

Zurich Research Laboratory
Switzerland

Diploma Thesis
Gero Dittmann

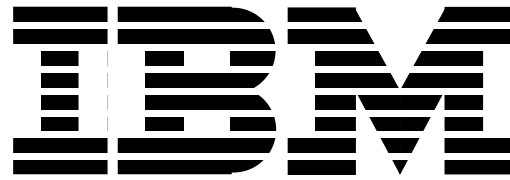
*Programmable Finite State Machines for
High-speed Communication Components*

March 2000

Advisors:
Dr. Andreas Herkersdorf, IBM Research
Dr. Juergen Becker, TU Darmstadt



Institute of Microelectronic Systems
Prof. Dr. Dr. h.c. mult. Manfred Glesner
Department of Electrical Engineering and Information Technology
Darmstadt University of Technology, Germany



Forschungslaboratorium Zürich
Schweiz

Diplomarbeit
Gero Dittmann

*Programmierbare Finite State Machines fuer
Hochgeschwindigkeits
Kommunikationsbausteine*

März 2000

Betreuer:
Dr. Andreas Herkersdorf, IBM Research
Dr. Jürgen Becker, TU Darmstadt



Fachgebiet Mikroelektronische Systeme
Prof. Dr. Dr. h.c. mult. Manfred Glesner
Fachbereich Elektrotechnik und Informationstechnik

Technische Universität Darmstadt, Deutschland

Ehrenwörtliche Erklärung (Word of Honour)

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen

Zürich, den 20. März 2000

Gero Dittmann

Table of Contents

Ehrenwörtliche Erklärung (Word of Honour)	v
Table of Contents	vii
List of Figures	xi
List of Tables	xiii
1 Summary	15
1.1 Context and Motivation.....	15
1.2 Project Objective	15
1.3 Approach	16
1.4 Results	16
1.5 Structure of the Document.....	16
2 Background	17
2.1 OSI Reference Model.....	17
2.2 Internet Protocol Suite.....	18
2.2.1 IPv4	18
2.2.2 IPv6	19
2.2.3 Internet Transport Protocols.....	20
2.3 Evolution of Switching.....	21
2.3.1 Layer 1 Switching	21
2.3.2 Layer 2 Switching	21
2.3.3 Layer 3 Switching	22
2.3.4 Layer 4 Switching	22
2.4 Network Processors	22
2.4.1 History of Packet Processing	22
2.4.2 Architecture Principles.....	23
3 Project Characterization	24
3.1 Goals.....	24
3.2 Challenges	24

3.2.1	High Speed	24
3.2.2	Reconfigurability	24
3.2.3	Real-time	25
3.2.4	Multi-protocol Capability	25
4	Environment	26
4.1	Network Processor Integration	26
5	Instruction Set Architecture	27
5.1	IP State Machine.....	27
5.2	IPv4.....	27
5.2.1	Relevant Header Fields.....	27
5.2.2	Pseudo Code	29
5.3	IPv6.....	29
5.3.1	Relevant Header Fields.....	29
5.3.2	Pseudo Code	30
5.4	Derived Design Decisions	31
5.4.1	Minimal Register Set	31
5.4.2	Essential Functions	31
5.4.3	Parallelism	32
5.4.4	Function Realization.....	33
5.4.5	Summary: The Instruction Set and Format	36
5.4.6	Revised Code.....	38
6	Hardware Architecture	40
6.1	Functional Blocks	40
6.1.1	Instruction Registers	40
6.1.2	Protocol Data Masking	40
6.1.3	Next PC Calculation	40
6.1.4	Constant Registers	42
6.1.5	Accumulator	42
6.1.6	IPv6 HdrExtLen.....	42
6.1.7	Counter	43
6.1.8	Outputs	43
6.1.9	Central Execution Unit.....	43

6.2 Register Scheduling.....	44
7 Implementation	45
7.1 IBM ASIC Design Methodology	45
7.2 Design Structure	46
7.2.1 Partitioning.....	46
7.2.2 Coding Style.....	46
7.2.3 Program Storage.....	47
7.3 Simulation	47
7.4 Synthesis.....	49
7.5 Timing Optimization	50
7.6 Scaling Options	51
7.6.1 Data Bus Width.....	51
7.6.2 Pipelining	52
7.6.3 Parallel Entities	53
7.6.4 Modifying the Internal Architecture	55
8 Conclusion	58
Appendix A: VHDL Source Code	59
Appendix B: Acronyms	71
Appendix C: Bibliography	72

List of Figures

Figure 1: The OSI reference model.....	17
Figure 2: The IPv4 header.....	18
Figure 3: The IPv6 header.....	19
Figure 4: The TCP header.....	20
Figure 5: The UDP header.....	21
Figure 6: Parser interfaces.....	26
Figure 7: IP state machine.....	28
Figure 8: Secondary memory control.....	33
Figure 9: The INIT_CASE command.....	34
Figure 10: Parser components.....	41
Figure 11: HdrExtLen to Counter conversion.....	42
Figure 12: Register scheduling.....	44
Figure 13: Design methodology flow.....	45
Figure 14: Structure of the top level entity.....	46
Figure 15: Testbench.....	47
Figure 16: Simulation wave chart.....	50
Figure 17: Optimized PC2 calculation.....	51
Figure 18: Pipelining the header parser.....	52
Figure 19: Parallel parser entities.....	54
Figure 20: Worst case scheduling for IPv4.....	54
Figure 21: Timing for HdrExtLen routine.....	56
Figure 22: Worst slack report.....	57

List of Tables

Table 1:	Relevant IPv4 header fields.	27
Table 2:	Relevant IPv6 header fields.	29
Table 3:	Instruction set.	36
Table 4:	Standard operand format.	37
Table 5:	LD_CONST operand format.	37
Table 6:	Primary memory word format.	37
Table 7:	Secondary memory word format.	37
Table 8:	IPv6 code for headers with HdrExtLen.	43
Table 9:	IPv6 code for headers with HdrExtLen.	55
Table 10:	Modified IPv6 code for headers with HdrExtLen.	56

1 Summary

1.1 Context and Motivation

In the past partitioning between hardware and software in routers was such that layer 1 was built in hardware, while everything from layer 2 up was handled in software. With the advent of high-speed networking technology with data rates of up to several hundred megabits and more the software part in the forwarding path turned out to be a major bottleneck.

As a consequence, Application Specific Integrated Circuits (ASICs) were developed that are able to handle tasks for layer 2 and 3 forwarding at much higher speed than the software solutions. These networking ASICs took over more and more functions in the upper layer processing. Today, ASICs are able to handle IP routing at wire-speed with data rates of up to 10 Gbps.

However, a new problem has emerged: Today's networking world is changing at a fast pace, new protocols are constantly evolving that need to be supported and they often need to be implemented while still in an immature state. Since algorithms in ASICs are hardcoded they cannot be adapted to recent developments.

A new solution had to be found that would combine the very high speed of ASICs with the flexibility of software based routing. The idea is to design Application Specific Integrated Processors (ASIPs) that are programmable but implement time-critical functions in hardware in a generic and configurable fashion that allows these functions to be used for operations on all kinds of protocols. This kind of ASIP is called a Network Processor (NP).

1.2 Project Objective

One key element of the suite of network processor building blocks is the *Header Parser*. The parser interprets incoming protocol headers, extracts relevant header fields and forwards the contents to other NP components. The objective of this project is to develop a header parser for a next generation high-speed network processor that meets the following requirements:

- Operate at a data rate of 10 Gbps.
- Adapt to future changes or extensions in the protocol definitions and be extendable towards new protocols by means of reconfiguration.
- Parse the headers in real-time, i.e. at the rate they come in.
- Perform parsing on a multitude of protocols: IPv4, IPv6, MPLS, PPP, ATM, ...
- No packets should be dropped while normal operation.
- Packet order should be preserved.

1.3 Approach

First some example protocols were analyzed to find all operations that a parser must perform on protocol headers. Sophisticated Internet protocols were examined to make sure that even complex protocol structures can be handled. Then from these operations a flexible instruction set and a minimal register set were derived. Next, a hardware architecture was developed to realize this specification. The architecture was implemented in VHDL and synthesized to IBM CMOS technology. Finally, several optimization steps were taken in order to meet the timing requirements.

1.4 Results

The instruction set that was found is very flexible and enables the parser to handle future extensions of protocol definitions as well as to be configured for a variety of other protocols. Timing of the circuit is still critical but is expected to be fixed soon. All other requirements have been satisfied completely.

It has been shown that traditional scaling techniques such as pipelining are not applicable to the field of programmable real-time header parsing. However, a one-cycle many-targets branch instruction ('hardware case') and an efficient memory management strategy for parallel systems have been proposed that help to complete complex tasks under real-time conditions and to minimize chip area, respectively.

Future work would be to investigate the applicability of these concepts to other components within the network processor framework. A goal might be to develop a generic building block for various tasks in the processor.

1.5 Structure of the Document

Chapter 2 gives an introduction to network protocols, switching, and the emergence of network processors. Chapter 3 specifies the project goals and challenges. Next, Chapter 4 defines the interface of the header parser. In Chapter 5 an instruction set for the parser is developed using Internet protocols as an example case. This instruction set is then mapped to a hardware architecture in chapter 6. In chapter 7 the implementation and optimization of the architecture is presented. Finally, chapter 8 wraps up the thesis.

2 Background

2.1 OSI Reference Model

In the networking literature protocol stacks are often described using the **Open Systems Interconnection (OSI) Reference Model** as developed by the **International Standards Organization (ISO)** in 1984. Seven layers make up the OSI model:

1. The **physical layer** is responsible for transmitting raw bits over a communication channel. This involves bit encoding, voltage levels, timing, connectors, transmission medium and alike.
2. The **data link layer** structures the bit stream of the physical layer into data frames. It repartitions the data from the network layer into frame payloads and puts them onto the wire. In this scope it takes care of flow control, error handling, and it may provide different service classes. On broadcast networks the data link layer handles the **Medium Access Control (MAC)** and addressing issues.
3. The **network layer** provides the means to cross subnet borders and to route data to remote subnets. It has to control the paths along which data is routed and handle network congestion. Also, billing issues are often handled here.
4. The **transport layer** is the first end-to-end layer. Its functions typically include flow control, multiplexing, and error checking and recovery.
5. The services provided by the **session layer** include token management, for cases where communicating users must not attempt the same operation at the same time, and synchronization, which allows to resume a session after a crash without a complete retransmission.
6. The **presentation layer** is concerned with coding and conversion functions. It manages abstract data structures and copes with different data representations on different computer systems.
7. The **application layer** deals with functions like identifying communication partners, determining resource availability, and synchronizing communication. Typical protocols in this layer handle e-mail, file transfer, or remote terminal access.

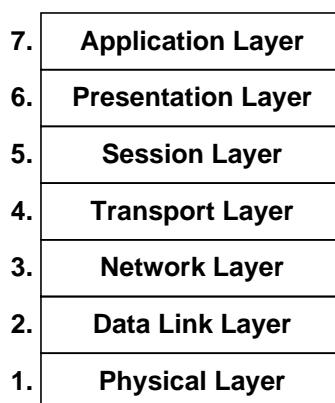


Figure 1 The OSI reference model.

2.2 Internet Protocol Suite

The **Internet Protocol (IP)** suite was designed to allow data transfers over all kinds of different data link layers. Thus, there is no specific IP link layer definition. In the LAN the most common data link technology in IP networks is still IEEE 802.3, popularly called Ethernet. In the backbone market one will find e.g. Asynchronous Transfer Mode (ATM) or Packet over SONET (PoS).

IP itself represents the network layer. The major issues here are routing packets between hosts that may reside on different subnets with possibly different data link protocols, and avoiding network congestion. There are currently two versions of IP: The older version 4 (IPv4) is predominant in today's Internet, and version 6 (IPv6) is supposed to replace it some time in the future. Both will be introduced in the following sections.

2.2.1 IPv4

An IPv4 packet consists of a header and a data part. The header format as defined in [Pos81] is shown in Figure 2.

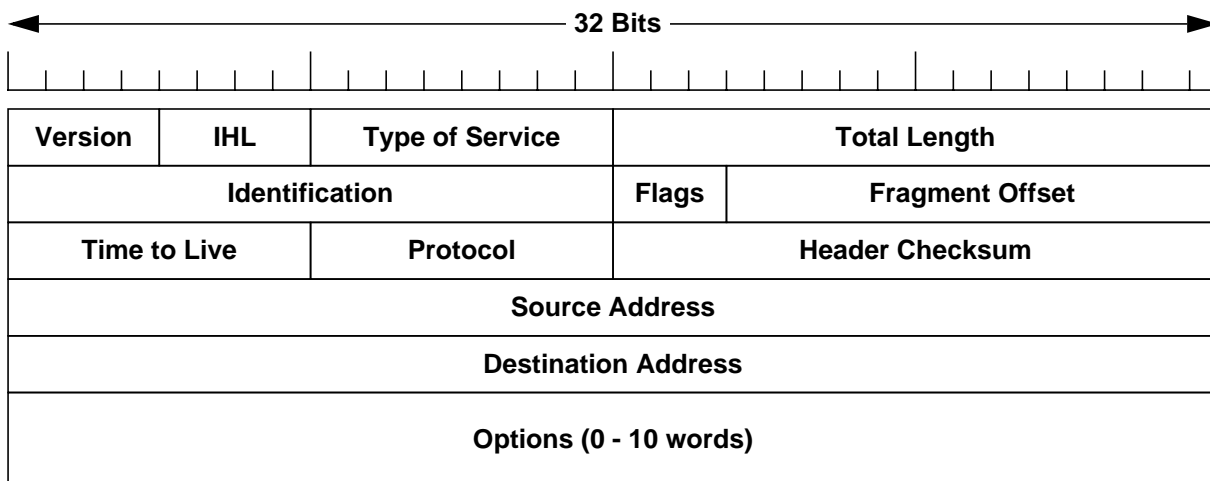


Figure 2 The IPv4 header.

The Version field is used to determine the IP version the packet belongs to. The Internet Header Length (IHL) indicates the length of the header in 32-bit words. Its minimum value is 5, which applies if no options are present. The maximum value of this 4-bit field is 15, limiting the options to 10 words.

The Type of Service (ToS) field is supposed to carry information about the Quality of Service (QoS) required by the packet. It used to be ignored by most routers but now it is revived by new QoS architectures such as Differentiated Services (DiffServ).

The Total Length includes header and payload and is measured in octets. Identification, Flags and Fragment Offset handle datagram fragmentation. The Time to Live (TTL) field is a counter used to limit packet lifetimes. It was supposed to count time in seconds, but it really just counts hops. When the TTL field is zero the packet is discarded.

The Protocol field identifies the upper layer protocol that the payload belongs to, e.g. TCP or UDP. The up-to-date list of all defined numbers can be found at [IANA]. The Header Checksum verifies the header only. Source and Destination Address obviously indicate the originating host and the delivery address. Finally, there is room for Options, e.g. source routing or record route.

2.2.2 IPv6

A major problems of today's Internet is that it is running out of addresses with more and more devices being connected to it. This and other issues triggered the development of IPv6 [DH98]. The IPv6 header is shown in Figure 3.

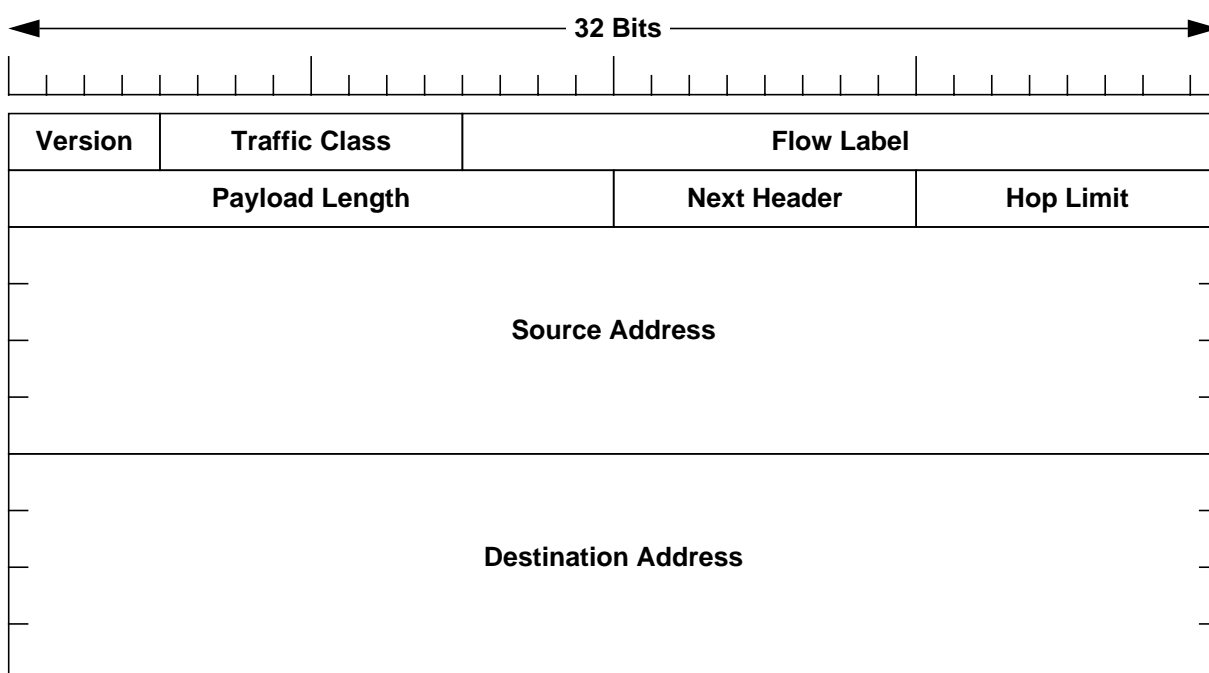


Figure 3 The IPv6 header.

The Version field identifies the version 6 header. The Traffic Class field provides a way to distinguish between different classes or priorities of packets. The Flow Label allows to label sequences of packets for which a special handling by the routers is requested. Clearly, the latter two fields are intended to support QoS.

The length of the payload in octets is given by the Payload Length field. Extension headers are considered part of the payload, but the IPv6 header is not. The Next Header field identifies the type of header following the IPv6 header, which is either an extension header or an upper layer header. The numbers used here are the same as for the IPv4 Protocol field.

The Hop Limit value is decremented by 1 by each node. If it is decremented to zero the packet is discarded. Finally, the Source and Destination Addresses have been expanded to 128 bits.

2.2.2.1 IPv6 Extension Headers

The IPv4 header options have been substituted in IPv6 by Extension Headers that are placed between the IPv6 header and the upper layer header in a packet. At this time the following Extension Headers have been defined:

- Hop-by-Hop Options
- Destination Options
- Routing
- Fragment
- Authentication ([KA98])
- Encapsulating Security Payload (ESP) ([KA98b])

The first octet in every Extension Header is the Next Header field that identifies the header following that particular Extension Header.

The Fragment header has a defined length of 8 octets. In the other Extension Headers the second octet is a **HdrExtLen** field that gives the length of the Extension Header in 8-octets units, not including the first 8 octets.

The ESP header has a special format which is completely different from the others. This is due to the fact that all other headers following the ESP header are encrypted.

2.2.3 Internet Transport Protocols

In the Internet transport layer there are two main protocols: the connection-oriented **Transmission Control Protocol (TCP)** as defined in [ISI81], and the connectionless **User Datagram Protocol (UDP)** which has been defined in [Pos80].

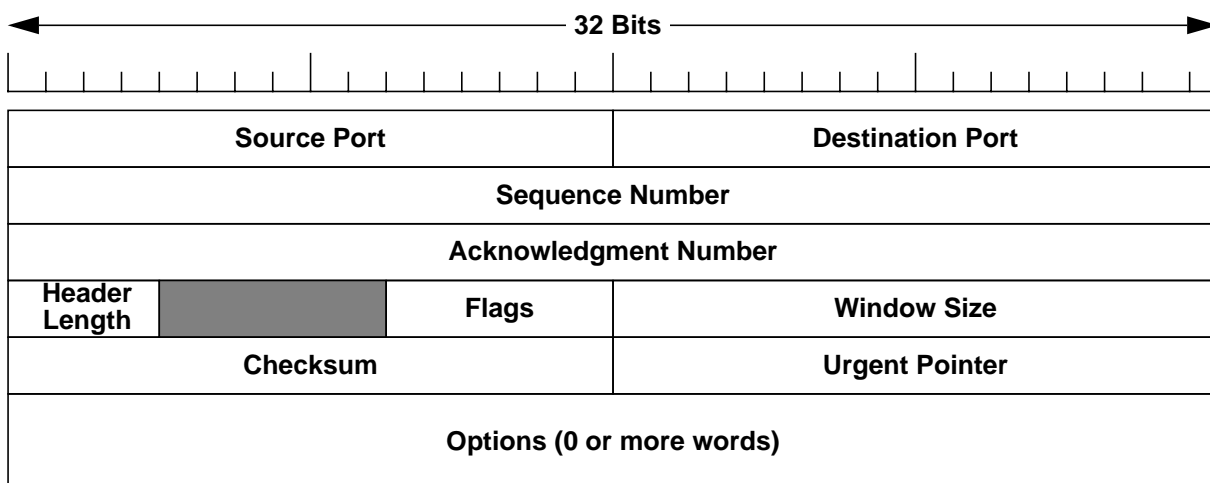


Figure 4 The TCP header.

In IP the transport service access points are called **ports**. They identify the application that a datagram belongs to. TCP and UDP both have in common that their port numbers reside in the first 32 bits of their headers. The first two octets are the source port and the next two octets are the destination port.

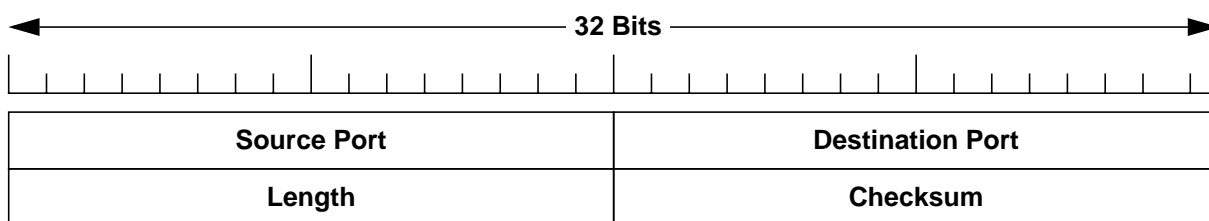


Figure 5 The UDP header.

2.3 Evolution of Switching

‘Switching’ has been one of the big buzz-words in the networking industry for the last few years. There is no precise definition of ‘switching’, as it is often only used as a marketing instrument, and it has different meanings in different layers of the protocol stack. All these meanings have in common that they refer to the way of data relay specific to the respective layer.

2.3.1 Layer 1 Switching

Layer 1 relay nodes are usually called ‘**repeaters**’ or ‘**hubs**’. The term ‘hub’ is often used for repeaters with only one host per port. Repeaters regenerate the electrical and timing characteristics of signals. But all ports on a repeater are part of the same LAN and share the LAN’s bandwidth.

Some repeaters offer a feature called ‘port switching’ which allows to assign individual ports to different LAN’s inside the repeater, which might then be called a ‘switching hub’. This is only a configuration feature and should not be confused with bridging in layer 2.

2.3.2 Layer 2 Switching

Layer 2 interconnection nodes in packet-oriented networks are called ‘**bridges**’, ‘layer 2 switches’ or ‘LAN switches’. By continuously monitoring traffic they learn which layer 2 addresses reside on which of their ports. This enables them to forward packets only to the port that the destination is attached to, thereby greatly increasing the available bandwidth of a data link.

Many bridges offer an additional feature often called ‘Virtual LAN’ (VLAN). This limits packet propagation between ports based on configured port groups, layer 2 addresses, layer 3 protocol types, or layer 3 subnets. Some vendors even apply the term ‘layer 3 switching’ to what is really only bridging with this limited degree of layer 3 awareness.

Another type of layer 2 switches are the nodes in label swapping networks, such as Multi Protocol Label Switching (MPLS) or ATM. While determining routes or setting up connections using complex layer 3 signalling, traffic forwarding in such networks is performed at layer 2 by considering only labels or Virtual Circuit Identifier (VCI) / Virtual Path Identifier (VPI) pairs, respectively, instead of layer 3 addresses. Because these labels are valid only on a per-link basis and may change on every link from source to destination, this forwarding algorithm is called ‘label swapping’.

2.3.3 Layer 3 Switching

Nodes forwarding packets at layer 3 are called '**routers**'. Vendors also describe these products as 'layer 3 switches', 'multilayer switches', 'routing switches', 'switching routers' etc. Routers divide networks up into subnets that scale a lot better than bridged networks, because layer 2 broadcast traffic does not cross subnet borders and addressing is organized in a much more efficient fashion.

Routers exchange information about the network topology using routing protocols -- for IP this might be e.g. Open Shortest Path First (OSPF) or the Border Gateway Protocol (BGP). From this information they build routing tables that associate layer 3 destination addresses with a next hop. Destination addresses are usually aggregated by network classes that represent groups of hosts with a common next hop. For each incoming packet the destination address is looked up in the routing table and the packet is then forwarded to the associated next hop.

Forwarding decisions may be based not only on addresses but also on QoS policies, e.g. utilizing the pertinent header fields in IP: ToS, or Traffic Class and Flow Label.

2.3.4 Layer 4 Switching

While in layers 2 and 3 data is operated packet by packet, at layer 4 exists the notion of packet sequences and information about what higher-level application a sequence is associated with. In IP this information is represented by the protocol type, e.g. TCP or UDP and the respective port numbers. The sequences are typically called '**flows**'. Flow classification can be exploited to provide higher priorities or even QoS guarantees to traffic that belongs to certain applications, e.g. real-time service for multimedia streams.

Although not really forwarding at layer 4, devices with the described functionality are called 'layer 4 switches' or even 'layerless switches'.

2.4 Network Processors

2.4.1 History of Packet Processing

Some ten years ago partitioning between hardware and software in routers was such that layer 1 was built in hardware, while everything from layer 2 up was handled in software running on a **General Purpose Processor (GPP)**. With the advent of high-speed networking technology with data rates of up to several hundred megabits and more the software part in the forwarding path turned out to be a major bottleneck.

As a consequence, **Application Specific Integrated Circuits (ASICs)** were developed that are able to handle tasks for layer 2 and 3 forwarding at much higher speed than the software solutions. These networking ASICs took over more and more functions in the upper layer processing. Often the term 'switch' is applied to devices that substitute software solutions with ASICs. The process was further supported by the convergence to IP as the universal layer 3 protocol, making the multiprotocol literacy of software based routers obsolete. Today, ASICs are able to handle IP routing at wire-speed with data rates of up to 10 Gbps.

However, a new problem has emerged: Today's networking world is changing at a fast pace, new protocols are constantly evolving that need to be supported and they often need to be im-

plemented while still in an immature state. With protocol handling being hardcoded into an ASIC every time protocol specifications change or new protocols become relevant new ICs need to be developed, new boxes need to be designed around them, and thus new equipment needs to be purchased and deployed into the network. This process consumes a lot of time and financial resources.

A new solution had to be found that would combine the very high speed of ASICs with the flexibility of software based routing. The idea is to design **Application Specific Integrated Processors (ASIPs)** that are programmable but implement time-critical functions in hardware in a generic and configurable fashion that allows these functions to be used for operations on all kinds of protocols. This kind of ASIP is called a **Network Processor (NP)**.

Being able to deploy network processing changes in software instead of hardware is not only cheaper but also considerably reduces time-to-market.

2.4.2 Architecture Principles

Network processors sit on the data path between the physical interface chip and the backplane of a switch. Typical functions performed by an NP include:

- Packet Forwarding
- Packet Segmentation and Reassembly
- Protocol Recognition and Classification
- Queuing and Access Control
- Traffic Shaping and Engineering
- Quality of Service Provisioning and Management

The central control entity in a switch that is responsible for switch configuration, building routing tables, keeping track of policies etc. will still be software running on a GPP, since these tasks are not timing sensitive but too complex to be implemented in hardware.

From all the companies that now work on NPs two major architecture alternatives can be distilled:

- The *'brute force approach'*, with a multitude of GPP cores on one chip with only marginal dedicated networking hardware. These processors are able to handle huge amounts of packets just by their sheer computing power.
- The *'dedicated approach'*, with specialized components that have a new instruction set designed from scratch and highly optimized for real-time network processing.

This thesis considers the latter approach. The major advantage of the dedicated approach is that it is designed for real-time applications with a determined time for code execution. Because of techniques like branch-prediction and normal interrupt handling GPPs can only provide a statistical estimate about their code execution time. Just like with legacy software routers this leads to load-dependent performance and eventually to dropping packets. However, processors following the dedicated approach will be designed to provide true wire-speed processing with no dropping of incoming packets.

3 Project Characterization

One key element of the suite of network processor building blocks is the *Header Parser*. The parser interprets incoming protocol headers, extracts relevant header fields and forwards the contents to other NP components. In this project a header parser for a next generation high-speed network processor should be designed.

3.1 Goals

The following requirements for a header parser being part of a high-speed NP were given:

- Operate at a data rate of 10 Gbps.
- Adapt to future changes or extensions in the protocol definitions and be extendable towards new protocols by means of reconfiguration.
- Parse the headers in real-time, i.e. at the rate they come in.
- Perform parsing on a multitude of protocols: IPv4, IPv6, MPLS, PPP, ATM, ...
- No packets should be dropped while normal operation.
- Packet order should be preserved.

Subject of this thesis was to develop a header parser satisfying these requirements. In order to provide the demanded flexibility, a configurable or preferably even a programmable approach should be taken, while meeting the challenges of high speed in a real-time environment.

3.2 Challenges

3.2.1 High Speed

Assuming a protocol data width of 32 bit, a data rate of 10 Gbps results in a word rate of 312 MHz. Parsing header words at the rate they come in leaves only 3.2 ns to perform all necessary operations on a word.

While today's General Purpose Processors have already entered the 1 GHz domain, this is only possible for full custom design. For ASIC design using standard design libraries, which is mandatory in order to achieve a reasonable time-to-market, clock rates of more than 300 MHz still are a big challenge.

3.2.2 Reconfigurability

Configurations need to be held in storage elements. The access time of these elements must be considerably smaller than clock cycle time to allow for combinational operations in the same cycle. Nevertheless, storage capacity must be large enough for complex protocol definitions and a high degree of flexibility.

3.2.3 Real-time

Handling data traffic at wire-speed with no packet loss requires a deterministic, non-varying execution time per instruction. In particular, the instructions to be executed in the next step need to be determined within only one clock cycle in order to be available on time. Assuming a RAM based architecture, this entails several implications:

- The computation of the next program address and the memory access need to fit into one clock cycle. This puts great demands on the speed of address computation as well as on the memory access time.
- Instructions succeeding branch operations need to be executed in the clock cycle directly following the branch. No statistical methods known from GPP design, such as branch prediction, can be used.
- This also applies to jumps with a multitude of possible targets. These jumps are necessary to handle optional headers and extension headers.

It is also desirable to do the parsing as fast as possible in order to keep data latency through the NP low.

3.2.4 Multi-protocol Capability

A major feature of the header parser must be the ability to handle as many protocols as possible, existing as well as future developments. Thus, a very flexible instruction set needs to be found that is still not too complex to operate at high speed.

Also, it should be possible to switch between protocols, as physical links can be shared by several upper-layer protocols.

4 Environment

4.1 Network Processor Integration

The header parser is the first component in the NP that incoming data encounters. Data is not passed through the parser but is buffered and updated by the NP until the next hop has been determined.

Besides clock and reset, its main inputs are a Protocol Data port and a 'Delimiter' flag that marks the start of a new data packet. The outputs towards the following NP components are a port for the extracted data that has the same width as the data input, and an ID port that identifies the data. These ports will connect the parser e.g. to a route lookup engine and to a component that is responsible for updating header fields.

For the configuration there needs to be a micro processor interface through which the storage elements in the parser can be loaded, i.e. an address port, a data port, and a write enable flag.

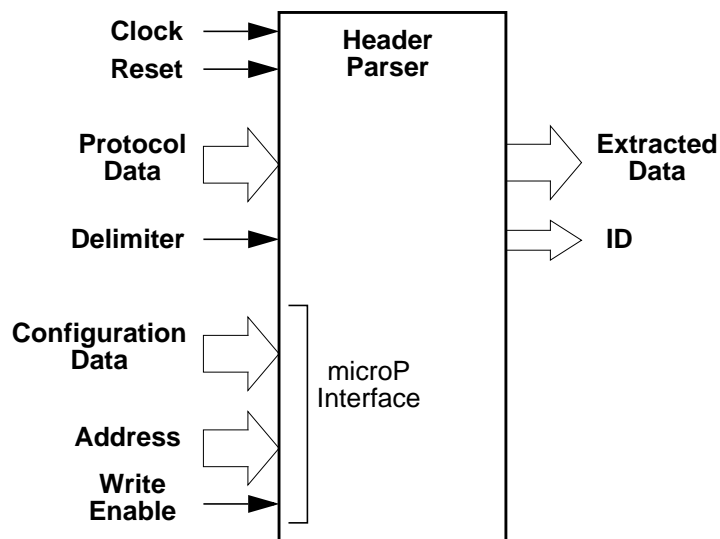


Figure 6 Parser interfaces.

Since many relevant protocols, namely IPv4 and IPv6, are defined in a 32 bit grid, the bus width for Protocol Data and Extracted Data has been assumed to be 32 bit. This definition will be reviewed later on.

5 Instruction Set Architecture

In order to find an appropriate instruction set that would enable the parser to handle a variety of protocols, two very complex but widely used protocols have been chosen as example cases: IPv4 and IPv6 with their transport protocols TCP and UDP. First it was investigated which header fields are relevant for assigning a given header to a flow. Then a pseudo code was developed that would parse a header of the given IP version. Finally, the necessary instruction set was derived from this analysis.

5.1 IP State Machine

Figure 7 gives an overview of a state machine for both IP versions that extracts the header fields which are relevant to the routing process. A major challenge for a hardware realization will be the many possible transitions from a single state in IPv6. For now, only one IP version at a time is handled by the parser. This is described in greater detail in the following sections.

5.2 IPv4

5.2.1 Relevant Header Fields

Assuming a 32 bit pipeline, Table 1 shows the relevant fields of the IPv4 header including the TCP/UDP port numbers. The ‘Step #’ column denotes clock cycles since the start of the header.

Step #	Fields relevant	
	internally	externally
1	IP Header Length (IHL) [4-7]	Type of Service (ToS) [8-15]
2	-	-
3	-	Protocol [8-15]
4	-	Source Address [0-31]
5	-	Destination Address [0-31]
wait (IHL-5) {max. 10} for layer 4 header	-	TCP/UDP: Source Port [0-15], Destination Port [16-31]

Table 1. Relevant IPv4 header fields.

Internally relevant information is needed to interpret the header structure and thus needs to be stored in registers for later reference. The IHL in particular needs to be count down in order to find the beginning of the layer 4 header.

Externally relevant fields are to be forwarded to the client components in the NP immediately.

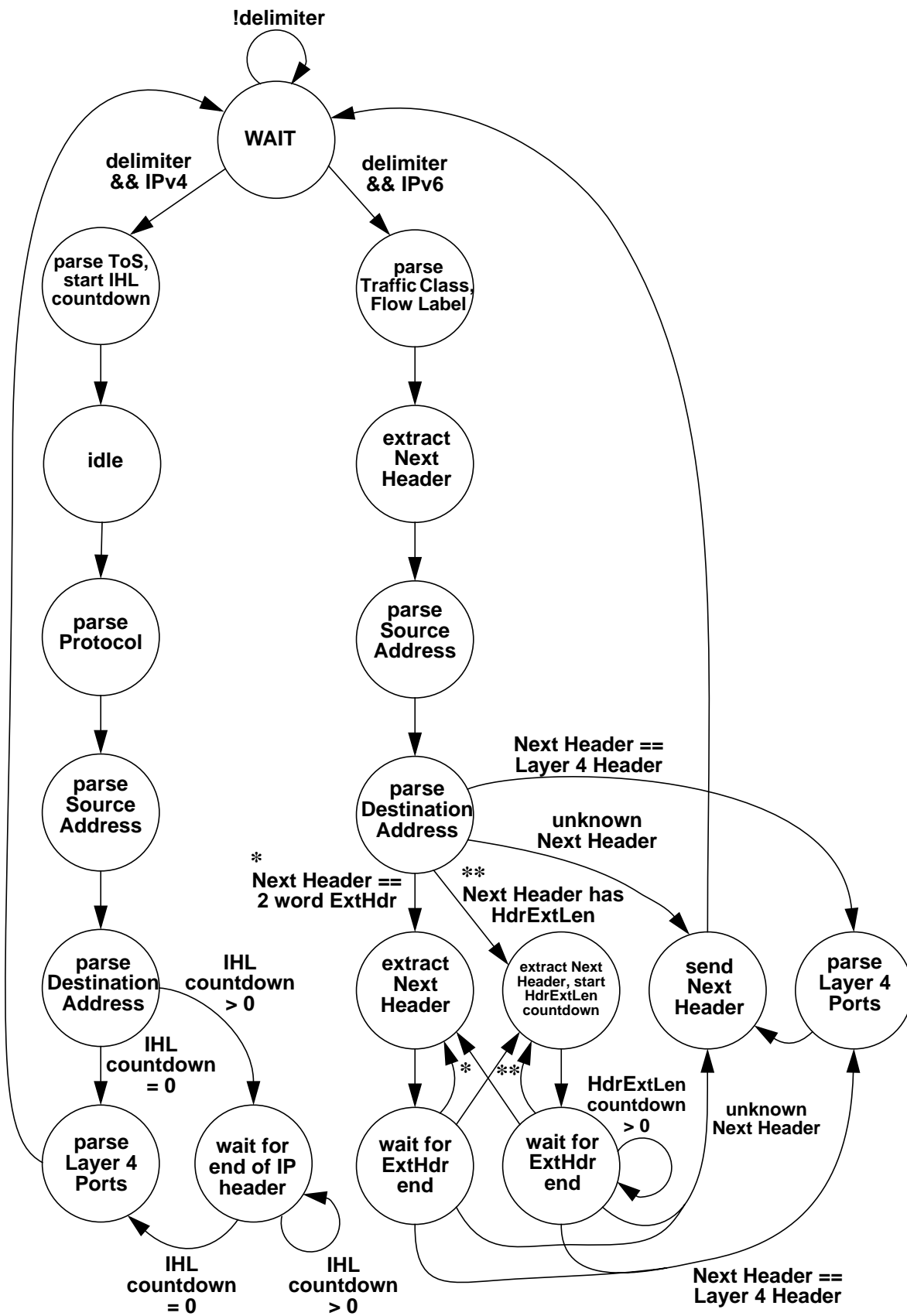


Figure 7 IP state machine.

5.2.2 Pseudo Code

The following conventions apply to the given pseudo code:

Commands on the same numbered line and separated by a semicolon are to be executed in the same cycle, i.e. in parallel. 'counter' is supposed to be a register that is implicitly decremented by one every clock cycle. 'send' is a command that applies the specified portion of the input data to the Extracted Data output port. The second argument specifies a signal to be sent to the ID port, indicating the type of Extracted Data.

All commands are described in section 5.4.2.

```

1)  if (!delimiter) then goto 1
2)  counter <= IHL[4-7]; send(TOS[8-15], 1)
3)  nop
4)  send (Protocol[8-15], 2)
5)  send (SourceAddress[0-31], 3)
6)  send (DestinationAddress[0-31], 4)
7)  if (counter=0) then send ((SourcePort, DestinationPort)[0-31], 4)
    // if Protocol is neither TCP nor UDP, there are no port numbers here!
    else goto 7
8)  goto 1

```

This is straight forward except for line 7. Here the program flow waits until the counter has counted down the IP header. Once the counter goes off, the first 32 bits of the following layer 4 header are sent. Provided it is a TCP or UDP header these are the transport ports. Otherwise this information will be ignored by client components.

5.3 IPv6

5.3.1 Relevant Header Fields

In the same fashion as before, Table 2 shows the relevant header fields for IPv6 + TCP / UDP.

Step #	Fields relevant	
	internally	externally
1	-	Traffic Class[4-11], Flow Label[12-31]
2	Next Header[16-23]	-
3-6	-	Source Address [0-31]
7-10	-	Destination Address [0-31]
wait for NextHeader = layer 4 header	NextHeader[0-7], HdrExtLen[8-15]	stored layer 4 NextHeader
wait for end of IP header => layer 4 header		TCP / UDP: Source Port [0-15], Destination Port [16-31]

Table 2. Relevant IPv6 header fields.

5.3.2 Pseudo Code

For IPv6 handling an additional register needs to be introduced that stores the NextHeader number for later interpretation. This register is named 'reg[NextHeader]'.

```
1)  if (!delimiter) goto 1
2)  send ((TrafficClass, FlowLabel)[4-31], 1)
3)  reg[NextHeader] <= NextHeader[16-23]
4)  send (SourceAddress[0-31], 2)
5)  send (SourceAddress[0-31], 3)
6)  send (SourceAddress[0-31], 4)
7)  send (SourceAddress[0-31], 5)
8)  send (DestinationAddress[0-31], 6)
9)  send (DestinationAddress[0-31], 7)
10) send (DestinationAddress[0-31], 8)
11) send (DestinationAddress[0-31], 9)
12) if ( reg[NextHeader] = (6|17) [~ (TCP|UDP)] ) then {
    12a) send (SourcePort, DestinationPort)[0-31], 10)
    13a) send (reg[NextHeader], 11)
    14a) goto 1}
    else if ( reg[NextHeader] = 44 [~ Fragment Header] ) then {
    12b) reg[NextHeader] <= NextHeader[0-7]
    13b) goto 12} // Fragment Header length is two words
    else if ( reg[NextHeader] = (0|43|51|60)
    [~ all other defined Extension Headers]) then {
    // they have a HdrExtLen field
    12c) reg[NextHeader]<=NextHeader[0-7]; counter<=HdrExtLen[8-15]*2+1
    13c) if (counter=1) then goto 12 else goto 13c}
    else // it is an upper layer or ESP header {
    12d) send (reg[NextHeader], 11)
    13d) goto 1}
```

A complex block is starting in line 12 where depending on the type of extension header different code has to be executed. If the NextHeader number corresponds to TCP or UDP then the port numbers are sent and the actual packet is done. In the case of a Fragment Header, which has a length of two 32 bit words, the NextHeader field is stored and a second cycle is let pass before another iteration of the block is started.

The numbers checked for in the third case correspond to Hop-by-Hop Options (0), Routing Header (43), Authentication Header (51), and Destination Options (60). These headers have a HdrExtLen field from which the number of cycles until the next header is calculated and stored in the Counter (see section 5.4.2.9). Also, the NextHeader number is extracted and stored. In the next line the Counter is tested in a loop until it has the value '1'. Then a new header starts in the following cycle and the block is reiterated.

The fourth case is the default case. It applies to unknown header numbers, to upper layer headers with no port numbers, or to an Encapsulating Security Payload (ESP) header ([KA98b]) which means that the rest of the packet will be encrypted and thus not be accessible to processing in the network. Since in these cases there is nothing for the parser to do on the rest of the packet, the last stored NextHeader number is sent to the client components which might apply special procedures to packets depending on this number. Finally, handling of this packet terminates.

5.4 Derived Design Decisions

5.4.1 Minimal Register Set

As already implied in the pseudo code, at least two registers are necessary:

1. A COUNTER register for counting down length indicators (IHL, HdrExtLen). It can be loaded with a slice of the input data and is decremented by one every clock cycle.
2. One general purpose register that is needed to store the NextHeader identifiers in IPv6. Since it is the only general purpose register so far, it will be called the ACCUMULATOR.

In both IP versions length fields as well as option IDs have a width of 8 bit. This allows for header lengths of 255 words or for 255 different options and should be sufficient for both registers.

5.4.2 Essential Functions

5.4.2.1 ‘if (!delimiter) goto 1’

The first line of both pseudo code programs only waits for an occurrence of the Delimiter signal. This can be implemented as a kind of special interrupt that restarts the whole program every time the Delimiter marks the start of a new packet.

5.4.2.2 ‘goto X’

This will be an unconditional branch to any address in the program.

5.4.2.3 ‘counter <= [4-7]’

One instruction loads the Counter with a slice of the input data.

5.4.2.4 ‘reg[NextHeader] <= [16-23] / [0-7]’

A slice of the input data is loaded into the Accumulator. The slice borders are variable. This function can be combined with the Counter load into one ‘write_reg’ command with the target register being specified by an operator.

5.4.2.5 ‘nop’

No operation. This is a means to wait for the next cycle with no action.

5.4.2.6 ‘if (counter=0) then’, ‘if (counter=1) then’

A conditional branch in the program flow has to be provided that depends on the result of an equality check of the Counter versus zero or one.

5.4.2.7 ‘send ([8-15] / [0-31] / [4-31] / [16-23], id)’

This function of course represents the main functionality of the parser: A slice of the input data is applied to the Extracted Data port and identified towards downstream components by a signal (id) on the ID port. Again, slice borders variable.

5.4.2.8 'send (reg[NextHeader], id)'

Here, a slice of the input data, that has been stored in the Accumulator before, is sent to the Extracted Data port along with an ID.

5.4.2.9 'counter <= HdrExtLen[8-15]*2+1'

This is a special function for IPv6. The HdrExtLen field of an IPv6 extension header gives the length in 8 octet units, not including the first 8 octets. Thus, to feed the Counter with the right number of cycles until the end of the extension header, the value of the field needs to be multiplied with 2, to get the length in 32 bit words. Then, a one has to be added to the result to make up for the missing first two 32 bit words. One of those two words will appear at the input in the current cycle -- the one that includes the HdrExtLen field -- and thus does not need to be counted.

After this operation, when the Counter reaches the value '1', a new extension header will start with the following clock cycle.

5.4.2.10 Handling IPv6 Extension Headers

In the pseudo code a big if/then/else construct determines the code to be executed depending on the occurring NextHeader number. Following the one-cycle-per-instruction goal, this has to be implemented as one command.

5.4.3 Parallelism

In the pseudo code some functions are issued in parallel (IPv4 - line 2; IPv6 - line 12c). This must be supported by the header parser.

One possibility to support parallel operations would be to introduce macro instructions that trigger two actions in the same cycle. This would entail one additional instruction for every needed combination of parallel functions. For the analyzed IP versions, three macro commands are necessary, but for other protocols other combinations of commands might be necessary.

Also, a program memory word would have to hold two sets of operands, one for each parallel function. Since the memory occupation of this solution would not be considerably lower than with two full blown instructions, the first approach was dropped in favor of a simple instruction set of which two instructions can be issued in parallel. This makes the architecture much more flexible.

There are a number of alternatives how to store these two instructions in memory. With the speed the parser is operating at it is not practical to read two instructions from two consecutive memory addresses in one clock cycle. Thus, the two instructions need to be read at the same time. This might be realized by making memory word width twice the size of an instruction and this way storing two instructions at each memory address. But the pseudo codes show that only very few parallel commands are needed and thus a lot of memory would be wasted with NOPs.

This analysis led to a sophisticated architecture with two program memories each of which is assigned a separate Program Counter (PC) as depicted in Figure 8. Every cycle an instruction is read and executed from the primary memory. Only if a 'parallel' bit is set in the primary memory, then a new PC2 is computed and in the next cycle an instruction is also fetched from the secondary memory which is executed in parallel to the instruction from the primary memory.

This way the secondary memory can be sized much smaller than the primary one which will save chip area.

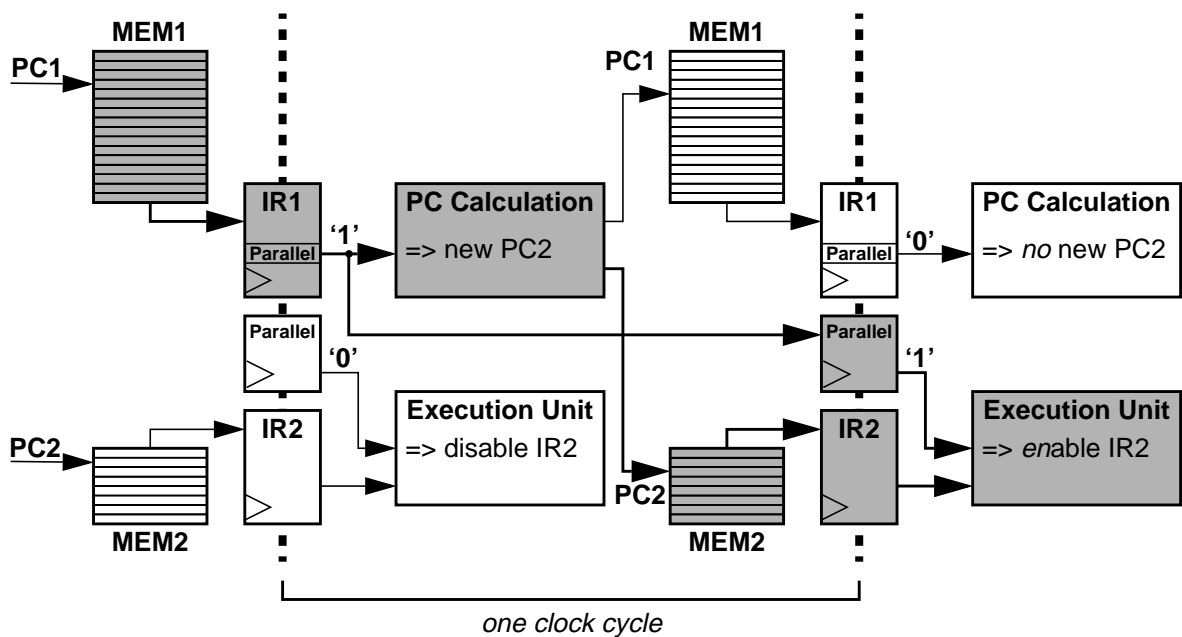


Figure 8 Secondary memory control.

After a reset, the ‘parallel’ bit will be set and the first instruction in the secondary memory will be executed.

5.4.4 Function Realization

This section describes the mapping of the required functions to the header parser instruction set.

5.4.4.1 Loading Registers

An instruction called ‘write_reg’ stores a specified slice of input data in either the Counter or the Accumulator. Therefore it has three operands: `low` and `high` represent the slice borders and `reg` specifies which of the two registers is the target.

For IPv6 there is a special instruction called ‘IP6_counter’ that does the IPv6 HdrExtLen calculation described in section 5.4.2.9 with a given data slice before storing it in the Counter. This instruction has also `low` and `high` operands.

As a future improvement it should be investigated how this calculation could be made more flexible through another operand or configuration.

5.4.4.2 Send

The ‘send’ instruction forwards a range of bits from the Protocol Data bus to the Extracted Data bus. The range is determined by `low` and `high` operands which denote the position of the first and last bit to send, respectively. An `id` parameter gives the signal to be sent to the ID port which identifies the sent data.

A variation of the send instruction is ‘send_reg’. Here, the data source is the Accumulator. Its content is forwarded to the Extracted Data bus and an `id` operand is applied to the ID port.

5.4.4.3 NOP

'nop' performs No OPeration, i.e. a wait cycle.

5.4.4.4 Optional / Extension Headers

A special case is the big if/then/else construct for IPv6. One way to implement this would be to define another special IPv6 instruction that would check the NextHeader value in the Accumulator and would perform the first command in the block according to the header type -- all in one clock cycle. Then it would branch to an address where the rest of the option handling would be done.

Since IPv6 is one of the protocol standards that will very likely be extended in the future, this rigid way of implementation does not seem appropriate. Besides, this command would not be of any use to other protocols.

In order to achieve a more flexible design an 'init_case' instruction was introduced (see Figure 9). This command checks the Accumulator one cycle in advance -- in the IPv6 pseudo code this would be in parallel to the last 'send (DestinationAddress[0-31], 9)' in line 11. Depending on the result, a jump for one out of a multitude of distances is executed. Jump distance is determined by multiplying the resulting case number with a number of code lines allowed for each case - e.g. 4. The last case would always be the 'default' case which is taken if no condition matches. At the respective jump destinations any code can be filled in. Thus, handling of specific headers is totally flexible.

The numbers that the Accumulator is compared to are not hard coded. Instead they are held in a special array of 8 bit wide "CONSTANT REGISTERS" in which the header numbers to look for have been stored before. The number of Constant Registers exceeds the number of extension headers in the pseudo code in order to be prepared for future protocol extensions. The implementation allows for 32 header numbers.

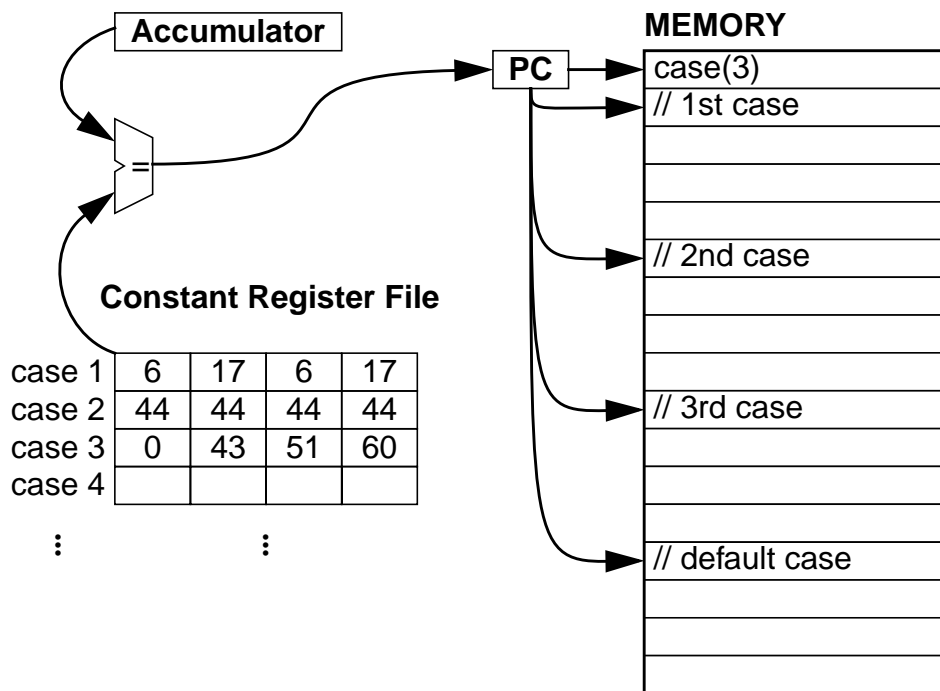


Figure 9 The INIT_CASE command.

In the pseudo code multiple header types are handled the same way, e.g. UDP and TCP or all headers with a HdrExtLen field. This is taken into account by assigning four Constant Registers to each case, i.e. all four numbers of one case will result in a jump to the same address.

The number of cases to consider, i.e. the number of code blocks to choose from, can also be held variable. The `init_case` command would have a `num_cases` operand that specifies the number of cases to be considered, excluding the default case. This approach could be extended to also parameterizing the jump distance per case or the starting point in the constant register file and the number of compare values per case, thus allowing for multiple `init_case` statements in the code.

Of course, every step of parameterization adds not only flexibility but also complexity to the design which has a negative impact on meeting timing constraints. Thus, the extent of parameterization should be carefully thought over. As of now, `num_cases` is the only parameter of the `init_case` instruction. The jump distance per case is 4 for memory #1 and 2 for memory #2.

Another question is how to load the Constant Registers. One way would be map them into the address space and load them via the micro processor interface. Another way would be to leave this to the program code.

For the implementation a new instruction has been introduced, called `ld_const`. Its parameters are `const_reg`, which specifies a particular Constant Register to be loaded, and `value`, which gives the value to be stored. Usually, a block of these instructions will be called on initialization. Since this block should not be interrupted before all Constant Registers are loaded the Delimiter will not be regarded while a `ld_const` instruction is executed.

5.4.4.5 If / Then

Since the big `if/elseif/else` block has been transformed into the `init_case` instruction, the only `if` statements left are a test for the Counter to be equal to '1' or '0', respectively. Just like the `init_case` instruction the `if` command has to be executed one cycle before the resulting branch can be performed. The `if` in line 13c) of the IPv6 pseudo code would then move to a line which bears two instructions already. Thus, it would be the third instruction to be executed in parallel.

However, it is not acceptable to provide the additional infrastructure to generally allow a third instruction just because of this single case. Instead, only the `if` command is allowed in parallel to two standard instructions. Since the Counter is the only source for the condition check and there are only two possible values necessary to compare it to, two more bits in an instruction word of the primary memory are sufficient for this functionality. The first bit activates the command, and the second bit specifies whether the Counter is to be compared to '1' or '0'.

The consequence of an `if` command are: IF the test result is TRUE, then the PC is incremented by one as usual. ELSE -- if the test result is FALSE -- then the PC is incremented by two, i.e. the following line of code is skipped.

5.4.4.6 Delimiter

Waiting for a new packet is implemented as suggested before: Every time the Delimiter signal is asserted, the program counters are set to address 1 and the program is restarted. This leaves address 0 for code that is only executed once after a reset or power up, e.g. an initial loop waiting for the first packet to come.

Also, the Parallel bit is set in order to always start under the same conditions, independent of the line in which the code was interrupted by the Delimiter.

5.4.4.7 Goto

With the instruction set so far a ‘goto’ command is often needed as a third parallel command. Because of the relatively small memory address space only a few more instruction bits are sufficient to allow a ‘goto’ in parallel to the two normal instructions in every cycle.

This mechanism also provides an easy way to extend program blocks that would otherwise be limited, such as the one line of initial code at address 0, or the four lines of code for every case after an ‘init_case’ operation. A goto offset has to be given in the last line of the block, and execution can be continued at any address in memory with no interruption. This mechanism will be used e.g. to load the Constant Registers on initialization.

Due to their differing code size the desired jump distances for the two memories will be different. Thus, goto bits are present in both memories.

5.4.5 Summary: The Instruction Set and Format

Table 3 shows all instructions of the header parser with their operands.

Instruction	OP3	OP2	OP1
SEND	id	low	high
SEND_REG	id	--	--
WRITE_REG	reg	low	high
IP6_COUNTER	--	low	high
INIT_CASE	num_cases	--	--
LD_CONST	const_reg / value		

Table 3. Instruction set.

To identify one out of these six instructions the op-code has to comprise three bits.

With a Protocol Data width of 32 bits operands ‘low’ and ‘high’ need five bits to address every possible slice. Thus, OP1 and OP2 are five bits wide.

Operand ‘reg’ selects one of only two alternatives: Counter or Accumulator. For this task two bits would be sufficient. With 32 Constant Registers of which four are assigned to each case, INIT_CASE’s ‘num_cases’ will be in a range from 1 to 8. This suggests to code the operand as ‘num_cases - 1’ to save one bit. Then, with a range from 0 to 7, three bits are enough for the ‘num_cases’ parameter. The third interpretation of OP3 is ‘id’. The pseudo code for IPv4 sends only 5 different data units while the highest ‘id’ value for IPv6 is 11. Thus, as long as only one of these two is considered, ‘id’ will be four bits wide. Since this is the widest requirement of all three interpretations OP3 is four bits wide.

Except for LD_CONST, the instructions have the operand format shown in Table 4.

3 bits	4 bits	5 bits	5 bits
0-2	3-6	7-11	12-16
op-code	OP3	OP2	OP1

Table 4. Standard operand format.

The parameters for LD_CONST do not fit into the three operand scheme. Since the Constant Registers are 8 bits wide, the ‘value’ operand needs to be 8 bits wide, too. The ‘const_reg’ parameter selects one out of the 32 Constant Registers and thus needs to be 6 bits wide. With the 14 bits reserved for operands in a memory word this leaves one bit unused. The operand format for the LD_CONST instruction is shown in Table 5.

3 bits	1 bit	6 bits	8 bits
0-2	3	4-8	9-16
op-code	--	const reg	value

Table 5. LD_CONST operand format.

In addition to the op-code and the corresponding operands several bits are used to provide the functionality described in the preceding sections. In the primary memory there is the ‘parallel’ bit to trigger consideration of the secondary memory, there are two bits for the ‘if counter’ mechanism, and there are some bits for the goto offset in K2 representation to allow for forward and backward jumps. For IP a ‘goto’ width of 6 bits was found to be sufficient for the code documented in section 5.4.6.

A whole entry in the primary program memory can be seen in Table 6, totalling to a width of 26 bits.

3 bits	14 bits	6 bits	1 bit	1 bit	1 bit
0-2	3-16	17-22	23	24	25
op-code	operands	goto offset	test Counter	test 0/1	parallel

Table 6. Primary memory word format.

For the secondary memory the only additional bits is the goto offset. Since this memory is only half the size of the primary memory there are only five goto bits.

Table 7 shows an entry in the secondary memory, totalling to a width of 22 bits.

3 bits	14 bits	5 bits
0-2	3-16	17-22
op-code	operands	goto offset

Table 7. Secondary memory word format.

5.4.6 Revised Code

This section documents how the pseudo codes would be mapped to the instruction set that has been developed above.

5.4.6.1 IPv4

Extracting data relevant for routing in IPv4 is done with the following code in the *primary memory*:

```
0) goto 0 # (wait for Delimiter)
1) write_reg(counter, IHL[4-7])
2) nop
3) send (Protocol[8-14], 2)
4) send (SrcAddress[0-31], 3)
5) send (DestAddress[0-31], 4); if (counter=1)
6) send ((SrcPort, DestPort)[0-31], 5); goto 8
7) if (counter=1); goto 6
8) goto 8 # (wait for Delimiter)
```

In the *secondary memory* resides this code:

```
0) goto 0 # (wait for delimiter)
1) send (ToS[8-15], 1); goto 0
```

The initializing code in line 0 in both memories only waits for the Delimiter to announce the first incoming packet, which will set the PCs to address 1. In the primary memory, the combination of ‘if’ and ‘goto’ in line 7 implicates that, if the Counter test is positive, execution will continue in line 6, and otherwise line 6 will be skipped and line 7 will be executed once more. This results in a loop until the Counter has reached the value ‘1’.

5.4.6.2 IPv6

For IPv6 the new code for the *primary memory* would be:

```

0) ld_const (0,6); goto 25; parallel # TCP
1) send (TrafficClass, FlowLabel[4-31], 1)
2) write_reg(accu, NextHeader[16-23])
3) send (SrcAddress[0-31], 2)
4) send (SrcAddress[0-31], 3)
5) send (SrcAddress[0-31], 4)
6) send (SrcAddress[0-31], 5)
7) send (DestAddress[0-31], 6)
8) send (DestAddress[0-31], 7)
9) send (DestAddress[0-31], 8); parallel
10) init_case(3); parallel
    # 1st case: TCP, UDP
11) send ((SrcPort, DestPort)[0-31], 10); parallel
12) send_reg (accu, 11); goto 24
13)
14)
    # 2nd case: Fragment Header
15) write_reg(accu, NextHeader[0-7]); parallel
16) init_case(3); goto 11; parallel
17)
18)
    # 3rd case: Headers with HdrExtLen field
19) IP6_counter[8-15]; if (counter=1); parallel
20) init_case(3); goto 11; parallel
21) if (counter=1); goto 20; parallel
22)
    # default case
23) send_reg (accu, 11)
24) goto 24 # (wait for delimiter)
    # initialization routine
25) ld_const (2,6); parallel           # TCP
26) ld_const (4,44); parallel         # Fragment
27) ld_const (6,44); parallel         # Fragment
28) ld_const (8,0); parallel          # Hop-by-Hop
29) ld_const (10,51); parallel; goto 24 # Authentication

```

The *secondary memory* would contain the following code:

```

0) ld_const (1,17); goto 12 # UDP
1) nop
2) send (DestAddr[0-31],9) # parallel to MEM1 line 10
    # 1st case
3) nop           # MEM1 line 11
4) goto 9       # MEM1 line 12
    # 2nd case
5) nop           # MEM1 line 15
6) goto 3       # MEM1 line 16
    # 3rd case
7) write_reg (accu, NextHdr[0-7]); goto 10 # MEM1 line 19
8)
    # default case
9) goto 9       # MEM1 line 23
10) goto 3      # MEM1 line 20
11) goto 10     # MEM1 line 21 (wait for delimiter)
    # initialization routine
12) ld_const (3,17) # UDP
13) ld_const (5,44) # Fragment
14) ld_const (7,44) # Fragment
15) ld_const (9,43) # Route
16) ld_const (11,60); goto 11 # Destination Option

```

Here, a 'goto' offset is used to extend the initialization block in order to load all needed Constant Registers. Since a 'ld_const' instruction cannot be interrupted by the Delimiter it is ensured that the initialization block will be finished before the first packet is accepted.

6 Hardware Architecture

6.1 Functional Blocks

The parser can be partitioned into functional blocks as shown in Figure 10. These blocks are described in the following sections.

6.1.1 Instruction Registers

In the reset state the multiplexers at the Instruction Registers (IRs) choose initial values that result in no instructions being executed and next PCs of '0' being calculated. Otherwise the memory outputs are passed through.

6.1.2 Protocol Data Masking

The block labeled MASKING is responsible for cutting slices out of the input data as defined by OP1 and OP2 of the instruction words. Since there may be two instructions, one from memory 1 and the other from memory 2, requiring different slices to be cut out at the same time this block provides the functionality twice. The slices appear right aligned at the outputs and can be applied to the Accumulator, to the Counter register, or to the Extracted Data parser output.

6.1.3 Next PC Calculation

For the calculation of the next PCs the header parser takes quite a few options into account:

- For the normal progress the PCs are incremented by one every cycle.
- The Counter is compared to 1 or 0 and, if the according bit is set in memory 1, the result determines whether the PCs are once more incremented by one in order to skip the next instruction.
- The 'goto' offsets in both memories are added to the PCs.
- The Accumulator is compared to the Constant Registers and, depending on the occurrence of an INIT_CASE instruction and its `num_cases` parameter, an offset is added to the PCs.
- All these changes apply to PC2 only if the Parallel flag is set. Otherwise it remains unaltered.
- If the Delimiter signals the start of a new header then both PCs are set to address 1.

The need for this many styles of branching becomes obvious when regarding the variety of transitions in Figure 7. However, it makes PC calculation very complex. In Figure 10 the operations have been split up into 3 components.

In 'PC + GOTO' the old PC value is incremented and the 'goto' offset is added. In 'CASE_INSTRUCTION' the comparison between Accumulator and Constant Registers determines the case offset, taking into account whether an INIT_CASE instruction occurred at all and making sure that only a 'num_cases' number of cases is considered.

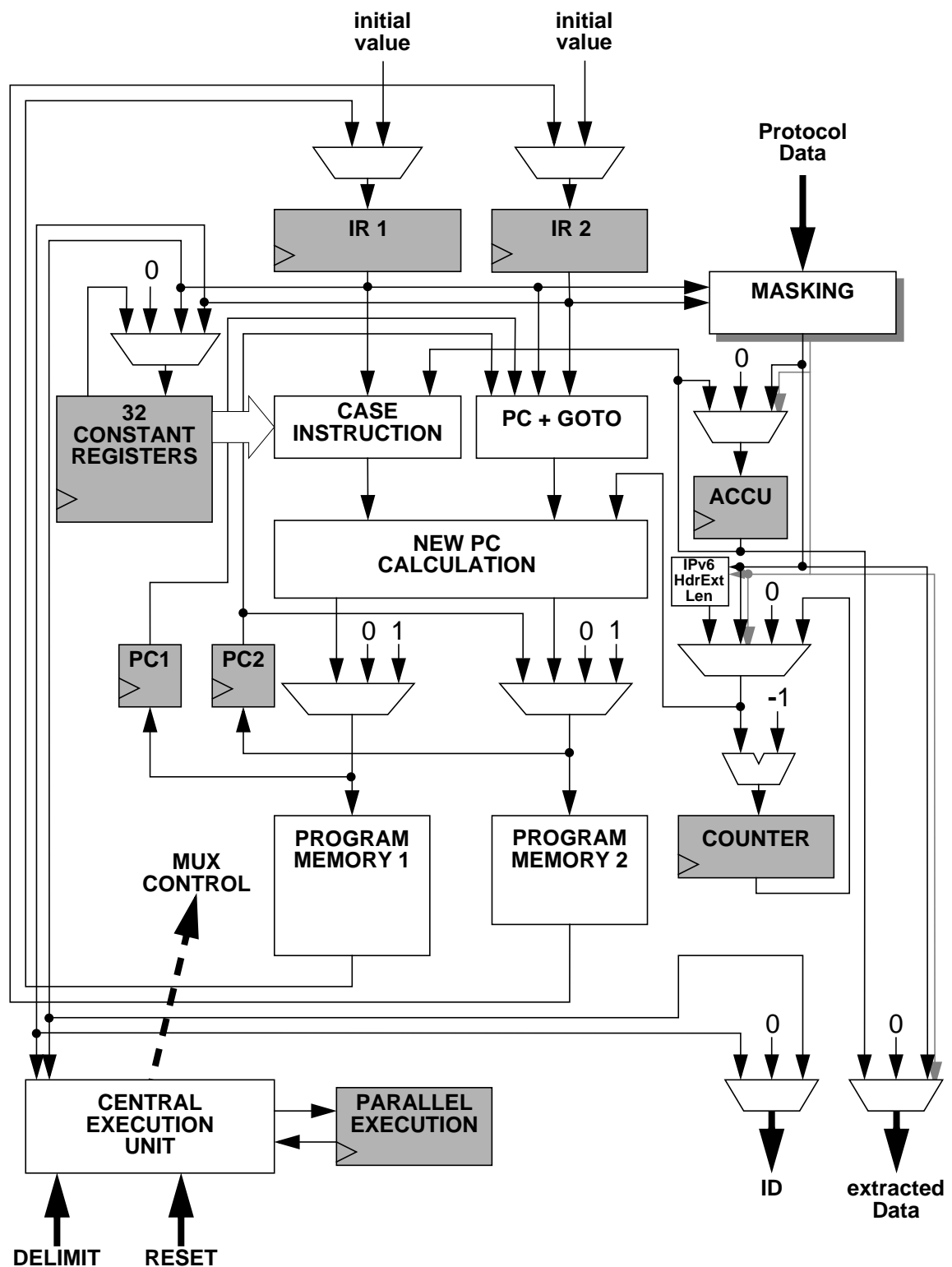


Figure 10 Parser components.

The results of these two blocks are then fed into 'NEW PC CALCULATION' where they are summed up. Also, the Counter is compared to bit 24 of instruction word 1. If the 'test Counter' bit is set and the comparison result is TRUE, then the PCs are again decremented by one to skip one instruction.

The multiplexers following the ‘NEW PC CALCULATION’ finally decide about the new PCs. The choices are:

- ‘0’ in the reset state.
- ‘1’ if the Delimiter is set.
- The calculation result.
- For PC2 the old value may be maintained, depending on the Parallel flag.

6.1.4 Constant Registers

The multiplexer at the Constant Registers block is exemplary for the input to every single Constant Register. In case of a reset they are initialized to ‘0’. If a LD_CONST instruction is issued the multiplexer at the respective Constant Register forwards the new value. Otherwise the previous content is passed through.

6.1.5 Accumulator

If the Accumulator is specified as the target of a WRITE_REG instruction it is loaded with the lower 8 bits of the according MASKING output. If a reset occurs it is set to ‘0’. Otherwise it retains its old value.

6.1.6 IPv6 HdrExtLen

In the HdrExtLen field of an IPv6 Extension Header the length of the header is given in 8-octet units not including the first 8 octets. However, the Counter can only be used to count down in 4-octet units because this is the number of header octets that pass through the parser every cycle.

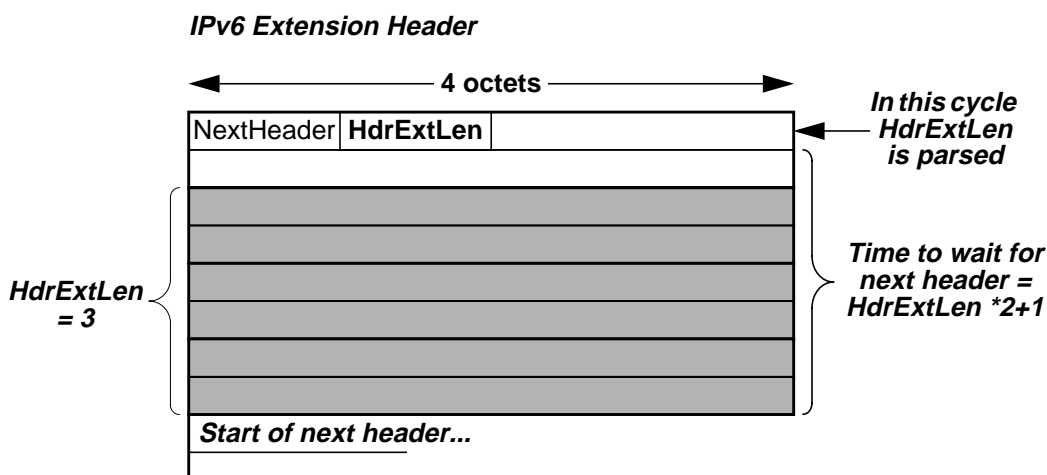


Figure 11 HdrExtLen to Counter conversion.

The ‘IPv6 HdrExtLen’ block performs the conversion from one format to the other by multiplying the header field with 2. Then the excluded first 8 octets need to be considered. Four of these eight octets are actually processed when the HdrExtLen field is parsed because they represent the word that contains the HdrExtLen field. Thus, only ‘1’ has to be added for the remaining second word of the header (see Figure 11).

6.1.7 Counter

The Counter multiplexer chooses between the lower 8 bits of a MASKING output if the Counter is target of a WRITE_REG instruction, the output of the 'HdrExtLen' block if the IP6_COUNTER instruction occurs, an initial value of '0' for a reset, and a feedback of the previous value. The output of the multiplexer is then decremented by one before being applied to the Counter register.

The Counter value that is used for comparison in the 'if' flag mechanism is branched off directly after the multiplexer. This is mandatory for the third case in the IPv6 code which is presented in Table 8.

Memory 1	Memory 2
19) <code>IPv6_counter[8-15]; if (counter=1)</code>	7) <code>write_reg (accu, NextHdr[0-7]); goto 10</code>
20) <code>init_case(3); goto 11</code>	10) <code>goto 3</code>
21) <code>if (counter=1); goto 20</code>	11) <code>goto 10</code>

Table 8. IPv6 code for headers with HdrExtLen.

In line 19 of memory 1 the length of an Extension Header is calculated from the HdrExtLen field and the result is loaded into the Counter. In the same cycle this new Counter value is compared to '1' by the 'if' statement.

This is necessary for Extension Headers that are only two 32-bit words long. For those headers the HdrExtLen field is '0' and the IP6_COUNTER calculation delivers '1'. Consequently, the Counter comparison in line 19 is TRUE already. Two cycles later the next header will start and thus in the following cycle the INIT_CASE instruction needs to be issued.

In order to realize this situation the Counter needs to be checked in line 19 already. Hence, the 'if' mechanism needs to perform the comparison with the *new* Counter value.

6.1.8 Outputs

When a SEND command occurs, then the according MASKING bus is applied to the 'Extracted Data' output. For the SEND_REG instruction the Accumulator is zero-extended to a width of 32 bits and forwarded to the 'Extracted Data' output. In both cases the 'id' parameter of the SEND instructions is sent out the ID port.

If no SEND command is executed then both outputs are reset to '0'.

6.1.9 Central Execution Unit

This component interprets the instruction words and controls the multiplexers accordingly. It is also responsible for setting and considering the Parallel flag.

6.2 Register Scheduling

In memory programmable systems like the header parser the Program Counter determines the instruction that is fetched from memory and the executed instruction affects the calculation of the next PC. Hence, they form a feedback circuit. In a clocked environment no logic loops are allowed. This leads to the important question where to introduce registers that synchronize the loop.

Obviously, there are two reasonable alternatives: The PC can be captured in a register before addressing the memory, or the memory output can be captured in an Instruction Register before being interpreted and executed. As Figure 12 shows the choice determines whether memory access is the first action in every clock cycle, or the last.

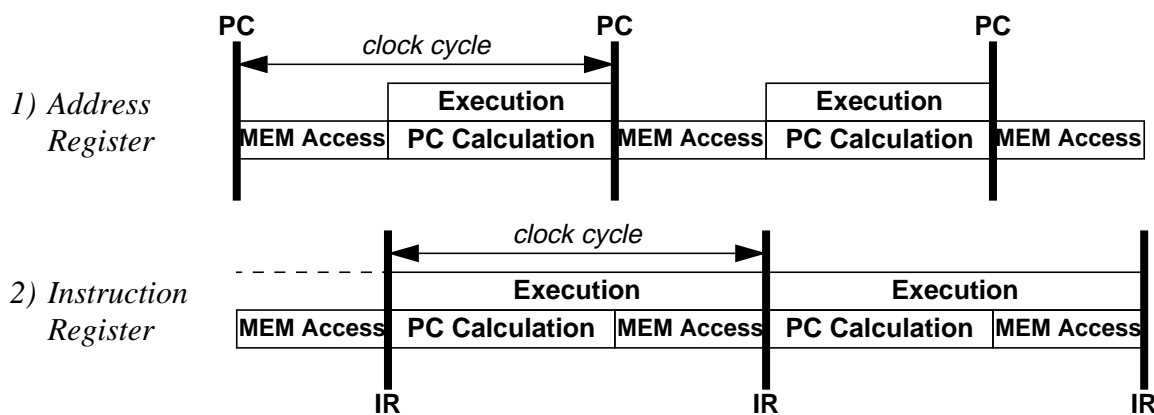


Figure 12 Register scheduling.

For the calculation of the next PC scheduling does not make any difference because it is dependent on the instruction word, e.g. through the 'goto' offset, and needs to be finished before the next memory access because it delivers the memory address that actually should be accessed. This leaves only the cycle time minus the memory access time for PC calculation, independent of whether an IR or a PC register is introduced.

However, the scheduling choice has a major impact on the duration of the execution phase. The results of an instruction execution are expected to be available by the end of a clock cycle. Consequently, the execution phase lasts from the availability of the instruction, i.e. the end of a memory access, to the subsequent clock edge. Figure 12 shows that with an address register (PC) this leads to an execution phase that is as short as the PC calculation interval, while with an IR instruction execution can additionally exploit the memory access time.

This has been the motivation to choose IRs as synchronization points in the design (Figure 10). However, the PCs need to be saved as well because calculation of the next PC relies on the previous value.

7 Implementation

The concept that was introduced in the previous chapters has been implemented in VHDL as an algorithmic proof of concept and to be synthesized later on.

7.1 IBM ASIC Design Methodology

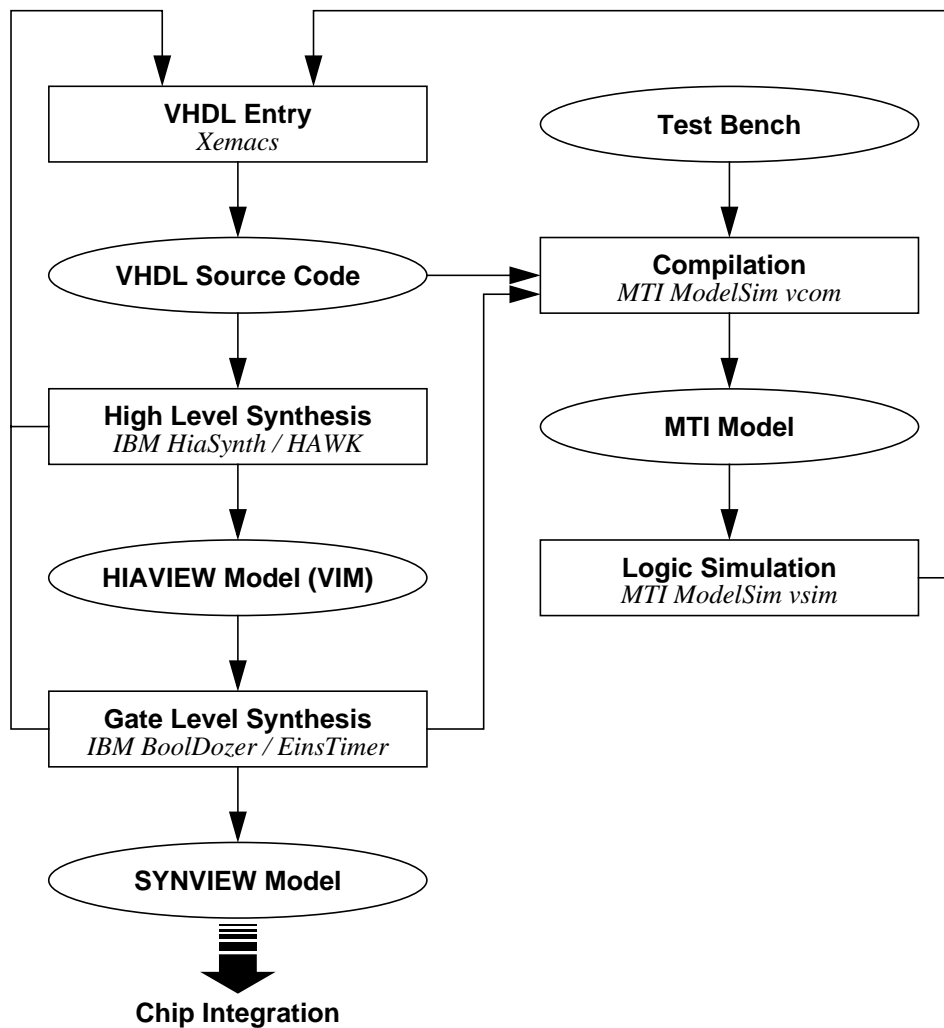


Figure 13 Design methodology flow.

The design process followed the methodology flow shown in Figure 13. After the VHDL Entry the design is verified on the logic level. Once it has passed this test High-Level Synthesis is run. The result can be evaluated using a network viewer which can lead to changes of the VHDL code. Otherwise the model is passed to Gate Level Synthesis. This comprises:

- Inclusion of test structures
- Technology mapping
- Redundancy removal

- Static timing analysis
- Delay optimization

The static timing analysis generates timing reports (see Figure 22) that deliver information about the critical paths. As long as timing requirements are not met the VHDL code is modified in order to improve timing behavior. This results in an iterative design process.

Finally, the gate level model will be exported to VHDL code which is fed back to logic simulation to ensure logic equivalence between the models.

7.2 Design Structure

7.2.1 Partitioning

The top level entity of the design is named ‘parser’. It contains the two program memories and the parser core.

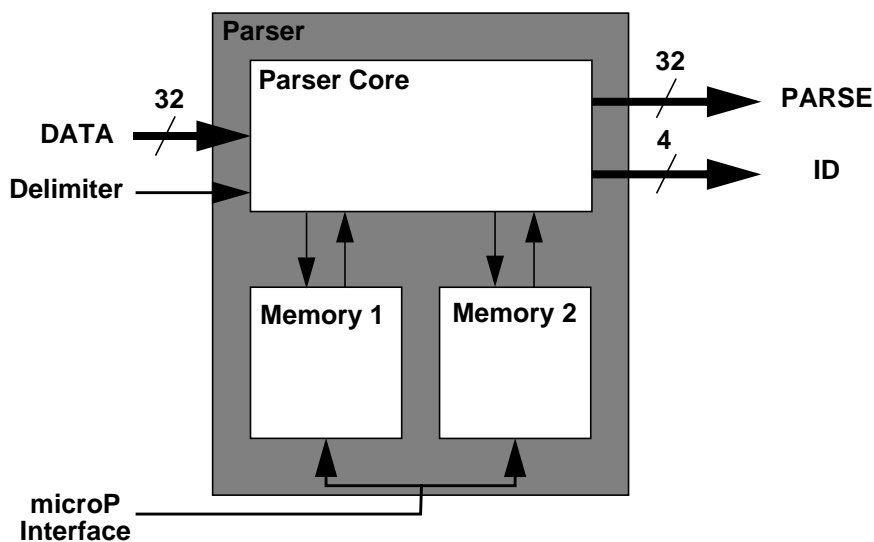


Figure 14 Structure of the top level entity.

7.2.2 Coding Style

In a first implementation the whole functionality offered by the Parser Core has been implemented in a high-level behavioral fashion in a single VHDL process. Instruction execution is performed in a procedure that is called twice -- one call for each program memory, respectively.

In order to make changing parameters of the design as easy as possible many useful definitions are held in a package. This also helps avoiding ‘magic numbers’ in the code. Definitions include:

- Word widths, e.g. for data buses, memories, registers, etc.
- Widths of components that make up an instruction word, e.g. operators, goto offsets, etc.
- Op-codes, i.e. bit patterns that identify a specific instruction.

7.2.3 Program Storage

The two program memories have been realized with IBM Growable Register Arrays (GRAs). These are addressed like RAM blocks but they comprise highly optimized registers for minimum access time. GRAs are available in all kinds of configurations. For this implementation of the header parser a 32 bit wide 64 word model with an access time of about 1.2 ns was used for both models for simplicity and flexibility reasons. For a deployed version the size of the GRAs should of course be optimized, i.e. each GRA should be only as wide as an instruction word and GRA2 may be only half the size of GRA1.

7.3 Simulation

Holding the functionality of the parser core in only one process with no additional concurrent statements resulted in code that is very easy to read. This helped a lot with simulation and debugging.

For the simulation a testbench was written that connects the header parser to a clocking module. It also loads program code from files into the GRAs and applies test protocol data from another file to the parser data input. The resulting parser output is written to a dump file.

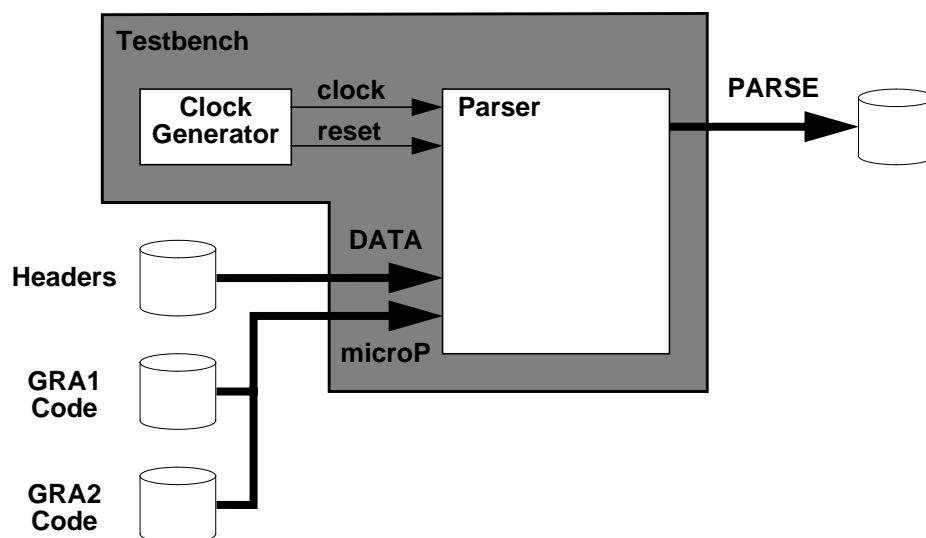


Figure 15 Testbench.

For example this is the GRA1 file for IPv6:

```

IR/op3/op2 /op1 /goto /tvp
1110000000000110011000001 # 0 ld_const (0,6); goto 25; parallel # TCP
0010001001001111100000000 # 1 send (TrafficClass, FlowLabel[4-31], 1)
0100001100001011100000000 # 2 write_reg(accu, NextHeader[16-23])
0010010000001111100000000 # 3 send (SrcAddress[0-31], 2)
0010011000001111100000000 # 4 send (SrcAddress[0-31], 3)
0010100000001111100000000 # 5 send (SrcAddress[0-31], 4)
0010101000001111100000000 # 6 send (SrcAddress[0-31], 5)
0010110000001111100000000 # 7 send (DestAddress[0-31], 6)
0010111000001111100000000 # 8 send (DestAddress[0-31], 7)
0011000000001111100000001 # 9 send (DestAddress[0-31], 8); parallel
10100100000000000000000001 # 10 init_case(3); parallel
# 1st case: TCP, UDP
0011010000001111100000001 # 11 send ((SrcPort, DestPort)[0-31], 10); parallel
01110110000000000001011000 # 12 send_reg (accu, 11); goto 24
00000000000000000000000000 # 13
00000000000000000000000000 # 14
# 2nd case: Fragment Header
01000100000011100000001 # 15 write_reg(accu, NextHeader[0-7]); parallel
1010011000000000111010001 # 16 init_case(3); goto 11; parallel
00000000000000000000000000 # 17
00000000000000000000000000 # 18
# 3rd case: Headers with HdrExtLen field
1000000010000111100000011 # 19 IP6_counter[8-15]; if (counter=1); parallel
1010011000000000110110001 # 20 init_case(3); goto 11; parallel
000000000000000011110111 # 21 if (counter=1); goto 20; parallel
00000000000000000000000000 # 22
# default case
01110110000000000000000000 # 23 send_reg (accu, 11)
0000000000000000111111000 # 24 goto 24 (wait for delimiter)
# init routine
11100010000011000000001 # 25 ld_const (2,6); parallel # TCP
1110001000010110000000001 # 26 ld_const (4,44); parallel # Fragment
1110001100010110000000001 # 27 ld_const (6,44); parallel # Fragment
1110010000000000000000001 # 28 ld_const (8,0); parallel # Hop-by-Hop
11100101000110011111010001 # 29 ld_const (10,51); parallel; goto 24 # Authentication

```

The following listing is the representation of two IPv4 headers in a file. The fields that should be parsed are filled with zeros and bounded with one or two '1'. Other relevant fields hold meaningful values. The rest of the headers is stuffed with patterns that would disturb the otherwise orderly layout of the parser output file. Thus, errors in the parsing process are easily recognized.

```

# Pipe data / Delimiter bit
10101010101010101010101010101010 1 # restart parsing
Ver/IHL/ToS /Total Length /
01000101100000011100110011001101 0 # v4, IHL 5
Fragment fields
11100011100011100011100011100011 0
TTL /Proto /HdrChkSum /
10101011000001101100110011001101 0 # Proto TCP
Source Address /
10000000000000000000000000000001 0
Destination Address /
11000000000000000000000000000011 0
L4 Source Port /Dest Port /
10000000000000011100000000000011 0
01010101010101010101010101010101 1

```

```
##### 2. packet #####
Ver/IHL/ToS /Total Length /
01000111100000011010101010101011 0 # v4, IHL 7
Fragment fields
10101010101010101010101010101011 0
TTL /Proto /HdrChkSum /
11100011000001101101001011010011 0 # Proto TCP
Source Address /
10000000000000000000000000000001 0
Destination Address /
11000000000000000000000000000011 0
10101010101010101010101010101011 0 # dummy options
11010101010101010101010101010101 0
L4 Source Port /Dest Port /
10000000000000011100000000000011 0
```

This header file would result in the following parser output file:

```
64.0 ns PARSE: 00000000000000000000000000000000, ID: 0000
67.2 ns PARSE: 00000000000000000000000000000001, ID: 0001 # ToS
70.4 ns PARSE: 00000000000000000000000000000000, ID: 0000
73.6 ns PARSE: 00000000000000000000000000000110, ID: 0010 # TCP
76.8 ns PARSE: 10000000000000000000000000000001, ID: 0011 # Source Address
80.0 ns PARSE: 11000000000000000000000000000011, ID: 0100 # Destination Address
83.2 ns PARSE: 10000000000000011100000000000011, ID: 0101 # Source / Destination Ports
86.4 ns PARSE: 00000000000000000000000000000000, ID: 0000
89.6 ns PARSE: 00000000000000000000000000000100, ID: 0001 # ToS
92.8 ns PARSE: 00000000000000000000000000000000, ID: 0000
96.0 ns PARSE: 00000000000000000000000000000110, ID: 0010 # TCP
99.2 ns PARSE: 10000000000000000000000000000001, ID: 0011 # Source Address
102.4 ns PARSE: 11000000000000000000000000000011, ID: 0100 # Destination Address
105.6 ns PARSE: 00000000000000000000000000000000, ID: 0000
108.8 ns PARSE: 00000000000000000000000000000000, ID: 0000
112.0 ns PARSE: 10000000000000011100000000000011, ID: 0101 # Source / Destination Ports
```

Figure 16 shows a simulation wave chart with a reset phase (until ~195ns), loading of the ‘const_regs’ (~225ns), and two complete incoming IPv6 packets (~260ns to ~385ns).

7.4 Synthesis

While the high-level coding style was very useful at the simulation stage, synthesis results were unsatisfactory. The performed optimizations were far from optimal which was partly due to inherent semantics that could not be derived from the code, i.e. the VHDL code covered a lot of situations that would never occur.

Trying to add these semantics to the code and this way write VHDL code that would be synthesized the intended way proved to be very difficult, time consuming, and still did not in every case lead to the desired results. Also, register inference was hard to control.

Another big problem with sequential coding is that it tends to use mostly variables as opposed to signals. Synthesis and timing optimization keep a lot of signal names and can be forced to do so by setting attributes, but lose most variable names in the process. This leads to output that is very difficult to comprehend since cryptic wire names do not carry any information about the wire’s purpose.

This motivated a complete re-engineering of the parser core VHDL code. This time a structure close to Figure 10 has been adopted with components being implemented as separate unlocked

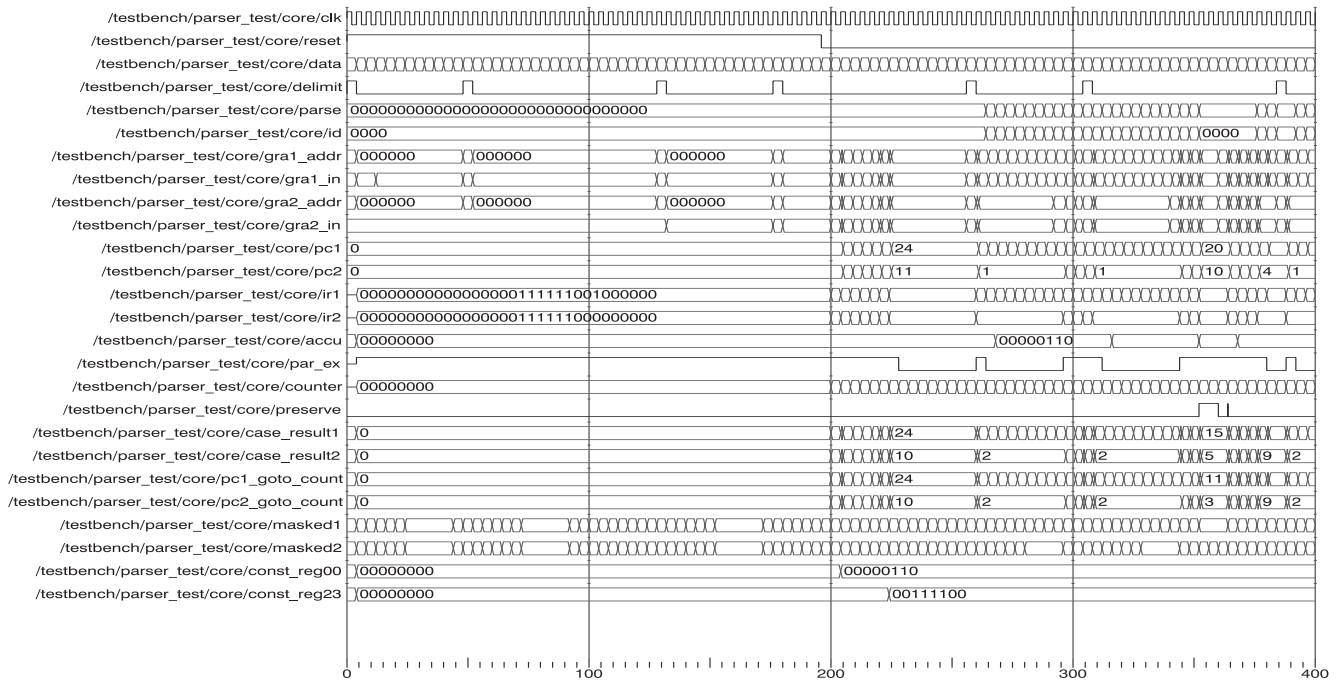


Figure 16 Simulation wave chart.

processes. The only clocked process represents the registers. This way register inference can be controlled precisely. Information is transmitted between the components using signals which are preserved through synthesis and timing optimization and thus can be easily observed.

7.5 Timing Optimization

From Figure 10 it is obvious that the critical path in the header parser is the PC calculation. The first timing analysis resulted in a negative slack of several cycle times. At this time the offsets imposed on the PC from the numerous branching options were added in a sequential fashion. By computing intermediary results in parallel, by moving complex operations such as additions out of the critical path, and with additional fine tuning the negative slack could be reduced to less than 2 ns.

Figure 17 shows the resulting data path for PC2. In the left part of the graph the `case` number is calculated. Since in GRA2 two lines of code are allowed for each case, the case number needs to be multiplied by two. This is realized by a simple shift operation.

In the middle the `goto` offset is added to the previous PC. Then two alternative values are added: '1' for the normal PC incrementation and '2' for an `if` command with a TRUE result. One of these alternatives is chosen by the circuit on the right where the `if` command is evaluated. What makes this path critical is the fact that not necessarily the Counter value that is stored in the register is used for the comparison. It might as well be a new Counter value coming from the 'Masking' block if an instruction loads the Counter in the same clock cycle.

In the lower part of the graph both branches are summed up. Finally, a multiplexer switches between a '0' for a reset, a '1' for a Delimiter, the old PC if the Parallel bit is no set, and else the newly computed PC.

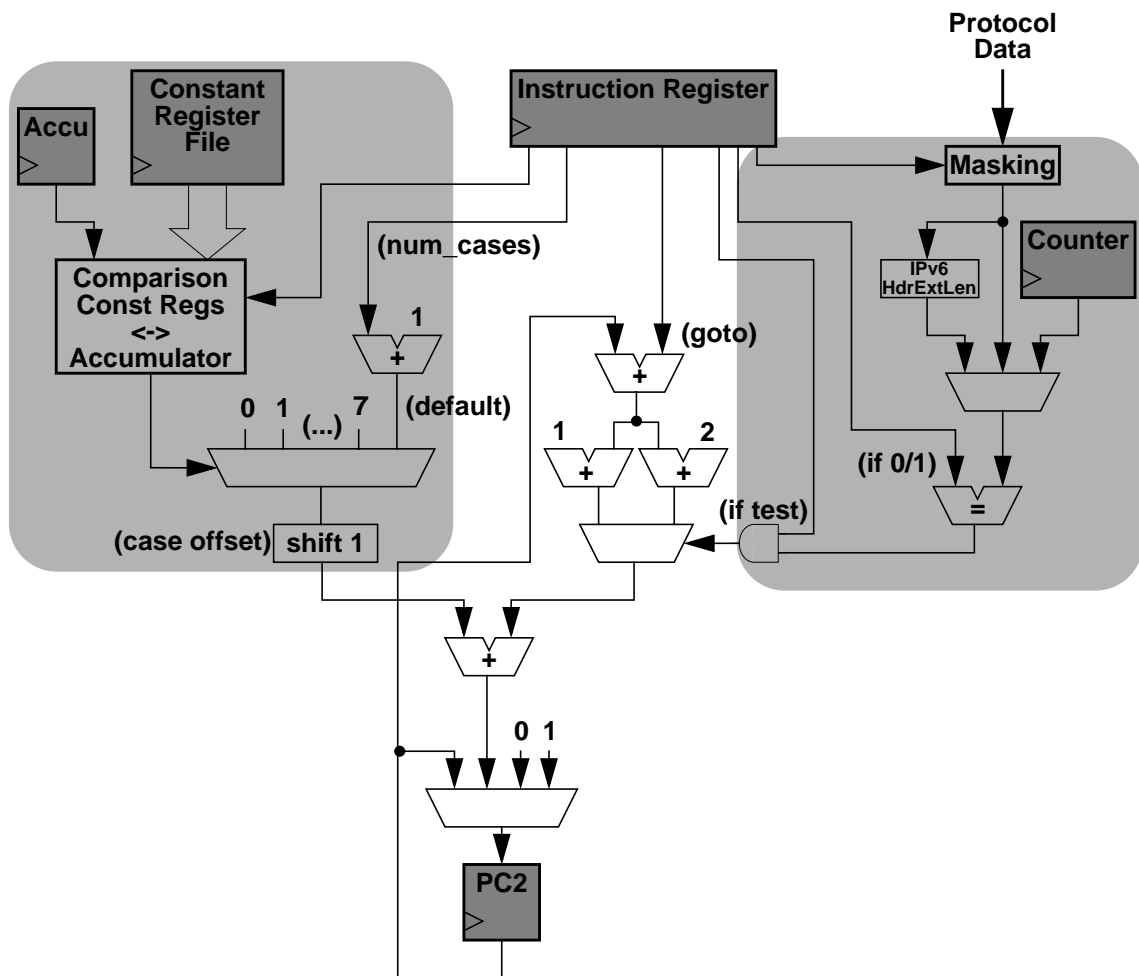


Figure 17 Optimized PC2 calculation.

Further moving of components, e.g. the last adder into one of the branches, only relocated the problem. Thus, other ways to improve the timing had to be found.

7.6 Scaling Options

With a negative slack of much less than one clock cycle the timing requirements would be already met if only a way could be found to double the data throughput through the header parser.

7.6.1 Data Bus Width

One attempt to double data throughput is doubling the data bus width to 64 bit. Unfortunately, both IP versions as well as many other protocols, e.g. MPLS, are defined in a 32-bit aligned fashion. This entails three implications:

1. Packets may consist of an odd number of 32-bit words. Consequently, each of the fields that the parser extracts can be located at two possible positions -- in the lower or in the upper half of a 64-bit word. Either a synchronization mechanism has to be introduced, or programming the parser becomes very complex.

2. Information from the first 32 bits might be decisive for the handling of the second 32-bit word. This means that consideration of the second word cannot start before interpretation of the first word is finished. This voids the advantage of handling twice the number of bits per clock cycle.
3. For a doubled number of bits two instructions in parallel may not be sufficient to perform all necessary operations. Again, this adds complexity.

Because of these issues, this approach was further pursued.

7.6.2 Pipelining

In modern processors the available execution time per instruction is extended by using registers to divide the path through the execution units into subdivisions which are called 'pipeline stages'. A pipelined GPP architecture might for example have the following stages:

1. Instruction Fetch (IF)
Load an instruction from memory.
2. Instruction Decode / Register Fetch (ID)
Interpret the instruction operators and read from the register file accordingly.
3. Execution (EX)
Use the Arithmetic Logic Unit (ALU) to execute the instruction.
4. Memory Access (MEM)
Write result of execution to memory or read data from memory.
5. Write Back (WB)
Write data to register file.

With five pipeline stages it takes five clock cycles to execute one instruction. A pipeline stage that is not in use by an instruction can already be occupied by the next instruction, i.e. when an instruction advances from IF to ID, then the next instruction can enter the IF stage and so on. Thus, there can be up to five instructions in the pipeline at one time with no infrastructure being replicated. Under optimal conditions this results in one instruction being executed per cycle while allowing five cycles of execution time per instruction.

This idea superimposed on the header parser would mean to pipeline the mutual dependency of PC calculation and memory access since this is the critical path. For example the PC registers can be moved in between the PC calculation and the memory. Then there would be one cycle reserved for memory access (IF) and a full cycle for instruction execution and PC calculation (EX). The overlapping of the stages is shown in Figure 18a.

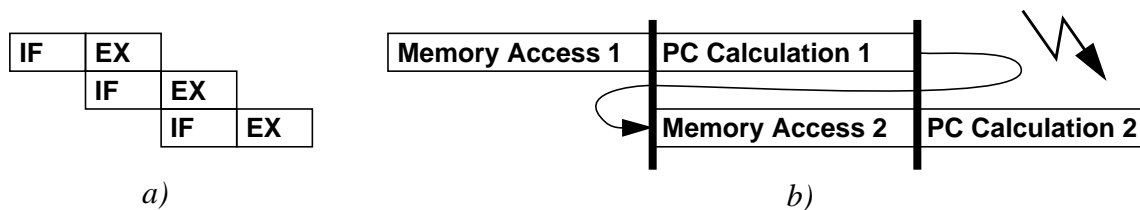


Figure 18 Pipelining the header parser.

The major problem with this configuration is the fact that every memory access is dependent on the previous PC calculation. But now the PC is calculated in parallel to the memory access that it is supposed to control. As Figure 18b shows, *PC calculation 1* delivers the address for *Memory access 2* but the result is only available when the memory access should be finished already.

For GPPs this problem is solved by **branch prediction**. The new PC is speculated before it is determined and the according memory word is loaded into the IR. When the PC calculation is finished it is compared to the speculated PC. If they match the program flow continues. If the speculation was wrong then the IR is rejected and the memory access is repeated, this time with the correct address. Consequently, if the speculation was wrong then a cycle has been wasted.

For two reasons this concept cannot be adapted for the header parser. First of all, a GPP has only two possible new PCs. Normally the PC is simply incremented by one and the program flow continues at the next memory address. This is also the case if the actual instruction is a branch that is not taken. If the branch is taken, then the branch target is the next PC. However, the *case* instruction of the header parser has nine possible branch targets, and combined with the *if* command this has to be multiplied by 2, resulting in 18 possible new PCs. This leaves only a small chance for a correct prediction.

The second issue is that this mechanism leads to a variable program execution time. For every wrong prediction one clock cycle is added to the execution time which is in that statistical. This is not compatible with the requirements for a real-time system like the header parser. The data that the parser works on are not accessed on demand but they come in a constant stream. Thus, the instruction order in the parser programs is tuned such, that the right instruction is executed for the data that is available at a certain time. This tuning is only possible if execution time is strictly deterministic with no statistical events being involved.

For example, when handling IPv6 Extension Headers the first instruction of the routine for the occurring header must be executed in the cycle directly following the *case* instruction. Otherwise the first word of the Extension Header is gone and the instruction that e.g. extracts the Next Header identifier is executed too late.

Pipelining is only applicable to systems where the goal is a high average throughput, but not for a determined execution time in real-time systems.

7.6.3 Parallel Entities

For GPPs a popular way to double instruction throughput is to integrate two or more execution paths in one processor, thus being able to issue multiple instructions per clock cycle. Apparently, this scales very well. For the header parser this should scale even better since the content of the GRAs is the same for all the entities and thus it might be possible to still have only one entity of each GRA.

It would not be adequate to send data words belonging to the same header to different parser entities because information in one word decides about the interpretation and handling of another. Information would have to be exchanged between entities and this would make hardware and software very complex.

The alternative would be to distribute incoming data packet-wise to multiple parser entities, i.e. packet 1 goes to parser 1, packet 2 goes to parser 2, and so on. This configuration is shown in Figure 19. With the instantiation of three parser entities the load on each of the entities is re-

duced to the half and one would expect to triple the time available for the execution of an instruction.

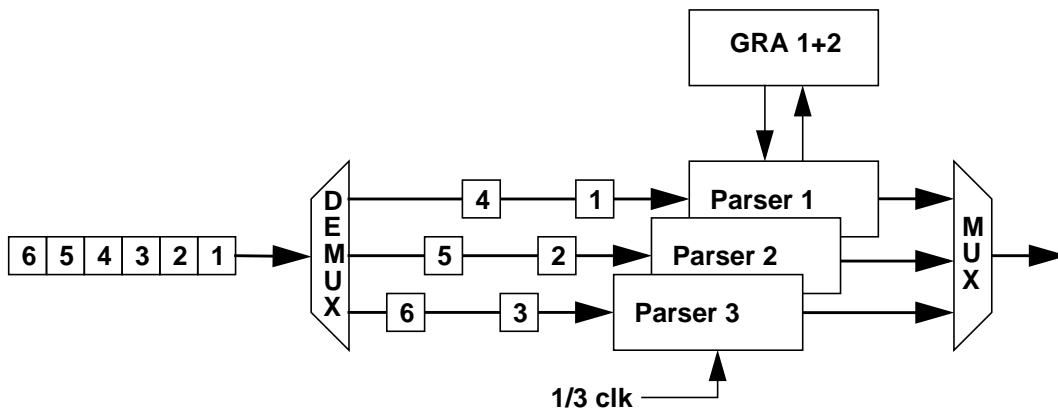


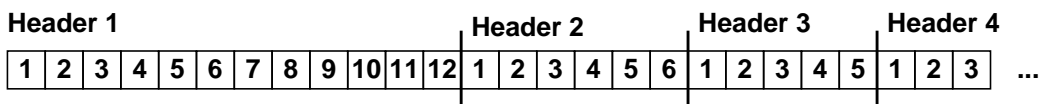
Figure 19 Parallel parser entities.

Unfortunately, this does only work if all headers are of equal length. If header lengths differ then a long header might block a parser entity for a long period while shorter headers keep coming in. Obviously, this problem occurs mainly under worst case conditions with packets that do not have any payload and thus come in back-to-back.

The minimum IPv4 packet is just a header with no options. Measured in 32-bit words this packet's length would be 5. On the other hand, due to the range of the IHL field an IPv4 header can have a maximum length of 15. Adding the first word of a TCP or UDP header where the port numbers are stored results in a maximum header length of 16. Payload length may be between 0 and 16384-IHL.

Figure 20 shows the worst case situation for IPv4 where four header parsers are occupied in parallel even though processing time per header word is only doubled compared to the data clock.

Packet Arrival:



Parser Scheduling:

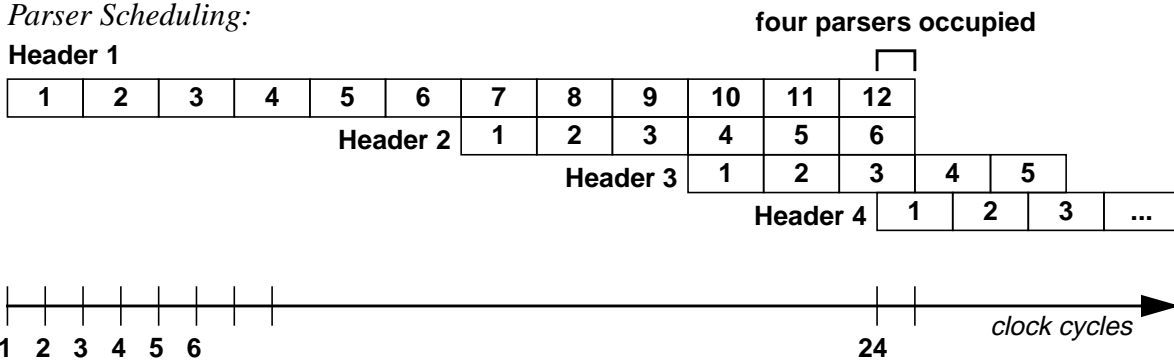


Figure 20 Worst case scheduling for IPv4.

IPv6 headers have a minimum length of 10 words and can be almost arbitrarily long since the upper bound is the maximum payload length of 64 Kilobyte. With no maximum header length

an arbitrary number of processors can be blocked at the same time. For a given number of parser entities similar considerations as above lead to maximum header lengths that can be parsed even in worst case situations. With four entities as needed for IPv4 the maximum IPv6 header length is 43. If both IP versions are mixed, i.e. minimum header length 5 and arbitrary maximum, with 4 entities the maximum length amounts to 23.

The generic formula delivering the necessary number of entities (P) for given minimum (HL_{min}) and maximum (HL_{max}) header length is:

$$P = \left\lceil \lg \frac{HL_{max} + 1}{HL_{min} + 1} \right\rceil + 2$$

Resolved for HL_{max} the formula changes to:

$$HL_{max} = (HL_{min} + 1) \cdot 2^{(P-2)} - 1$$

This has been found to be no reasonable scaling ratio. Thus, this approach was dropped, too. Still it should be noted that these are worst case considerations that are not necessarily very likely to occur.

7.6.4 Modifying the Internal Architecture

After all other approaches to meet the timing requirements by scaling have been disapproved it is clear that the functionality of the design has to be modified.

It was found that all protocol fields are aligned to 4-bit (nibble) boundaries. This was exploited by reducing the Masking parameterization from bit-wise to nibble-wise, this way reducing the levels of logic in the block. This improved the slack to -1.7 ns.

Next the reason for not always using the Counter register for the 'if' comparison was examined. Bypassing the register is necessary for the reasons explained in section 6.1.7. The critical code segment is repeated here in Table 9.

Memory 1	Memory 2
19) IPv6_counter[8-15]; if (counter=1)	7) write_reg (accu, NextHdr[0-7]); goto 10
20) init_case(3); goto 11	10) goto 3
21) if (counter=1); goto 20	11) goto 10

Table 9. IPv6 code for headers with HdrExtLen.

As Figure 21 shows, in the first clock cycle of the considered code the HdrExtLen and Next Header fields are available and need to be extracted. The resulting new Counter value needs to be examined as to find out if the Extension Header is only two words long. If so then a case

command has to be performed with the new NextHeader value to determine which routine to start next. All this must be done within 3/2 clock cycles before the end of the PC calculation part of the second cycle.

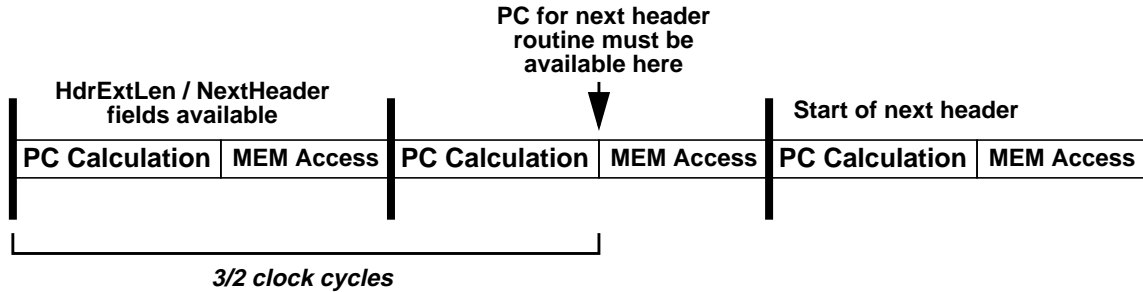


Figure 21 Timing for HdrExtLen routine.

The solution that was found to this problem was to postpone the Counter comparison to the second cycle in which also the case instruction is executed. Case offset and comparison result are computed in parallel and the case offset is only added to the PCs if the comparison result is TRUE. Since the Counter is loaded with its new value in the first cycle but the comparison is now in the second cycle, the Counter register can be used for the comparison and the ‘Masking’ block is no longer in the critical path.

This was implemented by using two bits from OP2 of the case command similar to the if mechanism: one bit indicates a conditional case and the other determines whether to test the Counter for ‘0’ or ‘1’. If the condition is activated and the Counter register does not match the second flag then a ‘preserve’ flag is set. This flag keeps the PCs and the Parallel register from being changed. Thus, the modified case command performs a loop waiting for the Counter to reach a specified value and then branches to one out of nine targets, and all this in a single line of code.

The IPv6 code segment needs to be adapted as given in Table 10.

Memory 1	Memory 2
19) IPv6_counter[8-15]	7) write_reg (accu, NextHdr[0-7]); goto 3
20) init_case(3, counter=0); goto 11	10) -/-
21) -/-	11) -/-

Table 10. Modified IPv6 code for headers with HdrExtLen.

With this modification the timing analysis reported a reduced negative slack of 1.14 ns. It occurs between Accumulator and the Instruction Registers, i.e. in the left part of Figure 17 where Accumulator and Constant Registers are compared.

Figure 22 shows the corresponding part of the timing report. The report is read bottom-up, i.e. the ‘ACCU’ register at the bottom is the start of the critical path, it passes through ‘CASE_RESULT2’ and ‘GRA2’, and ends in ‘IR2’.

The abbreviated column titles have the following meanings:

Num/ Test	PinName	LimitedAT/						Delay/ Failed Test/	
		E Phase	AT	Slack	Slew	CL	FO Cell	P Func	T.Adj NetName
1	CORE_REGISTERS_IR2#2(4)/D	R B1+R	4.01	-1.14	0.08	0.02	1 LPH0001_E	E PHSRL	0.28 NET241
	CORE_REGISTERS_IR2#2(4)/C	F C1+	3.15		0.50	8.27 385 LPH0001_E	E	E	0.00 C_CLK
---->{a}	XPNDAO473P0/Z	R B1+R	4.01	-1.14	0.08	0.02	1 AND2_G_H	H AND	0.00 NET241
---->	XPNDAO473P0/A	R B1+R	3.91	-1.14	0.10	0.03	1 AND2_G_H	H AND	0.10 GRA2_DATA(4)
---->{b}	GRA2_RA/DO27RP1	R B1+R	3.91	-1.14	0.10	0.03	1 RA064X32D2P2W1R1M1	A GRA	0.00 GRA2_DATA(4)
---->{c}	GRA2_RA/RowAccRP1	F B1+R	2.70	-1.14	0.17	0.00	0 RA064X32D2P2W1R1M1	A GRA	1.20
---->	GRA2_RA/A05RP1	F B1+R	2.70	-1.14	0.17	0.06	2 RA064X32D2P2W1R1M1	A GRA	0.00 GRA2_ADDR(0)
---->{d}	XPNDAO543P0/Z	F B1+R	2.70	-1.14	0.17	0.06	2 OAI22_H	H OR*AI	0.00 GRA2_ADDR(0)
---->	XPNDAO543P0/A1	R B1+R	2.58	-1.14	0.23	0.03	1 OAI22_H	H OR*AI	0.12 NET15150
---->{e}	BOX9286/Z	R B1+R	2.58	-1.14	0.23	0.03	1 XOR2_E	E XOR	0.00 NET15150
---->	BOX9286/B	R B1+R	2.42	-1.14	0.09	0.02	1 XOR2_E	E XOR	0.16 NET24052
---->{f}	XPNDAO868P1/Z	R B1+R	2.42	-1.14	0.09	0.02	1 NAND2_H	H NAND	0.00 NET24052
---->	XPNDAO868P1/A	F B1+R	2.35	-1.14	0.11	0.06	2 NAND2_H	H NAND	0.07 NET2162
---->{g}	XPNDAO871P1/COUT	F B1+R	2.35	-1.14	0.11	0.06	2 ADDF_H	H ADDER	0.00 NET2162
---->	XPNDAO871P1/CIN	F B1+R	2.12	-1.14	0.12	0.03	1 ADDF_H	H ADDER	0.23 NET2161
---->{h}	XPNDAO874P1/Z	F B1+R	2.12	-1.14	0.12	0.03	1 AO222_H	H A*OR	0.00 NET2161
---->	XPNDAO874P1/A1	F B1+R	1.96	-1.14	0.11	0.04	2 AO222_H	H A*OR	0.16 NET2160
---->{i}	XPNDAO879P0/COUT	F B1+R	1.96	-1.14	0.11	0.04	2 ADDF_E	E ADDER	0.00 NET2160
---->	XPNDAO879P0/CIN	F B1+R	1.76	-1.14	0.12	0.10	3 ADDF_E	E ADDER	0.20 CORE_CASE_RESULT2(3)
---->{j}	BOX21526/Z	F B1+R	1.75	-1.14	0.12	0.10	3 NAND3_L	L NAND	0.00 CORE_CASE_RESULT2(3)
---->	BOX21526/A	R B1+R	1.68	-1.14	0.10	0.08	1 NAND3_L	L NAND	0.08 NET36306
---->{k}	BOX25780/Z	R B1+R	1.67	-1.14	0.10	0.08	1 OR2_J	J OR	0.00 NET36306
---->	BOX25780/A	R B1+R	1.57	-1.14	0.09	0.06	2 OR2_J	J OR	0.11 NET55888
---->{l}	TBMOVE/Z	R B1+R	1.57	-1.14	0.09	0.06	2 NAND2_L	L NAND	0.00 NET55888
---->	TBMOVE/A	F B1+R	1.50	-1.14	0.13	0.10	2 NAND2_L	L NAND	0.06 NET19916
---->{m}	BOX20785/Z	F B1+R	1.50	-1.14	0.13	0.10	2 NAND3_L	L NAND	0.00 NET19916
---->	BOX20785/A	R B1+R	1.42	-1.14	0.12	0.08	1 NAND3_L	L NAND	0.08 NET55897
---->{n}	TORD236721/Z	R B1+R	1.42	-1.14	0.12	0.08	1 AND3_J	J AND	0.00 NET55897
---->	TORD236721/C	R B1+R	1.23	-1.14	0.27	0.10	3 AND3_J	J AND	0.19 NET19833
---->{o}	BOX17224/Z	R B1+R	1.23	-1.14	0.27	0.10	3 NOR3_L	L NOR	0.00 NET19833
---->	BOX17224/B	F B1+R	1.06	-1.14	0.10	0.06	1 NOR3_L	L NOR	0.16 NET43244
---->{p}	BOX34639/Z	F B1+R	1.06	-1.14	0.10	0.06	1 OA21_H	H OR*A	0.00 NET43244
---->	BOX34639/A2	F B1+R	0.86	-1.14	0.06	0.05	2 OA21_H	H OR*A	0.21 NET23997
---->{q}	BOX20839/Z	F B1+R	0.85	-1.14	0.06	0.05	2 AND4_J	J AND	0.00 NET23997
---->	BOX20839/B	F B1+R	0.72	-1.14	0.12	0.04	1 AND4_J	J AND	0.14 NET22437
---->{r}	XPNDXORS19640/Z	F B1+R	0.72	-1.14	0.12	0.04	1 COMP2_H	H COMP	0.00 NET22437
---->	XPNDXORS19640/B1	R B1+R	0.48	-1.14	0.09	0.19	8 COMP2_H	H COMP	0.24 CORE_ACCU&1(0)
---->	FANTOM23690/Z	R B1+R	0.47	-1.14	0.09	0.19	8 CLK_R	R DRVR	0.01 CORE_ACCU&1(0)
---->	FANTOM23690/A	R B1+R	0.32	-1.14	0.23	0.25	5 CLK_R	R DRVR	0.15 CORE_ACCU(0)
---->	CORE_REGISTERS_ACCU(0)/L2	R B1+R	0.31	-1.14	0.23	0.25	5 LMX0001_J	J PHSRL	0.01 CORE_ACCU(0)
---->	CORE_REGISTERS_ACCU(0)/B	R B1+	0.00	N/C	0.50	8.14 389	LMX0001_J	J PHSRL	0.31 CLK&0

Figure 22 Worst slack report.

- E - edge: rising or falling
- AT - arrival time
- CL - net capacitance
- FO - fan out
- P - power level
- Func - logical function

Further investigation will show how the slack can be increased to eventually become positive. Architectural possibilities include:

- Moving the comparison to the previous cycle and hold the result in a 'Case Offset' register that is then used for PC calculation.
- Reducing the number of Constant Registers.

Also, transition from IBM's SA-27 to SA-27E CMOS technology ([IBM]) which will be available soon will result in up to 20% faster logic. With a combination of further architectural improvements and the technology shift it is expected to be able to meet the timing requirements in the end.

8 Conclusion

In this diploma thesis a concept has been developed for a programmable header parser that can be used in a high-speed network processor. Using Internet protocols as an example, which are very complex protocols, an instruction set was specified that enables the user to extract all necessary information from incoming headers. This instruction set is very flexible and enables the parser to handle future extensions of protocol definitions as well as to be configured for a variety of other protocols.

The architecture guarantees that headers are parsed in a deterministic period of time. Even under worst case conditions no packets are dropped. Also, the packet order is preserved.

It has been shown that traditional scaling techniques such as pipelining are not applicable to the field of programmable real-time header parsing. However, a one-cycle many-targets branch instruction ('hardware case') and an efficient memory management strategy for parallel systems have been proposed that help to complete complex tasks under real-time conditions and to minimize chip area, respectively.

Although the final timing requirements have not yet been met, the parser operates at very high speeds already. It is expected that with some implementation improvements and the transition to IBM's 0.11 um CMOS technology the parser will be able to handle traffic at full 10 Gbps.

Some ideas for future work would be:

- Optimize the implementation.
- Make the `IP6_counter` instruction more generic, e.g. using a configurable shifter and adder.
- Extend the concept to parse ATM and PPP in the link layer, both IP versions in the network layer, and several upper layer protocols with one parser entity.
- Investigate the applicability of the found concepts for other components within the NP framework. Work towards a generic building block.

Appendix A: VHDL Source Code

The most interesting code is the functional core of the header parser. The essential source code files are documented in this chapter.

All design parameters have been defined as constants in “`parser_pck.vhdl`”:

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_signed.all;
use IEEE.Std_Logic_Arith.all;

package PARSE_PCK is

-----
-- Constant and Subtype Definitions
-----

    constant PIPE_WIDTH : NATURAL := 32;    -- DATA word width
    constant PARSE_WIDTH : INTEGER := 32;  -- parser output word width
    constant ID_WIDTH : NATURAL := 4;      -- signaling bits towards
        -- 'user' components
    constant REG_WIDTH : NATURAL := 8;     -- register word width
    constant CASE_WIDTH : NATURAL := 3;    -- bits for case number

    constant OPCODE_WIDTH : NATURAL := 3;  -- opcode bits
    constant OP3_WIDTH : NATURAL := 4;    -- bits for third parameter
    constant OP2_WIDTH : NATURAL := 5;    -- bits for second parameter
    constant OP1_WIDTH : NATURAL := 5;    -- bits for first parameter
    constant GOTO1_WIDTH : NATURAL := 6;   -- bits for GRA1 goto offset

    constant GRA1_WIDTH : NATURAL := 32;

-- ideally:
--   OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH + GOTO1_WIDTH + 3;
    constant GRA1_ADDR_WIDTH : NATURAL := 6;
    constant GRA2_WIDTH : NATURAL := 32;
-- ideally:
--   constant GRA2_WIDTH : NATURAL := GRA1_WIDTH - 4; -- no flags, 1 goto bit less
    constant GRA2_ADDR_WIDTH : NATURAL := 6;

    subtype PIPE_WORD is STD_LOGIC_VECTOR (0 to PIPE_WIDTH - 1);
        -- DATA word
    subtype PARSE_WORD is STD_LOGIC_VECTOR (0 to PARSE_WIDTH - 1);
        -- parser output word
    subtype ID_WORD is STD_LOGIC_VECTOR (0 to ID_WIDTH - 1);
        -- signaling word towards 'user' components
    subtype REG_WORD is STD_LOGIC_VECTOR (0 to REG_WIDTH - 1);
    subtype CASE_WORD is STD_LOGIC_VECTOR (0 to CASE_WIDTH - 1);
        -- holds a case number
    subtype GRA1_WORD is STD_LOGIC_VECTOR (0 to GRA1_WIDTH - 1);
    subtype GRA1_ADDR_VEC is STD_LOGIC_VECTOR (0 to GRA1_ADDR_WIDTH - 1);
    subtype GRA2_WORD is STD_LOGIC_VECTOR (0 to GRA2_WIDTH - 1);
    subtype GRA2_ADDR_VEC is STD_LOGIC_VECTOR (0 to GRA2_ADDR_WIDTH - 1);

    subtype OPCODE_VEC is STD_LOGIC_VECTOR (0 to OPCODE_WIDTH - 1);

```

```
-----  
-- Op-Codes  
-----  
  
constant NOP : OPCODE_VEC := "000";  
constant LD_CONST : OPCODE_VEC := "111";  
constant SEND : OPCODE_VEC := "001";  
constant SEND_REG : OPCODE_VEC := "011";  
constant WRITE_REG : OPCODE_VEC := "010";  
constant IP6_COUNTER : OPCODE_VEC := "100";  
constant INIT_CASE : OPCODE_VEC := "101";  
  
end PARSER_PCK;
```

The file “**parser_core.vhdl**” defines the entity for the functional core:

```
library IEEE, IBM;  
use IEEE.Std_Logic_1164.all;  
use IEEE.Std_Logic_Arith.all;  
use IBM.SYNTHESIS_SUPPORT.all;  
use IBM.DFT_SYNTHESIS.all;  
use STD.TEXTIO.all;  
use WORK.PARSER_PCK.all;  
  
entity PARSER_CORE is  
  
    port (CLK : in STD_LOGIC;  
          RESET : in STD_LOGIC;  
          DATA : in PIPE_WORD;  
            -- incoming header data  
          DELIMIT : in STD_LOGIC;  
            -- Packet Delimiter, set on start of next packet  
          PARSE : out PARSE_WORD;  
            -- extracted header fields  
          ID : out ID_WORD;  
            -- identifies PARSE value  
          GRA1_ADDR : out GRA1_ADDR_VEC;  
          GRA1_IN : in GRA1_WORD;  
            -- primary instruction memory  
          GRA2_ADDR : out GRA2_ADDR_VEC;  
          GRA2_IN : in GRA2_WORD);  
            -- secondary instruction memory for parallel commands  
  
    attribute INTERNAL_SCAN_STYLE of PARSER_CORE : entity is "LSSD";  
    attribute PIN_FUNCTION of CLK : signal is "SCAN_B_CLOCK_CHAIN1";  
    attribute PIN_DATA of CLK : signal is "TB_KFLAG=-BC/";  
    attribute LSSDPLUS_CONVERSION of PARSER_CORE : entity is TRUE;  
  
end PARSER_CORE;
```

Finally, “`parser_core_arc.vhd1`” holds the functionality of the design:

architecture BEHAVIOUR of PARSE_CORE is

```

signal PARSE_NEW : PARSE_WORD;
signal ID_NEW : ID_WORD;

-- REGISTERS:
signal PC1 : INTEGER range 0 to 63; -- program counter for GRA1
signal PC1_NEW : INTEGER range 0 to 63;
signal PC2 : INTEGER range 0 to 63; -- program counter for GRA2
signal PC2_NEW : INTEGER range 0 to 63;
-- NB: PCs are only saved for computation of next PCs
--      but are applied to GRAs asynchronously!
signal IR1 : GRA1_WORD; -- 1st instruction register
signal IR2 : GRA2_WORD; -- 2nd instruction register
signal ACCU : REG_WORD;-- general purpose register
signal ACCU_NEW : REG_WORD;
signal PAR_EX : STD_LOGIC;-- flag: execution of parallel command?
signal PAR_EX_NEW : STD_LOGIC;
signal COUNTER : REG_WORD;
-- wrapping; implicitly decremented by 1 every clock cycle
signal COUNTER_NEW : REG_WORD;

-- constant registers
signal CONST_REG00, CONST_REG01, CONST_REG02, CONST_REG03 : REG_WORD;
signal CONST_REG10, CONST_REG11, CONST_REG12, CONST_REG13 : REG_WORD;
signal CONST_REG20, CONST_REG21, CONST_REG22, CONST_REG23 : REG_WORD;
signal CONST_REG30, CONST_REG31, CONST_REG32, CONST_REG33 : REG_WORD;
signal CONST_REG40, CONST_REG41, CONST_REG42, CONST_REG43 : REG_WORD;
signal CONST_REG50, CONST_REG51, CONST_REG52, CONST_REG53 : REG_WORD;
signal CONST_REG60, CONST_REG61, CONST_REG62, CONST_REG63 : REG_WORD;
signal CONST_REG70, CONST_REG71, CONST_REG72, CONST_REG73 : REG_WORD;
signal CONST_REG00_NEW, CONST_REG01_NEW, CONST_REG02_NEW, CONST_REG03_NEW :
REG_WORD;
signal CONST_REG10_NEW, CONST_REG11_NEW, CONST_REG12_NEW, CONST_REG13_NEW :
REG_WORD;
signal CONST_REG20_NEW, CONST_REG21_NEW, CONST_REG22_NEW, CONST_REG23_NEW :
REG_WORD;
signal CONST_REG30_NEW, CONST_REG31_NEW, CONST_REG32_NEW, CONST_REG33_NEW :
REG_WORD;
signal CONST_REG40_NEW, CONST_REG41_NEW, CONST_REG42_NEW, CONST_REG43_NEW :
REG_WORD;
signal CONST_REG50_NEW, CONST_REG51_NEW, CONST_REG52_NEW, CONST_REG53_NEW :
REG_WORD;
signal CONST_REG60_NEW, CONST_REG61_NEW, CONST_REG62_NEW, CONST_REG63_NEW :
REG_WORD;
signal CONST_REG70_NEW, CONST_REG71_NEW, CONST_REG72_NEW, CONST_REG73_NEW :
REG_WORD;

-- INTERMEDIARY RESULTS:
signal OPCODE1 : OPCODE_VEC; -- opcode in IR1
signal OPCODE2 : OPCODE_VEC; -- opcode in IR2
signal IR1_OP1 : NATURAL range 0 to 7; -- 1st operand of IR1
signal IR1_OP2 : NATURAL range 0 to 7; -- 2nd operand of IR1
signal IR1_OP3 : STD_LOGIC_VECTOR(0 to OP3_WIDTH-1); -- 3rd operand of IR1
signal IR2_OP1 : NATURAL range 0 to 7; -- 1st operand of IR2
signal IR2_OP2 : NATURAL range 0 to 7; -- 2nd operand of IR2
signal GOTO1 : STD_LOGIC_VECTOR(0 to GOTO1_WIDTH - 1); -- GOTO offset GRA1
signal GOTO2 : STD_LOGIC_VECTOR(0 to GOTO1_WIDTH - 2); -- GOTO offset GRA2
signal COUNT_TST : STD_LOGIC; -- flag for COUNTER test
signal COUNT_VAL : STD_LOGIC; -- value to test COUNTER for

```

```

signal NUM_CASES : INTEGER range 0 to 7; -- for CASE instruction
signal CASE_CONDITIONAL : STD_LOGIC;
signal CASE_COUNT : STD_LOGIC;
-- for conditional INIT_CASE test COUNTER for this
signal PRESERVE : BOOLEAN;
-- if 'conditional case' is FALSE then preserve current state
signal CASE_RESULT1 : NATURAL range 0 to 31; -- CASE_OFFSET for GRA1
signal CASE_RESULT2 : NATURAL range 0 to 31; -- CASE_OFFSET for GRA2
signal PC1_GOTO_TRUE : INTEGER range 0 to 63; -- old PC1 + GOTO1 + 1
signal PC1_GOTO_FALSE : INTEGER range 0 to 63; -- old PC1 + GOTO1 + 2
signal PC2_GOTO_TRUE : INTEGER range 0 to 63; -- old PC2 + GOTO1 + 1
signal PC2_GOTO_FALSE : INTEGER range 0 to 63; -- old PC2 + GOTO1 + 2
signal MASKED1 : PIPE_WORD; -- DATA masked according to IR1
attribute NO_MODIFICATION of MASKED1 : signal is "TRUE";
signal MASKED2 : PIPE_WORD; -- DATA masked according to IR2
attribute NO_MODIFICATION of MASKED2 : signal is "TRUE";

begin-- BEHAVIOUR

-- parse IRs (also done in EXECUTION process!)
OPCODE1 <= IR1(0 to OPCODE_WIDTH-1);
IR1_OP1 <= CONV_INTEGER(UNSIGNED(
  IR1((OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + 2)
    to (OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH - 1)) ));
IR1_OP2 <= CONV_INTEGER(UNSIGNED(
  IR1((OPCODE_WIDTH + OP3_WIDTH + 2)
    to (OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH - 1)) ));
IR1_OP3 <= IR1((OPCODE_WIDTH) to (OPCODE_WIDTH + OP3_WIDTH-1));
OPCODE2 <= IR2(0 to OPCODE_WIDTH-1);
IR2_OP1 <= CONV_INTEGER(UNSIGNED(
  IR2((OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + 2)
    to (OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH - 1)) ));
IR2_OP2 <= CONV_INTEGER(UNSIGNED(
  IR2((OPCODE_WIDTH + OP3_WIDTH + 2)
    to (OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH - 1)) ));
GOTO1 <= IR1((OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH)
  to (OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH
    + GOTO1_WIDTH - 1));
GOTO2 <= IR2((OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH)
  to (OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH
    + GOTO1_WIDTH - 2));
COUNT_TST <= IR1(OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH + GOTO1_WIDTH);
COUNT_VAL <= IR1(OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH + GOTO1_WIDTH
+ 1);

-----
-- Registers
-- all assigned signals in this process are to be synthesized as registers
-----
REGISTERS : process (CLK, RESET)

begin -- process REGISTERS
  if CLK'event and CLK = '1' then
    if RESET = '1' then
      PARSE <= (others => '0');
      ID <= (others => '0');
      PC1 <= 0;
      PC2 <= 0;
      IR1 <= NOP & "00000000000000" & "111111" & "00" & "1" & "000000";
      -- NOP & operators & goto (-1) & IF_COUNTER & PARALLEL & padding
      IR2 <= NOP & "00000000000000" & "11111" & "00000000000";
      -- NOP & operators & goto (-1) & padding

```

```
-- this way the PCx_NEWS will always be set to 0 again
PAR_EX <= '1';
ACCU <= (others => '0');
COUNTER <= (others => '0');
CONST_REG00 <= (others => '0');
CONST_REG01 <= (others => '0');
CONST_REG02 <= (others => '0');
CONST_REG03 <= (others => '0');
CONST_REG10 <= (others => '0');
CONST_REG11 <= (others => '0');
CONST_REG12 <= (others => '0');
CONST_REG13 <= (others => '0');
CONST_REG20 <= (others => '0');
CONST_REG21 <= (others => '0');
CONST_REG22 <= (others => '0');
CONST_REG23 <= (others => '0');
CONST_REG30 <= (others => '0');
CONST_REG31 <= (others => '0');
CONST_REG32 <= (others => '0');
CONST_REG33 <= (others => '0');
CONST_REG40 <= (others => '0');
CONST_REG41 <= (others => '0');
CONST_REG42 <= (others => '0');
CONST_REG43 <= (others => '0');
CONST_REG50 <= (others => '0');
CONST_REG51 <= (others => '0');
CONST_REG52 <= (others => '0');
CONST_REG53 <= (others => '0');
CONST_REG60 <= (others => '0');
CONST_REG61 <= (others => '0');
CONST_REG62 <= (others => '0');
CONST_REG63 <= (others => '0');
CONST_REG70 <= (others => '0');
CONST_REG71 <= (others => '0');
CONST_REG72 <= (others => '0');
CONST_REG73 <= (others => '0');
else
  PARSE <= PARSE_NEW;
  ID <= ID_NEW;
  PC1 <= PC1_NEW;
  PC2 <= PC2_NEW;
  IR1 <= GRA1_IN;
  IR2 <= GRA2_IN;
  PAR_EX <= PAR_EX_NEW;
  ACCU <= ACCU_NEW;
  COUNTER <= UNSIGNED(COUNTER_NEW) - 1;
  CONST_REG00 <= CONST_REG00_NEW;
  CONST_REG01 <= CONST_REG01_NEW;
  CONST_REG02 <= CONST_REG02_NEW;
  CONST_REG03 <= CONST_REG03_NEW;
  CONST_REG10 <= CONST_REG10_NEW;
  CONST_REG11 <= CONST_REG11_NEW;
  CONST_REG12 <= CONST_REG12_NEW;
  CONST_REG13 <= CONST_REG13_NEW;
  CONST_REG20 <= CONST_REG20_NEW;
  CONST_REG21 <= CONST_REG21_NEW;
  CONST_REG22 <= CONST_REG22_NEW;
  CONST_REG23 <= CONST_REG23_NEW;
  CONST_REG30 <= CONST_REG30_NEW;
  CONST_REG31 <= CONST_REG31_NEW;
  CONST_REG32 <= CONST_REG32_NEW;
  CONST_REG33 <= CONST_REG33_NEW;
```

```
CONST_REG40 <= CONST_REG40_NEW;
CONST_REG41 <= CONST_REG41_NEW;
CONST_REG42 <= CONST_REG42_NEW;
CONST_REG43 <= CONST_REG43_NEW;
CONST_REG50 <= CONST_REG50_NEW;
CONST_REG51 <= CONST_REG51_NEW;
CONST_REG52 <= CONST_REG52_NEW;
CONST_REG53 <= CONST_REG53_NEW;
CONST_REG60 <= CONST_REG60_NEW;
CONST_REG61 <= CONST_REG61_NEW;
CONST_REG62 <= CONST_REG62_NEW;
CONST_REG63 <= CONST_REG63_NEW;
CONST_REG70 <= CONST_REG70_NEW;
CONST_REG71 <= CONST_REG71_NEW;
CONST_REG72 <= CONST_REG72_NEW;
CONST_REG73 <= CONST_REG73_NEW;
end if;
end if;
end process REGISTERS;
```

```
-----
-- New Register Values
-- determine new register values, depending on DELIMIT signal
-----
```

```
PAR_EX_PROCESS : process (DELIMIT, IR1, OPCODE1, OPCODE2, PRESERVE, PAR_EX)
begin -- process PAR_EX_PROCESS
  if DELIMIT = '1' and OPCODE1 /= LD_CONST and OPCODE2 /= LD_CONST then
    PAR_EX_NEW <= '1';
  elsif PRESERVE then
    PAR_EX_NEW <= PAR_EX;
  else
    PAR_EX_NEW <=
      IR1(OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH + OP1_WIDTH + GOTO1_WIDTH + 2);
  end if;
end process PAR_EX_PROCESS;
```

```
-----
-- Case Instruction
-- computes the PC offset introduced by a CASE instruction
-----
```

```
NUM_CASES <= CONV_INTEGER(UNSIGNED(
  IR1_OP3(OP3_WIDTH-CASE_WIDTH to OP3_WIDTH-1)));
-- NUM_CASES = number of checked cases (excluding 'default') MINUS ONE
-- (for encoding => 3 bits are enough, zero checked cases doesn't make sense)
CASE_CONDITIONAL <= IR1(OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH - 1);
CASE_COUNT <= IR1(OPCODE_WIDTH + OP3_WIDTH + OP2_WIDTH - 2);

PRESERVE <= (OPCODE1 = INIT_CASE) and (CASE_CONDITIONAL = '1') and
  (COUNTER /= "0000000"&CASE_COUNT);
```

```
CASE_INSTRUCTION : process (OPCODE1, NUM_CASES, CASE_CONDITIONAL, CASE_COUNT,
  ACCU,
  CONST_REG00, CONST_REG01, CONST_REG02, CONST_REG03,
  CONST_REG10, CONST_REG11, CONST_REG12, CONST_REG13,
  CONST_REG20, CONST_REG21, CONST_REG22, CONST_REG23,
  CONST_REG30, CONST_REG31, CONST_REG32, CONST_REG33,
  CONST_REG40, CONST_REG41, CONST_REG42, CONST_REG43,
  CONST_REG50, CONST_REG51, CONST_REG52, CONST_REG53,
  CONST_REG60, CONST_REG61, CONST_REG62, CONST_REG63,
```

```

CONST_REG70, CONST_REG71, CONST_REG72, CONST_REG73)

variable CASE_RESULT : NATURAL range 0 to 31;

begin -- process CASE_INSTRUCTION
-- a CASE instruction is only considered in GRA1!
if ( OPCODE1 /= INIT_CASE
    or UNSIGNED(ACCU) = UNSIGNED(CONST_REG00)
    or UNSIGNED(ACCU) = UNSIGNED(CONST_REG01)
    or UNSIGNED(ACCU) = UNSIGNED(CONST_REG02)
    or UNSIGNED(ACCU) = UNSIGNED(CONST_REG03) )
then
    CASE_RESULT := 0;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG10)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG11)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG12)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG13) )
    and NUM_CASES > 0 )
then
    CASE_RESULT := 1;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG20)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG21)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG22)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG23) )
    and NUM_CASES > 1 )
then
    CASE_RESULT := 2;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG30)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG31)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG32)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG33) )
    and NUM_CASES > 2 )
then
    CASE_RESULT := 3;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG40)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG41)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG42)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG43) )
    and NUM_CASES > 3 )
then
    CASE_RESULT := 4;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG50)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG51)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG52)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG53) )
    and NUM_CASES > 4 )
then
    CASE_RESULT := 5;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG60)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG61)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG62)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG63) )
    and NUM_CASES > 5 )
then
    CASE_RESULT := 6;
elsif ( ( UNSIGNED(ACCU) = UNSIGNED(CONST_REG70)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG71)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG72)
        or UNSIGNED(ACCU) = UNSIGNED(CONST_REG73) )
    and NUM_CASES > 6 )
then
    CASE_RESULT := 7;

```

```
else
    CASE_RESULT := NUM_CASES+1;
end if;

CASE_RESULT1 <= CASE_RESULT * 4;-- 4 = code lines per case
CASE_RESULT2 <= CASE_RESULT * 2;-- PC2 only 2 lines/case

end process CASE_INSTRUCTION;

-----
-- Masking protocol DATA
-- masks incoming DATA according to slice borders given by OP2 and OP1 and
-- right-aligns result; done separately for each GRA
-----

MASKING : process (DATA, IR1_OP1, IR1_OP2, IR2_OP1, IR2_OP2)

-- purpose: is called once for each IR
function MASK (SIGNAL DATA : in PIPE_WORD;
               SIGNAL OP1, OP2 : in NATURAL range 0 to 7) return PIPE_WORD is
    variable MASKED : PIPE_WORD;
begin -- MASK
    for I in 0 to 7 loop
        if (7 - (OP1 - OP2)) > I then
            MASKED(I*4)    := '0';
            MASKED(I*4+1) := '0';
            MASKED(I*4+2) := '0';
            MASKED(I*4+3) := '0';
        else
            MASKED(I*4)    := DATA( (I - (7 - OP1)) *4 );
            MASKED(I*4+1) := DATA( (I - (7 - OP1)) *4+1);
            MASKED(I*4+2) := DATA( (I - (7 - OP1)) *4+2);
            MASKED(I*4+3) := DATA( (I - (7 - OP1)) *4+3);
        end if;
    end loop; -- I
    return MASKED;
end MASK;

begin -- process MASKING
    MASKED1 <= MASK(DATA, IR1_OP1, IR1_OP2);
    MASKED2 <= MASK(DATA, IR2_OP1, IR2_OP2);
end process MASKING;

-----
-- Add GOTO offset to PCs, and obligatory 1-increment
-- for two alternatives: IF_COUNTER is TRUE or FALSE
-----

ADD_GOTO : process (PC1, PC2, GOTO1, GOTO2)

variable PC1_TRUE, PC1_FALSE : INTEGER range 0 to 63;
variable PC2_TRUE, PC2_FALSE : INTEGER range 0 to 63;

begin -- process ADD_GOTO
    PC1_TRUE := PC1 + 1; -- offset for IF_COUNTER = TRUE
    PC1_FALSE := PC1 + 2; -- offset for IF_COUNTER = FALSE
    PC2_TRUE := PC2 + 1;
    PC2_FALSE := PC2 + 2;

    PC1_GOTO_TRUE <= PC1_TRUE + CONV_INTEGER(SIGNED( GOTO1 ));
    PC1_GOTO_FALSE <= PC1_FALSE + CONV_INTEGER(SIGNED( GOTO1 ));
    PC2_GOTO_TRUE <= PC2_TRUE + CONV_INTEGER(SIGNED( GOTO2 ));
```

```

    PC2_GOTO_FALSE <= PC2_FALSE + CONV_INTEGER(SIGNED( GOTO2 ));
end process ADD_GOTO;

-----
-- Compute New PC
-----

NEW_PC : process (PRESERVE, PAR_EX_NEW, DELIMIT, CASE_RESULT1, CASE_RESULT2,
    PC1_GOTO_TRUE, PC1_GOTO_FALSE, PC2_GOTO_TRUE, PC2_GOTO_FALSE,
    COUNT_TST, COUNT_VAL, COUNTER, OPCODE1, OPCODE2, PC1, PC2)

    variable PC1_GOTO_COUNT, PC2_GOTO_COUNT : INTEGER range 0 to 63;

begin -- process NEW_PC
    if (COUNT_TST = '1') and (COUNTER /= "0000000"&COUNT_VAL) then
        PC1_GOTO_COUNT := PC1_GOTO_FALSE;
        PC2_GOTO_COUNT := PC2_GOTO_FALSE;
    else
        PC1_GOTO_COUNT := PC1_GOTO_TRUE;
        PC2_GOTO_COUNT := PC2_GOTO_TRUE;
    end if;

    if DELIMIT = '1' and OPCODE1 /= LD_CONST and OPCODE2 /= LD_CONST then
        PC1_NEW <= 1;
    elsif PRESERVE then
        PC1_NEW <= PC1;
    else
        PC1_NEW <= CASE_RESULT1 + PC1_GOTO_COUNT;
    end if;

    if PAR_EX_NEW = '1' then
        if DELIMIT = '1' and OPCODE1 /= LD_CONST and OPCODE2 /= LD_CONST then
            PC2_NEW <= 1;
        elsif PRESERVE then
            PC2_NEW <= PC2;
        else
            PC2_NEW <= CASE_RESULT2 + PC2_GOTO_COUNT;
        end if;
    else
        PC2_NEW <= PC2;
    end if;
end process NEW_PC;

-- apply new PCs to GRAs
GRA1_ADDR <= STD_LOGIC_VECTOR(CONV_UNSIGNED(PC1_NEW, GRA1_ADDR_WIDTH));
GRA2_ADDR <= STD_LOGIC_VECTOR(CONV_UNSIGNED(PC2_NEW, GRA2_ADDR_WIDTH));

-----
-- Instruction Execution
-----

EXECUTION : process (IR1, IR2, MASKED1, MASKED2, ACCU, COUNTER, PAR_EX,
    CONST_REG00, CONST_REG01, CONST_REG02, CONST_REG03,
    CONST_REG10, CONST_REG11, CONST_REG12, CONST_REG13,
    CONST_REG20, CONST_REG21, CONST_REG22, CONST_REG23,
    CONST_REG30, CONST_REG31, CONST_REG32, CONST_REG33,
    CONST_REG40, CONST_REG41, CONST_REG42, CONST_REG43,
    CONST_REG50, CONST_REG51, CONST_REG52, CONST_REG53,
    CONST_REG60, CONST_REG61, CONST_REG62, CONST_REG63,
    CONST_REG70, CONST_REG71, CONST_REG72, CONST_REG73)

    -- purpose: parses and executes instructions

```

```
procedure EXECUTE (signal ACCU : in REG_WORD; signal ACCU_NEW : out REG_WORD;
    signal COUNTER : in REG_WORD; signal COUNTER_NEW : out REG_WORD;
    signal CONST_REG00_NEW, CONST_REG01_NEW, CONST_REG02_NEW,
CONST_REG03_NEW : out REG_WORD;
    signal CONST_REG10_NEW, CONST_REG11_NEW, CONST_REG12_NEW,
CONST_REG13_NEW : out REG_WORD;
    signal CONST_REG20_NEW, CONST_REG21_NEW, CONST_REG22_NEW,
CONST_REG23_NEW : out REG_WORD;
    signal CONST_REG30_NEW, CONST_REG31_NEW, CONST_REG32_NEW,
CONST_REG33_NEW : out REG_WORD;
    signal CONST_REG40_NEW, CONST_REG41_NEW, CONST_REG42_NEW,
CONST_REG43_NEW : out REG_WORD;
    signal CONST_REG50_NEW, CONST_REG51_NEW, CONST_REG52_NEW,
CONST_REG53_NEW : out REG_WORD;
    signal CONST_REG60_NEW, CONST_REG61_NEW, CONST_REG62_NEW,
CONST_REG63_NEW : out REG_WORD;
    signal CONST_REG70_NEW, CONST_REG71_NEW, CONST_REG72_NEW,
CONST_REG73_NEW : out REG_WORD;
    signal PARSE_NEW : out PARSE_WORD;
    signal ID_NEW : out ID_WORD;
    signal IR : in GRA2_WORD;
    signal MASKED : in PIPE_WORD) is

    variable CONST_VAL : REG_WORD; -- temporary for new CONST_REG value

    alias OPCODE : OPCODE_VEC is IR(0 to OPCODE_WIDTH - 1);
    alias OP3 : STD_LOGIC_VECTOR(0 to OP3_WIDTH-1) is
        IR((OPCODE_WIDTH) to (OPCODE_WIDTH + OP3_WIDTH-1));

begin -- EXECUTE

    case OPCODE is
        when NOP =>
            null;

        when LD_CONST =>
            -- (special instruction operator format)

            CONST_VAL := IR((OPCODE_WIDTH + 6) to (OPCODE_WIDTH + 13));

            case CONV_INTEGER(UNSIGNED(
                IR(OPCODE_WIDTH + 1 to (OPCODE_WIDTH + 5)) )) is
                when 0 => CONST_REG00_NEW <= CONST_VAL;
                when 1 => CONST_REG01_NEW <= CONST_VAL;
                when 2 => CONST_REG02_NEW <= CONST_VAL;
                when 3 => CONST_REG03_NEW <= CONST_VAL;
                when 4 => CONST_REG10_NEW <= CONST_VAL;
                when 5 => CONST_REG11_NEW <= CONST_VAL;
                when 6 => CONST_REG12_NEW <= CONST_VAL;
                when 7 => CONST_REG13_NEW <= CONST_VAL;
                when 8 => CONST_REG20_NEW <= CONST_VAL;
                when 9 => CONST_REG21_NEW <= CONST_VAL;
                when 10 => CONST_REG22_NEW <= CONST_VAL;
                when 11 => CONST_REG23_NEW <= CONST_VAL;
                when 12 => CONST_REG30_NEW <= CONST_VAL;
                when 13 => CONST_REG31_NEW <= CONST_VAL;
                when 14 => CONST_REG32_NEW <= CONST_VAL;
                when 15 => CONST_REG33_NEW <= CONST_VAL;
                when 16 => CONST_REG40_NEW <= CONST_VAL;
                when 17 => CONST_REG41_NEW <= CONST_VAL;
                when 18 => CONST_REG42_NEW <= CONST_VAL;
```

```

        when 19 => CONST_REG43_NEW <= CONST_VAL;
        when 20 => CONST_REG50_NEW <= CONST_VAL;
        when 21 => CONST_REG51_NEW <= CONST_VAL;
        when 22 => CONST_REG52_NEW <= CONST_VAL;
        when 23 => CONST_REG53_NEW <= CONST_VAL;
        when 24 => CONST_REG60_NEW <= CONST_VAL;
        when 25 => CONST_REG61_NEW <= CONST_VAL;
        when 26 => CONST_REG62_NEW <= CONST_VAL;
        when 27 => CONST_REG63_NEW <= CONST_VAL;
        when 28 => CONST_REG70_NEW <= CONST_VAL;
        when 29 => CONST_REG71_NEW <= CONST_VAL;
        when 30 => CONST_REG72_NEW <= CONST_VAL;
        when 31 => CONST_REG73_NEW <= CONST_VAL;
        when others => assert FALSE report "parser_core: Illegal CONST_REG
number!?! " severity WARNING;
        end case;

    when SEND =>
        PARSE_NEW <= MASKED;
        ID_NEW <= OP3((OP3_WIDTH - ID_WIDTH) to OP3_WIDTH - 1);

    when SEND_REG =>
        PARSE_NEW((PARSE_WIDTH - REG_WIDTH) to (PARSE_WIDTH - 1)) <= ACCU;
        ID_NEW <= OP3((OP3_WIDTH - ID_WIDTH) to OP3_WIDTH - 1);

    when WRITE_REG =>
        if (UNSIGNED(OP3) = 0) then
            COUNTER_NEW <=
                MASKED(PIPE_WIDTH - REG_WIDTH to PIPE_WIDTH - 1);
        else
            ACCU_NEW <=
                MASKED(PIPE_WIDTH - REG_WIDTH to PIPE_WIDTH - 1);
        end if;

    when IP6_COUNTER =>
        -- HdrExtLen * 2 + 1
        -- MSB is dropped if COUNTER has not more bits
        -- than HdrExtLen!
        COUNTER_NEW <= MASKED(PIPE_WIDTH + 1 - REG_WIDTH to PIPE_WIDTH - 1)&'1';

    when INIT_CASE =>
        null; -- is handled at PC computation

    when others =>-- unknown instruction
        assert FALSE report "parser_core: Unknown instruction!!!" severity
WARNING;
        end case;

    end EXECUTE;

begin -- process EXECUTION
    -- in case the outputs/registers are not written to by an instruction:
    PARSE_NEW <= (others => '0');
    ID_NEW <= (others => '0');
    ACCU_NEW <= ACCU;
    COUNTER_NEW <= COUNTER;
    CONST_REG00_NEW <= CONST_REG00;
    CONST_REG01_NEW <= CONST_REG01;
    CONST_REG02_NEW <= CONST_REG02;
    CONST_REG03_NEW <= CONST_REG03;
    CONST_REG10_NEW <= CONST_REG10;
    CONST_REG11_NEW <= CONST_REG11;

```

```
CONST_REG12_NEW <= CONST_REG12;
CONST_REG13_NEW <= CONST_REG13;
CONST_REG20_NEW <= CONST_REG20;
CONST_REG21_NEW <= CONST_REG21;
CONST_REG22_NEW <= CONST_REG22;
CONST_REG23_NEW <= CONST_REG23;
CONST_REG30_NEW <= CONST_REG30;
CONST_REG31_NEW <= CONST_REG31;
CONST_REG32_NEW <= CONST_REG32;
CONST_REG33_NEW <= CONST_REG33;
CONST_REG40_NEW <= CONST_REG40;
CONST_REG41_NEW <= CONST_REG41;
CONST_REG42_NEW <= CONST_REG42;
CONST_REG43_NEW <= CONST_REG43;
CONST_REG50_NEW <= CONST_REG50;
CONST_REG51_NEW <= CONST_REG51;
CONST_REG52_NEW <= CONST_REG52;
CONST_REG53_NEW <= CONST_REG53;
CONST_REG60_NEW <= CONST_REG60;
CONST_REG61_NEW <= CONST_REG61;
CONST_REG62_NEW <= CONST_REG62;
CONST_REG63_NEW <= CONST_REG63;
CONST_REG70_NEW <= CONST_REG70;
CONST_REG71_NEW <= CONST_REG71;
CONST_REG72_NEW <= CONST_REG72;
CONST_REG73_NEW <= CONST_REG73;

EXECUTE( ACCU, ACCU_NEW, COUNTER, COUNTER_NEW,
         CONST_REG00_NEW, CONST_REG01_NEW, CONST_REG02_NEW, CONST_REG03_NEW,
         CONST_REG10_NEW, CONST_REG11_NEW, CONST_REG12_NEW, CONST_REG13_NEW,
         CONST_REG20_NEW, CONST_REG21_NEW, CONST_REG22_NEW, CONST_REG23_NEW,
         CONST_REG30_NEW, CONST_REG31_NEW, CONST_REG32_NEW, CONST_REG33_NEW,
         CONST_REG40_NEW, CONST_REG41_NEW, CONST_REG42_NEW, CONST_REG43_NEW,
         CONST_REG50_NEW, CONST_REG51_NEW, CONST_REG52_NEW, CONST_REG53_NEW,
         CONST_REG60_NEW, CONST_REG61_NEW, CONST_REG62_NEW, CONST_REG63_NEW,
         CONST_REG70_NEW, CONST_REG71_NEW, CONST_REG72_NEW, CONST_REG73_NEW,
         PARSE_NEW, ID_NEW, IR1(0 to GRA2_WIDTH-1), MASKED1 );
if (PAR_EX = '1') then
    EXECUTE( ACCU, ACCU_NEW, COUNTER, COUNTER_NEW,
            CONST_REG00_NEW, CONST_REG01_NEW, CONST_REG02_NEW, CONST_REG03_NEW,
            CONST_REG10_NEW, CONST_REG11_NEW, CONST_REG12_NEW, CONST_REG13_NEW,
            CONST_REG20_NEW, CONST_REG21_NEW, CONST_REG22_NEW, CONST_REG23_NEW,
            CONST_REG30_NEW, CONST_REG31_NEW, CONST_REG32_NEW, CONST_REG33_NEW,
            CONST_REG40_NEW, CONST_REG41_NEW, CONST_REG42_NEW, CONST_REG43_NEW,
            CONST_REG50_NEW, CONST_REG51_NEW, CONST_REG52_NEW, CONST_REG53_NEW,
            CONST_REG60_NEW, CONST_REG61_NEW, CONST_REG62_NEW, CONST_REG63_NEW,
            CONST_REG70_NEW, CONST_REG71_NEW, CONST_REG72_NEW, CONST_REG73_NEW,
            PARSE_NEW, ID_NEW, IR2, MASKED2 );
end if;
end process EXECUTION;

end BEHAVIOUR;
```

Appendix B: Acronyms

ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Integrated Processor
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
ESP	Encapsulating Security Payload
Gbps	Giga bit per second
GPP	General Purpose Processor
GRA	Growable Register Array
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IHL	Internet Header Length
IP	Internet Protocol
IR	Instruction Register
ISO	International Standards Organization
LAN	Local Area Network
MPLS	Multi Protocol Label Switching
NP	Network Processor
OSI	Open Systems Interconnect
OSPF	Open Shortest Path First
PC	Program Counter
PoS	Packet over SONET
PPP	Point-to-Point Protocol
QoS	Quality of Service
SONET	Synchronous Optical NETWORK
TCP	Transmission Control Protocol
ToS	Type of Service
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLAN	Virtual LAN
WAN	Wide Area Network

Appendix C: Bibliography

- [DH98] *Stephen E. Deering, Robert M. Hinden*
Internet Protocol, Version 6 (IPv6) Specification
IETF, RFC 2460, December 1998
- [HP96] *John L. Hennessy, David A. Patterson*
Computer Architecture: A Quantitative Approach
Morgan Kaufmann, Second Edition, 1996
- [IANA] **Internet Assigned Numbers Authority (IANA)**
<http://www.iana.org>
- [IBM] *IBM Microelectronics*
ASIC Technology
<http://www.chips.ibm.com/products/asics/>
- [ISI81] *Information Sciences Institute*
Transmission Control Protocol
IETF, RFC 793, September 1981
- [KA98] *Stephen Kent, Randall Atkinson*
IP Authentication Header
IETF, RFC 2402, November 1998
- [KA98b] *Stephen Kent, Randall Atkinson*
IP Encapsulating Security Payload (ESP)
IETF, RFC 2406, November 1998
- [Pos80] *John Postel*
User Datagram Protocol
IETF, RFC 768, August 1980
- [Pos81] *John Postel (Editor)*
Internet Protocol
IETF, RFC 791, September 1981
- [Tan96] *Andrew S. Tanenbaum*
Computer Networks
Prentice Hall, Third Edition, 1996