only the source node can fragment payloads and the reassembly process is done only at the destination. This extension header is not processed by the nodes/routers that the packet passes through on its way to the final destination.

- Authentication header - provides data authentication services (identity of the node that sent the packet), data integrity (data was not changed on the way) and anti-replay protection (captured packet cannot be retransmitted) for the entire IPv6 packet.

- Encapsulating Security Payload header - provides data confidentiality, data authentication and data integrity services for the payload of the packet.

4.1 ICMPv6

As in IPv4, IPv6 does not provide any means for reporting errors. Instead, IPv6 uses an updated version of the Internet Control Message Protocol (ICMP) called ICMP version 6 [5]. Basically, ICMPv6 provides functions for reporting errors and dealing with informational messages (echo service) for troubleshooting. In addition, ICMPv6 provides support for other protocols like:

- Multicast Listener Discovery (MLD) - replaces the Internet Group Management Protocol (IGMP) for IPv4 for managing subnet multicast membership.

- Neighbor Discovery (ND) - manages node-to-node communication on a link, by replacing the Address Resolution Protocol (ARP), ICMPv4 Router Discovery and the ICMPv4 Redirect message of IPv4.

Although the ICMPv6 is now included in the IPv6 protocol, it still works as a stand-alone subprotocol. There are two main types of ICMPv6 messages:

- Error Messages - used to report errors in delivery of IPv6 packets by either the destination or the intermediate nodes (routers) on the way.

- Informational Messages - provide diagnostic functions and also support for MLD and ND protocols.

The ICMPv6 packet structure (Figure 11) is composed of an IPv6 Header, Extension Headers and the ICMPv6 message. The ICMPv6 message is composed of an ICMPv6 header and ICMPv6 Data. The ICMPv6 header (Figure 12) consists of 3 fields:

- Type - indicates the type of the ICMPv6 message. Its value determines the format of the remaining data.

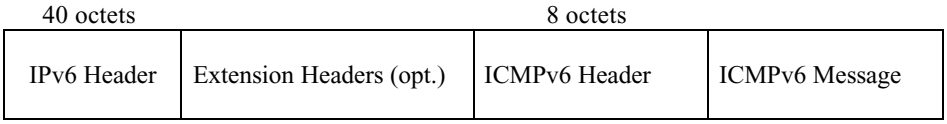


Figure 11: ICMPv6 Packet Format.

Figure 12: ICMPv6 Header Format.

- Code - depends on the message type and is used to create an additional level of granularity.
- Checksum - used to detect data corruption of the ICMPv6 message and parts of the IPv6 header. All ICMPv6 messages bear a Checksum field to verify data integrity. Checksum is computed as "*16-bit one's complement of the one's complement sum of the entire ICMPv6 message starting with the ICMPv6 Type field, prepended with the pseudoheader of the IPv6 packet*" [5]. For computing the checksum, the Checksum field is set to 0.

The pseudoheader (see Figure 13) of an IPv6 packet is comprised of the Source Address, Destination Address, the upper-layer payload length (expressed in octets) and an 1-octet field representing the Next Header identifier

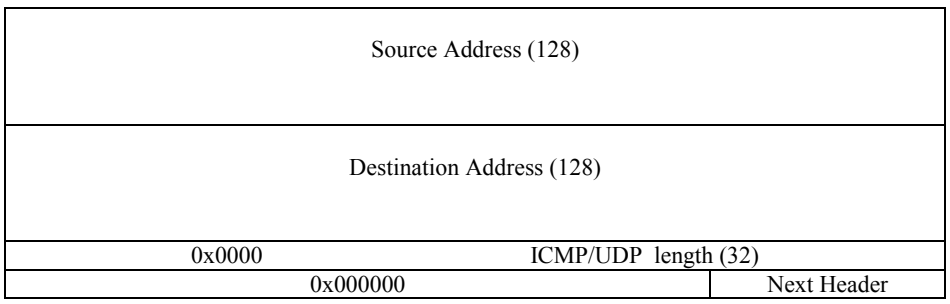


Figure 13: Upper-layer pseudoheader of an IPv6 packet.

of the upper-layer protocol (58 for ICMPv6).

Depending of the ICMPv6 message type, the structure of the ICMPv6 Data may vary.

4.1.1 ICMPv6 informational messages

The ICMPv6 informational messages are of two types: Echo Request (type 128) and Echo Reply (type 129). Both have the same ICMPv6 Data structure, where the ICMPv6 header is followed by the following fields:

- Identifier - 16-bit field set by the sender
- Sequence Number - 16-bit field set by the sender
- Data - zero or more data octets set by the sender

The Echo Request message is sent to a destination to solicit an immediate Echo Response message. Upon receiving an Echo Request message, a node creates an Echo Response message by copying the initial message and changing the Type field from 128 to 129. The Identifier, Sequence and Data fields are left unchanged. The checksum field is recalculated and the packet sent back.

4.1.2 ICMPv6 Error Messages

The ICMPv6 Error Messages are used to support forwarding or delivery errors by either hosts or routers. There are 4 types of ICMPv6 error messages with the following structure:

- Destination Unreachable (Figure 14) - sent either by a router or a destination host when a packet cannot be forwarded to its destination.
  - Type = 1
  - Code:
    - 0 - no route found in the routing table
    - 1 - route prohibited
    - 2 - address beyond the scope of the source address
    - 3 - destination address unreachable (not able to resolve link layer address)
    - 4 - destination port unreachable
  - Checksum

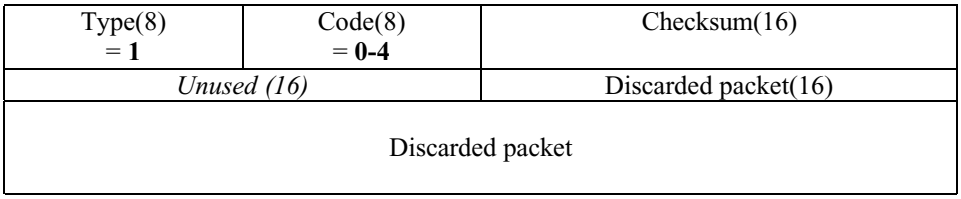


Figure 14: ICMPv6 Destination Unreachable message structure.

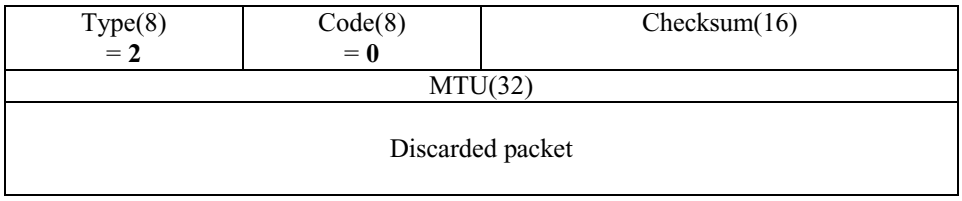


Figure 15: ICMPv6 Packet Too Large message structure.

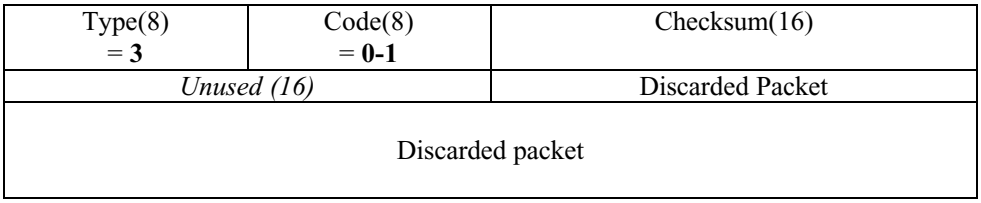


Figure 16: ICMPv6 Time Exceeded message structure.

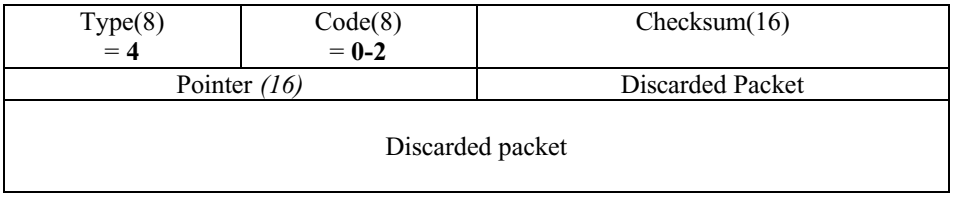


Figure 17: ICMPv6 Parameter Problem message structure.

  - Unused - 4-octet field, set to 0 by sender
  - portion of discarded packet

- Packet Too Large (Figure 15) - sent by routers when a packet cannot be forwarded because the link MTU on the forwarding link is smaller than the size of the IPv6 packet.
  - Type = 2
  - Code - set to 0 by the sender, ignored by the receiver
  - Checksum
  - MTU - 4-octet field containing the value of the MTU of the given interface of the router
  - portion of discarded packet

- Time exceeded (Figure 16) - sent by a router when the Hop Limit field in the IPv6 header of a packet becomes 0.
  - Type = 3
  - Code:
    * 0 - Hope Limit is 0
    * 1 - Reassembly Time Exceeded (only in destination node)
  - Checksum
  - unused - 4-octets set to zero
  - portion of discarded packet

- Parameter Problem (Figure 17) - sent either by a router or by a destination when an error is encountered either in the IPv6 header or in an extension header of an IPv6 packet, preventing the packet from being processed.
  - Type = 4
  - Code:
    * 0 - error within the IPv6 or Extension Headers
    * 1 - unrecognized Next Header Value
    * 2 - unrecognized next header option
  - Checksum
  - Pointer - 16-bit field whose value points to the offset of the field that caused the error in the initial packet

– portion of discarded packet

A set of rules for generating and responding to ICMPv6 error and informational messages is specified in [5]. When an error message is to be generated by the ICMPv6 protocol, it will be addressed to its source address and will be originated from the unicast address of the receiving interface. If the packet has a routing header, the ICMP message is sent back to the source without passing through the same route as in the routing header. All ICMPv6 packets should not exceed the minimum MTU for IPv6 (1280 octets), in order to ensure their deliverability over any IPv6 network.

4.2 IPv6 routing

An IPv6 router is a network device that deals with transferring data (IPv6 packets) from one network to another in order to reach its final destination. Two main functionalities have to be supported by a router: forwarding and routing. Forwarding is the process of determining on what interface of the router a packet has to be sent towards its destination. Routing is the process of building and maintaining a table (routing table) that contains information about the topology of the network. The router builds up the routing table by exchanging information with other routers in the network.

There are different protocols that specify the way forwarding and routing processes work. Routing protocols can be classified in different classes based on the algorithms they use (link-state or distance vector algorithms) and on their routing domain scope (interior or exterior gateway protocols).

The first classification refers to the way the protocols build and manage the topological information in their routing table. In Distance Vector Algorithm-based protocols, each router maintains lists of best-known distances to all other known routers. These lists are called *vectors*. Each router is assumed to know the exact distance (in delay, hop count, etc.) to other routers directly connected to it. Periodically, distance vectors are exchanged between adjacent routers, and each router updates its vectors. In Link State-based protocols, each router measures the distance (in delay, hop count, etc.) between itself and its adjacent routers. The router builds a packet containing all these distances. The packet also contains a sequence number and an age field. Each router distributes these packets using flooding (every incoming packet is sent out on every outgoing interface except the one it arrived on). To control flooding, the sequence numbers are used by routers to discard flood packets they have already received from a given router. The age field in the packet is an expiration date. It specifies how long the information in the packet is good for. Once a router receives all the link state packets

from the network, it can reconstruct the complete topology and compute a shortest path between itself and any other node.

The second classification is done based on the protocol the router uses to accomplish its functionality. We call a routing domain (autonomous system) a network administered by a single entity. Based on how routers are integrated with the routing domains, routing protocols fall into two categories: those that route information inside a single autonomous system - Interior Gateway protocols, and those that route information between different autonomous systems - Exterior Gateway protocols.

A number of protocols have been ported from IPv4 to IPv6. They use the same "longest-prefix match" approach as in IPv4. Out of these we can mention as Interior Gateway Protocols:

- Open Shortest Path First Protocol version 3 (OSFPv3) [4]
- Routing Information Protocol next generation (RIPng) ([13])
- Intermediate System to Intermediate System Intra-Domain for IPv6 (I/IS-IS) [8]

and as Exterior Gateway Protocols:

- multiprotocol extensions of the Border Gateway Protocol for IPv6 (BGP4) [20].

4.2.1 Routing Information Protocol next generation

We chose for our implementation the Routing Information Protocol next generation (RIPng) [13]. RIPng is intended to allow routers to exchange information for computing routes through an IPv6-based network. It is based on a distance vector protocol and is supposed to be implemented only in routers. Any router that uses RIPng is assumed to have interfaces to one or more networks. These are referred to as its directly-connected networks.

RIPng is an interior gateway protocol addressed to small networks. The protocol relies on access to certain information about each of these networks, the most important of which is the metric. RIPng metric of a network is an integer between 1 and 15, inclusively. Implementations should allow the system administrator to set the metric of each network. In addition to the metric, each network will have an IPv6 destination address prefix and prefix length associated with it. These are also to be set by the system administrator in a manner not specified by this protocol.

Each router that implements RIPng is assumed to have a routing table. This table has one entry for every destination network that is reachable

throughout the router operating RIPng. Each entry contains at least the following information:

- The IPv6 prefix of the destination network
- A metric, which represents the total cost of getting a packet from the router to that destination. This metric is the sum of the costs associated with the networks that would be traversed to get to the destination.
- The IPv6 address of the next router along the path to the destination (i.e., the next hop). If the destination is on one of the directly-connected networks, this item is not needed.
- A flag to indicate that information about the route has changed recently. This will be referred to as the "changed route flag".
- Various timers associated with the route.

The RIPng routing protocol is based on the User Datagram Protocol (UDP) [18], a connectionless transport protocol. Communication between hosts is done through ports. All communication intended for another router's RIPng process is directed to the RIPng port (521). In theory, the UDP and IPv6 protocol should be completely independent of each other. But in practice, there is not such a clear border in-between them. UDP protocol offers a basic mechanism for data correctness, each packet carrying a checksum field. The checksum is calculated the same way as in ICMPv6 by including the pseudoheader of the IPv6 packet. All incoming packets have to be checked that they are addressed to existing (correct) ports and that the checksum field is valid.

A RIPng packet (Figure 18) is composed of an IPv6 header, zero or many extension headers, a UDP header and the RIPng message. The UDP header is composed of the Source and Destination Ports of the packet, the UDP checksum and the UDP payload length (Figure 19).

The router that implements the RIPng protocol has to build up and maintain a routing table containing information about the topology of the network. This is done by sending REQUEST messages to interrogate other routers in the network in order to find out their topological information. These routers reply with RESPONSE messages containing requested information from their routing table. In addition, a RIPng router periodically informs the other routers in the network about the information it has in the routing table, by sending RESPONSE messages on all connected networks. These messages contain parts or a complete copy of the routing table of

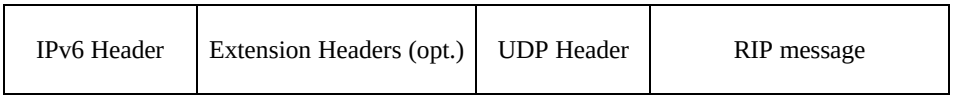


Figure 18: RIPng packet format.

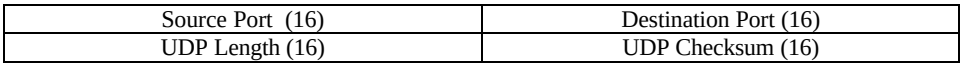


Figure 19: UDP Header Format.

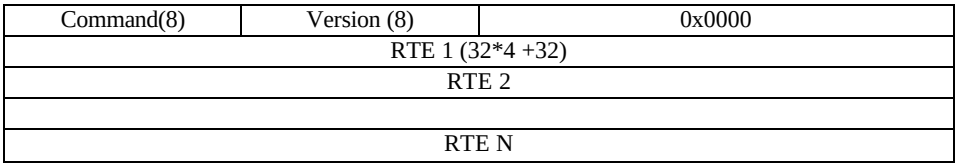


Figure 20: RIPng message format.

the router. The information exchanged by the routers is structured inside the RIPng messages under the form of Routing Table Entries (RTEs). The structure of a RIPng message is presented in Figure 20. It consists of a Command field specifying if the message is a REQUEST or a RESPONSE, a Version field that contains the version of the protocol used (1 for RIPng), a 2-octet field set to zero by the sender, and a number of Routing Table Entries (RTEs).

Each RTE in the RIPng message has similar structure and size containing information about existing routes in the Routing Table of the router. There are two types of RTEs:

- regular RTE (Figure 21) - includes the IPv6 Prefix (128-bit) of the route, a Route Tag (16-bit) to separate internal from external routes, the Prefix Length (8-bit) to specify the number of significant bits in the IPv6 Prefix, and the Metric (8-bit) (to define the current metric to the destination).

- Next Hop RTE (Figure 22) - provides RIPng with the ability to specify the intermediate next hop IPv6 address for packets. The Prefix field

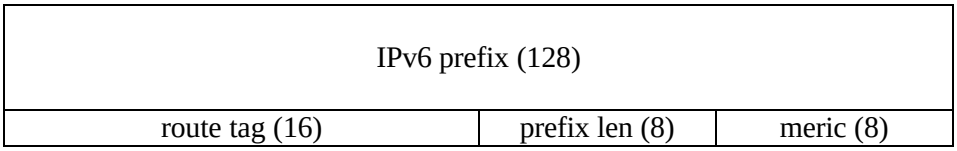


Figure 21: Regular RTE.

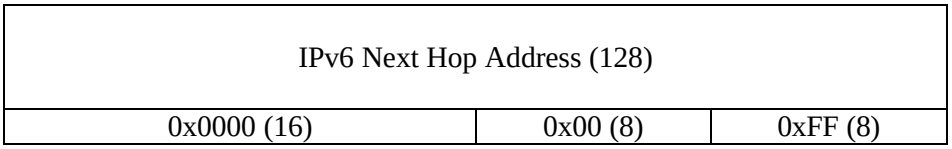


Figure 22: Next Hop RTE.

specifies the IPv6 address of the next hop, the Route Tag and Prefix Length are set to zero on transmission and ignored on reception.

4.3 Router Specifications and Requirements

Our goal was to create a *10 Gbps IPv6 Router over Ethernet using the Routing Information Protocol Next Generation*. The Ethernet is a link-layer protocol described in IEEE standard 802.3 [10].

Our router is configured to handle up to 4 interfaces, each interface having assigned a link-local unicast address (referred to as *link-local address*), an aggregatable global unicast address (referred to as *unicast address*) derived from the link-local address, a cost (metric) of sending the packets on the interface and an associated Maximum Transmission Unit (MTU) for each directly-connected network. Since the router is intended to work over Ethernet networks, according to the recommendations in RFC 2641 the maximum IPv6 datagram size that can be carried by Ethernet frames is limited to 1500 octets. In addition, the minimum MTU for IPv6 is 1280 octets.

In the following, the term *datagram* refers to a package of data transmitted over a connectionless network. *Connectionless* means that no data connection has been established between source and destination.

For the sake of simplicity, we assume that IPv6 datagrams may only have a Routing Extension Header, otherwise they are formed of IPv6 header and the upper layer payload. Datagrams that are not addressed to the router's upper layers (ICMP or UDP) have to be processed in order to determine the

next interface on which they should be forwarded. This is done by interrogating the router's routing table. If the datagram carries routing header, then the next hop address should be extracted from the routing header (instead of the routing table) and the datagram forwarded on the appropriate interface. Datagrams that are addressed to upper layer protocols of the router should be carefully checked for validity and then forwarded to the upper layer. The router only provides support for the RIPng protocol. Datagrams addressed to upper layer protocols other than UDP are discarded and an error message is sent to the emitter.

In our specification the main functionality of ICMPv6 is to generate error messages (if needed) as a result of receiving erroneous packets, and also to respond to Echo Request messages. When an error message needs to be generated by the ICMPv6 protocol, it will be addressed using the source address of the original message (even if the datagram contains a Routing Header) and will be originated from the unicast address of the receiving interface. ICMP datagrams may not exceed the minimum MTU for IPv6 (1280 octets) in order to insure their deliverability over all networks. When an Echo message is addressed to the ICMPv6 layer of the router, the checksum field should be checked for correctness and a reply generated by sending the same datagram (with modified *Type* field) to the originator. Any other incoming ICMPv6 messages types addressed to the router are simply discarded (in future implementations they will be either treated or sent to the upper layer). In future specifications ICMPv6 should also provide support for treating received informational messages and error messages, as well as for other ICMP-based protocols like Multicast Listener Discovery (MLD) and Neighbor Discovery (ND) protocols.

5 A TACO Configuration for IPv6 Routing

According to the previous specification, an IPv6 router should be able to receive IPv6 datagrams from connected networks, to check their validity for correct addressing and header fields, to interrogate the routing table for the interface(s) the datagrams should be forwarded on, and to send the datagrams on the appropriate interface(s). Additionally a router should build and maintain a routing table that contains information about network topology. The router builds up the Routing Table by listening for specific datagrams broadcasted by the adjacent routers, in order to find out information about the topology of the network. At regular intervals, the routing table information is broadcasted to the adjacent routers to inform them about changes in topology.

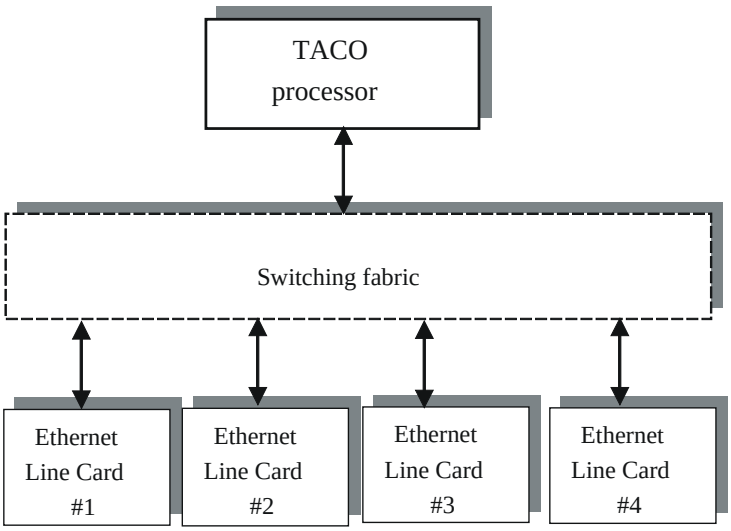


Figure 23: Generic router.

Routers have to handle two types of Internet traffic: (1) the type that updates the routing tables and (2) the type that requires packet forwarding onto adjacent networks. The forwarding process has to search the routing table for a specific network prefix with the longest prefix length possible. Since a routing table can consist of thousands of entries, finding the matching prefix can require long computational time. The current bandwidth demands of internet networks put a high pressure on the routing table look-up speed. To meet these demands, the router implementations need to use fast searching algorithms and dedicated hardware in order to improve the forwarding throughput. Today's routers are mainly composed of three parts: a central processor, a number of network interface cards connected to networks, and a switching fabric.

Our router uses a TACO processor and a number of Ethernet line cards corresponding to each connected network interface of the router. We are only interested in the design and performance of the TACO processor for implementing routing and forwarding tasks. The line cards can be chosen from the available products on the market (Intel IFX18103, Cisco GigE 12000, etc.). The interface between the cards and processor is dependent on the products used. Each network card contains a set of independent input and output buffers that can be read and written by the processor. The line cards deal with implementing the Ethernet protocol and its specific tasks, provide fully assembled decapsulated IPv6 datagrams to the processor, take care of Ethernet fragmentation and encapsulation of outgoing datagrams, and also

resolve ARP/RARP requests.

The TACO processor is used as a stand-alone processor that offers support for the IPv6 and UDP layers. The processor communicates with the line cards through input and output buffers. Each interface of the router has an associated input and output buffer, and they can receive or send datagrams independently. When a new datagram is received on one of the input buffers, it is saved in the main memory and the processor starts processing it. When a datagram needs to be sent, it is taken from the main memory and saved into the output buffer of the corresponding interface.

5.1 TACO architectural configuration

The development flow in TACO consists in 2 parts: (1) identification of the functional unit types needed by an application (qualitative configuration) and (2) deciding the number of resources of each type (quantitative configuration) with respect to performance requirements and physical constraints.

The qualitative configuration has been discussed in detail in [11]. There we start from the requirement specification of the application and go through a number of refinement steps, until the necessary level of detail is reached. The Unified Modelling Language [2] plays a fundamental role in the approach; it is used for the description and formalization of different steps. The analysis is done from a functional point of view in a platform-independent manner. Then, by using domain information on the given application, we select a list of operations, directly mappable onto the TACO architecture. The qualitative configuration is performed by selecting (from the existing TACO resources or by creating new ones) the functional units that implement the operations in the list.

Once the TACO resources have been selected, we perform the qualitative configuration step. In order to reach a good balance between the router's performance and its physical characteristics, we explore different architectural configurations by varying the number of FUs of each required type and the number of buses in the interconnection network. These different configurations are simulated using the TACO SystemC model [26] and the physical characteristics (power consumption and area) are estimated in a Matlab model [15]. In the end we select for hardware synthesis the configuration that is able to perform the target application within given timing, power and area constraints. More details on how we perform the quantitative configuration can be found in [12] and [24].

5.2 IPv6 FUs

From the qualitative configuration process for the IPv6 router (discussed in the previous section), we have identified a number of functional unit types that implement the required functionality of the router. From the UML design flow we obtained FUs that can be grouped into 3 main categories: logic units, data access units and input/output units. The logic units deal with implementing bit-wise computations and data manipulations. Data access units provide an interface for fast access to data that needs to be processed. Input/Output units deal with interfacing the router with the outside environment (in our case with the line cards).

The functional units in TACO processors consist of a number of input and output registers and internal logic. Some functional units may also have direct control signals wired to the network controller. In addition, the functional units that are concerned with external connections have the registers and connectors needed for the external connection in question.

In the following, the specifications of the functional units needed for to construct a TACO IPv6 router processor are given.

5.2.1 Comparator FU

Interface:

- Operand register OP (input operand type)
- Trigger register TR (input trigger type)

  Eight logical trigger addresses
- Result register R (output result type)
- Has a guard signal to network controller
- NO external connections

Operation:

- Opcode 0, TEQ: "="

  If **TR** = **OP**, **R** is all ones and the guard signal is raised
- Opcode 1, TNO: "≠"

  If **TR** ≠ **OP**, **R** is all ones and the guard signal is raised

- Opcode 2, TGZ: "> 0"

  If $\mathbf{TR} > \mathbf{0}$, **R** is all ones and the guard signal is raised

- Opcode 3, TEQZ: "$= 0$"

  If $\mathbf{TR} = \mathbf{0}$, **R** is all ones and the guard signal is raised

- Opcode 4, TLEQ: "$\leq$"

  If $\mathbf{TR} \leq \mathbf{OP}$, **R** is all ones and the guard signal is raised

- Opcode 5, TLT: "$<$"

  If $\mathbf{TR} < \mathbf{OP}$, **R** is all ones and the guard signal is raised

- Opcode 6, TGEQ: "$\geq$"

  If $\mathbf{TR} \geq \mathbf{OP}$, **R** is all ones and the guard signal is raised

- Opcode 7, TGT: "$>$"

  If $\mathbf{TR} > \mathbf{OP}$, **R** is all ones and the guard signal is raised

Opcodes are extracted from logical socket addresses so that the first logical socket address assigned to the trigger socket (hard-coded) is subtracted from the socket address that was used when writing data into the trigger register. Example: socket address space 51..58. Data is written into the trigger socket address 54. The opcode is then 54-51 = 3, which corresponds to the"$= 0$" operation (TEQZ).

Functional description of operation: The value written into the trigger register (**TR**) is compared to the value already stored in operand (**OP**) register. The type of comparison that is carried out depends on the logical socket address used for writing data into the trigger register, as shown in the specification above. If the comparison result is true, an all-ones value is stored into the result register (**R**), and the guard signal is raised. If the result is false, zero is given in the result register, and the guard signal is reset to zero.

Example: A comparator unit has been assigned logical socket addresses 51..58. The value 100 has already been stored into the operand register (**OP**). The value 99 is written into the trigger register (**TR**), using the socket address 57. Thus, the opcode is 57-51 = 6, which corresponds to the"$\geq$" operation (TGEQ). Since the expression "$99 \geq 100$" is false, the guard signal is reset to zero and the value zero is stored into the result register (**R**).

5.2.2 Masker FU

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  only one logical trigger address
- Result register R (output result type)
- No guard signal
- NO external connections

Operation: $\mathbf{R} = (\mathbf{OP} \wedge \mathbf{OD}) \vee (\mathbf{TR} \wedge \neg\mathbf{OP})$, bitwise.

Functional description of operation: Any part(s) of the data word given in the trigger register (**TR**) are replaced with bit sequences defined by a mask (**OP**) and another data word (**OD**).

Example: original data word is 1100 0101 0011 and is given in the trigger register **TR**. The 0101 sequence in the middle is to be changed to 1010. Thus, we define the mask **OP** = 0000 1111 0000, where a zero indicates a bit in the original word that is not to be modified, and a one indicates a bit that should be modified. Then, as the new data we give the data word 0110 1010 0110, where the first and last four bits could be either ones or zeros without effecting the outcome of the operation. Now, according to the function given above, we first calculate $\mathbf{OP} \wedge \mathbf{OD}$ = 0000 1010 0000. Then, we calculate $\mathbf{TR} \wedge \neg\mathbf{OP}$ = 1100 0000 0011. Finally, we do an **OR** between these minterms and obtain **R** = 1100 1010 0011.

5.2.3 Shifter FU

Interface:

- Operand register OP (input operand type)
- Trigger register TR (input trigger type)

  Two logical trigger addresses

- Result register R (output result type)
- Has a guard signal to network controller
- NO external connections

Logical triggers:

- TLR: Logical shift right (opcode 0)
- TLL: Logical shift left (opcode 1)

Operation:

```
if (OP >= 32) R = 0;
else {
  switch(opCode) {
  case 0:         // logic right TLR
    R.range(31-OP,0) = TR.range(31,OP);
    GuardBitSignal = TR(OP - 1);
    for(int idx = 0; idx < OP; idx++){
      R[31-idx]='0';
    }
    break;
  case 1:         // logic left TLL
    R.range(31,OP) = TR.range(31-OP,0);
    GuardBitSignal = TR(31 - OP + 1);
    for(int idx = 0; idx < OP; idx++){
      R[idx]='0';
    }
    break;
  }
}
```

Opcodes are extracted from logical socket addresses as described earlier.

Functional description of operation: The value given in the trigger register (**TR**) is shifted logically left or right (depending on the logical trigger address used) as many positions as defined by the the value given in the operand register (**OP**). The value of the guard signal is equal to the last removed bit: e.g. in a left shift with 5 positions bit 27 is the last removed bit, and in a right shift with 5 positions bit 4 is the last removed bit.

If the value in **OP** is greater than or equal to 32, the result given in **R** will be zero.

Example: original data word is 1100 0101 0011. This value is given as trigger (**TR**). The programmer wishes to perform a logical right shift for 4 positions, so the logical trigger used is TLR (opcode 0). The value 4 is stored into the operand register (**OP**). The result of the logical right shift is then 0000 1100 0101.

5.2.4 Matcher FU

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  only one logical trigger address
- Result register R (output result type)
- Has a guard signal to network controller
- NO external connections

Operation: $\mathbf{R} = (\neg\mathbf{OP} \vee \mathbf{OD} \vee \mathbf{TR}) \wedge (\mathbf{OP} \vee \neg\mathbf{OD} \vee \neg\mathbf{TR})$, bitwise. if **R** is all ones (i.e. maximum integer value), raise guard signal.

Functional description of operation: The operand (**OP**) and data (**OD**) registers specify a bit pattern (range of bits and their values). This pattern is compared to the data word given in the trigger register (**TR**). If the pattern matches the corresponding portion of the data word in the trigger register, the guard signal is raised (i.e. result of operation is true). The mask is specified so that **OP** contains the bit pattern(s) correct aligned (i.e. at their desired positions) that are looked for in the **TR** value, and **OD** contains the negation(s) of the desired bit pattern(s) also correctly aligned. The bits that are not to be matched are indicated as ones in both **OP** and **OD**.

Example: original data word is 1100 0101 0011 and is given in the trigger register **TR**. The 0101 sequence in the middle is the one that is to be matched. Thus, we define the operand **OP** = 1111 0101 1111, and the data value **OD** = 1111 1010 1111. All the bit positions that have the value of one in both **OP** and **OD** will not effect the evaluation.

Now, according to the function given above, we first calculate $\neg\mathbf{OP} \vee \mathbf{OD} \vee \mathbf{TR}$ = 1111 1111 1111. Then we calculate $\mathbf{OP} \vee \neg\mathbf{OD} \vee \neg\mathbf{TR}$ = 1111 1111 1111. Thus, the result of the final **AND** also results in all ones, indicating a true result (so the guard signal should be raised).

If the value stored in **TR** had been 1100 1101 0011, the first maxterm of the match equation would still have been all ones, but the second maxterm would have been 1111 0111 1111. This would have caused the final **AND** to produce a result not equal to all ones, indicating a false result.

5.2.5 IP Checksum FU

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  two logical trigger addresses
- Result register R (output result type)
- NO guard signal
- NO external connections

Operation:

- Opcode 0, TRC: Reset checksum (initialize for new calculation)
- opcode 1, TCC: Calculate checksum
  $\mathbf{OP} = \neg\mathbf{OP}$; $\mathbf{OD} = \neg\mathbf{OD}$; $\mathbf{TR} = \neg\mathbf{TR}$;
  $\mathbf{R'} = \mathbf{R'} + \mathbf{OP}.\mathbf{range}(\mathbf{31},\mathbf{16}) + \mathbf{OP}.\mathbf{range}(\mathbf{15},\mathbf{0}) + \mathbf{OD}.\mathbf{range}(\mathbf{31},\mathbf{16}) + \mathbf{OD}.\mathbf{range}(\mathbf{15},\mathbf{0}) + \mathbf{TR}.\mathbf{range}(\mathbf{31},\mathbf{16}) + \mathbf{TR}.\mathbf{range}(\mathbf{15},\mathbf{0})$;
  $\mathbf{R} = \neg[\mathbf{R'}.\mathbf{range}(\mathbf{15},\mathbf{0})]$;

R' is an internal register used for storing the cumulative one's complement sum. The opcode is extracted from the logical socket addresses as described earlier.

Functional description of operation: Although the IPv6 header does not include a header checksum, the internet checksum used in IPv4 is still needed in IPv6 routing when creating and validating upper layer messages. The internet checksum is calculated using 16-bit one's complement data words as described earlier. Because of the way the internet checksum is calculated, the Checksum FU needs a built-in register for storing a result needed in consequtive calculations. This register is marked as **R'** in the paragraph "Operation" above.

Initially **R** and **R'** are zero. The datagram, for which the checksum is calculated, is fed into the checksum unit as 32-bit words, three words at a time (32-bit words from **OP**, **OD** and **TR**). The checksum unit splits the inputs into six 16-bit words, takes their one's complements, sums them up with the current internal register (**R'**) value, stores the new result in **R'**, takes the one's complement of this value, and places its lowest 16 bits into the result register (**R**).

Example: In the following, for the sake of simplicity of representation, we consider a four-bit internet checksum calculated from three eight-bit inputs. **R'** already contains a value, which needs to be included in the calculation.
$\mathbf{R}' = 1011, \mathbf{OP} = 1010\ 0101, \mathbf{OD} = 1100\ 0011, \mathbf{TR} = 0101\ 0101$
$\mathbf{R}' = 1011 + 0101 + 1010 + 0011 + 1100 + 1010 + 1010 = 11\ 1101$
$\neg\mathbf{R}' = 00\ 0010$, $\neg\mathbf{R}'.\mathbf{range}(\mathbf{3}, \mathbf{0}) = 0010$, which is the result to be placed into the result register **R**. The value in **R'** is needed when the next three long data words are processed.

5.2.6 Counter FU

Interface:

- Trigger register TR (input trigger type)

  three logical trigger addresses

- Result register R (output result type)
- Guard signal
- NO external connections

Operation:

- Opcode 0, TSC: Set Counter

  $\mathbf{R} = \mathbf{TR}$

- Opcode 1, TIC: Increment Counter

  R = **R** + **TR**

  If (**R** == 0) raise guard signal

- Opcode 2, TDC: Decrement Counter

  R = **R** - **TR**

  If (**R** == 0) raise guard signal

The opcode is extracted from the logical socket addresses as described earlier.

Functional description of operation: Before the counter unit can be used, it has to be initialized by writing a start value to the trigger register (**TR**) using the logical trigger **TSC**. Then, whenever necessary, the counter is incremented or decremented using the logical triggers **TIC** and **TDC**. The data value written into **TR** is added to or subtracted from the value currently output as result in the result register (**R**). If the new value is zero, the result bit is raised.

Example: The counter is initialized to the value 10 by writing an immediate integer into the address that corresponds to the logical trigger **TSC**. Then, in the following cycles, the value 1 is written into the address corresponding to the logical trigger **TDC**. After 10 writes (or 10 cycles in this case), the result is zero, and the guard signal is raised.

5.2.7 Router Local Info FU

Interface:

- Operand register OP (input operand type)
- Trigger register TR (input trigger type)

  four logical trigger addresses

- Result register R (output result type)
- NO guard signal
- NO external connections

Operation:

- Opcode 0, TMTU: return max transmission unit (32 bits) for an interface
- Opcode 1, TLLA: return local link address (128 bits) for an interface
- Opcode 2, TUNI: return unicast address (128 bits) for an interface
- Opcode 3, TCST: return cost (32 bits) for sending a datagram on an interface
- Opcode 4, TSMTU: store max transmission unit (32 bits) for an interface
- Opcode 5, TSLLA: store local link address (128 bits) for an interface
- Opcode 6, TSUNI: store unicast address (128 bits) for an interface
- Opcode 7, TSCST: store cost (32 bits) for sending a datagram on an interface

The opcode is extracted from the logical socket addresses as described earlier. The 128-bit words are input and output 32 bits per cycle. The value in the operand register (**OP**) specifies the ordinal number of the 32-bit data word that is to be input or output (3 indicates the MSW and 0 indicates the LSW of the 128-bit value), and the value in the trigger register (**TR**) specifies the interface.

Functional description of operation: The Local Info unit is used for accessing and updating information regarding the local router unit and its interfaces. All the values stored into or read from the Local Info unit are protocol dependent and are used in routing decisions made by the used routing algorithm.

Example: Reading the local link address for interface 3 takes five cycles.

- cycle 1: write 0 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 2: read first 32 bits of the address from **R**, write 1 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 3: read next 32 bits of the address from **R**, write 2 to **OP**, write 3 to **TR** (**TLLA**)

- cycle 4: read next 32 bits of the address from **R**, write 3 to **OP**, write 3 to **TR** (**TLLA**)
- cycle 5: read last 32 bits of the address from **R**

5.2.8 Routing table FU

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  10 logical trigger addresses
- Result register R (output result type)
- NO guard signal
- NO external connections

The opcode is extracted from the logical socket addresses as described earlier.

Operation:

- Opcode 0, TRN: return number (32 bits) of entries (prefixes) in table
- Opcode 1, TRP: return 32-bit part of 128-bit prefix;
  prefix specified by **TR**, part specified by **OP** (3 = MSW, 0 = LSW).

  R = prefix[**TR**].range(127 - 32·**OP**, 96 - 32·**OP**)
- Opcode 2, TRL: return length of prefix (8 bits) specified by **TR**

  R = prefixLength[**TR**]
- Opcode 3, TRI: return interface ID (8 bits) specified by **TR**

  R = interface[**TR**]
- Opcode 4, TRM: return metric (8 bits) for interface specified by **TR**

  R = metric[**TR**]
- Opcode 56, TSRP: store 32-bit part of 128-bit prefix;
  prefix specified by **TR**, part specified by **OP** (3 = MSW, 0 = LSW), value specified by **OD**.

  prefix[**TR**].range(127 - 32·**OP**, 96 - 32·**O)P**) = **OD**

Prefix ID	Prefix	Prefix Length	Interface ID	Metric	Timer	CRF
32 bits	128 bits	8 bits	8 bits	8 bits	8 bits	1 bit

Table 1: Structure of the internal routing table in the Routing Table Unit.

- Opcode 6, TSRI: store interface ID (8 bits) for a prefix;
  prefix specified by **TR**, value specified by **OP**.

  interface[**TR**] = **OP**

- Opcode 7, TRM: store metric (8 bits) for prefix specified by **TR**

  metric[**TR**] = **OP**

The opcode is extracted from the logical socket addresses as described earlier. Table 1 shows the structure of the internal routing table.

In addition to the operations outlined above for logical triggers (opcodes) each routing table entry (RTE) has an associated 8-bit Timer value and a Changed Route Flag (CRF) as in Table 1. When a new RTE is placed into the table, the Timer and CRF are set to zero, and the Timer value starts to be incremented every second. Every time an update of the existing RTE is received, the Metric field is recomputed and the Timer restarted. If a new value for the metric is added, the CRF is set to one to signal that the route of this RTE has to be advertized as changed by the router. If for 120 seconds, an RTE is not updated, the route goes into IDLE mode, where the metric is set to 16 (infinity) and the CRF to one. If for another 180 seconds (value 300 of Timer) no update is still received, a garbage-collection mechanism is started, and the RTE is removed from the routing table). If during the IDLE mode an update is received, the new metric is computed and, if it is less than infinity, the Timer and CRF fields are reinitialized.

Functional description of operation: The Routing Table unit is used for accessing and updating routing table information.

Example: The most significant 32-bit word of a prefix corresponds to the ordinal number 0 and the least significant word to the ordinal number 3. Thus, to store the second 32-bit word of a 128-bit IPv6 address as the next hop address for prefix 5:

- The value 1 is input into the operand register (**OP**). The value 1 corresponds to the second most significant 32-bit word of the address.

- The 32-bit data word is input into the data register (**OD**).
- Finally, the value 5 is input into the trigger register (**TR**) using the **TSRH** logical trigger identifier (opcode 8).

5.2.9 ICMPv6 FU

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)
- One Result register R (output result type)
- NO guard signal
- NO external connections

Operation:
$\mathbf{R}.range\ (31, 24)\ = \mathbf{OP}.range\ (7, 0)$,
$\mathbf{R}.range\ (23, 16)\ =\ \mathbf{OD}.range\ (7, 0)$,
$\mathbf{R}.range\ (15, 0)\ =\ \mathbf{TR}.range\ (15, 0)$

Functional description of operation: This unit is used to construct the first of the two 32-bit data words needed for creating an ICMPv6 header. The second word is directly written into the memory, since it is directly obtainable from the Local Info FU (MTU for target interface) or through an immediate integer (pointer to erroneus field). See section 4.1 for a further description of the ICMPv6 functionality provided by this FU.

R is the first 32-bit word of the ICMPv6 header.

Contents of **OP** (MSB..LSB): Unused (24 b), Type (8 b)
Contents of **OD** (MSB..LSB): Unused (24 b), Code (8 b)
Contents of **TR** (MSB..LSB): Unused (16 b), Checksum (16 b)

Contents of **R** (MSB..LSB): Type (8 b), Code (8 b), checksum (16 b)

Example: An ICMPv6 message "Packet too large" needs to be sent. The value "2" is written into **OP** (type), the value "0" into **OD** (code) and the IPv6 checksum value (we assume 0x5A for the value) from the checksum unit to **TR**. On the next cycle, the value 0x205A is output from **R**.

5.2.10 Memory Management FUs

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)

  two logical trigger addresses
- Result register R (output result type)
- NO guard signal
- No external connections
- dMMU has a DMA interface for input and output FUs; this functionality discussed in section 5.3. uMMU has no DMA interfaces.

The opcode for both MMUs is extracted from the logical socket addresses as described earlier.

Operation:

- Opcode 0, TRMM: read from memory

  Read data word from memory address [**OP**+**TR**] (OP is base address, TR is offset).
- Opcode 1, TWMM: write to memory

  Write data in **OD** to memory address [**OP**+**TR**] (OP is base address, TR is offset).

Functional description of operation: The memory management FUs are used by other FUs to access the memories. The memories the MMUs are connected to are fast enough to provide one memory access per clock cycle, thus an MMU can provide its result in one clock cycle.

The dMMU is used for storing and accessing IPv6 datagrams. The dMMU uses the datagram memory in slots of 391 32-bit data words. Thus, each slot has room for one maximum-length datagram (1500 octets, or 375 32-bit words, maximum datagram for Ethernet) and also an additional IPv6 + ICMPv6 header pair (64 octets, or 16 32-bit words). With this organization, sending erroneus datagrams back to the sender as ICMPv6 messages becomes much easier in terms of memory accessing and organization.

The uMMU is used for storing and retrieving user data. Since the uMMU provides its result in one clock cycle, no general purpose registers are needed for variables and constants. The user memory locations can be initialized at compile time. This makes it possible for the programmer to place constants into the memory at compile time.

Both MMUs provide mechanisms for reading and writing data into/from the memory. In addition to this normal access through the interconnection network, the dMMU also provides DMA access to the memory for the Input and Output FUs. The DMA functionality is described later in this report.

Example: To read data from memory address 150 with the base address set to 128 (i.e. the value 128 is already stored in **OP**), the value 22 is written into **TR** using the logical trigger TRMM (opcode 0).

To write data to memory address 150 with the base address set to 128 (i.e. the value 128 is stored in **OP**), the data value to be stored into the memory location is written into the data register **OD**, and the value 22 is written into **TR** using the logical trigger TWMM (opcode 1).

5.2.11 Input FU

Interface:

- Trigger register TR (input trigger type)
- Three result registers R1, R2, R3 (output result type)
- Guard signal
- External connections; discussed in section 5.3.

Operation: When triggered, write the oldest entry in the

- Memory address FIFO to R1
- Ext interface ID FIFO to R2
- Datagram length FIFO to R3

The data value used in triggering (i.e. data moved to **TR**) has no relevance; the trigger register is used only for triggering the unit.

Functional description of operation: The Input FU acts as a triple read-only FIFO for FUs that access it through the interconnection network. For each incoming datagram it holds the starting memory address, the ID of the interface the datagram came from and the length of the datagram. When the Input FU is triggered, it places the oldest entries in its three FIFOs into corresponding result registers (**R1, R2, R3**).

Example: To get the starting memory address, input interface ID and datagram length for the oldest datagram in the memory, write any non-zero value to the trigger register. The datagram information is given in the result registers **R1, R2** and **R3**.

5.2.12 Output FU

Interface:

- Operand register OP (input operand type)
- Data register OD (input operand type)
- Trigger register TR (input trigger type)
- No result registers
- Guard signal
- External connections; discussed in section 5.3.

Operation: When triggered, add the value in

- **OP** to the Memory address FIFO
- **OD** to the Datagram length FIFO
- **TR** to the Ext interface ID FIFO

Functional description of operation: The Output FU acts as a triple write-only FIFO for FUs that access it through the interconnection network. For each outgoing (i.e. processed) datagram, it holds the starting memory address, the ID of the interface the datagram should be sent to, and the length of the datagram. When the Output FU is triggered, it places the information given in the input registers into its FIFOs.

Example: To store the starting memory address, output interface ID and datagram length for an outgoing datagram, the corresponding data words are written into the three input registers.

5.3 Network Interface

The datagram I/O of the TACO IPv6 router processor is organized as shown in Figure 24. The I/O functionality is performed by three functional units: the dMMU (Datagram Memory Management Unit), the Input FU and the Output FU. In the following we first discuss the I/O operations for datagram input, followed by a discussion of datagram output.

Datagram Input The Input FU is connected to the off-chip data link layer input buffer as shown in Figure 24. Incoming datagrams are queued in the input buffer and moved one by one into the datagram memory. The dMMU is responsible for providing the initial starting memory address for an incoming datagram. The datagram memory is organized into slots consisting of 391 32-bit words. Each slot provides enough space for a possible IPv6 + ICMPv6 header pair and a 1500-octet datagram. The Input FU moves the datagram as 32-bit data words into the memory starting from the provided starting address. The starting memory address, the incoming interface ID and the datagram length are stored into the internal FIFOs of the Input FU.

Datagram processing Processing a datagram residing in the datagram memory is started by reading its information from the Input FU. The header fields are analyzed and manipulated by accessing the datagram memory starting from the address provided by the Input FU. Once the datagram is processed, and ready to be sent, its information (starting memory address, target interface and length) are written into the Output FU.

Datagram Output The Output FU has FIFOs for the memory addresses, outgoing interface IDs and datagram lengths of all outgoing datagrams. The values in these FIFOs are used for sending the datagrams: each outgoing

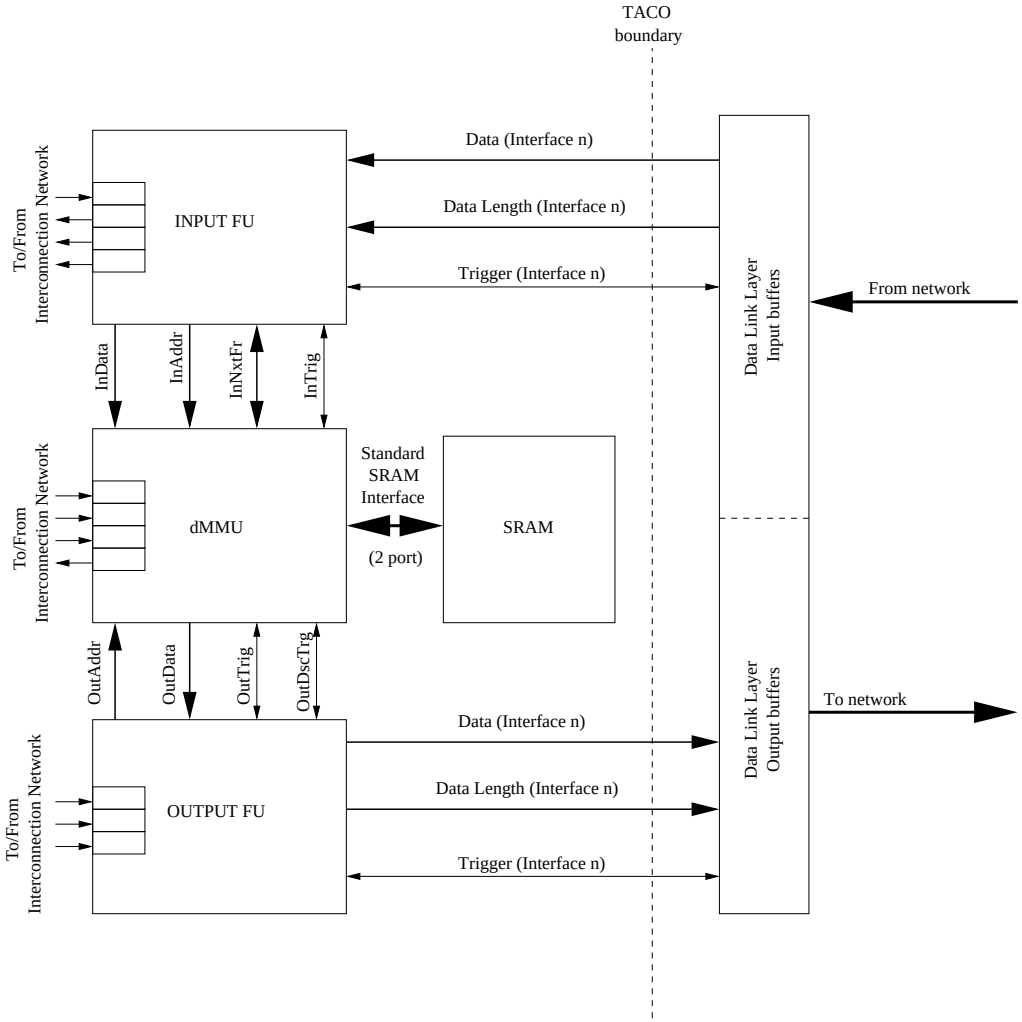


Figure 24: Network interface in the TACO IPv6 router. Thick arrows indicate signals with processor word width, thin lines indicate one-bit signals.

datagram is copied from the datagram memory to the ethernet buffers according to the information in the FIFOs.

About the Datagram Memory In the TACO IPv6 router processor there are three memory access request sources to the dual port datagram memory: the functional units connected to the interconnection network and needing access to datagram contents (read/write), the Input FU (write only) and the Output FU (read only). It is up to the dMMU to act as an arbiter to manage the memory access requests from these three possible sources. However, since the Input FU only writes and the Output memory only reads

the memory, access requests can be served quite efficiently. The requests are not queued, the dMMU simply utilizes a blocking mechanism (raises a busy signal when both ports are busy).

6 Conclusion

In this report we presented the TACO protocol processor platform and its use in application-specific processor design. We discussed a case study, in which we designed an protocol processor for IPv6 routing on the TACO platform. The modularity of the TACO platform (FUs are independent of each other and of the interconnection network) provides good support for design automation. Adding more FUs and/or buses to a given architectural configuration increases the level of execution parallelism, allowing processor performance to scale up accordingly.

The TACO architecture also offers good support for IP reuse, allowing the designer to create new configurations by using FUs already in existence, e.g. FUs that have been designed and implemented for another protocol processing application. This is an important feature for shortening the design cycle of industrial products and in particular of creating and managing product families.

Since the main emphasis of the TACO architecture is moving data, it provides an important platform for protocol processing and potentially many other data-intensive applications. Moreover, by combining the programmability of the platform with the dedicated hardware speed of the FUs, the TACO processor platform provides important benefits in achieving fast processing speeds and easy upgrades for different families of applications.

References

[1] Arm Ltd. web site (search for AMBA). *http://www.arm.com*.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, Reading, MA, USA, 1999.

[3] A. Both, B. Biermann, R. Lerch, Y. Manoli, and K. Sievert. Hardware-software-codesign of application specific microcontrollers with the ASM environment. In *Proceedings of the Conference on European Design Automation*, pages 72–76, Grenoble, France, September 1994.

[4] R. Coltun, D. Ferguson, and J. Moy. OSPF for IPv6. *RFC 2740*, December 1999.

[5] A. Conta and S. Deering. Internet control message protocol (ICMPv6) for internet protocol version 6 (IPv6) specification. *RFC 2463*, December 1998.

[6] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA*. John Wiley and Sons Ltd., Chichester, West Sussex, England, 1998.

[7] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. *RFC 2460*, December 1998.

[8] C. E. Hopps. Routing IPv6 with IS-IS (Internet Draft). *http://www.ietf.org/internet-drafts/draft-ietf-isis-ipv6-05.txt*, 2003.

[9] IBM web site (search for CoreConnect). *http://www.ibm.com*.

[10] The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA. *IEEE Std 802.3, 1998 Edition. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, 1998.

[11] J. Lilius and D. Truscan. UML-driven TTA-based protocol processor design. In *Proceedings of the 2002 Forum for Design and Specification Languages (FDL'02)*, Marseille, France, September 2002.

[12] J. Lilius, D. Truscan, and S. Virtanen. Fast Evaluation of Protocol Processing Architectures for IPv6 Routing. In *Proceedings of the 2003 Design, Automation and Test in Europe conference (DATE'03)*, Munich, Gemany, March 2003.

[13] G. Malkin and R. Minnear. RIPng for IPv6. *RFC 2080*, January 1997.

[14] M. A. Miller. *Implementing IPv6, 2nd Edition: Supporting the Next Generation Internet Protocols*. M & T Books, Foster City, CA, USA, 2000.

[15] T. Nurmi, S. Virtanen, J. Isoaho, and H. Tenhunen. Physical modeling and system level performance characterization of a protocol processor architecture. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 294–301, Turku, Finland, November 2000.

[16] M. O'Connor and C. A. Gomez. The iFlow address processor. *IEEE Micro*, pages 16–23, March-April 2001.

[17] The Open SystemC Initiative web site. *http://www.systemc.org*.

[18] J. Postel. User datagram protocol. *RFC 768*, August 1980.

[19] J. V. Praet, G. Goossens, D. Lanneer, and H. D. Man. Instruction set definition and instruction selection for ASIP. In *Proceedings of the Seventh International Symposium on High-Level Synthesis*, pages 11–16, Niagara-on-the-lake, Canada, May 1994.

[20] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). *RFC 1771*, March 1995.

[21] E. Smith. Bus protocols limit design reuse of IP. *EE times*, *http://www.eetimes.com/story/OEG20000515S0026*, May 2000.

[22] D. Tabak and G. J. Lipovski. MOVE architecture in digital controllers. *IEEE Transactions on Computers*, 29(2):180–190, February 1980.

[23] S. Virtanen. On communications protocols and their characteristics relevant to designing protocol processing hardware. Technical Report 305, Turku Centre for Computer Science, Turku, Finland, September 1999.

[24] S. Virtanen, J. Lilius, T. Nurmi, and T. Westerlund. TACO: Rapid design space exploration for protocol processors. In *the Ninth IEEE/DATC Electronic Design Processes Workshop Notes*, Monterey, CA, USA, April 2002.

[25] S. Virtanen, J. Lilius, and T. Westerlund. A processor architecture for the TACO protocol processor development framework. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 204–211, Turku, Finland, November 2000.

[26] S. Virtanen, D. Truscan, and J. Lilius. SystemC based object oriented system design. In *Proceedings of the 2001 Forum on Design Languages (FDL'01)*, Lyon, France, September 2001.

[27] T. Westerlund. Design and implementation of a protocol processor. Master's thesis, University of Turku, Finland, April 2001.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**