VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

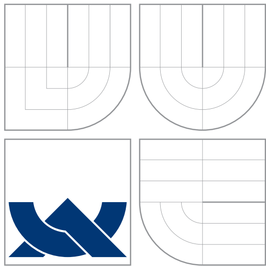
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATED GENERATION OF PROCESSING ELEMENTS FOR FPGA

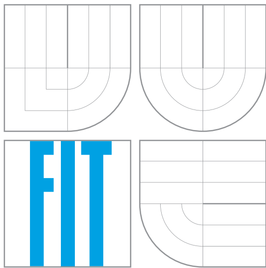
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ LENGÁL
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENEROVÁNÍ PROCESNÍCH ELEMENTŮ PRO FPGA

AUTOMATED GENERATION OF PROCESSING ELEMENTS FOR FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ LENGÁL
AUTHOR

VEDOUCÍ PRÁCE — Ing. MARTIN ŽÁDNÍK
SUPERVISOR

BRNO 2008

Abstrakt
Některé aplikace zpracovávající informace, jako je například monitorování počítačových sítí, vyžadují nepřetržité zpracovávání dat přicházejících vysokou rychlostí. S tím, jak tato rychlost vývojem stále stoupá, je žádoucí, aby bylo zpracovávání dat prováděno pomocí hardwarové implementace. Tato práce navrhuje konfigurační systém transformující uživatelem poskytnutou definici procesních funkcí na VHDL definici hardwarové implementace těchto funkcí. Systém je zaměřen na monitorování síťového provozu ve vysokorychlostních sítích.

Klíčová slova
pokročilé metody syntézy, plánování, zpracování toků, generovaní firmwaru, FPGA, monitorování sítí

Abstract
Some information processing applications, such as computer networks monitoring, need to continuously perform processing of rapidly incoming data. As the speed of the incoming data increases, it is desirable to perform the processing in the hardware. This work proposes a configuration system that generates a VHDL specification of a hardware data processing circuit based on a user-provided definition of data and computation operations. The system focuses on network traffic monitoring in multi-gigabit computer networks.

Keywords
high-level synthesis, scheduling, flow processing, firmware generation, FPGA, network monitoring

Citace
Ondřej Lengál: Automated Generation of Processing Elements for FPGA, bakalářská práce, Brno, FIT VUT v Brně, 2008

Automated Generation of Processing Elements for FPGA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Žádníka. Další informace mi poskytli kolegové z projektu Liberouter. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Ondřej Lengál
14. května 2008

Poděkování

V prvé řadě bych rád poděkoval vedoucímu své bakalářské práce panu Ing. Martinu Žádníkovi za odborné vedení a konzultace. Dále bych chtěl poděkovat panu Ing. Janu Kořenkovi za podnětné připomínky a kolegům z projektu Liberouter za vytvoření přátelské tvůrčí atmosféry. Své milé mamince vyjadřuji srdečný dík za dobroty, které mi připravovala po dobu mého jinošství.

© Ondřej Lengál, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction **3**

2 Theoretical Background **5**

2.1 Hardware Description Languages . . . 5
2.2 Parsing . . . 5
2.3 Code Generation Patterns . . . 6
2.4 OSI Reference Model . . . 7
2.5 Data Link Layer and Ethernet . . . 9
2.6 Network Layer and Internet Protocol . . . 9
2.7 Transport Layer . . . 11
2.7.1 Transmission Control Protocol . . . 11
2.7.2 User Datagram Protocol . . . 12
2.8 Network Flow Monitoring . . . 12

3 Current Approaches to High-Level Synthesis **13**

3.1 Scheduling . . . 14
3.2 Time-Constrained Scheduling . . . 14
3.2.1 Integer Linear Programming Method . . . 14
3.2.2 Force-Directed Heuristic Method . . . 15
3.3 Other Approaches to High-Level Synthesis . . . 15
3.4 Analysis of High-Level Synthesis Methods . . . 15

4 Design and Architecture **16**

4.1 Design . . . 16
4.1.1 Operation Definition Language . . . 16
4.1.2 Configuration . . . 17
4.2 Architecture . . . 17
4.3 Firmware Generation . . . 18
4.3.1 High-Level Language Parsing . . . 18
4.3.2 Complete Abstract Syntax Tree . . . 19
4.3.3 Operations Graph . . . 20
4.3.4 Data-Flow Graph . . . 21
4.3.5 Pipelined Data-Flow Graph . . . 22
4.3.6 Control/Data-Flow Graph . . . 27
4.3.7 VHDL Source Code Generation . . . 27

5 Evaluation **28**
5.1 Application 28
5.2 Flow Processing Unit 29
5.3 Properties 30

6 Conclusion **32**

A Storage Medium **36**

Chapter 1

Introduction

As the volume and speed of electronic communication increase, the demand for processing of communication at top speed occurs. The current speed of communication disables its processing in the computer processor, due to the limited system bus throughput of data to computer memory and processing of the data by a generic processor. High speed data processing may be conducted by *application-specific integrated circuits* (ASICs), which are chips optimized for certain data processing. The main disadvantages of ASICs are: *(i)* once designed and manufactured, the configuration options of the chip are restricted very much, and *(ii)* the price of the chip manufacturing process is high, which makes the production of a smaller amount of these chips uneconomical.

Nevertheless, in practice there are applications that require both high data processing speed and high degree of configurability. *Field-programmable gate arrays* (FPGAs) appear to be a convenient platform for such applications. These chips are composed of an array of configurable elements, which can be programmed to perform a specific data processing function at a given time. If a different function is needed later, it is possible to erase the chip configuration and replace it with another one performing the new function.

Despite the high level of FPGA chip configurability, developing an application-specific configuration of a chip requires an experienced hardware designer and a considerable amount of their time. This may be inefficient for the same family of applications where usually a minor deviation from the standard design is made. However, there is a possibility to create a higher-semantic level engine that automatically generates optimized lower level hardware description from the user's definition, which is then synthesized and downloaded into the reconfigurable chip. This allows a hardware-design inexperienced user to define the data processing function in a friendly way, either through a graphical user interface or simple definitions in configuration files.

An example of such a family of applications is computer networks monitoring — the speed of current computer networks exceeds 10 Gb/s, which results in the maximum packet rate in the order of ten million packets per second, while commodity PCs are able to process network traffic in the order of hundred thousand packets per second. Moreover, the requirements of network administrators on the monitoring process may vary greatly. Another example may be monitoring of a computer system bus, processing of data extracted from audio or video streams or processing of data provided by a large number of very fast sensors (such as those used in nuclear physics experiments).

This work proposes a system for automated generation of synthesizable VHDL description of processing elements according to the user's definition. The focus is placed on the generation of a processing element for a network monitoring probe intended for flexible

monitoring in high-speed networks (10 Gb/s and more).

The text is divided into several chapters. Chapter 2 focuses on hardware description languages and means of high-level language analysis through parsing; code generation patterns are also discussed. Further, OSI network reference model and network flow monitoring are introduced. In chapter 3, current high-level synthesis approaches are analysed and compared to the developed platform. Chapter 4 outlines the design of the firmware generation system and proceeds with the description of the architecture of the system, followed by detailed description of the algorithms developed for firmware generation. Chapter 5 describes the application of the designed platform in a flexible network monitoring probe and evaluates important properties of the solution. Finally, chapter 6 summarizes the status of the work and proposes possibilities of further development.

Chapter 2

Theoretical Background

The task of this thesis is transformation of a high-level language into VHDL code of synthesizable firmware description. For this purpose, chapter 2 first analyses current hardware description languages (section 2.1), which is followed by the description of programming language parsing (section 2.2). Approaches to code generation are described in section 2.3. With the intended focus of this work on the use in network monitoring, section 2.4 describes the OSI network reference model. Further, sections 2.5, 2.6 and 2.7 provide deeper insight into the layers of the model, which are important for network monitoring. Finally, in section 2.8, the principles of network monitoring are described.

2.1 Hardware Description Languages

There are many hardware description languages (HDL), each of them suitable for a different task. Some examples of currently used HDLs are: VHDL [1], Verilog [2], Handel-C [3] and SystemVerilog [4]. The older HDLs, such as VHDL or Verilog, were originally developed for the description of the behaviour of the marketed ASICs and their verification, and only later they were also employed for the synthesis of hardware. The advantage of using these languages for hardware design is good control over the resulting synthesized hardware; however, it is difficult to maintain the code which needs to be optimized for target architecture.

Newer languages (Handel-C, SystemVerilog, . . .), on the other hand, provide higher-level abstraction for easier design of synthesizable hardware evaluating complex functions. The code written in these languages is better maintainable, and higher abstraction allows for optimized use of hardware resources. In spite of such variety of available HDLs, their application still requires an experienced hardware designer. Therefore, these languages are not a feasible solution for the applications in which a hardware-design inexperienced user should be able to define processing functions. The solution may be to develop languages that provide higher-semantic level of data processing definition.

2.2 Parsing

The analysis of a complex user input is usually done by means of a *parser*, which is a program that reads the user input and determines whether it conforms to the given formal grammar. During this process, it also extracts semantic information and constructs *abstract syntax tree* (AST), which contains all necessary information from the user input [5].

A stage preceding the parsing itself is *lexical analysis* of the input. The task of the lexical analysis is to read the user input, strip unnecessary parts (such as whitespaces or comments) and transform it into a string of lexical elements (*lexemes*), which are then passed to the parser. Lexemes are usually defined using *regular expressions*. A lexical analyzer is then implemented using the *deterministic finite automaton*, which has the power to accept the set of languages denoted by regular expressions.

Programming languages are very often *deterministic context-free languages*; therefore, they are possible to be analyzed by parsers based on context-free grammars. *LR parsers* are powerful enough to process the family of deterministic context-free languages [6] and are used in most cases. A language is described using a language grammar G, which is a 4-tuple $G = (N, T, R, S)$, where

- N is a finite set of *nonterminals*,
- T is a finite set of *terminals*, $N \cap T = \emptyset$,
- R is a finite set of *rules* in the form of $A \rightarrow b$, where $A \in N$ and $b \in (N \cup T)^*$,
- S is the *start nonterminal*, $S \in N$.

Language $L(G)$ defined by grammar G is a set of strings of terminals which can be generated from the start nonterminal, $L(G) = \{a : a \in T^*, S \rightarrow^* a\}$.

The following solutions are possible for parsing expressions:

LL parser — does not provide the power to parse all deterministic context-free languages; the grammar needed for parsing expressions is complicated and not intuitive (new meaningless symbols are introduced, etc.).

Operator precedence parser — is limited to a subset of LR grammars where there are not two consecutive nonterminals in the right-hand side of any rule.

LR parser — has the power to parse all deterministic context-free languages [6]. The LR parser is the most complex one for this family of languages. Nonetheless, there are also parser generators (such as [7]) that generate the result parser from the definition of LR grammar. This solution provides a high degree of power and maintainability.

2.3 Code Generation Patterns

Many programming tasks require or may benefit from on-line generation of the program source code. The code generation may be described as the transformation of a higher-level language model description (including graphical language models, such as UML) into a lower-level (usually directly compilable or interpretable) language. Some of the reasons for the use of code generation are these [8]:

- requirements of a high degree of flexibility without the usage of the object-oriented paradigm,
- minimization of the program executable size,
- generation of a program code that is easier to be statically analyzed,

- providing a higher level of abstraction for easier programming, and
- bypassing the limitations of the target programming language.

There are several code generation patters, each suitable for a different situation. Some of them are described by Völter [8]:

1. **Templates and Filtering** — the result code is generated by applying templates to the model specification in the text format (often XML). Example: XSLT transformation of XML into HTML.
2. **Templates and the Metamodel** — the metamodel based on the model data is first instantiated. Afterwards, the templates are applied to this metamodel. Example: PHP web page that consists of HTML code with embedded PHP commands.
3. **Frame Processing** — this pattern uses *frames*, which are basically parametrizable programs or functions that generate the code as the result of their evaluation. Example: PHP web page that sequentially calls functions which generate HTML tags according to the functions' parameters.
4. **API-based Generators** — provide API for generating the result code. These APIs are usually based on the syntax of the target language. Example: libxml2 [9] API for XML tree generation.
5. **Inline Code Generation** — in this case, additional constructions are incorporated directly into the program source code. These constructions may be used e.g. to compile a different code for different operating systems, or for querying the language standard applied and using proper keywords conforming to the standard. In this case, the source code is typically modified prior to the compilation. Example: C99 preprocessor.
6. **Code Attributes** — an additional code may be generated in the runtime according to the metadata stored in a separate part of the program executable. Example: Java annotations.
7. **Code Weaving** — independent code blocks (often of orthogonal functionality) are joined together according to the defined join specification. Example: pool of static libraries with different implementation of functions with the same interface; at the compile-time, one of the implementations is selected and linked to the executable.

HDL code generation may exploit most of the abovementioned patterns. However, FPGA chips may also use some other techniques, such as on-line swapping of Configurable Logic Blocks (CLBs) or application of evolvable components [10].

2.4 OSI Reference Model

OSI (Open Systems Interconnection) reference model [11] is a layered model of network architecture created by the International Organization for Standardization (ISO). It contains 7 layers; each layer is a collection of related functions that provides services to the layer above it and receives services from the layer below. The hierarchy of layers is depicted in figure 2.1.

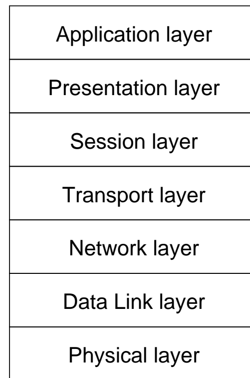


Figure 2.1: OSI reference model

The reference model precisely defines the interfaces between different computers; however, due to its complexity, the model is nowadays used mostly as a reference. A brief description of the layers follows:

Physical layer defines logical signals and the transfer medium at the lowest level, the task of this layer is to transfer bits between adjacent network nodes. This layer also provides functions for line encoding, signal modulation, detection of carrier signal and collisions.

Data Link layer creates data connection above the physical layer. Usually, the connection is created between two adjacent network nodes. The data link layer assembles stream of the physical layer bits into *frames* with well-defined structure. In general, the frame includes the source and destination addresses and a correction code for the detection and retransmission of corrupted frames.

Network layer carries out the routing of data *packets* from the sender to the recipient, who may not be in the same network. A unique node identification is therefore needed, independent on the data link layer addressing. This identification is called network address and is assigned to every node in the network. The network layer services may be either connection-oriented of connectionless.

Transport layer provides transparent transfer of the data between end elements in individual network nodes. These elements may be processes or user sessions. The data is transferred in *segments* with identification of the source and destination communicating elements in the nodes. The Transport layer may use both connection-oriented and connectionless services. The former perform connection establishment and reliable data delivery.

Session layer allows the processes or users in different network nodes to create a session that maintains the context of the communication.

Presentation layer delivers and formats information between systems of a different syntax and semantics.

Application layer provides services for application processes, such as mail transfer, file transfer, instant messaging, etc. This layer includes the processes which implement the functionality.

2.5 Data Link Layer and Ethernet

One of the most prevalent data link layer protocols is Ethernet [12]. The protocol maintains communication between two adjacent network nodes and defines the protocol for data transmission over a single link. The data link layer also defines the methods for medium sharing — Ethernet uses the CSMA/CD method [12].

The data from upper layers are grouped into frames, the beginning and end of which are delimited by special marks. The frame structure is outlined in Figure 2.2 and described further.

Preamble	SFD	Destination address	Source address	Length /Type	MAC client data	PAD	Frame check sequence

Figure 2.2: Ethernet frame structure

Preamble (7 bytes) — a field that allows the physical layer signaling circuitry to reach its steady-state synchronization with the received frame's timing. The content of each byte of this field is 10101010.

Start Frame Delimiter (SFD) (1 byte) — indicates the beginning of a frame. The content of this field is 10101011.

Destination address (6 bytes) — a field that specifies the destination addressee(s) for which the frame is intended. This may be either an individual or group address.

Source address (6 bytes) — a field that specifies the station from which the frame was initiated.

Length/Type (2 bytes) — this field takes one of two meanings, depending on its numeric value. If the value is greater than 1536 (600h), then it indicates the nature of the MAC client protocol (Type interpretation). Otherwise, it indicates the number of MAC client data octets contained in the subsequent data field of the frame (Length interpretation).

Data and PAD (46–1500 bytes) — a field that contains the higher layer data. A minimum frame size is required for correct CSMA/CD protocol operation. If necessary, the data field is extended by appending extra bits (i.e. a pad) in units of octets after the data field.

Frame check sequence (4 bytes) — 32-bit-wide cyclic redundancy check (CRC) for detection of corrupted frames. It is computed from the abovementioned fields except for the *Preamble* and *SFD*.

2.6 Network Layer and Internet Protocol

Internet Protocol (IP) belongs to the network layer of the OSI reference model. It is a datagram protocol for packet-switched networks transferring data from the source to the destination node [13]. The most prevalent protocol used today is IP version 4 (IPv4). Each node in IPv4 network has a unique IP address, which is composed of the address of the

network and the address of the node in the network. The structure of the IPv4 header is presented in Figure 2.3.

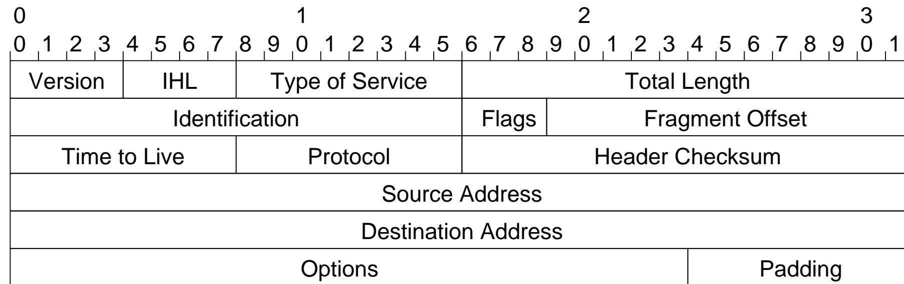


Figure 2.3: IPv4 header structure

The individual elements have the following meaning:

Version (4 bits) — indicates the format of the header. For IPv4 header, this field has the value 4.

Internet Header Length (IHL) (4 bits) — the length of the header in 32-bit words.

Type of Service (8 bits) — a field that provides an indication of the abstract parameters of the quality of the service desired.

Total Length (16 bits) — the length of the datagram, measured in octets, including Internet header and data.

Identification (16 bits) — an identifying value assigned by the sender to aid in assembling the fragments of a datagram.

Flags (3 bits) — various control flags controlling fragmentation of the packet.

Fragment Offset (13 bits) — a field indicating where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets.

Time to Live (8 bits) — this field was originally supposed to define the maximum time in seconds that the datagram was to remain in the network. The contemporary purpose of this field is to define the maximum number of network layer elements through which the packet is allowed to go. At each element, the value in this field is decremented by one and when the value reaches 0, the packet is discarded.

Protocol (8 bits) — the number of the next layer protocol used in the data portion of the Internet datagram.

Header Checksum (16 bits) — checksum of the IPv4 header.

Source Address (32 bits) — the address of the source.

Destination Address (32 bits) — the address of the destination.

Options (variable size) — various optional data.

Padding (variable size) — this field ensures that the Internet header ends on a 32-bit boundary. The padding value is zero.

2.7 Transport Layer

2.7.1 Transmission Control Protocol

Transmission Control Protocol (TCP) provides reliable, in-order stream of bytes. It is a connection-oriented protocol that implements congestion avoidance mechanisms [14]. The structure of TCP header is shown in Figure 2.4.

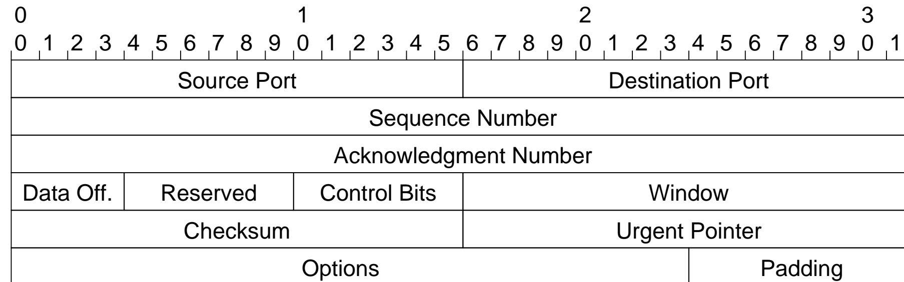


Figure 2.4: TCP header structure

Source Port (16 bits) — the source port number.

Destination Port (16 bits) — the destination port number.

Sequence Number (32 bits) — the sequence number of the first data octet in this segment.

Acknowledgment Number (32 bits) — if the ACK control bit is set, this field contains the value of the next sequence number which the sender of the segment is expecting to receive.

Data Offset (4 bits) — the number of 32-bit words in the TCP header. This field indicates where the data begins.

Reserved (6 bits) — reserved for future use. This field is filled with zeros.

Control Bits (6 bits) — a control flags array. The flags relating to this work are: ACK (Acknowledgment Number field is valid), RST (reset the connection), SYN (synchronize sequence numbers) and FIN (no more data from sender).

Window (16 bits) — the number of data octets beginning with the one indicated in the Acknowledgment Number field which the sender of this segment is willing to accept.

Checksum (16 bits) — checksum of certain IP header and TCP header fields and data.

Urgent Pointer (16 bits) — communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. This field is only interpreted if the URG control bit is set.

Options (variable size) — various optional data.

Padding (variable size) — this field ensures that the TCP header ends on a 32-bit boundary. The padding value is zero.

2.7.2 User Datagram Protocol

User Datagram Protocol (UDP) [15] is a connectionless datagram transfer protocol without the guarantee of reliable and in-order data delivery. The UDP header structure is outlined in Figure 2.5 and explained further.

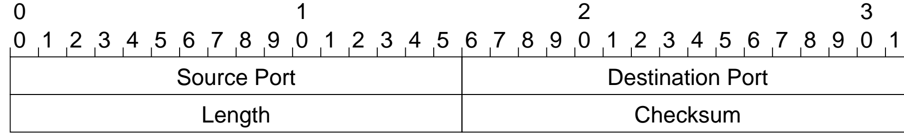


Figure 2.5: UDP header structure

Source Port (16 bits) — may indicate the port of the sending process to which the reply should be addressed. If it is not used, zero value is inserted.

Destination Port (16 bits) — the destination port number.

Length (16 bits) — the length of this user datagram including the header and the data.

Checksum (16 bits) — a checksum of certain IP header and UDP header fields and data.

2.8 Network Flow Monitoring

Network flow is defined in [16] as a set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties. Each property is defined as the result of applying a function to the values of:

1. one or more packet header fields (e.g. destination IP address), transport header field (e.g. destination port number), or application header field (e.g. RTP header field);
2. one or more characteristics of the packet itself (e.g. number of MPLS labels);
3. one or more of the fields derived from packet treatment (e.g. next hop IP address, the output interface).

A packet is defined to belong to a flow if it completely satisfies all the defined properties of the flow. The observation point is often a probe or a router interface. The information about a specific flow which was metered at the observation point are exported to collectors in the format of *flow records*, which contain the measured properties of the flow (e.g. the total number of bytes of all packets of the flow) and usually also characteristic properties of the flow (e.g. source IP address).

In recent years the flow monitoring technology application has become widespread in the form of Cisco NetFlow and IPFIX. While the set of possible flow key definitions and measured properties of the mostly used NetFlow v5 version is fixed, newer versions (such as NetFlow v9) and IPFIX [17] allow flexible definition of both the flow key and the measured properties. The flexible definition is very important for higher semantic level protocol analysis [18]. For some applications, such as traffic classification, application protocol detection or anomaly detection, it is necessary to measure different properties.

Chapter 3

Current Approaches to High-Level Synthesis

The task of high-level synthesis is to simplify the hardware design process allowing the hardware designer to use higher-semantic level constructs while performing functionality-independent optimization, which is too complex or error-prone to be done by a hardware designer. This optimization may be done:

- in the **physical domain** in the form of optimized placement of transistors, wire networks and floorplan creation in ASIC chip or optimized placement and routing for FPGA;
- in the **structural domain** as optimized partitioning, scheduling of registers, multiplexers and functional units usage, scheduling of bus usage, allocation of registers and functional units, etc.;
- in the **behavioral domain** in the form of Boolean expression minimization, algorithm optimization, loop unrolling, death code elimination, etc.

Optimization in the physical domain is performed by synthesis tools and depends on the target technology. The first optimization computer-aided design (CAD) tools usually perform is in the behavioral domain. This process is very similar to the machine-independent optimization done in software programming languages compilers [19]. In this case, good practices (such as constant propagation or death code elimination) are able to decrease both resource usage and propagation delay of the design.

The structural domain optimization works on the register-transfer logic (RTL) level and is focused on the instantiation of registers, multiplexers and functional units and their partitioning, scheduling and allocation. The task of partitioning is to cluster related objects together, especially to minimize the layout area and propagation delay. In case of FPGAs this is part of the physical domain optimization and is done by the synthesis tool. Scheduling maps operations into separate clock cycles, while allocation defines for each clock cycle which register and which functional unit is taken by which operand. Scheduling plays a key role in the optimization in the structural domain for high-speed FPGA applications.

3.1 Scheduling

The task of hardware scheduling is to partition the control/data-flow graph (CDFG) into subgraphs, each of which is executed in a single control step [20]. Every step corresponds to one state of the controlling finite-state machine (FSM). There are two possible goals for the scheduling problem: *(i)* minimization of the number of functional units for the fixed number of control steps, and *(ii)* minimization of the number of control steps for a given design cost (e.g. number of functional units). Good scheduling algorithms create designs that reuse expensive components, such as adders and multipliers, while retaining high throughput.

An important concept in scheduling is the data-flow graph (DFG). This is defined [20] as a directed acyclic graph $G(V, E)$, where V is a set of nodes and E is a set of edges. Each element $v_i \in V$ represents an operation (o_i) in the behavioral description of the hardware. A directed edge e_{ij} from $v_i \in V$ to $v_j \in V$ — denoted as (v_i, v_j) — exists in case that data produced by o_i (represented by v_i in the DFG) is an input operand of the operation o_j (represented by v_j). The set of all immediate predecessors of v_j is defined as $Pred_{v_j} = \{v_i : (v_i, v_j) \in E\}$; the set of all immediate successors of v_i is defined as $Succ_{v_i} = \{v_j : (v_i, v_j) \in E\}$.

Using the DFG, we are able to detect which operations can be carried out in parallel and which need sequential processing. This information is crucial for scheduling algorithms.

Two fundamental algorithms are ASAP (as soon as possible) and ALAP (as late as possible) scheduling [20]. The ASAP algorithm assigns an ASAP value E_i (denoting the first clock cycle in which it is possible to execute the operation) to each operation o_i according to the following equation:

$$E_i = \max(W) + 1, \quad W = \begin{cases} \{E_j : j \in Pred_{v_i}\} & \text{for } Pred_{v_i} \neq \emptyset \\ \{0\} & \text{for } Pred_{v_i} = \emptyset \end{cases} \tag{3.1}$$

The ALAP algorithm generates an ALAP value L_i (which is the last clock cycle in which it is possible to execute the operation) to every operation o_i in a way very similar to the previous algorithm:

$$L_i = \min(W) - 1, \quad W = \begin{cases} \{L_j : j \in Succ_{v_i}\} & \text{for } Succ_{v_i} \neq \emptyset \\ \{T + 1\} & \text{for } Succ_{v_i} = \emptyset \end{cases} \tag{3.2}$$

where T is the given constraint of control steps. If there exist L_i such that $L_i < 1$, the desired function cannot be executed in T steps.

From the output of ASAP and ALAP algorithms for every operation o_i, it is possible to define the range of control steps $< E_i, L_i >$, into which the operation can be scheduled.

In general, there are two fundamental approaches to scheduling: time-constrained and resource-constrained. The former operates over a given number of clock cycles in which the operation needs to execute and the later works with a limited silicon area. For the purpose of this work, the time-constrained scheduling is relevant.

3.2 Time-Constrained Scheduling

3.2.1 Integer Linear Programming Method

This method finds an optimal schedule for a given constant time using a branch and bound search algorithm with backtracking. The ASAP and ALAP values are used to determine the mobility range of operation. The algorithm then finds a schedule which meets the

constrained time while consuming minimum resources. Because this solution is NP-hard, it is not suitable for larger descriptions.

3.2.2 Force-Directed Heuristic Method

This method attempts to uniformly distribute the operations of the same type (e.g. addition) into all available states. The algorithm works with the expected operator cost (EOC) value for each operator and each clock cycle. The process tries to balance the EOC value for each operation type by restructuring the operation graph with regard to the cost function.

3.3 Other Approaches to High-Level Synthesis

Oh and Ha in [21] propose a method for VHDL code generation from the data-flow graph for the use in digital signal processing applications. The paper is focused on generation of coarse-grained pipelined design for signal-related applications. The framework assumes the use of large high-latency processing blocks (such as multipliers or Fast Fourier transform blocks).

Another approach [22] deals with the application of genetic algorithms to scheduling instructions in a compiler targeted towards parallel architectures. The algorithm works upon data dependency directed acyclic graph and provides good results in scheduling; nonetheless, due to its focus on machine language generation, it does not incorporate the means for fine-grained hardware units allocation.

Paxson et al. [18] argue that current network elements do not provide satisfactory high-level processing of network traffic — features like TCP stream reassembly and semantic processing of application data at line speed are missing. High-level definition of semantic processing with compilation into FPGA design is proposed, however, no further details are given.

3.4 Analysis of High-Level Synthesis Methods

None of the studied high-level synthesis methods cope with the problem of transformation of high-level language expressions to pipelined high-speed low-latency fine-grained hardware description. Studied approaches are limited to generation of coarse-grained high-latency designs, which are not convenient for network traffic processing, due to the fine granularity of network protocols header fields.

In the light of these findings, we argue that it is essential to develop an approach that creates an optimized high-speed low-latency fine-grained description of hardware defined by the user employing high-level language constructs. The design of such approach is outlined in the following chapter.

Chapter 4

Design and Architecture

4.1 Design

The task of the processing unit is high-speed processing of data. The structure of the input data is assumed to have the form of several fields each of which may be of different width and each may be associated with different operation (this comes from the analysis of network protocols in 2). Since the input is continuous and the delay of the unit is needed to be as low as possible, we cannot use a single ALU to process the data. Thus, a fine-grained processing pipeline is needed to be generated according to the user definition of operations.

4.1.1 Operation Definition Language

High level language based on C99 [23] syntax has been proposed for the definition of user operations. A user operation is defined by a single expression. However, various functions may be used in the expression; the language may therefore be considered to be functional. The language syntax includes:

- common arithmetic operators (+, -, *, /, %),
- logical operators (&&, ||, !),
- bit manipulation operators (&, |, ^, ~, >>, <<),
- relation operators (==, !=, <, <=, >, >=),
- ternary conditional operator (?:),
- the parenthesis ((and)),
- the assignment operator (=),
- the assignment operator combined with an arithmetic or bit manipulation operator (+=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=),
- bit addressing operator ([,]), and
- concatenation operator (.).

The language includes two data types: `unsigned` and `signed`; expressions can be cast using the cast operators: `(unsigned)` and `(signed)`. The precedence of operators remains the same as in C99. However, semantics of the operators is not strictly given — during parsing, they are transformed into ordinary functions which are defined in the configuration. The language is described using an LR grammar in order to enable processing of the expressions by a LR parser, which was described in section 2.2.

There are two types of operations: *(i)* the update operations, and *(ii)* the control operations. The former type is used for transformation of input data blocks into the result block, and the later type is used for providing control operations over the input blocks; these usually consist of a single bit expressing the state of the control operation (succeeded/failed) and may be used e.g. as interrupt request signals.

4.1.2 Configuration

The configuration is composed of definitions of user functions and operators (see 4.1.1). The operators are basically a special type of a function with the name `operatorXX`, where `XX` is the operator sign, such as `operator+` or `operator?:`. There are two possible types of functions:

- **Hardware-mapped functions** — these functions define the mapping of the operands onto component ports and generics and also other hardware related parameters, such as propagation delay, maximum width of a block etc.
- **Macro functions** — these functions are defined using other functions, i.e. they serve as macros which are expanded during the generation process.

The usage of both function types is uniform. However, it needs to be ensured that the configuration does not contain cyclic definitions, i.e. recursion (either direct or indirect) is not used; in case it were used, it would not be possible to determine the pipeline length statically during firmware generation.

4.2 Architecture

The data processing takes place in a processing unit. The processing unit is a pipelined computational unit the interface of which can be seen in Figure 4.1.

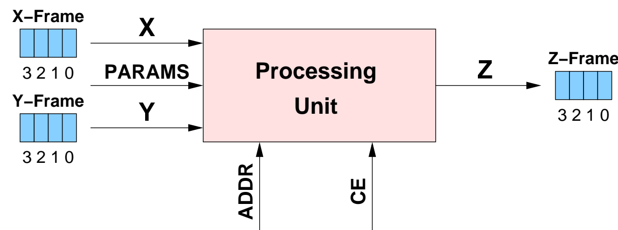


Figure 4.1: The interface of the processing unit

The input operands are passed to the unit via ports **X** and **Y**. The input data are in the form of a frame with well-defined structure. The frame is split into words of equal width, each of them with an address. The words are passed to the unit consecutively from the one

with the lowest address (i.e. zero) together with the address of the word (port **ADDR**) for every stage of the pipeline and the clock enable signal (port **CE**) that indicates valid values on the input. The address is provided for every stage of the pipeline to enable processing more frames at once. **PARAMS** is a port with parameters, i.e. data that changes scarcely, regardless of the input data frames. Port **Z** is the output port with the resulting frame.

4.3 Firmware Generation

The process of firmware generation is composed of several parts. The task of the process is the transformation of user definition of processing functions in the form of high-level programming language expressions into optimized description of the processing unit in VHDL. The output of the process is therefore the control/data-flow graph of the components used in the processing unit and also the definition of structure of the input and output frames. This definition may be used for the generation of the extractor unit that prepares the input data to the form the processing unit expects, and for the generation of another unit that transforms the structure of the output data to the structure expected by the following element on the data route.

4.3.1 High-Level Language Parsing

The parsing of the expressions defining the user operation written in the language proposed in section 4.1.1 is done using a C++ LR parser generated from the language grammar description by the Flex [24] and Bison [7] tools.

Let us consider the following example of operation definitions for describing the generation process:

```
x = max(x, a - b);
y += sqr(a - b);
```

and the definition of macro functions `max`, `sqr` and of the `+` operator. Other functions are supposed to have a proper hardware mapping.

```
signed max(signed first, signed second)
{
  return (first > second)? first : second;
}

signed sqr(signed value)
{
  return value * value;
}

signed operator+=(signed lhs, signed rhs)
{
  return lhs = lhs + rhs;
}
```

Parsing of these expressions creates two abstract syntax trees (ASTs) — one for each expression (see Figure 4.2). Fields used in the left-hand side of the expression are labeled as *new* and fields in the right-hand side of the expression are labeled as *old*.

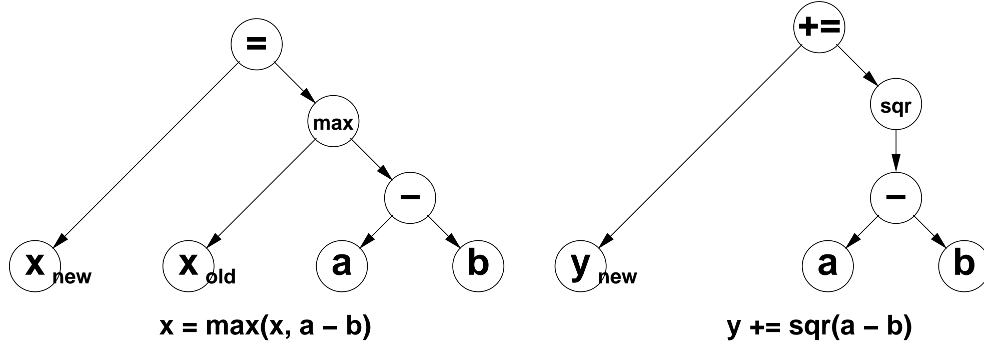


Figure 4.2: Example ASTs

4.3.2 Complete Abstract Syntax Tree

Complete abstract syntax tree (CAST) is an AST containing only atomic operations, i.e. only the functions which are in the configuration (see 4.1.2) defined as hardware-mapped functions. All ASTs from the parser are transformed to CASTs using Algorithm 1. The algorithm expands the tree root node operation until it is hardware-mapped. Then, it continues to recursively expand all children of the node, until only hardware-mapped operations are present in the tree.

The output of the algorithm for the example expressions is in Figure 4.3.

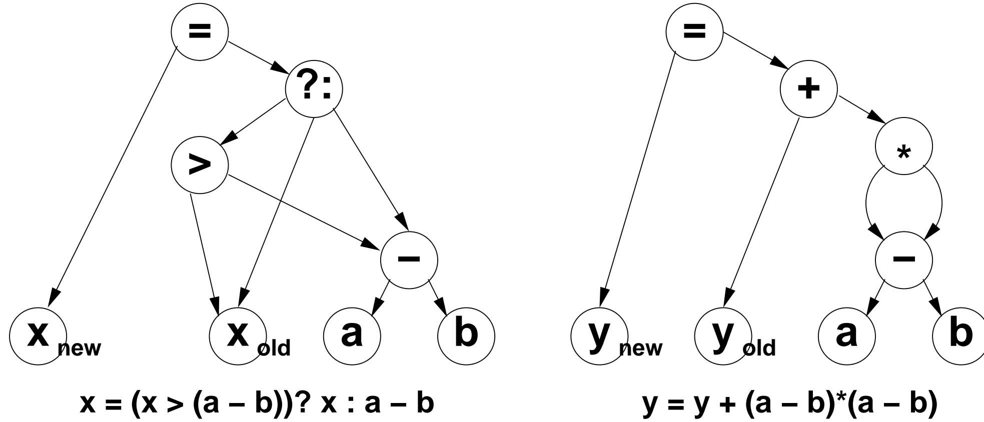


Figure 4.3: Complete ASTs inferred from the example ASTs using Algorithm 1

Algorithm 1: Abstract syntax tree completion

Input: `IN_TREE`: AST from the parser;
`IN_OPERATIONS`: Set of operations description;
Output: `OUT_TREE`: Complete AST

1 **begin**
2 `OUT_TREE` ← `expand_node(IN_TREE.Root, IN_OPERATIONS)`;
3 **end**

Function `expand_node(`*IN_NODE, IN_OPERATIONS*`)`

Input: `IN_NODE`: Input node;
`IN_OPERATIONS`: Set of operations description;
Result: Expanded node;

```
1  begin
2      if IN_NODE is a hardware-mapped function then
3          foreach child ∈ IN_NODE.Children do
4              child ← expand_node(child, IN_OPERATIONS);
5          endfch
6          return IN_NODE
7      else
8          operation ← IN_OPERATIONS.GetOperationAST(IN_NODE.Operation);
9          foreach i ∈ ℕ ∩ ⟨1; |operation.Operands|⟩ do
10             operation.Operands[i] ← IN_NODE.Children[i];
11         endfch
12         return expand_node(operation.Root, IN_OPERATIONS);
13     endif
14 end
```

4.3.3 Operations Graph

When the CASTs are generated, they are transformed into the operations graph. The operations graph is a directed acyclic graph $G = \{N, E\}$, where the set of nodes is $N = I \cup R \cup O$ with I being the set of input fields, R being the set of result fields and O being the set of operations. The set of edges is defined as $E = \{(a, b) : a \in I \cup O, b \in R \cup O\}$.

The operation graph expresses relations between operations and their operands. It does not contain redundant operations (i.e. operations that repeatedly appear in different CASTs) — these are merged into a single instance in order to be evaluated only once. The bottom row of the operations graph contains nodes with the input fields while the top row contains the result fields. The nodes in between represent data transformation operations.

The transformation process is described in Algorithm 2. The algorithm draws a node from a CAST and searches whether the operation of the node is already in the operation graph. In case it is not, the algorithm inserts it and creates edges in the operation graph that correspond to the children of the node. The output of the algorithm for the CASTs from Figure 4.3 is in Figure 4.4. Note the merged common subexpression `(a - b)`.

Algorithm 2: Complete ASTs to operations graph transformation

Input: `IN_TREES`: Set of complete ASTs;
Output: `OUT_NODES`: Nodes of the operations graph;
`OUT_EDGES`: Edges of the operations graph;

```
1  begin
2      OUT_NODES ← ∅ ; OUT_EDGES ← ∅;
3      foreach ast ∈ IN_TREES do
4          get_node(ast.Root, OUT_NODES, OUT_EDGES);
5      endfch
6  end
```

Function `get_node(`*IN_NODE, OG_NODES, OG_EDGES*`)`

```
   Input: IN_NODE: CAST node;
          ref OG_NODES: Nodes of the operations graph;
          ref OG_EDGES: Edges of the operations graph;
   Result: The operation graph node for the root node of the input CAST node;
 1 begin
 2   if IN_NODE is a leaf then
 3     if IN_NODE is in the left-hand side of an assignment then
 4       return OG_NODES.AddResultField(IN_NODE);
 5     else if IN_NODE ∈ OG_NODES.InputFields then
 6       return OG_NODES.GetReferenceTo(IN_NODE);
 7     else
 8       return OG_NODES.AddInputField(IN_NODE);
 9     endif
10   else
11     child_refs ← ∅;
12     foreach i ∈ ℕ ∩ ⟨1; |IN_NODE.Children|⟩ do
13       child_refs[i] ← get_node(IN_NODE.Children[i], OG_NODES, OG_EDGES);
14     endfch
15     op_nodes ← {x : x ∈ OG_NODES ∧ x.Operation = IN_NODE.Operation};
16     foreach node ∈ op_nodes do
17       if node.Children = child_refs then return node;
18     endfch
19     operation ← OG_NODES.AddOperation(IN_NODE);
20     foreach i ∈ ℕ ∩ ⟨1; |child_refs|⟩ do
21       if child_refs[i] is in the left-hand side of an assignment then
22         OG_EDGES.AddEdge(operation, child_refs[i].InputPorts[1]);
23       else
24         OG_EDGES.AddEdge(child_refs[i], operation.InputPorts[i]);
25       endif
26     endfch
27     return operation;
28   endif
29 end
```

4.3.4 Data-Flow Graph

When the operations graph is created, the next task is to generate components for evaluation of individual operations. The structure that contains the set of components and their interconnection is called the data-flow graph (DFG). The process of transformation is shown in Algorithm 3. The transformation process works in two stages: first, components are generated inside each operations graph node independently on other nodes — this is done by the `gen_comps()` procedure. The procedure generates a set of components and wires (i.e. an independent data-flow graph) with respect to the type of operation of the node (a

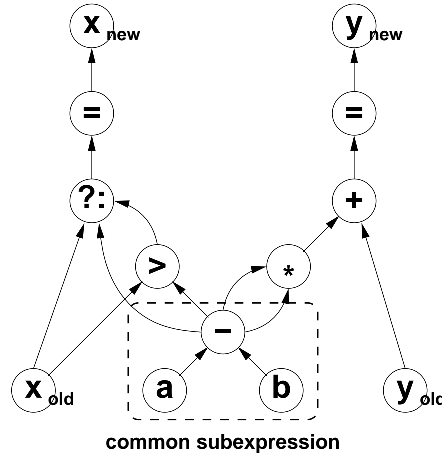


Figure 4.4: Output from Algorithm 2: operations graph for example CASTs

comparison operation is being constructed in a different way than a logical sum operation) and width of the operation input and output. The second stage joins all these independent data-flow graphs into a single one. Figure 4.5 shows the output data-flow graph when the algorithm is applied on the example operations graph.

Algorithm 3: Operations graphs to data-flow graph transformation

```
    Input: IN_NODES: Set of operations graph nodes;
           IN_EDGES: Set of operations graph edges;
    Output: OUT_COMPS: Components of the data-flow graph;
            OUT_WIRES: Wires of the data-flow graph;
 1  begin
 2      OUT_COMPS ← ∅;
 3      OUT_WIRES ← ∅;
 4      foreach node ∈ IN_NODES do
 5          gen_comps(node, OUT_COMPS, OUT_WIRES);
 6      endfch
 7      foreach edge ∈ IN_EDGES do
 8          OUT_WIRES.AddWire(edge.BeginNode.OutputPort,
            edge.EndNode.InputPorts[edge.PortNumber]);
 9      endfch
10  end
```

4.3.5 Pipelined Data-Flow Graph

With the data-flow graph showing the flow of data through components, the next step in order to achieve high processing throughput is creating a pipeline. This process also includes allocation of hardware resources and scheduling of operations. The process is dependent on the structure of the input and output data blocks.

As stated in 3.2.1, finding the optimal schedule is difficult for larger descriptions, because the problem is NP-hard. However, the optimal schedule is not always necessary, the search

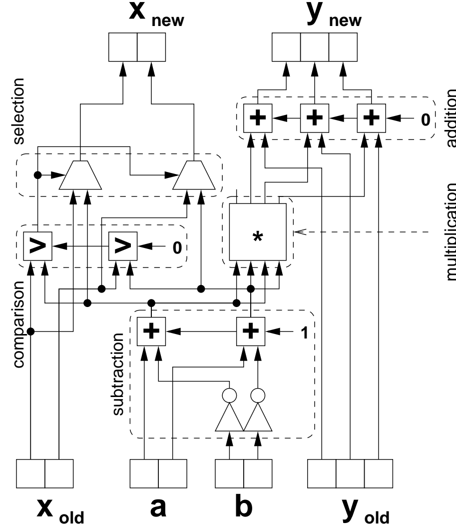


Figure 4.5: Output data-flow graph from Algorithm 3 for example operations graph

may be therefore conducted using a heuristic. The analysis of [22] in section 3.3 showed that genetic algorithms provide good results for the problem of scheduling. The nature of the problem for hardware makes the application of genetic algorithms even more convenient, as was argued in [25].

Because the scheduling depends on the structure of the input and output data, the search algorithm looks for such a structure that can be scheduled into the given number of clock cycles and consumes as few resources on the chip as possible. The algorithm for construction of the structure is described later. The genetic search algorithm works as follows [10]:

1. Generate a population of random chromosomes.
2. Calculate fitness of each chromosome.
3. Use roulette selection to select pairs of parents.
4. Generate offspring with crossover and mutation. If a new population has not been produced yet, go to 3.
5. If the best solution is not good enough yet, go to 2.

The data block is a sequence of integers (genes) $X = [x_1, x_2, \cdots, x_n]$, where x_1 through x_n are pointers into a structure that contains information about the fields of the data block. All the data blocks are grouped into a matrix $\mathbf{T}$

$$\mathbf{T} = \begin{bmatrix} X \\ Y \\ \vdots \\ Z \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ \vdots & \vdots & \ddots & \vdots \\ z_1 & z_2 & \cdots & z_n \end{bmatrix}$$

Grouping the data blocks into a matrix and operating over this matrix is supported by the fact that good pairs of fields of different data blocks can be preserved using this scheme. Chromosome C representing the matrix $\mathbf{T}$ is encoded in this way:

$$\mathbf{T} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ \vdots & \vdots & \ddots & \vdots \\ z_1 & z_2 & \cdots & z_n \end{bmatrix} \quad \longleftrightarrow \quad C = [x_1, y_1, \cdots, z_1, x_2, y_2, \cdots, z_2, \cdots, x_n, y_n, \cdots, z_n]$$

The suggested crossover operator creates a vertical cut through matrices of two parents' chromosomes and exchanges right-hand sides of the cut between the parents to create their offspring:

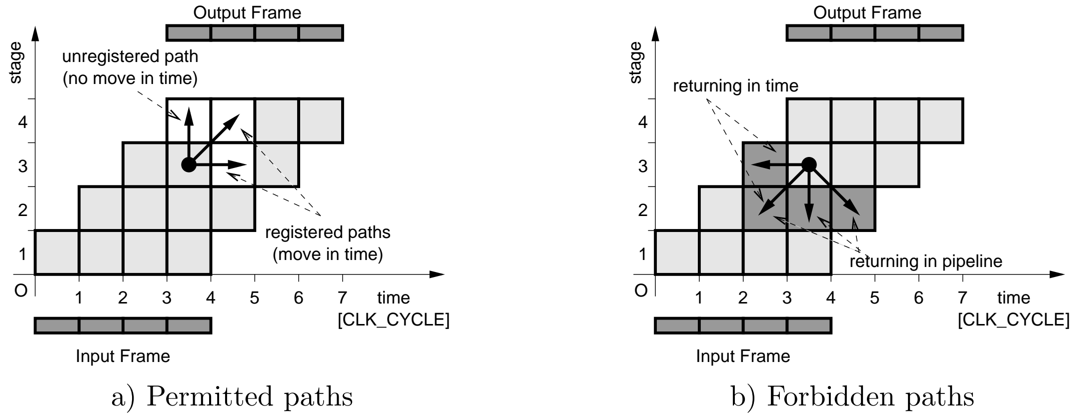
$$\left[\begin{array}{cc|c} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ \vdots & \vdots & \vdots \\ z_1 & z_2 & z_3 \end{array}\right]$$

There are two mutation operator proposed: the first one creates a vertical cut in the chromosome matrix and exchanges the left-hand side for the right-hand side of the matrix. The other one creates two vertical cuts in the matrix and inverts the order of the fields between those cuts. These operators are also defined for individual data blocks of the matrix in the same way — the difference is that only those fields that are in a randomly selected row of the matrix are changed. The currently used mutation operator during generation of new offspring is selected randomly. Resulting offspring must be checked and invalid members discarded — these are the members that does not contain all the fields for every data block exactly once.

The fitness function is evaluated by constructing the pipelined data-flow graph and computing the amount of resources (as described in 4.3.6) the resulting design consumes — lower numbers get higher fitness values.

During the search, the genetic algorithm generates proper attributes derived from the given data block for the set of input ports and output ports. These attributes for all components are: `TimeStart` for the earliest moment when all the input data are available on the input of the component, `TimeReady` for the earliest moment when the component yields the result, and `Stage` for the number of the pipeline stage the component is in. For the functional components, these attributes are evaluated inside the algorithm. All components also have another attribute — `Delay`, which is the propagation delay of the combinatorial path inside the component.

The data-flow graph pipelining algorithm (Algorithm 4) starts with generation of the *list* of the components which are connected directly to the input ports. The *list* is used throughout the process and comprises those components which has at least one scheduled predecessor component and has not been scheduled yet. The components are successively drawn from the front of the *list* and checked if all of their predecessors have been scheduled. In case they have not, the component is appended back to the end of the *list* to be processed later. When all of the component's predecessors have been scheduled, the component can be scheduled too. First, the component is checked if it does not create critical path (procedure `remove_critical_paths()`), that would violate the target clock cycle constraint (constant `CLK_TIME`). If the possibility of a critical path occurs, registers are inserted into those input



a) Permitted paths b) Forbidden paths

Figure 4.6: Permitted and forbidden paths in the pipeline

paths which are time-critical (using function `pipeline_wire()`). When the critical paths are identified and eliminated, values of `TimeStart`, `TimeReady` and `Stage` attributes are assigned to the component. To avoid race condition and similar problems, only the forward direction of flow of data through the pipeline is permitted — this means that the data can progress through the pipeline one stage per clock cycle, that the data can be stored in the stage where it was computed in order to be used later, or that the data can directly "jump over" several stages forward without actually going through the pipeline. Other directions of flow — i.e. going back in the pipeline or in time — are forbidden (see example for 4-stage pipeline in Figure 4.6). The following step processes the input paths of the component to fulfill this condition. At this moment, the component is properly scheduled and its yet unscheduled successors are appended to the *list*.

Procedure `remove_critical_paths`(*IN_COMP, REF_COMPS, REF_WIRES*)

```
Input: IN_COMP: Input component;
       ref REF_COMPS: Components;
       ref REF_WIRES: Wires;
1  begin
2      wires ← {w : w ∈ REF_WIRES ∧ w.To ∈ IN_COMP.InputPorts};
3      pred_comps ← {x : x ∈ REF_COMPS ∧ ∃w ∈ wires : w.From = x};
4      foreach pred ∈ pred_comps do
5          if cycle(pred.TimeReady) ≠ cycle(pred.TimeReady + IN_COMP.Delay)
           then
6              ws ← {w : w ∈ wires ∧ w.From = pred};
7              reg ← pipeline_wire(x : x ∈ ws);
8              REF_WIRES ← REF_WIRES − ws;
9              trg_ports ← {w.To : w ∈ ws};
10             REF_WIRES ← REF_WIRES ∪ {w : w.From = reg ∧ w.To ∈ trg_ports};
11         endif
12     endfch
13 end
```

Algorithm 4: Algorithm for data-flow graph pipelining

```
   Input: ref REF_COMPS: Components of the data-flow graph;
          ref REF_WIRES: Wires of the data-flow graph;
 1 begin
 2 |  unsched_comps ← REF_COMPS ;
 3 |  list ← empty list;
 4 |  foreach port ∈ {x : x ∈ unsched_comps ∧ x is an input port} do
 5 |  |  wires ← {w : w ∈ REF_WIRES ∧ w.From = port};
 6 |  |  comps ← {c : c ∈ unsched_comps ∧ ∃w ∈ wires : w.To ∈ c.InputPorts};
 7 |  |  foreach c ∈ comps do  list.PushBack(c);
 8 |  |  unsched_comps ← unsched_comps − {port};
 9 |  endfch
10 |  while list is not empty do
11 |  |  comp ← list.PopFront();
12 |  |  wires ← {w : w ∈ REF_WIRES ∧ w.To ∈ comp.InputPorts};
13 |  |  pred_comps ← {x : x ∈ REF_COMPS ∧ ∃w ∈ wires : w.From = x};
14 |  |  if ∃c ∈ pred_comps : c ∈ unsched_comps then
15 |  |  |  list.PushBack(comp);
16 |  |  else
17 |  |  |  remove_critical_paths(comp, REF_COMPS, REF_WIRES);
18 |  |  |  wires ← {w : w ∈ REF_WIRES ∧ w.To ∈ comp.InputPorts};
19 |  |  |  pred_comps ← {x : x ∈ REF_COMPS ∧ ∃w ∈ wires : w.From = x};
20 |  |  |  if comp is not an output port then
21 |  |  |  |  comp.TimeStart ← max({x.TimeReady : x ∈ pred_comps});
22 |  |  |  |  comp.TimeReady ← comp.TimeStart + comp.Delay;
23 |  |  |  |  comp.Stage ← max({x.Stage : x ∈ pred_comps});
24 |  |  |  endif
25 |  |  |  foreach w ∈ wires do
26 |  |  |  |  while cycle(w.From.TimeReady) ≠ cycle(w.To.TimeStart) do
27 |  |  |  |  |  reg ← pipeline_wire(w);
28 |  |  |  |  |  reg.Stage ← min({reg.Stage, w.To.Stage});
29 |  |  |  |  |  REF_WIRES ← REF_WIRES − {w};
30 |  |  |  |  |  w ← x : x.From = reg ∧ x.To = w.To;
31 |  |  |  |  |  REF_WIRES ← REF_WIRES ∪ {w};
32 |  |  |  |  endw
33 |  |  |  endfch
34 |  |  |  unsched_comps ← unsched_comps − {comp};
35 |  |  |  out_wires ← {w : w ∈ REF_WIRES ∧ w.From = comp};
36 |  |  |  succ_comps ← {c : c ∈ unsched_comps − {x : x is in list} ∧
37 |  |  |                ∃w ∈ out_wires : w.To ∈ c.InputPorts};
38 |  |  |  foreach c ∈ succ_comps do  list.PushBack(c);
39 |  |  endif
40 |  endw
41 end
```

Function cycle(*IN_TIME*)

Input: IN_TIME: Input time;
Result: Number of the clock cycle the input time falls into;

```
1 begin
2   return ⌊IN_TIME/CLK_TIME⌋;
3 end
```

Function pipeline_wire(*IN_WIRE, REF_COMPS, REF_WIRES*)

Input: IN_WIRE: Input wire to be pipelined;
ref REF_COMPS: Components;
ref REF_WIRES: Wires;
Result: The result register reference;

```
1  begin
2    reg ← register component;
3    reg.Stage ← IN_WIRE.From.Stage + 1;
4    reg.TimeStart ← IN_WIRE.From.TimeReady ;
5    reg.TimeReady ← (cycle(reg.TimeStart) + 1) · CLK_TIME;
6    REF_COMPS ← REF_COMPS ∪ {reg};
7    wire_to_reg ← w : w.From = pred ∧ w.To = reg.InputPorts[1];
8    REF_WIRES ← REF_WIRES ∪ {wire_to_reg};
9    return reg;
10 end
```

4.3.6 Control/Data-Flow Graph

The process of control/data-flow graph creation deals with allocation of functional units for all stages of the pipeline in the pipelined data-flow graph. For each stage of the pipeline and each component type, the number of these components of this type for every clock cycle is computed. The maximum value is used as the number of instantiated components. These components are reused for various inputs (depending on the clock cycle), so they need to be multiplexed and a mapping between the clock cycle number and the value on the selection ports of the multiplexers established. A string with this mapping for all multiplexers in the stage constitutes the *program of the stage*.

4.3.7 VHDL Source Code Generation

When the control/data-flow graph is generated, the result needs to be transformed into a form expected at the input of synthesis tools. Because VHDL language enables high degree of control over the synthesis output and is supported by the majority of synthesis tools, it has been chosen as the result of the generation process.

From the code generation patterns described in 2.3, the hierarchical *Templates and the Metamodel* pattern has been chosen: premade code blocks at different structural levels are made with special marks in the places where the metamodel-dependent code shall be placed.

Chapter 5

Evaluation

5.1 Application

The application of this framework is for generation of the *Flow Processing Unit* in the Flexible FlowMon network monitoring probe [26]. A rough outline of the probe can be seen in Figure 5.1. The captured packet is processed in the preprocessing block, where the packet headers are extracted and combined into the *unified header* [27]. The FlowContext [28] stores the *context* for every network flow (see 2.8) and binds proper context to an input packet; it can also load-balance incoming stream of data among several context update blocks. When packet data are received by the FlowContext, the data (unified header, payload and packet flow context) are sent to the *context update block* (CUB) to be processed. Expired contexts are released from the CUB in the form of *flow records* to be further processed by the host PC, which is in turn able to set the monitoring process parameters.

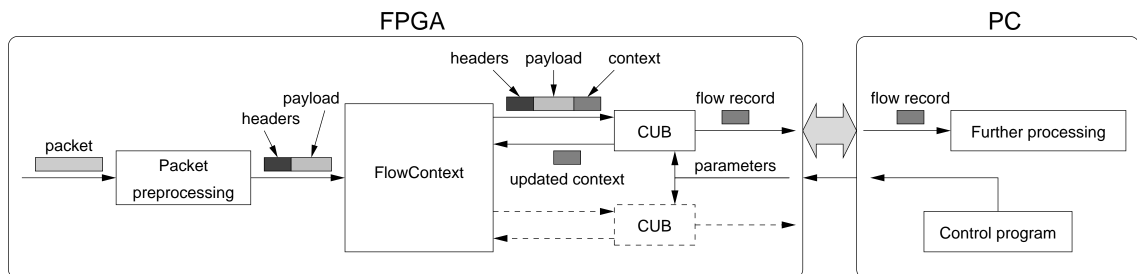


Figure 5.1: Architecture of the Flexible FlowMon probe

The context update block is outlined in Figure 5.2. It consists of these five elements:

Fetch Block decodes the command that is sent together with the packet information in **DATA** and determines how the packet will be processed. This block also checks for collisions, i.e. it ensures that the context bound to the packet is correct, and arbitrary other things. The result are separated data and control paths — the data are sent to the Endpoint via **FLOW_DATA**, and the control signals are sent through **USER_DATA**.

Return Block combines the command sent from the endpoint in **USER_DATA** with the data **FLOW_DATA** into **DATA** frame which is to be stored in the memory.

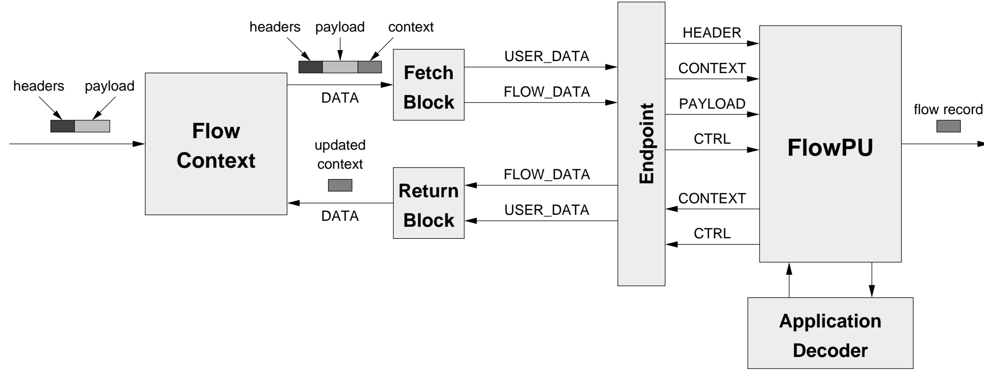


Figure 5.2: Detail of the context update block

Endpoint divides the header, payload and context into separate memory banks in order to make these fields available in parallel; this unit also serves as a cache for the FlowPU — it stores the context of the flow to avoid FlowContext overhead that would otherwise limit throughput of the whole system for large flows. Control commands are also stored in the endpoint's memory.

Flow Processing Unit (FlowPU) contains the computational pipeline that is configured to perform the user-defined update and control operations. The result of the update is also checked in this block and if a control condition is violated, the result context is exported for further processing to the host PC.

Application Decoder performs various analysis of the packet's payload [29].

The purpose of the Flow Processing Unit is to update flow context fields according to the user definition of the monitoring process. The update operations are carried out using a tailor-made computational engine generated by a core generator program. Expired flow contexts are sent to the PC in the form of flow records for further processing.

5.2 Flow Processing Unit

The Flow Processing Unit architecture is outlined in Figure 5.3. The interface of this unit consists of the following ports: set of ports for the input (**IN_ENDPOINT_INTERFACE**) and output (**OUT_ENDPOINT_INTERFACE**) interface to the Endpoint (the detailed description of this interface is beyond the scope of this thesis). Ports **TX_APP_DEC** and **RX_APP_DEC** constitute the interface to the Application Decoder. Port **DO** is used for export of flow records to the host PC and port **MI** is a memory interface that enables setting unit's parameters and reading the debug register from the host PC. The FrameLink protocol used for marked ports is a high-speed frame-oriented protocol that has been developed as a part of the Liberouter [30] project.

The core of the unit is ALU, which is a computational pipeline generated using the process described in 4.3. The inputs of the unit are context of the flow and packet header, the output consists of updated context and control signals. The Address Counter generates proper address for each stage of the pipeline and also for the Masking Unit, which is used to update the Reg Valid register by the correct value. This register maintains the state of the

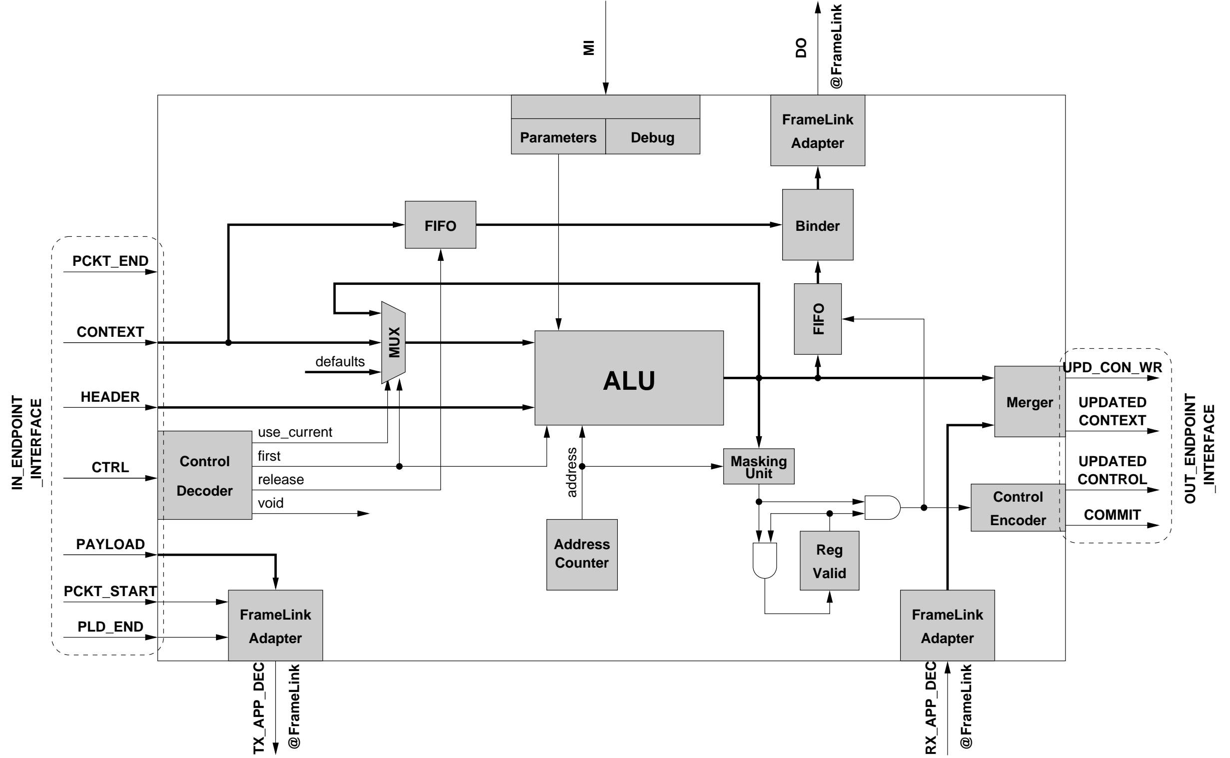


Figure 5.3: Flow Processing Unit architecture

currently processed frame — whether it has violated any of the control operations or not. In the former case, the output frame is released via the FIFO and Binder to the **DO** port for further processing in the host PC. In the later one, the frame is dropped from the FIFO and is stored back to the Endpoint after being merged with the data from the Application Decoder. The Control Decoder and Control Encoder encode and decode, respectively, the metadata that maintain the state of the flow context.

The feedback route from the output of ALU to the input multiplexer MUX enables processing of two consecutive packets belonging to the same context. If this route were not present, the input would need to be stopped (because the atomicity of the context update must be ensured) and throughput of the unit would drop significantly for large flows. The feedback path is activated by a special command on the **CTRL** port. Other control commands are: release the context (in this case, the context is sent directly to FIFO and the ALU is not used), no operation, release the context and create a new one (this deals with the problem of hash collision detected in the Fetch Block), and others.

5.3 Properties

The properties of the application of the platform in the flexible network flow monitoring probe has been studied and the results are following: the flexibility of the solution is satisfactory for NetFlow and IPFIX [31]. Moreover, the solution provides extended possibilities. An example may be measuring of interpacket gaps of a flow and providing the mean value and standard deviation of the parameter for the detection of application protocol.

The throughput of the Flow Processing Unit depends on two factors:

1. **Complexity of desired operations** — this factor directly affects the number of pipeline stages. More precisely, it is the complexity of the most complex operation, that determines the pipeline length L_P.
2. **Size of the context and header** — because of the demand for atomicity of context updates, there can be a maximum of a single "full" frame with the size S_F bits, or parts of two frames with the same size as a single frame, in the unit.

Considering both aforementioned factors, the resulting throughput T of the unit can be expressed by the following equation:

$$T = \frac{S_F \text{ bits}}{L_P \text{ clock cycles}} \tag{5.1}$$

The unit is designed to be used in the Xilinx Virtex-II Pro and Xilinx Virtex-5 FPGAs and work in in a 100 MHz clock domain, the resulting throughput T_R in Gb/s is therefore

$$T_R = 10^8 \cdot \frac{S_F}{L_P} \text{ Gb/s} \tag{5.2}$$

Table 5.1 shows the resulting throughput for the most common context sizes and pipeline lengths.

Context size [bytes]	Pipeline length [stages]	Throughput [Gb/s]
64	2	25.6 Gb/s
64	4	12.8 Gb/s
128	2	51.2 Gb/s
128	4	25.6 Gb/s

Table 5.1: The throughput of the processing unit for various parameters

Analysis of the firmware generation process showed that the most time-critical part is the generation of the pipelined data-flow graph (section 4.3.5). This part uses global search engine which in each step needs to generate complete description of the processing pipeline and compute the number of resources the pipeline consumes. This is done by a complex algorithm; nevertheless, its complexity has not been properly studied yet. However, analysis showed that effective implementation is crucial for the performance of the algorithm. We also believe that devising a heuristic used for the generation of initial population of the genetic algorithm used for global search instead of random generation could significantly increase the convergence speed of the algorithm.

We are also interested in the results that would be yielded by exchanging line 3 in function `pipeline_wire()` (see section 4.3.5):

```
| reg.Stage ← IN_WIRE.From.Stage + 1;
```

by the following line:

```
| reg.Stage ← max(IN_WIRE.From.Stage, cycle(IN_WIRE.From.TimeReady) - STAGES);
```

where `STAGES` is the number of stages of the pipeline. In this case, the data would not be implicitly propagated through the pipeline (nonetheless, it would still be ensured that the operations are in the permitted window) and thus more complex operations could be scheduled into less stages.

Chapter 6

Conclusion

The task of this work was the design and implementation of a firmware generation platform focused on the use in flexible network monitoring. Only the design and part of the implementation has been done so far, the reason being the complexity and multi-domain nature of the task. Several architectures has been proposed, however, their evaluation showed that the desired throughput (10 Gb/s) is obtainable only with the architecture which is described in this thesis.

To complete the task, it was necessary to study literature about hardware description languages [1, 2, 3, 4], high-level language parsing [6, 5, 19] and code generation patterns [8]. The focus of the platform also demanded to study details of the OSI reference model [11, 12, 13, 14, 15] and the principles of network flow monitoring [16, 17]. The obtained information are summarized in the theoretical background (chapter 2) of this thesis. Furthermore, current high-level synthesis methods [20, 21] were also studied; their description can be found in chapter 3.

The proposed framework is focused on providing higher-semantic level for the definition of flexible network monitoring in multi-gigabit networks. Design of the firmware generation process considers this — the process therefore generates optimized high-speed fine-grained computational pipeline for aggregation of flow information into flow records. The firmware generation and network probe aggregation unit are described in chapters 4 and 5 respectively. Chapter 5 also evaluates the properties of the proposed platform.

The future work will be focused on an effective implementation of the framework and further evaluation of the properties of the firmware generation algorithms and their refinement. A challenging task would be to conduct formal verification of the algorithms to prove that they yield correct results for all inputs.

Bibliography

[1] Peter J. Ashenden. *VHDL Tutorial*, 2004.
Available at URL http://www.tutground.net/Files/VHDL_TUTORIAL.pdf (April 2008).

[2] Open Verilog International. *Verilog-A Language Reference Manual*, 1996.
Available at URL http://www.eda.org/verilog-ams/htmlpages/public-docs/lrm/VerilogA/verilog-a-lrm-1-0.pdf (April 2008).

[3] Matthew Bowen. *Handel-C Language Reference Manual*. Embedded Solutions.
Available at URL http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%20C.PDF (April 2008).

[4] IEEE Computer Society. *IEEE Std 1800-2005: IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*, 2008.

[5] Alexander Meduna. *Elements of Compiler Design*. Taylor & Francis Informa plc, 2008.

[6] Alexander Meduna. *Automata and Languages: Theory and Applications*. Springer Verlag, 2005.

[7] Bison — GNU parser generator. URL http://www.gnu.org/software/bison/ (April 2008).

[8] Markus Völter. A Catalog of Patterns for Program Generation. In *EuroPLoP 2003*, 2003. Available at
URL http://www.voelter.de/data/pub/ProgramGeneration.pdf (April 2008).

[9] libxml2: Web pages of the libxml2 library. URL http://xmlsoft.org (April 2008).

[10] Lukáš Sekanina. *Evolvable Components — From Theory to Hardware Implementations*. Natural Computing Series. Springer Verlag, 2003.

[11] ISO/IEC 7498-1. *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*, June 1996.
Available at URL http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip (April 2008).

[12] IEEE Computer Society. *IEEE Std 802.3-2005: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method And Physical Layer Specification*, 2005. Available at URL http://standards.ieee.org/getieee802/ (April 2008).

[13] The Internet Engineering Task Force. *RFC 791: Internet Protocol*, September 1981. Available at URL http://www.ietf.org/rfc/rfc791.txt (April 2008).

[14] The Internet Engineering Task Force. *RFC 793: Transmission Control Prototol*, September 1981. Available at URL http://www.ietf.org/rfc/rfc793.txt (April 2008).

[15] The Internet Engineering Task Force. *RFC 768: User Datagram Prototol*, August 1980. Available at URL http://www.ietf.org/rfc/rfc768.txt (April 2008).

[16] J. Quittek, T. Zseby, B. Claise, and S. Zander. *RFC 3917: Requirements for IP Flow Information Export (IPFIX)*. The Internet Engineering Task Force, October 2004. Available at URL http://www.ietf.org/rfc/rfc3917.txt (April 2008).

[17] The Internet Engineering Task Force. *RFC 5101: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*, January 2008.
Available at URL http://www.ietf.org/rfc/rfc5101.txt (April 2008).

[18] V. Paxson, K. Asanović, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *First USENIX Workshop on Hot Topics in Security (HotSec '06)*, pages 63–68, July 2006.
Available at URL http://www.icir.org/vern/papers/hotsec06.pdf (April 2008).

[19] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[20] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin. *High-Level Synthesis — Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[21] Moonwook Oh and Soonhoi Ha. Synthesizable VHDL Code Generation from Data Flow Graph. In *5th Asia Pacific Conference on Hardware Description Languages*, 1998. Available at URL http://citeseer.ist.psu.edu/211167.html (April 2008).

[22] Steven J. Beaty. Genetic Algorithms and Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1991.
Available at URL http://citeseer.ist.psu.edu/beaty91genetic.html (April 2008).

[23] ISO/IEC 9899:TC3. *WG14/N1256*, September 2007. Available at URL http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf (April 2008).

[24] Flex. URL http://www.gnu.org/software/flex/ (April 2008).

[25] E. Bonsma and S. Gerez. A Genetic Approach to the Overlapped Scheduling of Iterative Data-Flow Graphs for Target Architectures with Communication Delays. In *ProRISC Workshop on Circuits, Systems and Signal Processing*, November 1997.

[26] Martin Žádník, Jan Kořenek, Petr Kobierský, and Ondřej Lengál. Network Probe for Flexible Flow Monitoring. In *2008 IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 213–218. IEEE Computer Society, April 2008.

[27] Tomáš Dedek, Tomáš Marek, and Tomáš Martínek. High Level Abstraction Language as an Alternative to Embeded Processors for Internet Packet Processing in FPGA. In *2007 International Conference on Field Programmable Logic and Applications*, pages 648–651. IEEE Computer Society, 2007.

[28] Martin Košek and Jan Kořenek. FlowContext: Flexible Platform for Multigigabit Stateful Packet Processing. In *2007 International Conference on Field Programmable Logic and Applications*, pages 804–807. IEEE Computer Society, 2007.

[29] Petr Kobierský. Hardware acceleration of protocol identification. Master's thesis, FIT Brno University of Technology, 2008.

[30] Liberouter. URL http://www.liberouter.org (April 2008).

[31] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer. *RFC 5102: Information Model for IP Flow Information Export.* The Internet Engineering Task Force, January 2008. Available at URL http://www.ietf.org/rfc/rfc5102.txt (April 2008).

Appendix A

Storage Medium

A storage medium (CD) containing an electronic version of the technical report and source codes of the part of the framework that has been implemented is enclosed to this thesis.