

CESNET technical report number 5/2003

**Packet Analysis for IPv6 Router  
Implemented by a PCI Acceleration Card**

Bc. Thesis

Brno, May 2003

Filip Höfer

## **Acknowledgments**

I would like to thank my supervisor, David Antoš for help and encouragement throughout the work. I am very grateful for his advice and valuable discussions.

This work is supported by the CESNET association in scope of the Liberouter project. I would like to thank the members of this team for cooperation, especially Ivo Hažmuk, Štěpán Friedl and Kateřina Minaříková.

## **Declaration**

I declare that this thesis was composed by myself, and all presented results are my own, unless otherwise stated. All sources and literature that I have used during the elaboration of this thesis are cited with complete reference to the corresponding source.

Filip Höfer

## **Abstract**

This document concerns the development of the Combo6 card that aims at accelerating IPv6 and IPv4 routing. Three main processing blocks are distinguished in its architecture: packet parsing blocks, header matching blocks and output blocks.

My work consists of several tools dealing with the simulation of packet parsing blocks. These tools provide possibilities for simulation, testing and developing algorithms for the hardware. Further, outputs of these tools may be used for simulation and testing of header matching blocks, even parallel simulation of packet parsing blocks and header matching blocks is possible.

## **Keywords**

IPv6, router, packet, packet parsing, simulation, VHDL, nanoprocessor, Liberouter, Combo6, header field extractor, assembler, HFE

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>I</b>	<b>Packet Parsing</b>	<b>2</b>
<b>2</b>	<b>Packet Analysis</b>	<b>3</b>
2.1	TCP/IP Network Model . . . . .	3
2.2	Internet Protocol . . . . .	3
2.3	Data Important for Routing . . . . .	6
2.4	Layer Encapsulation . . . . .	7
2.5	Packet Capturing . . . . .	7
<b>3</b>	<b>Hardware Architecture</b>	<b>8</b>
3.1	IPv6 Router Overview . . . . .	8
3.2	Header Field Extractor . . . . .	10
<b>II</b>	<b>Simulation</b>	<b>13</b>
<b>4</b>	<b>VHDL Tools</b>	<b>14</b>
4.1	VHDL Simulation Example . . . . .	14
<b>5</b>	<b>L2_strip</b>	<b>16</b>
5.1	L2_strip Simulator . . . . .	16
5.2	Parallel Simulator . . . . .	18
<b>6</b>	<b>Nanoprocessor Simulator</b>	<b>23</b>
6.1	Development History . . . . .	23
6.2	Architecture of the Tool . . . . .	24
6.3	Input . . . . .	24
6.4	Output . . . . .	24
6.5	Usage . . . . .	25
6.6	The Debugger . . . . .	25
6.7	The Use of the Tool . . . . .	25
6.8	Implementation Details . . . . .	27
6.9	Cross-platform Compatibility Issues . . . . .	31
6.10	Evaluation . . . . .	32

6.11 Future Plans and Improvements . . . . .	32
<b>7 Conclusion</b>	<b>33</b>
<b>A Nanoprocessor Assembler</b>	<b>35</b>
A.1 Basic Elements . . . . .	35
A.2 Directives . . . . .	36
A.3 Instructions . . . . .	41
A.4 Labels . . . . .	41
<b>B Contents of the CD</b>	<b>42</b>

# Chapter 1

## Introduction

The increasing importance of Internet Protocol version 6 (IPv6) is a great phenomenon in the area of networking. The transition to IPv6 brings a wide range of changes—from application software to routers. Development of IPv6 router is the basis topic of this thesis.

The router is implemented by a PCI card that is intended to work as an IPv4 and IPv6 routing accelerator in UNIX PCs. The routing and firewalling (r/f) tables will be maintained by the operating system and the packet switching will be kept in hardware. The tables must be passed over the PCI bus in a proper format. The format is determined by the fact that the core of the accelerator utilizes CAM (Contents Addressable Memory)—the CAM and some auxiliary memories must be filled. CAM enables packet headers to be matched to entries of r/f tables at a high speed. However, this way of fast searches cannot be applied to raw packets. The reason is that it would work only if all IP packets had the same bitwise structure, but they have not. That is why a new structure has been issued—Unified Header. This structure is produced for each packet, it holds enough information for routing and meets the requirement of universality. The Unified Header is produced by another part of the accelerator. For now, we can call it a packet parser.

The job of the packet parser is to find the way through the string of headers of an IP packet, extract the data needed by the accelerator core and store it at the right positions in the Unified Header. Since the Internet Protocol distinguishes many headers (i.e., hop-by-hop options, routing, fragment, authentication, destination options, etc.) and the speed matters, the hardware implementation choice was paid high attention. The final choice—a full scope processor optimized for the purpose of the parsing—brings many advantages. Above all, the parsing program may be updated without changing the hardware design.

However, the elegance of the solution would be lost if the programs had to be written in the machine code. The machine code is modified during the development and that makes older binary programs useless. Assembler is a lot smarter for the programmer and costs no speed loss. If we have a program, we want to see how it works. We might either want to compile it and load it into the hardware or just have it simulated. It is quite difficult to find out what is happening in the hardware and it usually requires the use of devices such as logical analysers. The simulation aims at providing a transparent look at the hardware functionality.

Part I is devoted to the issues outlined in the first three paragraphs. Part II concerns software tools that deal with hardware development with an emphasis on the packet parsing simulation. Finally, the contributions of this thesis are summarized.

Part I

**Packet Parsing**

## Chapter 2

# Packet Analysis

### 2.1 TCP/IP Network Model

The Internet is based on TCP/IP (Transmission Control Protocol/Internet Protocol) network model [1]. TCP/IP uses a simplified architecture in comparison to the standard OSI model. It recognizes only four layers: host-to-network layer, Internet layer, transport layer and application layer. Within the host-to-network layer it is possible to recognize physical layer and link layer. In further text, we will use this hierarchical notation of layers:

- L1 physical layer
- L2 link layer
- L3 network layer (Internet uses IP)
- L4 transport layer (Internet uses TCP)
- L5 application layer

Each layer uses its own protocol. While L3 and L4 protocols are defined by the TCP/IP model, the other layers may use any protocol relevant to the given layer.

### 2.2 Internet Protocol

IP is widely acknowledged as a standardized network layer protocol. In recent years, its current version IPv4 is running out of address space. That is the main reason why its successor, IPv6, has been developed since the nineties. IPv6 is expected to spread widely in the following years, however IPv6 and IPv4 will coexist some time. Currently, the vast majority of the Internet is still IPv4 based and the time of switching to IPv6 is questionable. This is mostly because of technologies that prolong the life of the old protocol. First of all, we should mention NAT (Network Address Translation) and associated technologies<sup>1</sup>. NAT allows one valid public IP address to be shared by many machines, via a gateway that dynamically modifies source and destination IP address and port numbers of packets. Despite the conservation of the address space and security improvement, this technique has many shortcomings. Problems arise when peer-to-peer applications<sup>2</sup> come into play because they demand unsolicited in-bound connectivity. There are efforts to bypass these problems, but the transition to IPv6 is inevitable for full Internet functionality preservation and its smooth growth.

---

<sup>1</sup>Dan Kegel's UDP Hack, Realm-Specific IP, Address Virtualization Enabling Service

<sup>2</sup>file sharing, instant messaging, personal video conferencing, multiplayer games, etc.

Figure 2.1: IPv4 header

8				8				8				8				<i>bits</i>
0	1	0	0	header len.	type of service				total length							
identification								flags		fragment offset						
time to live				protocol				header checksum								
source address																
destination address																
options																

Currently, the valid IPv4 specification is the RFC 791 issued in 1981 and the valid IPv6 specification is the RFC 2460 issued in 1998. The structure of IPv4 and IPv6 packets is shown in Figure 2.1 and Figure 2.2, respectively.

Just for completeness, we can briefly mention the other versions of IP. Versions 0 through 3 were either reserved or unused. Version 5 was used for an experimental multicast protocol. Other version numbers have been assigned, usually for experimental protocols, but have not been widely used.

### 2.2.1 IPv4 Datagram Description

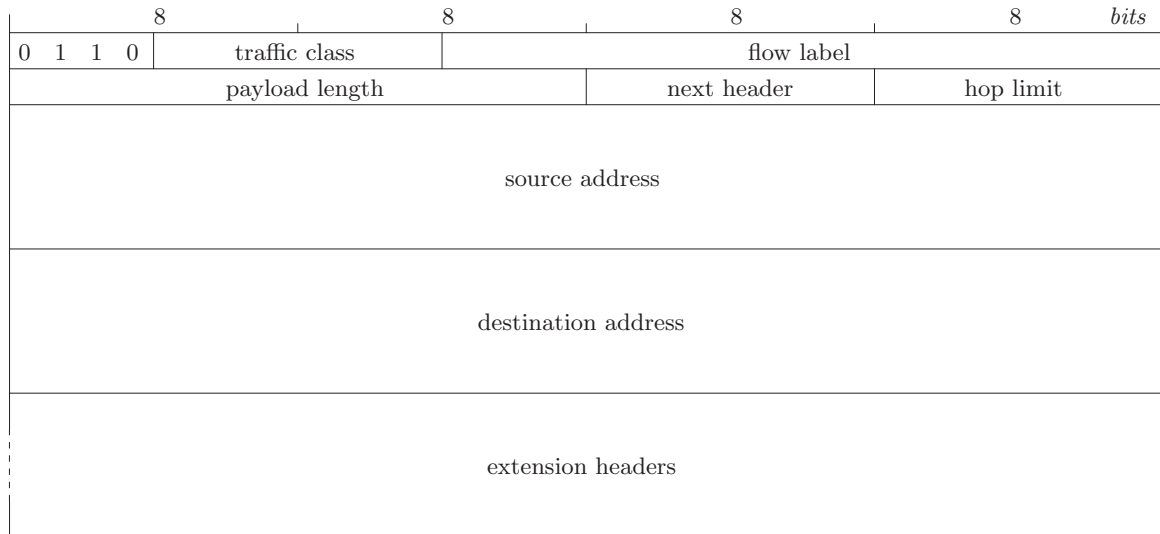
The IPv4 header is shown in Figure 2.1.

The first header field in an IPv4 datagram is the 4-bit version field. The second field is a 4-bit Internet header length (IHL) denoting the number of 32-bit words in the IPv4 header. Since an IPv4 header may contain a variable number of options, this field essentially specifies the offset to the data portion of an IPv4 datagram. A minimum IPv4 header is 20 bytes long so the value in decimal in the IHL field would be 5. In RFC 791, the following 8 bits were allocated to a type of service (ToS) field. The original intention was for a sending host to specify a preference for how the datagram would be handled<sup>3</sup> on its way through the Internet. In practice, the ToS field has not been widely implemented. The 16-bit total length field defines the entire datagram size, including header and data, in 8-bit bytes. The minimum-length datagram is 576 bytes and the maximum is 65535. However, subnetworks may impose further restrictions on the size. If the packet size exceeds the maximum transfer unit (MTU), the packet must be fragmented. Fragmentation is handled in either the host or packet switch in IPv4 (or in the host only in IPv6).

The following 32 bits are used for fragmentation handling. The 16-bit identification field is primarily used for uniquely identifying fragments of an original IP datagram. Some experimental work has suggested using the identification field for other purposes, such as for adding packet tracing information. A 3-bit field follows and is used to control or identify fragments. The fragment offset field is 13-bits long and helps a receiver determine the place of a particular fragment in the original IP datagram.

<sup>3</sup>For instance, one host could set its IPv4 datagrams' ToS field value to prefer for low delay, while another might prefer high reliability.

Figure 2.2: IPv6 header



The 8-bit time to live (TTL) is a value in an IP packet that tells a network router whether or not the packet has been in the network too long and should be discarded. An 8-bit Protocol field follows. This field defines the next protocol used in the data portion of the IP datagram. The Internet Assigned Numbers Authority maintains a list of Protocol numbers [3]. Common protocols and their decimal values include ICMP (1), TCP (6) and UDP (17). The following field is a 16-bit header checksum field. Some values in an IPv4 datagram header may change at each packet switch hop, so the checksum must be recomputed on its way through the Internet.

The checksum is followed by a 32-bit source address and 32-bit destination address respectively.

Optional header fields may follow the destination address field, but these are not used in most IP networks. Option fields may be followed by a pad field that ensures the datagram header is aligned on a 32-bit word boundary.

## 2.2.2 IPv6 Datagram Description

The IPv6 header is shown in Figure 2.2.

Again, the first header field in an IPv6 datagram is the 4-bit version field. The second field is an 8-bit traffic class telling packet priority or its enlistment into a certain traffic class. This field is not used yet and is required to be zero. The meaning is similar to the type of service field in IPv4. The following 20 bits should label a flow of packets with common properties helping a router to quickly determine the handling of each packet in the flow. This flow label has been just a matter of experiments yet.

The 16-bit payload length carries the information on datagram size, excluding the base header, in 8-bit bytes. The maximum length is 65535 bytes. If the payload is bigger than that, this field is set to zero and a special Jumbo Payload option is set up. The next 8-bit field defines either header or data type that follows the base header. The extension headers are separate blocks that are linked by the next header field. The order of the headers is specified.

See [8] for detailed information on the extension headers. The 8-bit hop limit has the same meaning as the TTL in IPv4, it is only a more accurate name for the same value. Historically, the TTL field limited lifetime of a datagram in seconds, but has come to be a hop count field. Each packet switch or router that a datagram crosses decrements this field by one. When it hits zero, the packet is discarded.

Two 128-bit addresses are the last items in the IPv6 base header.

### 2.2.3 IPv4 and IPv6 Comparison

The addresses are four times longer in IPv6 occupying 80% of the base header. The base header length rose from 20 bytes to 40 bytes while the number of fields fell from 12 to 8.

Unlike the IPv4, the length of the IPv6 base header is constant. Variable-length fields in IPv4 are replaced by extension headers in IPv6. Fragmentation and options fields are not included in the IPv6 base header, but this information may be present in extension headers. However, the fragmentation is expected to be minimized in IPv6. The reason is that IPv6 requires that the minimum size for a link-layer packet is 1280 bytes. Since the vast majority of end points are connected via Ethernet [7] with maximum transfer unit (MTU) equal to 1500 bytes, this size is expected to become a general maximum packet size. The header checksum is not supported in IPv6 assuming that the link layer will maintain the error detection. Header alignment changed from 32 bits to 64 bits.

## 2.3 Data Important for Routing

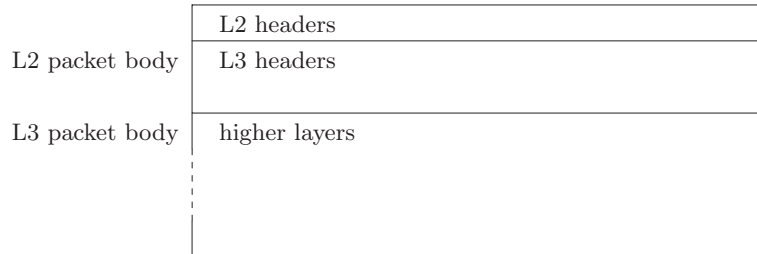
We will assume that Ethernet [7] is the L2 protocol and either IPv4 or IPv6 is the L3 protocol. If we consider packet routing and firewalling, the most important information includes:

- destination IP address,
- destination port,
- next hop from routing header,
- destination MAC address,
- source IP address,
- source port,
- source MAC address,
- time to live (TTL).
- VLAN tag<sup>4</sup>,
- L2 and L3 packet types,
- L2 and L3 present headers,
- L2 and L3 errors.

---

<sup>4</sup>VLAN tag contains VLAN id and priority. See [6] for details.

Figure 2.3: Layer encapsulation



## 2.4 Layer Encapsulation

In order to gather all this information, we must consider the encapsulation of the packet layers. Each layer provides a different view on the data. From the L2 point of view, we distinguish only L2 header followed by L2 packet body (and some extra information in dependence on protocol, i.e., Ethernet adds a 32-bit CRC). However, L2 packet body contains the whole L3 packet, i.e., both L3 headers and L3 packet body. This is shown in Figure 2.3.

## 2.5 Packet Capturing

Some packet data have to be collected for the purpose of testing and simulation. It is important that the data should contain L2 layer (and therefore all higher layers) and that the data should be available in the very same form as it comes from the network interface. Especially, network byte order (NBO)<sup>5</sup> is required instead of host byte order (HBO)<sup>6</sup>. These requirements are met by the well-known UNIX packet sniffer and analyzer `tcpdump` and its Windows version `windump`. Examples of capturing network traffic:

```
tcpdump -e -c 1 -w packet.bin
```

```
windump -i 1 -e -c 8 -w packets.bin vlan or ip6
```

The meaning of the switches used is the following:

- `-e` L2 headers are included
- `-c number` number of packets to be caught
- `-w file` output file
- `-i interface` network interface
- `vlan or ip6` condition for packets to be saved  
(in this case those that contain VLAN tag or IPv6 packets)

<sup>5</sup>The most significant byte is stored at the lowest address.

<sup>6</sup>In PC architecture, the least significant byte is stored at the lowest address.

## Chapter 3

# Hardware Architecture

This chapter concerns the Combo6 card, IPv6 router developed in scope of the *Liberouter* project<sup>1</sup> [4]. An emphasis is laid on the Header Field Extractor block which deals with the packet analysis.

### 3.1 IPv6 Router Overview

The developed router is based on the standard PC architecture. The idea of using a UNIX PC as a router is not new, PCs with BSD or Linux have been used quite widely in the Czech academic networks since the early nineties. However, maximum throughput of the PCI bus (4 Gbps) and the CPU (4 Gbps) is not enough though for today's gigabit networks. In other cases, CPU performance can be the bottleneck, especially when special functions like multicast come into play. That is why we develop an accelerator that is intended to take over the bandwidth- and CPU-demanding tasks from the main processor and the PCI bus and perform them at wire speed.

The Combo6 card is based upon the concept of programmable hardware using Field-Programmable Gate Arrays (FPGA). These electronic chips come from its vendor as a tabula rasa providing no functionality until a custom design is loaded. The hardware design may be described by one of hardware-specific high level programming languages; our project uses VHDL (Very High Speed Integrated Circuit Hardware Description Language). This high level description is then converted by a chain of tools (see Chapter 4) into bitstream—it is a microprogram, that can be loaded into the FPGA.

There are four input interfaces and four output interfaces on the card. In simple words, the job of the router is to analyse each incoming packet, update its headers and pass it to output interface(s). This process is intended to respect routing tables (and firewalling rules). Figure 3.1 shows the life-cycle of the packet passing through the router.

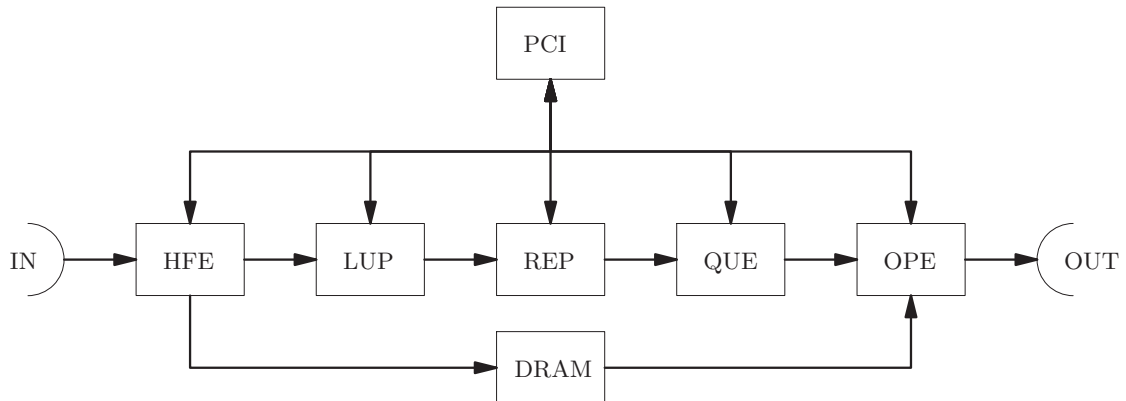
The most of the design blocks are currently under construction. Only the Header Field Extractor is ready to use, but there are still many optimizations left to be done. The work on integration of all design blocks has started recently.

The Ethernet [7] has been chosen as the L2 protocol because of its good price/performance and wide acceptance.

---

<sup>1</sup>This activity is a subproject in CESNET project *IPv6 Implementation in the CESNET2 Network* and is supported by the *6NET* project.

Figure 3.1: Life-cycle of the packet passing through the router



Description of the blocks:

- HFE Header Field Extractor performs packet analysis and stores packets into DRAM.
- LUP Look-Up Processor matches Unified Headers to routing table entries using CAM (Contents Addressable Memory); returns queue value (this value expresses how to handle the packet).
- REP Packet Replicator replicates the packet according to queue value.
- QUE Output Queue takes packets from the DRAM and passes them along with data from previous blocks to the OPE respecting a certain policy (i.e., simple FIFO, priority queues, round robin, etc.).
- OPE Output Editor updates packet headers and sends packets to output interfaces.
- PCI Peripheral Component Interconnect—the standard PCI bus provides communication with the PC operating system for the purpose of statistics, control and handling complicated packets.
- DRAM Dynamic Random Access Memory is used to store processed packets.

## 3.2 Header Field Extractor

Header Field Extractor (HFE) performs especially the packet analysis. The packet analysis results are stored in the structure named Unified Header (UH), this structure is described in Section 3.2.2. For each incoming packet HFE has to:

- request a free UH,
- request a block in DRAM,
- perform L2 and L3 parsing,
- fill UH,
- store the packet in DRAM.

HFE is implemented as a full scope processor that is controlled by binary instructions. These instructions form a program that may be described by an assembler. If the parsing program has to be modified or extended in the future, only the parsing program will have to be updated without changing the hardware design<sup>2</sup>. This model is robust enough to allow even handling different L2 protocols. Let us remind that the HFE is not a self-contained piece of hardware, but it is a microprogram inside the FPGA. This kind of processors has been assigned a new term *nanoprocessor* in the scope of the Liberouter project. Programs for nanoprocessors are called *nanoprograms*. Instructions of nanoprograms are executed by FPGA microprogram.

The memory requests follow this rule: HFE waits for the memory resources as long as the input buffer is less than half full. If memory resources are not allocated in time, the packet is trashed.

### 3.2.1 HFE Input

The FPGA that will contain HFE<sup>3</sup> includes a built-in L1 parser. Thus HFE has a L2 Ethernet packet on input. Along with packet data, the L1 parser provides status signals. These signals enable catching SOP<sup>4</sup> and EOP<sup>5</sup> events and provide information on input buffer status, L2 CRC<sup>6</sup> and L1 processing exceptions. The packet is read sequentially in 16-bit words.

### 3.2.2 HFE Output

The key output of the HFE is the Unified Header (UH). It is the result of the packet analysis and is further processed by the LUP. Figure 3.2 describes UH structure. UH is filled by the packet parsing program that is carried out by the core of the HFE—the processor.

In parallel, each incoming packet is automatically stored into DRAM. This does not interfere with the packet analysis because the packet storing is carried out by auxiliary circuits.

---

<sup>2</sup>Software that enables this way of development is described in Chapter 6.

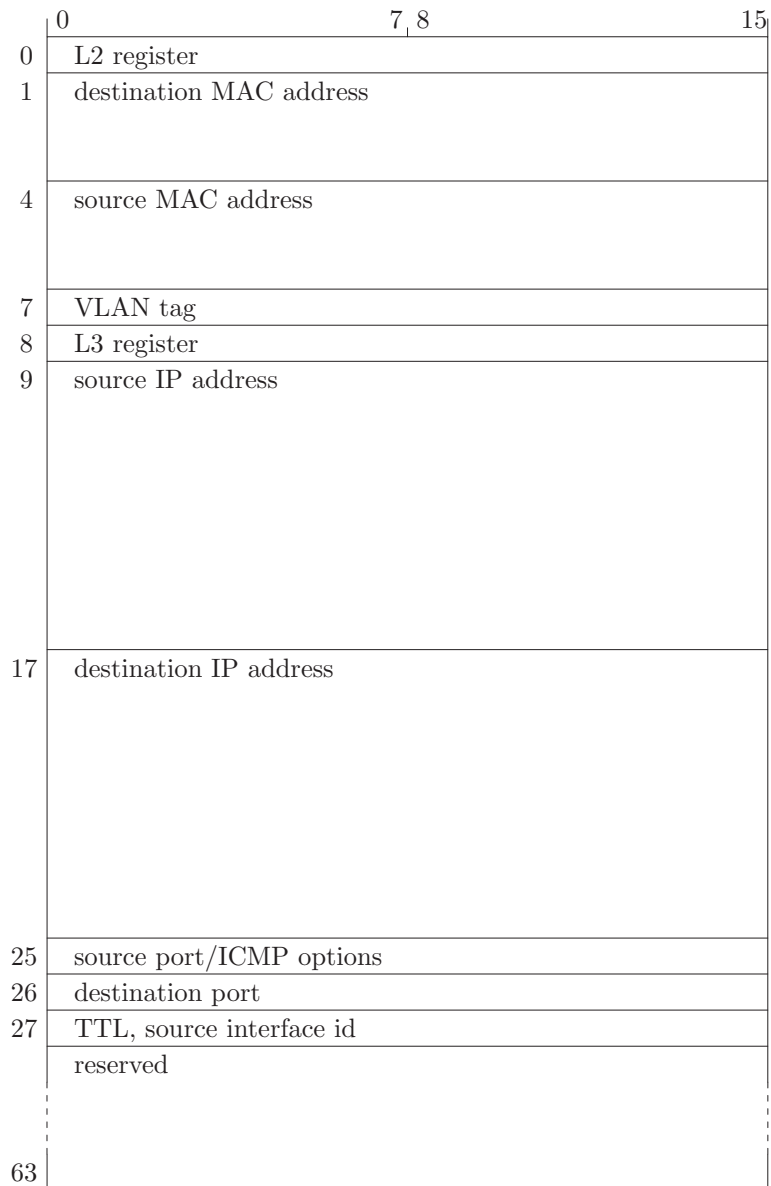
<sup>3</sup>It will be either Virtex-II or Virtex-II Pro; these chips are manufactured by Xilinx.

<sup>4</sup>Start of Packet

<sup>5</sup>End of Packet

<sup>6</sup>Cyclic Redundancy Check is used to verify data integrity.

Figure 3.2: Unified Header (UH)



UH is 16bit wide and contains 64 items. 444 of 1024 bits are currently effective. Note: UH format is still not final and is being optimized (especially order of items).

### 3.2.3 HFE Workspace

The program that performs packet analysis is stored in a block RAM inside the FPGA. The program is read-only and consists of 18-bit instructions. These instructions form a complete set of arithmetic, logic, I/O and control instructions. The instructions may access an array of internal 16-bit registers and the output UH. Input data as well as other specific registers (Accumulator, Loop Counter, Indirect Data Address, etc.) are mapped into the array of internal registers. This array is linear and supports random access. However, it is actually larger than the maximum directly addressable cell. The rest of the array can be addressed by means of indirect addressing. Any access to register 0 is processed as access to the register whose address is stored in register INDD (Indirect Data Address).

See files in directory `ipv6/hw/HdrXtr/` on the CD for further information on HFE. See file `ipv6/hw/HdrXtr/simulation/instruction_m.def` on the CD for definition of HFE instruction set in nanoprocessor assembler format (explained in Appendix A). Detailed information on HFE hardware implementation is in [9].

### 3.2.4 HFE Placement in the Architecture of the Router

Each Combo6 input interface is served by a single instance of the HFE. It means that there are four instances of HFE. Each instance of HFE communicates with a single instance of LUP. These parallel branches are independent except for the fact that they share the capacity of DRAM. This state is a development result. In the original design, there was only a single LUP. The new solution assures higher data throughput provided that there is enough space for four look-up processors in the FPGA.

The HFE communicates with the LUP via a FIFO of Unified Headers. Parallel to this FIFO, there are flags called *ready bits* denoting the status of individual UHs. If the ready bit is low it means that the UH may be filled by the HFE. If the ready bit is high it means that the UH is ready to be processed by the LUP. The HFE is allowed to set a ready bit from low to high and the LUP is allowed to set it from high to low.

### 3.2.5 L2\_strip

Originally, the L2 and L3 parsing was intended to be split among two devices: L2\_strip (L2 parsing) and HFE (L3 parsing). Both the L2\_strip and the HFE should have filled the UH. The problem of exclusive access arose and this solution was abandoned. HFE processor was chosen to take over L2 parsing job and perform all the required functionality.

The purpose of the L2\_strip mechanism was to:

- request a free UH,
- request a block in DRAM,
- perform L2 parsing,
- fill L2 part of UH,
- pass data to HFE and DRAM in parallel.

L2\_strip was projected as a state machine.

**Part II**  
**Simulation**

## Chapter 4

# VHDL Tools

If we consider simulation and development of a VHDL-based hardware design, the first logical step is to investigate available tools that deal with VHDL. There is a variety of tools on the market of different power and features. A survey of VHDL tools is available, see [2].

The basic VHDL design flow is shown in Figure 4.1. Working out a diagram prior to coding is a good start for every project. The coding and compilation follow. The compilation (or analysis) results in a library unit that enables simulation of the design. So far, this is the front-end of VHDL design flow. The back-end begins with synthesis. Synthesis tools take the VHDL code and convert it to logic elements. Implementation of a design consists of translation, mapping and place & routing. Translation and mapping assign logic elements of the design to the specific physical elements that actually implement logic functions in a device (i.e., an FPGA chip). The result is run through the place & route procedure. Placing is the process of putting logic from the design into physical cell locations in the FPGA. Routing assigns logical nets to physical wire segments in the FPGA that interconnect logic cells. The results of place & route contain both information for timing simulation and for hardware programming. Finally, a hardware programming file may be generated and loaded into the FPGA.

The major part of the development deals with simulation. Simulation tools allow the designer to logically debug the code early in the design process, and then test timing after synthesis, place and route. Inputs and outputs are usually simulated by means of text or data files.

### 4.1 VHDL Simulation Example

The most common form of VHDL simulation is a waveform. It is a chart that has time on the horizontal axis and values of different signals<sup>1</sup> on the vertical axis. These values change in time the same way they would change if the simulated design was loaded into the FPGA.

Figure 4.2 shows simulation of HFE that processes a packet parsing program (this program is shown in Figure 6.1). Because waveforms are very extensive, we can see only a fraction of the simulation—the very beginning of the program. On the other hand, we can clearly see how HFE allocates memory resources (UH, DRAM).

---

<sup>1</sup>Hardware equivalent of a signal is usually a wire or a bus (a set of wires).

Figure 4.1: VHDL design flow

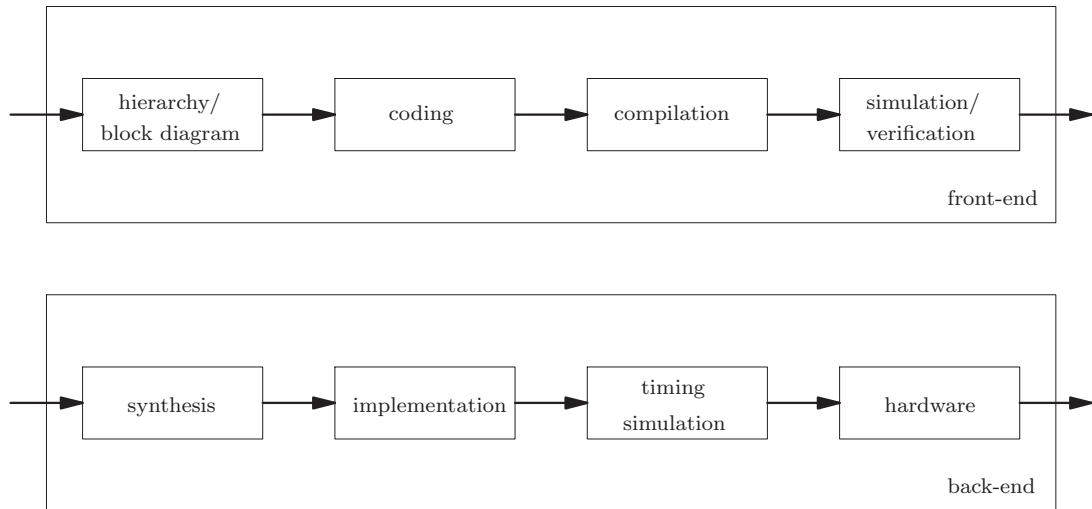
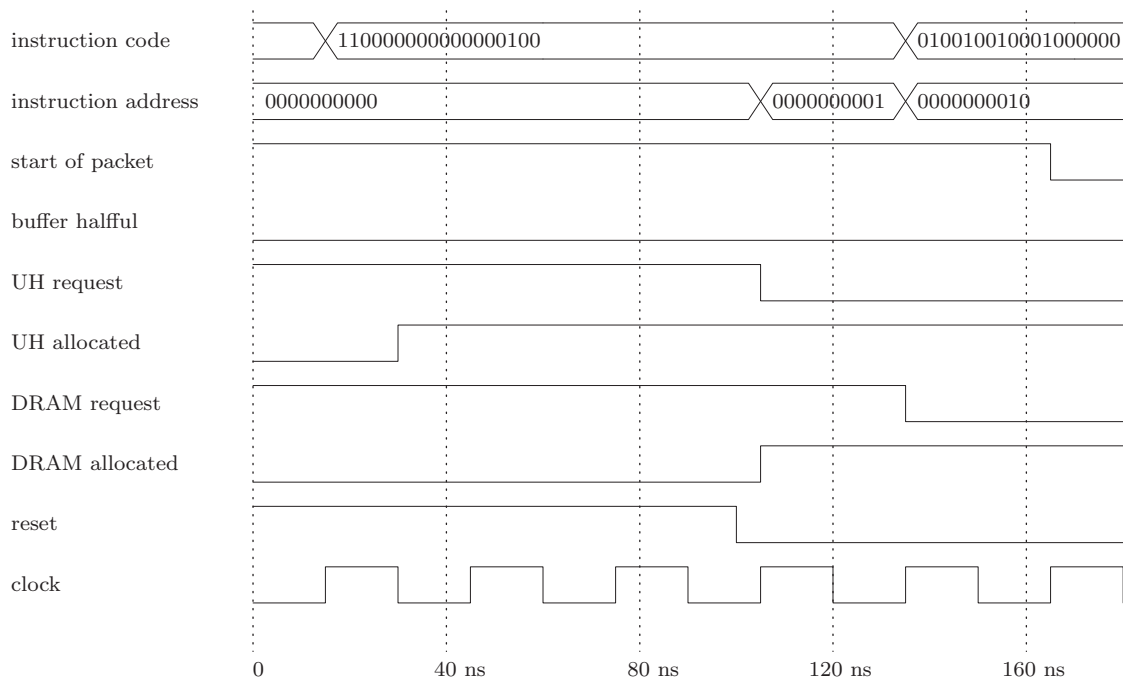


Figure 4.2: HFE is processing a packet parsing program—waveform



# Chapter 5

## L2\_strip

All the following programs are written in the C programming language and run on PCs with Linux or BSD operating systems.

### 5.1 L2\_strip Simulator

#### 5.1.1 Brief Introduction to the Simulator Development

L2\_strip simulator is a C program that provides the very same functionality as the projected hardware. Several versions of this simulator have been issued according to the changing hardware design—the hardware design and the simulator have been very close. Both the I/O definition and the functionality requirements have gone through changes. While the changes of I/O definitions lead from a very general layout to a precised interface, the functionality requirements had been gradually decreased<sup>1</sup>.

#### 5.1.2 Input and Output

The latest version of the L2\_strip simulator, `L2_strip_dump`, simulates the input using packet data saved by `tcpdump`. The output is the following:

<code>BRAM_out.txt</code>	text file containing the UH output
<code>Hdr_field.extract_out.bin</code>	binary file containing information to be passed on to the HFE (whole packet)
<i>console</i>	simulation progress, packet parsing results

---

<sup>1</sup>I.e., information validity checks (such as MAC addresses) had been left out in later versions supposing that this job would be taken over by the look-up unit. This concept has been accepted also by the HFE unit.

Figure 5.1: L2\_strip simulation example

```

$ ./L2_strip_dump -p -u data/ipv6.dump
UH_avail = 0
file header: 0xD4C3
etc.
file header: 0x0000
BEGINNING OF THE L2_STRIP SIMULATION
WAITING FOR THE START OF PACKET COMMAND
  packet header: 0xB1A7
  packet header: 0xCF3D
  packet header: 0x4750
  packet header: 0x0600
  caplen = 94 (size of captured portion of the packet)
  packet header: 0x5E00
  packet header: 0x0000
START OF PACKET COMMAND RECEIVED
[return]
Sending request to allocate new DRAM block for the incoming packet.
DRAM block for packet allocated, OK.
[return]
Sending request to allocate new Unified Header in block RAM.
[return]
Sending request to allocate new Unified Header in block RAM.
u [return]
UH_avail = 1
Sending request to allocate new Unified Header in block RAM.
Unified Header allocated, OK.
All necessary resources are ready - processing the packet:
e [return]
  DSTMAC: 0x00E0
  DSTMAC: 0x2974
  DSTMAC: 0x33CD
  SRCMAC: 0x0000
  SRCMAC: 0xC058
  SRCMAC: 0x8DDC
  LENGTH/TYPE = 0x86DD (conversion from NBO to HBO was necessary)
  TYPE interpretation
  setting Uh_ready...ok
  Packet body: 0x6000
etc.
  Packet body: 0xF9E6
END OF PACKET COMMAND RECEIVED
[return]
etc.

```

### 5.1.3 Usage

```
./L2_strip_dump input_file [options]
```

These options are supported:

- p pause each clock cycle
- e pause after end of packet
- q read to the end of file (default)
- b set/reset rxbufstatus0 bit (0 is default)
- d set/reset free DRAM available for packet (1 is default)
- s set/reset free UH available (1 is default)

Each time the program pauses, the user may continue by pressing return. Before pressing return, the user may press some of the keys **p**, **e**, **q**, **b**, **d**, **s** to modify the simulation environment. These keys have the same meaning as the `-key` option. The `rxbufstatus0` bit indicates if the buffer is at least half full when asserted 1.

### 5.1.4 Example

Figure 5.1 shows L2\_strip simulation example; unproportional font is used for **user input**, regular font for program output and italic font for *comments*.

### 5.1.5 The Benefits

Although the L2\_strip machine has been left out from the hardware design, the simulator is useful. Its benefits include:

- formalization<sup>2</sup> and implementation of algorithm that will be a part of the router regardless the performing device,
- implementation of a simple `tcpdump` file format parser<sup>3</sup>,
- easy packet content viewing.

## 5.2 Parallel Simulator

The latest version of the L2\_strip simulator is capable of running in a parallel environment utilizing the engine of the Parallel Simulator. This engine uses shared memory to run and synchronize several instances of the L2\_strip simulator and a instance of the look-up processor simulator [5].

The current version of the Parallel simulator runs N instances of the L2\_strip simulator and a single instance of the LUP simulator. If there are N instances of the L2\_strip simulator, they are assigned process numbers from 0 to N-1 and the LUP simulator is assigned process number N.

---

<sup>2</sup>Formal verification group has checked the algorithm according to the C code of the simulator and its flow diagram.

<sup>3</sup>This part of the L2\_strip simulator has already been used in the HFE simulator.

Figure 5.2: Parallel Simulator use example

```

[xhofer@cabernet L2_strip]$ ./simulator data/ipv6.dump
Shared memory has been got.
Shared memory has been attached.
Shared memory has been initialized.
cycle 0
  process 0: BEGINNING OF THE L2_STRIP SIMULATION
  process 1: Look-up: Searching Ready block
cycle 1
  process 1: Look-up: Searching Ready block
cycle 2
  process 0: processing packet 0
  process 1: Look-up: Searching Ready block
etc.
cycle 8
  process 1: Look-up: Searching Ready block
cycle 9
  process 0: Uh_ready set
  process 1: Look-up: Ready block found
cycle 10
  process 0: reading packet body
  process 1: Look-up: Dst MAC 57344
cycle 11
  process 0: reading packet body
  process 1: Look-up: Dst MAC 29737
cycle 12
  process 0: reading packet body
  process 1: Look-up: Dst MAC 52531
cycle 13
  process 0: reading packet body
  process 1: Look-up: Src MAC 0
cycle 14
  process 0: reading packet body
  process 1: Look-up: Src MAC 22720
cycle 15
  process 0: reading packet body
  process 1: Look-up: Src MAC 56461
cycle 16
  process 0: reading packet body
  process 1: Look-up: Starting matching
cycle 17
  process 0: reading packet body
  process 1: Look-up: Processing CAM instruction 0
cycle 18
  process 0: reading packet body
cycle 19
  process 0: reading packet body
cycle 20
  process 0: reading packet body
  process 1: Look-up: Match in CAM, line 23
cycle 21
  process 0: reading packet body
  process 1: Look-up: Processing CMP instruction 92
etc.

```

### 5.2.1 Input

The input of the Parallel Simulator is the sum of inputs of all parallel processes that are running within its engine. Since each instance of the `L2_strip_dump` simulator has one file on input and the LUP simulator has no external input<sup>4</sup>, the inputs of the Parallel Simulator are N files with packet data saved by `tcpdump`.

### 5.2.2 Output

```
par_output0 ... par_outputN  simulation reports
console                       immediate simulation status
```

Again, the output of the Parallel Simulator is the sum of outputs of all parallel processes that are running within its engine. The output of the `L2_strip` simulator running within the engine of the Parallel simulator differs from outputs from stand-alone simulation. In the current implementation, each simulator (either `L2_strip` or LUP) produces the file `par_outputX` containing simulation statistics for process number X.

### 5.2.3 Usage

```
./simulator input_file0 ... input_fileM5 [-s [sleep_time]]

-s          forces slow mode
sleep_time specifies how long should the simulator wait between clock cycles
           (in  $\mu$ s; 1,000,000 by default)
```

The simulator is running as long as there is some data to be processed, then the simulation is finished and all parallel processes are automatically shut down. The user may quit the simulation any time by pressing Ctrl+C. This signal is trapped enabling smart finish of all processes. Reports are properly generated in either case.

### 5.2.4 Example

Figure 5.2 shows Parallel Simulator use example; unproportional font is used for `user input`, regular font for program output and italic font for *comments*. Note that “process 0” is the `L2_strip_dump` simulator and “process 1” is the LUP simulator.

### 5.2.5 Implementation Details

#### The Shared Memory

The main process is forked so that each simulator runs in its own process. These processes communicate by means of shared memory. Its structure is shown in Table 5.1, the literals used are explained in Table 5.2.

Note that the `nInterfaces` constant equals to N from the previous text and the `nClocks` constant equals to N+1. All array type elements (quantity higher than one) are indexed by process numbers and their values correspond to respective processes<sup>6</sup>.

<sup>4</sup>It uses only outputs of the `L2_strip` simulator. These are passed via the shared memory.

<sup>5</sup>M equals to N-1.

<sup>6</sup>If speaking of elements with `nInterfaces` quantity, this is true only for `L2_strip` processes while the LUP simulator accesses the whole array.

Table 5.1: Shared memory structure

Element	type	quantity	purpose/description
ClockElement	BIT	nClocks	synchronization of processes (clock bits)
ReadyElement	BIT	nInterfaces * nSORs	indication of UH ready status (ready bits)
UhElement	UNIFIED_HEADER	nInterfaces * nSORs	UH
MessageElement	char[256]	nClocks	text strings to be printed on console
SleepElement	unsigned	1	sleep time in $\mu$ s
QuitElement	BIT	nClocks	indicates whether the process has no data on input and is ready to finish (HIGH) or not (LOW)
QuitElement	BIT	nClocks	when asserted HIGH the corresponding process has to finalize

Table 5.2: Description of used literals

literal	value type	description
BIT	LOW/HIGH	hardware-like type (logic zero or logic one)
nClocks	integer	total number of parallel processes
nInterfaces	integer	number of interfaces which pass data to a single look-up processor (equals to number of L2_strip simulator instances)
nSORs	integer	number of Sets of Registers (memory locations for UHs) that are filled by a single HFE

### Interprocess Synchronization

Initially, the shared memory contents are set to zero (BIT values are initialized to LOW respectively).

At the beginning of a simulation loop, the Parallel Simulator checks whether the simulation should end (all processes have no job<sup>7</sup> and there is no pending interprocess communication). If so, all processes are sent a request to finalize<sup>8</sup> and the simulation ends. Otherwise, each process is sent a clock cycle<sup>9</sup>. The process is supposed to do the work that the hardware can do in a single clock cycle and announce that it waits for the next cycle<sup>10</sup>. Once all the processes have finished the clock cycle, the Parallel Simulator prints out the messages saved in the shared memory<sup>11</sup>, erases them and then the loop repeats.

### Interprocess Communication

The interprocess communication models the communication within the hardware. In this section we will use the term *packet parsing unit* instead of L2\_strip because the same holds for both the L2\_strip and the HFE.

<sup>7</sup>This is signaled by means of the first QuitElement array. All values have to be set HIGH.

<sup>8</sup>This is signaled by means of the second QuitElement array. All values are set HIGH.

<sup>9</sup>This is signaled by means of the ClockElement array. The clock bit (*shared memory pointer*) + *sizeof(BIT) \* (process number)* is set HIGH.

<sup>10</sup>This is signaled by means of the ClockElement array. The clock bit (*shared memory pointer*) + *sizeof(BIT) \* (process number)* is set LOW.

<sup>11</sup>These text messages are printed by individual processes into the MessageElement array

When a packet parsing unit is about to parse a packet, it must find a free UH. The UH allocation is expressed by ready bits<sup>12</sup>. The correspondence between the ready bits and the UHs is the following:

```
ready_pointer = (BIT *) ((u_int_8 *) shm_pointer +
+ nClocks * ClockElement + (proc_num * nSORS + i) *
* ReadyElement)

Uh_pointer = (UNIFIED_HEADER *) ((u_int_8 *) shm_pointer +
+ nClocks * ClockElement + nInterfaces * nSORS *
* ReadyElement + (proc_num * nSORS + i) * UhElement)
```

where the range of *i* is from 0 to (nSORS-1).

If the ready bit is LOW it means that the UH may be filled by the packet parsing unit. If the ready bit is HIGH it means that the UH is ready to be processed by the LUP. The packet parsing unit is allowed to set a ready bit from LOW to HIGH and the LUP is allowed to set it from HIGH to LOW.

## 5.2.6 Evaluation

There were many obstacles in the parallel environment such as difficult debugging and the need for very accurate cooperation. The speed of the parallel simulation was an issue, too. The speed was initially very poor due to the use of active waiting. The speed has been rapidly improved by the use of the system function `usleep`<sup>13</sup>. If the speed becomes critical factor in the future, another means of interprocess synchronization will have to be used (i.e., pipes).

The Parallel Simulator provides the following benefits:

- successful attempt to simulate two hardware blocks in parallel,
- successful testing of the communication interface of the hardware blocks,
- possibility to simulate LUP on real data.

---

<sup>12</sup>These bits are stored in the ReadyElement array.

<sup>13</sup>sleep some number of microseconds

## Chapter 6

# Nanoprocessor Simulator

The previous tools simulated hardware behaviour by implementing the same functions in software. The Nanoprocessor Simulator chooses a different approach. It is based upon the fact that the simulated hardware is a processor and helps to fully utilize its power.

The design of the Nanoprocessor Simulator includes specification of the *nanoprocessor assembler*—it is a programming language for nanoprocessors. The tool itself is a generic compiler, interpreter and debugger—it is able to simulate different nanoprocessors. It is a cross-platform development tool. Nanoprograms are written and debugged on a PC and the resulting binary is executed by a nanoprocessor inside the router or possibly another device. The nanoprocessor assembler is based upon conventional assembler syntax, is enhanced by C-style directives and features embedded user C code. This C code provides means how to implement semantics of instructions, custom I/O simulation and auxiliary definitions. The full specification of the nanoprocessor assembler is in Appendix A.

The tool runs on PCs with Linux or BSD operating systems.

### 6.1 Development History

Originally, this tool was supposed to simulate HFE by means of interpreting its instructions. During the design of the tool, the decision was made that repeated HFE nanoprogram text parsing was not desirable. That is why a suitable structure to transform the text to had to be found. The binary form of the program was a logical choice. The tool became both interpreter and compiler.

However, the instruction set is modified during the development. If it is fixed in the tool, the tool must be modified each time the instruction set is changed. In order to avoid this, at least for the purpose of compiling, a basic instruction set definition was designed and implemented. It covered machine code format thus the tool became a generic compiler.

New features were added, such as conditional compiling and other directives. The semantics of instructions was put into a standalone file in order to separate it from the other code. The file was simply included into the source code of the interpreter. This solution improved accessibility of the semantics reducing the risk of accidental change of the other code of the tool.

Finally, the idea of generating the entire source code of the interpreter was realized. The semantics of instructions was incorporated into the instruction set definition. That was the step leading to a generic interpreter.

## 6.2 Architecture of the Tool

There are four main distinguished modules:

- preprocessor,
- compiler,
- interpreter,
- debugger.

The preprocessor performs basic operations on the code so that it could be easily processed by the compiler. These operations include mainly handling directives, labels, macroexpansion and compilation conditions. The result is an expanded code that includes only items to be compiled (any extra information is left out).

The compiler sequentially processes the expanded code. It parses instructions and their parameters and produces the binary code.

The interpreter carries out the simulation. First, it matches the binary codes of instructions in the compiled code to their semantics. Then it enters simulation loop and interprets the instructions. This is the basic principle. The performed actions depend on the instruction set definitions, especially on the semantics.

The debugger provides the user front-end. It is described in Section 6.6.

See Section 6.8 for implementation details.

## 6.3 Input

The compulsory input of the tool is a nanoprocesor assembler program. If the debugger is active, debugger commands are read from the standard input. The commands are described in Section 6.6.

Additionally, the interpreter may have its own inputs that are processed by user C code.

## 6.4 Output

The basic output is the compiled program. The user may choose any of the following formats:

- binary,
- text output in binary numbers,
- text output in hexadecimal numbers.

Additionally, the interpreter is allowed to produce its own outputs by means of user C code.

## 6.5 Usage

```
./nsim input_file [-q] [-v] [-o output_file] [-b] [-h]
                  [-c | -i [interpreter options] ]

-q                quiet parser
-v                print program version
-o output_file    compiled output
-b                text output in binary numbers
-h                text output in hexadecimal numbers
-c                compile only (default)
-i                run interpreter and debugger after compilation
```

The default output is a 64-bit binary. If no output file is specified, the console is used for the text output and the binary output is discarded.

The interpreter options are defined and parsed by user C code.

## 6.6 The Debugger

The debugger commands are inspired by `gdb`<sup>1</sup>. A tabular list of implemented commands can be obtained by the `help` command. Commands are case-insensitive.

The debugger uses the same parser as the compiler. That is why the syntax of the debugger commands follows the same rules as the nanoprocessor assembler instructions. Moreover, the same macroexpansion is applied allowing the user to utilize any macros that have been defined in the source code. For instance, memory locations may be accessed by name.

## 6.7 The Use of the Tool

HFE nanoprograms have been compiled and debugged utilizing the tool. A HFE instruction set definition is available that utilizes full power of the tool defining instruction format and semantics as well as I/O simulation.

Further, nanoprocessor assembler form of LUP instruction set definition is also available and has been used to compile LUP nanoprograms. This definition does not include embedded user C code (semantics, I/O simulation) yet, this part is optional.

The tool is expected to be utilized within the design of other nanoprocessors, too.

### 6.7.1 Examples

An example follows, Figure 6.1 shows a HFE nanoprogram and Figure 6.2 shows its interpretation. Unproportional font is used for `user input`, regular font for program output and italic font for *comments*.

The program was written by Štěpán Friedl and it performs Ethernet and base IPv6 headers parsing. The first two lines of the program are comments. Note that comments begin with a semicolon. The first part of the program consists of directives. Foremost, memory locations and instruction definitions are included. Then definitions of some macros follow. Any occurrence of the left side of the macro in the program is automatically expanded into the right side.

<sup>1</sup>UNIX command-line debugger for programs written in C, C++, and Modula-2 programming languages.

Figure 6.1: HFE nanoprogram example

```

; HFE packet processing - Ethernet and base IPv6 headers
; $Id: test2.hfe,v 1.3 2003/04/11 14:11:43 xfried00 Exp $
#include "../simulation/memory.def"
#include "../simulation/instruction.m.def"
#define MAC_ADDR_LENGTH    0x48
#define L2_IPv6_TYPE       0x49
#define L2_IPv4_TYPE       0x4A
#define L2_ARP_TYPE        0x4B
#define L2_VLAN_TYPE       0x4C
#define IPv6_ADDR_LENGTH   0x4D
#define L2_REG              0x60           ; L2 attribute register for UH
#define L3_REG              0x61           ; L3 attribute register for UH
#define NH_HL               0x62           ; Next Header/Hop Limit
#define DATA_LENGTH        0x63
#define SOP 1,PSTATUS
; UH fields addresses
#define UH_L2                0x00
#define UH_MAC               0x01
#define UH_VLAN              0x07
#define UH_L3                0x08
#define UH_SRC_IP            0x09
#define VLAN_PRESENT         14,L3_REG
init:
    movc    4
    movr   MAC_ADDR_LENGTH,ACC
    movc   0x86dd
    movr   L2_IPv6_TYPE,ACC
    movc   0x0800
    movr   L2_IPv4_TYPE,ACC
    movc   0x0806
    movr   L2_ARP_TYPE,ACC
    movc   0x8100
    movr   L2_VLAN_TYPE,ACC
    movc   14
    movr   IPv6_ADDR_LENGTH,ACC
start:
    clr    L2_REG
    clr    L3_REG
    movr   LC,MAC_ADDR_LENGTH
mac_addr:
    out    UH_MAC,DIN           ; read data from input
mac_addr.i:
    outi   DIN
    loop   mac_addr.i          ; repeat 5 times
    movr   ACC,DIN             ; type/length field
    cmp    L2_IPv6_TYPE        ; test this field
    jmpz   ipv6
    cmp    L2_VLAN_TYPE
    jmpz   vlan
; .
; .   Other L2 types
; .
    jmp    l2_unknown          ; unknown type
etc.

```

The `init` label denotes the beginning of the initialization part of the program. Initialization is performed by pairs of instructions `movc` and `movr`. The first instruction loads a constant into the accumulator and the second one copies the constant from accumulator to another register. This way, the memory content is initialized. The `start` label denotes the beginning of packet parsing. L2 and L3 attribute registers are set to zero and then the loop counter is initialized so that a loop may be used for the reading of MAC addresses. The `mac_addr` label is followed by `out` instruction. In this case, the instruction reads the first 16-bit word of the destination MAC address and stores it in the UH. The `mac_addr_i` label denotes the beginning of a loop. It is a repeat-until loop and its body contains only a single instruction. It is the `outi` instruction. This instruction increases data output address and then it writes the contents of the given register at the new data output address. Since loop counter contains number four in the beginning, the cycle repeats five times decreasing the loop counter each time the program reaches `loop` instruction. If the loop counter is zero, the cycle finishes. This way, the program reads both MAC addresses in six 16-bit words. The `cmp` instruction follows. This instruction compares the content of the accumulator to the content of a given register. If they equal, it sets zero flag to one, otherwise the zero flag is set to zero. The next instruction is `jmpz`, a conditional jump. If the zero flag is set, the program jumps to the given place (here it is specified by a label), otherwise it continues with the next instruction. Two analogous instructions follow. Just for completeness, `jmp` is a unconditional jump (the jump is always performed). The rest of the program uses similar means to perform the remaining portion of the analysis. Its full copy is on the CD: `ipv6/hw/HdrXtr/hfe-progs/test2.hfe`.

If we look at the simulation example, we should foremost explain the command that executed the simulator. The first parameter is the program to be processed and the second parameter forces quiet compilation. The second line of the command tells that the interpreter will be used and the rest of the line is passed to the interpreter. In this case, the parameter specifies the file containing input packets. The program advertises and the first prompt occurs along with the program line to be interpreted. The user types `break start` to set a breakpoint. The breakpoint is set at the instruction address that follows the label `start`. It is instruction `clr` at line 45. Then the user enters `run`. The simulation is executed and continues until a breakpoint or program end is reached. The program stops at line 45. The user does not want the passed source lines to be displayed anymore and thus types `silent`. After entering the `continue` command, the simulation goes on again. When it stops, the user types commands `printout UH.L2` and `printout 1,3` to view the results of the program. In our case, the UH contents are viewed. Please refer to Figure 3.2 to check that the data matches to the L2 attribute register and destination MAC address respectively.

## 6.8 Implementation Details

The core of the program is written in the ANSI C programming language. However, the binary form of the program is not intended to be run directly. A UNIX shell script that performs necessary calls is utilized. This solution was chosen because of the fact that some parts of the program (the interpreter and the debugger) are generated by the program itself.

Figure 6.2: HFE simulation example

```

$ ./nsim ../../../../hw/HdrXtr/hfe-progs/test2.hfe -q \
> -i -d ../../../../hw/L2_strip/data/ipv6.dump
Nanoprocessor interpreter
Beginning of program:
31  >                movc                4
(nsim) break start
Breakpoint set at line 45. (instruction address 12)
(nsim) run
32                movr                MAC_ADDR_LENGTH,ACC
33                move                0x86dd
34                movr                L2_IPv6_TYPE,ACC
35                move                0x0800
36                movr                L2_IPv4_TYPE,ACC
37                move                0x0806
38                movr                L2_ARP_TYPE,ACC
39                move                0x8100
40                movr                L2_VLAN_TYPE,ACC
41                move                14
42                movr                IPv6_ADDR_LENGTH,ACC
45  >                clr                L2_REG
(nsim) silent
(nsim) continue
(nsim) printout UH_L2
data output address 0x00 stores value 0x0000 (0).
(nsim) printout 1,3
data output address    value
0x01                  0x00e0 (224)
0x02                  0x2974 (10612)
0x03                  0x33cd (13261)
(nsim) quit

```

Figure 6.3: Nanoprocessor simulator script—`nsim`

```

if test ! -x nsim_bin; then
    echo "Please wait while the nsim binary (nsim_bin) is compiled..."
    make > nsim.log
    if test ! $? -eq 0; then
        echo "Error occurred!"
        exit
    fi
    echo "Done."
fi
./nsim_bin -1 $@
if test $? -eq 0; then
    make all > nsim.log
    if test $? -eq 0; then
        ./nsim_bin -2 $@
    else
        rm .interpreter
    fi
fi

```

The script is shown in Figure 6.3. The first `if` branch is executed only if the binary `nsim_bin` does not exist. The only purpose is to be as transparent to the user as possible. If the binary does not exist, it is made for the user and the script continues. The next step is to call `nsim_bin` and pass it the parameters that were entered from the command line plus the `-1` option. This option announces the program that it is being called by the script and that it is the first call. Note that it is better to pass this option as the first parameter rather than the last or anything between. If the option was anticipated by `-i` option, it would be passed to the interpreter unwillingly. During this call, the `nsim_bin` attempts to generate interpreter source code. Provided that `nsim_bin` succeeds (its return value is zero), the script continues. If it does not, it usually means that there was an error on input. These errors are reported so that the user can easily fix them. The next command of the script calls the UNIX `make` utility. During the `make`, the `gcc` (GNU C compiler) is called to compile the interpreter, the debugger and to link them to the main program. If the `make` succeeds, the `nsim_bin` is called for the second time. This call is similar to the first one. Otherwise, the script erases the configuration file `.interpreter` that has been generated along with interpreter source code. This denotes the fact that the interpreter source is not valid.

The `-1` and `-2` options modify the only global variable `run` that determines program behaviour. By default, this value is zero and corresponds to running the `nsim_bin` directly from the command line. The script calls `nsim_bin` with either `-1` or `-2` option assigning `run` value 1 or 2 respectively. The description of behaviour of the `nsim_bin` in dependence on the `run` value follows.

- `run = 0`
  1. The source code of the given nanoprogram is preprocessed. Any embedded C code is ignored;
  2. The preprocessed code is compiled;
  3. The compiled program is written on output;
  4. If the `-i` option is present, the interpreter is called. Note that this interpreter may be incompatible with the given code;
  5. The debugger is called by the interpreter before interpreting every single instruction.
- `run = 1`
  1. If the `-i` option is not present (the program will be compiled only thus an interpreter is not needed), the `nsim.bin` exits with zero return value. Otherwise, it continues:
  2. The source code of the nanoprogram is preprocessed. Embedded C code is processed and utilized to generate interpreter source code;
  3. Interpreter configuration file is generated;
  4. Once the `nsim.bin` successfully finishes all above tasks, it exits with zero return value.
- `run = 2`
  1. The source code of the nanoprogram is preprocessed. Any embedded C code is ignored;
  2. The preprocessed code is compiled;
  3. The compiled program is written on output;
  4. If the `-i` option is present, the interpreter is called;
  5. The debugger is called by the interpreter before interpreting every single instruction.

### 6.8.1 The Preprocessor

The preprocessor is the most sophisticated part of the tool. It performs the following sequence of actions:

1. The source code of the given nanoprogram is loaded. Note that the number of lines and their length are not limited. This procedure includes a call of tabs to spaces converter and a call of trailing white characters remover;
2. Concatenated lines are joined;
3. The code is trimmed. Letters are converted to uppercase if not in a string or embedded C code, trailing white characters and comments are removed;

4. Each line is processed by this sequence of actions:
  - (a) The line is macroexpanded if appropriate (i.e., macro definitions are not expanded);
  - (b) If it is a directive or label it is processed. If it is an instruction it is scheduled to be processed by the compiler. The processing of directives includes instruction set definition parsing, embedded C code handling and interpreter code generating. There is an exception for those lines that are removed by means of directives of conditional compilation.

### Interpreter and Debugger Generation

The interpreter source is generated on the base of C code that is embedded in the nanoprocesor assembler (see Appendix A). Embedded user C code is supported both in self-contained definitions and in instruction definitions (this C code specifies instruction semantics). The interpreter source is generated in two steps. First, a sequence of files is generated (done by the preprocessor), then these files are concatenated (done by `make`). The sequence of files is following:

1. file of directives that are stated in self-contained definitions (except for `#include`),
2. file of `#include` directives that are stated in self-contained definitions,
3. preprogrammed file containing common directives and the beginning of interpret function,
4. file containing definitions of variables generated from instruction definitions, definitions of variables and commands from self-contained definitions,
5. preprogrammed file containing the initial part of the interpreter and the beginning of simulation loop,
6. the core of simulation loop generated from instruction definitions (instruction semantics),
7. preprogrammed file containing the end of simulation loop and the rest of interpret function.

The first file is included to debugger source, too. Thanks to this, the debugger may be customized for a certain nanoprocesor (see Section A.2.1).

## 6.9 Cross-platform Compatibility Issues

From the very beginning, there was an effort to design a program that could run under different platforms. The choice of the ANSI C programming language brings quite a wide possibilities in this.

The original non-generic version of the Nanoprocesor Simulator worked well under UNIX systems as well as under Microsoft Windows. The current version is closely bound with UNIX shell interpreter, `gcc` compiler and `make` utility. That is why it does not run under standard installations of Windows. However, a UNIX emulator may be utilized. The Nanoprocesor

Simulator is successfully tested under *cygwin*, a UNIX emulator that is freely downloadable from <http://cygwin.com>.

The most serious incompatibilities between different UNIX platforms that have been encountered concern the `make` utility. The `Makefile` must have been updated so that there are more explicit rules. It is tested under Linux, NetBSD and FreeBSD.

## 6.10 Evaluation

The Nanoprocessor Simulator is capable of simulation of different nanoprocessors. These nanoprocessors may differ in many aspects, especially in the instruction set. The design of the simulator includes means how to define a instruction set—they are a part of the nanoprocessor assembler that is documented in Appendix A. We can emphasize that the semantics of instructions is written in the C programming language that is well-known, accepted and supported<sup>2</sup>. The semantics is a subset of Nanoprocessor Simulator input, it is not a part of the tool itself. A nanoprogram may be bound with according instruction set by a single directive. Since the linkage to the instruction set is handled inside nanoprograms, the tool is truly generic.

The fact that the tool generates and compiles some parts of itself brings interesting points. On one hand, the tool is dependent on `gcc` compiler and `make` utility. On the other hand, speed factors must be taken into account. It is clear that the compilation of interpreter and debugger slows down simulator loading. However, it takes usually a second or less, i.e., simulation of an instruction set with 40 instructions requires compilation of about 40 kilobytes of C code. Once the interpreter is compiled, it runs as fast as if it was a simulator dedicated to a specific instruction set. Moreover, the interpreter works with compiled assembler code, not with mnemonics; precomputed indexes replace repeated bitwise masking and matching when fetching an instruction. These techniques lead to a high-speed simulation.

## 6.11 Future Plans and Improvements

The Combo6 card is a routing accelerator. It means that it is intended to keep a vast majority of the work in hardware. However, a small subset of extraordinary packets will have to be handled by software anyway. There is an idea that the PC software could use nanoprograms interpreted by a simulator instead of standard PC programs. These nanoprograms could be extended versions<sup>3</sup> of those that run in hardware. The same principle could be used if a part of the router does not function. The speed of the Nanoprocessor Simulator is considerable and that is why it could be utilized this way.

If the user makes an error in the embedded C code, the interpreter cannot be compiled. In this case, `gcc` returns error messages concerning the generated code, not the source itself. It may be hard to orientate in this, that is why a solution is prepared. There will be a comment for each generated line of code. This comment will denote the location from which the code comes. If an error occurs, these comments will be used for translating `gcc` error messages so that they would directly point to the source.

---

<sup>2</sup>In contrary to specific languages for instruction set definition, such as Instruction Set Definition Language (ISDL), many programmers are familiar with the C programming language.

<sup>3</sup>Size of nanoprograms that may be loaded into hardware is limited.

## Chapter 7

# Conclusion

Part I attempts to provide a transparent look at the Internet protocol, packets and their structure. Further, a router concept is introduced with an emphasis on packet parsing blocks. In conjunction, the new term *nanoprocessor* is presented; it is a label for an FPGA based processor. The relation between microprograms and *nanoprograms* is explained letting us recognize a specific development platform.

Efficient hardware development requires use of suitable software tools. That is the topic of Part II. Essential software for development of the designed router is described giving an idea what possibilities are offered by this basic tool chain (VHDL tools). Further, tools that aim to enhance hardware development efficiency are introduced. They may be utilized either standalone or together with VHDL tools. An example of the second case is at hand—the binary that is utilized in the example of VHDL simulation was produced by the Nanoprocessor Simulator.

The set of examples in Part II creates a survey of different simulation forms. Now it remains to note that the same packet is parsed in all examples. This point of view lets us clearly see the difference between individual tools. The simulators that have been developed in scope of the Liberouter project are described in detail summarizing the benefits of each one in the end of the corresponding section.

A significant part of the thesis is devoted to the Nanoprocessor Simulator. It is a development tool for the class of processors. Design of this tool is based on experience with the development of previous simulators and discussions with hardware designers. It lead to issuing nanoprocessor assembler—a programming language that is derived from a conventional assembler and the C programming language. An emphasis has been put both on ease of use and on processing efficiency. The design is robust enough to allow retargetability—it is not dedicated to a processor or instruction set. That is why the tool is suitable for development of programs for platforms that are also under construction. Although there are improvements left to be done, the tool has been successfully utilized within the development of packet parsing blocks of the router. The development of the tool still continues. Some ideas have been already mentioned, other goals include a linker, arithmetic expansion, instruction set consistence checking<sup>1</sup> and more. The development of the Nanoprocessor Simulator has become a basis for the work of Tomáš Rybka, who develops a generic cross-assembler for his master thesis.

---

<sup>1</sup>It will test whether operation codes of all instructions are properly distinguished.

# Bibliography

- [1] Alena Kabelová, Libor Dostálek a kolektiv. *Velký průvodce protokoly TCP/IP a systémem DNS*. Computer Press, Hornocholupická 22, Praha, The Czech Republic, 3rd edition, 2002.
- [2] Filip Höfer and Kateřina Minaříková. VHDL Tools. Technical report, CESNET, 2002.
- [3] Internet Assigned Numbers Authority (IANA). Protocol numbers. Up-to-date list is available at <http://www.iana.org/assignments/protocol-numbers>.
- [4] PC based IPv6 Router. <http://www.liberrouter.org>.
- [5] Kateřina Minaříková. Simulation of look-up processor of IPv6 router, 2003.
- [6] The Institute of Electrical and Inc. Electronics Engineers. *IEEE Std 802.1Q*, pages 67–69. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017-2394, USA, 1998.
- [7] The Institute of Electrical and Inc. Electronics Engineers. *IEEE Std 802.3*, chapter 1.4.261, pages 38–43, 46–47, 58–59, 62, 81, 976. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2000.
- [8] Pavel Satrapa. *IPv6*. Neocortex, Na Rovnosti 2245/3, Praha, The Czech Republic, 2002.
- [9] Štěpán Friedl. Hardwarový analyzátor paketů pro směrovač IPv6, 2003.

# Appendix A

## Nanoprocessor Assembler

Nanoprocessor assembler (NA) is a low level programming language for nanoprocessors. This is the specification of the language. We will call any text that conforms to this specification a *nanoprogram*. A line of a nanoprogram will be called a *source line*. Tabs and spaces will be called *white characters*.

NA is a case-insensitive context-free language.

### A.1 Basic Elements

NA distinguishes these basic elements:

- directive,
- instruction,
- label,
- empty.

A single basic element is allowed to spread over several source lines. This is achieved by a backslash (\) at the end of source line that continues at the next source line. A single source line may contain at most one element. These elements may be placed in any order unless otherwise stated. An example is shown in Figure A.1.

An empty element consists of zero or more white characters; the only purpose of this element is format of the nanoprogram. Other elements are described in the following sections.

Figure A.1: Basic elements

```
#directive
label1:
        instruction operand1,\
        operand2,\
        operand3
label2:
```

Figure A.2: Comments

```

;this is an empty line
instruction ; this instruction has no operands

```

### A.1.1 Comments

Any element may be appended by a comment. A comment begins with a semicolon (;). An example is shown in Figure A.2.

### A.1.2 Identifiers

The first character of a valid identifier is alphabetical or an underscore. Other characters are alphabetical, numerical or underscore. The minimal length of an identifier is one, the maximal length is not limited. Note that NA is case-insensitive.

Identifiers must be unique unless otherwise stated.

### A.1.3 Strings

String is a sequence of printable characters except for double quote (") that is delimited by double quotation marks, i.e., "string", "also a string despite of characters ./@:#".

A string is treated as a single case-sensitive entity.

### A.1.4 Numbers

NA supports decimal and hexadecimal numbers. Decimal numbers are written in a normal way, hexadecimal numbers begin with 0x; i.e., 30 equals 0x1E.

## A.2 Directives

A directive begins with a sharp (#). Supported directives are enlisted in alphabetical order.

### A.2.1 #define

This directive provides three kinds of definitions.

#### Macro Definitions

Syntax:

```
#define macro1_name right side
```

or

```
#define macro2_name
```

The name of a macro must be a valid identifier (see Section A.1.2).

In the first case, the name of the macro is followed by a sequence of white characters and the right side. The right side can be any sequence of printable characters that starts with

a non-white character and contains even (or zero) number of double quotes—it means that there is no unterminated string. Any occurrence of the left side of the macro in the nanoprogram is automatically expanded into the right side provided that the name of the macro is delimited by one of these separators: white character, comma (,), colon (:), semicolon (;), sharp(#), backslash (\), any brace ({, }, [, ], (, )) or arithmetic character (+, -, \*, /, >, <, =, ^, |, &) and it is not inside a string.

In the second case, everything is the same except for the fact that the right side of the macro is empty. These macros are useful for conditional compilation.

The definition of a macro must anticipate its use.

### Embedded User C Code

Syntax:

```
#define {embedded user C code}
```

The user code may be any valid C code except for functions. It means that it can be:

- C directive,
- definition of C variable(s),
- C command(s).

Note that definitions of variables should forego commands. The purpose of this code is to define a custom interpreter. The definitions of C variables and C directives allow the user to specify the entities that can be used within instruction definitions. The commands may be used for any required initialization of the interpreter, i.e., processing of parameters.

The embedded user C code may contain these automatic variables:

<code>ip</code>	instruction pointer ( <b>unsigned</b> )
<code>code1</code>	code of interpreted instruction ( <b>unsigned long long</b> )
<code>force_finish</code>	assign 1 to force end of interpretation ( <b>int</b> )
<code>current_line</code>	line in assembler source ( <b>unsigned</b> )
<code>current_file</code>	assembler source file ( <b>char *</b> )
<code>current_include_line</code>	top level file line from which the file is included ( <b>unsigned</b> )
<code>argc</code>	interpreter parameters count ( <b>int</b> )
<code>argv</code>	interpreter parameters ( <b>char**</b> )
<code>p_stack</code>	pointer to call stack ( <b>STACK *</b> )

and these predefined functions:

```
void push(STACK* p_stack, unsigned what);
unsigned pop(STACK* p_stack);
```

Definitions of the following macros affect debugger commands `print` and `printout`.

```
#define { #define PRINT_MEM pointer }
#define { #define PRINT_TYPE type }
#define { #define PRINT_MIN minimum address }
#define { #define PRINT_MAX maximum address }
#define { #define PRINTOUT_MEM pointer }
#define { #define PRINTOUT_TYPE type }
#define { #define PRINTOUT_MIN minimum address }
#define { #define PRINTOUT_MAX maximum address }
```

Commands `print` and `printout` display contents of linear memories. The first line defines the pointer to the memory. The second line defines the type of values that are stored in the memory. The third and fourth lines define minimum and maximum addresses. The other four lines are analogous, but they concern `printout` command.

## Instruction Definitions

Syntax:

```
#define name  $\$id_1, \dots, \$id_n$  :opcode  $\$id_{i_1}[l_{i_1}][opcode_{i_1}] \dots \$id_{i_m}[l_{i_m}][opcode_{i_n}]$  [{\
instruction semantics (embedded user C code)}]
```

Please note that unproportional brackets (`[, ]`) are used only to denote optional parts while proportional brackets (`(, )`) are the mandatory part of the syntax.

The above notation defines an instruction with mnemonic *name* (it must be a valid identifier). This instruction has *n* operands. Each operand has an id that must be a valid identifier, but it need not be unique in the nanoprogram. It is used only to match left-side ids to right-side ids inside definition of a single instruction; the sides are split by the semicolon (`:`). The left side describes the syntax of the new instruction. This generic instruction has the following syntax:

```
name operand1, ..., operandn
```

The right side of the definition describes the machine code. It begins with an opcode (operation code). It is a string of ones and zeros, i.e., `0100` or `"0100"`; quotes are optional in this case. Sequence of operands eventually interleaved with the rest of opcode follows. The number in brackets  $l_{i_j}$  denotes the length of the operand  $id_{i_j}$  in bits. If the  $id_{i_j}$  matches to a  $id_k$  on the left side, the operand number *k* is put on the position of  $id_{i_j}$  every time the instruction is compiled. Otherwise, the part of the machine code occupied by  $id_{i_j}$  is undefined. The sum of lengths of operands and opcode (all its parts) should be equal to defined instruction length (see Section A.2.7). Example of instruction definition and its machine code is shown in Figure A.3.

The semantics of instructions follows the same rules as embedded user C code (see previous topic in this Section) except for the fact that user defined variables are local. Most importantly, the same automatic variables and predefined functions may be utilized. The semantics is expected to contain definitions of variables and commands only.

Figure A.3: Example of instruction definition and machine code

Definition:

```
instruction1 $a, $b, $c : 101011 $a[2] $b[8] 00 $x[2] $c[4]
```

Machine code:

```
101011aabbbbbbbb00xxccc
```

### A.2.2 #else

This is a directive for conditional compilation; `#else` may be utilized only in conjunction with `#ifdef` or `#ifndef` (see below).

### A.2.3 #endif

This is a directive for conditional compilation; `#endif` may be utilized only in conjunction with `#ifdef` or `#ifndef`. The correct syntax as well as semantics are described in the specification of these directives (see below).

### A.2.4 #ifdef

This is a directive for conditional compilation. Syntax is the following:

```
#ifdef identifier
```

```
...
```

```
#endif
```

or

```
#ifdef identifier
```

```
...
```

```
#else
```

```
...
```

```
#endif
```

The part of nanoprogram between `#ifdef` and `#endif` (or `#else` respectively) is processed if and only if *identifier* has been previously defined. This identifier may be a macro or a label. The else branch is processed in the opposite case.

### A.2.5 #ifndef

This is a directive for conditional compilation. Syntax is the following:

```
#ifndef identifier
```

```
...
```

```
#endif
```

or

```

    #ifndef identifier
    ...
    #else
    ...
    #endif

```

The part of nanoprogram between `#ifndef` and `#endif` (or `#else` respectively) is processed if and only if *identifier* has been previously defined neither as a macro nor as a label. The else branch is processed in the opposite case.

### A.2.6 #include

Syntax:

```
#include string
```

The *string* is a filename; it may contain an absolute or a relative path. The referred file must contain a nanoprogram consisting of directives only. Recursive including is possible.

The included directives have the same meaning as those that are directly in the nanoprogram. Instruction definitions are recommended to be included from an external file.

### A.2.7 #instrlen

Syntax:

```
#instrlen number
```

This instruction specifies the length of a single instruction in bits. It is a mandatory attribute of the instruction set and it must be defined prior to defining instructions.

### A.2.8 #name

Syntax:

```
#name string
```

This macro specifies the name of the instruction set (or the nanoprocessor). Its use is optional.

### A.2.9 #step

Syntax:

```
#step
```

If *ip* is the instruction pointer of the previous instruction, this directive inserts *ip modulo stepval* NOPs. NOP (no operation) instruction and *stepval* (see directive `#stepval` below) must be defined.

### A.2.10 #stepval

Syntax:

```
#instrlen number
```

This directive defines *stepval* value. This value is utilized by **#step** directive (see above).

### A.2.11 #undef

Syntax:

```
#instrlen identifier
```

This directive allows to undefine a previously defined macro or instruction.

## A.3 Instructions

Only those instructions that are defined in the nanoprogram (or in included files) may be used. The correspondence between instruction definition and instruction syntax is specified in Section A.2.1. The syntax of a generic instruction is the following:

```
name operand1, ..., operandn
```

An operand must be a number or a label (or a macro). The future versions of NA will allow arithmetic expressions operating with these elements.

## A.4 Labels

Syntax:

```
label:
```

A *label* should be a unique identifier. If a label is utilized as an instruction operand, it is expanded to pointer to the instruction that follows the label.

## Appendix B

# Contents of the CD

topic/software	location	hint
HFE	ipv6/hw/HdrXtr/	
HFE nanoprograms	ipv6/hw/HdrXtr/hfe-progs/	
L2_strip	ipv6/hw/L2_strip/	
License	license.txt	
Nanoprocessor Simulator	ipv6/sw/hwsim/nsim/	./presentation
Parallel Simulator	ipv6/hw/L2_strip/	./run or ./runslow
Thesis	thesis/	
Thesis source	thesis/source/	

Note: all programs are provided with a UNIX Makefile.